

Simulink® Check™

User's Guide



MATLAB® & SIMULINK®

R2023a



How to Contact MathWorks



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



Phone: 508-647-7000



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

Simulink® Check™ User's Guide

© COPYRIGHT 2004–2023 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

September 2017	Online only	New for Version 4.0 (Release 2017b)
March 2018	Online only	Revised for Version 4.1 (Release 2018a)
September 2018	Online only	Revised for Version 4.2 (Release 2018b)
March 2019	Online only	Revised for Version 4.3 (Release 2019a)
September 2019	Online only	Revised for Version 4.4 (Release 2019b)
March 2020	Online only	Revised for Version 4.5 (Release 2020a)
September 2020	Online only	Revised for Version 5.0 (Release 2020b)
March 2021	Online only	Revised for Version 5.1 (Release 2021a)
September 2021	Online only	Revised for Version 5.2 (Release 2021b)
March 2022	Online only	Revised for Version 6.0 (Release 2022a)
September 2022	Online only	Revised for Version 6.1 (Release 2022b)
March 2023	Online only	Revised for Version 6.2 (Release 2023a)

Get Started

1

Simulink Check Product Description	1-2
Assess and Verify Model Quality	1-3
Detect and Fix Model Advisor Check Violations	1-4
Detect and Fix Model Advisor Check Violations While You Edit	1-5
Detect and Fix Model Advisor Check Violations Interactively	1-6
Collect Model Metric Data by Using the Metrics Dashboard	1-9
Analyze Metric Data	1-9
Explore Metric Data	1-10
Refactor Model Based on Metric Data	1-11
Detect and Fix Compliance Issues	1-12
Explore Compliance Results in the Dashboard	1-12
Update Model to Fix Compliance Issues	1-13
Rerun Model Metrics	1-13
Refactor Models to Improve Component Reuse	1-15
Identify and Replace Clones with Links to Library Blocks	1-15
Explore Other Options	1-17
Simplify Model for Targeted Analysis of Complex Models Using Model Slicer Tool	1-19
Assess Requirements-Based Testing Quality by Using the Model Testing Dashboard	1-22
Open the Project and Model Testing Dashboard	1-22
Assess Traceability of Artifacts	1-23
Explore Metric Results for a Unit	1-24
Track Testing Status of a Project Using the Model Testing Dashboard ...	1-25

Verification and Validation

2

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

Analyze Models for Standards Compliance and Design Errors	2-7
Standards and Analysis Overview	2-7
Check Model for Style Guideline Violations and Design Errors	2-7
Perform Functional Testing and Analyze Test Coverage	2-9
Incrementally Increase Test Coverage Using Test Case Generation	2-9
Analyze Code and Test Software-in-the-Loop	2-12
Code Analysis and Testing Software-in-the-Loop Overview	2-12
Analyze Code for Defects, Metrics, and MISRA C:2012	2-12
Test Code Against Model Using Software-in-the-Loop Testing	2-17

Checking Systems Interactively

3

Check Model Compliance by Using the Model Advisor	3-2
Model Advisor Overview	3-2
Run Model Advisor Checks and Review Results	3-4
Check Model Compliance Using Edit-Time Checking	3-6
Configure Your Model to Use Edit-Time Checking	3-6
View and Customize the Edit-Time Checks in a Model Advisor Configuration	3-8
Exclude Blocks from the Model Advisor Check Analysis	3-9
Model Advisor Exclusion Overview	3-9
Create Model Advisor Exclusions	3-11
Save Model Advisor Exclusions in a Model File	3-11
Save Model Advisor Exclusions in Exclusion File	3-12
Check Selector	3-12
Review Model Advisor Exclusions	3-13
Manage Exclusions	3-14
Compatibility Considerations after R2020b	3-14
Programmatically Change Model Advisor Exclusions	3-15
Generate Model Advisor Reports	3-16
Generate Results Report After Executing Model Advisor Checks	3-16
Modify Template for Model Advisor Check Results Report	3-16
Transform Model to Variant System	3-19
Transform Model to Variant System Using Model Transformer	3-19
Model Transformation Limitations	3-21
Improve Code Efficiency by Merging Multiple Interpolation Using Prelookup Blocks	3-23
Merge Interpolation Using Prelookup Blocks Using the Model Transformer App	3-23
Merge Interpolation Using Prelookup Blocks Programmatically	3-26
Conditions and Limitations	3-28
Enable Component Reuse by Using Clone Detection	3-29
Exact Clones and Similar Clones	3-29

Specify Where to Detect Clones	3-29
Identify Exact and Similar Clones	3-29
Replace Clones	3-33
Identify and Replace Clones in Model Libraries	3-34
Check the Equivalency of the Model	3-34
Improve Model Readability by Eliminating Local Data Store Blocks . . .	3-36
Improve Model Readability by Eliminating Local Data Store Blocks Using Model Transformer	3-36
Limitations	3-38
Improve Efficiency of Simulation by Optimizing Prelookup Operation of Lookup Table Blocks	3-40
Improve Efficiency of Simulation by Optimizing Prelookup Operation of Lookup Table Blocks Using Model Transformer	3-40
Conditions and Limitations	3-43
Model Advisor Checks for DO-178C/DO-331 Standards Compliance . . .	3-45
Model Advisor Checks for High Integrity Systems Modeling Guidelines .	3-46
Model Advisor Checks for DO-254 Standard Compliance	3-53
Model Checks for High Integrity Systems Modeling	3-53
HDL Code Advisor Checks	3-55
Model Advisor Checks for High Integrity Systems Modeling Guidelines	3-58
Model Advisor Checks for High-Integrity Systems Modeling Guidelines .	3-46
Model Advisor Checks for IEC 61508, IEC 62304, ISO 26262, ISO 25119, and EN 50128/EN 50657 Standards Compliance	3-65
Model Advisor Checks for High Integrity Systems Modeling Guidelines .	3-66
Model Advisor Checks for MISRA C:2012 Coding Standards	3-74
Model Advisor Checks for CERT C, SWE, and ISO/IEC TS 17961 Secure Coding Standards	3-75
Model Advisor Checks for Requirements Links	3-78
Replace Exact Clones with Subsystem Reference	3-79
Identify Exact Clones	3-79
Replace Clones	3-81
Check the Equivalency of the Model	3-81
Detect and Replace Subsystem Clones Programmatically	3-83
Identify Clones in a Model	3-83
Replace Clones in a Model	3-85
Identify Clones Using Subsystem Reference Blocks	3-85
Replace Clones with Conditions	3-87
Check the Equivalency of the Model	3-88
Find Clones Across the Model	3-89
Identify Clones Across the Model by Using the Clone Detector App	3-89
Identify Clones Programmatically	3-91

Detect Clones Programmatically on Multiple Models Across Different Folders	3-93
Detect and Replace Clones Programmatically in a Loop on Multiple Models	3-95
Running Clone Detection Custom Script in a Project	3-97
Run Custom Model Advisor Checks on Architecture Models	3-99
Justify Model Advisor Violations from Check Analysis	3-102

Check Systems Programmatically

4

Checking Systems Programmatically	4-2
Create a Function to Check Multiple Systems	4-3
Archive and View Results	4-6
Archive Results	4-6
View Results in Command Window	4-6
View Results in Model Advisor Command-Line Summary Report	4-6
View Results in Model Advisor GUI	4-7
View Model Advisor Report	4-7
Archive and View Model Advisor Run Results	4-9
Run Tasks Locally and in CI	4-10
Installation	4-10
MBD Pipeline	4-10
Build System	4-11
Process Advisor App	4-12
CI/CD System Integration	4-13

Model Metrics

5

Collect and Explore Metric Data by Using the Metrics Dashboard	5-2
Metrics Dashboard Widgets	5-3
Size	5-4
Modeling Guideline Compliance	5-5
Architecture	5-6
Metric Thresholds	5-7
Dashboard Limitations	5-7
Collect Model Metrics Using the Model Advisor	5-9
Create a Custom Model Metric for Nonvirtual Block Count	5-11

Collect Model Metrics Programmatically	5-15
Model Metric Data Aggregation	5-19
How Model Metric Aggregation Works	5-19
Access Aggregated Metric Data	5-20
Identify Modeling Clones with the Metrics Dashboard	5-23
Collect Compliance Data and Explore Results in the Model Advisor ...	5-25
Collect Metric Data Programmatically and View Data Through the Metrics Dashboard	5-28
Fix Metric Threshold Violations in a Continuous Integration Systems Workflow	5-31
Project Setup	5-31
GitLab Setup	5-33
Jenkins Setup	5-33
Continuous Integration Workflow	5-34
Customize Metrics Dashboard Layout and Functionality	5-37
Rearrange and Remove Widgets in Metrics Dashboard	5-48
Create Layout with Custom Metric	5-55
Modify, Remove, and Add Metric Thresholds in Metrics Dashboard ...	5-62
Change Model Advisor Checks in Metrics Dashboard	5-70
Compare Model Complexity and Code Complexity Metrics	5-76
Metric Threshold Values	5-76
Comparing Code and Model Complexity Metric Results	5-76
Explore Status and Quality of Testing Activities Using Model Testing Dashboard	5-80
Fix Requirements-Based Testing Issues	5-89
Manage Project Artifacts for Analysis in Dashboard	5-95
Manage Artifact Files in a Project	5-96
Trace Artifacts to Units and Components	5-96
Collect Metric Results	5-101
Assess Requirements-Based Testing for ISO 26262	5-102
Open the Model Testing Dashboard and Collect Metric Results	5-102
Test Review	5-103
Test Results Review	5-107
Unit Verification in Accordance with ISO 26262	5-112
Collect Metrics on Model Testing Artifacts Programmatically	5-115
Open the Project	5-115
Collect Metric Results	5-115
Access Results	5-116

Categorize Models in a Hierarchy as Components or Units	5-119
Units in the Dashboard	5-119
Components in the Dashboard	5-119
Specify Models as Components and Units	5-120
Monitor Low-Level Test Results in the Model Testing Dashboard	5-123
Resolve Missing Artifacts, Links, and Results	5-128
Issue	5-128
Possible Solutions	5-128
Collect Requirements-Based Testing Metrics Using Continuous Integration	5-138
Hide Requirements Metrics in the Model Testing Dashboard and in API Results	5-141
Open the Dashboard for the Project	5-141
Hide Requirements Metrics	5-141
View API Results with Requirements Metrics Hidden	5-142
Monitor the Complexity of Your Design Using the Model Maintainability Dashboard	5-144
Collect Model Maintainability Metrics Programmatically	5-150
Resolve Missing Artifacts and Results in the Model Maintainability Dashboard	5-154
Issue	5-154
Possible Solutions	5-154
Assess Model Size and Complexity for ISO 26262	5-158
Open Model Maintainability Dashboard	5-158
Review Maintainability Metrics for Design Artifacts	5-158
Analyze Your Project With Dashboards	5-163
Open a New Dashboard	5-163
View Help for Metric Information	5-163
Navigate the Dashboard Window	5-164
Digital Thread	5-167
Manage Design Artifacts for Analysis in the Model Maintainability Dashboard	5-168
Manage Artifact Files in a Project	5-169
Trace Artifacts to Units and Components	5-169
Collect Metric Results	5-173
Explore Metric Results, Monitor Progress, and Identify Issues	5-175
Metric Results in the Dashboards	5-175
Use the Model Maintainability Dashboard to Collect Size, Architecture, and Complexity Metrics for a Model	5-182
View Status of Code Testing Activities for Software Units in Project ..	5-184
View Code Testing Status	5-184

Review Test Status and Coverage Results	5-186
Assess Test Results	5-188
Analyze Coverage Results	5-189
Identify and Troubleshoot Gaps in Code Testing Results and Coverage	5-192
Collect Code Testing Metrics Programmatically	5-201
Run Model and Code Tests	5-201
Collect Metric Results For Software Units in Project	5-201
Access Results	5-202
Explore Traceability Information for Units and Components	5-205
Access Trace Views	5-205
View Different Trace Views	5-207
Troubleshoot Missing Artifacts and Relationships	5-211
View Artifact Issues in Project	5-212
Open Artifact Issues Tab	5-212
View Details About Artifact Issues	5-212
Get Artifact Issues Programmatically	5-213
Fix Artifact Issues	5-213
PIL Coverage Recap	5-215
SIL Coverage Recap	5-216
PIL Coverage Breakdown	5-217
Metric ID	5-217
Description	5-217
Computation Details	5-217
Collection	5-217
Results	5-217
Compliance Thresholds	5-218
PIL Coverage Fragment	5-219
Metric ID	5-219
Description	5-219
Computation Details	5-219
Collection	5-219
Results	5-219
Compliance Thresholds	5-220
PIL and Model Test Statuses	5-221
Metric ID	5-221
Description	5-221
Computation Details	5-221
Collection	5-221
Results	5-221
Compliance Thresholds	5-221
PIL and Model Test Status Distributions	5-222
Metric ID	5-222
Description	5-222
Computation Details	5-222

Collection	5-222
Results	5-222
Compliance Thresholds	5-222
PIL Test Status	5-224
Metric ID	5-224
Description	5-224
Computation Details	5-224
Collection	5-224
Results	5-224
Compliance Thresholds	5-224
PIL Test Status Distribution	5-226
Metric ID	5-226
Description	5-226
Computation Details	5-226
Collection	5-226
Results	5-226
Compliance Thresholds	5-227
SIL Coverage Breakdown	5-228
Metric ID	5-228
Description	5-228
Computation Details	5-228
Collection	5-228
Results	5-228
Compliance Thresholds	5-229
SIL Coverage Fragment	5-230
Metric ID	5-230
Description	5-230
Computation Details	5-230
Collection	5-230
Results	5-230
Compliance Thresholds	5-231
SIL and Model Test Statuses	5-232
Metric ID	5-232
Description	5-232
Computation Details	5-232
Collection	5-232
Results	5-232
Compliance Thresholds	5-232
SIL and Model Test Status Distributions	5-233
Metric ID	5-233
Description	5-233
Computation Details	5-233
Collection	5-233
Results	5-233
Compliance Thresholds	5-233
SIL Test Status	5-235
Metric ID	5-235
Description	5-235

Computation Details	5-235
Collection	5-235
Results	5-235
Compliance Thresholds	5-235
SIL Test Status Distribution	5-237
Metric ID	5-237
Description	5-237
Computation Details	5-237
Collection	5-237
Results	5-237
Compliance Thresholds	5-238
Requirement with Test Case	5-239
Metric ID	5-239
Description	5-239
Computation Details	5-239
Collection	5-239
Results	5-239
Requirement with Test Case Distribution	5-240
Metric ID	5-240
Description	5-240
Computation Details	5-240
Collection	5-240
Results	5-240
Compliance Thresholds	5-241
Test Cases per Requirement	5-242
Metric ID	5-242
Description	5-242
Computation Details	5-242
Collection	5-242
Results	5-242
Test Cases per Requirement Distribution	5-243
Metric ID	5-243
Description	5-243
Computation Details	5-243
Collection	5-243
Results	5-243
Compliance Thresholds	5-244
Test Case with Requirement	5-245
Metric ID	5-245
Description	5-245
Computation Details	5-245
Collection	5-245
Results	5-245
Test Case with Requirement Distribution	5-247
Metric ID	5-247
Description	5-247
Computation Details	5-247
Collection	5-247

Results	5-247
Compliance Thresholds	5-248
Requirements per Test Case	5-249
Metric ID	5-249
Description	5-249
Computation Details	5-249
Collection	5-249
Results	5-249
Requirements per Test Case Distribution	5-251
Metric ID	5-251
Description	5-251
Computation Details	5-251
Collection	5-251
Results	5-251
Compliance Thresholds	5-252
Test Case Type	5-253
Metric ID	5-253
Description	5-253
Computation Details	5-253
Collection	5-253
Results	5-253
Test Case Type Distribution	5-254
Metric ID	5-254
Description	5-254
Computation Details	5-254
Collection	5-254
Results	5-254
Compliance Thresholds	5-255
Test Case Tag	5-256
Metric ID	5-256
Description	5-256
Computation Details	5-256
Collection	5-256
Results	5-256
Test Case Tag Distribution	5-257
Metric ID	5-257
Description	5-257
Computation Details	5-257
Collection	5-257
Results	5-257
Compliance Thresholds	5-257
Model Coverage Breakdown	5-258
Metric ID	5-258
Description	5-258
Computation Details	5-258
Collection	5-258
Results	5-258
Compliance Thresholds	5-259

Model Coverage Fragment	5-260
Metric ID	5-260
Description	5-260
Computation Details	5-260
Collection	5-260
Results	5-260
Compliance Thresholds	5-261
Model Test Status	5-262
Metric ID	5-262
Description	5-262
Computation Details	5-262
Collection	5-262
Results	5-262
Compliance Thresholds	5-262
Model Test Status Distribution	5-264
Metric ID	5-264
Description	5-264
Computation Details	5-264
Collection	5-264
Results	5-264
Compliance Thresholds	5-265
Test Case Verification Status	5-266
Metric ID	5-266
Description	5-266
Computation Details	5-266
Collection	5-266
Results	5-266
Test Case Verification Status Distribution	5-268
Metric ID	5-268
Description	5-268
Computation Details	5-268
Collection	5-268
Results	5-268
Compliance Thresholds	5-269
Requirements Execution Coverage Breakdown	5-270
Metric ID	5-270
Description	5-270
Computation Details	5-270
Collection	5-270
Results	5-270
Compliance Thresholds	5-270
Requirements Decision Coverage Breakdown	5-272
Metric ID	5-272
Description	5-272
Computation Details	5-272
Collection	5-272
Results	5-272
Compliance Thresholds	5-272

Requirements Condition Coverage Breakdown	5-274
Metric ID	5-274
Description	5-274
Computation Details	5-274
Collection	5-274
Results	5-274
Compliance Thresholds	5-274
Requirements MCDC Coverage Breakdown	5-276
Metric ID	5-276
Description	5-276
Computation Details	5-276
Collection	5-276
Results	5-276
Compliance Thresholds	5-276
Requirements Execution Coverage Fragment	5-278
Metric ID	5-278
Description	5-278
Computation Details	5-278
Collection	5-278
Results	5-278
Compliance Thresholds	5-278
Requirements Decision Coverage Fragment	5-279
Metric ID	5-279
Description	5-279
Computation Details	5-279
Collection	5-279
Results	5-279
Compliance Thresholds	5-279
Requirements Condition Coverage Fragment	5-280
Metric ID	5-280
Description	5-280
Computation Details	5-280
Collection	5-280
Results	5-280
Compliance Thresholds	5-280
Requirements MCDC Coverage Fragment	5-281
Metric ID	5-281
Description	5-281
Computation Details	5-281
Collection	5-281
Results	5-281
Compliance Thresholds	5-281
Unit Boundary Execution Coverage Breakdown	5-282
Metric ID	5-282
Description	5-282
Computation Details	5-282
Collection	5-282
Results	5-282
Compliance Thresholds	5-282

Unit Boundary Decision Coverage Breakdown	5-284
Metric ID	5-284
Description	5-284
Computation Details	5-284
Collection	5-284
Results	5-284
Compliance Thresholds	5-284
Unit Boundary Condition Coverage Breakdown	5-286
Metric ID	5-286
Description	5-286
Computation Details	5-286
Collection	5-286
Results	5-286
Compliance Thresholds	5-286
Unit Boundary MCDC Coverage Breakdown	5-288
Metric ID	5-288
Description	5-288
Computation Details	5-288
Collection	5-288
Results	5-288
Compliance Thresholds	5-288
Unit Boundary Execution Coverage Fragment	5-290
Metric ID	5-290
Description	5-290
Computation Details	5-290
Collection	5-290
Results	5-290
Compliance Thresholds	5-290
Unit Boundary Decision Coverage Fragment	5-291
Metric ID	5-291
Description	5-291
Computation Details	5-291
Collection	5-291
Results	5-291
Compliance Thresholds	5-291
Unit Boundary Condition Coverage Fragment	5-292
Metric ID	5-292
Description	5-292
Computation Details	5-292
Collection	5-292
Results	5-292
Compliance Thresholds	5-292
Unit Boundary MCDC Coverage Fragment	5-293
Metric ID	5-293
Description	5-293
Computation Details	5-293
Collection	5-293
Results	5-293
Compliance Thresholds	5-293

Overall Design Cyclomatic Complexity	5-294
Metric ID	5-294
Description	5-294
Computation Details	5-294
Collection	5-295
Results	5-295
Examples	5-295
Layer Depth	5-296
Metric ID	5-296
Description	5-296
Computation Details	5-296
Collection	5-296
Results	5-296
Examples	5-296
Maximum Layer Depth	5-298
Metric ID	5-298
Description	5-298
Computation Details	5-298
Collection	5-298
Results	5-298
Examples	5-298
Layer Breadth	5-300
Metric ID	5-300
Description	5-300
Computation Details	5-300
Collection	5-300
Results	5-300
Examples	5-301
Maximum Layer Breadth	5-302
Metric ID	5-302
Description	5-302
Computation Details	5-302
Collection	5-302
Results	5-302
Examples	5-303
Input and Output Component Interface Ports	5-304
Metric ID	5-304
Description	5-304
Computation Details	5-304
Collection	5-304
Results	5-304
Examples	5-304
Input and Output Component Interface Signals	5-306
Metric ID	5-306
Description	5-306
Computation Details	5-306
Collection	5-306
Results	5-306
Examples	5-307

Simulink Design Cyclomatic Complexity	5-308
Metric ID	5-308
Description	5-308
Computation Details	5-308
Collection	5-308
Results	5-308
Examples	5-308
Simulink Decision Count	5-310
Metric ID	5-310
Description	5-310
Decision Counts for Common Block Types	5-310
Computation Details	5-320
Collection	5-320
Results	5-320
Examples	5-320
Simulink Decision Distribution	5-322
Metric ID	5-322
Description	5-322
Collection	5-322
Results	5-322
Examples	5-322
Stateflow Design Cyclomatic Complexity	5-324
Metric ID	5-324
Description	5-324
Computation Details	5-324
Collection	5-324
Results	5-324
Examples	5-325
Stateflow Decision Count	5-326
Metric ID	5-326
Description	5-326
Decision Counts for Common Stateflow Components	5-326
Collection	5-326
Results	5-326
Examples	5-327
Stateflow Decision Distribution	5-328
Metric ID	5-328
Description	5-328
Collection	5-328
Results	5-328
MATLAB Design Cyclomatic Complexity	5-329
Metric ID	5-329
Description	5-329
Computation Details	5-329
Collection	5-329
Results	5-329
Examples	5-329

MATLAB Decision Count	5-331
Metric ID	5-331
Description	5-331
Decision Counts for Common MATLAB Keywords	5-331
Computation Details	5-332
Collection	5-332
Results	5-332
Examples	5-332
MATLAB Decision Distribution	5-334
Metric ID	5-334
Description	5-334
Collection	5-334
Results	5-334
Overall Blocks	5-335
Metric ID	5-335
Description	5-335
Collection	5-335
Results	5-335
Simulink Blocks	5-336
Metric ID	5-336
Description	5-336
Computation Details	5-336
Collection	5-336
Results	5-336
Simulink Blocks Distribution	5-337
Metric ID	5-337
Description	5-337
Computation Details	5-337
Collection	5-337
Results	5-337
Overall Signal Lines	5-339
Metric ID	5-339
Description	5-339
Collection	5-339
Results	5-339
Simulink Signal Lines	5-340
Metric ID	5-340
Description	5-340
Collection	5-340
Results	5-340
Simulink Signal Lines Distribution	5-341
Metric ID	5-341
Description	5-341
Collection	5-341
Results	5-341
Overall Goto Blocks	5-342
Metric ID	5-342

Description	5-342
Collection	5-342
Results	5-342
Simulink Goto Blocks	5-343
Metric ID	5-343
Description	5-343
Collection	5-343
Results	5-343
Simulink Goto Blocks Distribution	5-344
Metric ID	5-344
Description	5-344
Collection	5-344
Results	5-344
Overall Transitions	5-345
Metric ID	5-345
Description	5-345
Collection	5-345
Results	5-345
Stateflow Transitions	5-346
Metric ID	5-346
Description	5-346
Collection	5-346
Results	5-346
Stateflow Transitions Distribution	5-347
Metric ID	5-347
Description	5-347
Collection	5-347
Results	5-347
Overall States	5-348
Metric ID	5-348
Description	5-348
Collection	5-348
Results	5-348
Stateflow States	5-349
Metric ID	5-349
Description	5-349
Collection	5-349
Results	5-349
Stateflow States Distribution	5-350
Metric ID	5-350
Description	5-350
Collection	5-350
Results	5-350
Overall MATLAB Executable Lines of Code (eLOC)	5-351
Metric ID	5-351
Description	5-351

Collection	5-351
Results	5-351
MATLAB Effective Lines of Code (eLOC)	5-352
Metric ID	5-352
Description	5-352
Collection	5-352
Results	5-352
MATLAB Effective Lines of Code (eLOC) Distribution	5-353
Metric ID	5-353
Description	5-353
Collection	5-353
Results	5-353
Simulink Block Metric	5-355
Metric Information	5-355
Description	5-355
Computation Details	5-355
Collection	5-355
Results	5-355
Subsystem Metric	5-357
Metric Information	5-357
Description	5-357
Computation Details	5-357
Collection	5-357
Results	5-357
Library Link Metric	5-358
Metric Information	5-358
Description	5-358
Computation Details	5-358
Collection	5-358
Results	5-358
Effective Lines of MATLAB Code Metric	5-359
Metric Information	5-359
Description	5-359
Computation Details	5-359
Collection	5-359
Results	5-359
Stateflow Chart Objects Metric	5-361
Metric Information	5-361
Description	5-361
Computation Details	5-361
Collection	5-361
Results	5-362
Lines of Code for Stateflow Blocks Metric	5-363
Metric Information	5-363
Description	5-363
Computation Details	5-363
Collection	5-363

Results	5-363
Subsystem Depth Metric	5-365
Metric Information	5-365
Description	5-365
Computation Details	5-365
Collection	5-365
Results	5-365
Input Output Metric	5-366
Metric Information	5-366
Description	5-366
Computation Details	5-366
Results	5-366
Explicit Input Output Metric	5-367
Metric Information	5-367
Description	5-367
Computation Details	5-367
Results	5-367
File Metric	5-368
Metric Information	5-368
Description	5-368
Computation Details	5-368
Results	5-368
MATLAB Function Metric	5-369
Metric Information	5-369
Description	5-369
Computation Details	5-369
Results	5-369
Model File Count	5-370
Metric Information	5-370
Description	5-370
Computation Details	5-370
Results	5-370
Parameter Metric	5-371
Metric Information	5-371
Description	5-371
Computation Details	5-371
Results	5-371
Stateflow Chart Metric	5-372
Metric Information	5-372
Description	5-372
Computation Details	5-372
Results	5-372
Cyclomatic Complexity Metric	5-373
Metric Information	5-373
Description	5-373
Computation Details	5-373

Collection	5-374
Results	5-374
Clone Content Metric	5-375
Metric Information	5-375
Description	5-375
Computation Details	5-375
Results	5-375
Clone Detection Metric	5-376
Metric Information	5-376
Description	5-376
Computation Details	5-376
Results	5-376
Library Content Metric	5-377
Metric Information	5-377
Description	5-377
Computation Details	5-377
Results	5-377
MATLAB Code Analyzer Warnings	5-378
Metric Information	5-378
Description	5-378
Computation Details	5-378
Results	5-378
Diagnostic Warnings Metric	5-379
Metric Information	5-379
Description	5-379
Computation Details	5-379
Results	5-379
Model Advisor Check Compliance for High-Integrity Systems	5-380
Metric Information	5-380
Description	5-380
Computation Details	5-380
Results	5-380
Results Details	5-380
Model Advisor Check Compliance for Modeling Standards for MAB ..	5-382
Metric Information	5-382
Description	5-382
Computation Details	5-382
Results	5-382
Results Details	5-382
Model Advisor Check Issues for High-Integrity Systems	5-384
Metric Information	5-384
Description	5-384
Computation Details	5-384
Results	5-384
Model Advisor Check Issues for MAB Standards	5-385
Metric Information	5-385

Description	5-385
Computation Details	5-385
Results	5-385
Nondescriptive Block Name Metric	5-386
Metric Information	5-386
Description	5-386
Computation Details	5-386
Collection	5-386
Results	5-386
Data and Structure Layer Separation Metric	5-388
Metric Information	5-388
Description	5-388
Computation Details	5-388
Collection	5-388
Results	5-388

Create Model Advisor Checks

6

Overview of the Customization File for Custom Checks	6-2
Common Utilities for Creating Checks	6-4
Review a Model Against Conditions that You Specify with the Model Advisor	6-5
Create an sl_customization Function	6-5
Create the Check Definition Function for a Pass/Fail Check with No Fix Action	6-5
Create the Check Definition Function for an Informational Check	6-6
Run the Custom Checks in the Model Advisor	6-7
Define Edit-Time Checks to Comply with Conditions That You Specify with the Model Advisor	6-9
Define Custom Edit-Time Checks that Fix Issues in Architecture Models	6-17
Create a Simple Architecture Model	6-17
Create the Custom Edit-Time Check	6-18
Create a Custom Edit-Time Check Configuration	6-19
Fix a Model to Comply with Conditions that You Specify with the Model Advisor	6-21
Create the sl_customization File	6-21
Create the Check Definition File	6-21
Run the Check	6-24
Create Model Advisor Check for Model Configuration Parameters	6-27
Create a Data File for a Configuration Parameter Check	6-27
Create Check for Diagnostics Pane Model Configuration Parameters	6-29
Data File for Configuration Parameter Check	6-30

Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters	6-38
Define Custom Model Advisor Checks	6-45
Create sl_customization Function	6-45
Register Custom Checks	6-45
Create Check Definition Function	6-46
Define the Compile Option for Custom Model Advisor Checks	6-50
Checks that Evaluate the Code Generation Readiness of the Model	6-50
Create Custom Check to Evaluate Active and Inactive Variant Paths from a Model	6-51
Exclude Blocks From Custom Checks	6-57
Create Help for Custom Model Advisor Checks	6-59
See Also	6-59

Model Advisor Customization

7

Customize the Configuration of the Model Advisor Overview	7-2
Use the Model Advisor Configuration Editor to Customize the Model Advisor	7-3
Overview of the Model Advisor Configuration Editor	7-3
Open the Model Advisor Configuration Editor	7-4
Specify a Default Configuration File	7-5
Customize the Model Advisor Configuration	7-5
Suppress Warning Message for Missing Checks	7-7
Upgrade Incompatible Checks in Model Advisor Configuration Files	7-7
Use the Model Advisor Configuration Editor to Create a Custom Model Advisor Configuration	7-8
Programmatically Customize Tasks and Folders for the Model Advisor	7-13
Customization File Overview	7-13
Register Tasks and Folders	7-13
Define Custom Tasks	7-14
Define Custom Folders	7-15
Programmatically Create Procedural-Based Configurations	7-17
Create Procedural-Based Configurations	7-17
Update the Environment to Include Your Custom Configuration	7-20
Load and Associate a Custom Configuration with a Model	7-21
Deploy Custom Configurations	7-23
Create and Deploy a Model Advisor Custom Configuration	7-24

Highlight Functional Dependencies	8-2
Highlight Dependencies for Multiple Instance Reference Models	8-8
Refine Highlighted Model	8-12
Define a Simulation Time Window	8-12
Exclude Blocks	8-17
Exclude Inputs of a Switch Block	8-20
Refine Dead Logic for Dependency Analysis	8-23
Analyze the Dead Logic	8-23
Create a Simplified Standalone Model	8-29
Highlight Active Time Intervals by Using Activity-Based Time Slicing .	8-30
Highlighting the Active Time Intervals of a Stateflow® State or Transition	8-30
Activity-Based Time Slicing Limitations and Considerations	8-36
Stateflow State and Transition Activity	8-36
Simplify a Standalone Model by Inlining Content	8-37
Workflow for Dependency Analysis	8-39
Dependency Analysis Workflow	8-39
Dependency Analysis Objectives	8-39
Configure Model Highlight and Sliced Models	8-41
Model Slicer	8-41
Model Slicer Options	8-41
Storage Options	8-41
Refresh Highlighting Automatically	8-42
Sliced Model Options	8-42
Trivial Subsystems	8-42
Inline Content Options	8-43
Model Slicer Considerations and Limitations	8-44
Model Compilation	8-44
Model Highlighting and Model Editing	8-44
Standalone Sliced Model Generation	8-44
Sliced Model Considerations	8-44
Port Attribute Considerations	8-45
Simulation Time Window Considerations	8-46
Simulation-based Sliced Model Simplifications	8-46
Model Slicer Support Limitations for Simulink Software Features	8-47
Model Slicer Support Limitations for Simulation Stepper	8-47
Model Slicer Support Limitations for Simulink Blocks	8-47
Model Slicer Support Limitations for Stateflow	8-48
Using Model Slicer with Stateflow	8-50
Model Slicer Highlighting Behavior for Stateflow Elements	8-50
Using Model Slicer with Stateflow State Transition Tables	8-50

Support Limitations for Using Model Slicer with Stateflow	8-50
Isolating Dependencies of an Actuator Subsystem	8-52
Choose Starting Points and Direction	8-52
View Precedents and Generate Model Slice	8-53
Isolate Model Components for Functional Testing	8-56
Isolate Subsystems for Functional Testing	8-56
Isolate Referenced Model for Functional Testing	8-58
Refine Highlighted Model by Using Existing .slicex or Dead Logic Results	8-63
Simplification of Variant Systems	8-65
Use the Variant Reducer to Simplify Variant Systems	8-65
Use Model Slicer to Simplify Variant Systems	8-65
Programmatically Resolve Unexpected Behavior in a Model with Model Slicer	8-67
Refine Highlighted Model Slice by Using Model Slicer Data Inspector .	8-75
Investigate Highlighted Model Slice by Using Model Slicer Data Inspector	8-75
Debug Slice Simulation by Using Fast Restart Mode	8-82
Simulate and Debug a Test Case in a Model Slice	8-82
Isolate Referenced Model for Functional Testing	8-88
Analyze the Dead Logic	8-92
Investigate Highlighted Model Slice by Using Model Slicer Data Inspector	8-97
Programmatically Generate I/O Dependency Matrix	8-103
Observe Impact of Simulink Parameters Using Model Slicer	8-105
Analyze Models Containing Simulink Functions Using Model Slicer ..	8-107

Get Started

- “Simulink Check Product Description” on page 1-2
- “Assess and Verify Model Quality” on page 1-3
- “Detect and Fix Model Advisor Check Violations” on page 1-4
- “Collect Model Metric Data by Using the Metrics Dashboard” on page 1-9
- “Detect and Fix Compliance Issues” on page 1-12
- “Refactor Models to Improve Component Reuse” on page 1-15
- “Simplify Model for Targeted Analysis of Complex Models Using Model Slicer Tool” on page 1-19
- “Assess Requirements-Based Testing Quality by Using the Model Testing Dashboard” on page 1-22

Simulink Check Product Description

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

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

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

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

Assess and Verify Model Quality

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

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

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

In this tutorial, you will learn to:

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

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

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

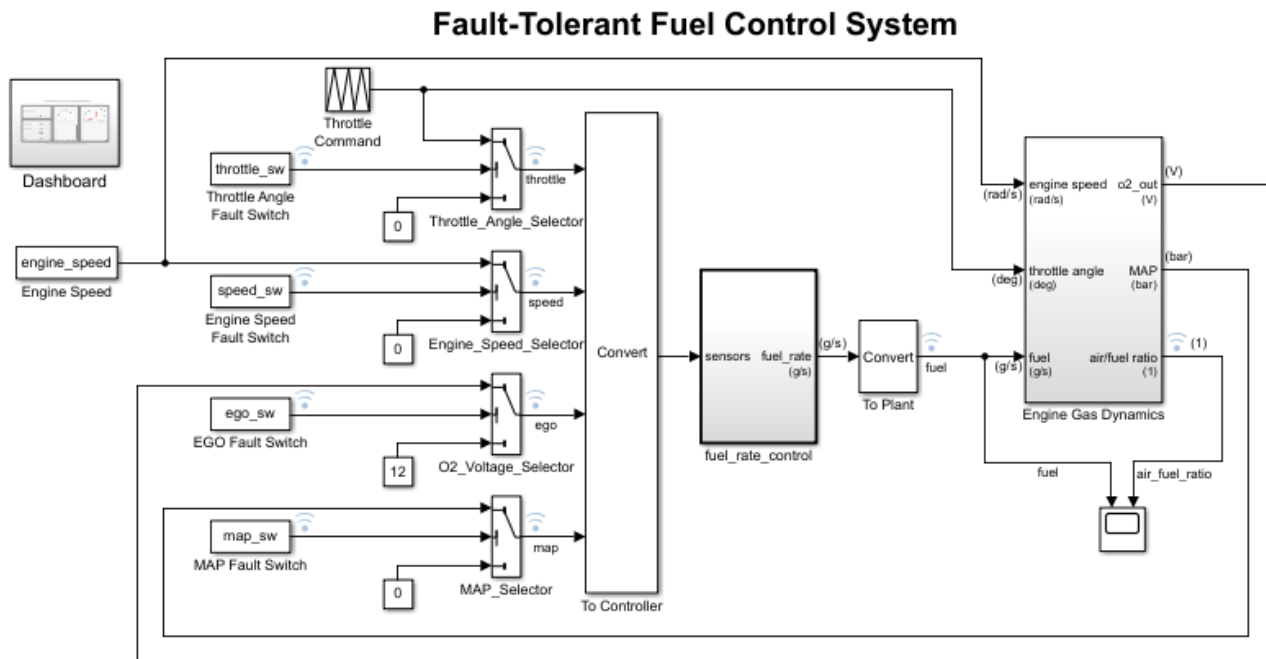
Detect and Fix Model Advisor Check Violations

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

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

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

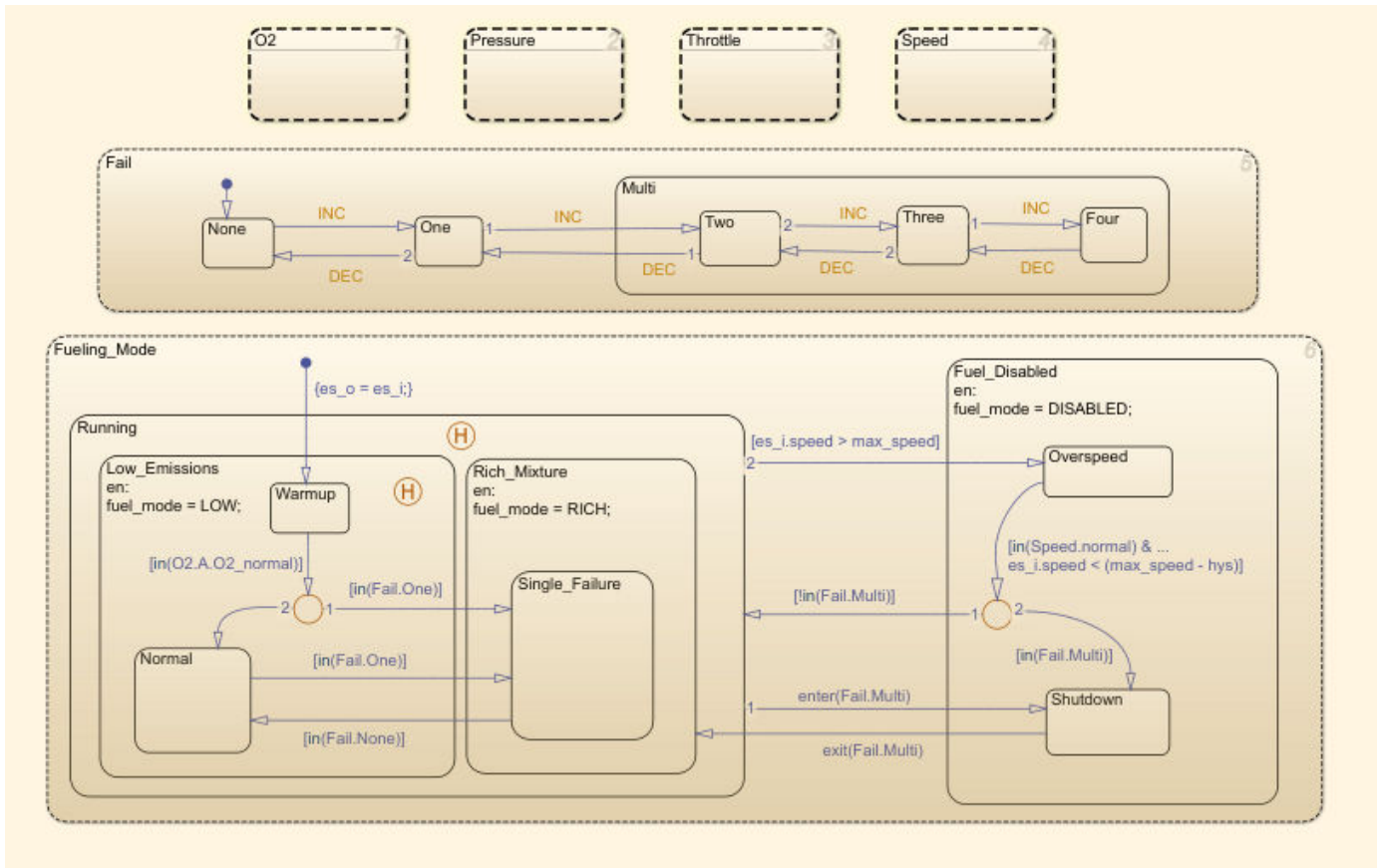
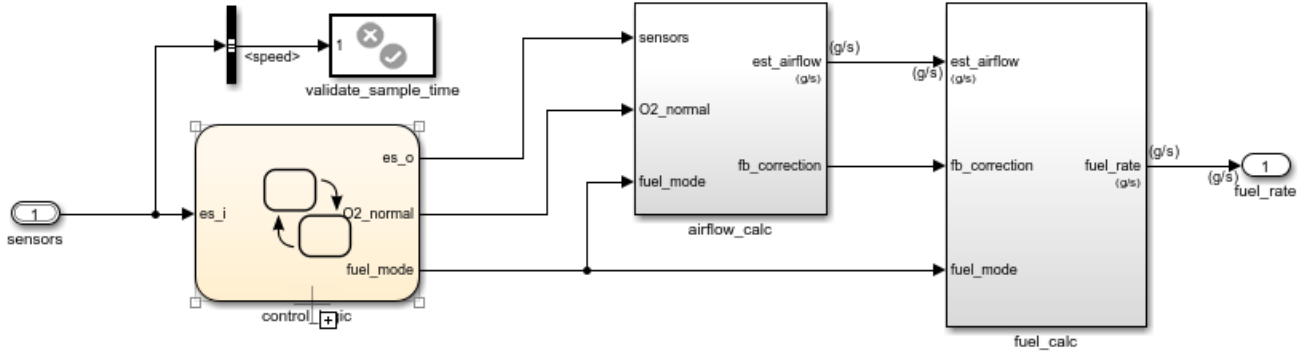
The figures show portions of the `sldemo_fuelsys` model. The top-level model is a closed-loop system that consists of a plant (Engine Gas Dynamics) and a controller (the Fuel Rate Control subsystem). The plant allows engineers to validate the controller through simulation early in the design cycle. The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



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

Copyright 1990-2017 The MathWorks, Inc.

Fuel Rate Control Subsystem



Detect and Fix Model Advisor Check Violations While You Edit

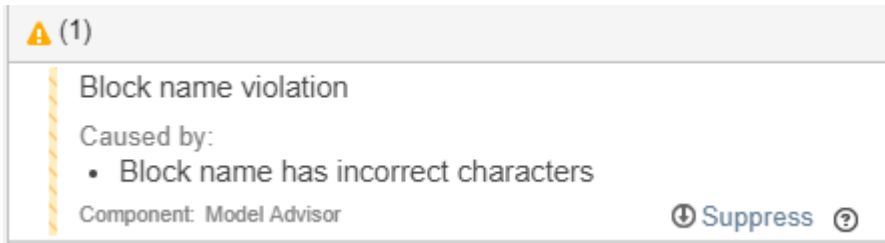
- 1 Set your current folder to a writeable directory.
- 2 Open the model `sldemo_fuelsys` by typing this command:

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

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

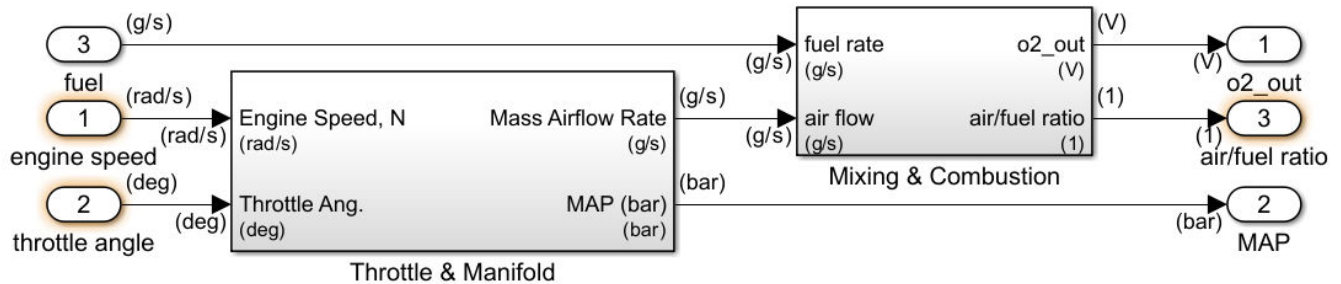
The highlighted blocks and subsystems indicate compliance issues.

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



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

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



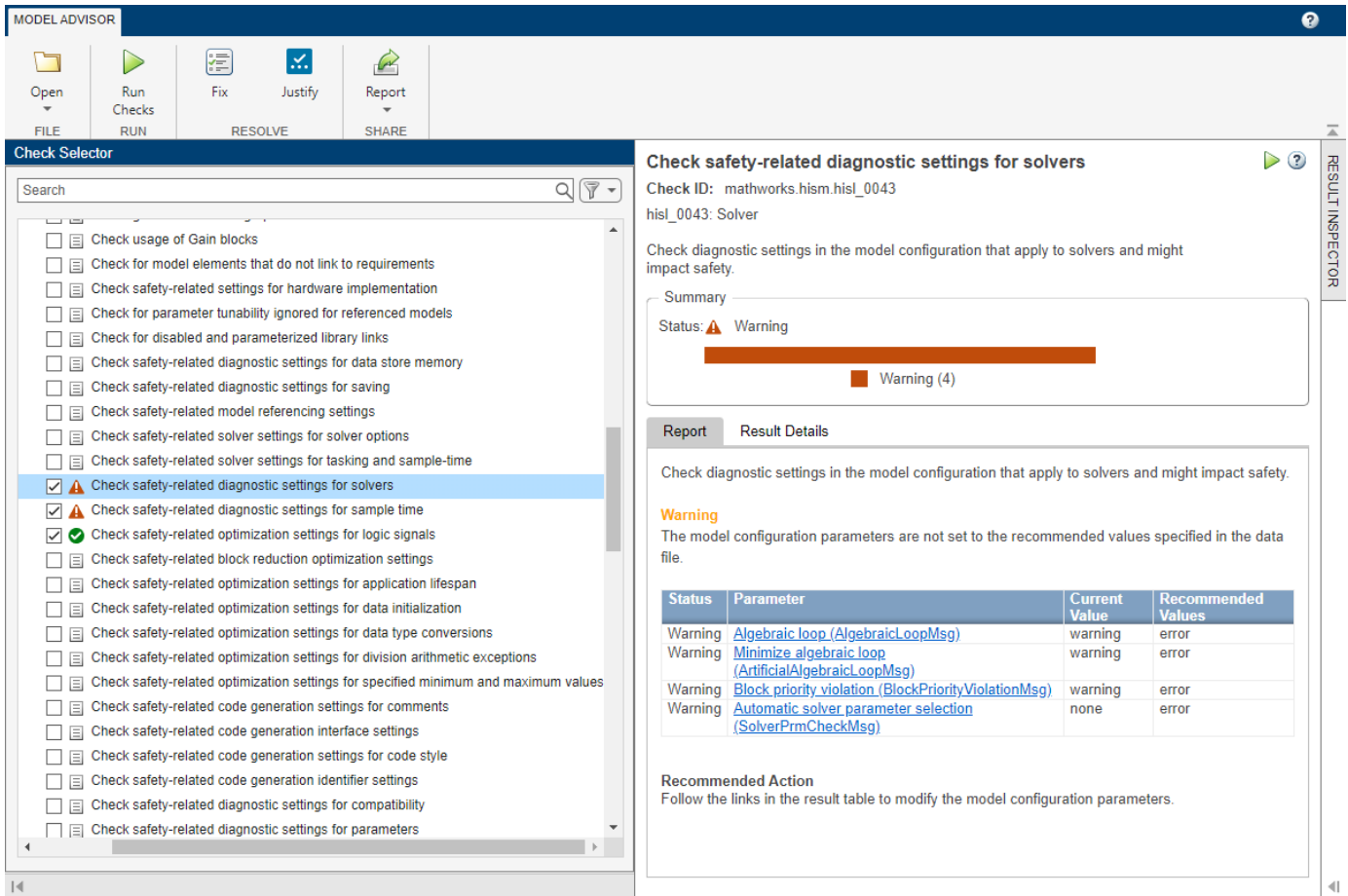
This output port returns warnings because its name violates two checks: “Check for unsupported block names” and “Check port block names”.

- Address the warnings by replacing the / symbol and the space in the block name with underscores. The block is no longer highlighted.
- Address the warnings for the other highlighted blocks in the Engine Gas Dynamics subsystem.

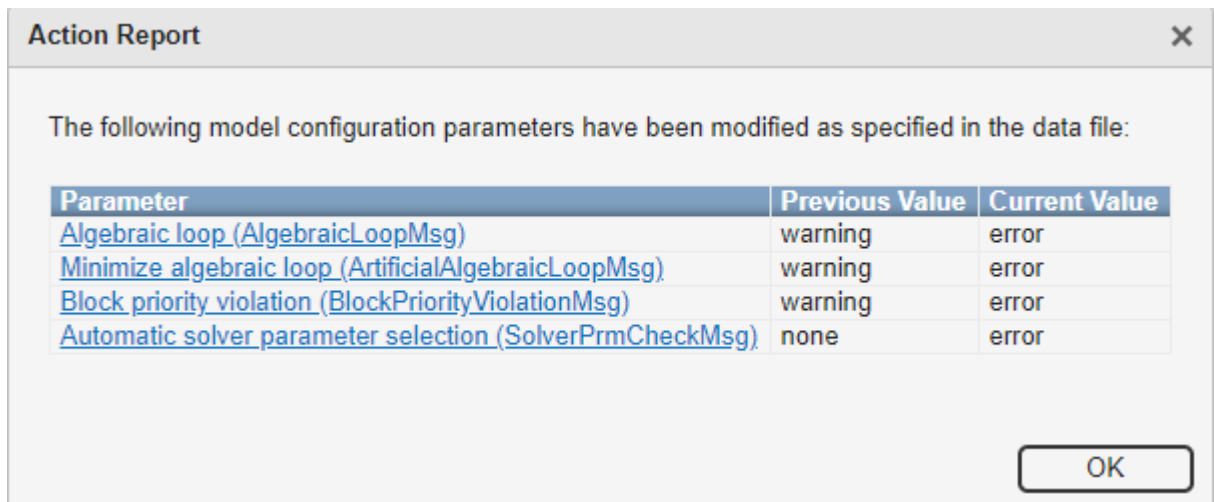
Detect and Fix Model Advisor Check Violations Interactively

- On the **Modeling** tab, select **Model Advisor**.
- Select the top-level model `sldemo_fuelSys` from the System Hierarchy and click **OK**.
- In the left pane, in the **By Product > Simulink Check > Model Standards > DO-178C/DO-331** folder, select:
 - **Check safety-related diagnostic settings for solvers**
 - **Check safety-related diagnostic settings for sample time**

- **Check safety-related optimization settings for logic signals**
- 4 Right-click **DO-178C/DO-331 Checks** node, and then select **Run Selected Checks**.



- 5 To review the configuration parameters that are not set to the recommended values, click **Check safety-related diagnostic settings for solvers**.
- 6 To update the parameters to the recommended values, in the toolbar, click **Fix**.



Action Report window displays the Model Advisor updates the parameters to the recommended values and details the result.

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

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

Collect Model Metric Data by Using the Metrics Dashboard

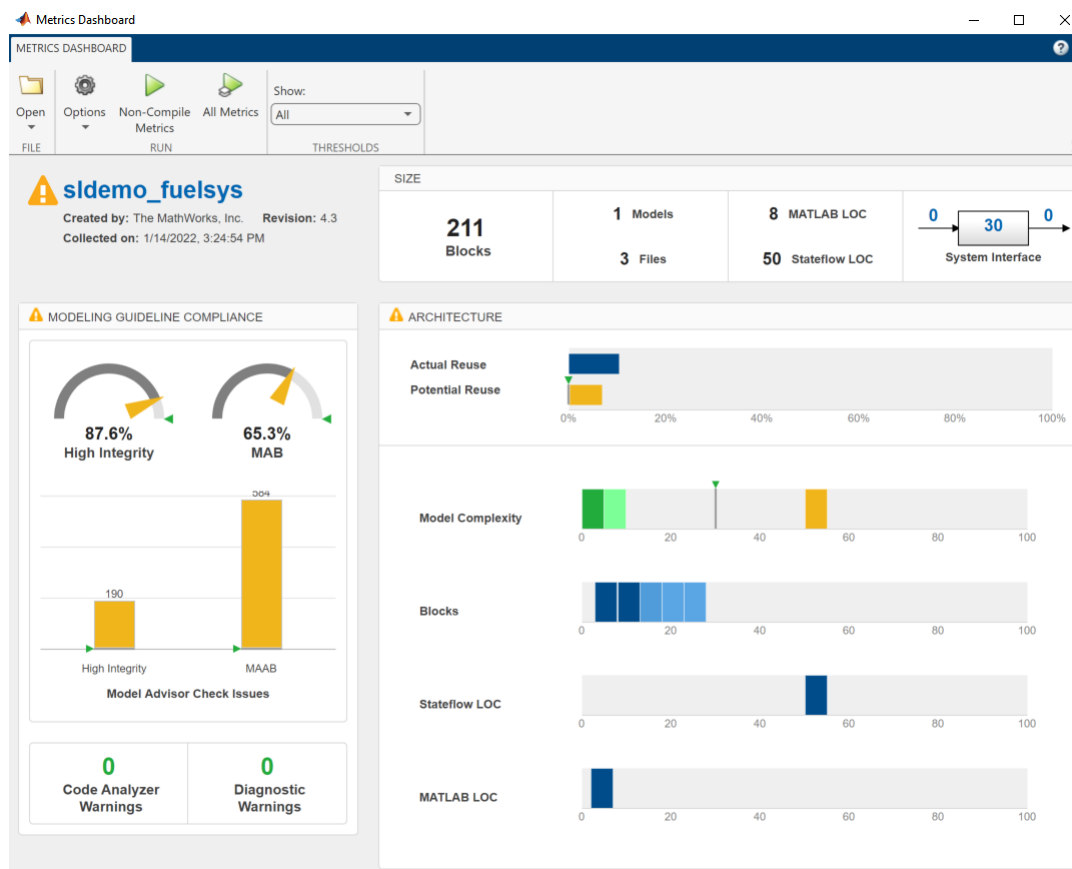
To collect model metric data and assess the design status and quality of your model, use the Metrics Dashboard. The Metrics Dashboard provides a view into the size, architecture, and guideline compliance of your model.

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



Analyze Metric Data

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



In the **Architecture** section of the dashboard, locate the **Model Complexity** widget. To view tooltips, pause over each vertical bar. This widget is a visual representation of the distribution of complexity

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

Explore Metric Data

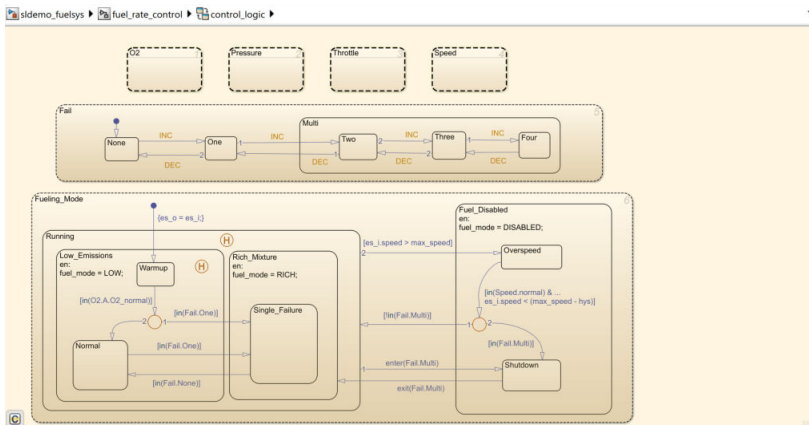
To explore metric data in more detail, click an individual metric widget. For your selected metric, a table displays the value, aggregated value, and measures (if applicable) at the model component level. From the table, the dashboard provides traceability and hyperlinks to the data source so that you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

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

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

Type	Component	Path	Qty	Model Complexity ↓	Model Complexity (Incl...
Chart	control_logic	aldemo_fuelsys/fuel_rate_control/control_logic	1	51	56
Model	aldemo_fuelsys	aldemo_fuelsys	1	5	80
Subsystem	Speed speed_estimate	.../fuel_rate_control/control_logic/Speed speed_estimate	1	3	3
Subsystem	feedforward_fuel_rate	...e/sys/fuel_rate_control/fuel_calc/feedforward_fuel_rate	1	2	2
Subsystem	switchable_compensation	...s/fuel_rate_control/fuel_calc/switchable_compensation	1	2	2
MATLAB Function	EGO Sensor	...gine Gas Dynamics/Mixing & Combustion/EGO Sensor	1	2	2
Subsystem	Throttle & Manifold	...emo_fuelsys/Engine Gas Dynamics/Throttle & Manifold	1	2	10
MATLAB Function	MATLAB Function	.../Throttle & Manifold/Intake Manifold/MATLAB Function	1	2	2
Subsystem	Throttle	...l/sys/Engine Gas Dynamics/Throttle & Manifold/Throttle	1	2	6
MATLAB Function	f(theta)	...gine Gas Dynamics/Throttle & Manifold/Throttle/f(theta)	1	2	2
MATLAB Function	g(pratio)	...ine Gas Dynamics/Throttle & Manifold/Throttle/g(pratio)	1	2	2
Subsystem	Throttle throttle_estimate	...uel_rate_control/control_logic/Throttle throttle_estimate	1	1	1
Subsystem	Pressure map_estimate	.../fuel_rate_control/control_logic/Pressure map_estimate	1	1	1
Subsystem	Mixing & Combustion	...o_fuelsys/Engine Gas Dynamics/Mixing & Combustion	1	1	3
Subsystem	fuel_rate_control	aldemo_fuelsys/fuel_rate_control	1	1	62
Subsystem	airflow_calc	aldemo_fuelsys/fuel_rate_control/airflow_calc	1	1	1
Subsystem	fuel_calc	aldemo_fuelsys/fuel_rate_control/fuel_calc	1	0	4
Subsystem	disabled_mode	...trol/fuel_calc/switchable_compensation/disabled_mode	1	0	0
Subsystem	low_mode	..._control/fuel_calc/switchable_compensation/low_mode	1	0	0
Subsystem	rich_mode	..._control/fuel_calc/switchable_compensation/rich_mode	1	0	0
Subsystem	System Lag	...gine Gas Dynamics/Mixing & Combustion/System Lag	1	0	0
Subsystem	Intake Manifold	...gine Gas Dynamics/Throttle & Manifold/Intake Manifold	1	0	2
Subsystem	Dashboard	aldemo_fuelsys/Dashboard	1	0	0
Subsystem	Throttle Command	aldemo_fuelsys/Throttle Command	1	0	0
Subsystem	To Controller	aldemo_fuelsys/To Controller	1	0	0

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



Refactor Model Based on Metric Data

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

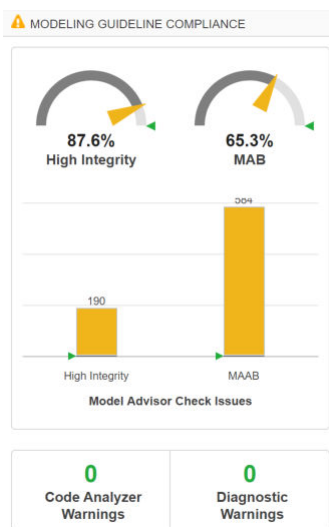
Next, you will use the **Modeling Guideline Compliance** widgets to fix issues associated with high-integrity Model Advisor checks.

Detect and Fix Compliance Issues

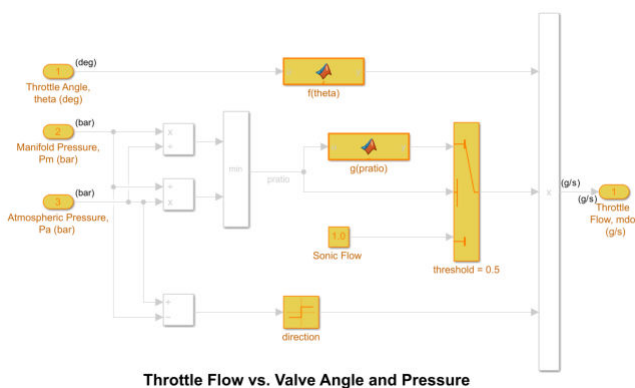
When you collect metric data, the Metrics Dashboard runs the MAB and high-integrity Model Advisor checks in the background. The **Modeling Guideline Compliance** section of the dashboard provides the percentages of checks that pass and a count of check warnings and errors. You can interactively investigate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

Explore Compliance Results in the Dashboard

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



- 2 To see a table that details the number of compliance issues by component, click on the **High Integrity** bar chart. For more information on this metric, see “Model Advisor Check Issues for High-Integrity Systems”.
- 3 From the table, click the **Throttle** component hyperlink. The **Throttle** component opens in the model editor. The model editor highlights blocks in the component that have compliance issues. The Model Advisor Highlighting dialog box lists checks that do not highlight results.



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

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

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

- 6 To see the status for each compliance check, click the **Table** view.
- 7 Expand the `sldemo_fuelsys` node.
- 8 To explore check results in more detail, click the **Check safety-related diagnostic settings for model referencing** hyperlink.
- 9 In the Model Advisor Highlighting dialog box, click **Check safety-related diagnostic settings for model referencing** hyperlink.

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

Update Model to Fix Compliance Issues

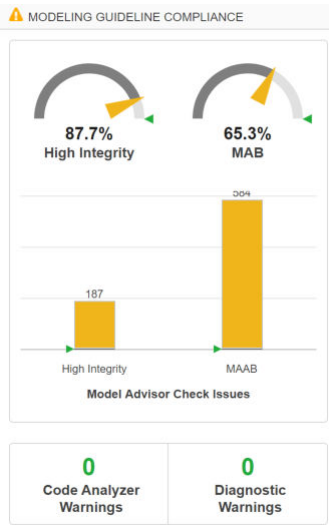
- 1 To change the current value of a parameter to the recommended value, click a parameter. The Configuration Parameters dialog box opens.
- 2 Change the parameter setting to what the **Recommended Value** column in the report indicates.
- 3 Click **Apply** and close the dialog box.
- 4 Close the Model Advisor report.
- 5 In the Model Advisor Highlighting dialog box, click the **Check safety-related diagnostic settings for compatibility** check.

The Model Advisor report opens.

- 6 For this check, repeat steps 1 through 4.
- 7 In the Model Advisor Highlighting dialog box, click the **Check safety-related diagnostic settings for bus connectivity** check.
- 8 For this check, repeat steps 1 through 4.
- 9 Close the Model Advisor Highlighting dialog box and return to the Metrics Dashboard table

Rerun Model Metrics

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



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

Refactor Models to Improve Component Reuse

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

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

Identify and Replace Clones with Links to Library Blocks

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

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

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

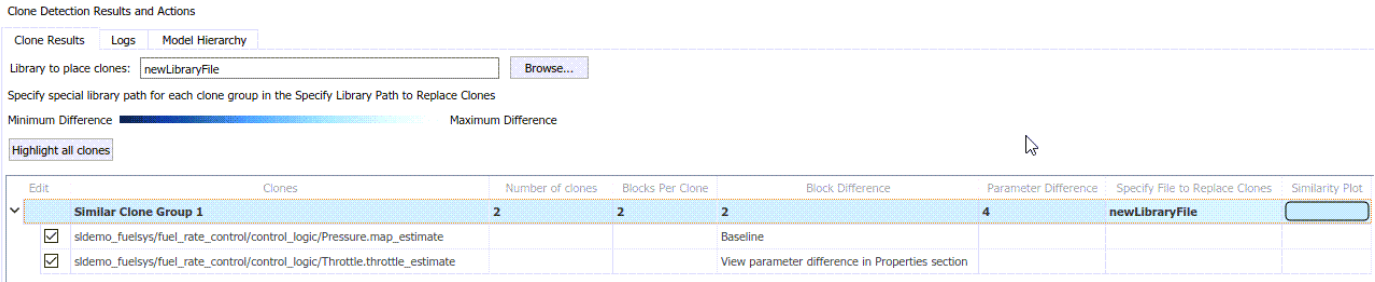
The **Clone Detector** app opens as a new tab in the model.

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

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

- 8 Expand the clone group.

This clone group consists of two subcharts.



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

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

- 10 In the **Clone Results** tab, for the **Library to place clones** parameter, use the **Browse** button to choose a library or specify a new library name. If you specify a new library name, the app creates the library.
- 11 Save the model to your working folder and, in the **Clone Detector** tab, click **Replace Clones**. The app replaces similar clones with links to masked library subsystems, if possible.

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

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

- 12 Close the Metrics Dashboard and the model.

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

Explore Other Options

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


Task	Reference
Simplify and debug complex models.	“Highlight Functional Dependencies” on page 8-2
Run Model Advisor checks for compliance with safety standards associated with High-Integrity System Modeling and MAB Control Algorithm Modeling guidelines.	“Check Model Compliance by Using the Model Advisor” on page 3-2
Write custom Model Advisor checks.	“Define Custom Model Advisor Checks” on page 6-45
Create and deploy a custom Model Advisor configuration.	“Create and Deploy a Model Advisor Custom Configuration” on page 7-24 and “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3

Task	Reference
Learn more about how to use the Metrics Dashboard to collect and view metric data for quality assessment.	“Collect and Explore Metric Data by Using the Metrics Dashboard” on page 5-2
Configure compliance metrics, add metric thresholds, and customize the Metrics Dashboard layout.	“Customize Metrics Dashboard Layout and Functionality” on page 5-37
Use the Model Transformer tool and the Clone Detector app to refactor a model to improve model componentization and readability and enable reuse.	“Transform Model to Variant System” on page 3-19 and “Enable Component Reuse by Using Clone Detection” on page 3-29
Learn more about how to use Simulink products to test models and code, check for design errors, check against standards, measure coverage, and validate the system.	“Verification and Validation”

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

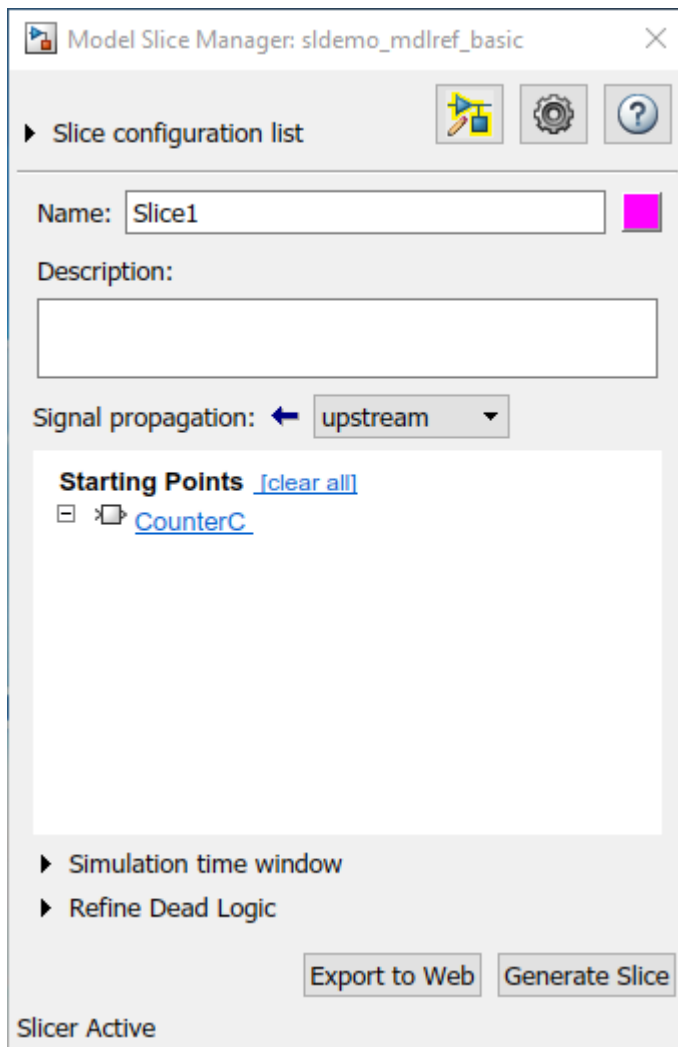
You can simplify simulation, debugging, and formal analysis of large, complex models by focusing on areas of interest in your model. After highlighting a portion of your model using the Model Slicer, you can generate a simplified standalone model. The simplified model contains the blocks and dependency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

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

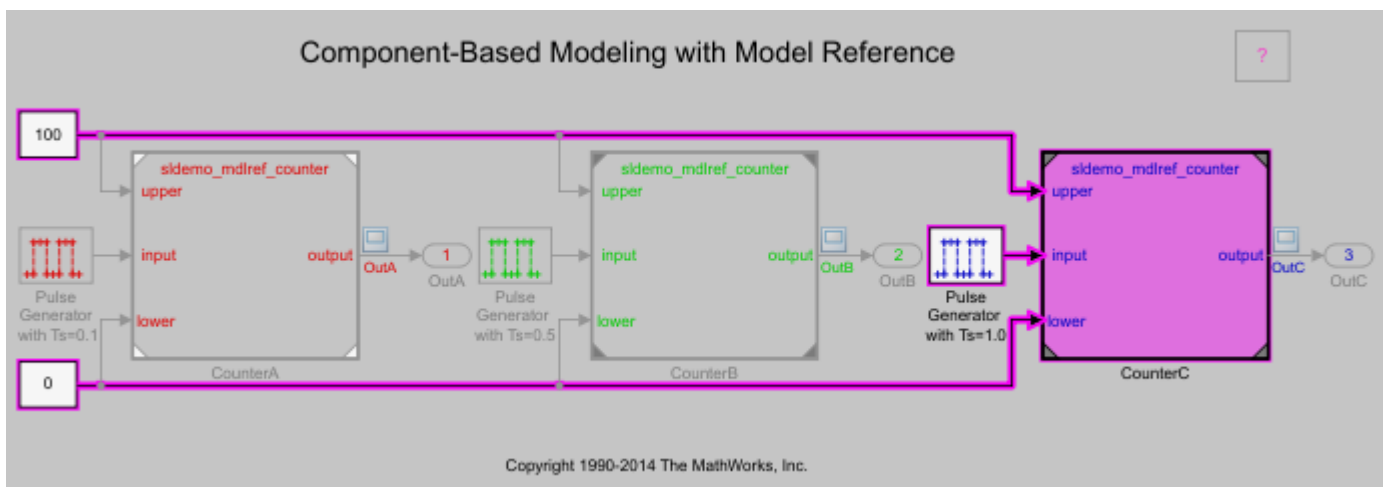
```
openExample('sldemo_mdhref_basic');
```
- 2 To open the Model Slicer Manager, on the Model Verification, Validation, and Test section of the **Apps** tab, click **Model Slicer**.
- 3 In the Model Slice Manager, click the arrow to expand the **Slicer configuration list**.
- 4 Set the slice properties:
 - **Name:** Slice1
 - **Color:**  (magenta)
 - **Signal Propagation:** upstream

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

- 5 Add CounterC as a starting point. In the model, right-click CounterC and select **Model Slicer > Add as Starting Point**.



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



- 6 In the Model Slice Manager, click **Generate slice**.
- 7 In the **Select File to Write** dialog box, select the save location and enter a model name. The simplified standalone model contains the highlighted model items.

Component-Based Modeling with Model Reference



Copyright 1990-2014 The MathWorks, Inc.

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

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

See Also

More About

- “Model Slicer Considerations and Limitations” on page 8-44
- “Highlight Functional Dependencies” on page 8-2
- “Refine Highlighted Model” on page 8-12

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

You can assess the status of your model testing activities by using the metrics in the Model Testing Dashboard. When you test your models against requirements, you maintain traceability between the requirements, models, tests, and results. The dashboard helps you to track the status of these artifacts and the traceability relationships between them. Each metric in the dashboard measures a different aspect of the quality of the testing artifacts and reflects guidelines in industry-recognized software development standards, such as ISO 26262 and DO-178C. For more information, see “Assess Requirements-Based Testing for ISO 26262” on page 5-102.

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

Open the Project and Model Testing Dashboard

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

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

```
dashboardCCProjectStart("incomplete")
```

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

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

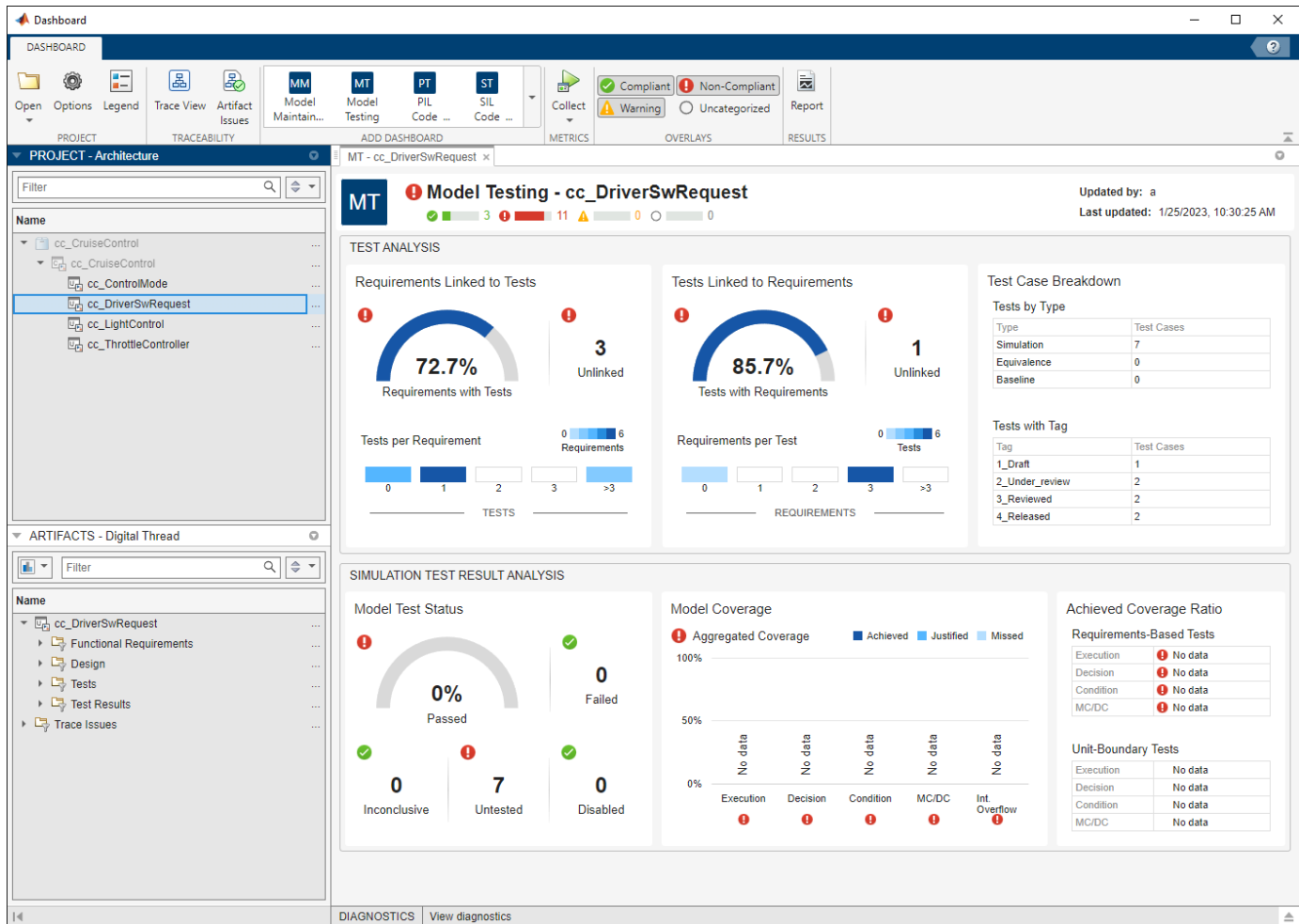
The first time that you open the dashboard for the project, the dashboard must identify the artifacts in the project and collect traceability information.

The **Project** panel shows the architecture of the software units and components in the current project. The **Artifacts** panel shows the artifacts that the dashboard traced to the current unit or component selected in the **Project** panel.

The dashboard performs an initial traceability analysis and collects metric results for the metrics available in your installation. Collecting results for each of the MathWorks® metrics requires licenses for Simulink Check, Requirements Toolbox™, and Simulink Test. If metric results have been collected, viewing the results requires only a Simulink Check license.

- 3 In the **Project** panel, select the unit **cc_DriverSwRequest**.

The dashboard analyzes the traceability links from the artifacts to the models in the project and populates the widgets with metric results for the unit that is selected in the **Project** panel.



Assess Traceability of Artifacts

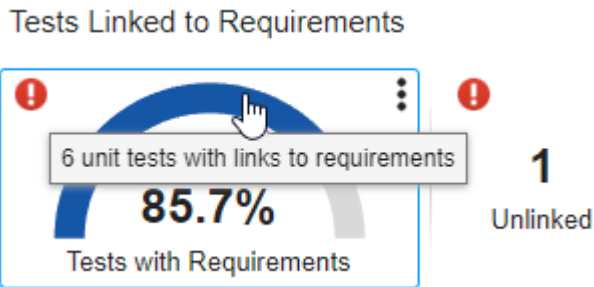
When the dashboard collects and reports metric data, it scopes the results to the artifacts in one unit in the project. Use the **Artifacts** panel to view the artifacts that trace to the unit that you selected in the **Project** panel.

- 1 In the **Artifacts** panel, expand the **Functional Requirements** section and then expand the **Implemented** and **Upstream** sections. This unit implements the requirements in the file `cc_SoftwareReqs.slreqx` and links to the upstream, system-level requirements in `das_SystemReqs.slreqx`.
- 2 Click the arrow to the left of a file name to see the individual requirements that trace to the model.
- 3 To see the artifact type and the path to the artifact, click the three dots next to the artifact name to open a tooltip.

You can explore the sections in the **Artifacts** panel to see which requirements, tests, and test results trace to each unit in the project. For more information on how the dashboard analyzes this traceability, see “Trace Artifacts to Units and Components” on page 5-96.

Explore Metric Results for a Unit

- 1 In the **Project** panel, click the unit **cc_DriverSwRequest** to view the **Model Testing** results. The dashboard widgets populate with the metric results for the unit.
- 2 In the **Test Analysis** section of the dashboard, locate the **Tests with Requirements** widget. To view tooltips with details about the results, point to the sections of the gauge or to the percentage result.



- 3 To explore the metric data in more detail, click an individual metric widget. For example, click the **Tests with Requirements** widget to view the **Metric Details** for the metric.



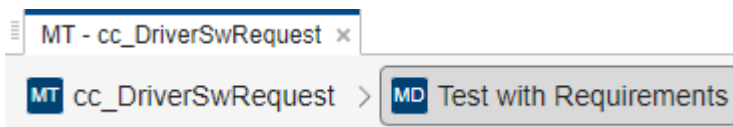
Metric Details - Tests linked to requirements

Metric that determines if each test case or test iteration for the model is linked to at least one requirement in the project.

Artifact	Source	Requirement Link Status
▶ Detect short decrement	cc_DriverSwRequest_Tests.mldatx	Linked to requirements
▶ Detect long increment	cc_DriverSwRequest_Tests.mldatx	Linked to requirements
▶ Detect set	cc_DriverSwRequest_Tests.mldatx	Linked to requirements
▶ Detect short increment	cc_DriverSwRequest_Tests.mldatx	Linked to requirements
▶ Detect cruise	cc_DriverSwRequest_Tests.mldatx	Linked to requirements
▶ Detect long decrement	cc_DriverSwRequest_Tests.mldatx	Missing linked requirements
▶ Detect cancel	cc_DriverSwRequest_Tests.mldatx	Linked to requirements

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

- 4 The table shows that the test **Detect long decrement** is missing linked requirements. You can use the Model Testing Dashboard to open the test in the Test Manager. In the **Artifact** column, click **Detect long decrement** to open the test case in Test Manager.
- 5 At the top of the Model Testing Dashboard, there is a breadcrumb trail from the **Metric Details** back to the **Model Testing** results.



Click the breadcrumb button for **cc_DriverSwRequest** to return to the **Model Testing** results for the unit.

You can click on any of the widgets in the dashboard to view the details of their metric results. Use the hyperlinks in the tables to open the artifacts and address testing gaps. For more information on using the data in the dashboard, see “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80.

Track Testing Status of a Project Using the Model Testing Dashboard

To use the Model Testing Dashboard to track your testing activities, set up and maintain your project using the best practices described in “Manage Project Artifacts for Analysis in Dashboard” on page 5-95. As you develop and test your models, use the dashboard to identify testing gaps, fix the underlying artifacts, and track your progress towards model testing completion. For more information on finding and addressing gaps in your model testing, see “Fix Requirements-Based Testing Issues” on page 5-89.

See Also

“Model Testing Metrics”

More About

- “Assess Requirements-Based Testing for ISO 26262” on page 5-102
- “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80

Verification and Validation

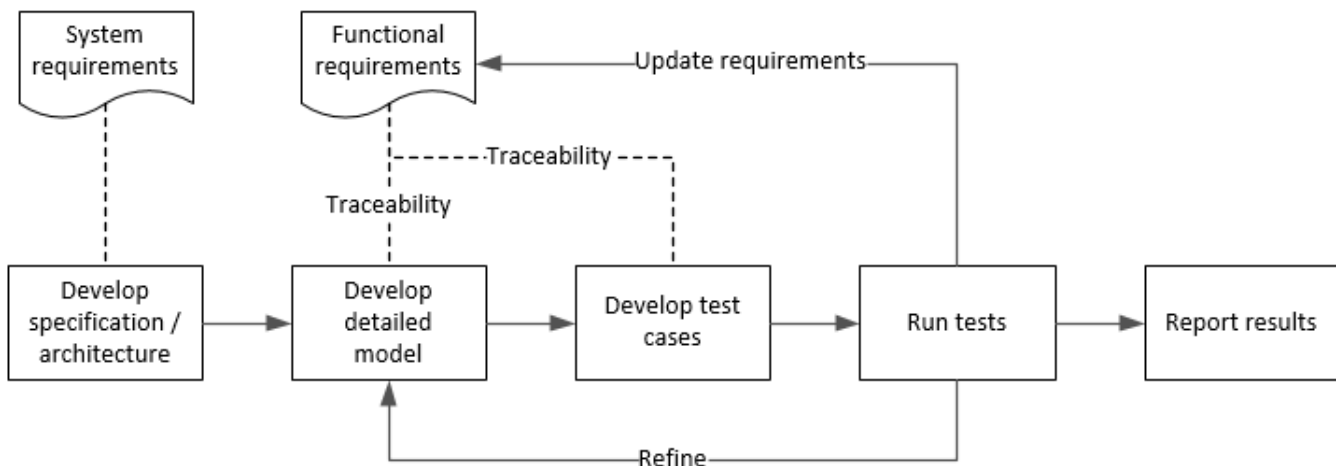
- “Test Model Against Requirements and Report Results” on page 2-2
- “Analyze Models for Standards Compliance and Design Errors” on page 2-7
- “Perform Functional Testing and Analyze Test Coverage” on page 2-9
- “Analyze Code and Test Software-in-the-Loop” on page 2-12

Test Model Against Requirements and Report Results

Requirements - Test Traceability Overview

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

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




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

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

Display the Requirements

- 1 Open the example project.

```
openExample("shared_vnv/CruiseControlVerificationProjectExample");
pr = openProject("SimulinkVerificationCruise");
```

- 2 In the `models` folder, open the `simulinkCruiseAddReqExample` model.
- 3 Display the requirements. Click the  icon in the lower-right corner of the model canvas, and select **Requirements**. The requirements appear below the model canvas.
- 4 Display the verification and implementation status. Right-click a requirement and select **Verification Status** and **Implementation Status**.

The screenshot displays the Simulink environment with a model titled 'simulinkCruiseAddReqExample'. The model includes several input blocks: 'CruiseOnOff' (boolean), 'Brake' (boolean), 'Speed' (single), 'CoastSetSw' (boolean), and 'AccelResSw' (boolean). These inputs feed into a central 'Compute target speed' block. The output of this block is 'tspeed', which is connected to two 'engaged' outputs. The 'Requirements' window shows a table with the following data:

Index	ID	Summary	Verified	Implemented
simulinkCruiseChar...				
1	Architecture	Architecture		
1.1	A 1.1	Enable Disable Switch		
1.2	A 1.2	Set Speed / Decelerate Bu...		
1.3	A 1.3	Resume Speed / Accelerat...		
1.4	A 1.4	Engaged Output		
1.5	A 1.5	Target Speed Output		
1.6	A 1.6	Vehicle Speed Input		
1.7	A 1.7	Vehicle Brake Input		
2	Functional Requirements	Functional Requirements		
3	Safety Requirements	Safety Requirements		

The Property Inspector on the right shows details for Requirement: A 1.2, including its type (Functional), index (1.2), and summary (Set Speed / Decelerate Button). The description field contains the text: 'The controller shall have an input button to: set the target speed to the current vehicle speed when the cruise control is **not engaged (active)** decelerate (reduce) the target speed when the cruise control is **engaged (active)**'.

- 5 In the Project window, open the Simulink Test file `s1ReqTests.mldatx` from the `tests` folder. The test file opens in the Test Manager.

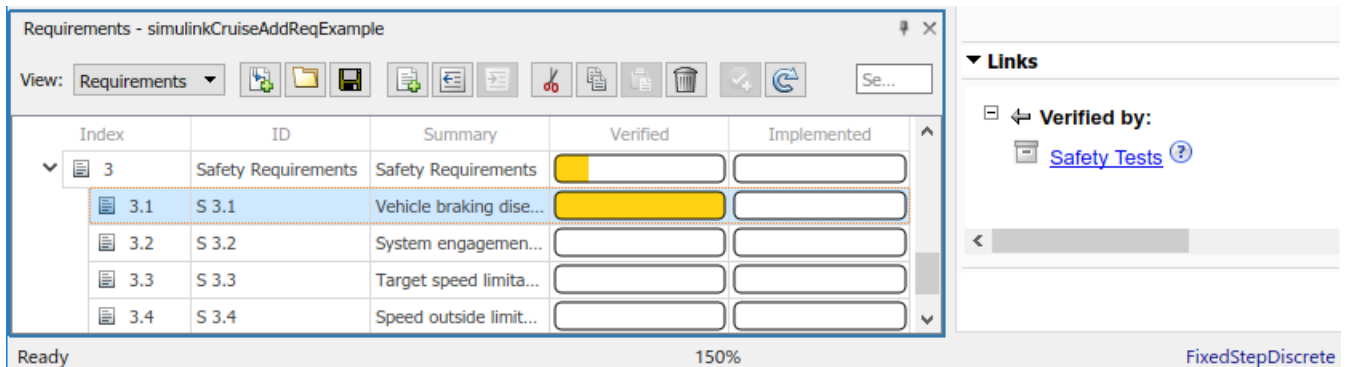
Link Requirements to Tests

Link the requirements to the test case.

- 1 In the Project window, open the Simulink Test file `s1ReqTests.mldatx` from the `tests` folder. The test file opens in the Test Manager. Explore the test suite and select `Safety Tests`.

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

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



- 2 Also add a link for item S 3.4.

Run the Test

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

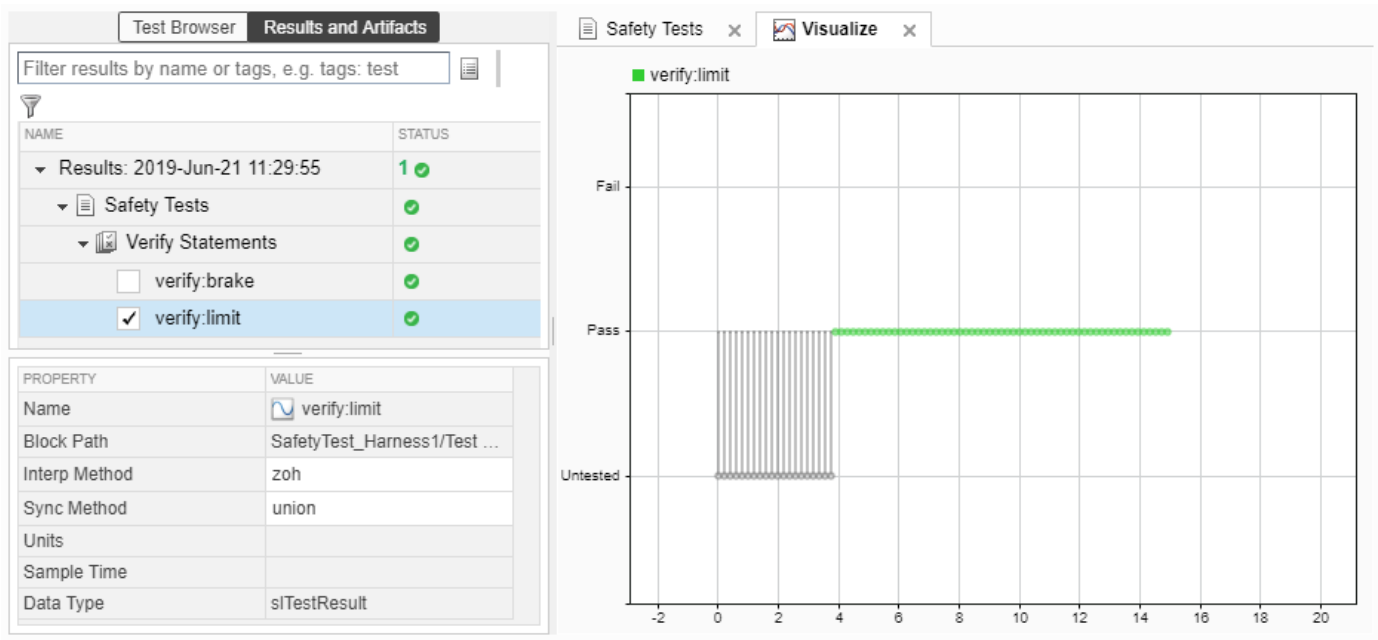
- The `BrakeTest` sequence engages the cruise control, then applies the brake. It includes the `verify` statement

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

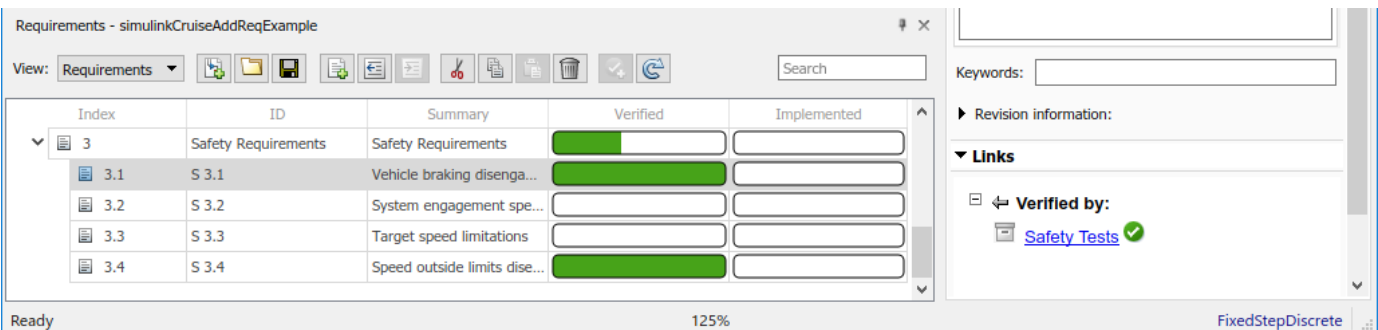
- The `LimitTest` sequence engages the cruise control, then ramps up the vehicle speed until it exceeds the upper limit. It includes the `verify` statement.

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

- 1 Return to the Test Manager. To run the test case, click **Run**.
- 2 When the test finishes, review the results. The Test Manager shows that both assessments pass and the plot provides the detailed results of each `verify` statement.



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



Report the Results

- 1 Create a report using a custom Microsoft Word template.
 - a From the Test Manager results, right-click the test case name. Select **Create Report**.
 - b In the Create Test Result Report dialog box, set the options:
 - Title — SafetyTest
 - Results for — All Tests
 - File Format — DOCX
 - For the other options, keep the default selections.
 - c Enter a file name and select a location for the report.
 - d For the **Template File**, select the ReportTemplate.dotx file in the **documents** project folder.
 - e Click **Create**.

- 2 Review the report.
 - a The **Test Case Requirements** section specifies the associated requirements
 - b The **Verify Result** section contains details of the two assessments in the test, and links to the simulation output.

See Also

Related Examples

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

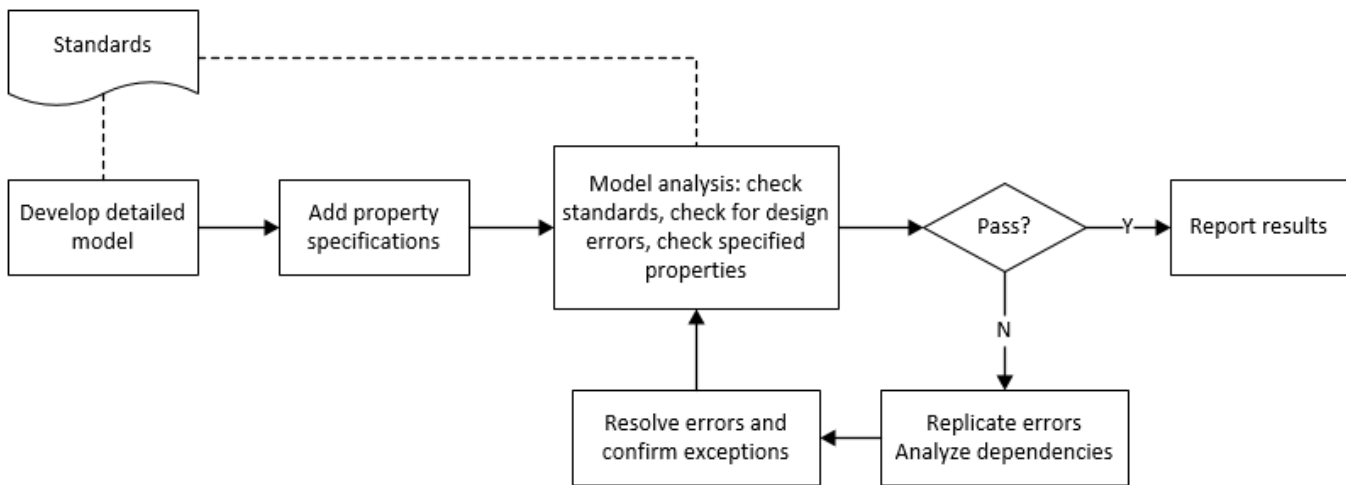
External Websites

- Requirements-Based Testing Workflow

Analyze Models for Standards Compliance and Design Errors

Standards and Analysis Overview

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



Check Model for Style Guideline Violations and Design Errors

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

Check Model for MAB Style Guideline Violations

Check that your model complies with MAB guidelines by using the Model Advisor.

- 1 Open the example project. On the command line, enter


```
openExample("shared_vnv/CruiseControlVerificationProjectExample");
pr = openProject("SimulinkVerificationCruise");
```
- 2 Open the `simulinkCruiseErrorAndStandardsExample` model.


```
open_system simulinkCruiseErrorAndStandardsExample
```
- 3 In the **Modeling** tab, select **Model Advisor**.
- 4 Click **OK** to select `simulinkCruiseErrorAndStandardsExample` from the System Hierarchy.
- 5 Check your model for MAB style guideline violations using Simulink Check.
 - a In the left pane, in the **By Product > Simulink Check > Modeling Standards > MAB Checks** folder, select:

- **Check Indexing Mode**
 - **Check model diagnostic parameters**
- b** Right-click on the **MAB Checks** node and select **Run Checks**.
 - c** To review the configuration parameter settings that violate MAB style guidelines, run the **Check model diagnostic parameters** check.
 - d** The analysis results appear in the right pane on the **Report** tab. Report displays the violation details and the recommended action.
 - e** Click the parameter hyperlinks, which opens the Configuration Parameters dialog box, and update the model diagnostic parameters. Save the model.
 - f** To verify that your model passes, rerun the check. Repeat steps from c to e, if necessary, to reach compliance.
 - g** To generate a results report of the Simulink Check checks, select the **MAB Checks** node, and then, from the toolbar, click **Report**.

Check Model for Design Errors

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

- 1** In the left pane, in the **By Products > Simulink Design Verifier** folder, select **Design Error Detection**.
- 2** If not already checked, click the box beside **Design Error Detection**. All checks in the folder are selected.
- 3** From the tool strip, click **Run Checks**.
- 4** After the Model Advisor analysis, from the tool strip, click **Report**. This generates a HTML report of the check analysis.
- 5** In the generated report, click a **Simulink Design Verifier Results Summary** hyperlink. The dialog box provides tools to help you diagnose errors and warnings in your model.
 - a** Review the analysis results on the model. Click the **Compute target speed** subsystem. The Simulink Design Verifier Results Inspector window provides derived ranges that can help you understand the source of an error by identifying the possible signal values.
 - b** Review the harness model or create one if it does not already exist.
 - c** View tests and export test cases.
 - d** Review the analysis report. To see a detailed analysis report, click **HTML** or **PDF**.

See Also

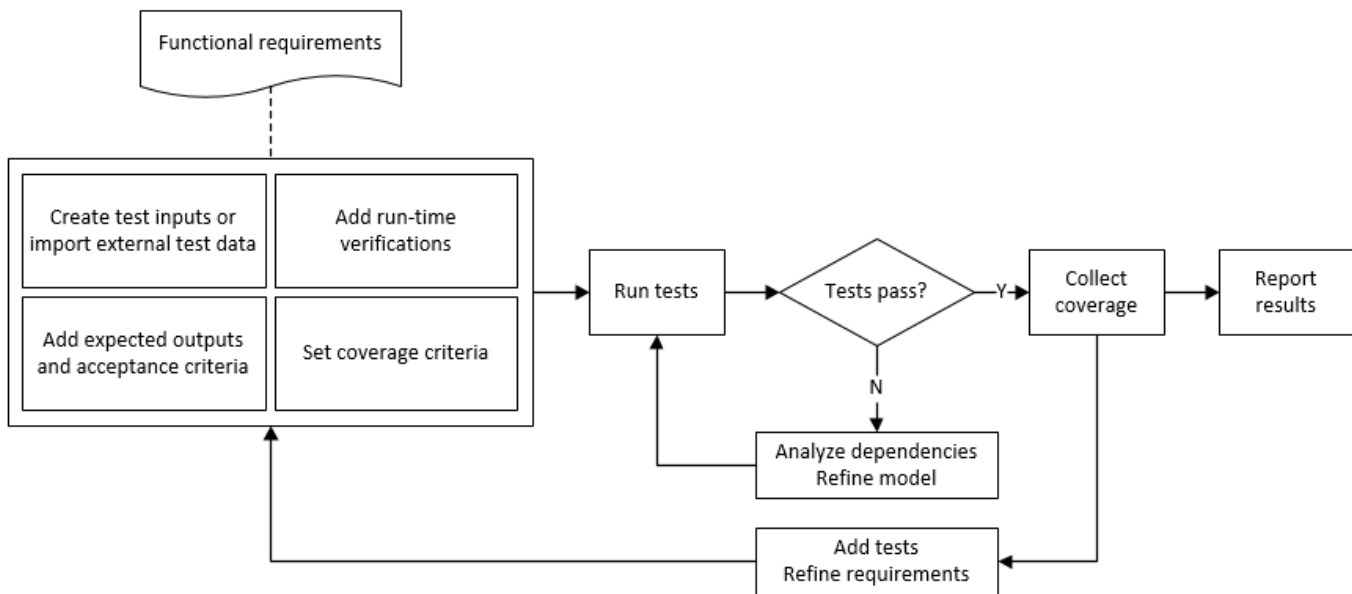
Related Examples

- “Check Model Compliance by Using the Model Advisor” on page 3-2
- “Collect Model Metrics Using the Model Advisor” on page 5-9
- “Analyze Models for Design Errors” (Simulink Design Verifier)
- “Prove Properties in a Model” (Simulink Design Verifier)

Perform Functional Testing and Analyze Test Coverage

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

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



Incrementally Increase Test Coverage Using Test Case Generation

This example shows how to perform requirements-based tests for a cruise control model. The tests link to a requirements document. You:

- 1 Run the tests.
- 2 Determine test coverage by using Simulink Coverage.
- 3 Increase coverage with additional tests generated by Simulink Design Verifier.
- 4 Report the results.

Open the Test Harness and Model

- 1 Open the project:

```
openExample("shared_vnv/CruiseControlVerificationProjectExample");
pr = openProject("SimulinkVerificationCruise");
```

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

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

- 3 Load the test suite from “Test Model Against Requirements and Report Results” (Simulink Test) and open the Simulink Test Manager.

```
pf = fullfile(pr.RootFolder,"tests","slReqTests.mldatx");
tf = sltest.testmanager.TestFile(pf);
sltest.testmanager.view
```

- 4 Open the Test Sequence block. The sequence verifies system disengagement when either:
 - The brake pedal is pressed.
 - Speed exceeds a limit.

Measure Model Coverage

- 1 In the Simulink Test Manager, select the `slReqTests` test file.
- 2 To enable coverage collection, in the right page under **Coverage Settings**:
 - Select **Record coverage for referenced models**.
 - Specify a coverage filter by using **Coverage filter filename**.
 - Select **Decision**, **Condition**, and **MCDC**.
- 3 Click **Run** on the Test Manager toolstrip.
- 4 After the test completes, select **Results**. The test achieves 50% decision coverage, 41% condition coverage, and 25% MCDC coverage.

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

Generate Tests to Increase Model Coverage

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

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

See Also

Related Examples

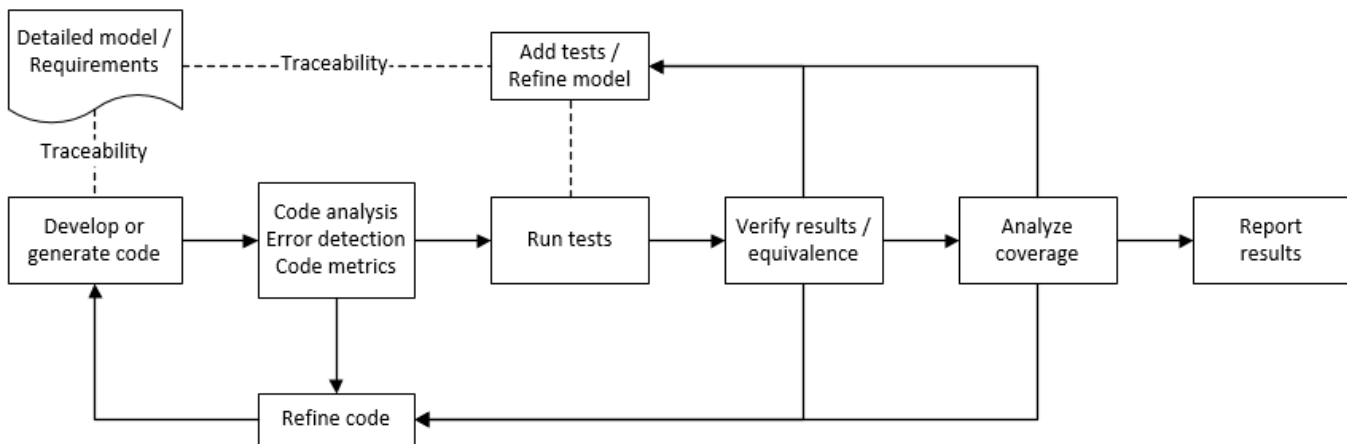
- “Link to Requirements” (Simulink Test)
- “Assess Model Simulation Using verify Statements” (Simulink Test)
- “Compare Model Output to Baseline Data” (Simulink Test)
- “Generate Test Cases for Model Decision Coverage” (Simulink Design Verifier)
- “Increase Test Coverage for a Model” (Simulink Test)

Analyze Code and Test Software-in-the-Loop

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

You can analyze code to detect errors, check standards compliance, and evaluate key metrics such as length and cyclomatic complexity. For handwritten code, you typically check for run-time errors with static code analysis and run test cases that evaluate the code against requirements and evaluate code coverage. Based on the results, you refine the code and add tests.

In this example, you generate code and demonstrate that the code execution produces equivalent results to the model by using the same test cases and baseline results. Then you compare the code coverage to the model coverage. Based on test results, add tests and modify the model to regenerate code.



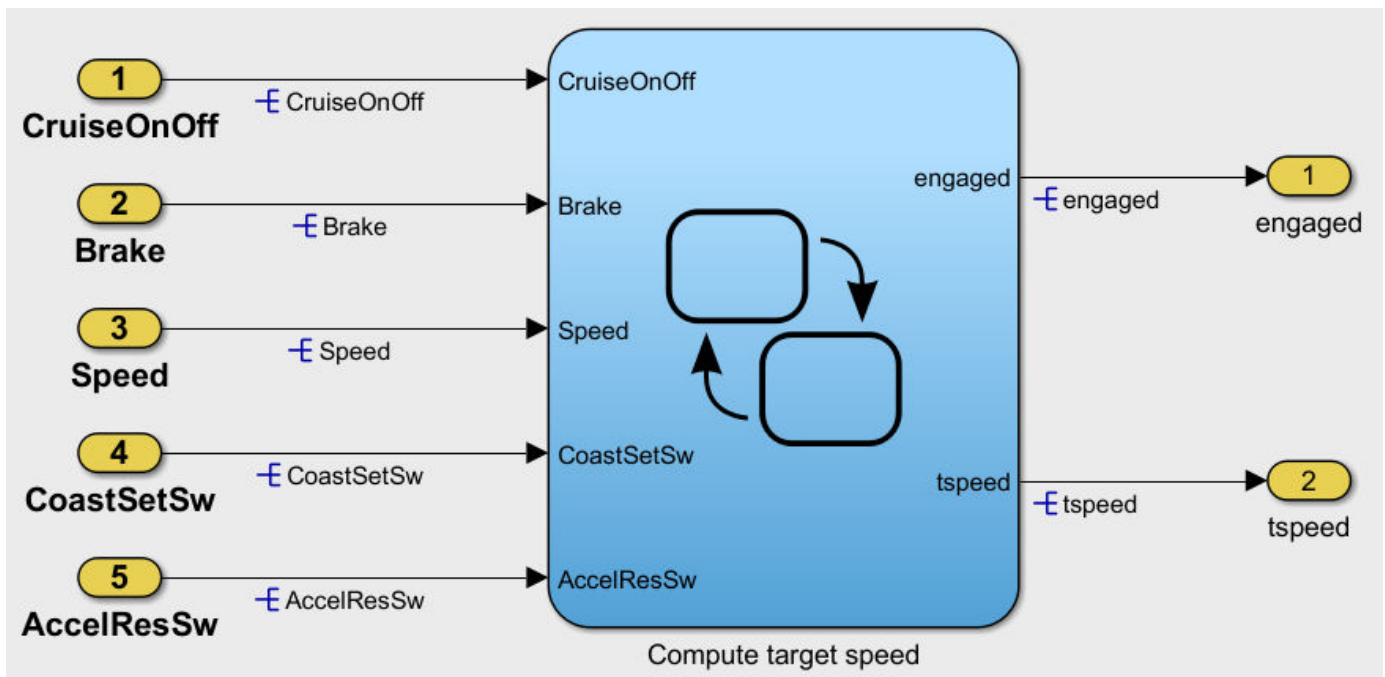
Analyze Code for Defects, Metrics, and MISRA C:2012

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

- 1 Open the project.

```
openExample("shared_vnv/CruiseControlVerificationProjectExample");
pr = openProject("SimulinkVerificationCruise");
```

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

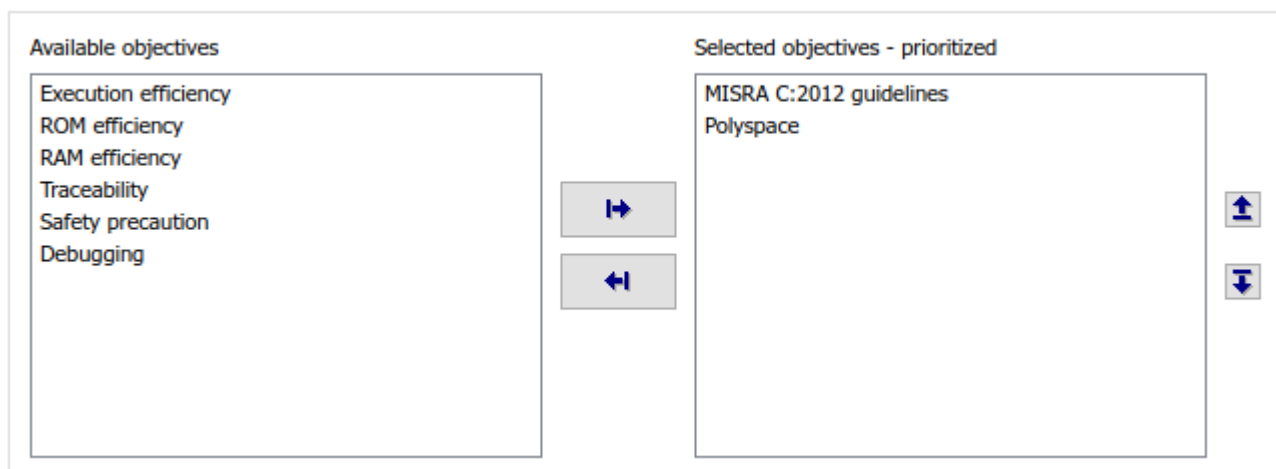


Run Code Generator Checks

Check your model by using the Code Generation Advisor. Configure code generation parameters to generate code more compliant with MISRA C and more compatible with Polyspace.

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

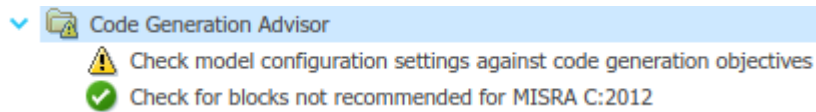
Code Generation Objectives (System target file: ert.tlc)



- 3 Click **Run Selected Checks**.

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

model, the check for incompatible blocks passes, but some configuration settings are incompatible with MISRA compliance and Polyspace checking.



- 4 Click the check that did not pass. Accept the parameter changes by selecting **Modify Parameters**.
- 5 Rerun the check by selecting **Run This Check**.

Run Model Advisor Checks

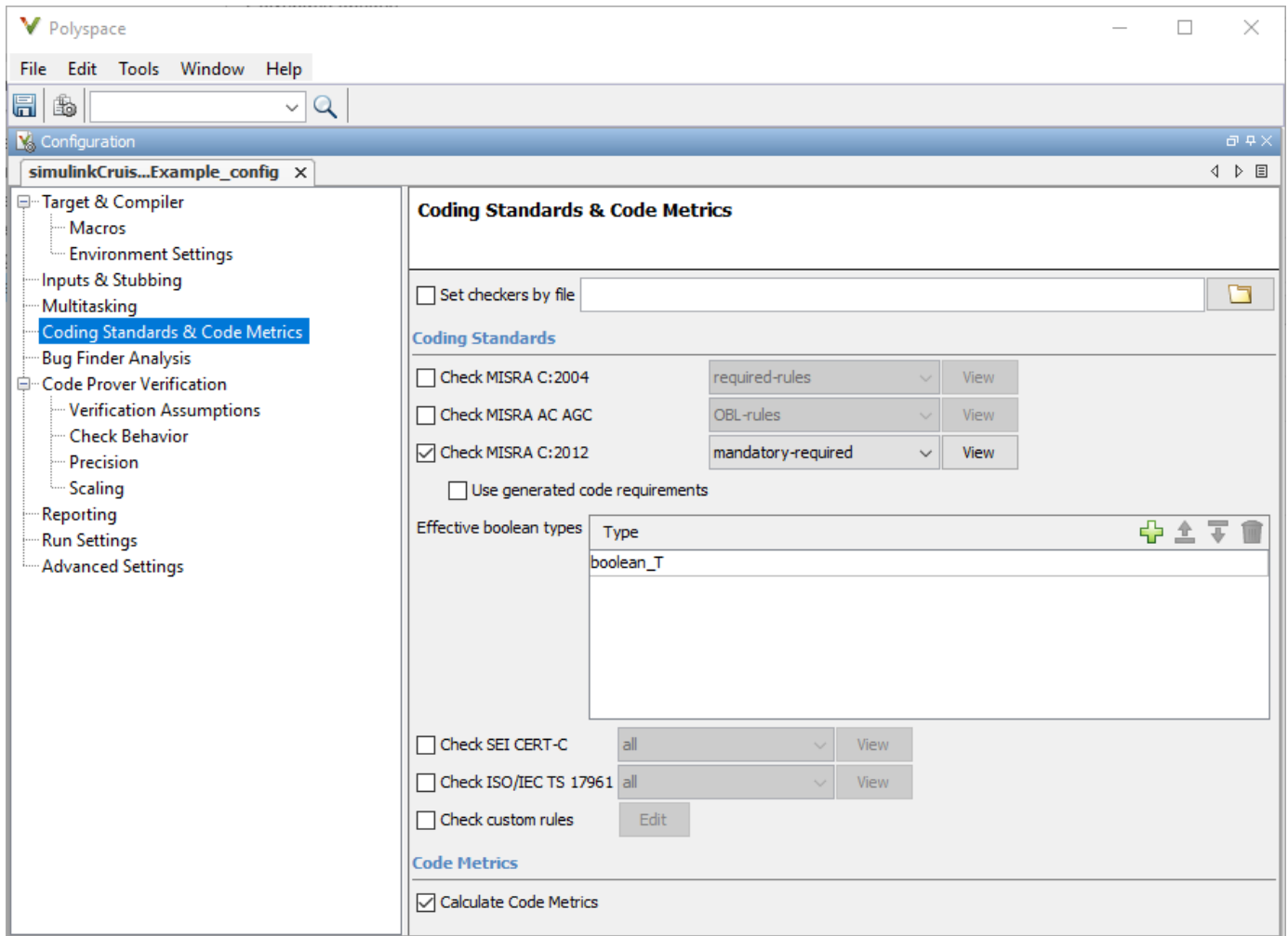
Before you generate code from your model, use the Model Advisor to check your model for MISRA C and Polyspace compliance. This example shows you how to use the Model Advisor to check your model before generating code.

- 1 At the bottom of the Code Generation Advisor window, select **Model Advisor**.
- 2 Under the **By Task** folder, select the **Modeling Standards for MISRA C:2012** advisor checks.
- 3 Click **Run Checks** and review the results.
- 4 If any of the tasks fail, make the suggested modifications and rerun the checks until the MISRA modeling guidelines pass.

Generate and Analyze Code

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

- 1 In the Simulink editor, right-click Compute target speed and select **C/C++ Code > Build This Subsystem**.
- 2 Use the default settings for the tunable parameters and select **Build**.
- 3 After the code is generated, in the Simulink Editor, right-click Compute target speed and select **Polyspace > Options**.
- 4 Click **Configure** to choose more advanced Polyspace analysis options in the Polyspace configuration window.



- 5 On the left pane, click **Coding Standards & Code Metrics**, then select **Calculate Code Metrics** to enable code metric calculations for your generated code.
- 6 Save and close the Polyspace configuration window.
- 7 From your model, right-click Compute target speed and select **Polyspace > Verify > Code Generated For Selected Subsystem**.

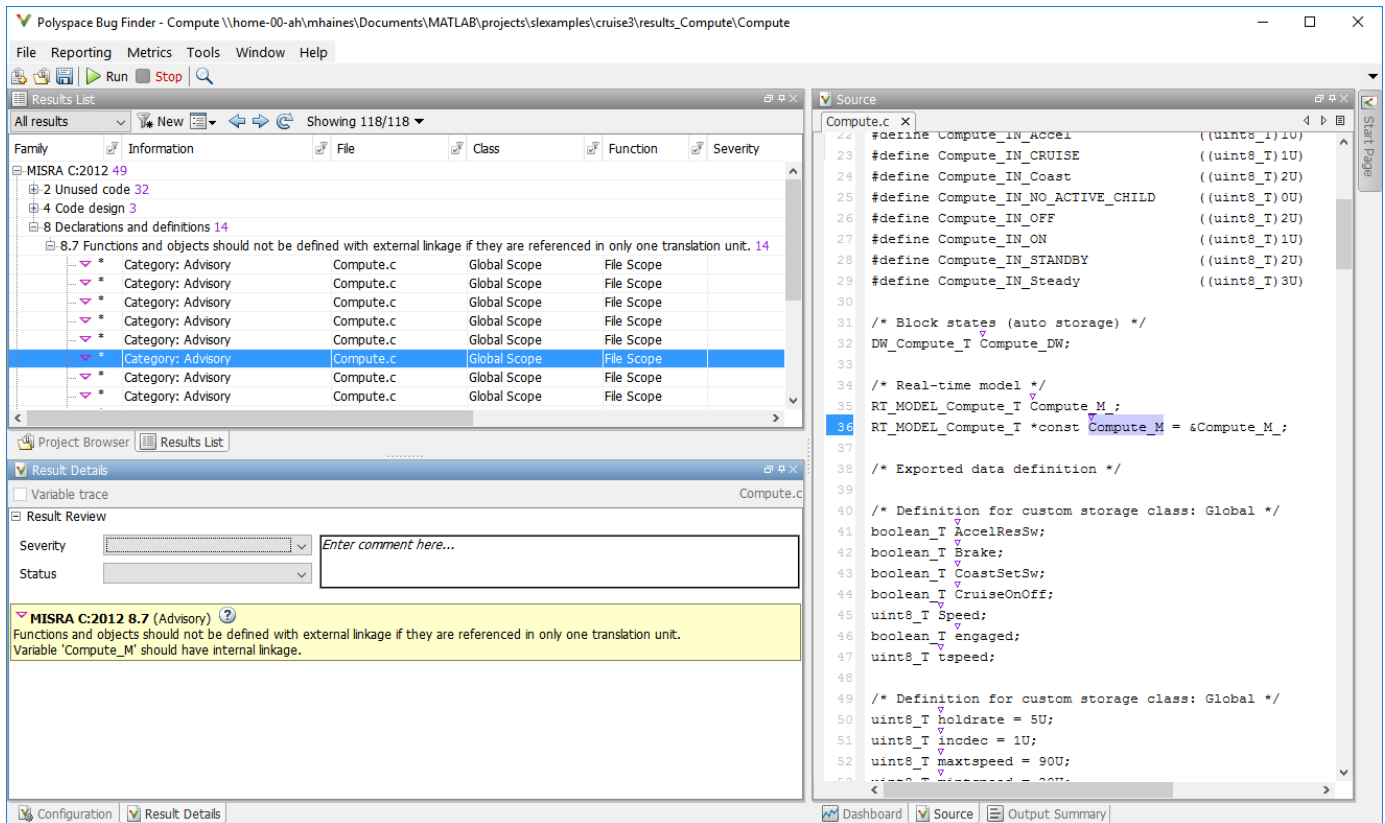
Polyspace Bug Finder analyzes the generated code for a subset of MISRA checks. You can see the progress of the analysis in the MATLAB Command Window. After the analysis finishes, the Polyspace environment opens.

Review Results

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

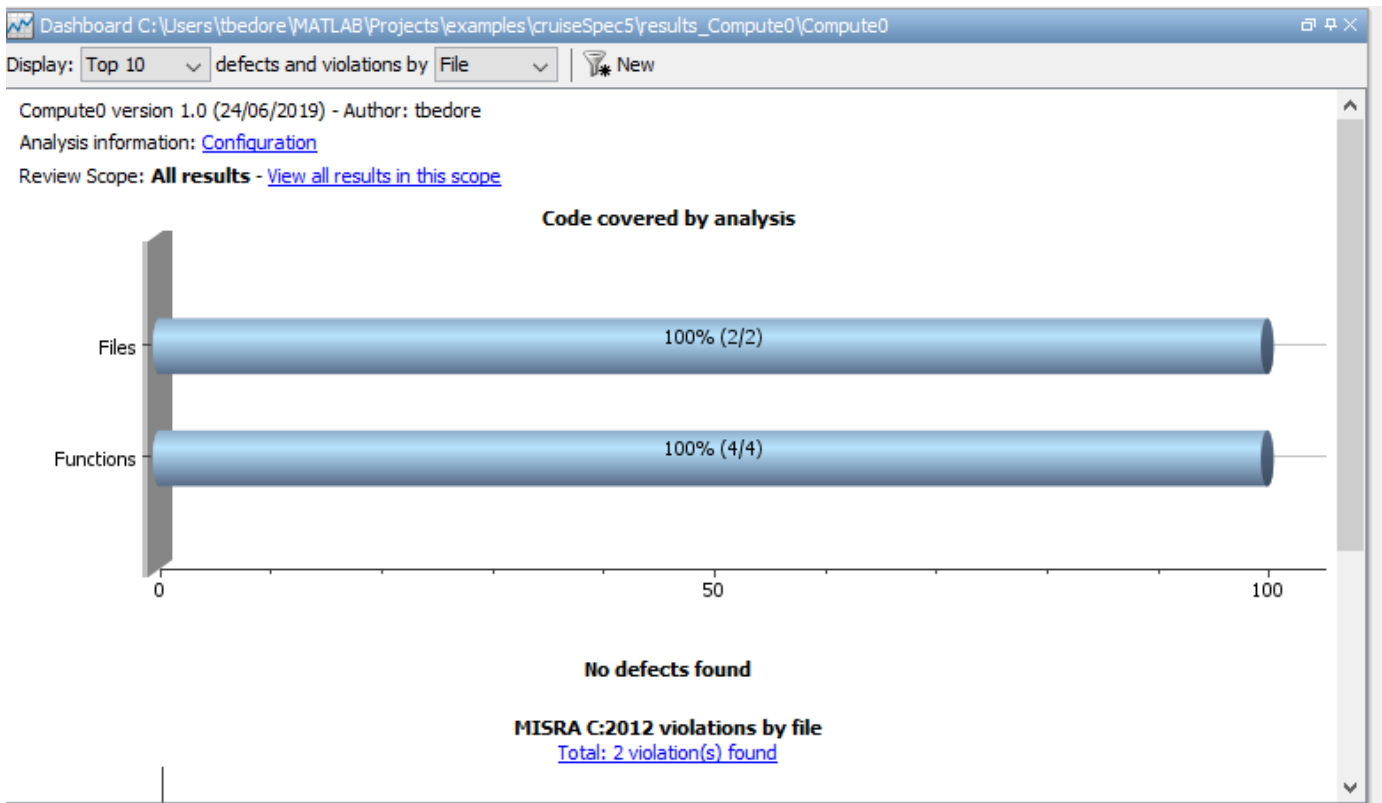
- 1 Expand the tree for rule 8.7 and click through the different results.

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



- 2 In your model, right-click Compute target speed and select **Polyspace > Options**.
- 3 Set the **Settings from** option to Project configuration to choose a subset of MISRA rules in the Polyspace configuration.
- 4 Click **Configure**.
- 5 In the Polyspace window, on the left pane, click **Coding Standards & Code Metrics**. Then select **Check MISRA C:2012** and, from the drop-down list, select **single-unit-rules**. Now Polyspace checks only the MISRA C:2012 rules that are applicable to a single unit.
- 6 Save and close the Polyspace configuration window.
- 7 Rerun the analysis with the new configuration.

The rules Polyspace showed previously were found because the model was analyzed by itself. When you limited the rules Polyspace checked to the single-unit subset, Polyspace found only two violations.



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

Generate Report

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

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

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

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

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

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

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

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

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

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

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

The screenshot shows the Test Manager interface with the following components:

- Toolbar:** Includes icons for New, Open, Save, Cut, Copy, Paste, Delete, Test Spec Report, Run, Run with Stepper, Stop, Parallel, Report, Visualize, Highlight in Model, Import, Export, Testing Dashboard, Preferences, and Help.
- Test Browser:** Shows a list of test results for 'Results: 2021-Feb-04 10:35:44'. The 'SIL Equivalence Test Case' is selected and expanded, showing sub-items like 'Equivalence Criteria Result', 'Verify Statements 1', 'Verify Statements 2', 'Current: Sim Output 1 (simulink)', and 'Current: Sim Output 2 (simulink)'. All sub-items have a green status icon.
- Results and Artifacts:** Shows the 'SIL Equivalence Test Case' selected. The 'COVERAGE RESULTS' section is expanded, displaying a table of analyzed models.
- Coverage Results Table:**

ANALYZED MODEL	REPORT	SIM MODE	COM...	DECISION	CONDITION	MCDC	EXECUTION	FUNCTION	FUNCTION...
<code>dlv_su32.c</code>		SIL	2	0%	--	--	0%	0%	--
<code>simulinkCruiseErrorAndStandardsExample</code>		ModelRe...	22	54%	44%	17%	70%	100%	33%
<code>simulinkCruiseErrorAndStandardsExample</code>		Normal	31	50%	41%	25%	--	--	--
- Property Window:** Shows details for the selected test case:

PROPERTY	VALUE
Name	SIL Equivalence Te...
Status	1
Start Time	02/04/2021 10:36:10
End Time	02/04/2021 10:38:36
Type	Equivalence Test

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

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

See Also

Related Examples

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

Checking Systems Interactively

Check Model Compliance by Using the Model Advisor

Model Advisor Overview

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

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

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

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

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

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

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

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

Model Advisor Checks Documentation

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

Product	Model Advisor Check Documentation
Simulink	"Simulink Checks"
Embedded Coder	"Embedded Coder Checks" (Embedded Coder)
AUTOSAR Blockset	"AUTOSAR Blockset Checks" (AUTOSAR Blockset)
Simulink Coder™	"Model Advisor Checks" (Simulink Coder)
HDL Coder™	"HDL Code Advisor Checks" (HDL Coder)
Simulink Code Inspector™	"Simulink Code Inspector Checks" (Simulink Code Inspector)
Simulink Check	<p>"Model Advisor Checks for DO-178C/DO-331 Industry Standards"</p> <p>"Model Advisor Checks for IEC 61508, IEC 62304, ISO 26262, ISO 25119, and EN 50128/EN 50657 Industry Standards"</p> <p>"Model Advisor Checks for DO-254 Standard Compliance" on page 3-53</p> <p>"Model Advisor Checks for High Integrity System Modeling Guidelines"</p> <p>"Model Advisor Checks for MAB Modeling Guidelines"</p> <p>"Model Advisor Checks for JMAAB Modeling Guidelines"</p> <p>"Model Advisor Checks for MISRA C:2012 Coding Standards"</p> <p>"Model Advisor Checks for CERT C, CWE, and ISO/IEC TS 17961 Secure Coding Standards"</p> <p>"Model Metrics"</p>
Simulink Design Verifier	"Simulink Design Verifier Checks" (Simulink Design Verifier)
Simulink PLC Coder™	"PLC Model Advisor Checks" (Simulink PLC Coder)
Requirements Toolbox	"Requirements Consistency Checks" (Requirements Toolbox)


Product	Model Advisor Check Documentation
Simscape™	Documentation is available only in the Model Advisor. To review the documentation for the check, in the Model Advisor, right-click on the check title and select What's This?
Simulink Control Design™	“Simulink Control Design Checks” (Simulink Control Design)
IEC Certification Kit	“IEC Certification Kit Checks” (IEC Certification Kit)
DO Qualification Kit	“DO Qualification Kit Checks” (DO Qualification Kit)


Run Model Advisor Checks and Review Results

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


- 1 To open the Model Advisor example model, at the MATLAB command line, enter :







```
openExample('sldemo_mdadv')
```
- 2 To open the Model Advisor, in the Simulink editor, click the **Modeling** tab and select **Model Advisor**. A System Selector dialog opens. Select the model or system that you want to review and click **OK**.
- 3 In the left pane of the Model Advisor, select the checks you want to run on your model:
 - a You can select the checks by using the **By Product** or **By Task** folders.
 - **Show By Product Folder** — Displays checks available for each product
 - **Show By Task Folder** — Displays checks related to specific tasks

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

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

- 4 Click on the folder that contains the checks and, on the toolstrip, select **Run Checks** to execute the analysis. To run a single check, right-click the check in the folder and select **Run This Check**.
- 5 View the results on the Model Advisor User Interface. This table shows the common check status results; notice that different icons are used depending on the parameter set for **Check result when issues are flagged** in the Model Advisor Configuration Editor (requires a Simulink Check license). For more information about this parameter, see “Specify Parameters for Check Customization” on page 7-6.

Check Result Status	Icon	Description
Passed		Model does not have any violations for the given check(s).

Check Result Status	Icon	Description
Failed		Check has identified severe violations.
Warning		Check has identified violations.
Justified		Check violations are justified.
Not Run		Check not selected for Model Advisor analysis.
Incomplete		Check analysis is incomplete or check execution has resulted in exceptions.

- 6 Fix the warnings or failures as desired. For more information, see “Address Model Check Results”.
- 7 Use the **Exclusions** tab to review checks that were marked for exclusion from the analysis.
- 8 View and save the report. For additional information, see “Save and View Model Advisor Check Reports” and “Generate Model Advisor Reports” on page 3-16.

See Also

Related Examples

- “Address Model Check Results”
- “Generate Model Advisor Reports” on page 3-16
- “Save and View Model Advisor Check Reports”
- “Find Model Advisor Check IDs”
- “Archive and View Results” on page 4-6
- “Exclude Blocks from the Model Advisor Check Analysis” on page 3-9
- “Check Model Compliance Using Edit-Time Checking” on page 3-6
- “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3

Check Model Compliance Using Edit-Time Checking

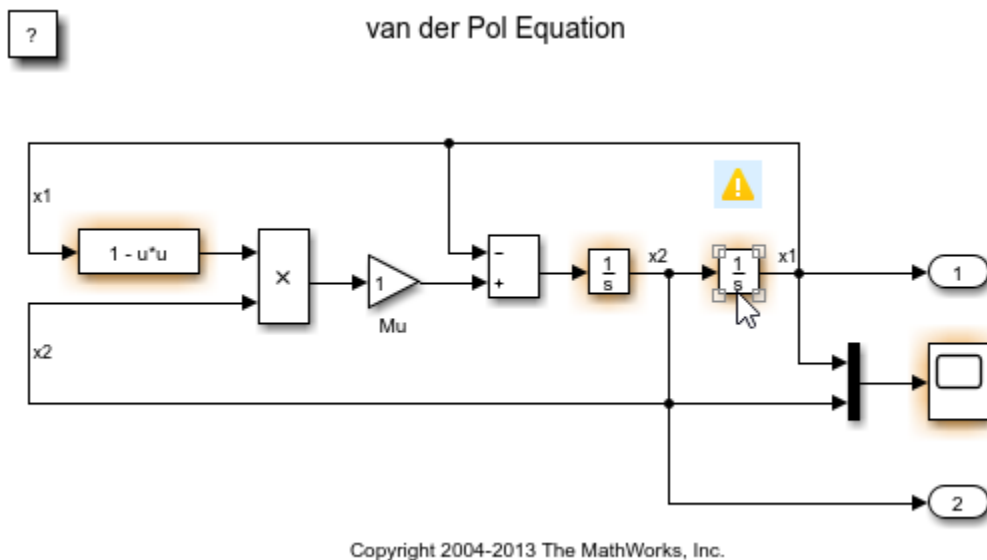
When you enable edit-time checks, the Model Advisor evaluates the model against a subset of Model Advisor checks. Highlighted blocks in the model editor window alert you to issues in your model. This enables you to identify modeling issues earlier in the model design process.

Configure Your Model to Use Edit-Time Checking

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

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

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



If you edit the model when edit-time checking is enabled, the Model Advisor checks out a Simulink Check license.

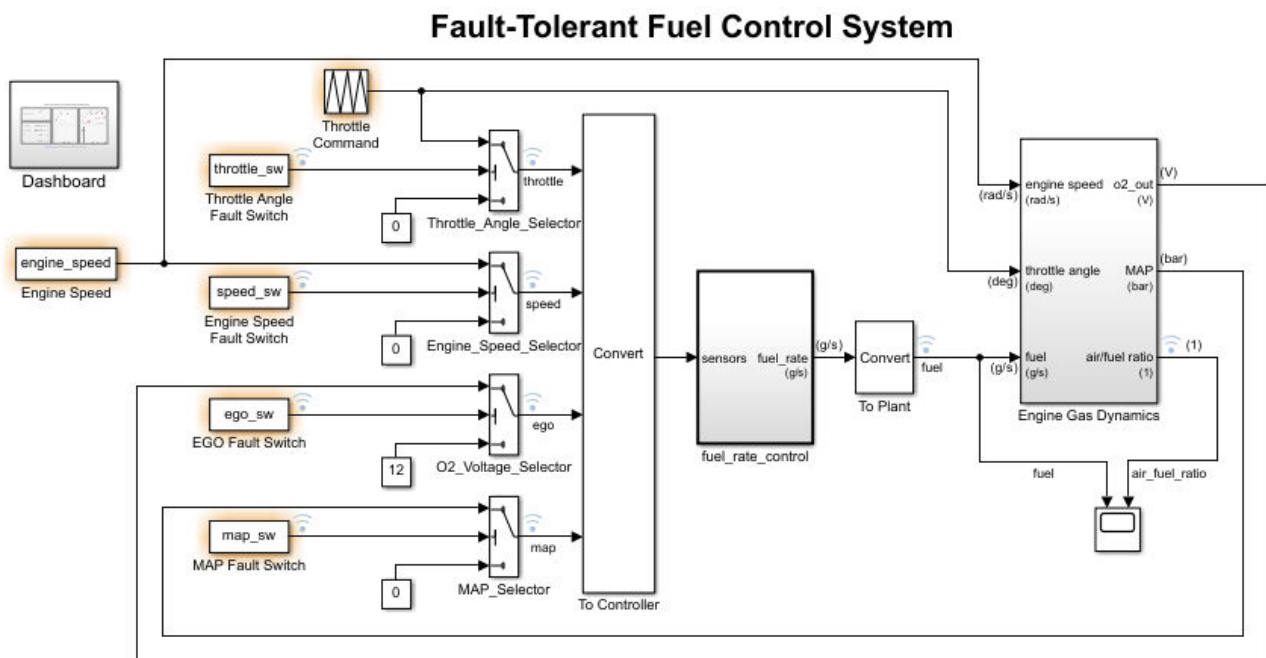
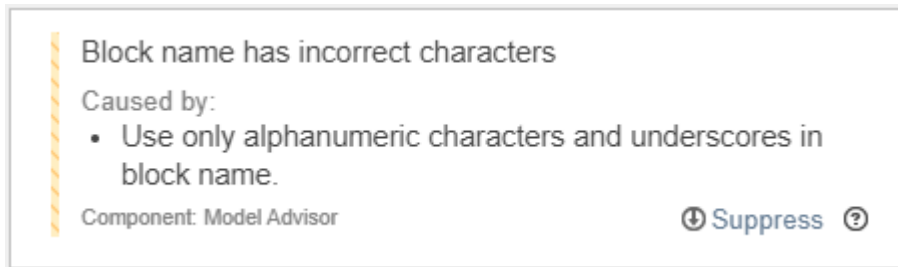
To review a check violation, click the error or warning icon above the highlighted block. A diagnostics window opens, which provides information about the modeling issue that violates the Model Advisor check. When a block violates multiple checks, you can use the diagnostics window to review issues.

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

- Review the cause and explore suggested options for fixing the issue, if any.
- Click the question mark to access detailed documentation about the violated Model Advisor check.
- Add a justification for the violation by clicking **Suppress**.

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

- 1 Open a model. For this example, at the command prompt, type: `sldemo_fuelsys`.
- 2 To enable the edit-time checking, in the **Modeling** tab, select **Model Advisor > Edit-Time Checks**. The Configuration Parameters dialog box opens and you select the check box for **Edit-Time Checks**.
- 3 The Model Advisor highlights several blocks. Place your cursor over the warning of the **Throttle Angle Fault Switch** block to discover the issue.



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

Copyright 1990-2017 The MathWorks, Inc.

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

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

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

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

- 1 In the Simulink editor, click the **Modeling** tab and select **Model Advisor > Customize Edit-Time Checks**.
- 2 The Model Advisor Configuration Editor opens. The file name for the configuration that is currently being used by the Model Advisor is displayed at the top of the window. Verify that you are evaluating the correct configuration file. To open a different configuration file, click **Open** and browse to the file you would like to review.
- 3 In the Model Advisor Configuration Editor, on the Model Advisor tab, select the **Edit time supported checks** option. The filtered list identifies the Model Advisor checks that are supported for edit-time checking.

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

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

See Also

Related Examples

- “Address Model Check Results”
- “Generate Model Advisor Reports” on page 3-16
- “Save and View Model Advisor Check Reports”
- “Find Model Advisor Check IDs”
- “Archive and View Results” on page 4-6
- “Exclude Blocks from the Model Advisor Check Analysis” on page 3-9

Exclude Blocks from the Model Advisor Check Analysis

Model Advisor Exclusion Overview

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

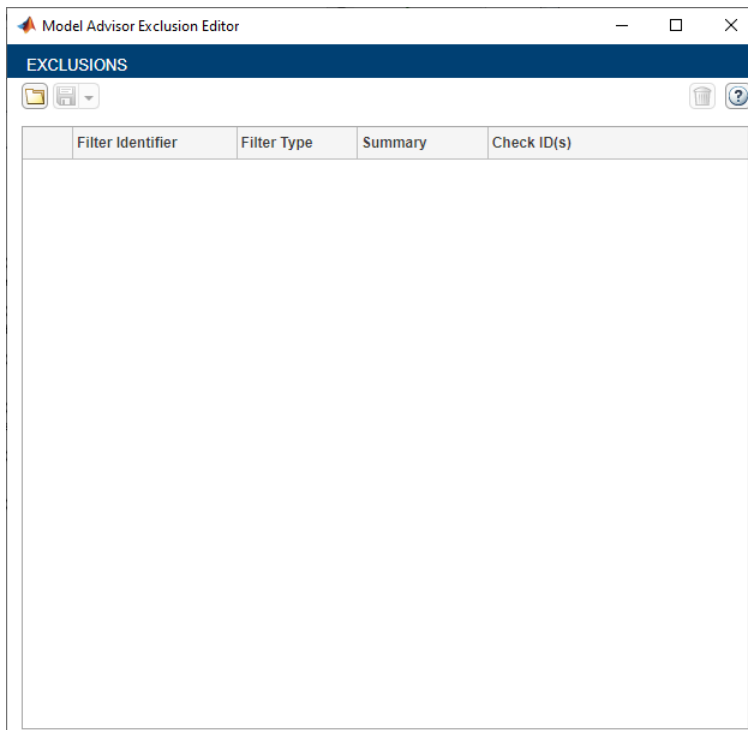
- Simulink blocks
- Stateflow charts

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

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

The Model Advisor Exclusion Editor dialog box includes the following information for each exclusion:

- Filter Identifier
- Filter Type
- Summary
- Check ID(s)



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

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

Filter Types

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

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

Create Model Advisor Exclusions


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

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

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

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


Save Model Advisor Exclusions in a Model File

To save Model Advisor exclusions to the model .slx file, in the Model Advisor Exclusion Editor dialog box, click on the save icon(). When you open the model .slx file, the model contains the exclusions.

Save Model Advisor Exclusions in Exclusion File

A Model Advisor exclusion file specifies the collection of blocks to exclude from specified checks in an exclusion file. You can create exclusions and save them in an exclusion file.

To save Model Advisor exclusions to the exclusion .xml file:

- 1 Open the Model Advisor Exclusion Editor.
- 2 click the drop-down next to the save icon()
- 3 Select **Save as** and enter the desired name for the exclusion file in the **File name** field.
- 4 Click **Save**.


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

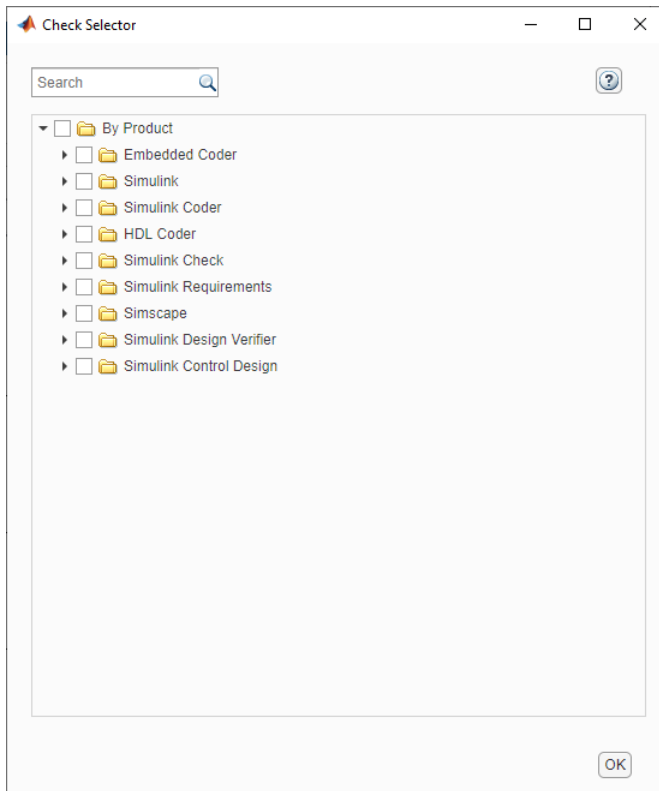
If you create an exclusion file and save your model, you attach the exclusion file to your model. Each time that you open the model, the blocks and checks specified in the exclusion file are excluded from the analysis.

Check Selector

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

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

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



Review Model Advisor Exclusions

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

- Right-click in the model window or right-click a block and select **Model Advisor > Open Model Advisor Exclusion Editor**. The Model Advisor Exclusion Editor dialog box lists the exclusions for your model.
- After the Model Advisor analysis, you can view exclusion information for individual checks in the report file.
- After you run the checks, in the left pane of the Model Advisor window, the checks that contain exclusion rules are highlighted in orange. The Model Advisor results include additional information about the exclusion.

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


Manage Exclusions

Load an Exclusion File

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

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

Remove an Exclusion

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

Add Summary to an Exclusion

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

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

Compatibility Considerations after R2020b

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

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

Exclusion File Saved Inside the Model

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

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

Exclusion File Is Saved in a Separate Exclusion File

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

The Exclusion Editor reports that a change has taken place.

Programmatically Change Model Advisor Exclusions

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

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

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

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

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

See Also

Related Examples

- “Exclude Blocks From Custom Checks” on page 6-57
- “Run Model Advisor Checks and Review Results” on page 3-4
- “Address Model Check Results”
- “Generate Model Advisor Reports” on page 3-16
- “Save and View Model Advisor Check Reports”
- “Find Model Advisor Check IDs”
- “Archive and View Results” on page 4-6
- “Limit Model Checks by Excluding Gain and Output Blocks”
- “Exclude Blocks from Edit Time Checking”

More About

- “Check Your Model Using the Model Advisor”

Generate Model Advisor Reports

By default, when the Model Advisor runs checks, it generates an HTML report of check results in the `s\prj\modeladvisor\model_name` folder. Additionally, if you have a MATLAB Report Generator license, on Windows® platforms, you can generate Model Advisor reports in Adobe® PDF, and Microsoft Word .docx formats.

The beginning of the Model Advisor reports contain the:

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

Generate Results Report After Executing Model Advisor Checks

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

- 1 In the left pane of the Model Advisor, select the checks you want to run. Click on the folder that contains the checks and, from the toolstrip, click **Run Checks**.
- 2 When complete, from the toolstrip, click **Report**.
- 3 In the **Save Report** dialog box:
 - Enter the path to the folder where you want to generate the report.
 - Provide a file name.
 - Click **Save** to generate a report in HTML format.
- 4 Use can change the **File format** to PDF, or WORD using the drop-down options of the **Report** button.
- 5 The Model Advisor generates the report and saves it to the designated location.

Modify Template for Model Advisor Check Results Report

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

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

Template Field	In generated report, displays
ModelName	Model name
SimulinkVersion	Simulink version
SystemName	System name

Template Field	In generated report, displays
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
JustifiedCount	Number of checks that are justified
IncompleteCount	Number of checks that fail to run to completion
FailCount	Number of checks that fail
WarningCount	Number of checks that cause a warning
NrunCount	Number of checks that did not run
TotalCount	Total number of checks
CheckResults	Results for each check

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

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

My Custom Header

Model Advisor Report – Model name

Simulink version: Simulink version

Model version: Model version

System: System name

Current run: Timestamp

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

Run Summary

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

Results of all checks

- 4 From the Model Advisor toolstrip, click **Report** drop-down, and select **Template File**.
- 5 In the **Select Template for Report** dialog box, enter the path to the folder where your custom template is placed. In this case, the path is `C:/work/ma_format/mytemplate.dotx`.
- 6 Click **OK**.
- 7 From the toolstrip, click **Report** drop-down, and select **PDF**. The Model Advisor generates the report in PDF format with the custom header.

My Custom Header

Model Advisor Report – sldemo_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

`ModelAdvisor.summaryReport` | `viewReport`

Related Examples

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

Transform Model to Variant System

You can use the Model Transformer tool to improve model componentization by replacing qualifying modeling patterns with Variant Source and Variant Subsystem, Variant Model, Variant Assembly 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:

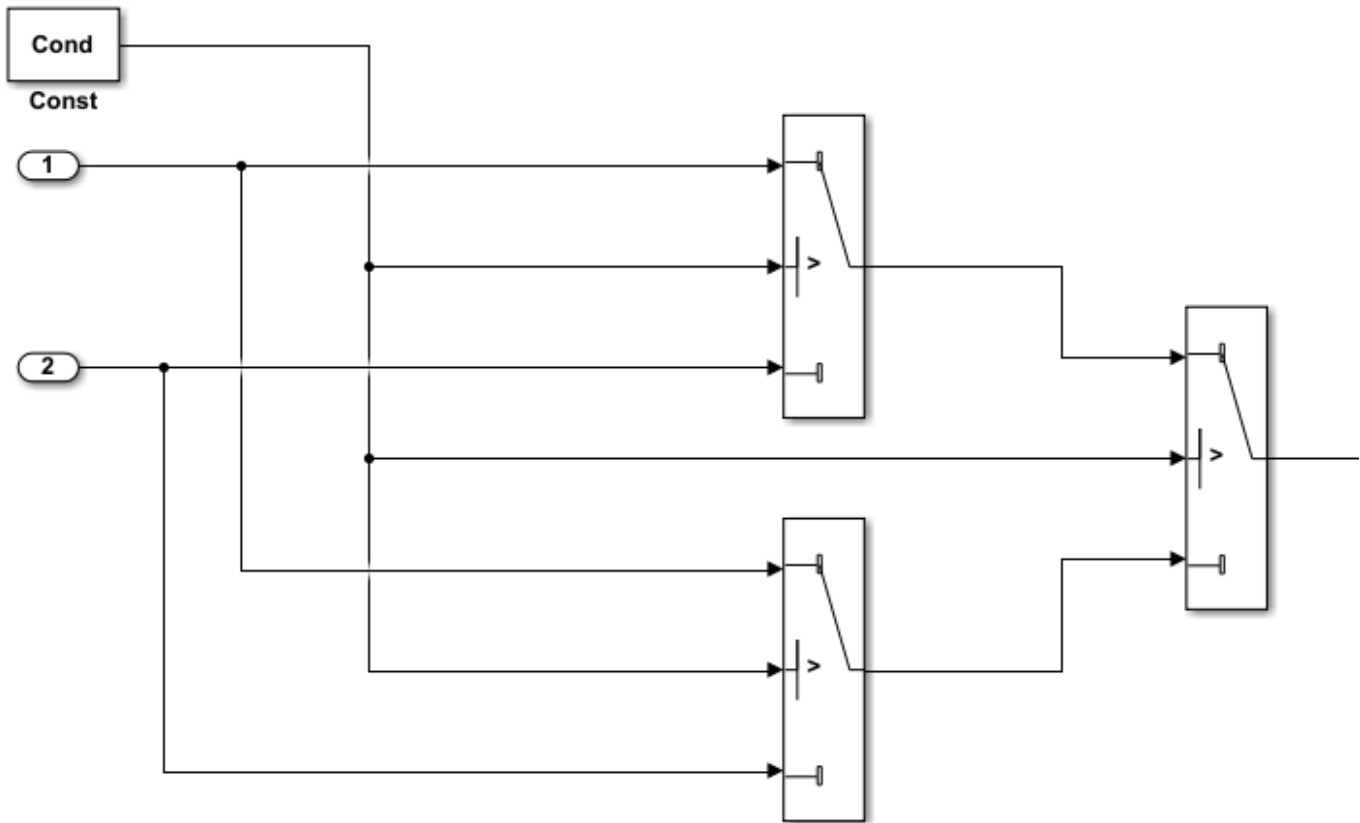
- If an If block connects to one or more If Action Subsystems and each one has one output, replace this modeling pattern with a subsystem and a Variant Source block.
- If an If block connects to an If Action Subsystem that does not have an output or has two or more outputs, replace this modeling pattern 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 “Types of Operators in Variant Blocks for Different Activation Times” are supported except for bitwise operations.

Transform Model to Variant System Using Model Transformer

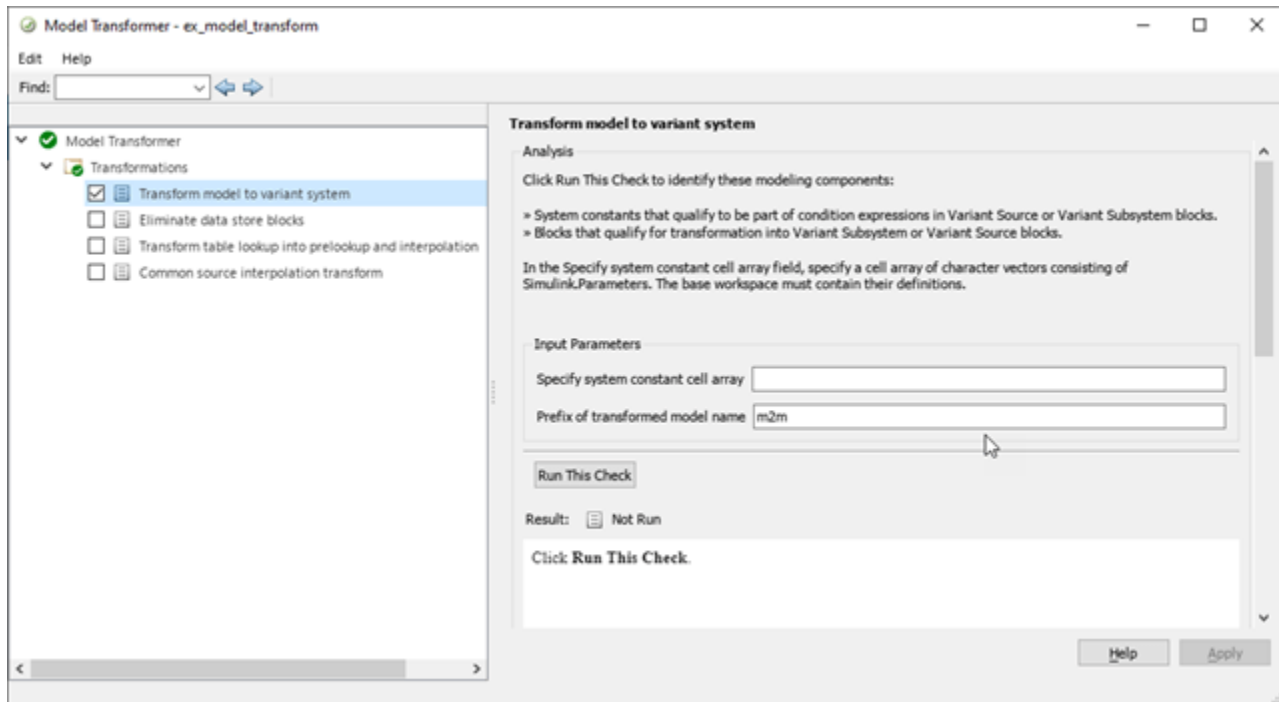
This example shows how to use the Model Transformer to transform a model into a variant system. The example uses the model `ex_model_transformer`. This model has three Switch blocks. The control input to these Switch blocks is the `Simulink.Parameter` `cond`. The Model Transformer dialog box and this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



- 1 Open the model `ex_model_transformer`.
- 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

- In the **Apps** tab, Open the Model Transformer by selecting **Model Transformer**. Or, in the Command Window, type: `mdltransformer('ex_model_transformer')`;
- Select the check "Replace Modeling Patterns with Variant Blocks".



- 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.
- 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`.
- 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.
- If you do not want one of the transformations to occur, you can clear the check box next to it.
- 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_ex_model_transformer`.
- In the original model `ex_model_transformer`, right-click one of the Switch blocks. In the menu, select **Model Transformer > Traceability to Transformed Block**. In the transformed model `gen0_ex_model_transformer`, the corresponding Variant Source block is highlighted.
- In the transformed model `gen0_ex_model_transformer`, right-click one of the Switch blocks. In the menu, select **Model Transformer > Traceability to Original Block**. In the original model `ex_model_transformer`, the corresponding Switch block is highlighted.

Model Transformation Limitations

The Model Transformer tool has these limitation:

- In order to run the Model Transformer on a model, you must be able to simulate the model.
- If an If Action Subsystem block drives a Merge block, and the Merge block has another inport that is either unconnected or driven by another conditional subsystem, the Model Transformer does

not add a Variant Source block. This modeling pattern produces a warning and an excluded candidate message.

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

See Also

Related Examples

- “Variant Systems”

Improve Code Efficiency by Merging Multiple Interpolation Using Prelookup Blocks

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

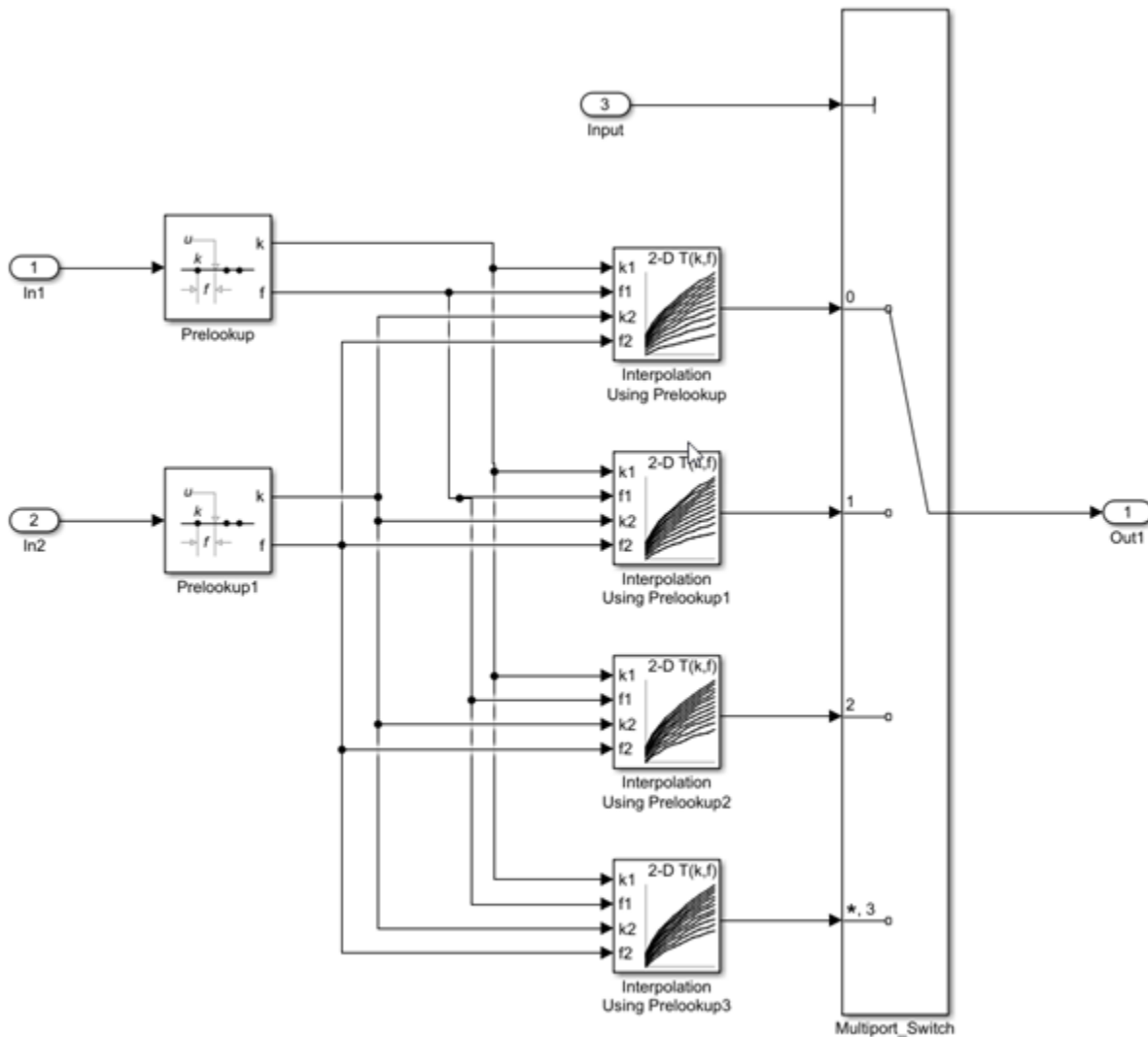
The Model Transformer works if the properties of Interpolation Using Prelookup blocks are same except for **Table data**. Reducing the number of Interpolation Using Prelookup blocks in a model reduces the number of variable assignments in the code, which improves the efficiency of the generated code.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

The Model Transformer can replace multiple Interpolation Using Prelookup that:

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

Merge Interpolation Using Prelookup Blocks Using the Model Transformer App

This example shows how to use the Model Transformer to identify redundant Interpolation Using Prelookup blocks, and then refactor the model.

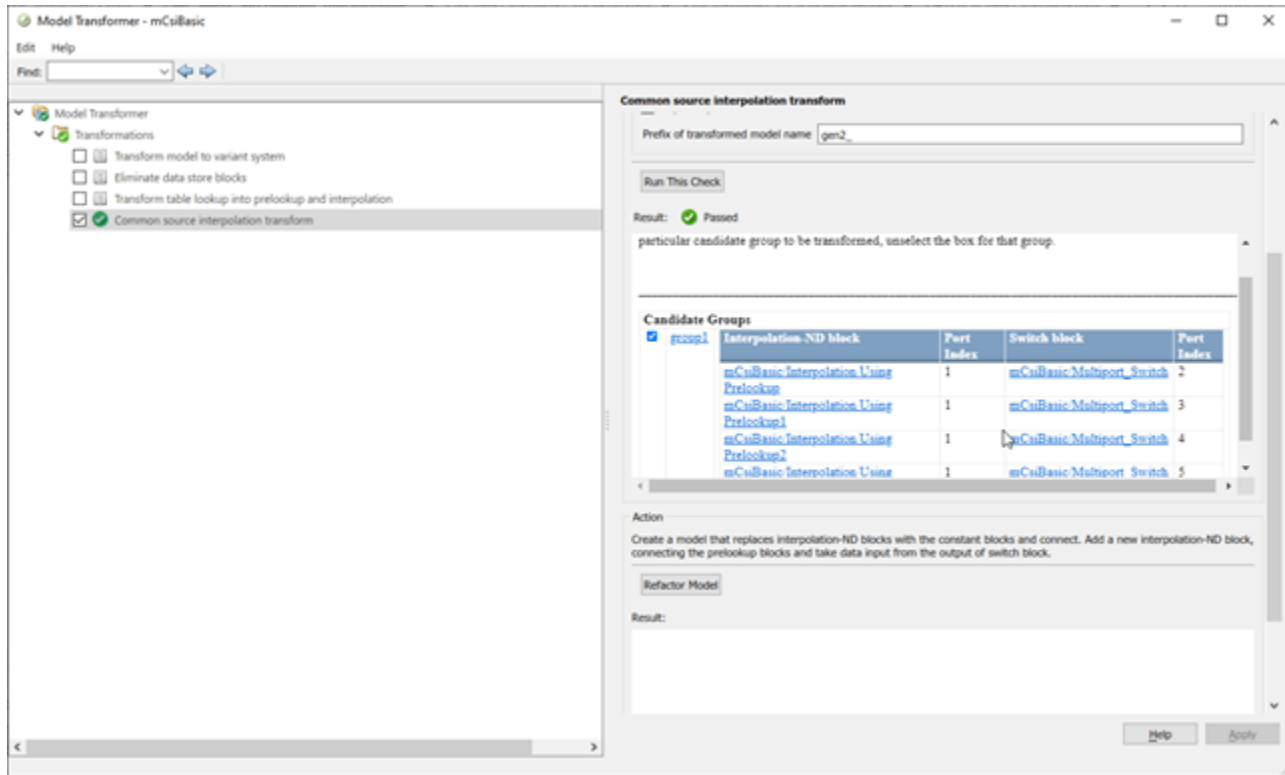


The model `ex_interpolation_optimize` uses Prelookup blocks to input signals to several Interpolation Using Prelookup blocks. The output of these Interpolation blocks are connected to a Multiport Switch block.

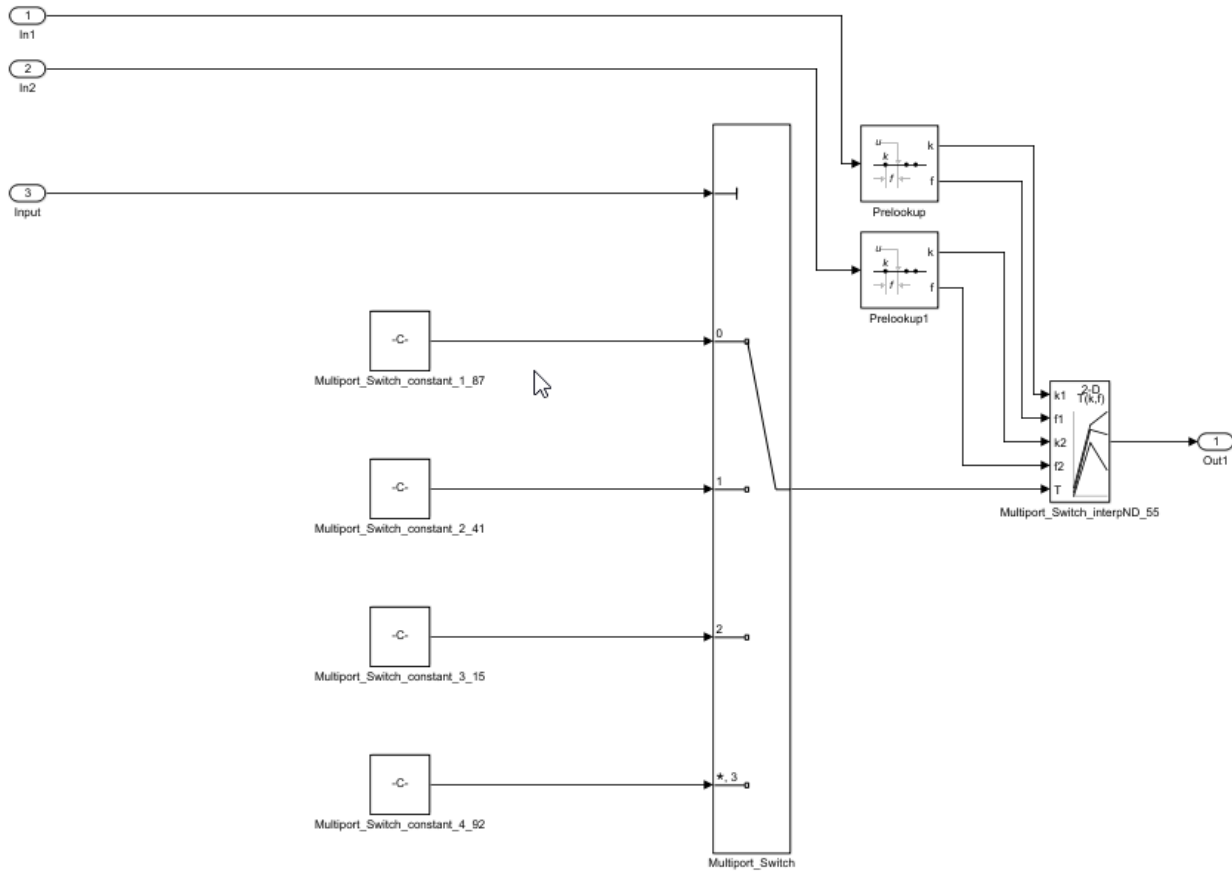
In this example, you identify Interpolation Using Prelookup blocks that qualify for transformation and replace them with a single Interpolation Using Prelookup block and Constant blocks connected to Multiport Switch block.

- 1 Save the model to your working folder.
- 2 On the **Apps** tab, click **Model Transformer**.
- 3 In the **Transformations** folder, select the “Replace Interpolation Using Prelookup Blocks” check.
- 4 Select the **Skip Interpolation-ND blocks in libraries from this transformation** option to avoid replacing Interpolation Using Prelookup blocks that are linked to a library.
- 5 In the **Prefix of transformed model** field, specify a prefix for the refactored model.
- 6 Click **Run This Check**. The top **Result** table contains hyperlinks to the Interpolation Using Prelookup blocks and the corresponding Multiport Switch block port indices.

- 7 Clear the check boxes under **Candidate Groups** for the groups that you do not want to transform.
- 8 Click **Refactor Model**. The **Result** table contains a hyperlink to the new model. The table contains hyperlinks to the shared Interpolation Using Prelookup blocks and corresponding Multiport Switch block ports. The tool also creates an `m2m_ex_interpolation_optimize` folder that contains the new `gen_ex_interpolation_optimize.slx` model.



The two Prelookup blocks Prelookup and Prelookup1 in the `gen_ex_interpolation_optimize.slx` model connect to the single Interpolation Using Prelookup block and the Constant blocks are connected to the Multiport Switch block port to give the **Table data** as input.



Merge Interpolation Using Prelookup Blocks Programmatically

To use the Model Transformer programmatically, use:

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

- 1 Save the model `ex_interpolation_optimize` in the current working directory.
- 2 To identify candidates qualified for transformation, use the function `Simulink.ModelTransform.CommonSourceInterpolation.identifyCandidates` to create the object `transformResults`.

```
transformResults = Simulink.ModelTransform.CommonSourceInterpolation.identifyCandidates('ex_interpolation_optimize')
```

```
transformResults =
```

```
Results with properties:
```

```
Candidates: [1x1 struct]
```

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

```
transformResults.Candidates =
    struct with fields:
        InterpolationPorts: [4x1 struct]
        SwitchPorts: [4x1 struct]
```

3 View the `InterpolationPorts` field.

```
transformResults.Candidates.InterpolationPorts =

    4x1 struct array with fields:
        Block
        Port
```

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

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

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

```
ans =
    4x2 table
           Block                               Port
    _____
    {'ex_interpolation_optimize/Interpolation_Using Prelookup' } 0
    {'ex_interpolation_optimize/Interpolation_Using Prelookup1'} 0
    {'ex_interpolation_optimize/Interpolation_Using Prelookup2'} 0
    {'ex_interpolation_optimize/Interpolation_Using Prelookup3'} 0
```

```
ans =
    4x2 table
           Block                               Port
    _____
    {'ex_interpolation_optimize/Multiport_Switch'} 1
    {'ex_interpolation_optimize/Multiport_Switch'} 2
    {'ex_interpolation_optimize/Multiport_Switch'} 3
    {'ex_interpolation_optimize/Multiport_Switch'} 4
```

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

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

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

refactorResults =

    RefactorResults with properties:
        ModelName: 'ex_interpolation_optimize'
        ModelDirectory: ''
        TraceabilityInfo: [4x1 containers.Map]
```

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

Conditions and Limitations

The Model Transformer cannot replace Interpolation Using Prelookup blocks if:

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

See Also

Related Examples

- “Refactor Models”
- “Using the Prelookup and Interpolation Blocks”

Enable Component Reuse by Using Clone Detection

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

Exact Clones and Similar Clones

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

Exact clones and similar clones can have these differences:

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

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

Specify Where to Detect Clones

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

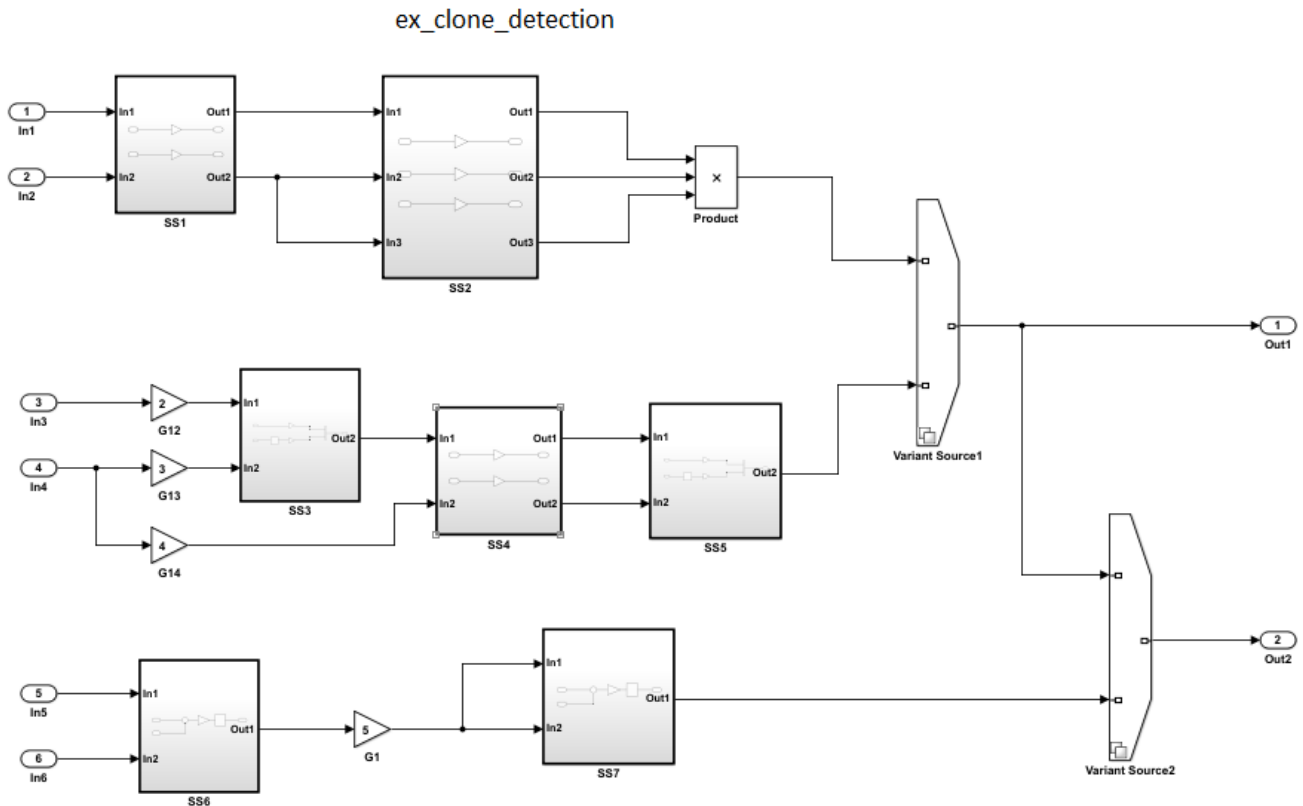
- Subsystem clones: Identifies clones only in a subsystems.
- Clones across model: Identifies clones across the model.

Identify Exact and Similar Clones

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

- 1 Open the model `ex_detect_clones`.

```
openExample('ex_detect_clones')
```



Copyright 2017 The MathWorks Inc.

- 2 Save the model to your working folder. A model must be open to access the app.
- 3 On the **Apps** tab, click **Clone Detector**. Alternatively, on the MATLAB command line enter:


```
clonedetection("ex_detect_clones")
```
- 4 The app opens the Clone Detector tab. This example takes you through each section.



Set Up panes for Clone Detection

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

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

Set the Parameters for Clone Detection

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

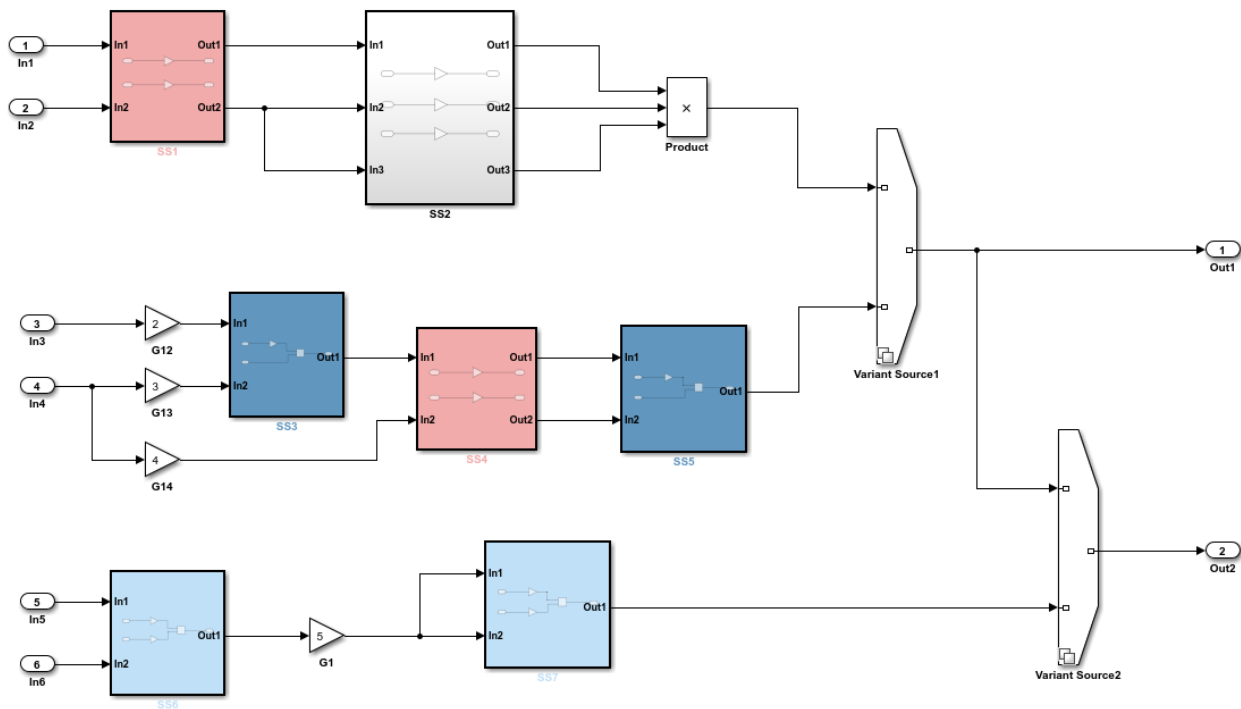
- Select **Ignore differences in > Signal Names** to identify and classify clones when the signal names are different.
- Select **Ignore differences in > Block Properties** to identify and classify clones when the block properties are different. For more information about block properties, see “Specify Block Properties”.
- Click **Replace Exact Clones With Subsystem References** to find and replace exact clones with subsystem reference blocks.
- Click **Exclude Components** to access the **Exclude model references**, **Exclude library Links**, and **Exclude inactive and commented out regions** options. Enabling the **Exclude inactive and commented out regions** option identifies variable number clones because of Variant Source block in the model. For more information, see “Exclude Components from Clone Detection”. Enabling the **Exclude model references** and **Exclude library Links** options will lead to identification of fewer clones, depending on the model.
- Click **Match Patterns with Libraries** and select an external library to look for clones. For more information, see “Identify and Replace Clones in Model Libraries” on page 3-34.
- The **Maximum number of unmatched block parameters** is 50 by default. This represents the number of parameters that can vary among subsystems and still be classified as similar clones. You may reduce this number to identify and classify fewer similar clones. Setting the value to zero, will identify only exact clones.
- Click **Detect Clones Across Model** to enable detect clones anywhere across the model. You can choose the values of **Minimum Region Size** and **Minimum Clone Group Size** to detect the clones with these matching blocks. The default size is set to 2.

Identify Subsystem Clones in the Model

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



- 2 Click **Find Clones** to identify clones.
- 3 The color of the subsystems changes to reflect the similar and exact clones identified. The red highlighting represents exact clones and the different shades of blue highlighting represent similar clones.



Copyright 2017 The MathWorks Inc.

Clone Detector app creates a backup folder in the working directory. The backup folder name has the prefix `m2m_<model name>`. It saves the clones data in a MAT-file. You can also find the backup of the original model in this folder after refactoring the model to replace clones with links to library blocks.

Analyze the Clone Detection Results

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

- 1 In the **Clone Detection Results and Actions** panel, on the **Clone Results** tab, a list of clone groups are displayed.
- 2 Click the > symbol next to Exact Clone Group 1 to see all of the subsystems that are exact clones, the number of blocks, and the block differences. Repeat the same for Similar Clone Group 1 and Similar Clone Group 2.
- 3 In the **Clone Detection Results and Actions** pane, click the **Logs** tab. Click the hyperlink on the **Logs** pane.

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

- 4 Click the **Model Hierarchy** tab. Click the hyperlinks to highlight the particular subsystems in the model. To go back to highlighting all clones, on the **Clone Results** tab, click the **Highlight all clones**.
- 5 On the **Clone Results** tab, expand Similar Clone Group 1 and click the **View Parameter Difference** hyperlink.

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

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

Category	Potential reuse percentage	Percentage of Total
Overall	<div style="width: 22.5806%; background-color: black;"></div>	22.5806
Exact	<div style="width: 6.4516%; background-color: red;"></div>	6.4516
Similar	<div style="width: 16.129%; background-color: blue;"></div>	16.129

Replace Clones

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

2 Click **Replace Clones**.

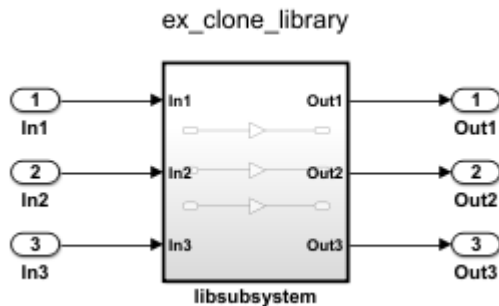
The model is refactored and the clones are replaced with links to the newLibraryFile library file in your working directory.

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

Identify and Replace Clones in Model Libraries

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

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



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

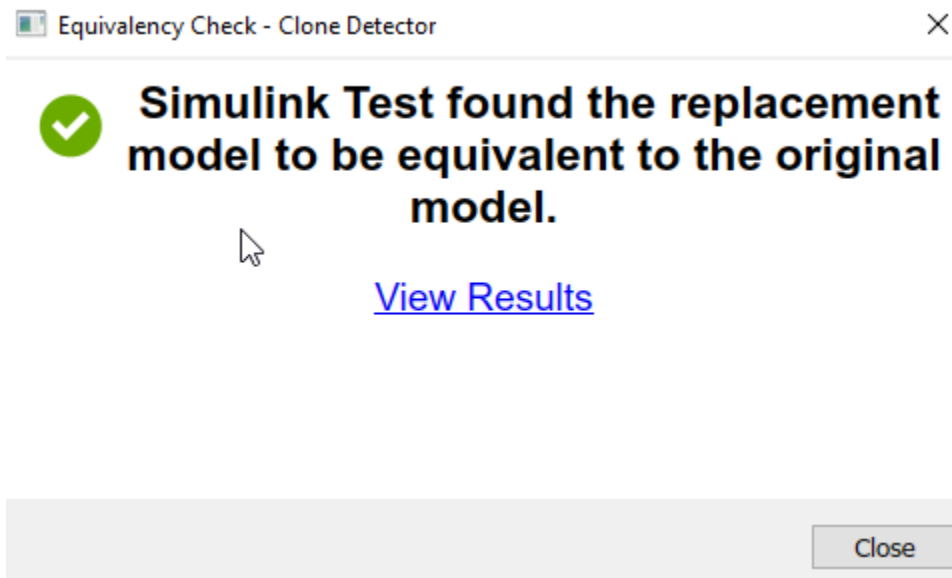
Note Identifying and refactoring clones using external libraries must be done separately in the model. During model refactoring only exact clones within the libraries will be replaced with library links.

3 Click **Replace Clones**.

The model is refactored with the exact clones replaced.

Check the Equivalency of the Model

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



See Also

Related Examples

- “Custom Libraries”
- “Generate Reusable Code from Library Subsystems Shared Across Models” (Simulink Coder)
- Clone Detector
- “Replace Exact Clones with Subsystem Reference” on page 3-79

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.

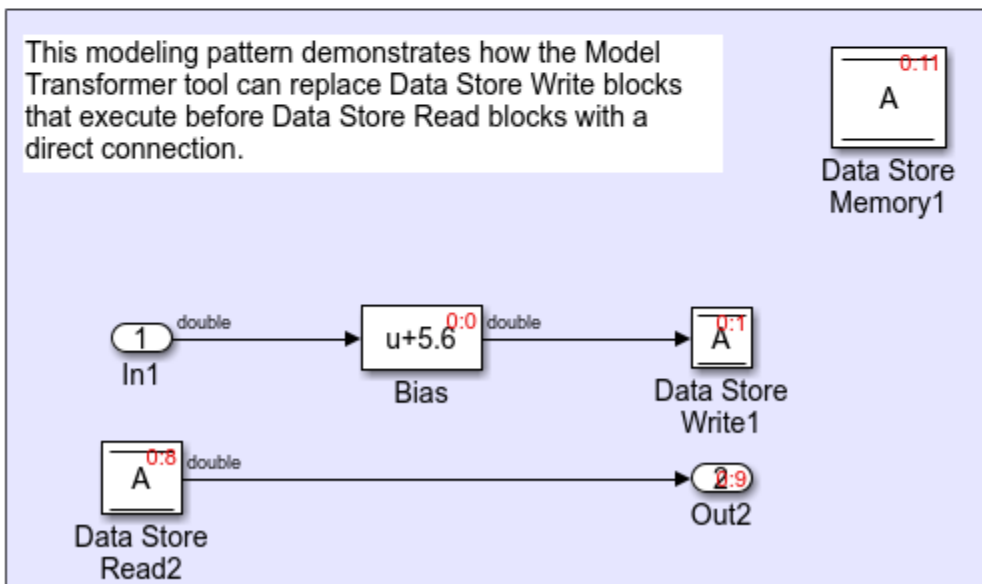
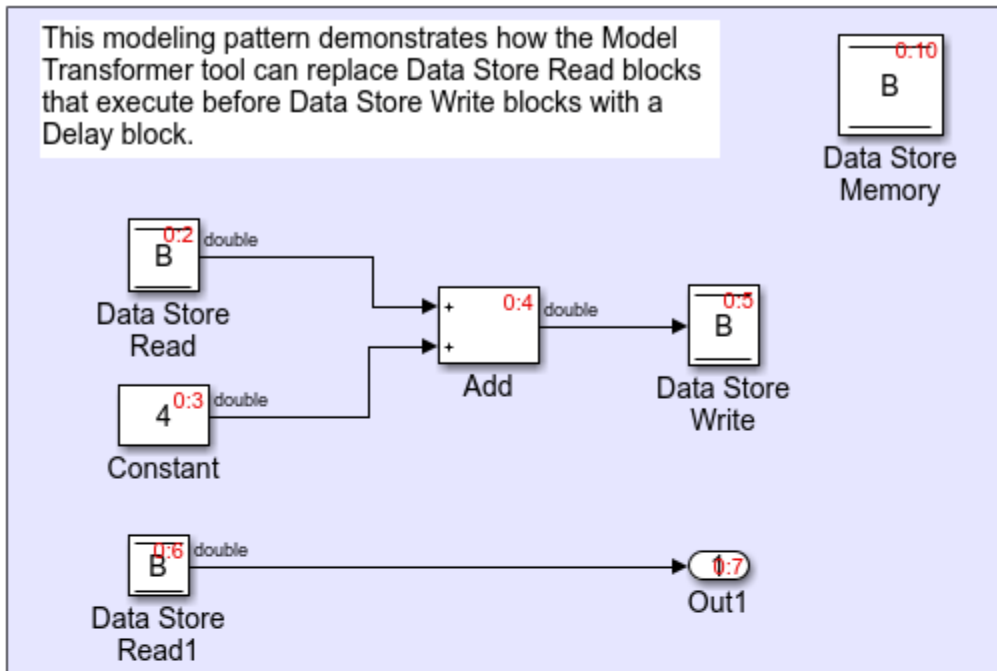
The Model Transformer can replace these data stores:

- 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.

Improve Model Readability by Eliminating Local Data Store Blocks Using Model Transformer

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

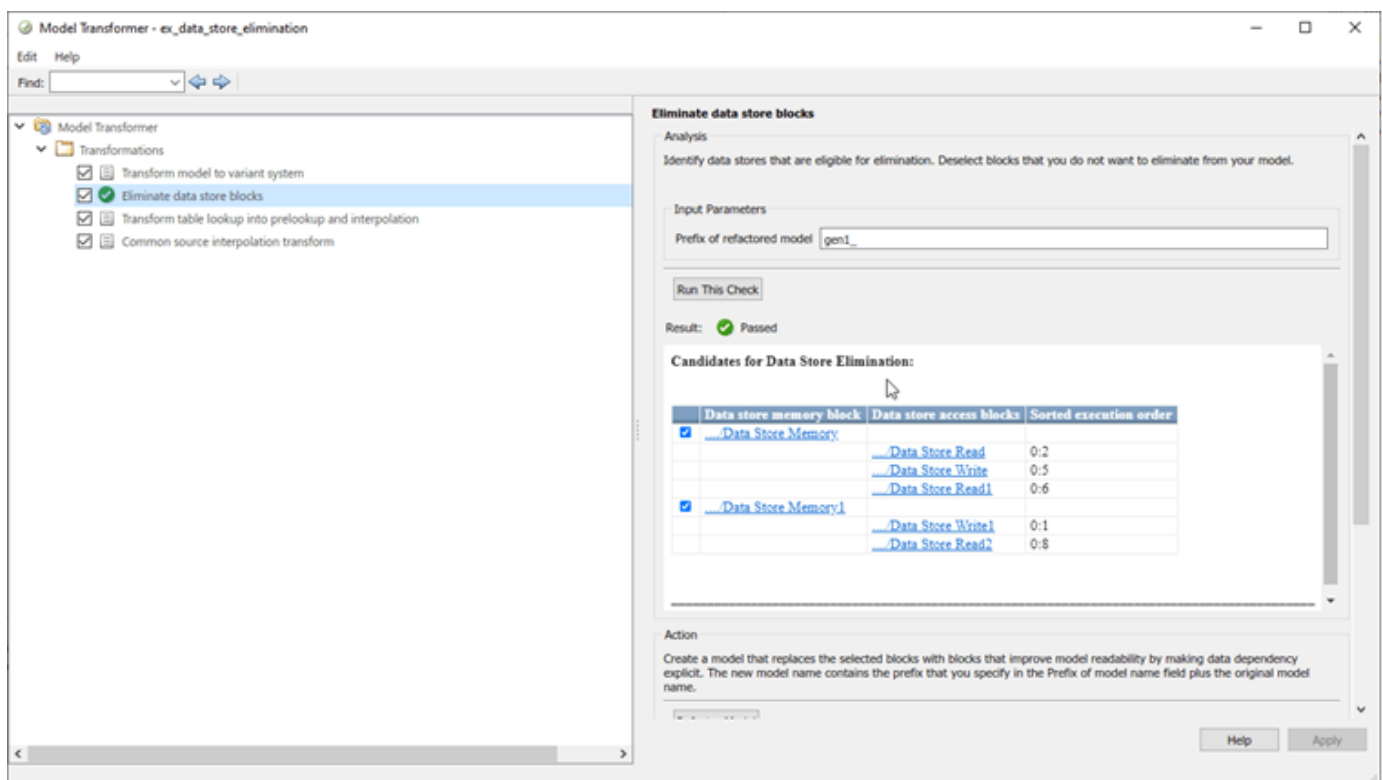


Copyright 2017 The MathWorks, Inc.

Replace Data Store Blocks

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

- 1 Open the model `data_store_elimination`.
- 2 Save the model to your working folder.
- 3 On the **Apps** tab, click **Model Transformer**.
- 4 In the **Transformations** folder, select the **Eliminate data store blocks** check.
- 5 In the **Prefix of refactored model** field, specify a prefix for the refactored model.
- 6 Click the **Run This Check** button. The top **Result** table contains hyperlinks to the Data Store Memory blocks and the corresponding Data Store Read and Data Store Write blocks that qualify for elimination.
- 7 Click the **Refactor Model** button. The bottom **Result** table contains a hyperlink to the new model. The tool creates an `m2m_ex_data_store_replacement` folder. This folder contains the `gen_ex_data_store_replacement.slx` model.



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

Limitations

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

- They cross boundaries of conditionally executed subsystems such as Enabled, Triggered, or Function-Call subsystems and Stateflow Charts.

- They do not complete mutually exclusive branches of If-Action subsystems.
- They cross boundaries of variants.
- They have more than one input or output.
- They access part of an array.
- They execute at different rates.
- They are inside different instances of library subsystems and have a different relative execution order.

See Also

Related Examples

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

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

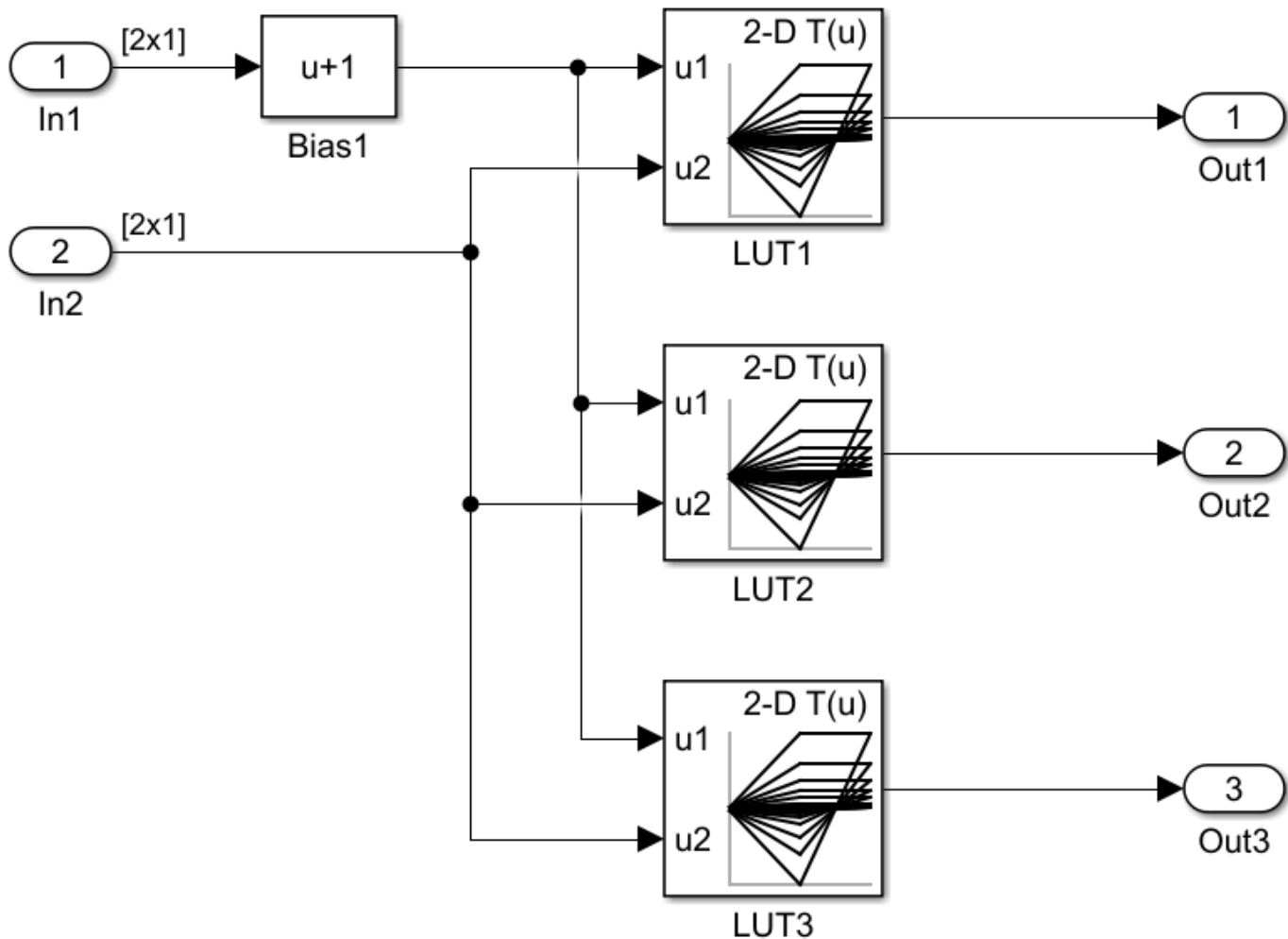
Improve the efficiency of your model simulation by using the Model Transformer tool to identify n-D Lookup Table blocks that qualify for transformation and replacing them with Interpolation blocks and shared Prelookup blocks. Eliminating the redundant Prelookup blocks improves the simulation speed for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

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

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

Improve Efficiency of Simulation by Optimizing Prelookup Operation of Lookup Table Blocks Using Model Transformer

This example shows how to use Model Transformer to identify Lookup Table blocks that qualify for transformation and replacing them with Interpolation blocks and shared Prelookup blocks.



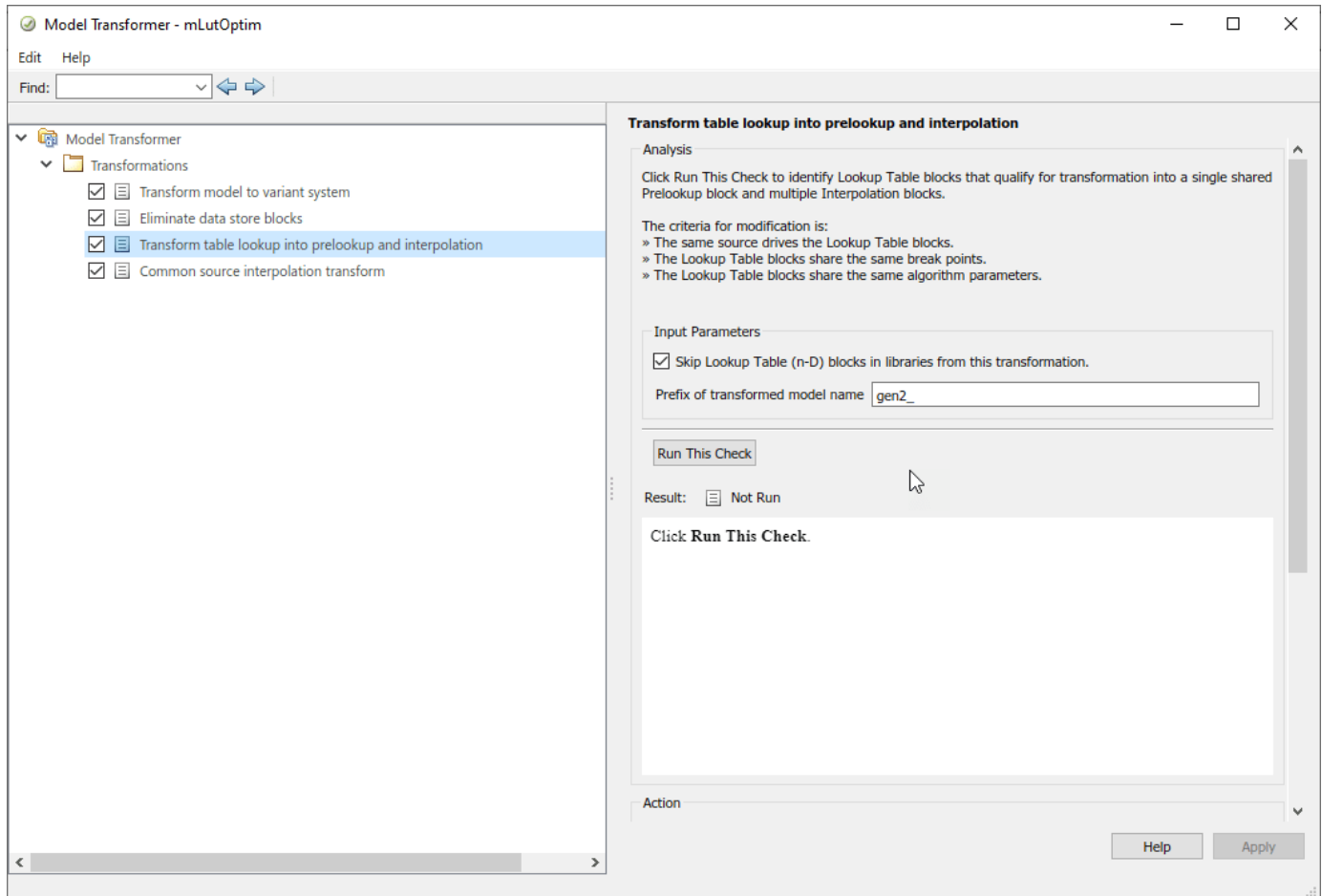
The model `ex_lut_optimize` contains three Lookup Table blocks: LUT1, LUT2 and LUT3. The blocks are driven from the same input sources In1 and In2.

Merge Prelookup Operation

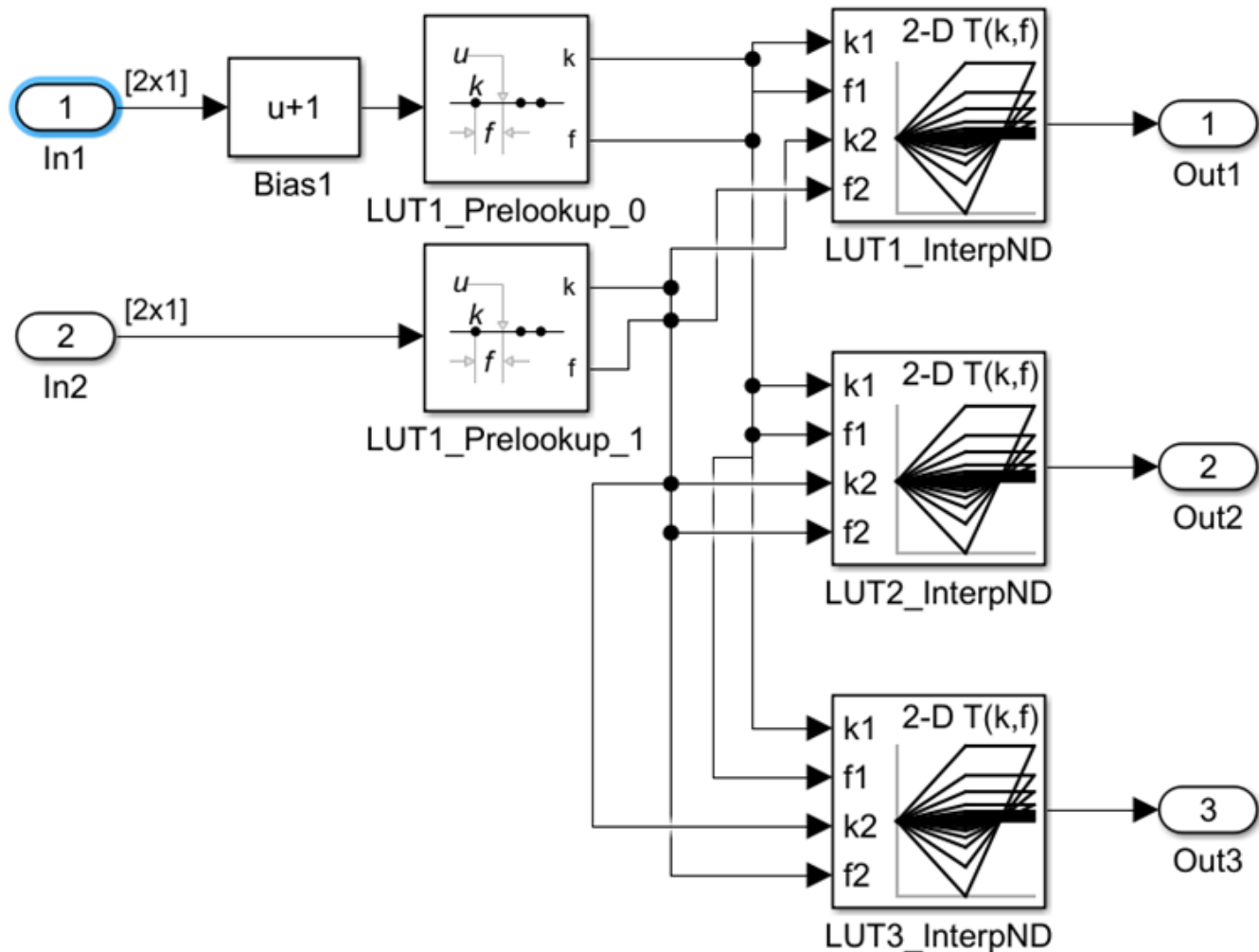
Identify n-D Lookup Table blocks that qualify for transformation and replace them with a single shared Prelookup block and multiple Interpolation blocks.

- 1 Open the model `ex_lut_optimize`.
- 2 Save the model to your working folder.
- 3 On the **Apps** tab, click **Model Transformer**.
- 4 In the **Transformations** folder, select the “Transform Table Lookup Blocks to Prelookup and Interpolation Using Prelookup Blocks” check.
- 5 Select the **Skip Lookup Table (n-D) blocks in the libraries from this transformation** option to avoid replacing Lookup Table blocks that are linked to a library.
- 6 In the **Prefix of refactored model** field, specify a prefix for the new refactored model.

- 7 Click the **Run This Check** button. The top **Result** table contains hyperlinks to the Lookup Table blocks and the corresponding input port indices.
- 8 Clear the **Candidate Groups** that you do not want to transform.
- 9 Click the **Refactor Model** button. The **Result** table contains a hyperlink to the new model. The table also contains hyperlinks to the shared Prelookup block and corresponding Interpolation blocks. Those blocks replaced the original Lookup Table blocks. The tool creates an `m2m_ex_lut_optimize` folder. This folder contains the new `gen_ex_lut_optimize.slx` model.



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



Conditions and Limitations

The Model Transformer cannot replace Lookup Table blocks if:

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

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

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

See Also

Related Examples

- “Refactor Models”
- “Transform Table Lookup Blocks to Prelookup and Interpolation Using Prelookup Blocks”

Model Advisor Checks for DO-178C/DO-331 Standards Compliance

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

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

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

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

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

Model Advisor Checks for High Integrity Systems Modeling Guidelines

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

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

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

Model Advisor Checks for High-Integrity Systems Modeling Guidelines

The Simulink Check Model Advisor provides High-Integrity System Modelling checks that you can use to verify a compliance with safety standards, including:

- DO-178C / DO-331
- IEC 61508
- IEC 62304
- ISO 26262
- EN 50128 (and EN 50657)
- “ISO 25119 Standard” (Embedded Coder)

The high-integrity system modeling guidelines and their corresponding checks are summarized in this table. For the guidelines that do not have Model Advisor checks, it is not possible to automate checking of the guideline. Guidelines without a corresponding check are noted as not applicable.

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
“Check usage of lookup table blocks”	mathworks.hism.hisl_0033	“hisl_0033: Usage of Lookup Table blocks”
“Check for inconsistent vector indexing methods”	mathworks.hism.hisl_0021	“hisl_0021: Consistent vector indexing method”
“Check usage of variant blocks”	mathworks.hism.hisl_0021	“hisl_0023: Verification of variant blocks”
“Check for root Inports with missing properties”	mathworks.hism.hisl_0024	“hisl_0024: Inport interface definition”
“Check usage of Relational Operator blocks”	mathworks.hism.hisl_0017	“hisl_0017: Usage of blocks that compute relational operators (2)”
“Check relational comparisons on floating-point signals”	mathworks.hism.hisl_0016	“hisl_0016: Usage of blocks that compute relational operators”
“Check usage of Logical Operator blocks”	mathworks.hism.hisl_0018	“hisl_0018: Usage of Logical Operator block”

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

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
"Check safety-related code generation settings for code style"	mathworks.hism.hisl_0047	"hisl_0047: Configuration Parameters > Code Generation > Code Style"
"Check safety-related code generation identifier settings"	mathworks.hism.hisl_0049	"hisl_0049: Configuration Parameters > Code Generation > Identifiers"
"Check usage of Abs blocks"	mathworks.hism.hisl_0001	"hisl_0001: Usage of Abs block"
"Check usage of remainder and reciprocal operations"	mathworks.sldv.hism.hisl_0002	"hisl_0002: Usage of remainder and reciprocal operations"
"Check usage of square root operations"	mathworks.hism.hisl_0003	"hisl_0003: Usage of square root operations"
"Check usage of log and log10 operations"	mathworks.sldv.hism.hisl_0004	"hisl_0004: Usage of natural logarithm and base 10 logarithm operations"
"Check usage of Assignment blocks"	mathworks.hism.hisl_0029	"hisl_0029: Usage of Assignment blocks"
"Check usage of Signal Routing blocks"	mathworks.hism.hisl_0034	"hisl_0034: Usage of Signal Routing blocks"
"Check for root Inports with missing range definitions"	mathworks.hism.hisl_0025	"hisl_0025: Design min/max specification of input interfaces"
"Check for root Outports with missing range definitions"	mathworks.hism.hisl_0026	"hisl_0026: Design min/max specification of output interfaces"
"Check state machine type of Stateflow charts"	mathworks.hism.hisf_0001	"hisf_0001: State Machine Type"
"Check Stateflow charts for transition paths that cross parallel state boundaries"	mathworks.hism.hisf_0013	"hisf_0013: Usage of transition paths (crossing parallel state boundaries)"
"Check Stateflow charts for ordering of states and transitions"	mathworks.hism.hisf_0002	"hisf_0002: User-specified state/transition execution order"
"Check Stateflow debugging options"	mathworks.hism.hisf_0011	"hisf_0011: Stateflow debugging settings"
"Check Stateflow charts for uniquely defined data objects"	mathworks.hism.hisl_0061	"hisl_0061: Unique identifiers for clarity"
"Check Stateflow charts for strong data typing"	mathworks.hism.hisf_0015	"hisf_0015: Strong data typing (casting variables and parameters in expressions)"
"Check assignment operations in Stateflow Charts"	mathworks.hism.hisf_0065	"hisf_0065: Type cast operations in Stateflow to improve code compliance"

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
"Check Stateflow charts for unary operators"	mathworks.hism.hisf_0211	"hisf_0211: Protect against use of unary operators in Stateflow Charts to improve code compliance"
"Check for MATLAB Function interfaces with inherited properties"	mathworks.hism.himl_0002	"himl_0002: Strong data typing at MATLAB function boundaries"
"Check MATLAB Function metrics"	mathworks.hism.himl_0003	"himl_0003: Complexity of user-defined MATLAB Functions"
"Check MATLAB Code Analyzer messages"	mathworks.hism.himl_0004	"himl_0004: MATLAB Code Analyzer recommendations for code generation"
"Check safety-related model referencing settings"	mathworks.hism.hisl_0037	"hisl_0037: Configuration Parameters > Model Referencing"
"Check safety-related diagnostic settings for solvers"	mathworks.hism.hisl_0043	"hisl_0043: Configuration Parameters > Diagnostics > Solver"
"Check safety-related solver settings for simulation time"	mathworks.hism.hisl_0040	"hisl_0040: Configuration Parameters > Solver > Simulation time"
"Check safety-related solver settings for solver options"	mathworks.hism.hisl_0041	"hisl_0041: Configuration Parameters > Solver > Solver options"
"Check safety-related solver settings for tasking and sample-time"	mathworks.hism.hisl_0042	"hisl_0042: Configuration Parameters > Solver > Tasking and sample time options"
"Check safety-related diagnostic settings for sample time"	mathworks.hism.hisl_0044	"hisl_0044: Configuration Parameters > Diagnostics > Sample Time"
"Check safety-related diagnostic settings for parameters"	mathworks.hism.hisl_0302	"hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters"
"Check safety-related diagnostic settings for data used for debugging"	mathworks.hism.hisl_0305	"hisl_0305: Configuration Parameters > Diagnostics > Data Validity > Debugging"
"Check safety-related diagnostic settings for data store memory"	mathworks.hism.hisl_0013	"hisl_0013: Usage of data store memory"
"Check safety-related diagnostic settings for type conversions"	mathworks.hism.hisl_0309	"hisl_0309: Configuration Parameters > Diagnostics > Type Conversion"
"Check safety-related diagnostic settings for signal connectivity"	mathworks.hism.hisl_0306	"hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals"

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
"Check safety-related diagnostic settings for bus connectivity"	mathworks.hism.hisl_0307	"hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses"
"Check safety-related diagnostic settings that apply to function-call connectivity"	mathworks.hism.hisl_0308	"hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls"
"Check safety-related diagnostic settings for compatibility"	mathworks.hism.hisl_0301	"hisl_0301: Configuration Parameters > Diagnostics > Compatibility"
"Check safety-related diagnostic settings for model initialization"	mathworks.hism.hisl_0304	"hisl_0304: Configuration Parameters > Diagnostics > Data Validity > Model initialization"
"Check safety-related diagnostic settings for model referencing"	mathworks.hism.hisl_0310	"hisl_0310: Configuration Parameters > Diagnostics > Model Referencing"
"Check safety-related diagnostic settings for saving"	mathworks.hism.hisl_0036	"hisl_0036: Configuration Parameters > Diagnostics > Saving"
"Check safety-related diagnostic settings for Merge blocks"	mathworks.hism.hisl_0303	"hisl_0303: Configuration Parameters > Diagnostics > Data Validity > Merge blocks"
"Check safety-related diagnostic settings for Stateflow"	mathworks.hism.hisl_0311	"hisl_0311: Configuration Parameters > Diagnostics > Stateflow"
"Check model object names"	mathworks.hism.hisl_0032	"hisl_0032: Model element names"
"Check for model elements that do not link to requirements"	mathworks.hism.hisl_0070	"hisl_0070: Placement of requirement links in a model"
"Check for inappropriate use of transition paths"	mathworks.hism.hisf_0014	"hisf_0014: Usage of transition paths (passing through states)"
"Check usage of bit operation blocks"	mathworks.hism.hisl_0019	"hisl_0019: Usage of bitwise operations"
"Check data types for blocks with index signals"	mathworks.hism.hisl_0022	"hisl_0022: Data type selection for index signals"
"Check model file name"	mathworks.hism.hisl_0031	"hisl_0031: Model file names"
"Check if/elseif/else patterns in MATLAB Function blocks"	mathworks.hism.hisl_0022	"himl_0006: MATLAB code if / elseif / else patterns"
"Check switch statements in MATLAB Function blocks"	mathworks.hism.himl_0007	"himl_0007: MATLAB code switch / case / otherwise patterns"
"Check global variables in graphical functions"	mathworks.hism.hisl_0062	"hisl_0062: Global variables in graphical functions"

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
"Check for length of user-defined object names"	mathworks.hism.hisl_0063	"hisl_0063: Length of user-defined object names to improve MISRA C:2012 compliance"
"Check usage of Merge blocks"	mathworks.hism.hisl_0015	"hisl_0015: Usage of Merge blocks"
"Check usage of conditionally executed subsystems"	mathworks.hism.hisl_0012	"hisl_0012: Usage of conditionally executed subsystems"
"Check usage of standardized MATLAB function headers"	mathworks.hism.himl_0001	"himl_0001: Usage of standardized MATLAB function headers"
"Check usage of relational operators in MATLAB Function blocks"	mathworks.hism.himl_0008	"himl_0008: MATLAB code relational operator data types"
"Check usage of logical operators and functions in MATLAB Function blocks"	mathworks.hism.himl_0010	"himl_0010: MATLAB code with logical operators and functions"
"Check type and size of condition expressions"	mathworks.hism.himl_0011	"himl_0011: Data type and size of condition expressions"
"Check naming of ports in Stateflow charts"	mathworks.hism.hisf_0016	"hisf_0016: Stateflow port names"
"Check scoping of Stateflow data objects"	mathworks.hism.hisf_0017	"hisf_0017: Stateflow data object scoping"
"Check usage of Gain blocks"	mathworks.hism.hisl_0066	"hisl_0066: Usage of Gain blocks"
"Check for divide-by-zero calculations"	mathworks.hism.hisl_0067	"hisl_0067: Protect against divide-by-zero calculations"
"Check data type of loop control variables"	mathworks.hism.hisl_0102	"hisl_0102: Data type of loop control variables to improve MISRA C:2012 compliance"
"Check configuration parameters for MISRA C:2012"	mathworks.misra.CodeGenerationSettings	"hisl_0060: Configuration parameters that improve MISRA C:2012 compliance"
"Check for blocks not recommended for C/C++ production code deployment" "Check for blocks not recommended for MISRA C:2012"	mathworks.hism.hisl_0020 mathworks.misra.BlkSupport	"hisl_0020: Blocks not recommended for MISRA C:2012 compliance"
"Check safety-related optimization settings for specified minimum and maximum values"	mathworks.hism.hisl_0056	"hisl_0056: Configuration Parameters > Code Generation > Optimization > Optimize using the specified minimum and maximum values"

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
“Check usage of Reciprocal Sqrt blocks”	mathworks.hism.hisl_0028	“hisl_0028: Usage of Reciprocal Square Root blocks”
“Check safety-related settings for hardware implementation”	mathworks.hism.hisl_0071	“hisl_0071: Configuration Parameters > Hardware Implementation >Inconsistent hardware implementation settings”
“Check usage of recursions”	mathworks.hism.hisf_0004	“hisf_0004: Protect against recursive function calls to improve code compliance”
“Check MATLAB functions not supported for code generation”	mathworks.hism.himl_0012	“himl_0012: Usage of MATLAB functions for code generation”
“Metrics for generated code complexity”	mathworks.hism.himl_0013	“himl_0013: Limitation of built-in MATLAB Function complexity”
“Check for parameter tunability ignored for referenced models”	mathworks.hism.hisl_0072	“hisl_0072: Usage of tunable parameters for referenced models”
“Check usage of bit-shift operations”	mathworks.hism.hisl_0073	“hisl_0073: Usage of bit-shift operations”
“Check safety-related diagnostic settings for variants”	mathworks.hism.hisl_0074	“hisl_0074: Configuration Parameters > Diagnostics > Modeling issues related to variants”
“Check for disabled and parameterized library links”	mathworks.hism.hisl_0075	“hisl_0075: Usage of library links”
“Check for unreachable and dead code”	mathworks.hism.hisl_0101	“hisl_0101: Avoid operations that result in dead logic to improve code compliance”
“Check for root Outports with missing properties”	mathworks.hism.hisl_0077	“hisl_0077: Outport interface definition”

See Also

- “Check Your Model Using the Model Advisor”
- “High-Integrity System Modeling”

See Also
Related Examples

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

Model Advisor Checks for DO-254 Standard Compliance

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

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

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

The table below lists the DO-254 checks.

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

Model Checks for High Integrity Systems Modeling

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

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

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

High Integrity System Model Checks	Applicable High-Integrity System Modeling Guidelines
Check for inconsistent vector indexing methods	"hisl_0021: Consistent vector indexing method"
Check for variant blocks with 'Generate preprocessor conditionals' active	"hisl_0023: Verification of variant blocks"
Check for root Inports with missing properties	"hisl_0024: Inport interface definition"
Check for Relational Operator blocks that equate floating-point types	"hisl_0017: Usage of blocks that compute relational operators (2)"

High Integrity System Model Checks	Applicable High-Integrity System Modeling Guidelines
Check relational comparisons on floating-point signals	"hisl_0016: Usage of blocks that compute relational operators"
Check usage of Logical Operator blocks	"hisl_0018: Usage of Logical Operator block"
Check sample time-dependent blocks	"hisl_0007: Usage of For Iterator or While Iterator subsystems"
Check safety-related block reduction optimization settings	"hisl_0046: Configuration Parameters > Simulation Target > Block reduction"
Check usage of Abs blocks	"hisl_0001: Usage of Abs block"
Check usage of Assignment blocks	"hisl_0029: Usage of Assignment blocks"
Check for root Inports with missing range definitions	"hisl_0025: Design min/max specification of input interfaces"
Check for root Outports with missing range definitions	"hisl_0026: Design min/max specification of output interfaces"
Check Stateflow charts for transition paths that cross parallel state boundaries	"hisf_0013: Usage of transition paths (crossing parallel state boundaries)"
Check Stateflow charts for ordering of states and transitions	"hisf_0002: User-specified state/transition execution order"
Check Stateflow debugging options	"hisf_0011: Stateflow debugging settings"
Check Stateflow charts for uniquely defined data objects	"hisl_0061: Unique identifiers for clarity"
Check Stateflow charts for unary operators	"hisf_0211: Protect against use of unary operators in Stateflow Charts to improve code compliance"
Check MATLAB Code Analyzer messages	"himl_0004: MATLAB Code Analyzer recommendations for code generation"
Check safety-related model referencing settings	"hisl_0037: Configuration Parameters > Model Referencing"
Check safety-related diagnostic settings for parameters	"hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters"
Check safety-related diagnostic settings for type conversions	"hisl_0309: Configuration Parameters > Diagnostics > Type Conversion"
Check safety-related diagnostic settings for signal connectivity	"hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals"
Check safety-related diagnostic settings for bus connectivity	"hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses"
Check safety-related diagnostic settings for model initialization	"hisl_0304: Configuration Parameters > Diagnostics > Data Validity > Model initialization"
Check safety-related diagnostic settings for model referencing	"hisl_0310: Configuration Parameters > Diagnostics > Model Referencing"

High Integrity System Model Checks	Applicable High-Integrity System Modeling Guidelines
Check safety-related diagnostic settings for saving	"hisl_0036: Configuration Parameters > Diagnostics > Saving"
Check safety-related diagnostic settings for Stateflow	"hisl_0311: Configuration Parameters > Diagnostics > Stateflow"
Check model object names	"hisl_0032: Model element names"
Check for model elements that do not link to requirements	"hisl_0070: Placement of requirement links in a model"
Check for inappropriate use of transition paths	"hisf_0014: Usage of transition paths (passing through states)"
Check usage of Bitwise Operator block	"hisl_0019: Usage of bitwise operations"
Check data types for blocks with index signals	"hisl_0022: Data type selection for index signals"
Check model file name	"hisl_0031: Model file names"
Check if/elseif/else patterns in MATLAB Function blocks	"himl_0006: MATLAB code if / elseif / else patterns"
Check switch statements in MATLAB Function blocks	"himl_0007: MATLAB code switch / case / otherwise patterns"
Check global variables in graphical functions	"hisl_0062: Global variables in graphical functions"
Check for length of user-defined object names	"hisl_0063: Length of user-defined object names to improve MISRA C:2012 compliance"
Check usage of conditionally executed subsystems	"hisl_0012: Usage of conditionally executed subsystems"
Check usage of standardized MATLAB function headers	"himl_0001: Usage of standardized MATLAB function headers"
Check usage of relational operators in MATLAB Function blocks	"himl_0008: MATLAB code relational operator data types"
Check usage of logical operators and functions in MATLAB Function blocks	"himl_0010: MATLAB code with logical operators and functions"
Check naming of ports in Stateflow charts	"hisf_0016: Stateflow port names"
Check scoping of Stateflow data objects	"hisf_0017: Stateflow data object scoping"
Check usage of Gain blocks	"hisl_0066: Usage of Gain blocks"
Check data type of loop control variables	"hisl_0102: Data type of loop control variables to improve MISRA C:2012 compliance"
Check for root Outports with missing properties	"hisl_0077: Outport interface definition"

HDL Code Advisor Checks

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

- Model configuration settings

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

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

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

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

See Also

Related Examples

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

Model Advisor Checks for High Integrity Systems Modeling Guidelines

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

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

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

Model Advisor Checks for High-Integrity Systems Modeling Guidelines

The Simulink Check Model Advisor provides High-Integrity System Modelling checks that you can use to verify a compliance with safety standards, including:

- DO-178C / DO-331
- IEC 61508
- IEC 62304
- ISO 26262
- EN 50128 (and EN 50657)
- “ISO 25119 Standard” (Embedded Coder)

The high-integrity system modeling guidelines and their corresponding checks are summarized in this table. For the guidelines that do not have Model Advisor checks, it is not possible to automate checking of the guideline. Guidelines without a corresponding check are noted as not applicable.

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
“Check usage of lookup table blocks”	mathworks.hism.hisl_0033	“hisl_0033: Usage of Lookup Table blocks”
“Check for inconsistent vector indexing methods”	mathworks.hism.hisl_0021	“hisl_0021: Consistent vector indexing method”
“Check usage of variant blocks”	mathworks.hism.hisl_0021	“hisl_0023: Verification of variant blocks”
“Check for root Inports with missing properties”	mathworks.hism.hisl_0024	“hisl_0024: Inport interface definition”
“Check usage of Relational Operator blocks”	mathworks.hism.hisl_0017	“hisl_0017: Usage of blocks that compute relational operators (2)”

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

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
"Check safety-related code generation interface settings"	mathworks.hism.hisl_0039	"hisl_0039: Configuration Parameters > Code Generation > Interface"
"Check safety-related code generation settings for code style"	mathworks.hism.hisl_0047	"hisl_0047: Configuration Parameters > Code Generation > Code Style"
"Check safety-related code generation identifier settings"	mathworks.hism.hisl_0049	"hisl_0049: Configuration Parameters > Code Generation > Identifiers"
"Check usage of Abs blocks"	mathworks.hism.hisl_0001	"hisl_0001: Usage of Abs block"
"Check usage of remainder and reciprocal operations"	mathworks.sldv.hism.hisl_0002	"hisl_0002: Usage of remainder and reciprocal operations"
"Check usage of square root operations"	mathworks.hism.hisl_0003	"hisl_0003: Usage of square root operations"
"Check usage of log and log10 operations"	mathworks.sldv.hism.hisl_0004	"hisl_0004: Usage of natural logarithm and base 10 logarithm operations"
"Check usage of Assignment blocks"	mathworks.hism.hisl_0029	"hisl_0029: Usage of Assignment blocks"
"Check usage of Signal Routing blocks"	mathworks.hism.hisl_0034	"hisl_0034: Usage of Signal Routing blocks"
"Check for root Inports with missing range definitions"	mathworks.hism.hisl_0025	"hisl_0025: Design min/max specification of input interfaces"
"Check for root Outports with missing range definitions"	mathworks.hism.hisl_0026	"hisl_0026: Design min/max specification of output interfaces"
"Check state machine type of Stateflow charts"	mathworks.hism.hisf_0001	"hisf_0001: State Machine Type"
"Check Stateflow charts for transition paths that cross parallel state boundaries"	mathworks.hism.hisf_0013	"hisf_0013: Usage of transition paths (crossing parallel state boundaries)"
"Check Stateflow charts for ordering of states and transitions"	mathworks.hism.hisf_0002	"hisf_0002: User-specified state/transition execution order"
"Check Stateflow debugging options"	mathworks.hism.hisf_0011	"hisf_0011: Stateflow debugging settings"
"Check Stateflow charts for uniquely defined data objects"	mathworks.hism.hisl_0061	"hisl_0061: Unique identifiers for clarity"
"Check Stateflow charts for strong data typing"	mathworks.hism.hisf_0015	"hisf_0015: Strong data typing (casting variables and parameters in expressions)"
"Check assignment operations in Stateflow Charts"	mathworks.hism.hisf_0065	"hisf_0065: Type cast operations in Stateflow to improve code compliance"

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
"Check Stateflow charts for unary operators"	mathworks.hism.hisf_0211	"hisf_0211: Protect against use of unary operators in Stateflow Charts to improve code compliance"
"Check for MATLAB Function interfaces with inherited properties"	mathworks.hism.himl_0002	"himl_0002: Strong data typing at MATLAB function boundaries"
"Check MATLAB Function metrics"	mathworks.hism.himl_0003	"himl_0003: Complexity of user-defined MATLAB Functions"
"Check MATLAB Code Analyzer messages"	mathworks.hism.himl_0004	"himl_0004: MATLAB Code Analyzer recommendations for code generation"
"Check safety-related model referencing settings"	mathworks.hism.hisl_0037	"hisl_0037: Configuration Parameters > Model Referencing"
"Check safety-related diagnostic settings for solvers"	mathworks.hism.hisl_0043	"hisl_0043: Configuration Parameters > Diagnostics > Solver"
"Check safety-related solver settings for simulation time"	mathworks.hism.hisl_0040	"hisl_0040: Configuration Parameters > Solver > Simulation time"
"Check safety-related solver settings for solver options"	mathworks.hism.hisl_0041	"hisl_0041: Configuration Parameters > Solver > Solver options"
"Check safety-related solver settings for tasking and sample-time"	mathworks.hism.hisl_0042	"hisl_0042: Configuration Parameters > Solver > Tasking and sample time options"
"Check safety-related diagnostic settings for sample time"	mathworks.hism.hisl_0044	"hisl_0044: Configuration Parameters > Diagnostics > Sample Time"
"Check safety-related diagnostic settings for parameters"	mathworks.hism.hisl_0302	"hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters"
"Check safety-related diagnostic settings for data used for debugging"	mathworks.hism.hisl_0305	"hisl_0305: Configuration Parameters > Diagnostics > Data Validity > Debugging"
"Check safety-related diagnostic settings for data store memory"	mathworks.hism.hisl_0013	"hisl_0013: Usage of data store memory"
"Check safety-related diagnostic settings for type conversions"	mathworks.hism.hisl_0309	"hisl_0309: Configuration Parameters > Diagnostics > Type Conversion"
"Check safety-related diagnostic settings for signal connectivity"	mathworks.hism.hisl_0306	"hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals"

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
"Check safety-related diagnostic settings for bus connectivity"	mathworks.hism.hisl_0307	"hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses"
"Check safety-related diagnostic settings that apply to function-call connectivity"	mathworks.hism.hisl_0308	"hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls"
"Check safety-related diagnostic settings for compatibility"	mathworks.hism.hisl_0301	"hisl_0301: Configuration Parameters > Diagnostics > Compatibility"
"Check safety-related diagnostic settings for model initialization"	mathworks.hism.hisl_0304	"hisl_0304: Configuration Parameters > Diagnostics > Data Validity > Model initialization"
"Check safety-related diagnostic settings for model referencing"	mathworks.hism.hisl_0310	"hisl_0310: Configuration Parameters > Diagnostics > Model Referencing"
"Check safety-related diagnostic settings for saving"	mathworks.hism.hisl_0036	"hisl_0036: Configuration Parameters > Diagnostics > Saving"
"Check safety-related diagnostic settings for Merge blocks"	mathworks.hism.hisl_0303	"hisl_0303: Configuration Parameters > Diagnostics > Data Validity > Merge blocks"
"Check safety-related diagnostic settings for Stateflow"	mathworks.hism.hisl_0311	"hisl_0311: Configuration Parameters > Diagnostics > Stateflow"
"Check model object names"	mathworks.hism.hisl_0032	"hisl_0032: Model element names"
"Check for model elements that do not link to requirements"	mathworks.hism.hisl_0070	"hisl_0070: Placement of requirement links in a model"
"Check for inappropriate use of transition paths"	mathworks.hism.hisf_0014	"hisf_0014: Usage of transition paths (passing through states)"
"Check usage of bit operation blocks"	mathworks.hism.hisl_0019	"hisl_0019: Usage of bitwise operations"
"Check data types for blocks with index signals"	mathworks.hism.hisl_0022	"hisl_0022: Data type selection for index signals"
"Check model file name"	mathworks.hism.hisl_0031	"hisl_0031: Model file names"
"Check if/elseif/else patterns in MATLAB Function blocks"	mathworks.hism.hisl_0022	"himl_0006: MATLAB code if / elseif / else patterns"
"Check switch statements in MATLAB Function blocks"	mathworks.hism.himl_0007	"himl_0007: MATLAB code switch / case / otherwise patterns"
"Check global variables in graphical functions"	mathworks.hism.hisl_0062	"hisl_0062: Global variables in graphical functions"

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
"Check for length of user-defined object names"	mathworks.hism.hisl_0063	"hisl_0063: Length of user-defined object names to improve MISRA C:2012 compliance"
"Check usage of Merge blocks"	mathworks.hism.hisl_0015	"hisl_0015: Usage of Merge blocks"
"Check usage of conditionally executed subsystems"	mathworks.hism.hisl_0012	"hisl_0012: Usage of conditionally executed subsystems"
"Check usage of standardized MATLAB function headers"	mathworks.hism.himl_0001	"himl_0001: Usage of standardized MATLAB function headers"
"Check usage of relational operators in MATLAB Function blocks"	mathworks.hism.himl_0008	"himl_0008: MATLAB code relational operator data types"
"Check usage of logical operators and functions in MATLAB Function blocks"	mathworks.hism.himl_0010	"himl_0010: MATLAB code with logical operators and functions"
"Check type and size of condition expressions"	mathworks.hism.himl_0011	"himl_0011: Data type and size of condition expressions"
"Check naming of ports in Stateflow charts"	mathworks.hism.hisf_0016	"hisf_0016: Stateflow port names"
"Check scoping of Stateflow data objects"	mathworks.hism.hisf_0017	"hisf_0017: Stateflow data object scoping"
"Check usage of Gain blocks"	mathworks.hism.hisl_0066	"hisl_0066: Usage of Gain blocks"
"Check for divide-by-zero calculations"	mathworks.hism.hisl_0067	"hisl_0067: Protect against divide-by-zero calculations"
"Check data type of loop control variables"	mathworks.hism.hisl_0102	"hisl_0102: Data type of loop control variables to improve MISRA C:2012 compliance"
"Check configuration parameters for MISRA C:2012"	mathworks.misra.CodeGenerationSettings	"hisl_0060: Configuration parameters that improve MISRA C:2012 compliance"
"Check for blocks not recommended for C/C++ production code deployment" "Check for blocks not recommended for MISRA C:2012"	mathworks.hism.hisl_0020 mathworks.misra.BlkSupport	"hisl_0020: Blocks not recommended for MISRA C:2012 compliance"
"Check safety-related optimization settings for specified minimum and maximum values"	mathworks.hism.hisl_0056	"hisl_0056: Configuration Parameters > Code Generation > Optimization > Optimize using the specified minimum and maximum values"

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
“Check usage of Reciprocal Sqrt blocks”	mathworks.hism.hisl_0028	“hisl_0028: Usage of Reciprocal Square Root blocks”
“Check safety-related settings for hardware implementation”	mathworks.hism.hisl_0071	“hisl_0071: Configuration Parameters > Hardware Implementation >Inconsistent hardware implementation settings”
“Check usage of recursions”	mathworks.hism.hisf_0004	“hisf_0004: Protect against recursive function calls to improve code compliance”
“Check MATLAB functions not supported for code generation”	mathworks.hism.himl_0012	“himl_0012: Usage of MATLAB functions for code generation”
“Metrics for generated code complexity”	mathworks.hism.himl_0013	“himl_0013: Limitation of built-in MATLAB Function complexity”
“Check for parameter tunability ignored for referenced models”	mathworks.hism.hisl_0072	“hisl_0072: Usage of tunable parameters for referenced models”
“Check usage of bit-shift operations”	mathworks.hism.hisl_0073	“hisl_0073: Usage of bit-shift operations”
“Check safety-related diagnostic settings for variants”	mathworks.hism.hisl_0074	“hisl_0074: Configuration Parameters > Diagnostics > Modeling issues related to variants”
“Check for disabled and parameterized library links”	mathworks.hism.hisl_0075	“hisl_0075: Usage of library links”
“Check for unreachable and dead code”	mathworks.hism.hisl_0101	“hisl_0101: Avoid operations that result in dead logic to improve code compliance”
“Check for root Outports with missing properties”	mathworks.hism.hisl_0077	“hisl_0077: Outport interface definition”

See Also

- “Check Your Model Using the Model Advisor”
- “High-Integrity System Modeling”

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

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

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

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

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

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

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

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

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

Model Advisor Checks for High Integrity Systems Modeling Guidelines

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

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

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

Model Advisor Checks for High-Integrity Systems Modeling Guidelines

The Simulink Check Model Advisor provides High-Integrity System Modelling checks that you can use to verify a compliance with safety standards, including:

- DO-178C / DO-331
- IEC 61508
- IEC 62304
- ISO 26262
- EN 50128 (and EN 50657)
- "ISO 25119 Standard" (Embedded Coder)

The high-integrity system modeling guidelines and their corresponding checks are summarized in this table. For the guidelines that do not have Model Advisor checks, it is not possible to automate checking of the guideline. Guidelines without a corresponding check are noted as not applicable.

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
"Check usage of lookup table blocks"	mathworks.hism.hisl_0033	"hisl_0033: Usage of Lookup Table blocks"
"Check for inconsistent vector indexing methods"	mathworks.hism.hisl_0021	"hisl_0021: Consistent vector indexing method"
"Check usage of variant blocks"	mathworks.hism.hisl_0021	"hisl_0023: Verification of variant blocks"
"Check for root Inports with missing properties"	mathworks.hism.hisl_0024	"hisl_0024: Inport interface definition"
"Check usage of Relational Operator blocks"	mathworks.hism.hisl_0017	"hisl_0017: Usage of blocks that compute relational operators (2)"
"Check relational comparisons on floating-point signals"	mathworks.hism.hisl_0016	"hisl_0016: Usage of blocks that compute relational operators"
"Check usage of Logical Operator blocks"	mathworks.hism.hisl_0018	"hisl_0018: Usage of Logical Operator block"
"Check usage of While Iterator blocks"	mathworks.hism.hisl_0006	"hisl_0006: Usage of While Iterator blocks"
"Check usage of For and While Iterator subsystems"	mathworks.hism.hisl_0007	"hisl_0007: Usage of For Iterator or While Iterator subsystems"
"Check usage of For Iterator blocks"	mathworks.hism.hisl_0008	"hisl_0008: Usage of For Iterator Blocks"
"Check usage of If blocks and If Action Subsystem blocks"	mathworks.hism.hisl_0010	"hisl_0010: Usage of If blocks and If Action Subsystem blocks"
"Check usage of Switch Case blocks and Switch Case Action Subsystem blocks"	mathworks.hism.hisl_0011	"hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks"
"Check safety-related optimization settings for logic signals"	mathworks.hism.hisl_0045	"hisl_0045: Configuration Parameters > Math and Data Types > Implement logic signals as Boolean data (vs. double)"
"Check safety-related block reduction optimization settings"	mathworks.hism.hisl_0046	"hisl_0046: Configuration Parameters > Simulation Target > Block reduction"
"Check safety-related optimization settings for application lifespan"	mathworks.hism.hisl_0048	"hisl_0048: Configuration Parameters > Math and Data Types > Application lifespan (days)"
"Check safety-related optimization settings for data initialization"	mathworks.hism.hisl_0052	"hisl_0052: Configuration Parameters > Code Generation > Optimization > Data initialization"

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
"Check safety-related optimization settings for data type conversions"	mathworks.hism.hisl_0053	"hisl_0053: Configuration Parameters > Code Generation > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values"
"Check safety-related optimization settings for division arithmetic exceptions"	mathworks.hism.hisl_0054	"hisl_0054: Configuration Parameters > Code Generation > Optimization > Remove code that protects against division arithmetic exceptions"
"Check safety-related code generation settings for comments"	mathworks.hism.hisl_0038	"hisl_0038: Configuration Parameters > Code Generation > Comments"
"Check safety-related code generation interface settings"	mathworks.hism.hisl_0039	"hisl_0039: Configuration Parameters > Code Generation > Interface"
"Check safety-related code generation settings for code style"	mathworks.hism.hisl_0047	"hisl_0047: Configuration Parameters > Code Generation > Code Style"
"Check safety-related code generation identifier settings"	mathworks.hism.hisl_0049	"hisl_0049: Configuration Parameters > Code Generation > Identifiers"
"Check usage of Abs blocks"	mathworks.hism.hisl_0001	"hisl_0001: Usage of Abs block"
"Check usage of remainder and reciprocal operations"	mathworks.sldv.hism.hisl_0002	"hisl_0002: Usage of remainder and reciprocal operations"
"Check usage of square root operations"	mathworks.hism.hisl_0003	"hisl_0003: Usage of square root operations"
"Check usage of log and log10 operations"	mathworks.sldv.hism.hisl_0004	"hisl_0004: Usage of natural logarithm and base 10 logarithm operations"
"Check usage of Assignment blocks"	mathworks.hism.hisl_0029	"hisl_0029: Usage of Assignment blocks"
"Check usage of Signal Routing blocks"	mathworks.hism.hisl_0034	"hisl_0034: Usage of Signal Routing blocks"
"Check for root Inports with missing range definitions"	mathworks.hism.hisl_0025	"hisl_0025: Design min/max specification of input interfaces"
"Check for root Outports with missing range definitions"	mathworks.hism.hisl_0026	"hisl_0026: Design min/max specification of output interfaces"
"Check state machine type of Stateflow charts"	mathworks.hism.hisf_0001	"hisf_0001: State Machine Type"

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
“Check Stateflow charts for transition paths that cross parallel state boundaries”	mathworks.hism.hisf_0013	“hisf_0013: Usage of transition paths (crossing parallel state boundaries)”
“Check Stateflow charts for ordering of states and transitions”	mathworks.hism.hisf_0002	“hisf_0002: User-specified state/transition execution order”
“Check Stateflow debugging options”	mathworks.hism.hisf_0011	“hisf_0011: Stateflow debugging settings”
“Check Stateflow charts for uniquely defined data objects”	mathworks.hism.hisl_0061	“hisl_0061: Unique identifiers for clarity”
“Check Stateflow charts for strong data typing”	mathworks.hism.hisf_0015	“hisf_0015: Strong data typing (casting variables and parameters in expressions)”
“Check assignment operations in Stateflow Charts”	mathworks.hism.hisf_0065	“hisf_0065: Type cast operations in Stateflow to improve code compliance”
“Check Stateflow charts for unary operators”	mathworks.hism.hisf_0211	“hisf_0211: Protect against use of unary operators in Stateflow Charts to improve code compliance”
“Check for MATLAB Function interfaces with inherited properties”	mathworks.hism.himl_0002	“himl_0002: Strong data typing at MATLAB function boundaries”
“Check MATLAB Function metrics”	mathworks.hism.himl_0003	“himl_0003: Complexity of user-defined MATLAB Functions”
“Check MATLAB Code Analyzer messages”	mathworks.hism.himl_0004	“himl_0004: MATLAB Code Analyzer recommendations for code generation”
“Check safety-related model referencing settings”	mathworks.hism.hisl_0037	“hisl_0037: Configuration Parameters > Model Referencing”
“Check safety-related diagnostic settings for solvers”	mathworks.hism.hisl_0043	“hisl_0043: Configuration Parameters > Diagnostics > Solver”
“Check safety-related solver settings for simulation time”	mathworks.hism.hisl_0040	“hisl_0040: Configuration Parameters > Solver > Simulation time”
“Check safety-related solver settings for solver options”	mathworks.hism.hisl_0041	“hisl_0041: Configuration Parameters > Solver > Solver options”
“Check safety-related solver settings for tasking and sample-time”	mathworks.hism.hisl_0042	“hisl_0042: Configuration Parameters > Solver > Tasking and sample time options”

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
"Check safety-related diagnostic settings for sample time"	mathworks.hism.hisl_0044	"hisl_0044: Configuration Parameters > Diagnostics > Sample Time"
"Check safety-related diagnostic settings for parameters"	mathworks.hism.hisl_0302	"hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters"
"Check safety-related diagnostic settings for data used for debugging"	mathworks.hism.hisl_0305	"hisl_0305: Configuration Parameters > Diagnostics > Data Validity > Debugging"
"Check safety-related diagnostic settings for data store memory"	mathworks.hism.hisl_0013	"hisl_0013: Usage of data store memory"
"Check safety-related diagnostic settings for type conversions"	mathworks.hism.hisl_0309	"hisl_0309: Configuration Parameters > Diagnostics > Type Conversion"
"Check safety-related diagnostic settings for signal connectivity"	mathworks.hism.hisl_0306	"hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals"
"Check safety-related diagnostic settings for bus connectivity"	mathworks.hism.hisl_0307	"hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses"
"Check safety-related diagnostic settings that apply to function-call connectivity"	mathworks.hism.hisl_0308	"hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls"
"Check safety-related diagnostic settings for compatibility"	mathworks.hism.hisl_0301	"hisl_0301: Configuration Parameters > Diagnostics > Compatibility"
"Check safety-related diagnostic settings for model initialization"	mathworks.hism.hisl_0304	"hisl_0304: Configuration Parameters > Diagnostics > Data Validity > Model initialization"
"Check safety-related diagnostic settings for model referencing"	mathworks.hism.hisl_0310	"hisl_0310: Configuration Parameters > Diagnostics > Model Referencing"
"Check safety-related diagnostic settings for saving"	mathworks.hism.hisl_0036	"hisl_0036: Configuration Parameters > Diagnostics > Saving"
"Check safety-related diagnostic settings for Merge blocks"	mathworks.hism.hisl_0303	"hisl_0303: Configuration Parameters > Diagnostics > Data Validity > Merge blocks"
"Check safety-related diagnostic settings for Stateflow"	mathworks.hism.hisl_0311	"hisl_0311: Configuration Parameters > Diagnostics > Stateflow"
"Check model object names"	mathworks.hism.hisl_0032	"hisl_0032: Model element names"

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
“Check for model elements that do not link to requirements”	mathworks.hism.hisl_0070	“hisl_0070: Placement of requirement links in a model”
“Check for inappropriate use of transition paths”	mathworks.hism.hisf_0014	“hisf_0014: Usage of transition paths (passing through states)”
“Check usage of bit operation blocks”	mathworks.hism.hisl_0019	“hisl_0019: Usage of bitwise operations”
“Check data types for blocks with index signals”	mathworks.hism.hisl_0022	“hisl_0022: Data type selection for index signals”
“Check model file name”	mathworks.hism.hisl_0031	“hisl_0031: Model file names”
“Check if/elseif/else patterns in MATLAB Function blocks”	mathworks.hism.hisl_0022	“himl_0006: MATLAB code if / elseif / else patterns”
“Check switch statements in MATLAB Function blocks”	mathworks.hism.himl_0007	“himl_0007: MATLAB code switch / case / otherwise patterns”
“Check global variables in graphical functions”	mathworks.hism.hisl_0062	“hisl_0062: Global variables in graphical functions”
“Check for length of user-defined object names”	mathworks.hism.hisl_0063	“hisl_0063: Length of user-defined object names to improve MISRA C:2012 compliance”
“Check usage of Merge blocks”	mathworks.hism.hisl_0015	“hisl_0015: Usage of Merge blocks”
“Check usage of conditionally executed subsystems”	mathworks.hism.hisl_0012	“hisl_0012: Usage of conditionally executed subsystems”
“Check usage of standardized MATLAB function headers”	mathworks.hism.himl_0001	“himl_0001: Usage of standardized MATLAB function headers”
“Check usage of relational operators in MATLAB Function blocks”	mathworks.hism.himl_0008	“himl_0008: MATLAB code relational operator data types”
“Check usage of logical operators and functions in MATLAB Function blocks”	mathworks.hism.himl_0010	“himl_0010: MATLAB code with logical operators and functions”
“Check type and size of condition expressions”	mathworks.hism.himl_0011	“himl_0011: Data type and size of condition expressions”
“Check naming of ports in Stateflow charts”	mathworks.hism.hisf_0016	“hisf_0016: Stateflow port names”
“Check scoping of Stateflow data objects”	mathworks.hism.hisf_0017	“hisf_0017: Stateflow data object scoping”
“Check usage of Gain blocks”	mathworks.hism.hisl_0066	“hisl_0066: Usage of Gain blocks”
“Check for divide-by-zero calculations”	mathworks.hism.hisl_0067	“hisl_0067: Protect against divide-by-zero calculations”

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
"Check data type of loop control variables"	mathworks.hism.hisl_0102	"hisl_0102: Data type of loop control variables to improve MISRA C:2012 compliance"
"Check configuration parameters for MISRA C:2012"	mathworks.misra.CodeGenSettings	"hisl_0060: Configuration parameters that improve MISRA C:2012 compliance"
"Check for blocks not recommended for C/C++ production code deployment" "Check for blocks not recommended for MISRA C:2012"	mathworks.hism.hisl_0020 mathworks.misra.BlkSupport	"hisl_0020: Blocks not recommended for MISRA C:2012 compliance"
"Check safety-related optimization settings for specified minimum and maximum values"	mathworks.hism.hisl_0056	"hisl_0056: Configuration Parameters > Code Generation > Optimization > Optimize using the specified minimum and maximum values"
"Check usage of Reciprocal Sqrt blocks"	mathworks.hism.hisl_0028	"hisl_0028: Usage of Reciprocal Square Root blocks"
"Check safety-related settings for hardware implementation"	mathworks.hism.hisl_0071	"hisl_0071: Configuration Parameters > Hardware Implementation > Inconsistent hardware implementation settings"
"Check usage of recursions"	mathworks.hism.hisf_0004	"hisf_0004: Protect against recursive function calls to improve code compliance"
"Check MATLAB functions not supported for code generation"	mathworks.hism.himl_0012	"himl_0012: Usage of MATLAB functions for code generation"
"Metrics for generated code complexity"	mathworks.hism.himl_0013	"himl_0013: Limitation of built-in MATLAB Function complexity"
"Check for parameter tunability ignored for referenced models"	mathworks.hism.hisl_0072	"hisl_0072: Usage of tunable parameters for referenced models"
"Check usage of bit-shift operations"	mathworks.hism.hisl_0073	"hisl_0073: Usage of bit-shift operations"
"Check safety-related diagnostic settings for variants"	mathworks.hism.hisl_0074	"hisl_0074: Configuration Parameters > Diagnostics > Modeling issues related to variants"
"Check for disabled and parameterized library links"	mathworks.hism.hisl_0075	"hisl_0075: Usage of library links"

Model Advisor Check	Check ID	High-Integrity System Modeling Guidelines
"Check for unreachable and dead code"	mathworks.hism.hisl_0101	"hisl_0101: Avoid operations that result in dead logic to improve code compliance"
"Check for root Outports with missing properties"	mathworks.hism.hisl_0077	"hisl_0077: Outport interface definition"

See Also

- "Check Your Model Using the Model Advisor"
- "High-Integrity System Modeling"

See Also

Related Examples

- "Run Model Advisor Checks and Review Results" on page 3-4
- "Assess Requirements-Based Testing for ISO 26262" on page 5-102

Model Advisor Checks for MISRA C:2012 Coding Standards

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

Model Advisor Checks for MISRA C:2012 Coding Standards

These Model Advisor checks improve the likelihood of generating code that complies with MISRA C:2012 coding standards.

Execution of these checks requires either Embedded Coder or Simulink Check.

Model Advisor Check	Check ID
"Check usage of Assignment blocks"	mathworks.misra.AssignmentBlocks
"Check for blocks not recommended for MISRA C:2012"	mathworks.misra.BlkSupport
"Check for blocks not recommended for C/C++ production code deployment"	mathworks.codegen.PCGSupport
"Check for unsupported block names"	mathworks.misra.BlockNames
"Check configuration parameters for MISRA C:2012"	mathworks.misra.CodeGenSettings
"Check for equality and inequality operations on floating-point values"	mathworks.misra.CompareFloatEquality
"Check for bitwise operations on signed integers"	mathworks.misra.CompliantCGIRConstructions
"Check for recursive function calls"	mathworks.misra.RecursionCompliance
"Check for switch case expressions without a default case"	mathworks.misra.SwitchDefault
"Check for missing error ports for AUTOSAR receiver interfaces"	mathworks.misra.AutosarReceiverInterface
"Check for missing const qualifiers in model functions"	mathworks.misra.ModelFunctionInterface
"Check integer word length"	mathworks.misra.IntegerWordLengths
"Check bus object names that are used as bus element names"	mathworks.misra.BusElementNames

See Also

Related Examples

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

Model Advisor Checks for CERT C, SWE, and ISO/IEC TS 17961 Secure Coding Standards

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

Model Advisor Checks for CERT C, SWE, and ISO/IEC TS 17961 Coding Standards

These Model Advisor checks improve the likelihood of generating code that complies with CERT C, CWE, and ISO/IEC TS 17961 (Embedded Coder) secure coding standards.

Unless otherwise noted, execution of these checks requires either Embedded Coder or Simulink Check.

Secure Coding Standards			Model Advisor Check	Check ID
CERT C	CWE	ISO/IEC TS 17961		
✓	✓	✓	"Check configuration parameters for secure coding standards"	mathworks.security.CodeGenSettings
✓	✓	✓	"Check for blocks not recommended for C/C++ production code deployment"	mathworks.codegen.PCGSupport
✓	✓	✓	"Check for blocks not recommended for secure coding standards"	mathworks.security.BlockSupport
✓	✓	✓	"Check usage of Assignment blocks"	mathworks.misra.AssignmentBlocks
✓	✓	✓	"Check for switch case expressions without a default case"	mathworks.misra.SwitchDefault
✓	✓	✓	"Check for bitwise operations on signed integers"	mathworks.misra.CompliantCGIRConstructions
✓	✓	✓	"Check for equality and inequality operations on floating-point values"	mathworks.misra.CompareFloatEquality
✓	✓	✓	"Check integer word length"	mathworks.misra.IntegerWordLengths
✓	✓	✓	"Detect Dead Logic" This check requires Simulink Design Verifier.	mathworks.sldv.deadlogic

Secure Coding Standards			Model Advisor Check	Check ID
CERT C	CWE	ISO/IEC TS 17961		
✓	✓	✓	“Detect Integer Overflow” This check requires Simulink Design Verifier.	mathworks.sldv.integeroverflow
✓	✓	✓	“Detect Division by Zero” This check requires Simulink Design Verifier.	mathworks.sldv.divbyzero
✓	✓	✓	“Detect Out Of Bound Array Access” This check requires Simulink Design Verifier.	mathworks.sldv.arraybounds
✓	✓	✓	“Detect Specified Minimum and Maximum Value Violations” This check requires Simulink Design Verifier.	mathworks.sldv.minmax
✓	N/A	N/A	“Check configuration parameters for MISRA C:2012”	mathworks.misra.CodeGenSettings
✓	N/A	N/A	“Check usage of Abs blocks” This check requires Simulink Check.	mathworks.hism.hisl_0001
✓	N/A	N/A	“Check usage of remainder and reciprocal operations” This check requires Simulink Design Verifier.	mathworks.sldv.hism.hisl_0002
✓	N/A	N/A	“Check usage of square root operations” This check requires Simulink Design Verifier.	mathworks.hism.hisl_0003
✓	N/A	N/A	“Check usage of While Iterator blocks” This check requires Simulink Check.	mathworks.hism.hisl_0006
✓	N/A	N/A	“Check data types for blocks with index signals” This check requires Simulink Check.	mathworks.hism.hisl_0022
✓	N/A	N/A	“Check usage of Reciprocal Sqrt blocks” This check requires Simulink Design Verifier.	mathworks.hism.hisl_0028

Secure Coding Standards			Model Advisor Check	Check ID
CERT C	CWE	ISO/IEC TS 17961		
✓	N/A	N/A	“Check global variables in graphical functions” This check requires Simulink Check and Stateflow.	mathworks.hism.hisl_0062
✓	N/A	N/A	“Check usage of bit-shift operations” This check requires Simulink Check.	mathworks.hism.hisl_0073
✓	N/A	N/A	“Check safety-related optimization settings for data type conversions” This check requires Simulink Check.	mathworks.hism.hisl_0053
✓	N/A	N/A	“Check safety-related optimization settings for division arithmetic exceptions” This check requires Simulink Check.	mathworks.hism.hisl_0054
✓	N/A	N/A	“Check model file name” This check requires Simulink Check.	mathworks.hism.hisl_0031
✓	N/A	N/A	“Check model object names” This check requires Simulink Check.	mathworks.hism.hisl_0032

See Also

- “Check Your Model Using the Model Advisor”

See Also

Related Examples

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

Model Advisor Checks for Requirements Links

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

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

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

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

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

See Also

Related Examples

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

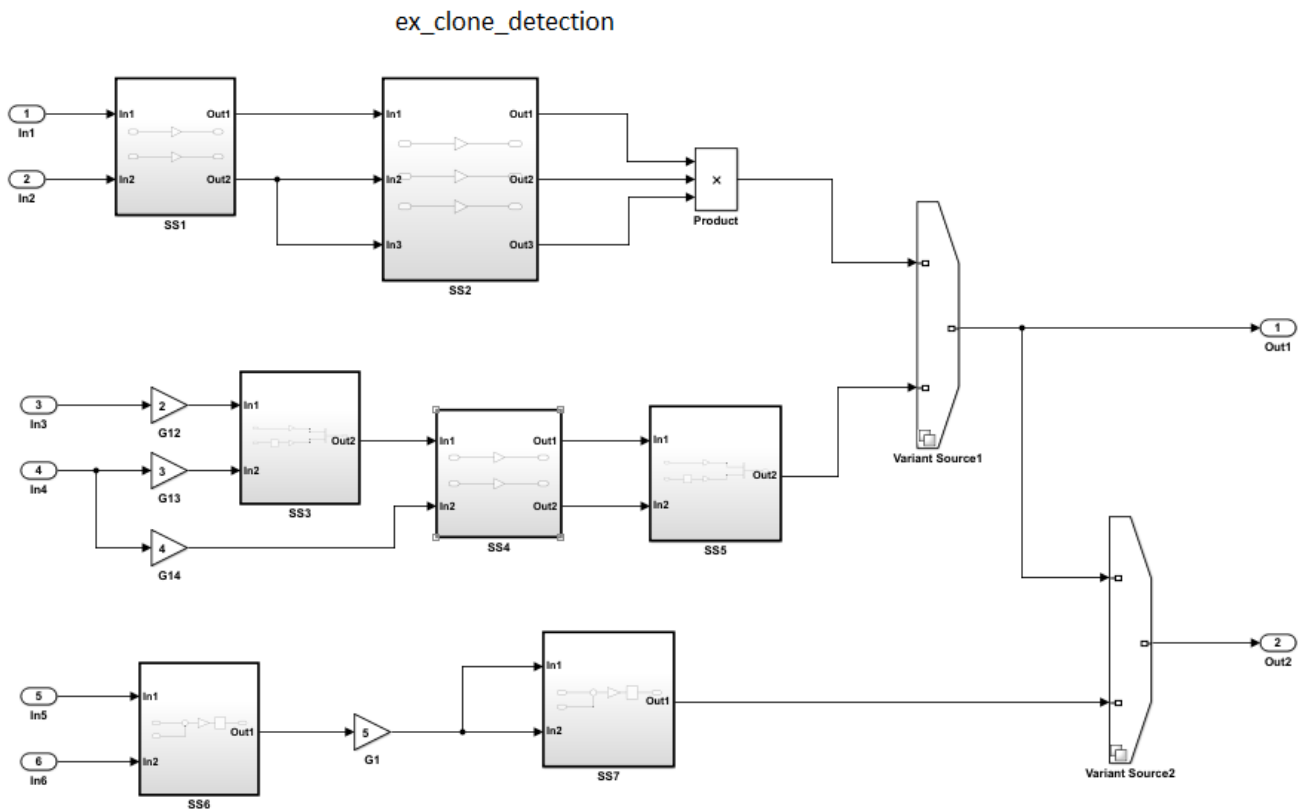
Replace Exact Clones with Subsystem Reference

Exact clones are modeling patterns that have identical block types, connections, and parameter values. The Clone Detector identifies these clones across referenced model boundaries. You can then reuse components by replacing exact clones with library links and Subsystem Reference blocks. To replace exact clones with library links, see “Enable Component Reuse by Using Clone Detection” on page 3-29. This example demonstrates how to replace subsystem clones with “Subsystem Reference” blocks.

Identify Exact Clones

- 1 Open the model `ex_detect_clones`.

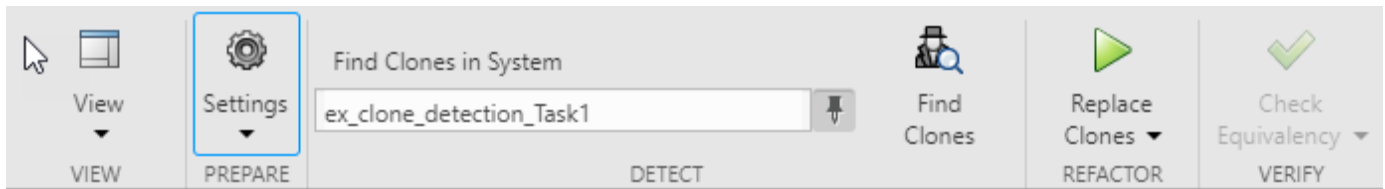
```
openExample('ex_detect_clones');
```



Copyright 2017 The MathWorks Inc.

- 2 Save the model to your working folder. The model must be open to access the app.
- 3 On the **Apps** tab, click **Clone Detector**. Alternatively, on the MATLAB command line enter:


```
clonedetection("ex_detect_clones")
```
- 4 The app opens the **Clone Detector** tab. This example takes you through each section.



Set Up Panes for Clone Detection

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

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

Set the Parameters for Clone Detection

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

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

Identify Clones in the Model

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

Analyze the Clone Detection Results

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

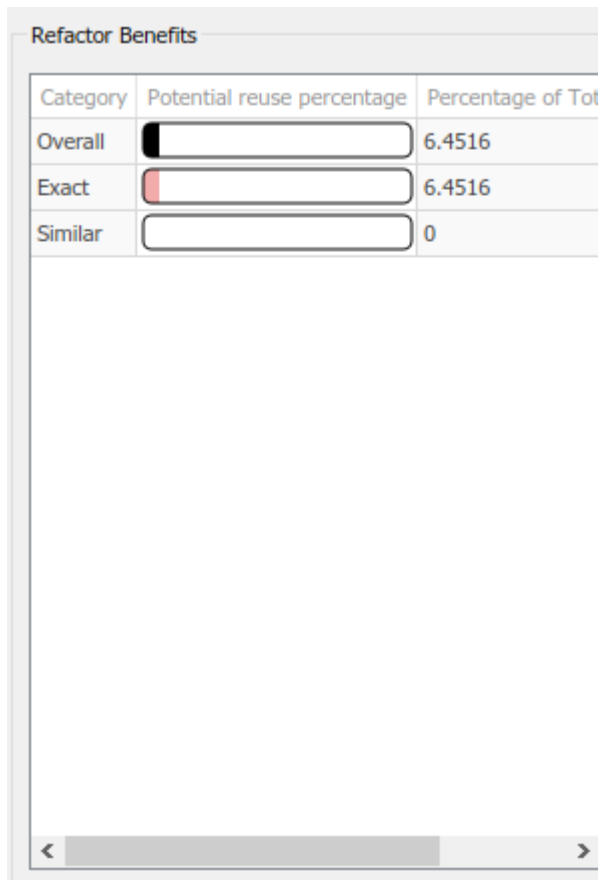
- 1 In the **Clone Detection Results and Actions** panel, on the **Clone Results** tab, a clone group Exact Clone Group 1 is displayed.
- 2 Click the > symbol next to Exact Clone Group 1 to see all of the subsystems that are exact clones and the number of blocks per clone.
- 3 In the **Clone Detection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

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

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

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

Refactoring the model reduces 6.4516% of the model reuse.

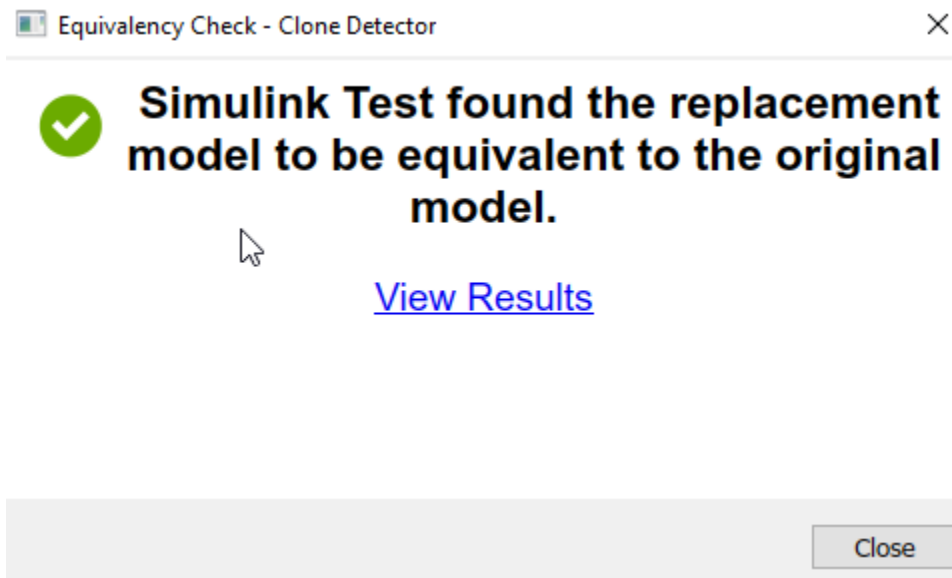


Replace Clones

- 1 In the **Clone Detector** tab, click **Replace Clones**. The exact clones are replaced with Subsystem Reference blocks. You can see the `.slx` files of the subsystem references in your working folder.
- 2 You can restore the model to its original configuration in the **Clone Detection Results and Actions** pane. Open the **Logs** tab, click the hyperlink for the version you want to restore, and click **Restore**.

Check the Equivalency of the Model

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



See Also

Related Examples

- "Generate Reusable Code from Library Subsystems Shared Across Models" (Simulink Coder)
- Clone Detector
- "Enable Component Reuse by Using Clone Detection" on page 3-29

Detect and Replace Subsystem Clones Programmatically

Clones are modeling patterns that have identical block types and connections. You can refactor your model by creating library blocks from subsystem clones and replacing the clones with links to those library blocks, which enable you to reuse components. For more information about clones, see [Enable Component Reuse by Using Clone Detection](#) on page 3-29.

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

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

In this example, you learn how to use:

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

Identify Clones in a Model

- 1 Open the model `ex_detect_clones_B`.

```
openExample('ex_detect_clones_B');
```

Save the model in the current working directory.

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

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

cloneResults =
  Results with properties:
    Clones: [1x1 struct]
  ExceptionLog: ''
```

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

```
cloneResults.Clones

ans =
  struct with fields:
    Summary: [1x1 struct]
    CloneGroups: [1x2 struct]
```

- 3 View the `Summary` field.

```
cloneResults.Clones.Summary
```

```
ans =
  struct with fields:
    CloneGroups: 2
    SimilarClones: 5
    ExactClones: 0
    Clones: 5
    PotentialReusePercentage: [1x1 struct]
```

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

- 4** View the `CloneGroups` field.

```
cloneResults.Clones.CloneGroups
```

```
ans =
  1x2 struct array with fields:
    Name
    Summary
    CloneList
```

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

- 5** View the details of first clone group.

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

```
ans =
  struct with fields:
    Name: 'Similar Clone Group 1'
    Summary: [1x1 struct]
    CloneList: {3x1 cell}
```

- 6** View the `Summary`.

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

```
ans =
  struct with fields:
    ParameterDifferences: [1x1 struct]
    Clones: 3
    BlocksPerClone: 8
    CloneType: 'Similar'
    BlockDifference: 1
```

- 7** View the `CloneList` of the first `CloneGroup`.

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

```
ans =
  3x1 cell array
    {'ex_detect_clones_B/Subsystem1'}
    {'ex_detect_clones_B/Subsystem2'}
    {'ex_detect_clones_B/Subsystem3'}
```

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

Replace Clones in a Model

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

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

cloneReplacementResults =
  ReplacementResults with properties:
    ReplacedClones: [1x5 struct]
    ExcludedClones: {}
```

The `cloneReplacementResults` object includes two properties, `ReplacedClones` and `ExcludedClones`.

- 2 View the contents of `ReplacedClones` property.

```
cloneReplacementResults.ReplacedClones

ans =
  1x5 struct array with fields:
    Name
    ReferenceSubsystem
```

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

- 3 View the list of replaced subsystem clones.

```
struct2table(cloneReplacementResults.ReplacedClones)

ans =
  5x2 table
           Name                               ReferenceSubsystem
   _____  _____
   {'ex_detect_clones_B/Subsystem1'}  {'newLibraryFile/Subsystem1'}
   {'ex_detect_clones_B/Subsystem2'}  {'newLibraryFile/Subsystem1'}
   {'ex_detect_clones_B/Subsystem3'}  {'newLibraryFile/Subsystem1'}
   {'ex_detect_clones_B/SS3'}         {'newLibraryFile/SS1'}
   {'ex_detect_clones_B/SS4'}         {'newLibraryFile/SS1'}
```

Identify Clones Using Subsystem Reference Blocks

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

```
ex_detect_clones_E
clones_library
```

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

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

cloneDetectionSettings =
  Settings with properties:
    IgnoreSignalName: 0
    IgnoreBlockProperty: 0
    ExcludeModelReferences: 0
    ExcludeLibraryLinks: 0
    ExcludeInactiveRegions: 0
```

```

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

```

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

```
cloneDetectionSettings.ParamDifferenceThreshold = 0
```

```
cloneDetectionSettings =
```

```
Settings with properties:
```

```

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

```

A value of 0 indicates the subsystems must be identical.

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

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

```
cloneDetectionSettings =
```

```
Settings with properties:
```

```

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

```

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

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

```
cloneResults =
```

```
Results with properties:
```

```
Clones: [1x1 struct]
```

```
cloneResults.Clones.Summary
```

```
ans =
```

```
struct with fields:
```

```

CloneGroups: 2
SimilarClones: 5
ExactClones: 0
Clones: 5
PotentialReusePercentage: [1x1 struct]

```

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

- 6 View the details of first CloneGroup.

```

cloneResults.Clones.CloneGroups(1)

ans =
  struct with fields:
    Name: 'clones_library/SSL1'
    Summary: [1x1 struct]
    CloneList: {3x1 cell}

```

Replace Clones with Conditions

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

```

cloneReplacementConfig = Simulink.CloneDetection.ReplacementConfig()

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

```

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

```

cloneReplacementConfig.addCloneToIgnoreList('ex_detect_clones_E/Subsystem1')

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

```

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

```

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

cloneReplacementResults =
  ReplacementResults with properties:
    ReplacedClones: [1x4 struct]
    ExcludedClones: [1x1 struct]

```

- 4 View the ReplacedClones property.

```

struct2table(cloneReplacementResults.ReplacedClones)

ans =
  4x2 table
           Name                               ReferenceSubsystem
   _____|_____
   {'ex_detect_clones_E/SS3' }               {'clones_library/SSL1'}
   {'ex_detect_clones_E/SS4' }               {'clones_library/SSL1'}

```

```
{'ex_detect_clones_E/Subsystem1'}    {'clones_library/SSL2'}
{'ex_detect_clones_E/Subsystem2'}    {'clones_library/SSL2'}
```

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

- 5 View the ExcludedClones property.

```
struct2table(cloneReplacementResults.ExcludedClones)
```

```
ans =
    1x2 table
           Name                ReferenceSubsystem
    _____
    {'ex_detect_clones_E/Subsystem1'}    {'unselected'}
```

Check the Equivalency of the Model

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

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

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

checkEquiResults =
    EquivalencyCheckResults with properties:
        List: [1x1 struct]
```

View the check equivalence results.

```
checkEquiResults.List

ans =
    struct with fields:
        IsEquivalencyCheckPassed: 1
        OriginalModel: 'm2m_ex_detect_clones_E/snapshot_2020_12_21_16_35_06_ex_detect_clones_E.slx'
        UpdatedModel: 'ex_detect_clones_E.slx'
```

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

See Also

Related Examples

- Clone Detector
- “Enable Component Reuse by Using Clone Detection” on page 3-29

Find Clones Across the Model

Clones are modeling patterns that have identical block types and connections. You can refactor your model by creating library blocks from these clone patterns and replacing the clones with links to the library blocks, which enable you to reuse the components. For more information about clones, see [Enable Component Reuse by Using Clone Detection](#) on page 3-29.

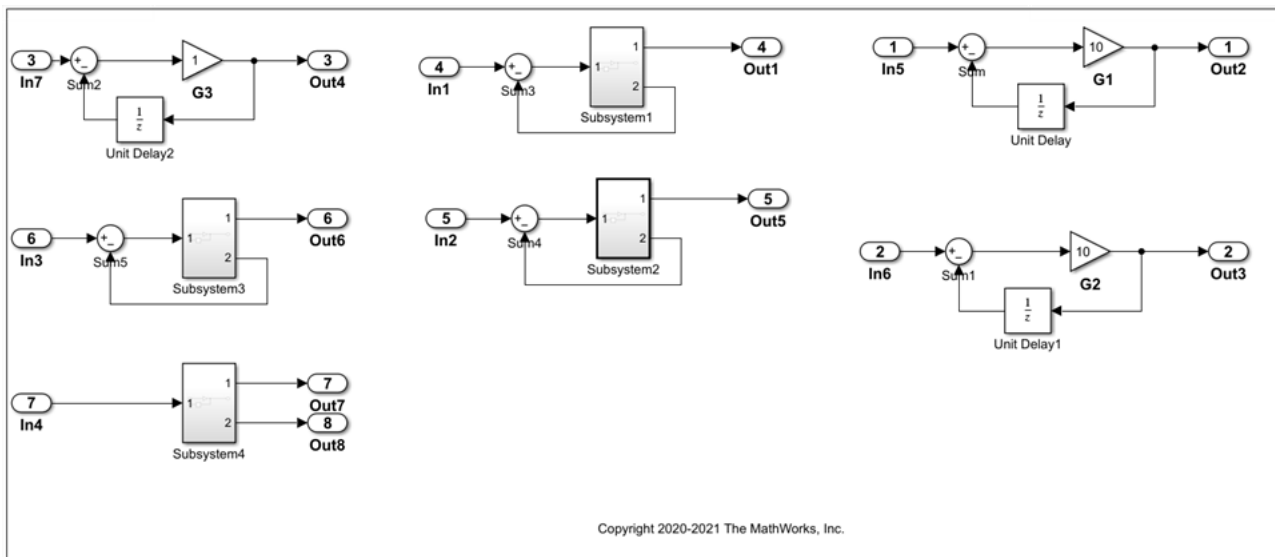
You can search for clones in a subsystem or anywhere across the model.

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

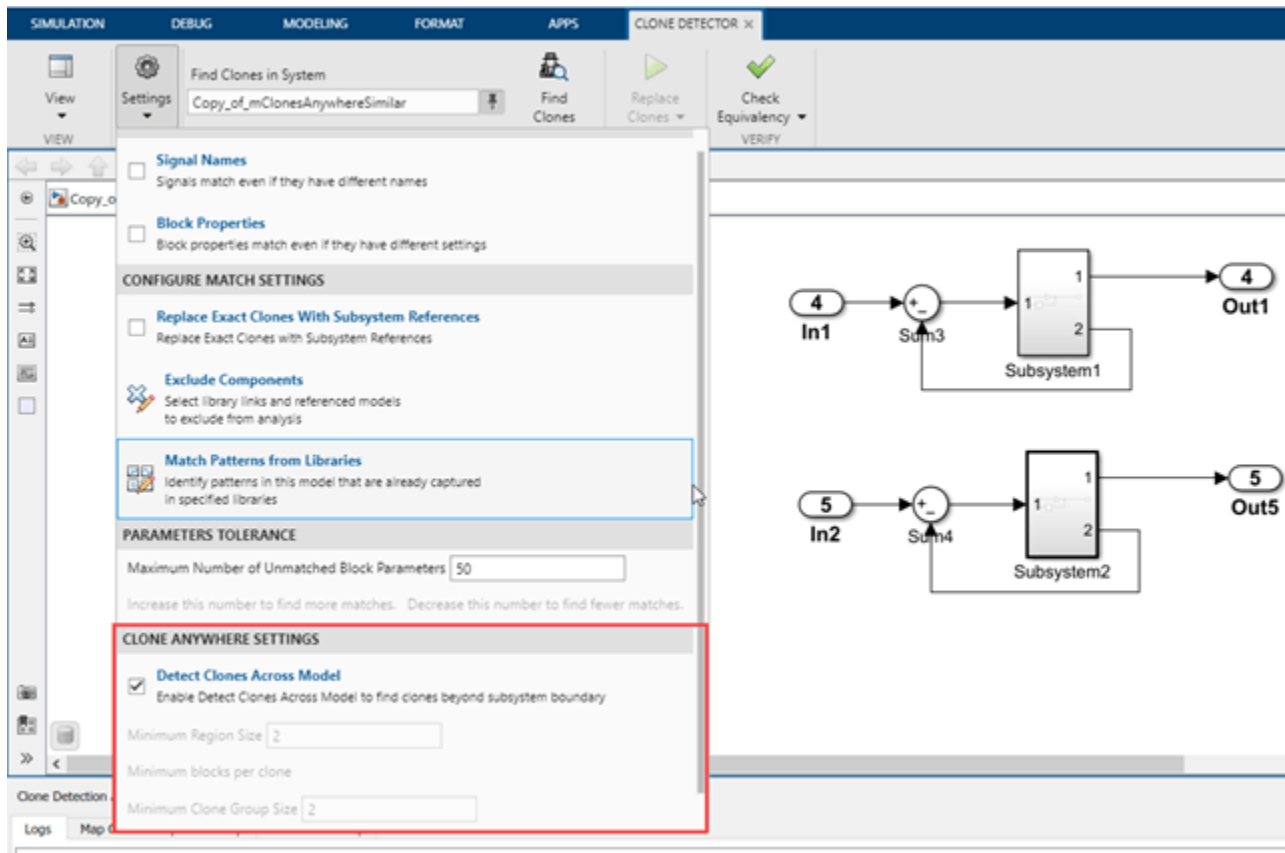
This example shows how to use the Clone Detector app and APIs to identify clones anywhere across the model, and then replace them with links to library blocks.

Identify Clones Across the Model by Using the Clone Detector App

This example shows how to identify clones across the model by using the **Clone Detector**.

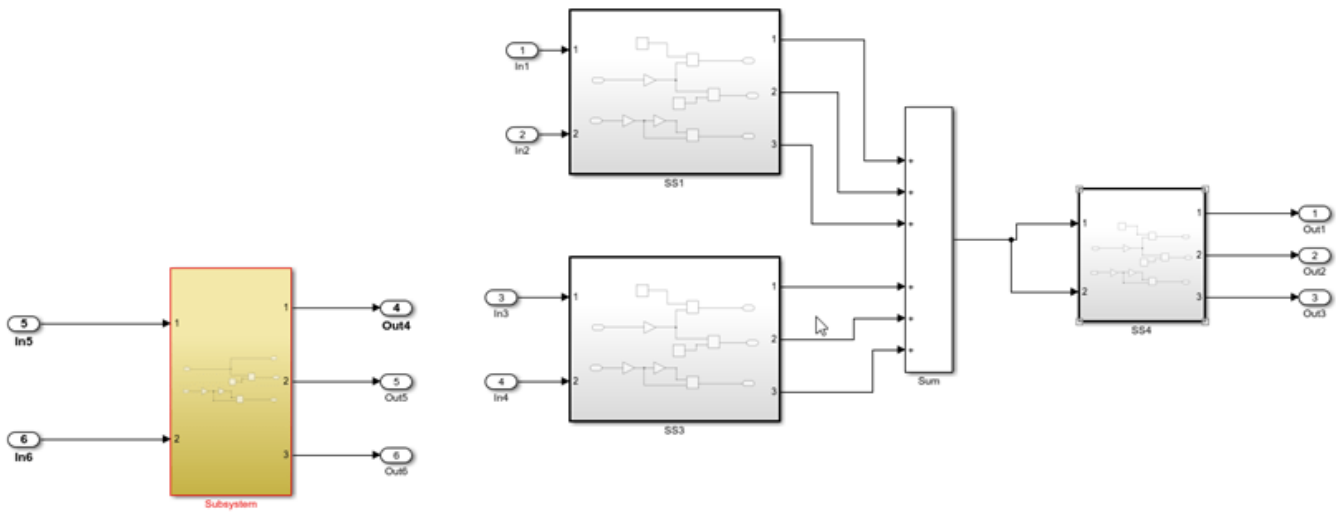


- Open the model `similar_clones_model`.
- Save the model in the current working directory.
- On the **Apps** tab, click **Clone Detector**. Alternatively, at the MATLAB command line, enter: `clonedetection("similar_clones_model")`
- To set up the parameters for clone detection, click **Settings**. Under **Clone anywhere settings**, click **Detect Clones Across Model**.



Minimum Region Size and **Minimum Clone Group Size** are set to 2 by default. The **Minimum Region Size** parameter represents the minimum blocks needed per clone region and the **Minimum Clone Group Size** parameter represents the minimum clone occurrences needed to define it as a clone group.

- Click **Find Clones** to identify clones.
- The app highlights the clones. Exact clones are highlighted in red and similar clones are highlighted in blue.



Copyright 2017 The MathWorks Inc.

The clones highlighted in this example include clones identified in and outside of subsystems and a Simulink block connected to a subsystem.

- Click **Replace Clones**.

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

Identify Clones Programmatically

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

```
cloneDetectionSettings = Simulink.CloneDetection.Settings()
cloneDetectionSettings =
    IgnoreSignalName: 0
    IgnoreBlockProperty: 0
    ExcludeModelReferences: 0
    ExcludeLibraryLinks: 0
    FindClonesRecursivelyInFolders: 1
    ParamDifferenceThreshold: 50
    ReplaceExactClonesWithSubsystemReference: 0
    Libraries: {}
    Folders: {}
    DetectClonesAcrossModel: 0
    ExcludeInactiveRegions: 0
```

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

```
cloneDetectionSettings.DetectClonesAcrossModel = 1
cloneDetectionSettings =
    IgnoreSignalName: 0
    IgnoreBlockProperty: 0
```

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

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

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

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

```
cloneResults =
```

```
    Clones: [1x1 struct]
ExceptionLog: ''
```

```
cloneResults.Clones =
```

```
Results with properties:
```

```
Summary: [1x1 struct]
CloneGroups: [1x2 struct]
```

For more details on the clone detection APIs, see “Detect and Replace Subsystem Clones Programmatically” on page 3-83.

See Also

Related Examples

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

Detect Clones Programmatically on Multiple Models Across Different Folders

This example shows how to programmatically detect clones across multiple models located in different folders. For more information about the clone detection APIs, see “Detect and Replace Subsystem Clones Programmatically” on page 3-83.

This example demonstrates how to use the clone detection APIs to identify clones in six Simulink® models present in a folder. Replacement of clones is not possible using this example workflow. To replace clones programmatically in multiple models, see “Detect and Replace Clones Programmatically in a Loop on Multiple Models” on page 3-95.

1. At the MATLAB® command line, enter:

```
openExample('ex_detect_clones_A')
openExample('ex_detect_clones_B')
openExample('ex_detect_clones_C')
openExample('ex_detect_clones_D')
openExample('ex_detect_clones_E')
```

Save the models to a writeable folder.

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

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

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

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

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

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

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

5. View the `Clones.Summary` field.

```
cloneResults.Clones.Summary
```

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

5. View the details of first clone group.

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

6. View the summary of first clone group.

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

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

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

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

See Also

Related Examples

- Clone Detector
- “Enable Component Reuse by Using Clone Detection” on page 3-29
- “Detect and Replace Subsystem Clones Programmatically” on page 3-83

Detect and Replace Clones Programmatically in a Loop on Multiple Models

This example shows how to programmatically detect and replace clones on a multiple models in a loop by operating on models individually. For more information about Clone Detection APIs, see “Detect and Replace Subsystem Clones Programmatically” on page 3-83.

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

1. Open the models: `ex_detect_clones_A` `ex_detect_clones_B` `ex_detect_clones_C` `ex_detect_clones_D` `ex_detect_clones_E` `clones_library` Save the models and library file in the current working directory.

2. Create an array to add the models to:

```
modelList = {};
```

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

```
modelList{end+1,1} = 'ex_detect_clones_A';
modelList{end+1,1} = 'ex_detect_clones_B';
modelList{end+1,1} = 'ex_detect_clones_C';
modelList{end+1,1} = 'ex_detect_clones_D';
modelList{end+1,1} = 'ex_detect_clones_E';
modelList{end+1,1} = 'ex_detect_clones_F';
```

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

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

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

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

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

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

You can access the results of `cloneResultsStorage`, `cloneReplacementStorage`, and `equivalencyCheckStorage` objects for individual models. For more details, see “Detect and Replace Subsystem Clones Programmatically” on page 3-83.

See Also

Related Examples

- Clone Detector
- “Enable Component Reuse by Using Clone Detection” on page 3-29
- “Detect Clones Programmatically on Multiple Models Across Different Folders” on page 3-93

Running Clone Detection Custom Script in a Project

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

This example shows how to create and run a custom script to automate the clone detection across all the models in a Simulink project. For more information on clone detection, see “Enable Component Reuse by Using Clone Detection” on page 3-29.

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

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

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

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

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

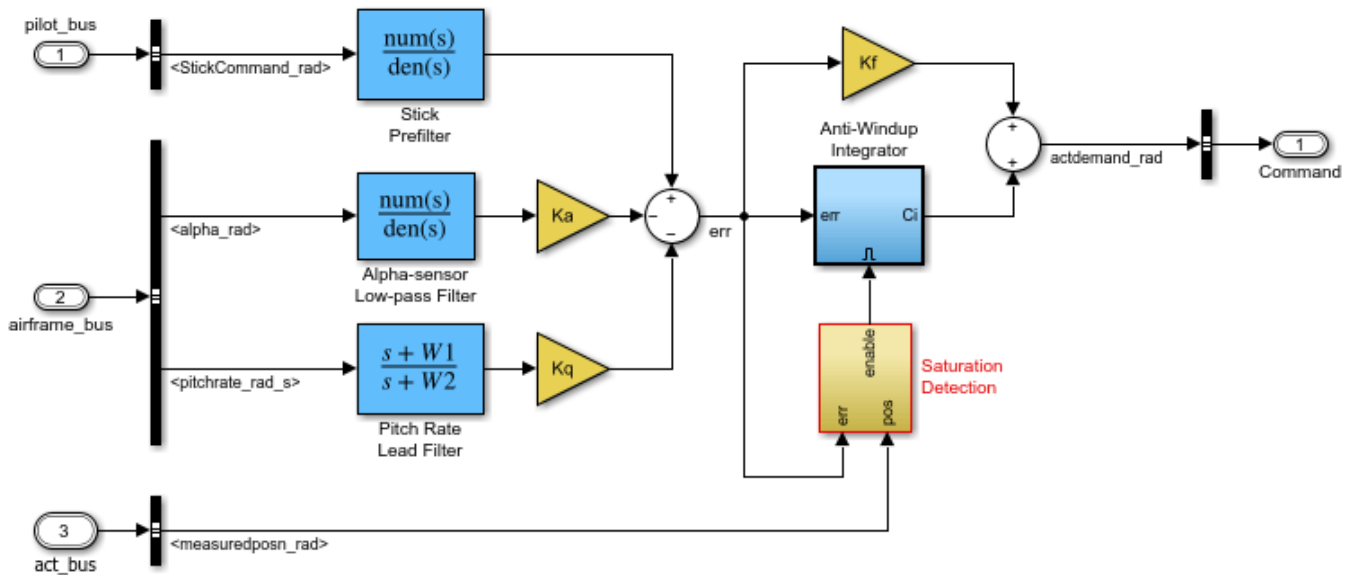
```
cloneDetectionSettings = Simulink.CloneDetection.Settings();  
cloneDetectionSettings.Folders = project.RootFolder;
```

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

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

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

```
Simulink.CloneDetection.highlightClone(cloneResults, 'AnalogControl/Saturation Detection');
```



Copyright 1990-2018 The MathWorks, Inc.

The clone results is an object of `Simulink.CloneDetection.Results` class. For detailed information on how to see the clone results, see “Detect and Replace Subsystem Clones Programmatically” on page 3-83.

See Also

Related Examples

- Clone Detector
- “Enable Component Reuse by Using Clone Detection” on page 3-29
- “Detect Clones Programmatically on Multiple Models Across Different Folders” on page 3-93

Run Custom Model Advisor Checks on Architecture Models

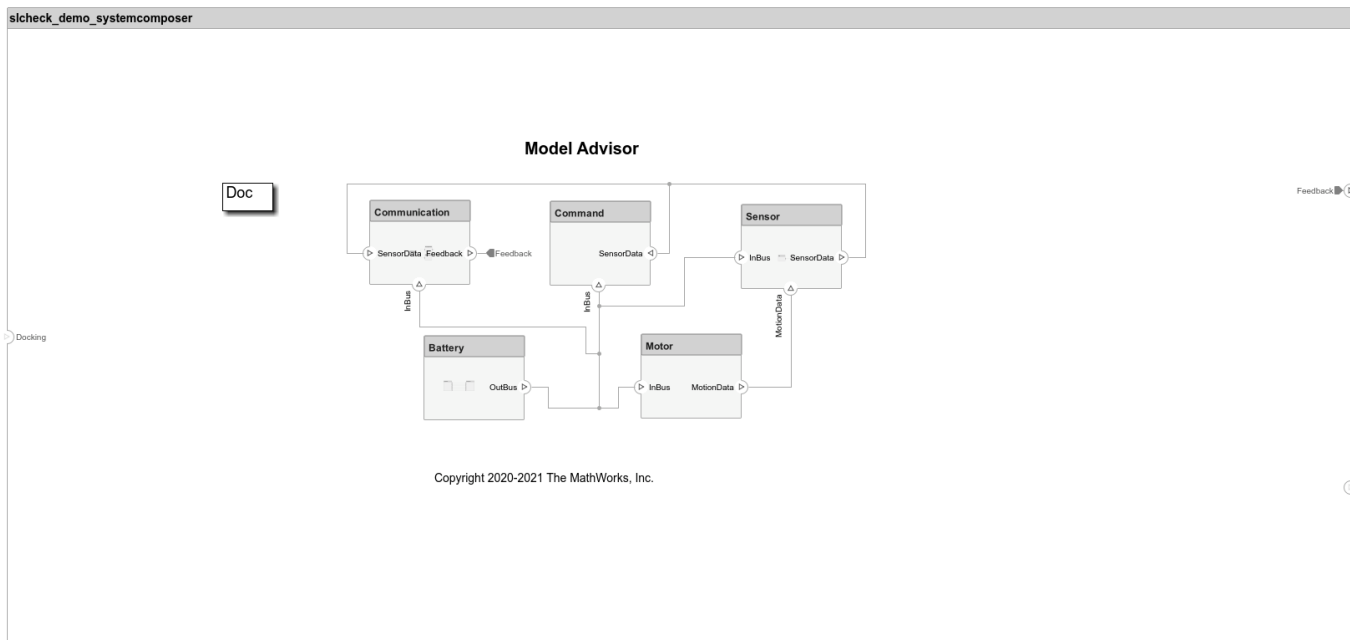
Use Custom Model Advisor checks on Architecture models (System Composer).

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

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

2. Open the architecture model.

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



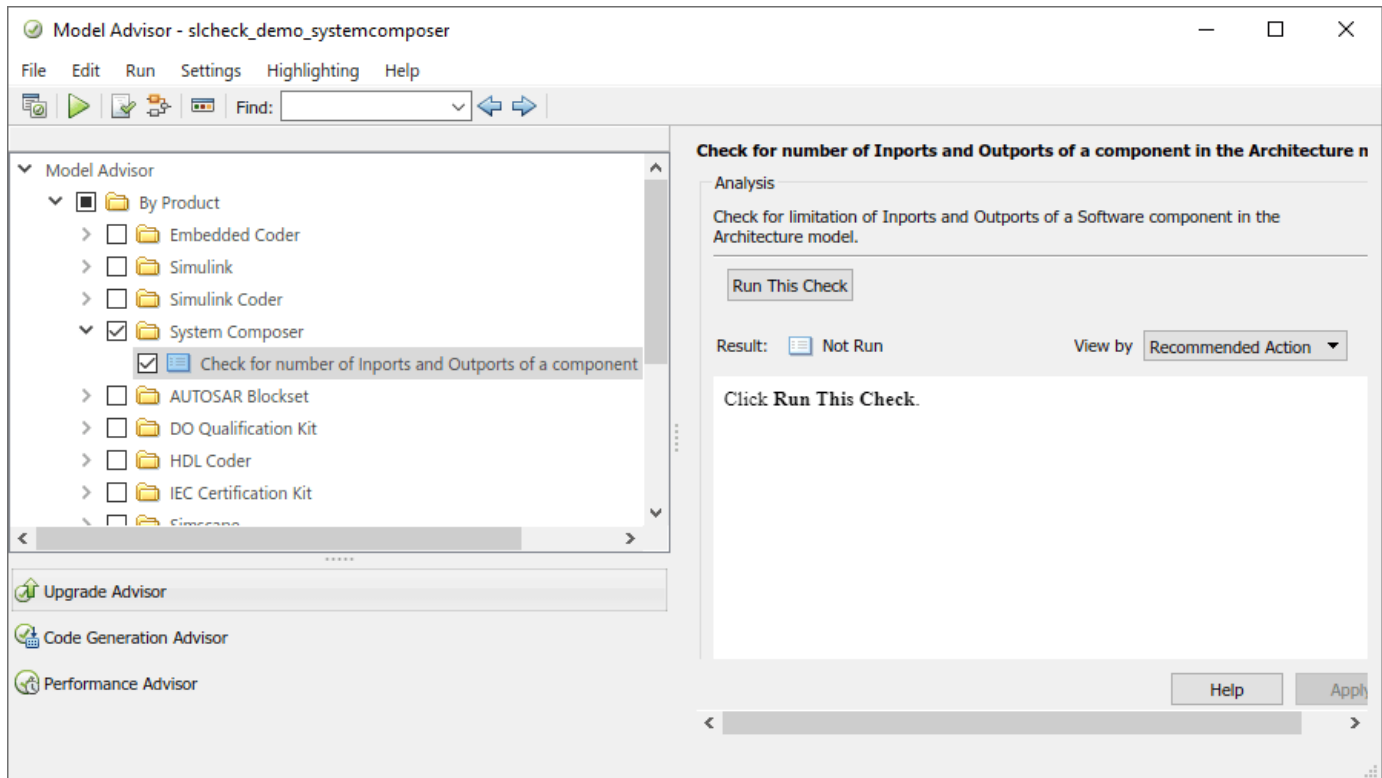
3. From the **Modeling** tab, open **Model Advisor**. You can also open the Model Advisor by typing this command at the MATLAB command prompt:

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

```
Updating Model Advisor cache...
```

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

4. On the Model Advisor window, click **By Product** > **System Composer** and Select Check number of Inports and Outports of a component check.



5. Right-click the check and select **Run This Check**. Model Advisor runs the selected check on the architecture model and displays the results.

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

```
edit sl_customization.m
```

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

```

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

```


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

See Also

Related Examples

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

Justify Model Advisor Violations from Check Analysis

When you run checks using the Model Advisor app or the `ModelAdvisor.run` function, Model Advisor analyzes your design for adherence to modeling guidelines. The results from analysis show which checks pass, fail, are incomplete, are justified, or return a warning. Model Advisor checks that fail or return a warning are considered *violations* because the results do not comply with the modeling guidelines checked by the Model Advisor check.

If you review a violation and consider the violation to either not be relevant or not feasible for your design, you can justify the result. When you *justify* a result, you provide a rationale for why you are allowing the violation to exist in your design. Justified checks have a check result status of **Justified** in Model Advisor and the Model Advisor Report.

Justifications allow you to add a rationale for violations in the Model Advisor analysis results. If you want Model Advisor to not analyze specific blocks, use exclusions instead. Exclusions can save time during model development and verification, since Model Advisor does not analyze what you exclude. For more information, see “Exclude Blocks from the Model Advisor Check Analysis” on page 3-9.

This example shows how to:

- Justify a violation for check results in Model Advisor.
- Justify an edit-time violation from the Simulink® canvas.
- Load, view, edit, and delete justifications in Model Advisor.
- Create different justifications files for different sets of justifications.

Justify Violation Using Model Advisor

After you run a Model Advisor check, you can justify violations in the results in Model Advisor. For information about how to run Model Advisor checks, see “Check Model Compliance by Using the Model Advisor” on page 3-2.

1. Open the example model `sldemo_fuelsys`.

```
open_system("sldemo_fuelsys")
```

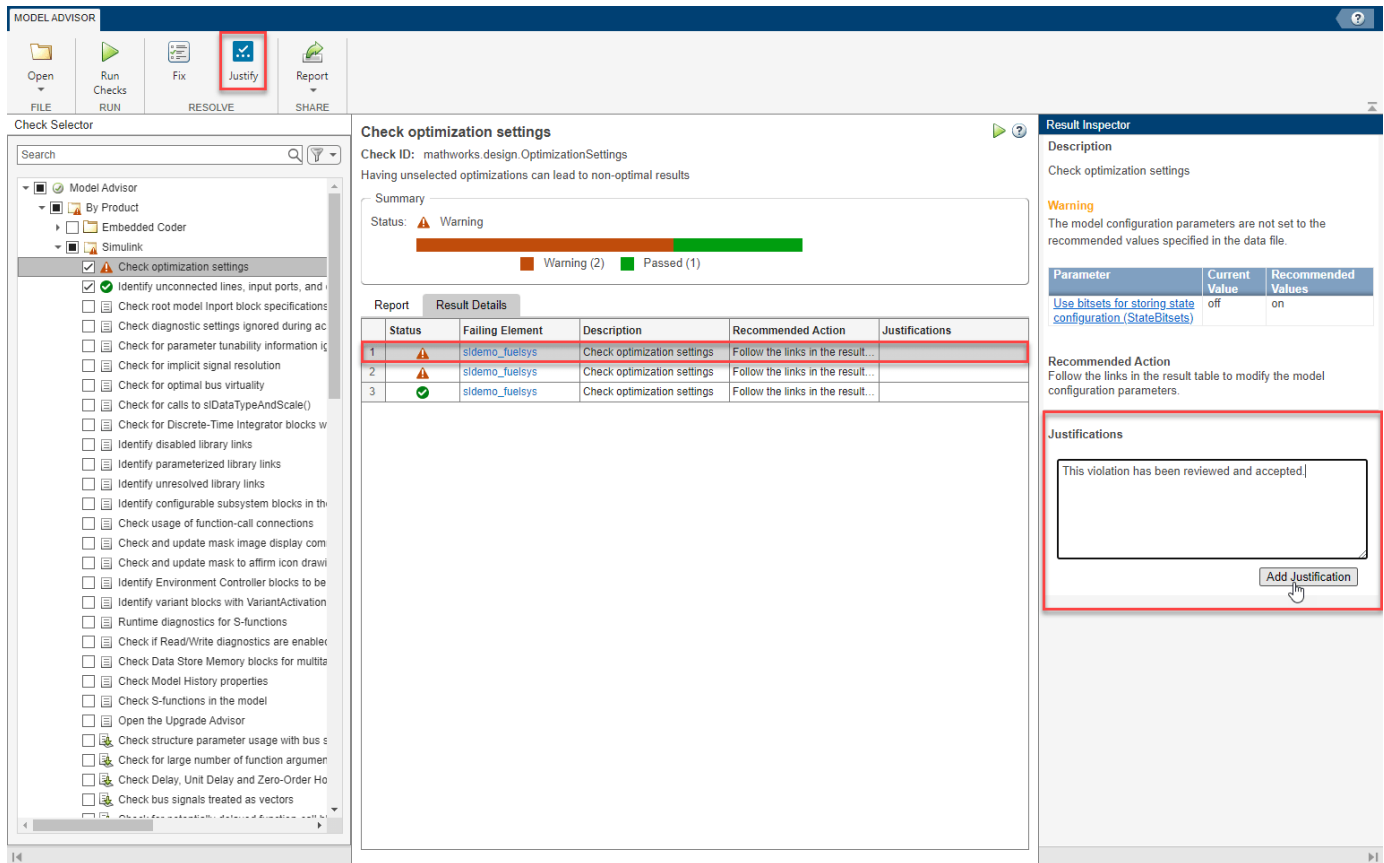
2. Open Model Advisor. In the Simulink editor, click the **Modeling** tab and select **Model Advisor**. When the **System Selector** dialog box opens, click **OK** to analyze the model.

3. In the **Check Selector** pane, select the checks that you want to run on the model. For this example, navigate to the folder **Model Advisor > By Product > Simulink** and select the check named **Check optimization settings**.

4. Run the check by clicking **Run Checks** in the toolbar. The Model Advisor results show a warning for the check **Check optimization settings**.

5. In the toolbar, click **Justify**. The **Result Inspector** pane opens, and the **Result Details** tab shows the results for each violation of the check **Check optimization settings**.

6. In the **Result Inspector** pane, enter a rationale in the **Justifications** field and click **Add Justification**. The justification applies to the violation that you select in the **Result Details** tab. By default, Model Advisor automatically selects the first result.



Since you are justifying a check for the first time, Model Advisor prompts you with a Save As dialog box that allows you to create a justifications file with your specified file name and in your specified directory.

7. Specify the file name and directory where you want to save the justifications file. By default, the file name is `modelname_justifications.json` and the directory is your current working directory in MATLAB®.

When you click **Save**, a banner on the Simulink canvas shows the path to the justifications file that you saved, and Model Advisor shows the result as justified.

Check optimization settings ▶ ?

Check ID: mathworks.design.OptimizationSettings

Having unselected optimizations can lead to non-optimal results

Summary

Status: ⚠ Warning

■ Warning (1)
 ■ Justified (1)
 ■ Passed (1)

Report **Result Details**

	Status	Failing Element	Description	Recommended Action	Justifications
1	☑	sldemo_fuelsys	Check optimization settings	Follow the links in the result...	This violation has been revi...
2	⚠	sldemo_fuelsys	Check optimization settings	Follow the links in the result...	
3	✔	sldemo_fuelsys	Check optimization settings	Follow the links in the result...	

Justify Violation During Edit-Time Checking

After you enable edit-time checks for a model, you can justify violations directly from the Simulink canvas. For information on edit-time checks, see “Check Model Compliance Using Edit-Time Checking” on page 3-6.

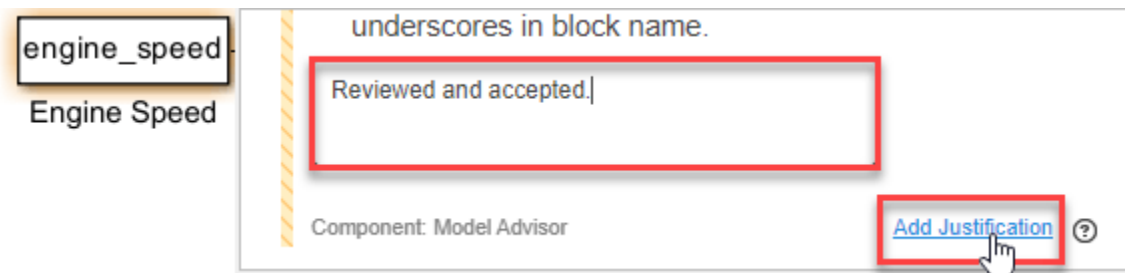
1. Enable edit-time checking for the model. In the Simulink editor, click the **Modeling** tab and select **Model Advisor > Edit-Time Checks**. When the Configuration Parameters dialog box opens, select the check box for **Edit-Time Checks** and click **OK**.

In the Simulink canvas, Model Advisor highlights blocks in the model that violate checks in the default Model Advisor configuration.

2. Point to a highlighted block and click the icon above the block. For this example, point to the Engine Speed block and click the warning icon ⚠.

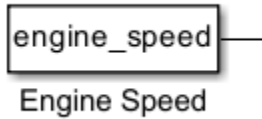
Model Advisor displays a violation summary and the title of the violated check or sub-check. The Engine Speed block violates the sub-check **Characters allowed for block names**.

3. Click **Suppress** and enter your rationale for the justification in the comment field.
4. Click **Add Justification**.



Since you already created a justifications file for the model, Model Advisor adds the justification to that file.

The violation no longer appears in the Simulink canvas.



If you justify other results, Model Advisor automatically adds those justifications to the current justifications file.

Manage Justifications

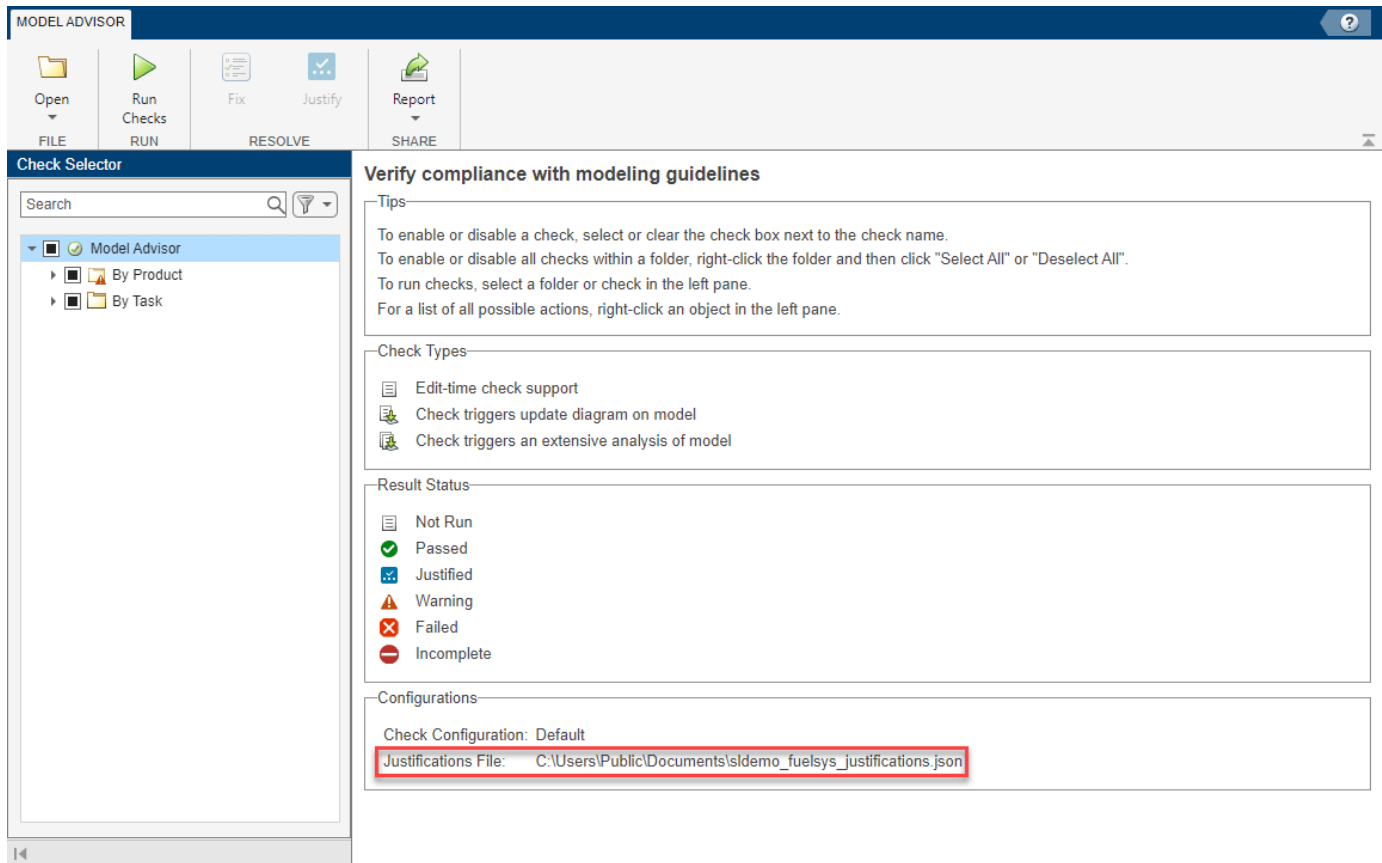
You can load, view, edit, and delete your justifications by using Model Advisor.

Load Justifications

After you save a justifications file for a model, Model Advisor automatically loads that justifications file the next time that you open the model.

If you want to load a specific justifications file, open Model Advisor on the model and, in the toolstrip, select **Open > Load Justifications File**.


To see which justifications file is currently loaded for a model, open Model Advisor and, in the **Check Selector** pane, select **Model Advisor**. The **Configurations** section shows the currently loaded justifications file.



If you want to create a different justifications file for a different set of justifications, see [Create Multiple Justifications Files](#) on page 3-107.

View Justifications

View the justifications that you added in Model. Results that you justify have a check result status of

Justified and the Justified icon  appears next to the results in Model Advisor and the Model Advisor report.

Note that to view justifications that you add during edit-time checking, you must re-run the checks from the Model Advisor app. For information on how to run checks and view results, see “[Check Model Compliance by Using the Model Advisor](#)” on page 3-2.

Edit Justifications

To edit the rationale for an existing justification:

1. Open Model Advisor.
2. Make sure the current justifications file is the file containing the justification you want to edit. In the **Check Selector** pane, click **Model Advisor** to view the current justifications file.
3. Select the result for which you want to edit the justification. In the **Check Selector** pane, select the check and in the **Result Details** tab, select the justified result.

4. Click **Edit** in the **Result Inspector** pane to edit the rationale in the **Justifications** field.
5. Click **Add Justification** to update the justification with the new rationale.

Delete Justifications

To delete an existing justification:

1. Open Model Advisor.
2. Make sure the current justifications file is the file containing the justification you want to delete. In the **Check Selector** pane, click **Model Advisor** to view the current justifications file.
3. Select the result for which you want to delete the justification. In the **Check Selector** pane, select the check and in the **Result Details** tab, select the justified result.
4. Click **Delete** in the **Result Inspector** pane to delete the justification from the currently loaded justifications file. The check result status returns to its original status of **Error** or **Warning**.

Create Multiple Justifications Files

If you already have a justifications file and you want to create a set of justifications in a new file, you must unload your current file. To unload your current file, move or rename it. When you add a justification for the model, Model Advisor prompts you to save a new justifications file.

Note that you can also use the `Justifications` argument in `ModelAdvisor.run` to perform a Model Advisor analysis with a specific justifications file.

See Also

`ModelAdvisor.run`

Related Examples

- “Check Model Compliance by Using the Model Advisor” on page 3-2
- “Check Model Compliance Using Edit-Time Checking” on page 3-6
- “Exclude Blocks from the Model Advisor Check Analysis” on page 3-9

Check Systems Programmatically

Checking Systems Programmatically

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

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

To define the workflow for running multiple checks on systems:

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

See Also

`ModelAdvisor.run`

Related Examples

- “Archive and View Results” on page 4-6

More About

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

Create a Function to Check Multiple Systems

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

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

Write a Function to Run Checks and Return Results

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

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

4. Delete the body of the function.

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

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

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

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

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

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

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

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

```
fail = 0;
warn = 0;

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

The `run_configuration` function now contains this content:

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

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

fileName = 'myCheckConfiguration.json';

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

fail = 0;
warn = 0;

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

end
```

Test the Function

1. Save the `run_configuration` function.

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

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

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

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

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

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

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

Systems passed: 3 of 3

Systems with warnings: 0 of 3

Systems failed: 0 of 3

Systems justified: 0 of 3
To view the summary report, use the 'ModelAdvisor.summaryReport(SystemResultObjArray)' o
```

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

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

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

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

```
Number of checks that return failures: 0
```

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

```
Number of checks that return warnings: 5
```

Checking Multiple Systems in Parallel

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

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

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

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

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

See Also

`ModelAdvisor.run` | `ModelAdvisor.setDefaultConfiguration`

Related Examples

- “Checking Systems Programmatically” on page 4-2
- “Use the Model Advisor Configuration Editor to Create a Custom Model Advisor Configuration” on page 7-8

Archive and View Results

Archive Results

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

View Results in Command Window

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

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

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

You can review additional results in the Command Window by calling the `DisplayResults` parameter when you run the Model Advisor. For example, open the example model `sldemo_auto_climatecontrol`.

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

Run the Model Advisor as follows:

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

The results displayed in the Command Window are:

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

Systems passed: 0 of 1

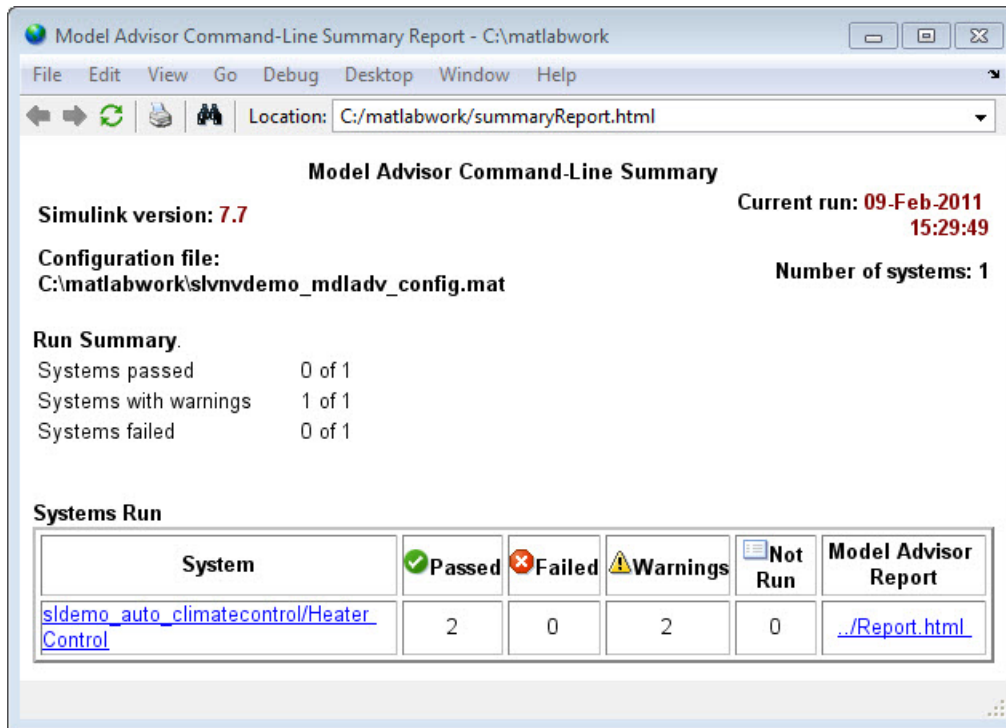
Systems with warnings: 1 of 1

Systems failed: 0 of 1
Summary Report
```

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

View Results in Model Advisor Command-Line Summary Report

When you run the Model Advisor programmatically, a Summary Report link is displayed in the Command Window. Clicking this link opens the Model Advisor Command-Line Summary report.



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

View Results in Model Advisor GUI

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

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

View Model Advisor Report

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

To view the Model Advisor report for a system:

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

To view individual check results:

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

- Use the view function.

See Also

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

Related Examples

- “Archive and View Model Advisor Run Results” on page 4-9
- “Create a Function to Check Multiple Systems” on page 4-3

More About

- “Run Model Advisor Checks and Review Results” on page 3-4
- “Address Model Check Results”
- “Generate Model Advisor Reports” on page 3-16
- “Save and View Model Advisor Check Reports”
- “Find Model Advisor Check IDs”
- “Save and Load Process for Objects”

Archive and View Model Advisor Run Results

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

- 1 Open the example model `sldemo_auto_climatecontrol`.

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

- 2 Call the `ModelAdvisor.run` function:

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

- 3 Save the `SysResultObjArray` for use at a later time:

```
save my_model_advisor_run SysResultObjArray
```

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

```
clear
```

- 5 Load the results of the Model Advisor run:

```
load my_model_advisor_run SysResultObjArray
```

- 6 View the results in the Model Advisor:

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

See Also

`ModelAdvisor.run`

Related Examples

- “Archive and View Results” on page 4-6

Run Tasks Locally and in CI

The support package CI/CD Automation for Simulink Check provides tools to help you integrate your model-based process into a Continuous Integration / Continuous Deployment (CI/CD) system.

The support package provides:

- A customizable process modeling system that you can use to define your build and verification process
- A build system that can automatically generate and efficiently execute a process in your CI system
- The **Process Advisor** app for deploying and automating your prequalification process
- Integration with common CI systems

You can use the support package to help you set up a model-based design (MBD) pipeline, reduce build time, reduce build failures, debug build failures, and deploy a consistent build and verification process.

Note The support package supports:

- R2022b Update 1 and later updates
 - R2022a Update 4 and later updates
-

Installation

To install the support package, use one of the following methods:

- Download the support package installer from the MATLAB File Exchange. To download and install, see CI/CD Automation for Simulink Check.
- Use the Add-On Explorer. In the MATLAB toolstrip, click **Add-Ons > Get Add-Ons** and search for CI/CD Automation for Simulink Check.

MBD Pipeline

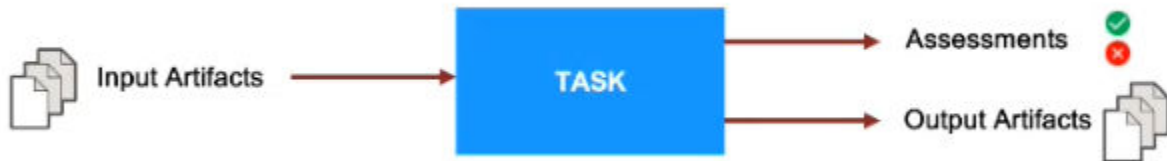
In a typical CI/CD pipeline, the CI/CD system automatically builds your source code, performs testing, packages deliverables, and deploys the packages to production. With the support package CI/CD Automation for Simulink Check, you can create a pipeline for the steps in your build and verification process, and maintain a repeatable CI/CD process for model-based design.

For example, this diagram shows an MBD pipeline that checks modeling standards, runs tests, generates code, and performs a custom task.



You can use the customizable process modeling system to define the steps in your model-based design (MBD) pipeline. You define the steps by using a process model. A *process model* is a MATLAB script that specifies the tasks in the CI/CD process, dependencies between the tasks, and artifacts that you associate with each task.

A *task* is a single step in your process. Tasks can accept your project artifacts as inputs, perform actions, generate pass, fail, or warning assessments, and return project artifacts as outputs.



The support package contains built-in tasks for several common steps, including:

- Creating Web views for your models with Simulink Report Generator
- Checking modeling standards with the Model Advisor
- Running tests with Simulink Test
- Detecting design errors with Simulink Design Verifier
- Generating a System Design Description (SDD) report with Simulink Report Generator
- Generating code with Embedded Coder
- Checking coding standards with Polyspace Bug Finder
- Inspecting code with Simulink Code Inspector
- Running tests with Simulink Test
- Generating a consolidated test results report and a merged coverage report with Simulink Test and Simulink Coverage

The support package contains a default process model for an MBD pipeline, but you can also customize the default process model to fit your development workflow goals. For example, your custom process model might include the built-in tasks for checking modeling standards, running tests, and generating code before performing a custom task. You can customize the process model to add or remove tasks in the MBD pipeline. You can also reconfigure the tasks in your process model to change what action a task performs or how a task performs the action.

Build System

The support package CI/CD Automation for Simulink Check provides a build system that you can use to automate the steps in your MBD pipeline. The *build system* is software that can create the pipeline of tasks, efficiently execute tasks in the pipeline, and perform other actions related to the pipeline.

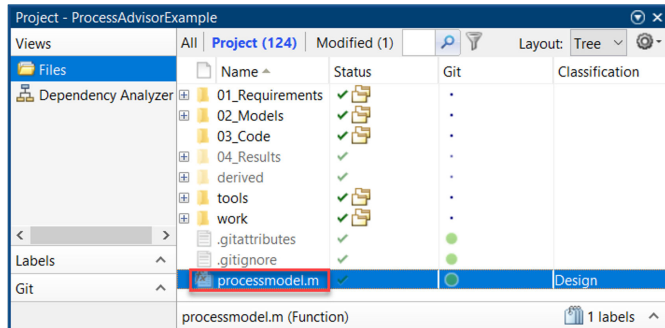
To create the pipeline of tasks, the build system needs:

- 1 A MATLAB project to analyze
- 2 A process model in the project that defines the tasks in the pipeline

If the project does not contain a process model, the build system copies the default process model into the project and uses the default process model to create a default MBD pipeline.

When you call the build system, the build system loads the process model, analyzes the project, and creates a pipeline of tasks for the project.

MATLAB Project with a Process Model



Build System



Pipeline of Tasks

Tasks
<input type="radio"/> Generate Simulink Web View
<input type="radio"/> Generate Code (Ref)
<input type="radio"/> Inspect Code (Ref)
<input type="radio"/> Check Coding Standards (Ref)
<input type="radio"/> Generate Code (Top)
<input type="radio"/> Inspect Code (Top)
<input type="radio"/> Check Coding Standards (Top)
<input type="radio"/> Check Modeling Standards
<input type="radio"/> Detect Design Errors
<input type="radio"/> Generate SDD Report
<input type="radio"/> Run Tests
<input type="radio"/> Merge Test Results

To run the tasks in the pipeline, you can call the build system using one of these approaches:

- In a CI environment by using the build system API. The build system API includes a function `runprocess` that you can use to run the tasks in a pipeline.
- Locally on your machine by using either the build system API or the **Process Advisor** app. The **Process Advisor** app is a user interface that can call the build system. The **Process Advisor** app has run buttons that you can use to run the tasks in a pipeline. If there is a failure in the CI environment, you can reproduce the issue locally on your machine by using the **Process Advisor** app.

The build system supports incremental builds. If you change an artifact in your project, the build system can detect the change and automatically determine which of the tasks in your MBD pipeline now have outdated results. In your next build, you can instruct the build system to run only the tasks with outdated results. By identifying the tasks with outdated results, the build system can help you reduce build time by reducing the number of tasks you need to re-run after making changes to your project artifacts.

Note There are limitations to the types of changes that the support package can detect. For more information, see the documentation that ships with the support package.

Process Advisor App

A prequalification process can help you prevent build and test failures from occurring in your CI/CD system. Use the **Process Advisor** desktop app to deploy and automate your prequalification process. You can use the app to locally run the tasks in your MBD pipeline and to prequalify your changes on your machine before submitting to source control. The **Process Advisor** app is a user interface for running the build system locally on your machine. The **Process Advisor** runs tasks locally, not in the CI environment. You can use the **Process Advisor** app on your local machine to run the tasks in your MBD pipeline and to check your progress towards completing tasks in your prequalification pipeline for your changes.

If you make a change to an artifact in your project, the **Process Advisor** can detect the change and automatically determine the impact of the change on your existing task results. For example, if you

complete a task but then update your model, the **Process Advisor** automatically invalidates the task completion and marks the task results as outdated.

CI/CD System Integration

You can use the support package CI/CD Automation for Simulink Check to integrate your model-based design process into common CI/CD systems. For example, you can configure and integrate your MBD pipeline by using a YAML file to configure your pipeline for GitLab® or a Jenkinsfile for configuring your pipeline for Jenkins®.

The support package contains example pipeline configuration files:

- To open an example project that contains a GitLab pipeline file, enter this code in the MATLAB Command Window:

```
processAdvisorGitLabExampleStart
```

This code creates an example project that contains an example YAML file, `.gitlab-ci.yml`, in the project root.

- To open an example project that contains an example Jenkins pipeline file, enter this code in the MATLAB Command Window:

```
processAdvisorJenkinsExampleStart
```

This code creates an example project that contains an example Jenkinsfile, `Jenkinsfile`, in the project root.

See Also

Related Examples

- “Continuous Integration with MATLAB on CI Platforms”
- <https://www.mathworks.com/company/newsletters/articles/continuous-integration-for-verification-of-simulink-models.html>
- <https://www.mathworks.com/company/newsletters/articles/continuous-integration-for-verification-of-simulink-models-using-gitlab.html>

Model Metrics

Collect and Explore Metric Data by Using the Metrics Dashboard

Note The Metrics Dashboard will be removed in a future release. For size, architecture, and complexity metrics, use the Model Maintainability Dashboard instead. For more information, see “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard” on page 5-144.

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

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

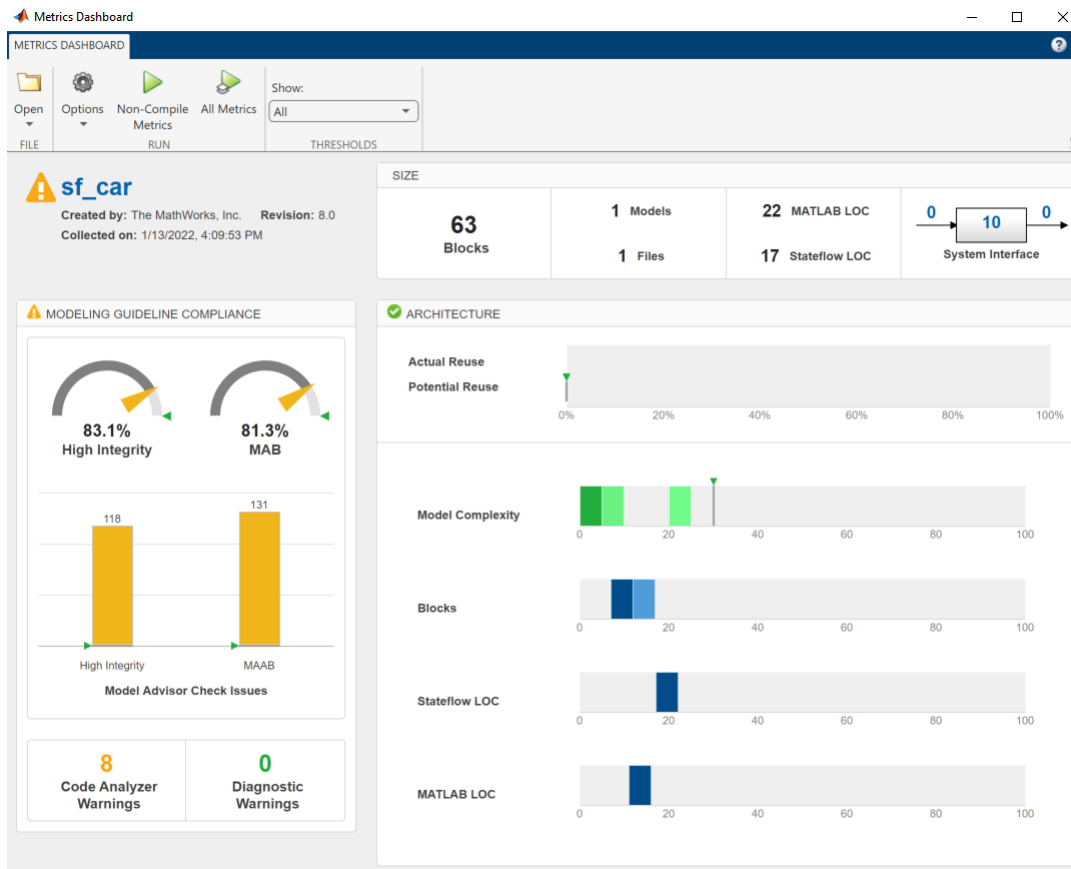
You can collect metric data by using the dashboard or programmatically by using the `slmetric.Engine` API. When you open the dashboard, if you have previously collected metric data for a particular model, the dashboard populates from existing data in the database.

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

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

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

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



Metrics Dashboard Widgets

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

- The Metrics Dashboard calculates metric data per component. A component can be a model, subsystem, chart, or MATLAB Function block.
- You can view results in either a **Table** or **Tree** view. For the **High Integrity** and **MAB** compliance widgets, you can also choose a **Grid** view. To view highlighted results, in the grid view, click a cell.
- To sort the results by value or aggregated value, click the corresponding value column header.
- For metrics other than the **High Integrity** and **MAB** compliance widgets, you can filter results. To filter results, in the **Table** view, select the context menu on the right side of the **TYPE**, **COMPONENT**, and **PATH** column headers. From the **TYPE** menu, select applicable components. From the **COMPONENT** and **PATH** menus, type a component name or path in the search bar. The Metrics Dashboard saves the filters for a widget, so you can view metric details for other widgets and return to the filtered results.

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

Effective lines of code for Stateflow blocks
Effective number of lines of code for Stateflow blocks

Type	Component	Path	Qty	Stateflow LOC	Stateflow LOC (incl. Descendants)
Model	sf_car	sf_car	1	n/a	17
Chart	shift_logic	sf_car/shift_logic	1	17	17

Effective lines of code for Stateflow blocks
Effective number of lines of code for Stateflow blocks

Component	Type	Stateflow LOC	Stateflow LOC (incl. Descendants)
▼ sf_car	Model	n/a	17
Engine	Subsystem	n/a	n/a
Vehicle	Subsystem	n/a	n/a
▼ shift_logic	Chart	17	17
▼ selection_state.calc_th	Subsystem	n/a	n/a
Look-Up	MATLAB Function	n/a	n/a
▼ transmission	Subsystem	n/a	n/a
Torque Converter	Subsystem	n/a	n/a
▼ transmission_ratio	Subsystem	n/a	n/a
Look-Up Table	MATLAB Function	n/a	n/a

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

For custom metrics, you can specify widgets to add to the dashboard. You can also remove widgets. To learn more about customizing the Metrics Dashboard, see “Customize Metrics Dashboard Layout and Functionality” on page 5-37.

Size

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

Widget	Metric	Drill-In Data
Blocks	Simulink block count (mathworks.metrics.SimulinkBlockCount)	Number of blocks by component
Models	Model file count (mathworks.metrics.ModelFileCount)	Number of model files by component
Files	File count (mathworks.metrics.FileCount)	Number of model and library files by component
MATLAB LOC	Effective lines of MATLAB code (mathworks.metrics.MatlabLOCCount)	Effective lines of code, in MATLAB Function block and MATLAB functions in Stateflow, by component

Widget	Metric	Drill-In Data
Stateflow LOC	Effective lines of code for Stateflow blocks (<code>mathworks.metrics.StateflowLOCCount</code>)	Effective lines of code for Stateflow blocks by component
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_do178</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.</p>
MAB Compliance	Model Advisor standards check compliance - MAB (<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.</p>
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.
MAB Check Issues	Model Advisor standards issues - MAB (<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.

Widget	Metric	Drill-In Data
Diagnostic Warnings	Simulink diagnostic warning count (<code>mathworks.metrics.DiagnosticWarningsCount</code>)	<ul style="list-style-type: none"> Number of Simulink diagnostic warnings by component. If there are warnings, at the top of the dashboard, there is a hyperlink that opens the Diagnostic Viewer.

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

You can use the Metrics Dashboard to perform compliance and issues checking on your own group of Model Advisor checks. For more information, see “Customize Metrics Dashboard Layout and Functionality” on page 5-37.

Architecture

These widgets provide a view of your system architecture:

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

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

Widget	Metric	Drill-In Data
Potential Reuse / Actual Reuse	Potential Reuse (<code>mathworks.metrics.CloneContent</code>) and Actual Reuse (<code>mathworks.metrics.LibraryContent</code>)	<p>Fraction of total number of subcomponents that are clones as a percentage</p> <p>Fraction of total number of components that are linked library blocks as a percentage</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

Widget	Metric	Drill-In Data
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

Metric Thresholds

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

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

You can specify your own metric threshold values for all of the widgets in the Metrics Dashboard. You can also specify values corresponding to a noncompliant range. For more information, see “Customize Metrics Dashboard Layout and Functionality” on page 5-37.

Dashboard Limitations

When using the Metrics Dashboard, note these considerations:

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

See Also

More About

- “Collect Model Metrics Programmatically” on page 5-15
- “Model Metrics”
- “Collect Compliance Data and Explore Results in the Model Advisor” on page 5-25
- “Collect Metric Data Programmatically and View Data Through the Metrics Dashboard” on page 5-28

Collect Model Metrics Using the Model Advisor

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

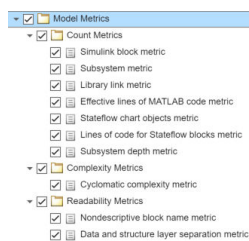
The results of these metrics can help you verify compliance with industry standards and guidelines.

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

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

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


```
openExample('sldemo_fuelsys')
```
- 2 In the model window, open the **Modeling** tab and click **Model Advisor**. A System Selector dialog box opens. Click **OK**.
- 3 In the left pane of the Model Advisor, navigate to **By Task > Model Metrics**. Select the **Count Metrics**, **Complexity Metrics**, and **Readability Metrics** checks.



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

Component	Blocks
.../fuel_rate_control/airflow_calc	24
sldemo_fuelsys	20
sldemo_fuelsys/Dashboard	14
.../Throttle & Manifold/Throttle	14
sldemo_fuelsys/To Controller	12
.../fuel_calc/switchable_compensation	11
.../Engine Gas Dynamics/Mixing & Combustion	9
.../fuel_calc/feedforward_fuel_rate	8
.../fuel_rate_control/validate_sample_time	8
.../Engine Gas Dynamics/Throttle & Manifold	8
.../Throttle & Manifold/Intake Manifold	8
sldemo_fuelsys/Engine Gas Dynamics	8
sldemo_fuelsys/fuel_rate_control	7
.../control_logic/Speed_speed_estimate	6
.../fuel_rate_control/fuel_calc	6
.../switchable_compensation/low_mode	6
.../switchable_compensation/rich_mode	6
.../control_logic/Throttle_throttle_estimate	5
.../control_logic/Pressure_map_estimate	5
sldemo_fuelsys/To Plant	4
.../switchable_compensation/disabled_mode	3
.../Mixing & Combustion/System Lag	3

model level and subsystem level.

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

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

See Also

More About

- “Model Metrics”
- “Model Metric Data Aggregation” on page 5-19
- “Collect Model Metrics Programmatically” on page 5-15
- “Create a Custom Model Metric for Nonvirtual Block Count” on page 5-11
- “Run Model Advisor Checks and Review Results” on page 3-4

Create a Custom Model Metric for Nonvirtual Block Count

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

Create Metric Class

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

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

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

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

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

3. Write the metric algorithm into the `slmetric.metric.Metric` method, `algorithm`. The algorithm calculates the metric data specified by the `Advisor.component.Component` class. The `Advisor.component.Types` class specifies the types of model objects for which you can calculate metric data. For this example, the file `nonvirtualblockcount_orig.m` contains the logic to create a metric that counts the nonvirtual blocks. Copy this file to the `nonvirtualblockcount.m` file.

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

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

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

Optionally, you can set these additional properties:

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

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

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

Collect Metric Data

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

1. Load the `sldemo_mdhref_conversion` model.

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

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

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

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

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

Display and Export Results

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

```
for n=1:length(rc)
    if rc(n).Status == 0
        results = rc(n).Results;

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

```
MetricID: nonvirtualblockcount
```

```
ComponentPath: sldemo_mdhref_conversion
```

```
Value: 7
```

```
MetricID: nonvirtualblockcount
```

```
ComponentPath: sldemo_mdhref_conversion/Bus Counter
```

Value: 8

MetricID: nonvirtualblockcount

ComponentPath: sldemo_mdref_conversion/More Info

Value: 0

MetricID: nonvirtualblockcount

ComponentPath: sldemo_mdref_conversion/SubSystem1

Value: 0

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

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

For this example, unregister the nonvirtual block count metric.

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

Close the model.

```
clear;
bdclose(model);
```

Limitations

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

- Linked Stateflow® charts
- MATLAB Function blocks

Custom metric algorithms do not follow library links.

See Also

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

More About

- “Model Metrics”
- “Model Metric Data Aggregation” on page 5-19

- “Collect Model Metrics Programmatically” on page 5-15

Collect Model Metrics Programmatically

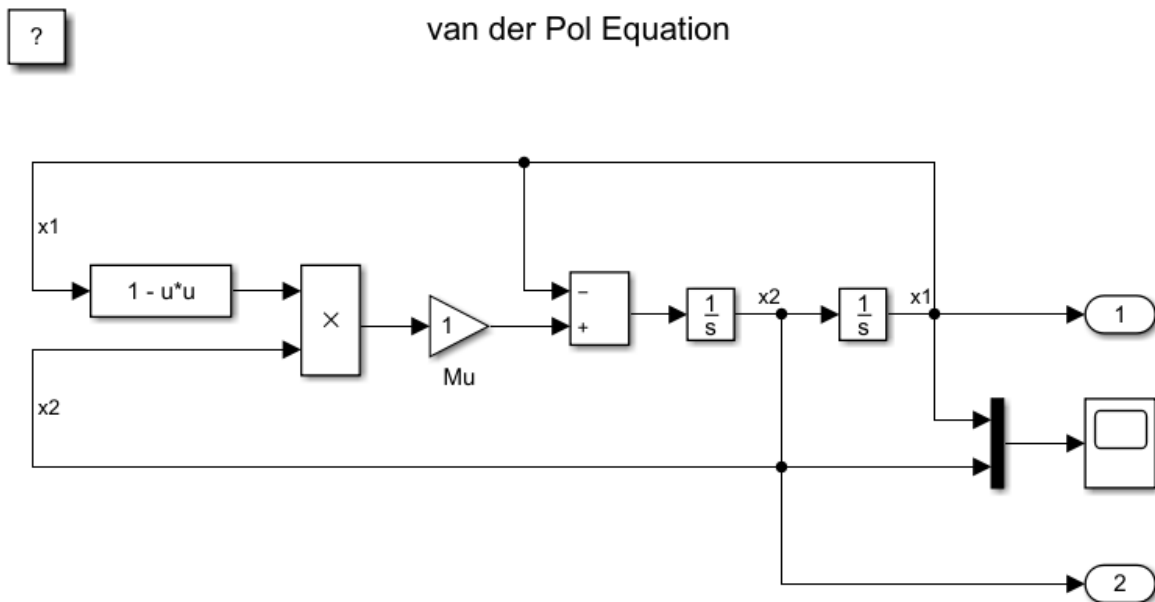
You can use the model metric API to programmatically collect model metrics that help you assess the architecture, complexity, and readability of your model. The results of these metrics can help you verify compliance with industry standards and guidelines.

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

Example Model

Open the vdp model.

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



Copyright 2004-2013 The MathWorks, Inc.

Collect Metrics

To collect metric data on a model, create an `slmetric.Engine` object and call `execute`.

```
metric_engine = slmetric.Engine();
```

Warning: The Metrics Dashboard and `slmetric.Engine` API will be removed in a future release. For size, architecture, and complexity metrics, use the Model Maintainability Dashboard and `metric.Engine`. The Model Maintainability Dashboard and `metric.Engine` API can identify outdated metric results, and for more information, see [matlab:helpview\(\['/slcheck/collect-model-metric-data-'\]\)](matlab:helpview(['/slcheck/collect-model-metric-data-'])).

```
setAnalysisRoot(metric_engine, 'Root', 'vdp', 'RootType', 'Model');
execute(metric_engine);
```

Updating Model Advisor cache...

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

Access Results

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

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

Store and Display Results

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

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

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

```
MetricID: mathworks.metrics.SimulinkBlockCount
```

```
ComponentPath: vdp
```

```
Value: 13
```

```
MetricID: mathworks.metrics.SimulinkBlockCount
```

```
ComponentPath: vdp/More Info
```

```
Value: 1
```

```
MetricID: mathworks.metrics.SimulinkBlockCount
```

```
ComponentPath: vdp/More Info/Model Info
```

```
Value: 1
```

```
MetricID: mathworks.metrics.SimulinkBlockCount
```

```
ComponentPath: vdp/More Info/Model Info/EmptySubsystem
```

```
Value: 0
```

```
MetricID: mathworks.metrics.SubSystemCount
```

```
ComponentPath: vdp
```

```
Value: 1
```

```
MetricID: mathworks.metrics.SubSystemCount
```

```
ComponentPath: vdp/More Info
```

```
Value: 0
```

```
MetricID: mathworks.metrics.SubSystemCount
```

```
ComponentPath: vdp/More Info/Model Info
```

```
Value: 1
```

```
MetricID: mathworks.metrics.SubSystemCount
```

```
ComponentPath: vdp/More Info/Model Info/EmptySubsystem
```

```
Value: 0
```

Export Results

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

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

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

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

Close the vdp model.

```
bdclose(model);
```

Limitations

When you collect metric data, it is stored in a database file, `Metrics.db`, inside the simulation cache folder. You cannot collect metric data on one platform, move the database file to another platform, and then continue to collect additional metric data in that database file. For example, if you collect metric data on a Windows machine and then move the database file to a Linux machine, you cannot

collect and store additional metric data in that database file. However, you are able to view that data in the Metrics Dashboard.

See Also

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

More About

- “Model Metrics”
- “Model Metric Data Aggregation” on page 5-19
- “Collect Model Metrics Using the Model Advisor” on page 5-9
- “Create a Custom Model Metric for Nonvirtual Block Count” on page 5-11

Model Metric Data Aggregation

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

How Model Metric Aggregation Works

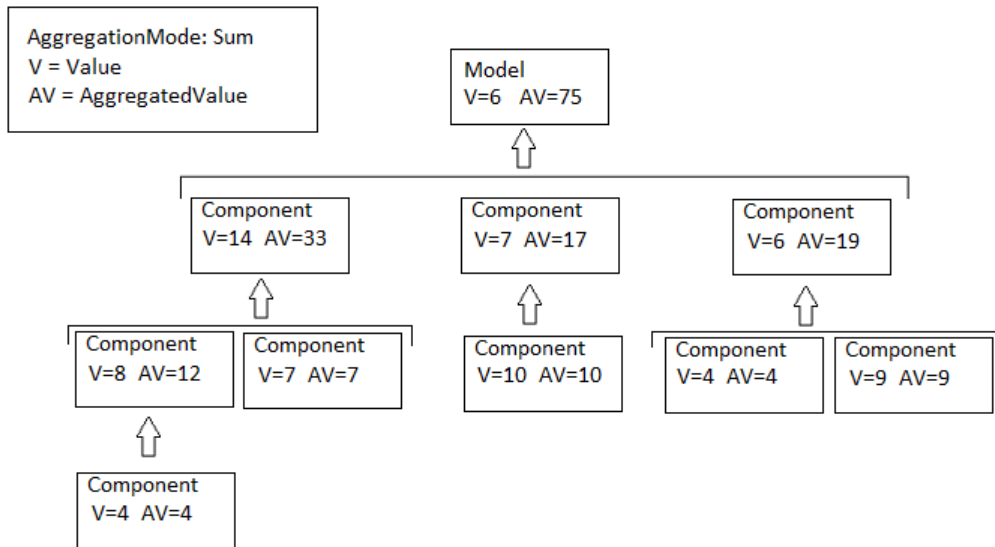
The implementation of a model metric defines how a metric aggregates data across a component hierarchy. For MathWorks model metrics, the `slmetric.metric.Metric` class defines model metric aggregation. This class includes the `AggregationMode` property, which has these options:

- **Sum:** Returns the sum of the `Value` property and the `Value` properties of its children components across the component hierarchy. Returns the sum of the `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.

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

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

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



In the diagram, the AggregationMode is Sum and the model and components in the hierarchy each have a Value and an AggregatedValue. The AggregatedValue for a parent model or component is the sum of its Value and the AggregatedValue of each direct child component. For example, in this diagram, the AggregatedValue of the parent model is 75. The AggregatedValue of the parent model is calculated as the sum of the Value of the parent model, 6, plus the AggregatedValue of each direct child component, 33, 17, and 19.

Access Aggregated Metric Data

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

- 1 Load the `sldemo_auto_climatecontrol` model.

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

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

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

- 3 Collect data for the Input output model metric.

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

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

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

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

- 5 Display the results.

```
metricData = {'MetricID', 'ComponentPath', 'Value', ...
'AggregatedValue', 'Measures', 'AggregatedMeasures'};
```

```

cnt = 1;
for n=1:length(res_col)
    if res_col(n).Status == 0
        results = res_col(n).Results;

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

```

Here are the results:

```

MetricID: mathworks.metrics.IOCCount
ComponentPath: sldemo_auto_climatecontrol
Value: 0
Aggregated Value: 9
Measures: 0 0 0 0
Aggregated Measures: 5 4 0 0
MetricID: mathworks.metrics.IOCCount
ComponentPath: sldemo_auto_climatecontrol/AC Control
Value: 6
Aggregated Value: 6
Measures: 5 1 0 0
Aggregated Measures: 5 1 0 0
MetricID: mathworks.metrics.IOCCount
ComponentPath: sldemo_auto_climatecontrol/External Temperature in Celsius
Value: 1
Aggregated Value: 1
Measures: 0 1 0 0
Aggregated Measures: 0 1 0 0
MetricID: mathworks.metrics.IOCCount
ComponentPath: sldemo_auto_climatecontrol/Heat from occupants
Value: 1
Aggregated Value: 1
Measures: 0 1 0 0
Aggregated Measures: 0 1 0 0
MetricID: mathworks.metrics.IOCCount
ComponentPath: sldemo_auto_climatecontrol/Heater Control
Value: 8
Aggregated Value: 8
Measures: 5 3 0 0
Aggregated Measures: 5 3 0 0
MetricID: mathworks.metrics.IOCCount
ComponentPath: sldemo_auto_climatecontrol/Interior Dynamics
Value: 3
Aggregated Value: 3
Measures: 2 1 0 0
Aggregated Measures: 2 1 0 0
MetricID: mathworks.metrics.IOCCount
ComponentPath: sldemo_auto_climatecontrol/More Info
Value: 0
Aggregated Value: 0
Measures: 0 0 0 0
Aggregated Measures: 0 0 0 0
MetricID: mathworks.metrics.IOCCount

```

```
ComponentPath: sldemo_auto_climatecontrol/Subsystem
Value: 2
Aggregated Value: 2
Measures: 1 1 0 0
Aggregated Measures: 1 1 0 0
MetricID: mathworks.metrics.IOCount
ComponentPath: sldemo_auto_climatecontrol/Subsystem1
Value: 2
Aggregated Value: 2
Measures: 1 1 0 0
Aggregated Measures: 1 1 0 0
MetricID: mathworks.metrics.IOCount
ComponentPath: sldemo_auto_climatecontrol/Temperature Control Chart
Value: 9
Aggregated Value: 9
Measures: 5 4 0 0
Aggregated Measures: 5 4 0 0
MetricID: mathworks.metrics.IOCount
ComponentPath: sldemo_auto_climatecontrol/User Setpoint in Celsius
Value: 1
Aggregated Value: 1
Measures: 0 1 0 0
Aggregated Measures: 0 1 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_auto_climatecontrol`, the `AggregatedValue` property is 9, which is the `sldemo_auto_climatecontrol/Temperature Control Chart` component value.

See Also

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

More About

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

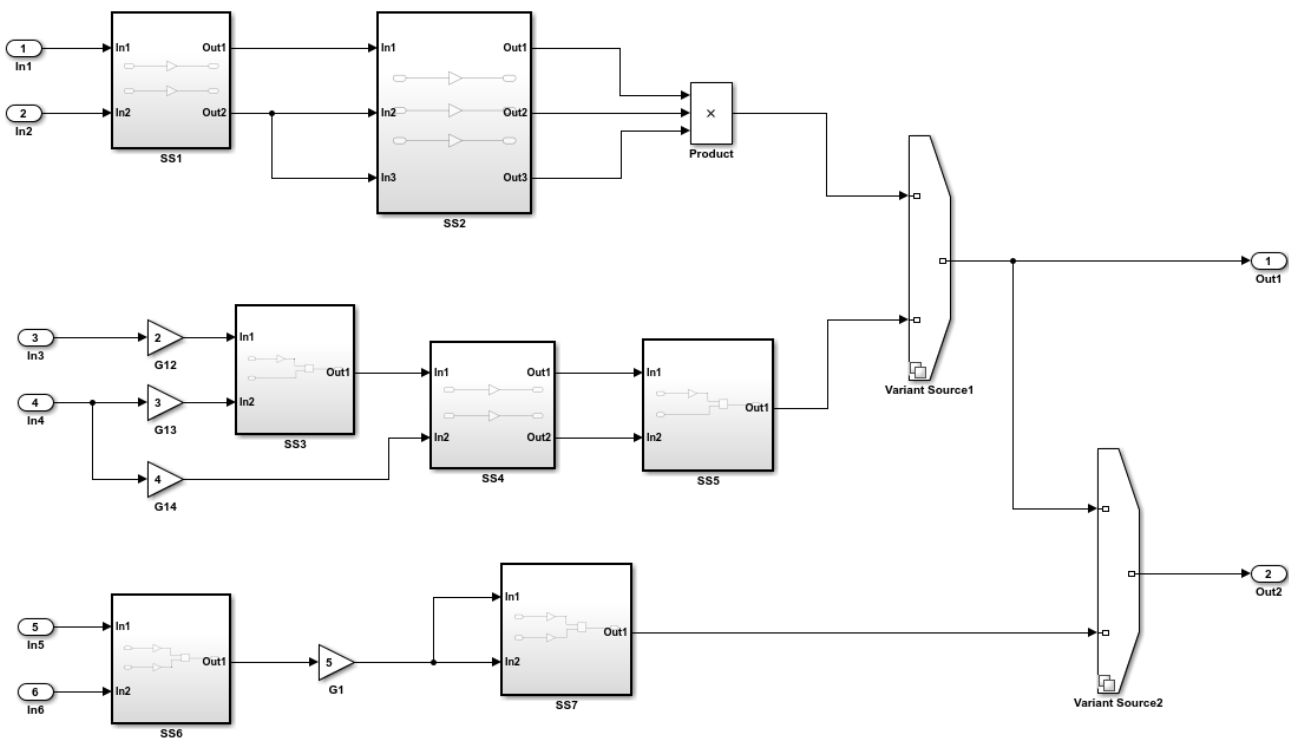
Identify Modeling Clones with the Metrics Dashboard

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

Identify Clones

Open model `ex_clone_detection`.

```
open_system('ex_clone_detection.slx')
```



Copyright 2017 The MathWorks Inc.

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

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

Replace Clones with Links to Library Blocks

- 1 Open the Clone Detector app by clicking **Open Conversion Tool**. The Clone Detector app opens. For more information on the app, see “Enable Component Reuse by Using Clone Detection” on page 3-29.
- 2 Click **Replace Clones**. The Clone Detector app replaces the clones with links to library blocks. The library blocks are in the library specified by the **Library to place clones** parameter. This parameter is on the **Clone Results** tab. The library is on the MATLAB® path. It has a default name of newLibraryFile.

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

Run Model Metrics on the Refactored Model

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

See Also

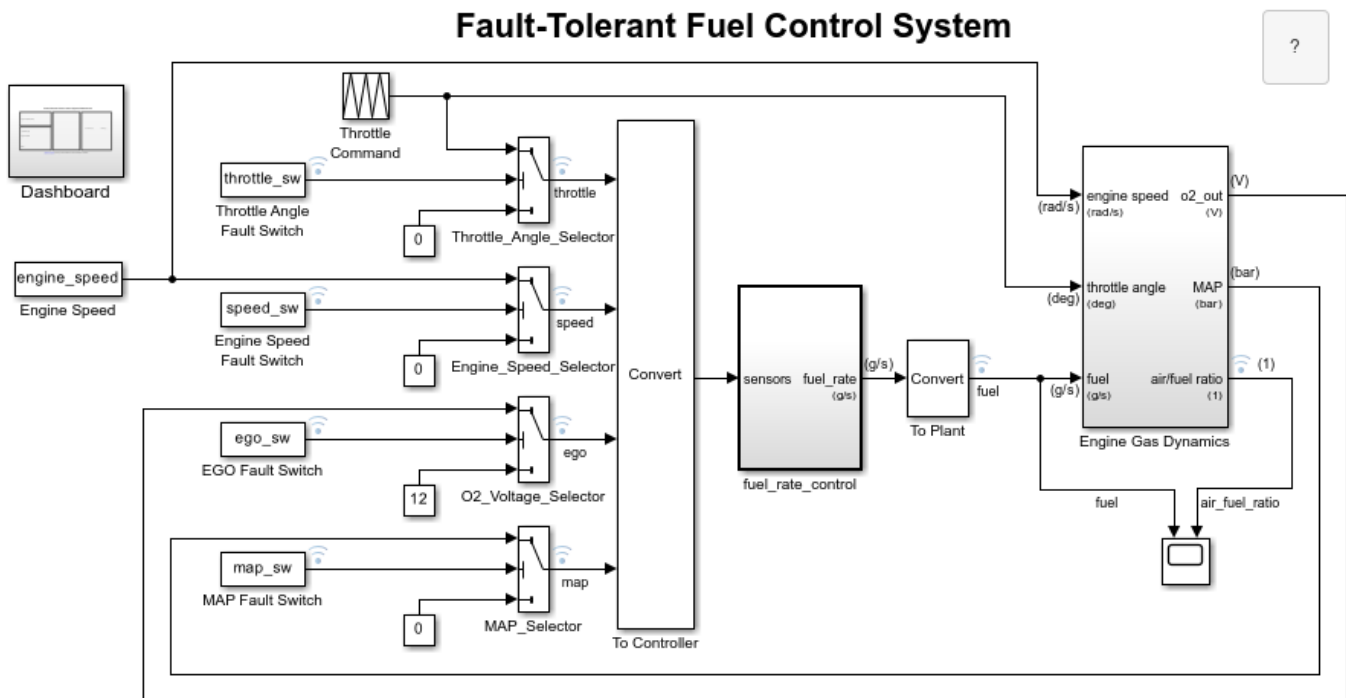
More About

- “Collect Model and Testing Metrics”

Collect Compliance Data and Explore Results in the Model Advisor

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

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



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

Copyright 1990-2022 The MathWorks, Inc.

Open the Metrics Dashboard

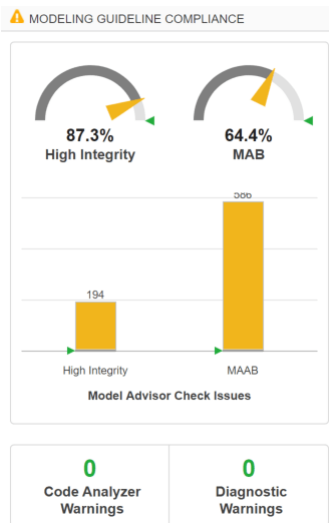
On the **Apps** tab, open the Metrics Dashboard by clicking **Metrics Dashboard**.

Collect Model Metrics

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

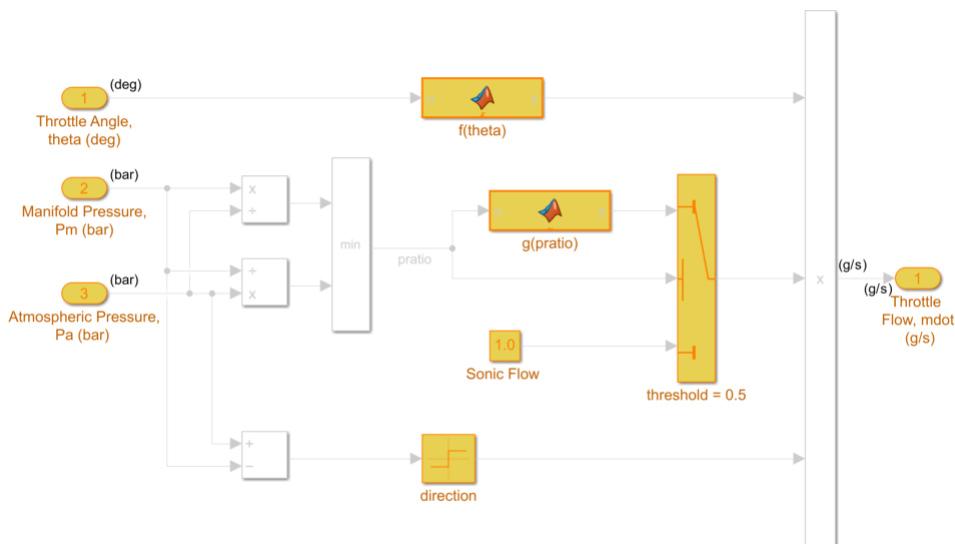
Explore Compliance Results

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



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

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



Throttle Flow vs. Valve Angle and Pressure

Explore Compliance Results in the Model Advisor

- 1 In the Metrics Dashboard, return to the main dashboard page by clicking the **Dashboard** icon.
- 2 Click the **High Integrity** percentage gauge.
- 3 To see the status for each compliance check, click the **Table** view.

- 4 Expand the `sldemo_fuelsys` node.
- 5 To explore check results in more detail, click the **Check safety-related diagnostic settings for sample time** hyperlink.
- 6 In the Model Advisor Highlight dialog box, click **Check safety-related diagnostic settings for sample time** hyperlink.

Fix a Compliance Issue

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

Recollect Metrics

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

See Also

More About

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

Collect Metric Data Programmatically and View Data Through the Metrics Dashboard

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

Collect Metric Data Programmatically

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

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

Determine Model Compliance with MAB Guidelines

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

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

```
MAAB compliance: 64.4%
```

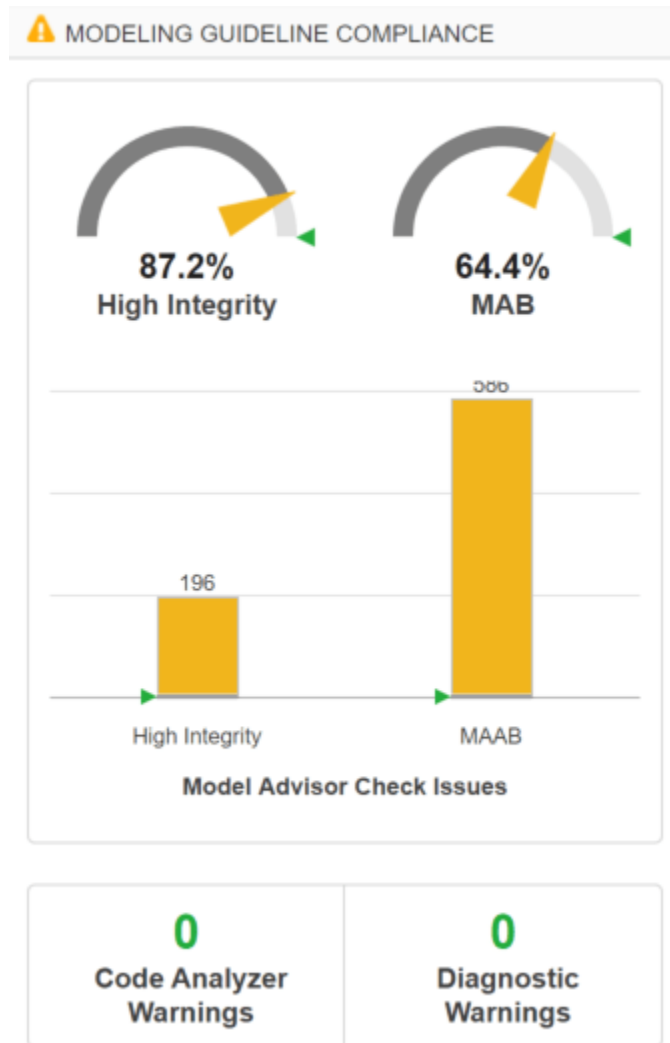
Open the Metrics Dashboard

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

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

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

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



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


Explore the MAB Compliance Results


Underneath the percentage gauges, the bar chart indicates the number of compliance check issues.


Click the MAAB bar in the bar chart to view a table of the Model Advisor Check Issues for MAB.

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

METRICS DETAILS ?



























 Dashboard
VIEW


 Table
VISUALIZATIONS


 Tree

Model Advisor standards issues for MAB ?

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

Type	Component	Path	Q..	Issues	Issues (Incl...
Chart	control_logic	 sldemo_fuelsys/fuel_rate_control/control_logic	1	264	300
Model	sldemo_fuelsys	 sldemo_fuelsys	1	62	586
Subsystem	airflow_calc	 sldemo_fuelsys/fuel_rate_control/airflow_calc	1	49	49
Subsystem	Dashboard	 sldemo_fuelsys/Dashboard	1	32	32
Subsystem	Throttle	 ...s/Engine Gas Dynamics/Throttle & Manifold/Throttle	1	25	26
Subsystem	Throttle & Manifold	 ...emo_fuelsys/Engine Gas Dynamics/Throttle & Manifold	1	19	59
Subsystem	Speed.speed_estimate	 .../fuel_rate_control/control_logic/Speed.speed_estimate	1	14	14
Subsystem	Intake Manifold	 ...gine Gas Dynamics/Throttle & Manifold/Intake Manifold	1	14	14
Subsystem	Throttle.throttle_estimate	 ...uel_rate_control/control_logic/Throttle.throttle_estimate	1	11	11
Subsystem	Pressure.map_estimate	 .../fuel_rate_control/control_logic/Pressure.map_estimate	1	11	11
Subsystem	switchable_compensation	 ...s/fuel_rate_control/fuel_calc/switchable_compensation	1	11	26
Subsystem	fuel_rate_control	 sldemo_fuelsys/fuel_rate_control	1	11	401
Subsystem	Mixing & Combustion	 ...o_fuelsys/Engine Gas Dynamics/Mixing & Combustion	1	9	9
Subsystem	To Controller	 sldemo_fuelsys/To Controller	1	9	9
Subsystem	Engine Gas Dynamics	 sldemo_fuelsys/Engine Gas Dynamics	1	9	77
Subsystem	feedforward_fuel_rate	 ...elsys/fuel_rate_control/fuel_calc/feedforward_fuel_rate	1	8	8
Subsystem	low_mode	 ..._control/fuel_calc/switchable_compensation/low_mode	1	6	6
Subsystem	rich_mode	 ..._control/fuel_calc/switchable_compensation/rich_mode	1	6	6
Subsystem	To Plant	 sldemo_fuelsys/To Plant	1	5	5
Subsystem	fuel_calc	 sldemo_fuelsys/fuel_rate_control/fuel_calc	1	4	38
Subsystem	validate_sample_time	 sldemo_fuelsys/fuel_rate_control/validate_sample_time	1	3	3
Subsystem	disabled_mode	 ...trol/fuel_calc/switchable_compensation/disabled_mode	1	3	3
MATLAB Function	g(pratio)	 ...ine Gas Dynamics/Throttle & Manifold/Throttle/g(pratio)	1	1	1
MATLAB Function	EGO Sensor	 ...gine Gas Dynamics/Mixing & Combustion/EGO Sensor	1	0	0
MATLAB Function	MATLAB Function	 .../Throttle & Manifold/Intake Manifold/MATLAB Function	1	0	0

See Also

More About

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

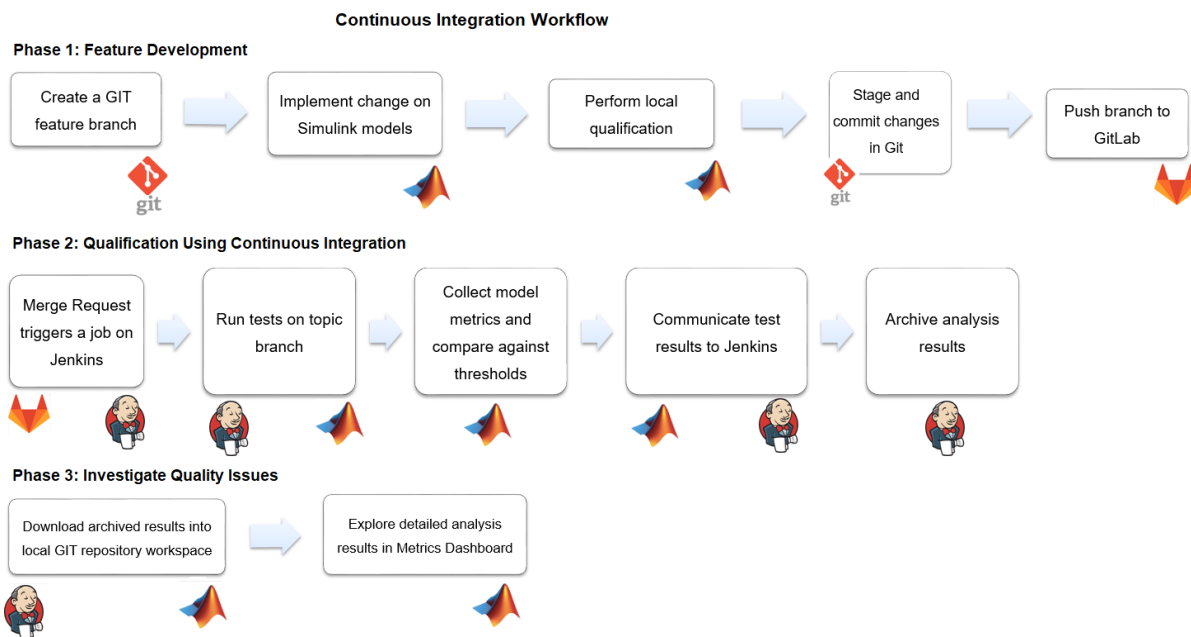
Fix Metric Threshold Violations in a Continuous Integration Systems Workflow

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

This example refers to a project that contains the shipped project `matlab:sldemo_slproject_airframe` and these files that you must provide:

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

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



Project Setup

In addition to the files in the `matlab:sldemo_slproject_airframe` project, you must provide these additional files:

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

- A MATLAB script that specifies metric thresholds and customizes the Metrics Dashboard. For more information on how to customize the Metrics Dashboard, see “Customize Metrics Dashboard Layout and Functionality” on page 5-37.
- A `setup.m` file that activates the configuration XML files that define metric thresholds, sets custom metric families, and customizes the Metrics Dashboard layout. For this example, the `setup.m` script contains this code:

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

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

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

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

- An `sl_customization.m` file that activates the Model Advisor configuration file to customize the Model Advisor checks. For more information on creating your own Model Advisor configuration, see *Configure Compliance Metrics* on page 5-0 .
- A run script that executes during a Jenkins build. For this example, this code is in the `run.m` file:

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

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

    % execute tests
    runUnitTest();

    slproj.close();

    if IN_CI
        exit
    end
end
```

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

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

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

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

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

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

GitLab Setup

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

- 1 Install the Git client.
- 2 Set up a branching workflow. With GitLab, from the main branch, create a temporary branch for implementing changes to the model files. Integration engineers can use Jenkins test results to decide whether to merge a temporary branch into the main branch. For more information, see <https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows>.
- 3 On the left side bar, under **Settings** > **Repository**, protect the main branch by enforcing the use of merge requests when developers want to merge their changes into the main branch.
- 4 Under **Settings**, on the **Integrations** page, add a webhook to the URL of your Jenkins project. This webhook triggers a build job on the Jenkins server.

Jenkins Setup

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

Create a Jenkins project. Specify these configurations:

- 1 In your Jenkins project, click **Configure**.
- 2 On the **General** tab, specify a project name.
- 3 On the **Source Code Management** tab, for the **Repository URL** field, specify the URL of your GitLab repository.
- 4 On the **Build Triggers** tab, select **Build when a change is pushed to GitLab**.
- 5 In the **Build Environment** section, select **Use MATLAB Version** and specify the **MATLAB root**, for example, `C:\Program Files\MATLAB\R2022a`.
- 6 In the **Build** section, execute MATLAB to call the run script. The run script opens the project and runs all unit tests. For the project in this example, the code is:

```
matlab -nodisplay -r...
"cd /var/lib/jenkins/workspace/'18b Metrics CI Demo'; run(true)"
```

For more information, see “Continuous Integration Using MATLAB Projects and Jenkins”.

- 7 In the **Post-build Actions** tab, configure the TAP plugin to publish TAP results to Jenkins. In the **Test Results** field, specify `reports/*.tap`. For **Files to archive**, specify `reports/**,work/**`.

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

Continuous Integration Workflow

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

Phase 1: Feature Development

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

Phase 2: Qualification by Using Continuous Integration

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

3 failures

29 tests
Took 0 ms
[add description](#)

All Failed Tests

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

All Tests

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

Phase 3: Investigate Quality Issues Locally

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

The screenshot shows the Simulink Metrics Dashboard for a project named 'sproject_f14'. The dashboard provides a comprehensive overview of the project's quality metrics. Key features include:

- Project Overview:** Created by The MathWorks, Inc., Revision: 1.93, Collected on: 5/16/2018, 3:50:13 AM.
- Summary Metrics:** 122 Blocks, 5 Models, 6 Files, 0 MATLAB LOC, 0 Stateflow LOC, and 43 System Interface elements.
- Modeling Guideline Compliance:** Two gauges show 85.0% compliance with Required Guidelines and 75.0% with Recommended Guidelines. A bar chart below compares 'Required' (red) and 'Recommended' (green) metrics.
- Architecture:** A horizontal bar chart compares 'Actual Reuse' (blue) against 'Potential Reuse' (grey).
- Model Complexity and Blocks:** Horizontal bar charts showing the distribution of model complexity and the number of blocks.
- Warnings:** 0 Code Analyzer Warnings and 0 Diagnostic Warnings.
- Location Metrics:** Horizontal bar charts for Stateflow LOC and MATLAB LOC.

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

See Also

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

More About

- “Collect Model Metric Data by Using the Metrics Dashboard” on page 1-9
- “Collect and Explore Metric Data by Using the Metrics Dashboard” on page 5-2

External Websites

- <https://www.mathworks.com/company/newsletters/articles/continuous-integration-for-verification-of-simulink-models.html>

Customize Metrics Dashboard Layout and Functionality

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

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

Configure Compliance Metrics

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

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

```
vdp
```

2. Open the default configuration and save a corresponding `slmetric.config.Configuration` object to the base workspace.

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

3. Create a cell array, `values`, that specifies the Model Advisor **Check Group IDs** for MAAB, High-Integrity, and MISRA check groups.

- The value `maab` corresponds to a subset of the MAAB checks.
- The value `hisl_do178` corresponds to a subset of the High-Integrity System checks.
- The value `_SYSTEM_By Task_misra_c` is the **Check Group ID** for the MISRA check group Modeling Standards for MISRA C:2012.

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

To obtain the Model Advisor **Check Group ID** for a group of checks, open the Model Advisor Configuration Editor and select the folder that contains the desired group of checks. The **Check Group ID** is shown in the **Information** tab. For more information on the Model Advisor Configuration Editor, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.

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

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

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

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

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

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

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

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

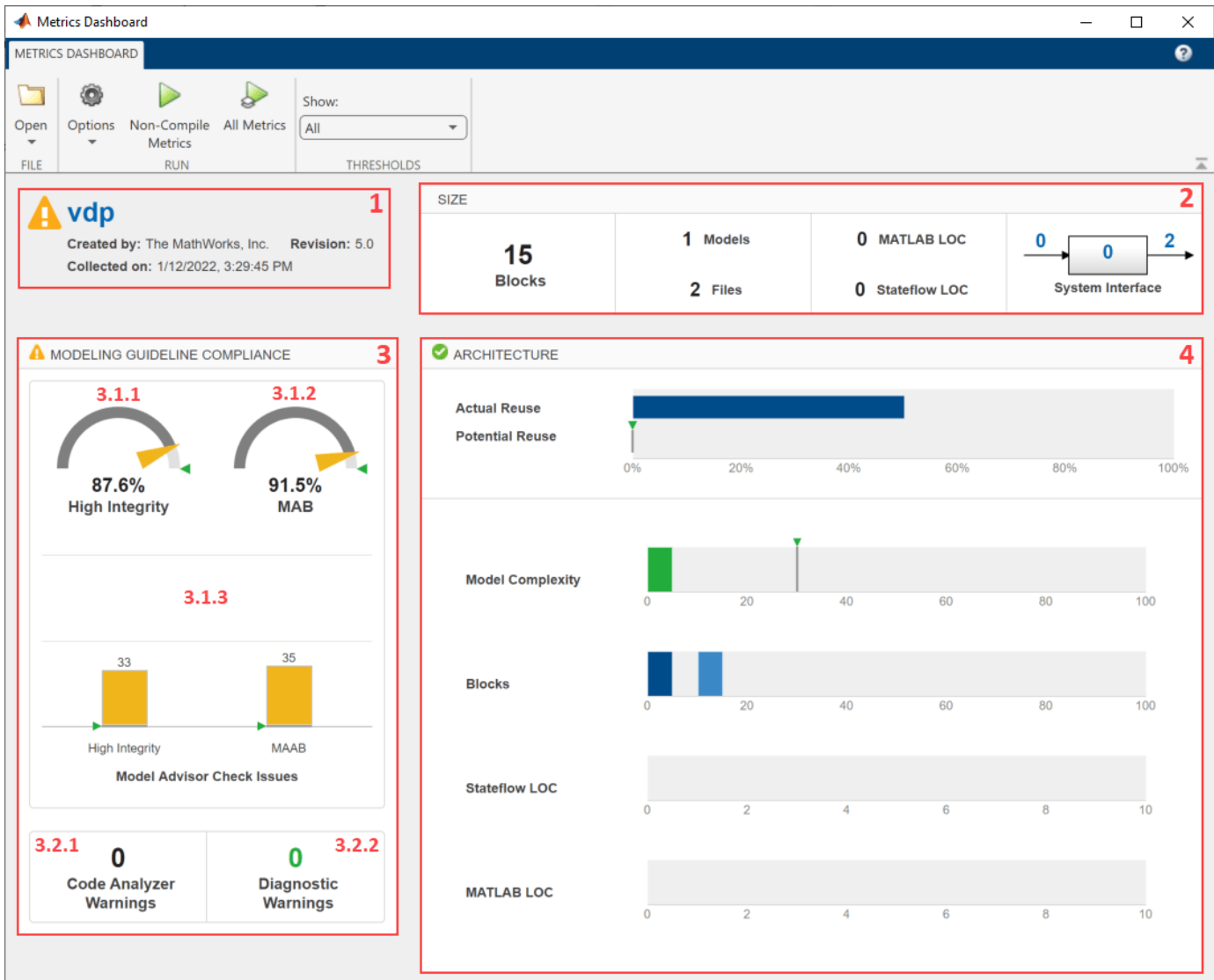
8. Obtain the compliance group from the layout.

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

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

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

In the diagram, the red numbers 1, 2, 3, and 4 indicate their order in the `layoutWidget` array.



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

Remove the **High Integrity** compliance widget.

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

10. The **Metric ID** for the configured MISRA check compliance metric is 'mathworks.metrics.ModelAdvisorCheckCompliance._SYSTEM_By Task_misra_c'.

```
misraComplianceMetricID = 'mathworks.metrics.ModelAdvisorCheckCompliance._SYSTEM_By Task_misra_c';
```

The **Metric ID** for a configured check compliance metric is of the form <Family ID>.<Model Advisor Check Group ID>.

- Metrics configured for Model Advisor compliance use the **<Family ID>** `mathworks.metrics.ModelAdvisorCheckCompliance`. Configured check compliance metrics calculate the fraction of Model Advisor checks that pass for the selected Model Advisor **Check Group ID**.
- The Model Advisor **Check Group ID**, `_SYSTEM_By Task_misra_c`, is the **Check Group ID** for the MISRA check group Modeling Standards for MISRA C:2012.

To obtain the Model Advisor **Check Group ID** for a group of checks, open the Model Advisor Configuration Editor and select the folder that contains the desired group of checks. The **Check Group ID** is shown in the **Information** tab. For more information on the Model Advisor Configuration Editor, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.

For more information on configured compliance metrics, see “Model Metrics”.

11. Create a custom widget for visualizing MISRA check compliance metrics.

```
misraWidget = complianceContainers(1).addWidget('Custom', 1);
misraWidget.Title = ('MISRA');
misraWidget.VisualizationType = 'RadialGauge';
misraWidget.setMetricIDs(misraComplianceMetricID);
misraWidget.setWidths(slmetric.dashboard.Width.Medium);
```

12. The bar chart widget currently visualizes the High Integrity and MAAB check groups. Point this widget to the **Metric IDs** for the MISRA check issues and MAAB check issues.

```
misraIssuesMetricID = 'mathworks.metrics.ModelAdvisorCheckIssues._SYSTEM_By Task_misra_c';
maabIssuesMetricID = 'mathworks.metrics.ModelAdvisorCheckIssues.maab';
```

```
setMetricIDs(complianceContainerWidgets(3),...
({misraIssuesMetricID,maabIssuesMetricID}));
complianceContainerWidgets(3).Labels = {'MISRA', 'MAAB'};
```

The **Metric ID** for a configured check compliance metric is of the form **<Family ID>**.**<Model Advisor Check Group ID>**.

- Metrics configured for Model Advisor compliance issues use the **<Family ID>** `mathworks.metrics.ModelAdvisorCheckIssues`. Configured check compliance issues metrics calculate the number of issues reported by the selected Model Advisor **Check Group ID**.
- The Model Advisor **Check Group ID**, `_SYSTEM_By Task_misra_c`, is the **Check Group ID** for the MISRA check group Modeling Standards for MISRA C:2012. `maab` is a **Check Group ID** that corresponds to a subset of MAAB checks.

To obtain the Model Advisor **Check Group ID** for a group of checks, open the Model Advisor Configuration Editor and select the folder that contains the desired group of checks. The **Check Group ID** is shown in the **Information** tab. For more information on the Model Advisor Configuration Editor, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.

For more information on configured compliance metrics, see “Model Metrics”.

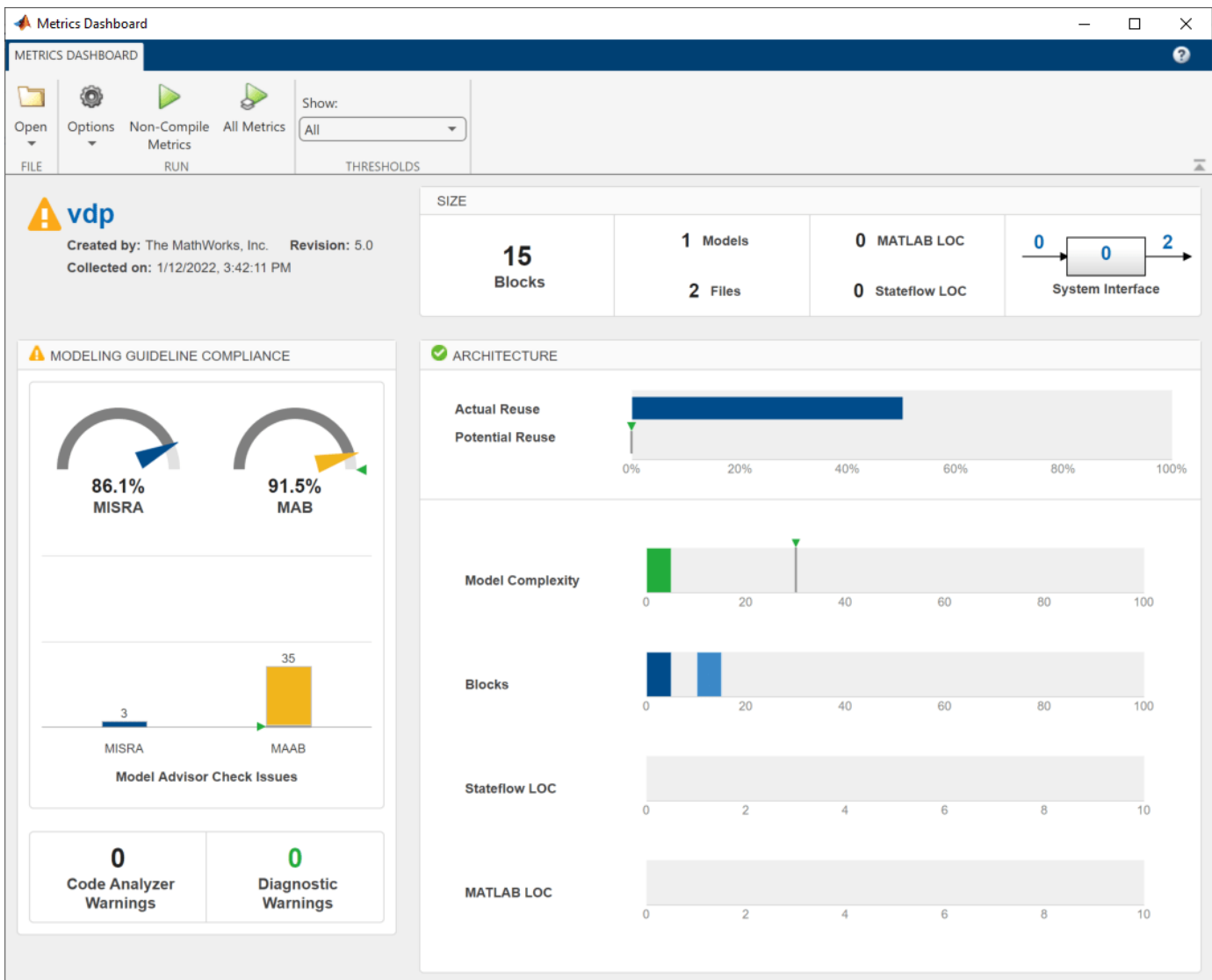
13. To run and view the Metrics Dashboard at this point in the example, enter the following lines of code in the MATLAB Command Window. The `save` commands serialize the API information to XML files. The `slmetric.config.setActiveConfiguration` and `slmetric.dashboard.setActiveConfiguration` commands set the active configuration objects.

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

14. To open the Metrics Dashboard, enter the following code in the MATLAB Command Window.

```
metricsdashboard vdp
```

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



16. Close the Metrics Dashboard.

Add a Custom Metric to Dashboard

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

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

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

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

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

3. Register the new metric in the metric repository.

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

The new nonvirtual block count metric has the metric ID `nonvirtualblockcount`.

To view the available metrics for your metric engine, use

```
slmetric.metric.getAvailableMetrics.
```

```
availableMetricIDs = slmetric.metric.getAvailableMetrics
```

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

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

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

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

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

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

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

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

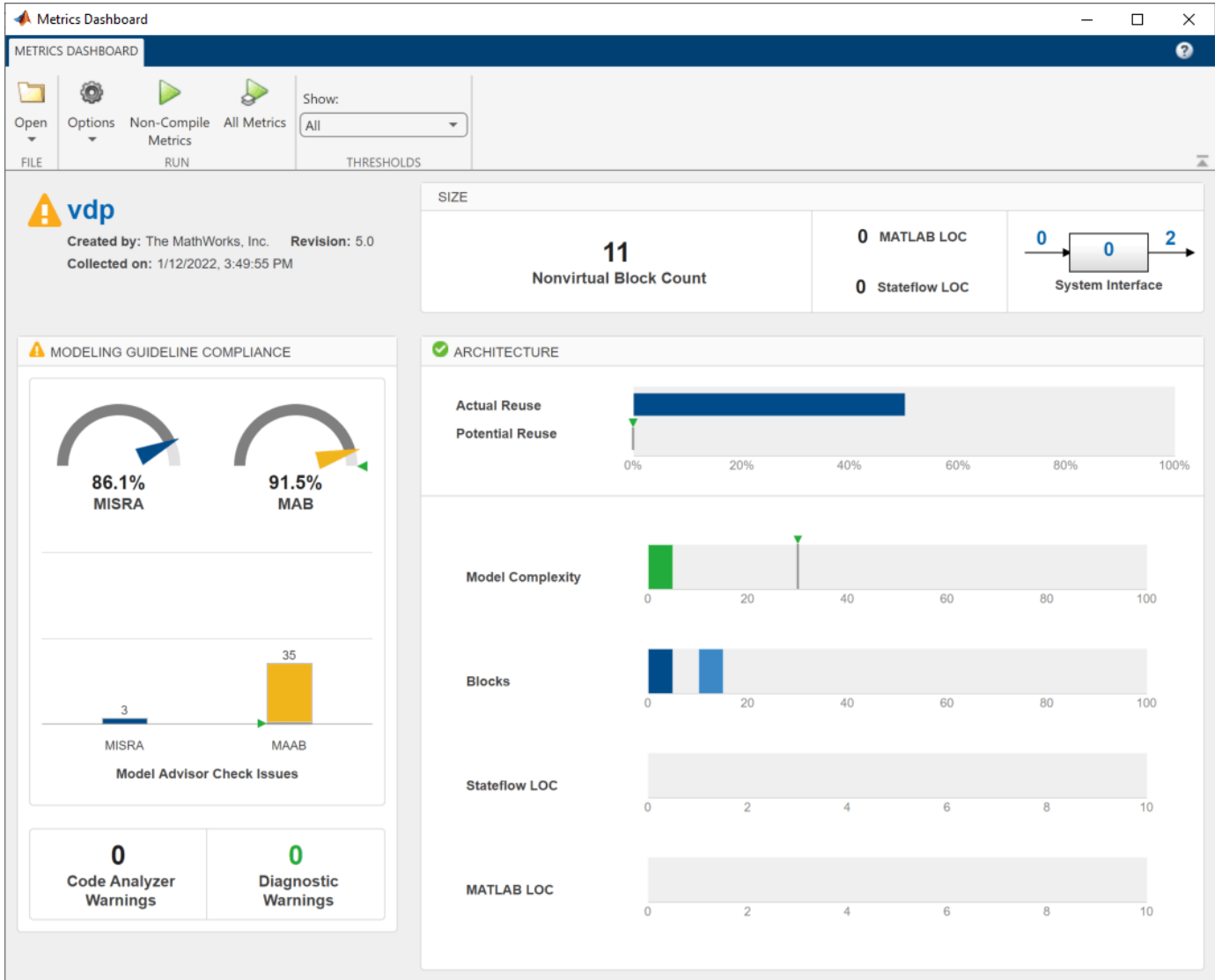


```
slmetric.config.setActiveConfiguration(fullfile(pwd, 'MetricConfig.xml'));
slmetric.dashboard.setActiveConfiguration(fullfile(pwd, 'DashboardConfig.xml'));
```

8. To open the Metrics Dashboard, enter the following code in the MATLAB Command Window.

```
metricsdashboard vdp
```

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



10. Close the Metrics Dashboard.

Add Metric Thresholds

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

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

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

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

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

```
T1 = addThreshold(TC, misraIssuesMetricID, 'AggregatedValue');  
C = getClassifications(T1);  
C.Range.Start = -inf;  
C.Range.End = 0;  
C.Range.IncludeStart = 0;  
C.Range.IncludeEnd = 1;
```

```
T2 = addThreshold(TC, misraComplianceMetricID, 'AggregatedValue');  
C = getClassifications(T2);  
C.Range.Start = 1;  
C.Range.End = inf;  
C.Range.IncludeStart = 1;  
C.Range.IncludeEnd = 0;
```

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

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

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

These commands specify that if the model has MISRA check issues, the model status is warning. If the model does not have MISRA check issues, the model status is compliant.

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

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

```
C = addClassification(T3, 'Warning');  
C.Range.Start = 20;
```

```
C.Range.End = 30;  
C.Range.IncludeStart = 0;  
C.Range.IncludeEnd = 1;
```

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

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

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

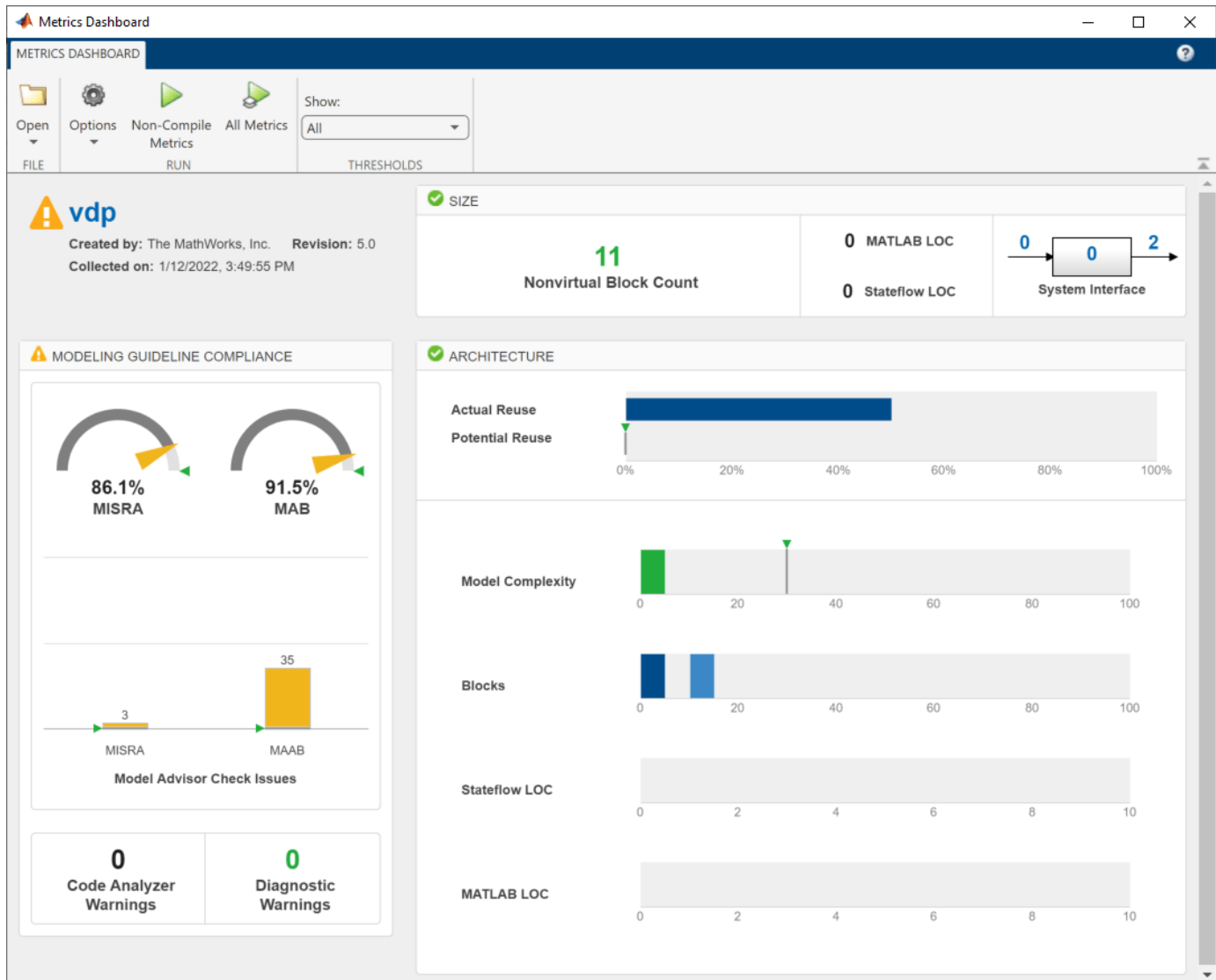
```
save(metricconfig, 'FileName', 'MetricConfig.xml');  
save(dashboardconfig, 'Filename', 'DashboardConfig.xml');
```

6. Set the active configurations.

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

7. For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```



For the MISRA check compliance issues, the gauge is yellow because 86.1% of the checks pass. A percentage less than 100% generates a warning. The bar chart also displays a yellow because the model contains three MISRA check issues. A number greater than zero generates a warning.

The **Nonvirtual Block Count** widget is in the compliant range because there are 11 nonvirtual blocks.

8. To reset the configuration and unregister the metric, enter the following lines of code in the MATLAB Command Window.

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

See Also

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

More About

- “Model Metrics”
- “Collect and Explore Metric Data by Using the Metrics Dashboard” on page 5-2
- “Rearrange and Remove Widgets in Metrics Dashboard” on page 5-48
- “Create Layout with Custom Metric” on page 5-55
- “Modify, Remove, and Add Metric Thresholds in Metrics Dashboard” on page 5-62

Rearrange and Remove Widgets in Metrics Dashboard

This example shows how to use the metric APIs to rearrange existing widgets in the default layout of the Metrics Dashboard and remove widgets from the Metrics Dashboard.

Rearrange Widgets in Layout

Suppose you want to take the default layout of the Metrics Dashboard and modify the layout to move the bar charts for **Actual Reuse** and **Potential Reuse** to the bottom of the **Architecture** section.

1. Open the default configuration for the Metrics Dashboard.

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

2. Get the dashboard layout from the configuration.

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

Dashboard layouts contain nested layers of objects. You can use the `getWidgets` function to get the objects in the next layer of the layout. For more information, see `getWidgets`.

3. To return the sections in the dashboard layout, use the `getWidgets` function.

```
sections = getWidgets(layout)
```

```
sections=1x4 object
```

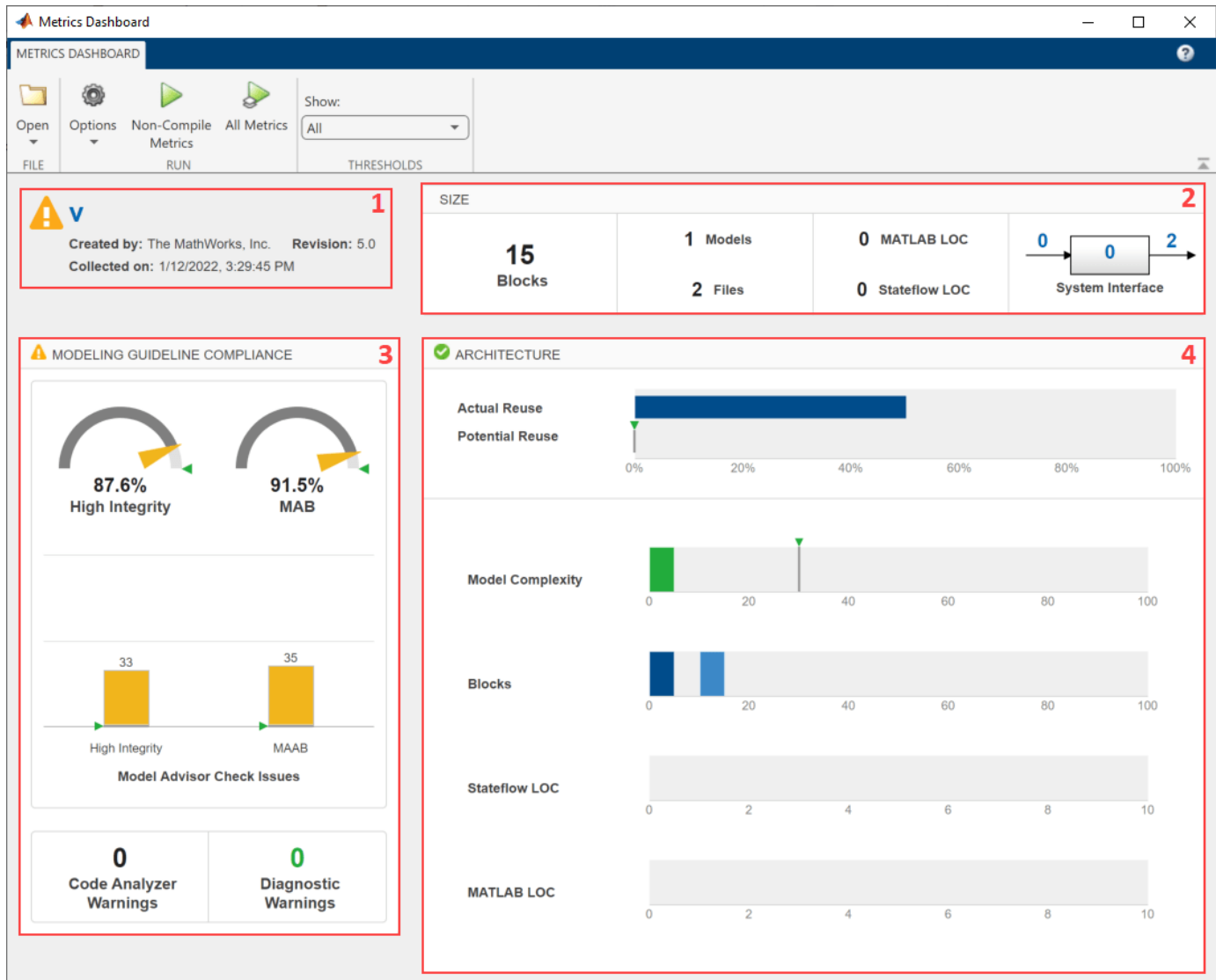
```
1x4 heterogeneous Container (Container, Group) array with properties:
```

```
ShowBorder  
Type  
ID
```

`sections` is a 1x4 `Container` because the default layout for the Metrics Dashboard has four main sections:

- `sections(1)` contains the system information for the current model.
- `sections(2)` contains the widgets shown in the **Size** section. The widgets use the Size Metrics from the “Model Metrics”.
- `sections(3)` contains the widgets shown in the **Modeling Guideline Compliance** section. The widgets use the Compliance Metrics from the “Model Metrics”.
- `sections(4)` contains the widgets shown in the **Architecture** section. The widgets use the Architecture Metrics from the “Model Metrics”.

In the diagram, the red numbers 1, 2, 3, and 4 indicate the locations of the 4 main sections in the Metrics Dashboard.



4. Save the **Architecture** section to the variable `archSection`. You can access the **Architecture** section by using sections(4).

```
archSection = sections(4);
```

5. To see what the **Architecture** section contains, use the `getWidgets` function.

```
archSectionContents = getWidgets(archSection)
```

```
archSectionContents=1x2 object
```

```
1x2 heterogeneous WidgetBase (Widget, Container) array with properties:
```

```
Type
ID
```

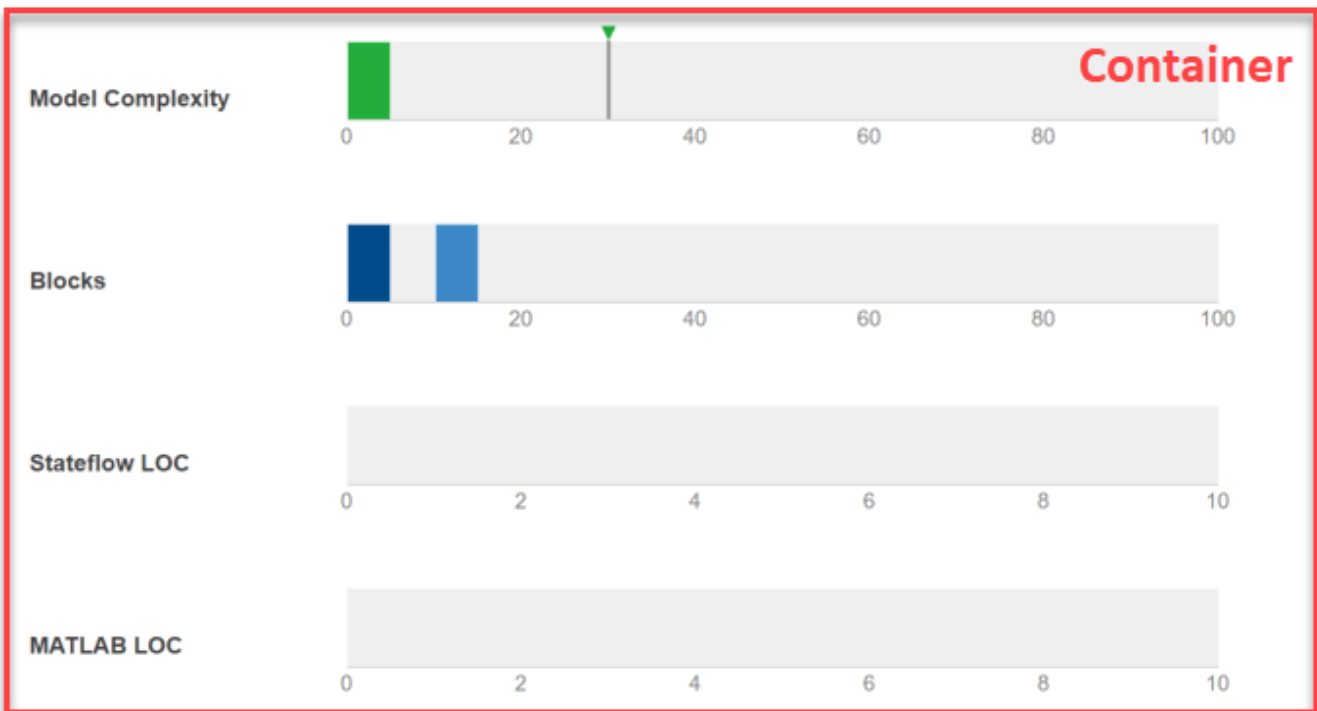
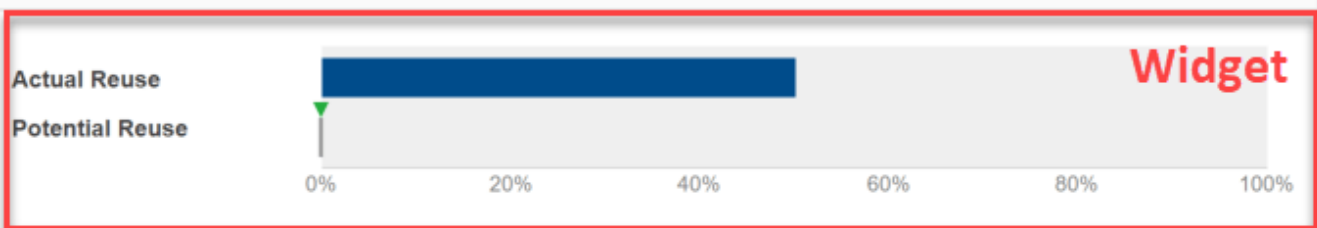
The **Architecture** section contains a `Widget` object and a `Container` object inside an array. The `Widget` is the first element of the array. The `Container` is the second element of the array.

The order of the objects in the array corresponds to the order the objects appear in the dashboard. In the **Architecture** section, the first element appears at the top of the section. The second element appears at the bottom of the section.

In the default configuration:

- The first element of the array corresponds to the **Widget** that creates the bar charts for **Actual Reuse** and **Potential Reuse**.
- The second element of the array corresponds to the **Container** for the **Model Complexity**, **Blocks**, **Stateflow LOC**, and **MATLAB LOC** widgets. Note that *LOC* refers to lines of code. For more information, see “Model Metrics”.

✓ ARCHITECTURE



6. Save the first element of the array to the variable `reuseWidget`.

```
reuseWidget = archSectionContents(1) % widget
```

```
reuseWidget =  
    Widget with properties:
```



```
Title: 'Library Reuse'
Type: 'LibraryReuse'
ID: '94b6e8b1-c79a-43c6-9099-87a0fa1d7521'
```

The Type of reuseWidget is 'LibraryReuse'. The 'LibraryReuse' type of widget creates the **Actual Reuse** and **Potential Reuse** bar charts.

7. Use the getPosition function to confirm that the reuseWidget widget is currently in the first position inside the **Architecture** section.

```
getPosition(reuseWidget)
```

```
ans = 1
```

8. Move the reuseWidget widget to the bottom of the **Architecture** section by setting the position of the reuseWidget object to be the second element in the array of objects in the section.

```
setPosition(reuseWidget,2);
```

9. Save the updated configuration, dashboardconfig, as an XML file. The XML file contains the new, rearranged layout for the **Architecture** section.

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

10. Set the XML file as the active configuration for the Metrics Dashboard.

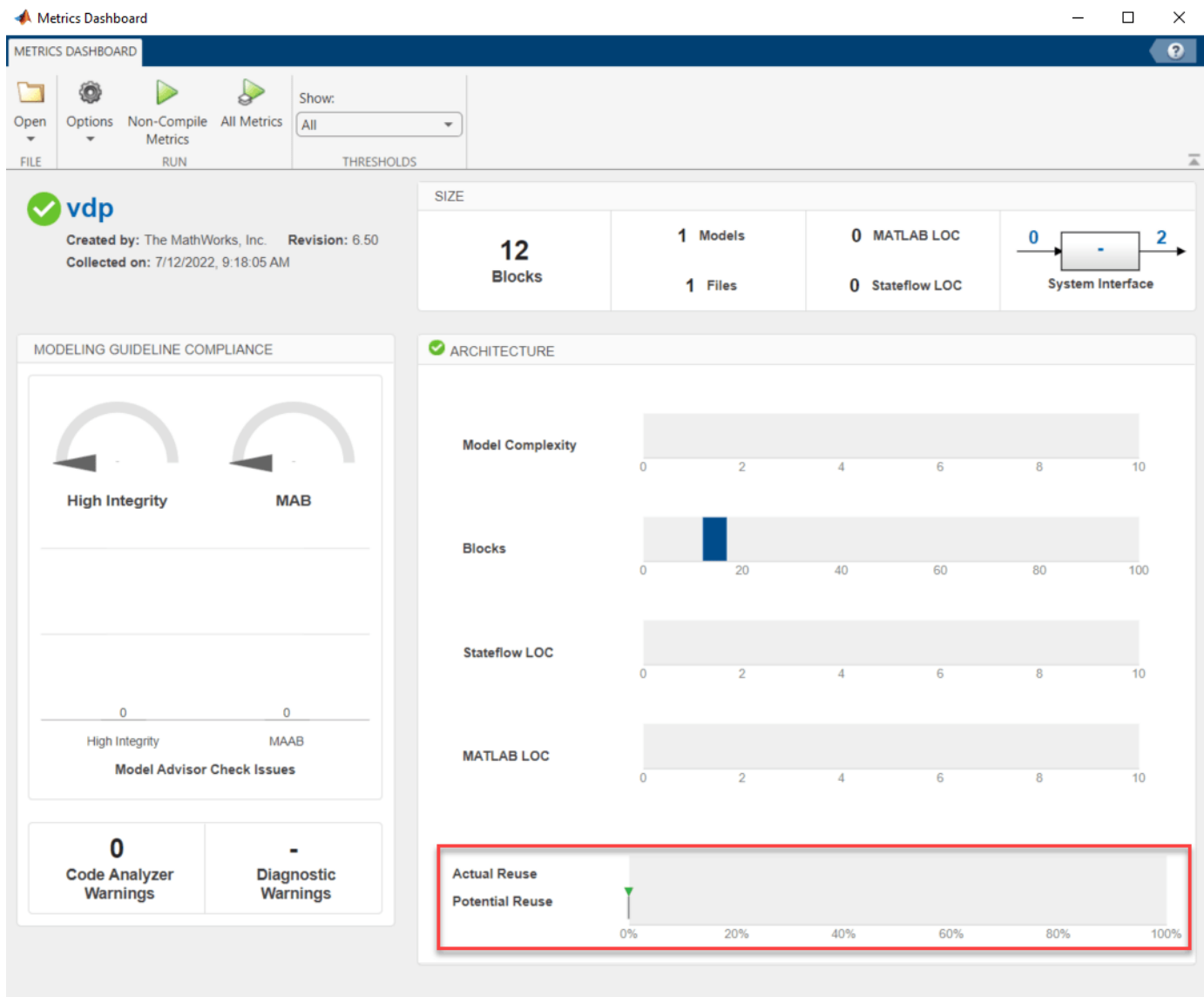
```
slmetric.dashboard.setActiveConfiguration(fullfile(pwd, 'DashboardConfig.xml'));
```

11. The Metrics Dashboard uses the active configuration the next time you open the dashboard on a model.

```
metricsdashboard vdp
```

Warning: The Metrics Dashboard and slmetric.Engine API will be removed in a future release. For size, architecture, and complexity metrics, use the Model Maintainability Dashboard and metric. The Model Maintainability Dashboard and metric.Engine API can identify outdated metric results, and For more information, see [matlab:helpview\(\[docroot 'slcheck/collect-model-metric-data-](matlab:helpview([docroot 'slcheck/collect-model-metric-data-3)

Since the reuseWidget widget is set to the second position, the bar charts for **Actual Reuse** and **Potential Reuse** now appear at the bottom of the **Architecture** section.



12. Close the Metrics Dashboard.

Remove Widgets from Layout

Suppose you want to remove the **Stateflow LOC** and **MATLAB LOC** widgets from the **Architecture** section of the Metrics Dashboard.

1. Get the new dashboard layout for the configuration that you updated in the previous section.

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

2. Use the `getWidgets` function to return the sections in the dashboard layout.

```
sections = getWidgets(layout);
```

3. Save the **Architecture** section to the variable `archSection`.

```
archSection = sections(4);
```

4. Use the `getWidgets` function to see what the **Architecture** section contains.

```
archSectionContents = getWidgets(archSection)
```

```
archSectionContents=1x2 object
```

```
1x2 heterogeneous WidgetBase (Container, Widget) array with properties:
```

```
Type
ID
```

Since you rearranged the widgets in the previous section, the **Container** is now the first element in the array. The **Container** contains the **Model Complexity**, **Blocks**, **Stateflow LOC**, and **MATLAB LOC** widgets.

5. Save the **Container** to the variable `archContainer`.

```
archContainer = archSectionContents(1);
```

6. Use the `getWidgets` function to get the widgets inside the **Container**.

```
archContainerWidgets = getWidgets(archContainer);
```

`archContainerWidgets` is a 1x4 **CustomWidget** array where:

- `archContainerWidgets(1)` contains the widget for **Model Complexity**
- `archContainerWidgets(2)` contains the widget for **Blocks**
- `archContainerWidgets(3)` contains the widget for **Stateflow LOC**
- `archContainerWidgets(4)` contains the widget for **MATLAB LOC**

7. Remove the **Stateflow LOC** widget from the **Container**.

```
removeWidget(archContainer,archContainerWidgets(3));
```

8. Remove the **MATLAB LOC** widget from the **Container**.

```
removeWidget(archContainer,archContainerWidgets(4));
```

9. Save the updated configuration, `dashboardconfig`, as an XML file. The XML file contains the new, rearranged layout for the **Architecture** section.

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

10. Set the XML file as the active configuration for the Metrics Dashboard.

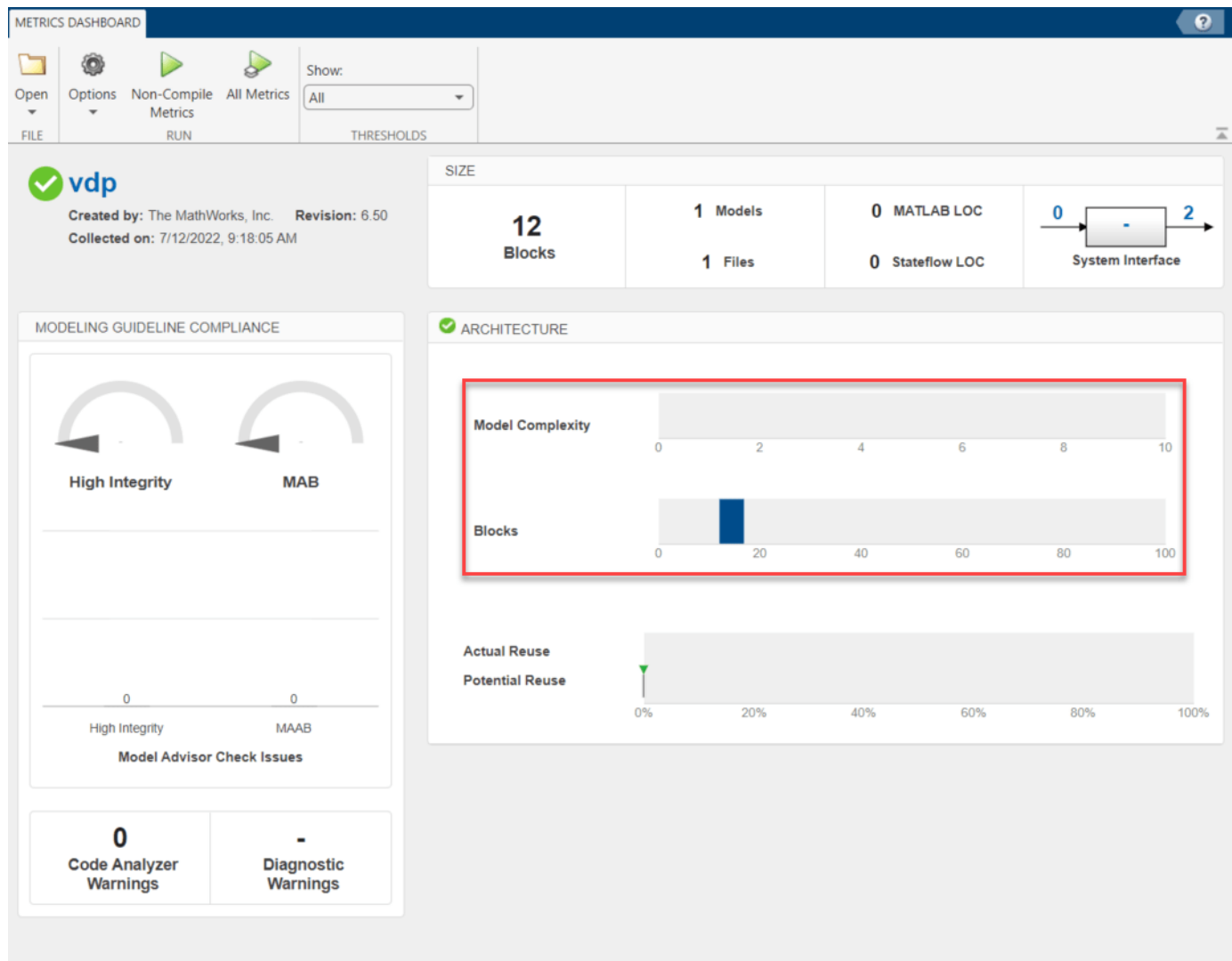
```
slmetric.dashboard.setActiveConfiguration(fullfile(pwd, 'DashboardConfig.xml'));
```

11. The Metrics Dashboard uses the active configuration the next time you open the dashboard on a model.

```
metricsdashboard vdp
```

Warning: The Metrics Dashboard and `slmetric.Engine` API will be removed in a future release. For size, architecture, and complexity metrics, use the Model Maintainability Dashboard and `metric`. The Model Maintainability Dashboard and `metric.Engine` API can identify outdated metric results, and for more information, see [matlab:helpview\(\[docroot '/slcheck/collect-model-metric-data-3'\]\)](matlab:helpview([docroot '/slcheck/collect-model-metric-data-3'])).

The **Architecture** section no longer shows the **Stateflow LOC** and **MATLAB LOC** widgets.



See Also

getWidgets | removeWidget | slmetric.dashboard.Configuration

Related Examples

- “Model Metrics”
- “Create Layout with Custom Metric” on page 5-55
- “Modify, Remove, and Add Metric Thresholds in Metrics Dashboard” on page 5-62

Create Layout with Custom Metric

This example shows how to create a custom metric and modify the layout for the Metrics Dashboard to show only information about the model, results from the custom metric, and other custom widgets.

Create Custom Metric

You can define a custom metric by creating a new metric class and specifying what the metric calculates inside the `algorithm` function of the metric class.

1. Create a new metric class named `nonvirtualblockcount`.

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

The `createNewMetricClass` function creates a class named `nonvirtualblockcount.m` in the current working folder.

2. For this example, copy the contents of the example custom metric defined in `nonvirtualblockcount_orig.m` into the metric class file `nonvirtualblockcount.m`.

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

The `algorithm` function inside the class definition defines what the custom metric calculates. The example file `nonvirtualblockcount_orig.m` contains an algorithm for a metric that counts the nonvirtual blocks in the current model.

3. Register the custom metric with the model metric repository. The Metrics Dashboard can only use metrics that are available from the model metric repository.

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

Note that the custom metric stays in the model metric repository until you unregister the metric by using the `slmetric.metric.unregisterMetric` function.

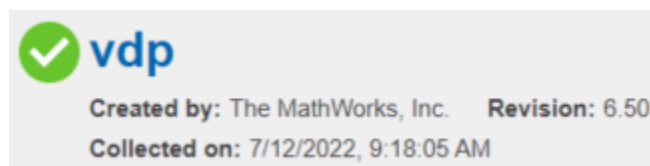
Create Layout with Custom Widget

Metrics Dashboard Layout

At a minimum, the Metrics Dashboard requires that a layout contain a `'SystemInfo'` widget and at least one widget of type `slmetric.dashboard.Widget` or `slmetric.dashboard.CustomWidget`.

The `'SystemInfo'` widget is the widget that shows information about the current system, including:

- The name of the current system.
- The author of the model.
- The revision of the current model.
- When you collected metric results.



1. Create a new configuration object. The configuration contains the Layout that the Metrics Dashboard uses to arrange the widgets. For this example, specify the Name of the new configuration as "Minimal". For more information on configuration objects, see `slmetric.dashboard.Configuration`.

```
dashboardconfig = slmetric.dashboard.Configuration.new(Name='Minimal')

dashboardconfig =
  Configuration with properties:
    Name: 'Minimal'
    FileName: ''
    Location: ''
```

2. Get the dashboard layout from the configuration.

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

3. Use the `addWidget` function to add a 'SystemInfo' widget to the layout.

```
addWidget(layout, 'SystemInfo')

ans =
  Widget with properties:
    Title: ''
    Type: 'SystemInfo'
    ID: '63a87dd2-4d78-46b9-928c-2317fc021425'
```

Display Single Value

By default, custom widgets display a single, integer value next to the widget title.

11 Nonvirtual Block Count

1. Add a custom widget, `customWidget`, to the Layout by specifying the `widgetType` as 'Custom'.

```
customWidget = addWidget(layout, 'Custom');
```

By default, custom widgets specify the `VisualizationType` 'SingleValue' to display a single, integer value in the dashboard.

2. Assign the custom metric 'nonvirtualblockcount' to the custom widget.

```
customWidget.setMetricIDs('nonvirtualblockcount');
```

3. Specify a title for the custom widget.

```
customWidget.Title = 'Nonvirtual Block Count'
```

```
customWidget =
  CustomWidget with properties:
    VisualizationType: 'SingleValue'
    Labels: {0x1 cell}
```

```
Title: 'Nonvirtual Block Count'
Type: 'Custom'
ID: 'd1d33f3f-6252-4979-8ea1-70c7c77dd81a'
```

Use New Layout in Metrics Dashboard

1. Save the updated Configuration, `dashboardconfig`, as an XML file. The XML file contains the new, minimal layout for the dashboard.

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

2. Set the XML file as the active configuration for the Metrics Dashboard.

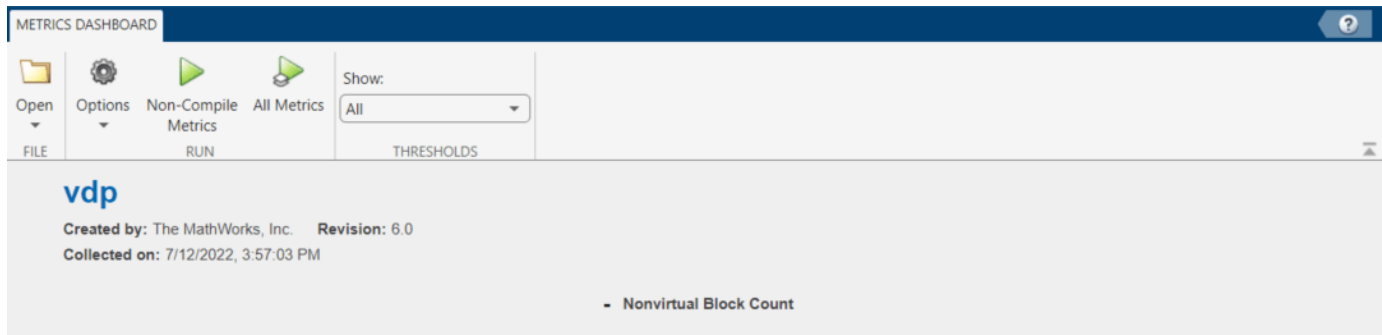
```
slmetric.dashboard.setActiveConfiguration(fullfile(pwd, 'DashboardConfig.xml'));
```

3. Open the Metrics Dashboard for the model `vdp`.

```
metricsdashboard vdp
```

Warning: The Metrics Dashboard and `slmetric.Engine` API will be removed in a future release. For size, architecture, and complexity metrics, use the Model Maintainability Dashboard and `metric.Engine` API. The Model Maintainability Dashboard and `metric.Engine` API can identify outdated metric results, and for more information, see [matlab:helpview\(\[docroot '/slcheck/collect-model-metric-data-'\]](matlab:helpview([docroot '/slcheck/collect-model-metric-data-'])

The Metrics Dashboard shows only the 'SystemInfo' widget and the custom widget for the custom metric 'nonvirtualblockcount'.



Change How Widgets Display Metric Results

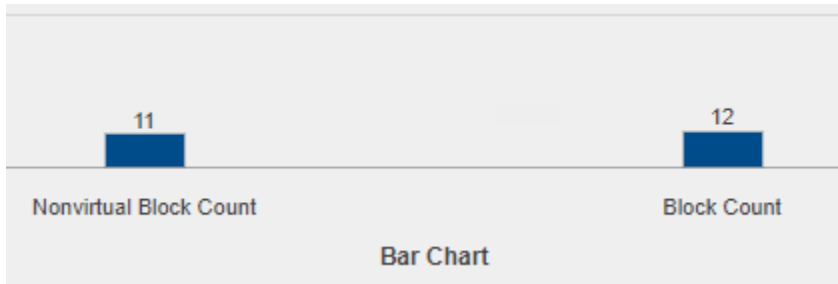
By default, custom widgets use the `VisualizationType` 'SingleValue'.

When you run the metrics for a 'SingleValue' widget, the widget only displays a single, integer value next to the widget Title.

11 Nonvirtual Block Count

However, you can change the visualization type for a custom widget to a bar chart, radial gauge, or distribution heatmap. For more information, see `slmetric.dashboard.CustomWidget`.

Display Bar Chart



To make a custom widget display metric results in a bar chart:

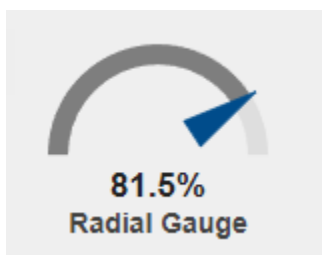
- Specify the `VisualizationType` as `'BarChart'`.
- Use the `Labels` property to provide labels for each bar in the bar chart. The label must be specified by a cell array and the number of labels must equal the number of metric IDs associated with the widget.
- Use the `setHeight` function to set the height of the widget and make the widget appear in the layout.

Add a new custom widget, `barChartWidget`, to the `Layout`.

```
barChartWidget = addWidget(layout, 'Custom');
barChartWidget.VisualizationType = 'BarChart';
barChartWidget.Title = 'Bar Chart';
barChartWidget.setMetricIDs({'nonvirtualblockcount', 'mathworks.metrics.SimulinkBlockCount'})
barChartWidget.Labels = {'Nonvirtual Block Count', 'Block Count'};
barChartWidget.setHeight(220);
```

For this example, the custom widget shows one bar for each metric ID. The bar chart uses the metric IDs for the custom metric `'nonvirtualblockcount'` and the Simulink® Check™ metric `'mathworks.metrics.SimulinkBlockCount'`. To see a list of the available metrics, use the `slmetric.metric.getAvailableMetrics` function. For more information, see “Model Metrics”.

Display Radial Gauge



To make a custom widget display metric results in a radial gauge:

- Specify the `VisualizationType` as `'RadialGauge'`.
- Provide a metric result value between 0 and 1. The radial gauge supports only values between 0 and 1.

Create a new metric class named `randomnumber`.

```
slmetric.metric.createNewMetricClass('randomnumber');
```


For this example, copy the contents of the example custom metric defined in `randomnumber_orig.m` into the metric class file `randomnumber.m`.

```
copyfile randomnumber_orig.m randomnumber.m f
```

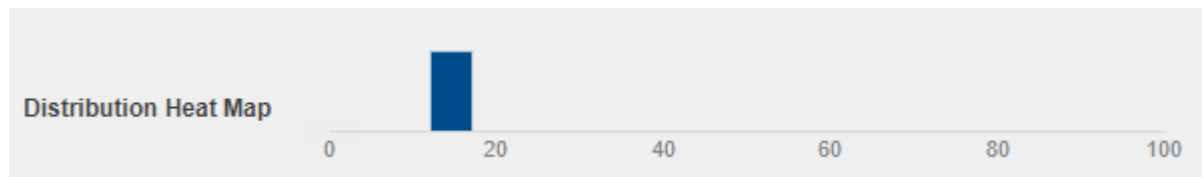
Register the example custom metric. The `algorithm` function in the custom metric definition specifies a metric value, `res.Value`, that is a random number between 0 and 1.

```
slmetric.metric.registerMetric('randomnumber');
```

Add a new custom widget, `gaugeWidget`, to the Layout.

```
gaugeWidget = addWidget(layout, 'Custom');
gaugeWidget.VisualizationType = 'RadialGauge';
gaugeWidget.Title = 'Radial Gauge';
gaugeWidget.setMetricIDs('randomnumber');
```

Display Distribution Heatmap



To make a custom widget display metric results in a distribution heatmap:

- Specify the `VisualizationType` as `'DistributionHeatmap'`.
- Provide a metric result value as an array of positive, integer values. The distribution heatmap supports only an array of positive, integer values.

Add a new custom widget, `heatmapWidget`, to the Layout.

```
heatmapWidget = addWidget(layout, 'Custom');
heatmapWidget.VisualizationType = 'DistributionHeatmap';
heatmapWidget.Title = 'Distribution Heat Map';
heatmapWidget.setMetricIDs('mathworks.metrics.SimulinkBlockCount');
```

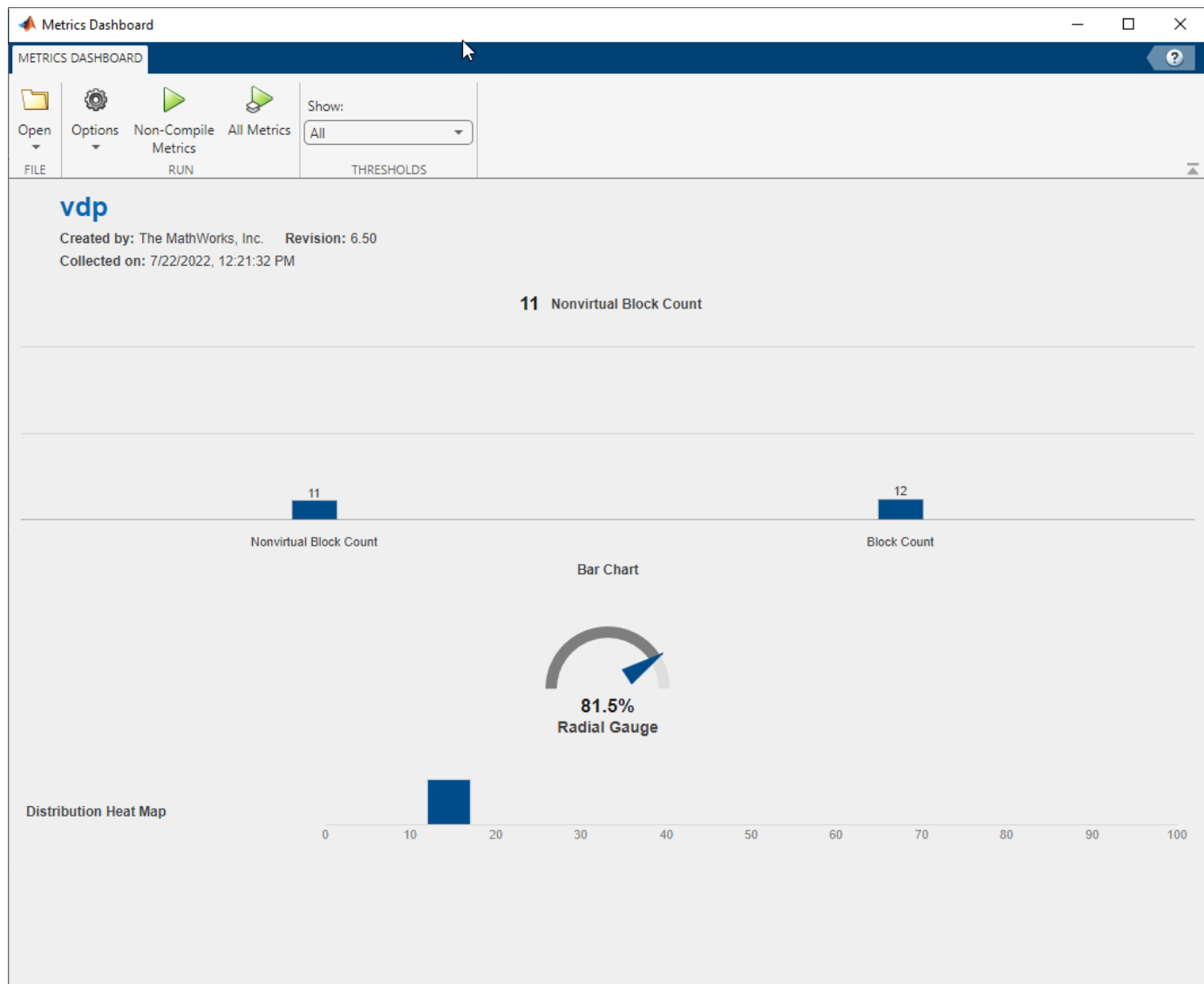
For this example, the custom widget displays the metric results for the metric `'mathworks.metrics.SimulinkBlockCount'`. The metric `'mathworks.metrics.SimulinkBlockCount'` returns the number of blocks in the model and the model subsystems.

Save Updated Layout and View Metrics Dashboard

To save the updated layout and view the Metrics Dashboard, enter:

```
save(dashboardconfig, 'Filename', 'DashboardConfig.xml');
slmetric.dashboard.setActiveConfiguration(fullfile(pwd, 'DashboardConfig.xml'));
metricsdashboard vdp
```

Warning: The Metrics Dashboard and `slmetric.Engine` API will be removed in a future release. For size, architecture, and complexity metrics, use the Model Maintainability Dashboard and `metric.Engine` API. The Model Maintainability Dashboard and `metric.Engine` API can identify outdated metric results, and you can use the `slmetric.dashboard` API to view the outdated results. For more information, see [matlab:helpview\(\[docroot '/slcheck/collect-model-metric-data-](matlab:helpview([docroot '/slcheck/collect-model-metric-data-'])



Unregister Custom Metrics

Note that the custom metrics stay in the model metric repository until you unregister the metrics by using the `slmetric.metric.unregisterMetric` function.

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

See Also

`setMetricIDs` | `slmetric.dashboard.Configuration` |
`slmetric.dashboard.CustomWidget` | `slmetric.dashboard.Widget` |
`slmetric.metric.unregisterMetric`


Related Examples

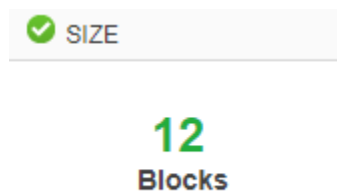
- “Create a Custom Model Metric for Nonvirtual Block Count” on page 5-11
- “Rearrange and Remove Widgets in Metrics Dashboard” on page 5-48
- “Modify, Remove, and Add Metric Thresholds in Metrics Dashboard” on page 5-62


Modify, Remove, and Add Metric Thresholds in Metrics Dashboard

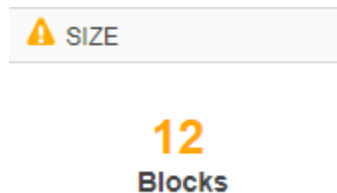
The Metrics Dashboard uses metric threshold values to determine how metric data appears in the dashboard.


Based on the metric threshold values, the dashboard categorizes metric data as:

Compliant — Metric data that is in an acceptable range. The metric data appears green in the widget. Widgets that only contain compliant metric data show a green check mark icon .



Warning — Metric data that requires review. The metric data appears yellow in the widget. Widgets that contain warnings, but no noncompliant metric data, show a yellow triangle icon .



NonCompliant — Metric data that requires you to modify your model. The metric data appears red in the widget. Widgets that contain noncompliant metric data show a red circle icon .



You can use the categorized metric data to assess the design status and quality of your model. Some widgets in the dashboard already contain default metric threshold values. If you want to use threshold values other than the default thresholds, you can customize the metric thresholds.

This example shows how you can customize the metric thresholds to help you identify issues and noncompliant metric data for your model.

Get Default Metric Thresholds

The Metrics Dashboard has two types of configurations:

- `slmetric.dashboard.Configuration` — Dashboard configuration that specifies the layout and types of widgets shown in the Metrics Dashboard.
- `slmetric.config.Configuration` — Metric configuration that specifies thresholds and custom metric families.

In this example, you use an `slmetric.config.Configuration` object to specify thresholds for the Metrics Dashboard.

1. Open the default metric configuration for the Metrics Dashboard.

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

2. Get the threshold configuration from the metric configuration.

```
thresholdConfig = getThresholdConfigurations(metricConfig);
```

3. Use the function `getThresholds` to return the default thresholds.

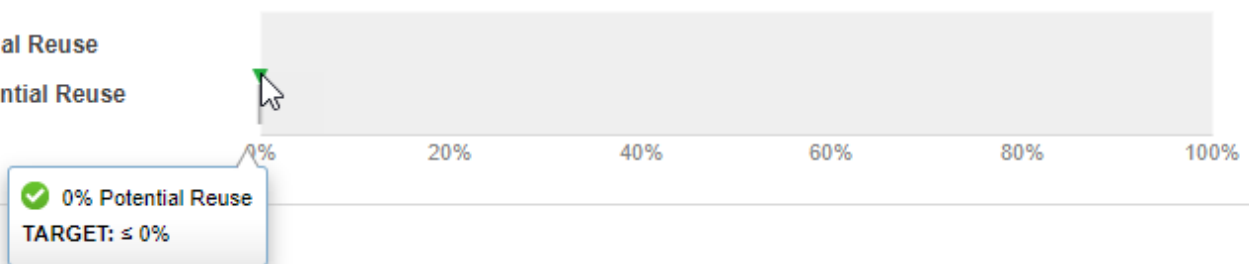
```
defaultThresholds = getThresholds(thresholdConfig);
```

The default thresholds include a threshold for the metric `mathworks.metrics.CloneContent`. The **Potential Reuse** widget displays the metric data provided by `mathworks.metrics.CloneContent`. In the **Architecture** section of the Metrics Dashboard, the **Potential Reuse** results have a default threshold of 0%. The Metrics Dashboard only categorizes the metric data in the **Potential Reuse** widget as compliant if the value is less than or equal to 0%.

✓ ARCHITECTURE

Actual Reuse

Potential Reuse



4. Get the threshold associated with the **Potential Reuse** results.

```
potentialReuse = findobj(defaultThresholds, "MetricID", "mathworks.metrics.CloneContent")
```

```
potentialReuse =  
  Threshold with properties:  
  
  MetricID: 'mathworks.metrics.CloneContent'  
  AppliesTo: 'Value'
```

Modify Existing Metric Threshold

The Metrics Dashboard uses classifications to classify metric data as compliant, warning, or noncompliant for the specified threshold.

Use the function `getClassifications` to get the classifications that the dashboard uses to categorize the **Potential Reuse** results against the threshold.

```
C = getClassifications(potentialReuse);
```

The `Classification` array `C` contains:

- `C(1)` — The classification for compliant **Potential Reuse** results
- `C(2)` — The classification for warning **Potential Reuse** results

Change Compliant Values

1. View the current range for compliant **Potential Reuse** results.

```
C(1).Range
```

```
ans =  
    MetricRange with properties:  
        Start: -Inf  
        End: 0  
    IncludeStart: 0  
    IncludeEnd: 1
```

With the current classification, metric data are compliant if the **Potential Reuse** value is greater than negative infinity and less than or equal to zero. Which means that if the **Potential Reuse** value is 0%, the Metrics Dashboard shows the metric data as compliant. The properties `IncludeStart` and `IncludeEnd` determine whether the range is inclusive or exclusive of the `Start` and `End` values.

2. Modify the range for compliant **Potential Reuse** results. For this example, allow **Potential Reuse** values less than 20% to be compliant.

```
C(1).Range.End = 0.2;
```

Note that the metric `mathworks.metrics.CloneContent` expects the value to be a fraction that represents the total number of subcomponents that are clones, so for this example, the `End` is specified as `0.2`. For more information, see “Model Metrics”.

Change Warning Values

1. View the current range for warning **Potential Reuse** results.

```
C(2).Range
```

```
ans =  
    MetricRange with properties:  
        Start: 0  
        End: Inf  
    IncludeStart: 0  
    IncludeEnd: 0
```

With the current classification, the Metrics Dashboard shows a warning if the **Potential Reuse** value is between zero and infinity. Which means that if the **Potential Reuse** value is greater than 0%, the Metrics Dashboard shows the metric data with a warning.

2. Modify the range for warning **Potential Reuse** results. For this example, specify that the Metrics Dashboard only show a warning if the **Potential Reuse** is greater than 20%.

```
C(2).Range.Start = 0.2;
```

3. View the modified threshold ranges for **Potential Reuse**.

C.Range

```
ans =
  MetricRange with properties:
      Start: -Inf
      End: 0.2000
  IncludeStart: 0
  IncludeEnd: 1
```

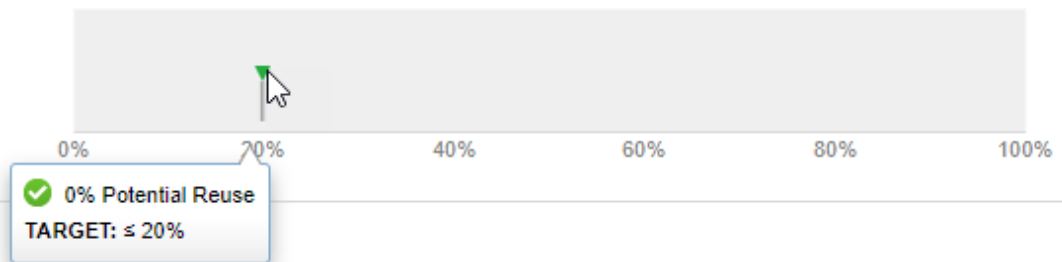
```
ans =
  MetricRange with properties:
      Start: 0.2000
      End: Inf
  IncludeStart: 0
  IncludeEnd: 0
```

Now the Metrics Dashboard shows **Potential Reuse** values less than or equal to 20% as compliant and values greater than 20% with a warning. For the model vdp, the **Potential Reuse** values are less than 20% and are shown as compliant.

✓ ARCHITECTURE

Actual Reuse

Potential Reuse



If you want to view the updated Metrics Dashboard thresholds at this point in the example, enter this code in the MATLAB® Command Window:

```
save(metricConfig, "FileName", "MetricConfig.xml");
slmetric.config.setActiveConfiguration(fullfile(pwd, "MetricConfig.xml"));
metricsdashboard vdp
```

The code saves the updated metric configuration to an XML file, specifies the file as the active configuration, and opens the Metrics Dashboard. In the toolbar of the Metrics Dashboard, click **Non-Compile Metrics** to collect the metric data and view the metric thresholds.

Remove Metric Threshold

Suppose you want to remove the default threshold. You can remove a threshold from the configuration by using the function `removeThreshold`.

Remove the threshold for the **Potential Reuse** results.

```
removeThreshold(thresholdConfig,potentialReuse);
```

Now the metric configuration does not contain a threshold for **Potential Reuse** results, but contains the default thresholds for the other metrics data.

If you want to view the updated Metrics Dashboard thresholds at this point in the example, save the updated metric configuration to an XML file, specify the file as the active configuration, and re-open the Metrics Dashboard. In the toolbar of the Metrics Dashboard, click **Non-Compile Metrics** to collect the metric data and view the metric thresholds.

✓ ARCHITECTURE

Actual Reuse
Potential Reuse



Add Metric Threshold

Suppose you do not want a model to contain more than a specific number of blocks. You can add a threshold to the Simulink® block metric `mathworks.metrics.SimulinkBlockCount` and if a model contains more than the specified number of blocks, the Metrics Dashboard shows the metric data as noncompliant. For this example, you define a threshold so that the Metrics Dashboard shows the block count metric as noncompliant if a model contains more than 11 blocks.

Add a new threshold to the threshold configuration and specify the metric as `"mathworks.metrics.SimulinkBlockCount"` and the property that the metric applies to as `"AggregatedValue"`.

```
customThreshold = addThreshold(thresholdConfig,"mathworks.metrics.SimulinkBlockCount","AggregatedValue");
```

```
customThreshold =  
    Threshold with properties:  
  
    MetricID: 'mathworks.metrics.SimulinkBlockCount'  
    AppliesTo: 'AggregatedValue'
```

For the function `addThreshold`, you can specify the threshold property as either `"Value"` or `"AggregatedValue"`.

In this case, for the metric `mathworks.metrics.SimulinkBlockCount`:

- `Value` — Number of blocks
- `AggregatedValue` — Number of blocks for a component and its subcomponents

For information on the `Value` and `AggregatedValue` properties for other metrics, see “Model Metrics”.

Specify Compliant Values

1. Get the default classifications for the custom threshold. The default classification contains a range of values that the threshold considers compliant.

```
defaultClassification = getClassifications(customThreshold)
```

```
defaultClassification =
  Classification with properties:
    Category: 'Compliant'
    Range: [1x1 slmetric.config.MetricRange]
```

2. View the `Range` property in the classification.

```
defaultClassification.Range
ans =
  MetricRange with properties:
    Start: -Inf
    End: Inf
    IncludeStart: 0
    IncludeEnd: 0
```

By default, the classification shows any value as compliant because the range of compliant values is from negative infinity to infinity.

3. Modify the range of compliant values so that only values less than 10 are compliant.

```
defaultClassification.Range.End = 10;
```

Specify Warning Values

1. Add a classification for metric data that lead to a warning.

```
warningClassification = addClassification(customThreshold, "Warning");
```

2. Modify the range of warning values so that values between 10 and 11 return a warning.

```
warningClassification.Range.Start = 10;
warningClassification.Range.End = 11;
warningClassification.Range.IncludeStart = 1; % greater than or equal to 10
warningClassification.Range.IncludeEnd = 1; % less than or equal to 11
```

The range includes the values 10 and 11 because `IncludeStart` and `IncludeEnd` are set to 1 (true).

Specify Noncompliant Values

1. Add a classification for noncompliant metric data.

```
noncompliantClassification = addClassification(customThreshold, "NonCompliant");
```

2. Modify the range of noncompliant values so that values greater than 11 are noncompliant.

```
noncompliantClassification.Range.Start = 11;  
noncompliantClassification.Range.IncludeStart = 0; % greater than 11 (but not equal to 11)
```

View Updated Metrics Dashboard

1. Save the updated metric configuration as an XML file that contains the updated threshold and classification information.

```
save(metricConfig, "FileName", "MetricConfig.xml");
```

2. Set the XML file as the active metric configuration for the Metrics Dashboard.

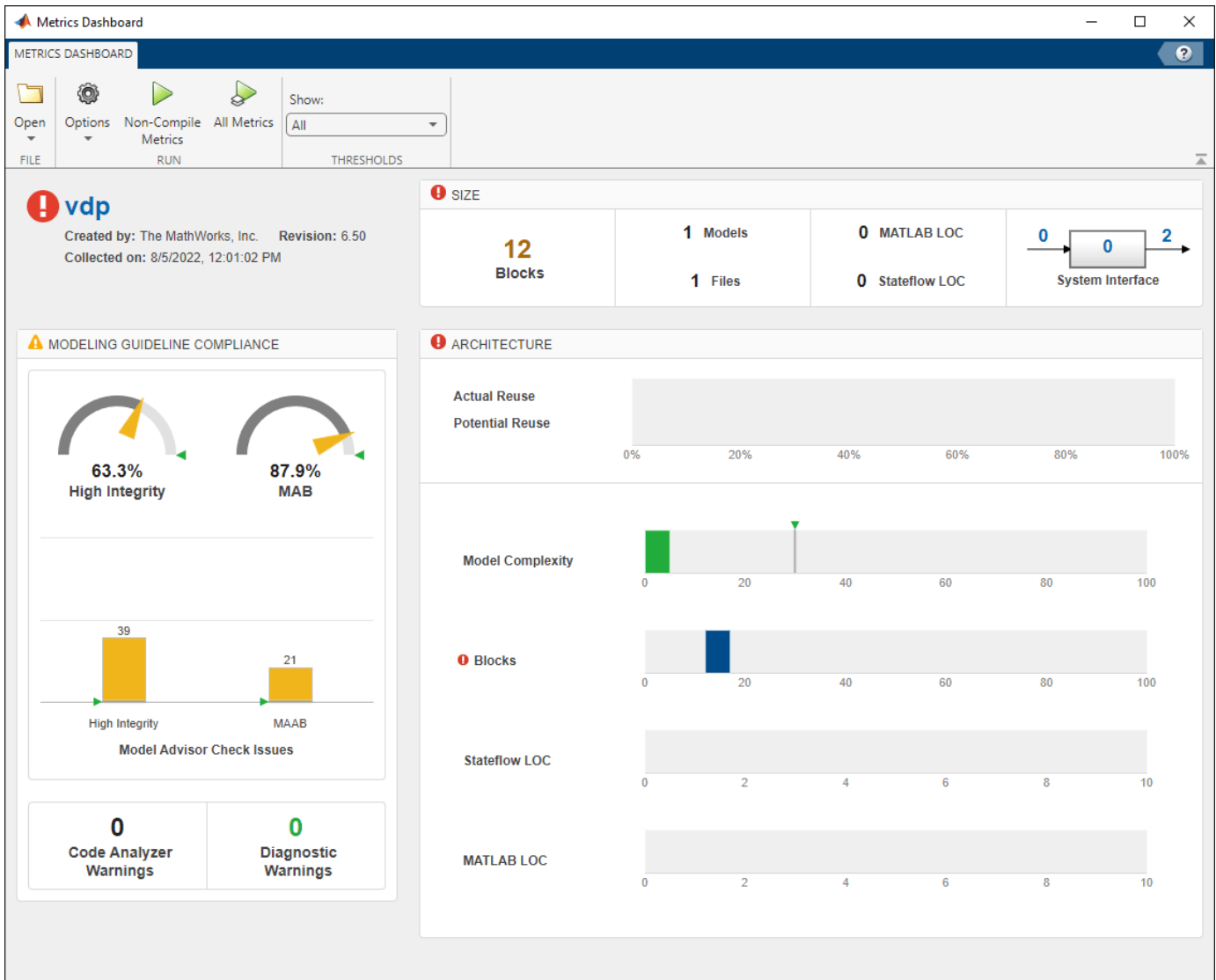
```
slmetric.config.setActiveConfiguration(fullfile(pwd, "MetricConfig.xml"));
```

3. The Metrics Dashboard uses the active configuration the next time you open the dashboard on a model.

```
metricsdashboard vdp
```

4. In the toolbar of the Metrics Dashboard, click **Non-Compile Metrics** to collect the metric data and view the metric thresholds.

Since the model vdp contains more than 11 Simulink blocks, the Metrics Dashboard shows the metric data as noncompliant.



See Also

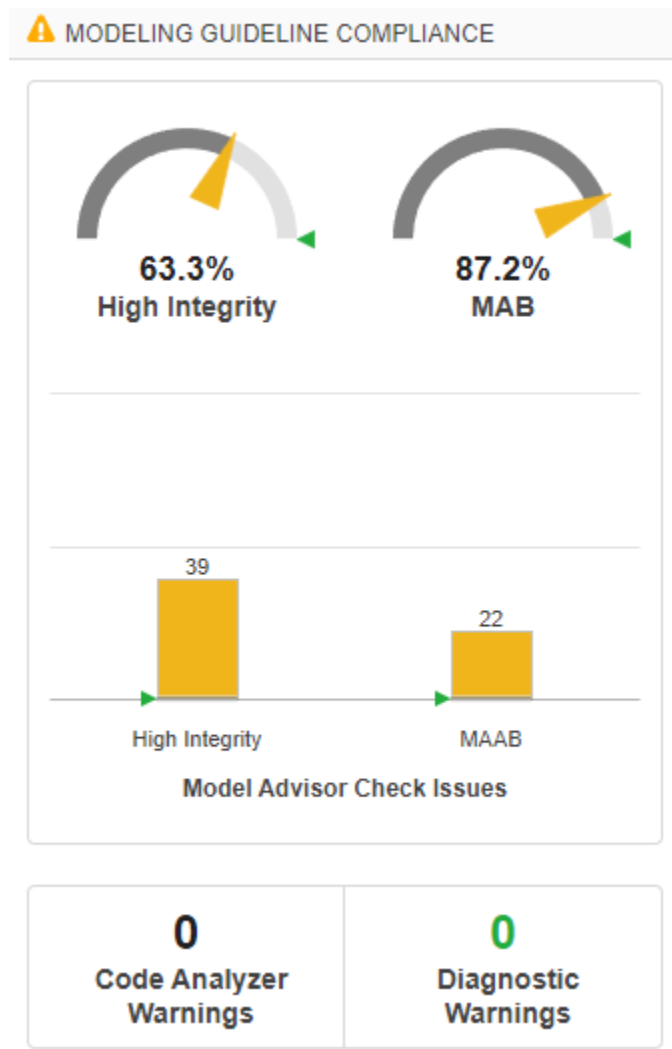
[addThreshold](#) | [getClassifications](#) | [getThresholds](#)

Related Examples

- “Model Metrics”
- “Collect and Explore Metric Data by Using the Metrics Dashboard” on page 5-2
- “Rearrange and Remove Widgets in Metrics Dashboard” on page 5-48
- “Create Layout with Custom Metric” on page 5-55
- “Create a Custom Model Metric for Nonvirtual Block Count” on page 5-11

Change Model Advisor Checks in Metrics Dashboard

The Metrics Dashboard can collect and display metric results that help you to assess the guideline compliance of your model. When you run the metrics in the Metrics Dashboard, the dashboard can run groups of Model Advisor checks, analyze code with MATLAB® Code Analyzer, and report on Simulink® diagnostic results. The dashboard shows these metric results in the **Modeling Guideline Compliance** group. You can use these metric results to monitor warnings and issues with your model compliance.



By default, the dial widgets and bar chart widget in the **Modeling Guideline Compliance** group use metric data from two groups of Model Advisor checks: **High-Integrity Systems** and **Modeling Standards for MAB**. You can customize the **Modeling Guideline Compliance** group to display different metric results, use different metric thresholds, or remove specific widgets.

This example shows how to change the **MAB** dial widget and **MAAB** bar in the bar chart widget to show metric results for a check group other than the default. For information on how to remove

widgets or change metric thresholds, see “Rearrange and Remove Widgets in Metrics Dashboard” on page 5-48 and “Modify, Remove, and Add Metric Thresholds in Metrics Dashboard” on page 5-62.

Create Metric IDs for Custom Model Advisor Metrics

If you want the Metrics Dashboard to collect metric results for a specific Model Advisor check group, create a custom Model Advisor metric that runs the checks and calculates metric results. To create a custom Model Advisor metric, you need to create a metric ID.

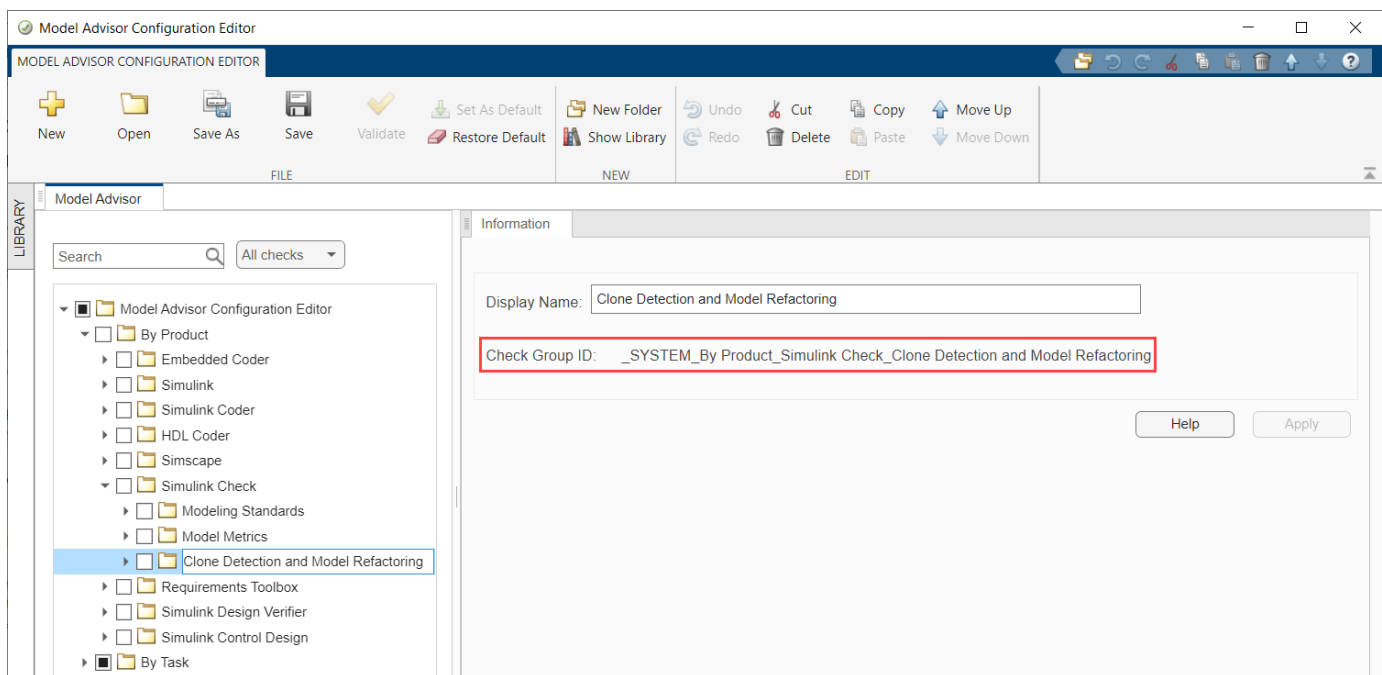
The metric ID for a custom Model Advisor metric must be in the form <Family ID>.<Check Group ID>. The *Family ID* defines the type of data that the metric returns. The *Check Group ID* defines which Model Advisor check group the metric runs and analyzes.

1. Find the Check Group ID for a check group by opening the Model Advisor Configuration Editor and selecting the desired check group folder. You can use the function `Simulink.ModelAdvisor.openConfigUI` to open the Model Advisor Configuration Editor.

`Simulink.ModelAdvisor.openConfigUI`

For this example, suppose you want to find the Check Group ID for the Model Advisor check group **Clone Detection and Model Refactoring**. In the **Model Advisor** pane, select **Model Advisor Configuration Editor > By Product > Simulink Check > Clone Detection and Model Refactoring**. When you select the folder, the **Information** tab shows that the Check Group ID is `_SYSTEM_By Product_Simulink Check_Clone Detection and Model Refactoring`.

For more information on check groups and the Model Advisor Configuration Editor, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.



2. Save the Check Group ID as a string variable in the workspace.

```
checkGroupID = "_SYSTEM_By Product_Simulink Check_Clone Detection and Model Refactoring";
```

When you create the metric ID for your custom Model Advisor metric, you can specify the Family ID as either:

- `mathworks.metrics.ModelAdvisorCheckCompliance` — The metric returns the fraction of passing checks in the check group specified by the Check Group ID.
- `mathworks.metrics.ModelAdvisorCheckIssues` — The metric returns the number of issues reported by the check group specified by the Check Group ID.

3. Create a custom Model Advisor metric ID for a metric that returns the fraction of passing checks in the check group.

```
metricID_for_Fraction_Passed = strcat("mathworks.metrics.ModelAdvisorCheckCompliance.", checkGroupID);
```

4. Create a custom Model Advisor metric ID for a metric that returns the number of issues reported by the check group.

```
metricID_for_Number_of_Issues = strcat("mathworks.metrics.ModelAdvisorCheckIssues.", checkGroupID);
```

Access Widgets You Want to Change

The dashboard configuration determines where and how widgets appear in the Metrics Dashboard. You can create a dashboard configuration object and use the object to access the widgets in the layout.

1. Open the default dashboard configuration and save the configuration object to the base workspace.

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

2. Get the widget layout from the dashboard configuration. The layout contains several layers of dashboard objects.

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

3. Use the `getWidgets` function to get the dashboard objects that are at the top level of the layout.

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

The top level of the default layout contains widgets that show information about the current model and size, architecture, and modeling guideline compliance metrics. The **Modeling Guideline Compliance** group is the third group at the top level of the layout.

4. Get the **Modeling Guideline Compliance** group.

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

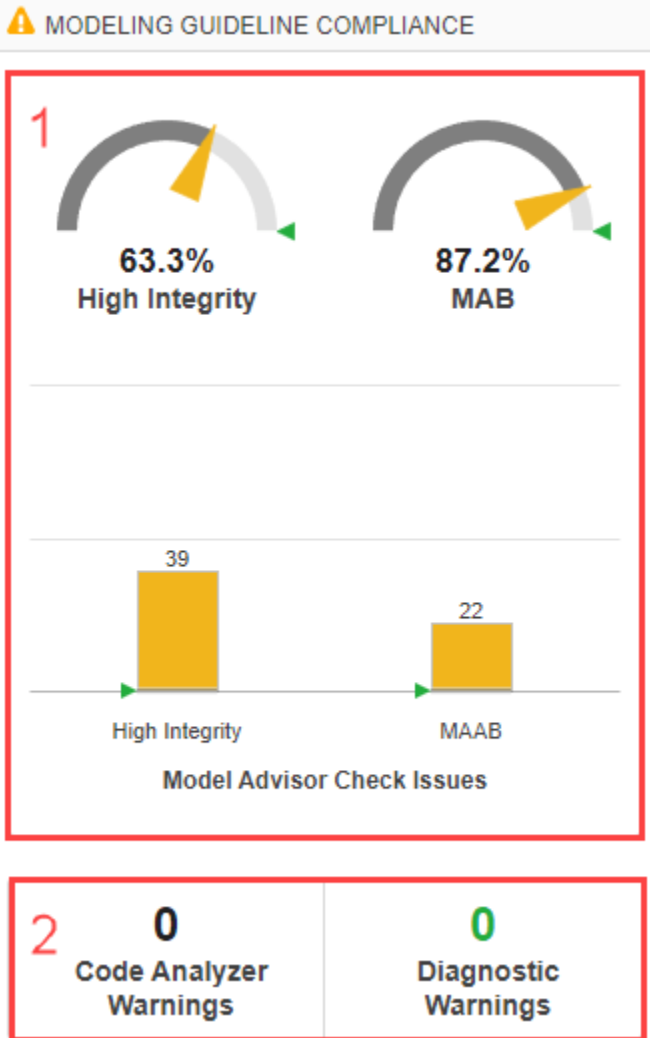
The **Modeling Guideline Compliance** group contains two containers:

- The top container contains the dial widgets and bar charts.
- The bottom container contains the warnings widgets.

In this diagram, the red numbers 1 and 2 indicate the top and bottom containers respectively. There are three widgets in the top container: the **High Integrity** dial widget, **MAB** dial widget, and **Model Advisor Check Issues** bar chart.

To view the widgets in the top and bottom containers, you can open the Metrics Dashboard by entering:

metricsdashboard vdp



5. Use the `getWidgets` function to get the two containers in the **Modeling Guideline Compliance** group.

```
complianceContainers = getWidgets(complianceGroup);
```

6. Get the top container.

```
topContainer = complianceContainers(1);
```

7. Get the widgets in the top container.

```
topContainerWidgets = getWidgets(topContainer);
```

8. Get the **MAB** dial widget and the **Model Advisor Check Issues** bar chart from the top container.

```
dialwidget = topContainerWidgets(2);  
barChartWidget = topContainerWidgets(3);
```

Assign Custom Model Advisor Metrics to Widgets

The metric configuration determines the metric IDs that you can assign to widgets in the Metrics Dashboard. You can update the metric configuration, assign your custom Model Advisor metric IDs to the widgets, and update the widget attributes to work for your custom Model Advisor metrics. Note that metric configuration does not use the Model Advisor configuration file to determine which checks to run. If you have specified a custom Model Advisor configuration, the Metrics Dashboard runs the checks specified by your metric configuration, even if you do not select those checks in your Model Advisor configuration file.

1. Open the default metric configuration and save the configuration object to the base workspace.

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

2. Update the metrics configuration to recognize the **Clone Detection and Model Refactoring** Check Group ID. The **Modeling Guideline Compliance** group can only get metric results for check groups that you specify with the `ModelAdvisorStandard` argument of the function `setMetricFamilyParameterValues`.

```
originalCheckGroupIds = {'maab', 'hisl_do178'};
CheckGroupIDs = [originalCheckGroupIds checkGroupID];
setMetricFamilyParameterValues(metricconfig,ModelAdvisorStandard=CheckGroupIDs);
```

3. Update the dial widget to report the fraction of passing checks in the **Clone Detection and Model Refactoring** check group, instead of the default **Modeling Standards for MAB** check group.

```
setMetricIDs(dialwidget,metricID_for_Fraction_Passed);
dialwidget.Labels = "Clones and Refactoring";
dialwidget.Title = "Clone/Refactor";
```

The dial widget accepts fractional data values between zero and one, but displays the data as a percentage in the dashboard.

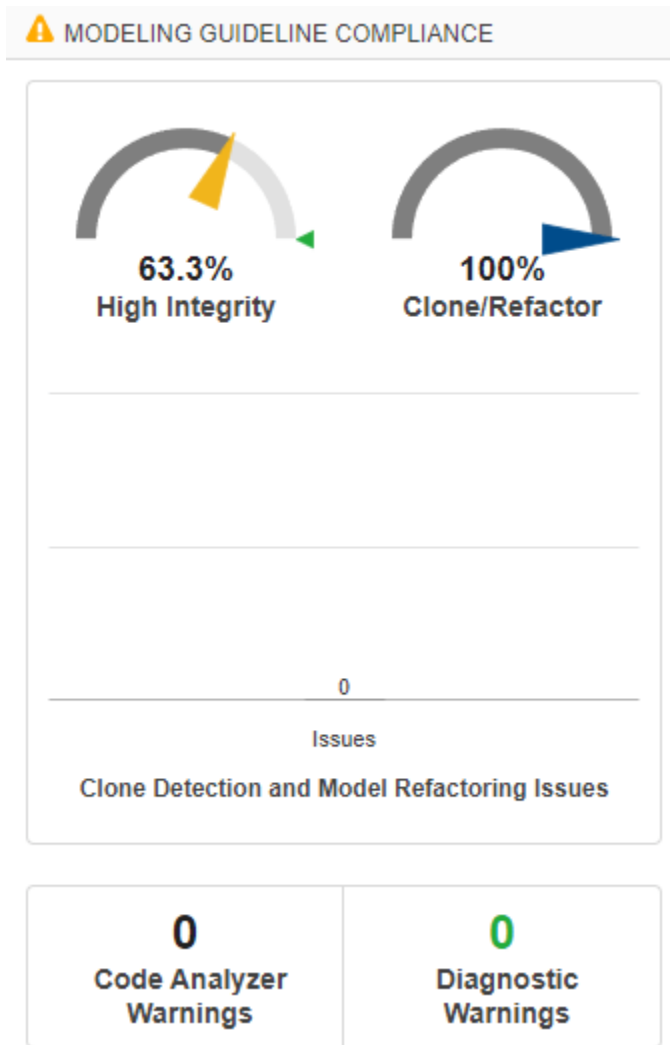
4. Update the bar chart widget to report the number of check issues from the **Clone Detection and Model Refactoring** check group, instead of the default **Modeling Standards for MAB** check group.

```
setMetricIDs(barChartWidget,metricID_for_Number_of_Issues);
barChartWidget.Labels = "Issues";
barChartWidget.Title = "Clone Detection and Model Refactoring Issues";
```

5. Save the updated dashboard and metric configurations, set the configurations as the active configurations, and open the Metrics Dashboard to see the updated dashboard layout.

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

To run the metrics and view the metric results, click **All Metrics** in the toolbar.



For the model vdp, the dial widget shows that 100% of the checks in the **Clone Detection and Model Refactoring** check group pass and the bar chart shows that there are zero check issues in the **Clone Detection and Model Refactoring** check group. The dial widget appears blue because there are no metric thresholds associated with the custom Model Advisor metric. For information on how to add or change metric thresholds, see “Modify, Remove, and Add Metric Thresholds in Metrics Dashboard” on page 5-62.

For more information on metrics dashboard customization, see “Customize Metrics Dashboard Layout and Functionality” on page 5-37.

Compare Model Complexity and Code Complexity Metrics

Note The Metrics Dashboard and `slmetric`.Engine API will be removed in a future release. To collect and compare complexity metrics for the Simulink models, Stateflow charts, and MATLAB code in a project, use the Model Maintainability Dashboard instead. For more information, see “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard” on page 5-144 and “Overall Design Cyclomatic Complexity” on page 5-294.

Analyze the complexity of your system by using the cyclomatic complexity metrics. The metrics indicate the structural complexity of a system by measuring the number of linearly independent paths in the system. By limiting the cyclomatic complexity of your system, you can make it more readable, maintainable, and portable. You can measure the cyclomatic complexity for both your model and the code generated from your model. Note that differences between the code and the model may result in different levels of cyclomatic complexity. To measure the cyclomatic complexity of a model, use the Metrics Dashboard and the “Cyclomatic Complexity Metric” on page 5-373.

Metric Threshold Values

Code Complexity Threshold

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

Model Complexity Threshold

When you use the model-based design workflow to model an algorithm and generate code, you can assess the readability of the system by using the cyclomatic complexity metric of the model instead of measuring the cyclomatic complexity of the generated code. The graphical modeling of Simulink allows you to manage complex algorithms better than traditional hand code does. To account for this, the default cyclomatic complexity metric threshold for the model is 30, which is higher than the standard code complexity threshold of 10. To change the model metric threshold value, see “Customize Metrics Dashboard Layout and Functionality” on page 5-37.

Comparing Code and Model Complexity Metric Results

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

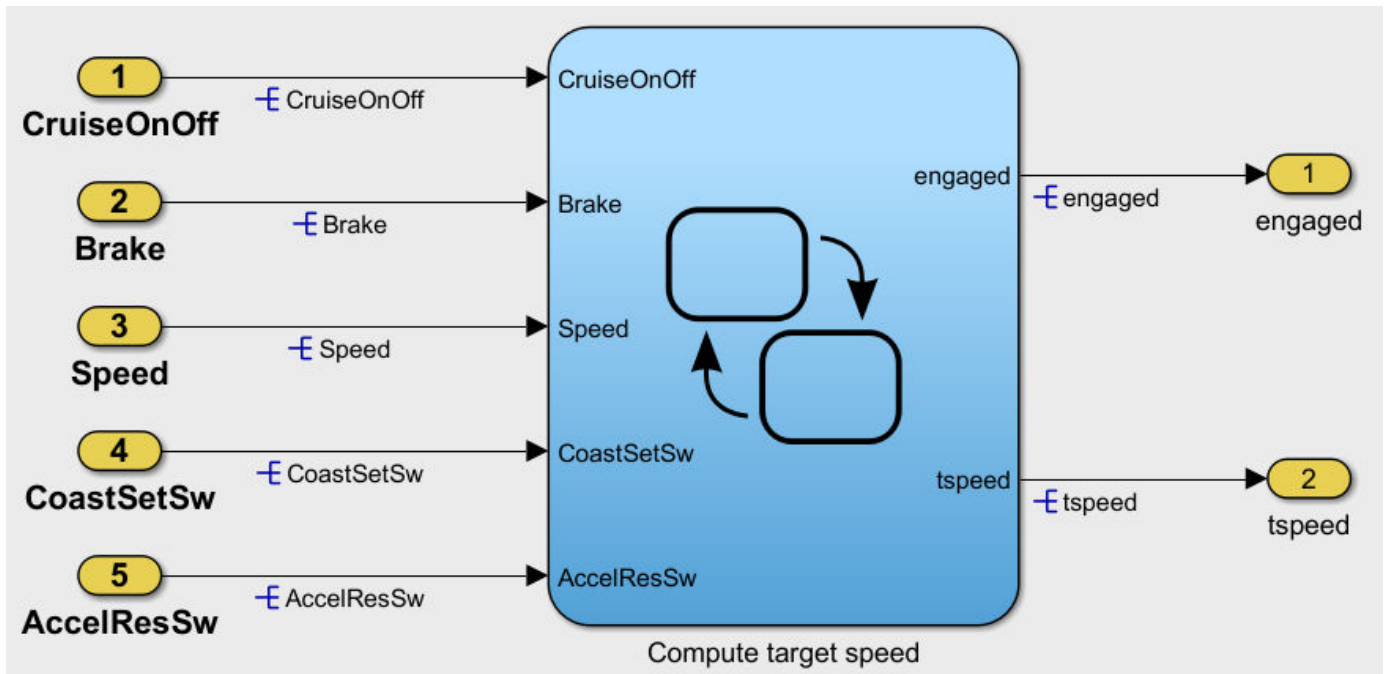
- Code generation optimizations that eliminate extra logic that the model contains. These optimizations can reduce the complexity of the code.
- Error checks in the generated code that the model metric analysis does not consider. These error checks can increase the complexity of the code.
- Additional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

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

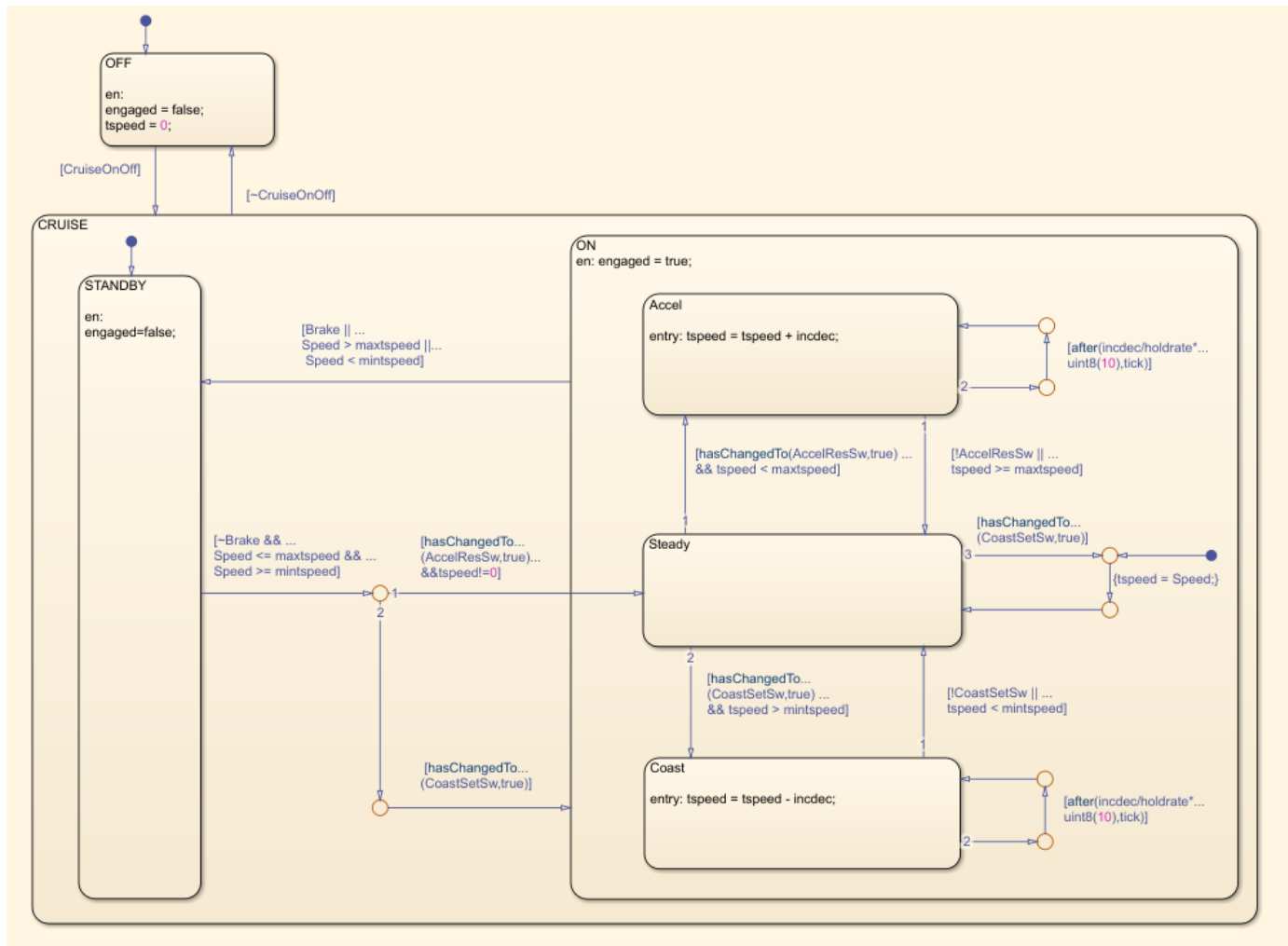
- 1 Open the project.

```
openExample("shared_vnv/CruiseControlVerificationProjectExample");
pr = openProject("SimulinkVerificationCruise");
```

- 2 From the project, open the `models` folder and open `simulinkCruiseErrorAndStandardsExample`.



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



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

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

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

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

See Also

“Cyclomatic Complexity Metric” on page 5-373

More About

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

Explore Status and Quality of Testing Activities Using Model Testing Dashboard

The Model Testing Dashboard collects metric data from the model design and testing artifacts in a project to help you assess the status and quality of your requirements-based model testing.

The dashboard analyzes the artifacts in a project, such as requirements, models, and test results. Each metric in the dashboard measures a different aspect of the quality of the testing of your model and reflects guidelines in industry-recognized software development standards, such as ISO 26262 and DO-178C.

This example shows how to assess the testing status of a unit by using the Model Testing Dashboard. If the requirements, models, or tests in your project change, use the dashboard to assess the impact on testing and update the artifacts to achieve your testing goals.

Explore Testing Artifacts and Metrics for a Project

Open the project that contains the models and testing artifacts. For this example, in the MATLAB® Command Window, enter:

```
dashboardCCProjectStart("incomplete")
```

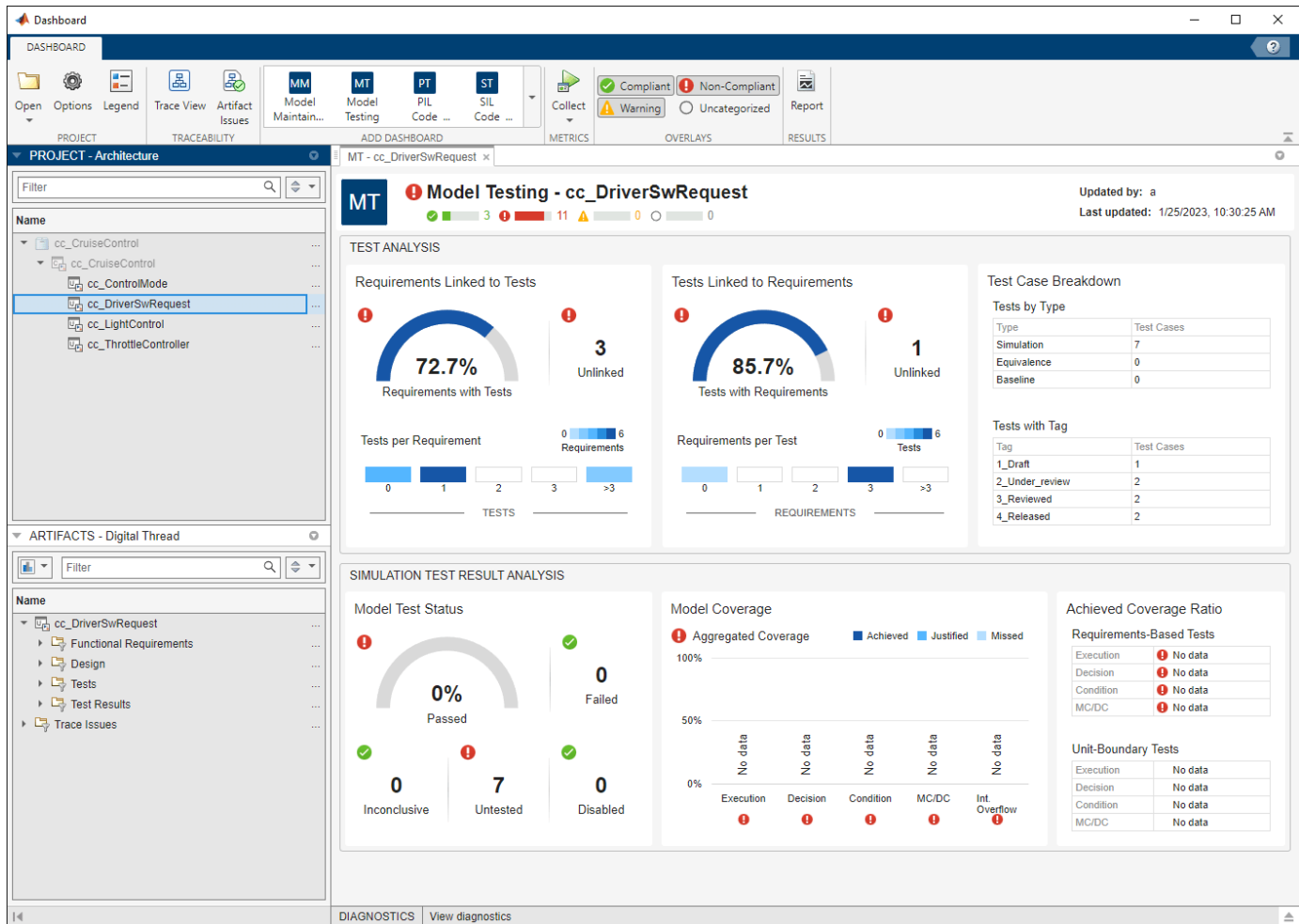
Open the Model Testing Dashboard by using one of these approaches:

- On the **Project** tab, click **Model Testing Dashboard**.
- In the Command Window, enter:


```
modelTestingDashboard
```

The first time that you open the dashboard for a project, the dashboard must identify the artifacts in the project and collect traceability information. The **Project** panel organizes the units in the project under the components according to the model reference hierarchy.

The dashboard displays metric results for the unit you select in the **Project** panel. Click the unit **cc_DriverSwRequest** to view its metric results. When you initially select a unit in the **Project** panel, the dashboard automatically collects the metric results for the unit. If you want to collect metrics for each of the units in the project, click **Collect > Collect All**. If you previously collected metric data for a unit, the dashboard populates with the existing data. Collecting data for a metric requires a license for the product that supports the underlying artifacts, such as Requirements Toolbox™, Simulink® Test™, or Simulink Coverage™. After you collect metric results, you only need a Simulink® Check™ license to view the results. For more information, see “Model Testing Metrics”.



View Traceability of Design and Testing Artifacts

The **Artifacts** panel shows artifacts that trace to the unit. Expand the folders and subfolders to see the artifacts in the project that trace to the unit selected in the **Project** panel. To view more information for a folder or subfolder, click the three dots and click the Help icon , if available.


For each unit in the project, the traced artifacts include:

- **Functional Requirements** — Requirements of **Type** Functional that are either implemented by or upstream of the unit. Use the Requirements Toolbox to create or import the requirements in a requirements file (.slreqx).
- **Design** artifacts — The model file that contains the unit that you test and the libraries, data dictionaries, and other design artifacts that the model uses.
- **Tests** — Tests and test harnesses that trace to the unit. Create the tests by using Simulink Test. A test can be either a test iteration in a test case or a test case without iterations.
- **Test Results** — Results of the tests for the unit. The dashboard shows the latest results from the tests.

An artifact appears under the folder **Trace Issues** if there are unexpected requirement links, requirement links which are broken or not supported by the dashboard, or artifacts that the

dashboard cannot trace to a unit. The folder includes artifacts that are missing traceability and artifacts that the dashboard is unable to trace. If an artifact generates an error during traceability analysis, it appears under the **Errors** folder. For more information about artifact tracing issues and errors, see “Trace Artifacts to Units and Components” on page 5-96.

Navigate to the requirement artifact for **Cancel Switch Detection**. Expand **cc_DriverSwRequest > Functional Requirements > Implemented > cc_SoftwareReqs.slreqx** and select the requirement **Cancel Switch Detection**. To view the path from the project root to the artifact, click the three dots


to the right of the artifact name. You can use the menu button  to the right of the search bar to collapse or expand the artifact list, or to restore the default view of the artifacts list.


For more information on how the dashboard traces the artifacts shown in the **Artifacts** panel, see “Manage Project Artifacts for Analysis in Dashboard” on page 5-95.

View Metric Results for a Unit


You can collect and view metric results for each unit that appears in the **Project** panel. To view the results for the unit **cc_DriverSwRequest**, in the **Project** panel, click **cc_DriverSwRequest**. When you click on a unit, the dashboard shows the **Model Testing** information for that unit. The top of the dashboard tab shows the name of the unit, the data collection timestamp, and the user name that collected the data.

If you collect results, but then make a change to an artifact in the project, the dashboard detects the change and shows a warning banner at the top of the dashboard to indicate that the metric results are stale.

 Metric results are stale.

 Collect



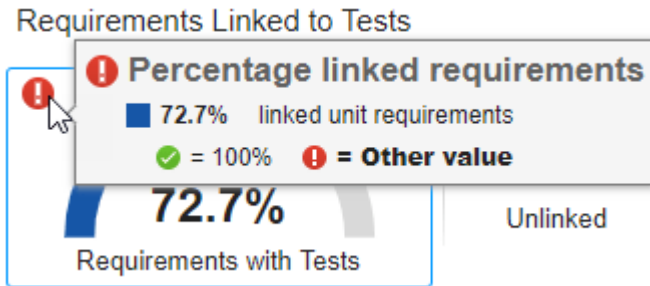
The Stale icon  appears on dashboard widgets that might show stale data which does not include the changes. If you see the warning banner, click the **Collect** button on the warning banner to re-collect the metric data and to update the stale widgets with data from the current artifacts. You can also find the **Collect** button on the dashboard toolstrip in the **Metrics** section. For the unit in this example, the metric results in the dashboard are not stale.

The dashboard widgets summarize the metric data results and show testing issues you can address, such as:

- Missing traceability between requirements and tests
- Tests or requirements with a disproportionate number of links between requirements and tests
- Failed or disabled tests
- Missing model coverage



You can use the overlays in the Model Testing Dashboard to see if the metric results for a widget are compliant, non-compliant, or generate a warning that the metric results should be reviewed. Results are compliant if they show full traceability, test completion, or model coverage. In the **Overlays** section of the toolstrip, check that the **Compliant** and **Non-Compliant** buttons are selected. The overlay appears on the widgets that have results in that category. You can see the total number of widgets in each compliance category in the top-right corner of the dashboard.

To see the compliance thresholds for a metric, point to the overlay icon.



You can hide the overlay icons by clicking a selected category in the **Overlays** section of the toolstrip. For more information on the compliance thresholds for each metric, see “Model Testing Metrics”.

To explore the data in more detail, click an individual metric widget to open the **Metric Details**. For the selected metric, a table displays a metric value for each artifact. The table provides hyperlinks to open the artifacts so that you can get detailed results and fix the artifacts that have issues. When exploring the tables, note that:

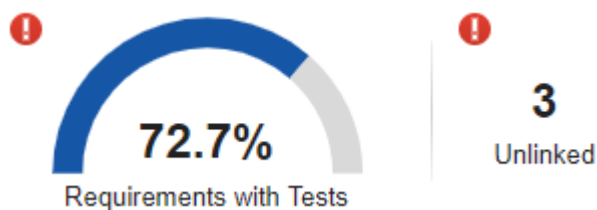
- You can filter the results by the value returned for each artifact. To filter the results, click the filter icon  in the table header.
- By default, some widgets apply a filter to the table. For example, for the **Requirements Linked to Tests** section, the table for the **Unlinked** widget is filtered to only show requirements that are missing linked tests. Tables that have filters show a check mark in the bottom right corner of the filter icon .
- To sort the results by artifact, source file, or value, click the corresponding column header.

Evaluate Testing and Traceability of Requirements

A standard measure of testing quality is the traceability between individual requirements and the tests that verify them. To assess the traceability of your tests and requirements, use the metric data in the **Test Analysis** section of the dashboard. You can quickly find issues in the requirements and tests by using the data summarized in the widgets. Click a widget to view a table with detailed results and links to open the artifacts.

Requirements Missing Tests

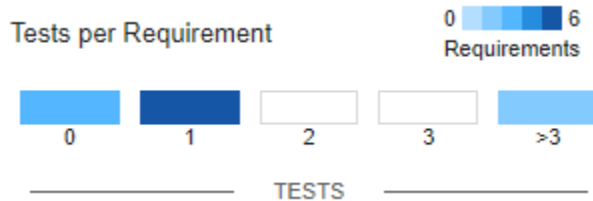
In the **Requirements Linked to Tests** section, the **Unlinked** widget indicates how many requirements are missing links to tests. To address unlinked requirements, create tests that verify each requirement and link those tests to the requirement. The **Requirements with Tests** gauge widget shows the linking progress as the percentage of requirements that have tests.




Click any widget in the section to see the detailed results in the **Requirement linked to tests** table. For each requirement artifact, the table shows the source file that contains the requirement and whether the requirement is linked to at least one test. When you click the **Unlinked** widget, the table is filtered to show only requirements that are missing links to tests.

Requirements with Disproportionate Numbers of Tests

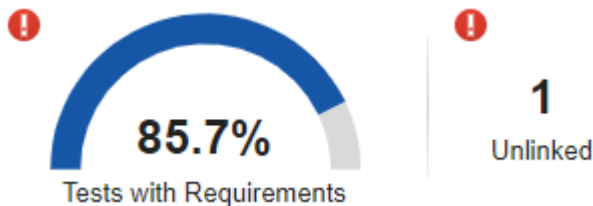
The **Tests per Requirement** section summarizes the distribution of the number tests linked to each requirement. For each value, a colored bin indicates the number of requirements that are linked to that number of tests. Darker colors indicate more requirements. If a requirement has a too many tests, the requirement might be too broad, and you may want to break it down into multiple more granular requirements and link each of those requirements to the respective tests. If a requirement has too few tests, consider adding more tests and linking them to the requirement.



To see the requirements that have a certain number of tests, click the corresponding number to open a filtered **Tests per requirement** table. For each requirement artifact, the table shows the source file that contains the requirement and the number of linked tests. To see the results for each of the requirements, in the **Linked Tests** column, click the filter icon , then select **Clear Filters**.

Tests Missing Requirements

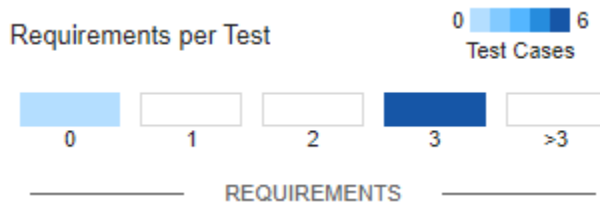
In the **Tests Linked to Requirements** section, the **Unlinked** widget indicates how many tests are not linked to requirements. To address unlinked tests, add links from these tests to the requirements they verify. The **Tests with Requirements** gauge widget shows the linking progress as the percentage of tests that link to requirements.




Click any widget in the section to see detailed results in the **Tests linked to requirements** table. For each test artifact, the table shows the source file that contains the test and whether the test is linked to at least one requirement. When you click the **Unlinked** widget, the table is filtered to show only tests that are missing links to requirements.

Tests with Disproportionate Numbers of Requirements

The **Requirements per Test** widget summarizes the distribution of the number of requirements linked to each test. For each value, a colored bin indicates the number of requirements that are linked to that number of tests. Darker colors indicate more tests. If a test has too many or too few requirements, it might be more difficult to investigate failures for that test, and you may want to change the test or requirements so that they are easier to track. For example, if a test verifies many more requirements than the other tests, consider breaking it down into multiple smaller tests and linking them to the requirements.



To see the tests that have a certain number of requirements, click the corresponding bin to open the **Requirements per test** table. For each test artifact, the table shows the source file that contains the test and the number of linked requirements. To see the results for each of the tests, in the **Linked Requirements** column, click the filter icon , then select **Clear Filters**.

Disproportionate Number of Tests of One Type

The **Tests by Type** and **Tests with Tags** widgets show how many tests the unit has of each type and with each custom tag. In industry standards, tests are often categorized as normal tests or robustness tests. You can tag test cases with **Normal** or **Robustness** and see the total count for each tag by using the **Tests with Tag** widget. Use the **Test Case Breakdown** to decide if you want to add tests of a certain type, or with a certain tag, to your project.


Test Case Breakdown


Tests by Type

Type	Test Cases
Simulation	7
Equivalence	0
Baseline	0

Tests with Tag

Tag	Test Cases
1_Draft	1
2_Under_review	2
3_Reviewed	2
4_Released	2

To see the test cases of one type, click the corresponding row in the **Tests by Type** table to open the **Test case type** table. For each test case artifact, the table shows the source file that contains the test and the test type. To see results for each of the test cases, in the **Type** column, click the filter icon , then select **Clear Filters**.

To see the test cases that have a tag, click the corresponding row in the **Tests with Tag** table to open the **Test case tags** table. For each test case artifact, the table shows the source file that contains the test and the tags on the test case. To see results for each of the test cases, in the **Tags** column, click the filter icon , then select **Clear Filters**.

Analyze Test Results and Coverage

To see a summary of the test results and coverage measurements, use the widgets in the **Simulation Test Result Analysis** section of the dashboard. Find issues in the tests and in the model by using the test result metrics. Find coverage gaps by using the coverage metrics and add tests to address missing coverage.

Run the tests for the model and collect the dashboard metrics to check for model testing issues.

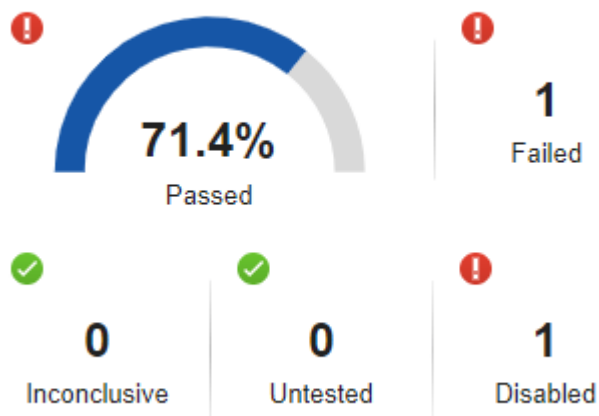
Tests That Have Not Passed

In the **Model Test Status** section, the **Untested** and **Disabled** widgets indicate how many tests for the unit have not been run. Run the tests by using the Test Manager and export the new results.

The **Failed** widget indicates how many tests failed. Click on the **Failed** widget to view a table of the tests that failed. Click the hyperlink for each failed test artifact to open it in the Test Manager and investigate the artifacts that caused the failure. Fix the artifacts, re-run the tests, and export the results.

The **Inconclusive** widget indicates how many tests do not have pass/fail criteria such as verify statements, custom criteria, baseline criteria, and logical or temporal assessments. If a test does not contain pass/fail criteria, then it does not verify the functionality of the linked requirement. Add one or more of these pass/fail criteria to your tests to help verify the functionality of your model.

Model Test Status



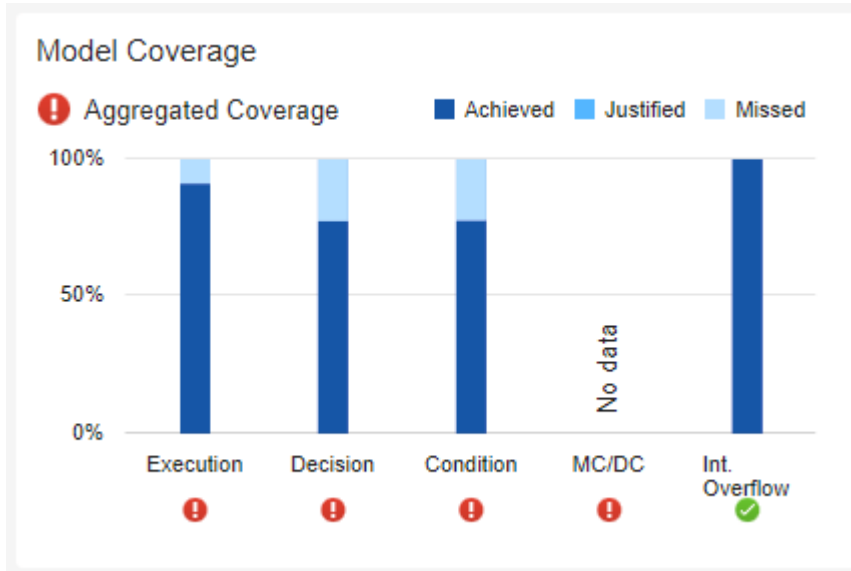
Click the **Passed**, **Failed**, **Disabled**, or **Untested** widgets to open the **Test run status** table. For each test artifact, the table shows the source file that contains the test and the status of the test result. When you click the **Failed**, **Disabled**, or **Untested** widgets, the table is filtered to show only tests for those test result statuses. The dashboard analyzes only the latest test result that it traces to each test.

Click the **Inconclusive** widget to open the **Tests with pass/fail criteria table**. For each test artifact that is missing pass/fail criteria, the table shows the source file that contains the test.

Missing Coverage

The **Model Coverage** subsection shows whether there are model elements that are not covered by the tests. If one of the coverage types shows less than 100% coverage, you can click the dashboard

widgits to investigate the coverage gaps. Add tests to cover the gaps or justify points that do not need to be covered. Then run the tests again and export the results. For more information on coverage justification, see “Fix Requirements-Based Testing Issues” on page 5-89.



To see the detailed results for one type of coverage, click the corresponding bar. For the model and test artifacts, the table shows the source file and the achieved and justified coverage.




Sources of Overall Achieved Coverage

The **Achieved Coverage Ratio** subsection shows the sources of the overall achieved coverage. Industry-recognized software development standards recommend using requirements-based, unit-boundary tests to confirm the completeness of coverage. For more information, see “Monitor Low-Level Test Results in the Model Testing Dashboard” on page 5-123.




- *Requirements-based tests* are tests that link to at least one requirement in the design. When you link a test to a requirement, you provide traceability between the test and the requirement that the test verifies. If you do not link a test to a requirement, it is not clear which aspect of the design the test verifies.
- *Unit-boundary tests* are tests that test the whole unit. When you execute a test on the unit-boundary, the test has access to the entire context of the unit design. If you only test the lower-level, sub-elements in a design, the test might not capture the actual functionality of those sub-elements within the context of the whole unit. Types of *sub-elements* include subsystems, subsystem references, library subsystems, and model references.

Achieved Coverage Ratio

Requirements-Based Tests

Execution	✓ 
Decision	✓ 
Condition	✓ 
MC/DC	! No data available

Unit-Boundary Tests

Execution	
Decision	
Condition	
MC/DC	No data available

The section **Requirements-Based Tests** identifies the percentage of the overall achieved coverage coming from requirements-based tests. If one of the coverage types shows that less than 100% of the overall achieved coverage comes from requirements-based tests, add links from the associated tests to the requirements they verify.

The section **Unit-Boundary Tests** identifies the percentage of the overall achieved coverage coming from unit-boundary tests. If one of the coverage types shows that less than 100% of the overall achieved coverage comes from unit-boundary tests, consider either adding a test that tests the whole unit or reconsidering the unit-model definition.

See Also

“Model Testing Metrics”

More About

- “Assess Requirements-Based Testing for ISO 26262” on page 5-102
- “Fix Requirements-Based Testing Issues” on page 5-89
- “Resolve Missing Artifacts, Links, and Results” on page 5-128

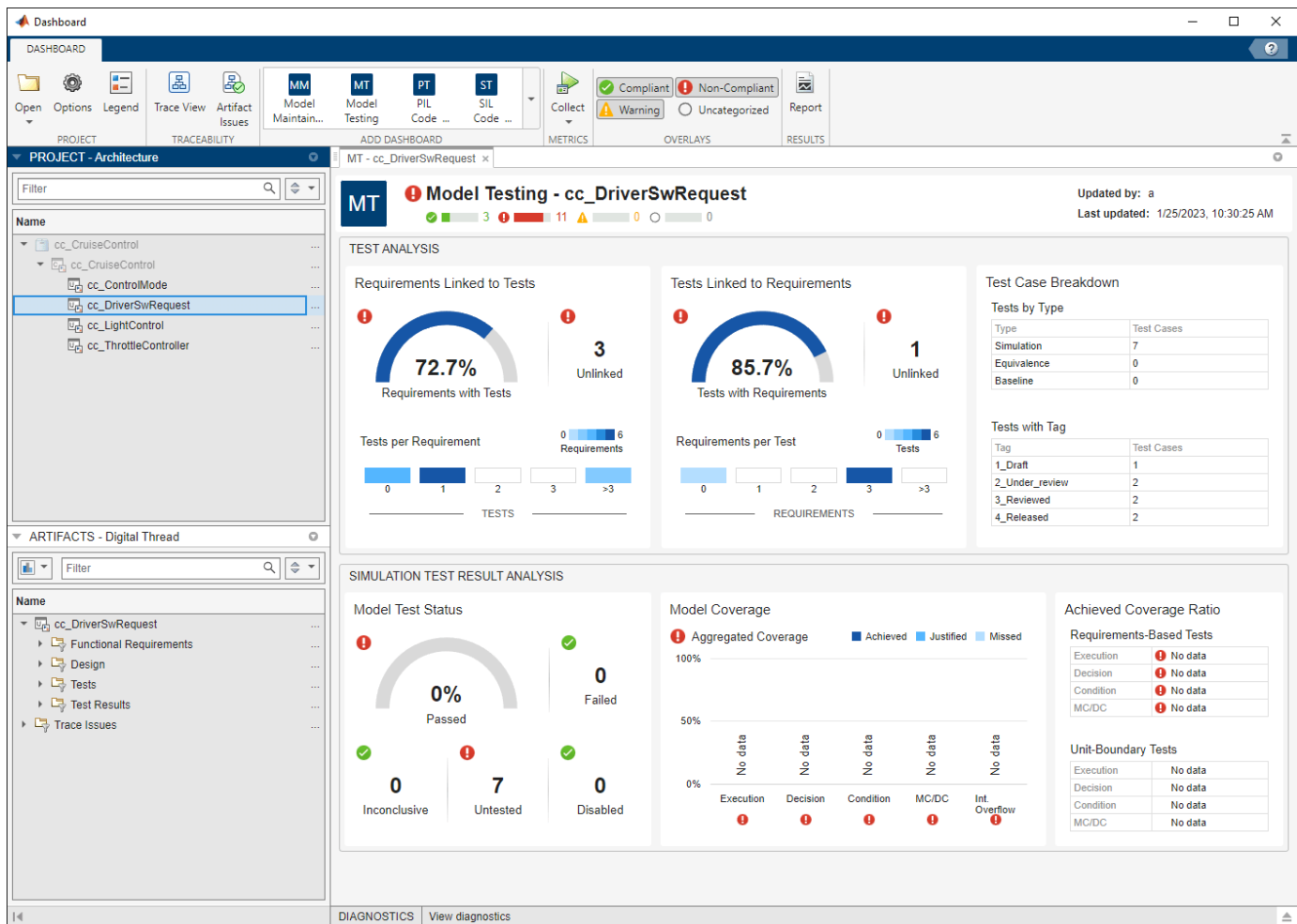
Fix Requirements-Based Testing Issues

This example shows how to address common traceability issues in model requirements and tests by using the Model Testing Dashboard. The dashboard analyzes the testing artifacts in a project and reports metric data on quality and completeness measurements such as traceability and coverage, which reflect guidelines in industry-recognized software development standards, such as ISO 26262 and DO-178C. The dashboard widgets summarize the data so that you can track your requirements-based testing progress and fix the gaps that the dashboard highlights. You can click the widgets to open tables with detailed information, where you can find and fix the testing artifacts that do not meet the corresponding standards.

Collect Metrics for the Testing Artifacts in a Project

The dashboard displays testing data for a model and the artifacts that the unit traces to within a project. For this example, open the project and collect metric data for the artifacts.

- 1 Open the project that contains the models and testing artifacts. For this example, in the MATLAB® Command Window, enter `dashboardCCProjectStart("incomplete")`.
- 2 Open the Dashboard window. To open the Model Testing Dashboard: on the **Project** tab, click **Model Testing Dashboard** or enter `modelTestingDashboard` at the command line.
- 3 In the **Project** panel, the dashboard organizes unit models under the component models that contain them in the model hierarchy. View the metric results for the unit `cc_DriverSwRequest`. In the **Project** panel, click the name of the unit, **cc_DriverSwRequest**. When you initially select **cc_DriverSwRequest**, the dashboard collects the metric results for uncollected metrics and populates the widgets with the data for the unit.



Link a Requirement to its Implementation in a Model

The **Artifacts** panel shows artifacts such as requirements, tests, and test results that trace to the unit selected in the **Project** panel.

In the **Artifacts** panel, the **Trace Issues** folder shows artifacts that do not trace to unit models in the project. The **Trace Issues** folder contains subfolders for:

- **Unexpected Implementation Links** — Requirement links of **Type Implements** for a requirement of **Type Container** or **Type Informational**. The dashboard does not expect these links to be of **Type Implements** because container requirements and informational requirements do not contribute to the Implementation and Verification status of the requirement set that they are in. If a requirement is not meant to be implemented, you can change the link type. For example, you can change a requirement of **Type Informational** to have a link of **Type Related to**.
- **Unresolved and Unsupported Links** — Requirement links which are broken or not supported by the dashboard. For example, if a model block implements a requirement, but you delete the model block, the requirement link is now unresolved. The Model Testing Dashboard does not support traceability analysis for some artifacts and some links. If you expect a link to trace to a unit and it does not, see the troubleshooting solutions in “Resolve Missing Artifacts, Links, and Results” on page 5-128.

- **Untraced Tests** — Tests that execute on models or subsystems that are not on the project path.
- **Untraced Results** — Results that the dashboard can no longer trace to a test. For example, if a test produces results, but you delete the test, the results can no longer be traced to the test.


Address Testing Traceability Issues

The widgets in the **Test Analysis** section of the Model Testing Dashboard show data about the unit requirements, tests for the unit, and links between them. The widgets indicate if there are gaps in testing and traceability for the implemented requirements.

Link Requirements and Tests

For the unit `cc_DriverSwRequest`, the **Tests Linked to Requirements** section shows that some of the tests are missing links to requirements in the model.

To see detailed information about the missing links, in the **Tests Linked to Requirements** section, click the widget **Unlinked**. The dashboard opens the **Metric Details** for the widget with a table of metric values and hyperlinks to each related artifact. The table shows the tests that are implemented in the unit, but do not have links to requirements. The table is filtered to show only tests that are missing links to requirements.



Metric Details - Tests linked to requirements

Metric that determines if each test case or test iteration for the model is linked to at least one requirement in the project.

Artifact	Source	Requirement Link Status
Detect long decrement	cc_DriverSwRequest_Tests.mldatx	Missing linked requirements

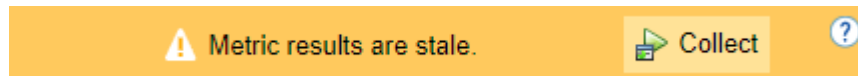
The test `Detect long decrement` is missing linked requirements.

- 1 In the **Artifact** column of the table, point to **Detect long decrement**. The tooltip shows that the test **Detect long decrement** is in the test suite **Unit test for DriverSwRequest**, in the test file **cc_DriverSwRequest_Tests**.
- 2 Click **Detect long decrement** to open the test in the Test Manager. For this example, the test needs to link to three requirements that already exist in the project. If there were not already requirements, you could add a requirement by using the Requirements Editor.
- 3 Open the software requirements in the Requirements Editor. In the **Artifacts** panel of the Dashboard window, expand the folder **Functional Requirements > Implemented** and double-click the requirement file **cc_SoftwareReqs.slreqx**.
- 4 View the software requirements in the container with the summary **Driver Switch Request Handling**. Expand **cc_SoftwareReqs > Driver Switch Request Handling**.
- 5 Select multiple software requirements. Hold down the **Ctrl** key as you click **Output request mode**, **Avoid repeating commands**, and **Long Increment/Decrement Switch recognition**. Keep these requirements selected in the Requirements Editor.
- 6 In the Test Manager, expand the **Requirements** section for the test `Detect long decrement`. Click the arrow next to the **Add** button and select **Link to Selected Requirement**. The traceability link indicates that the test `Detect long decrement` verifies the three requirements **Output request mode**, **Avoid repeating commands**, and **Long Increment/Decrement Switch recognition**.

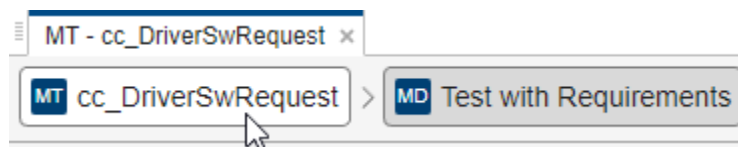
- 7 The metric results in the dashboard reflect only the saved artifact files. To save the test suite `cc_DriverSwRequest_Tests.mldatx`, in the **Test Browser**, right-click `cc_DriverSwRequest_Tests` and click **Save**.

Refresh Metric Results in the Dashboard

The dashboard detects that the metric results are now stale and shows a warning banner at the top of the dashboard.



- 1 Click the **Collect** button on the warning banner to re-collect the metric data so that the dashboard reflects the traceability link between the test and requirements.
- 2 View the updated dashboard widgets by returning to the **Model Testing** results. At the top of the dashboard, there is a breadcrumb trail from the **Metric Details** back to the **Model Testing** results. Click the breadcrumb button for `cc_DriverSwRequest` to return to the **Model Testing** results for the unit.



The **Tests Linked to Requirements** section shows that there are no unlinked tests. The **Requirements Linked to Tests** section shows that there are 3 unlinked requirements. Typically, before running the tests, you investigate and address these testing traceability issues by adding tests and linking them to the requirements. For this example, leave the unlinked artifacts and continue to the next step of running the tests.

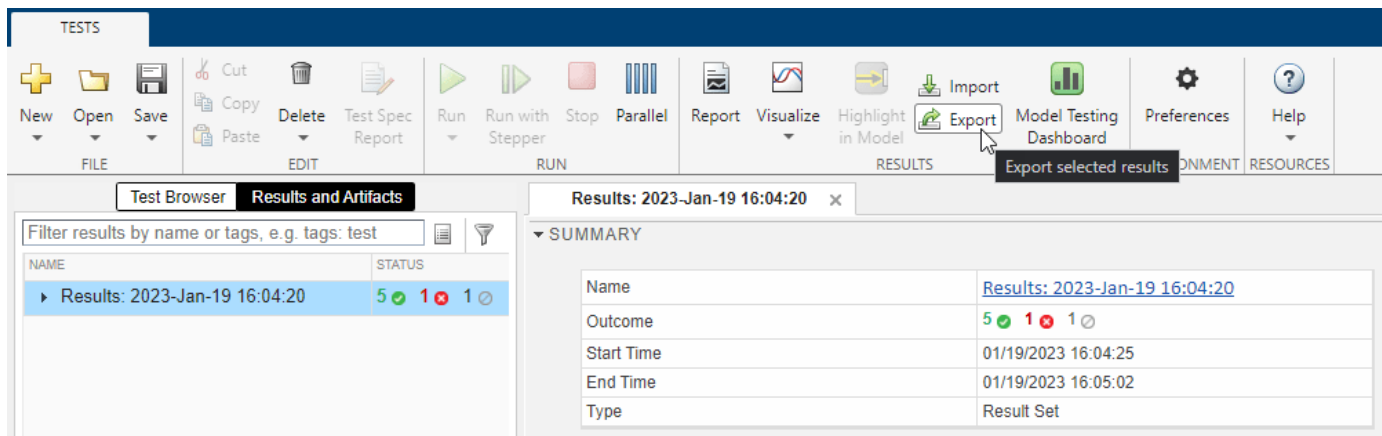
Test the Model and Analyze Failures and Gaps

After you create and link unit tests that verify the requirements, run the tests to check that the functionality of the model meets the requirements. To see a summary of the test results and coverage measurements, use the widgets in the **Simulation Test Result Analysis** section of the dashboard. The widgets help show testing failures and gaps. Use the metric results to analyze the underlying artifacts and to address the issues.

Perform Unit Testing

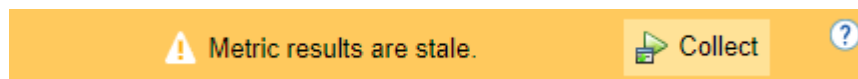
Run the tests for the model by using the Test Manager. Save the test results in your project and review them in the Model Testing Dashboard.

- 1 Open the unit tests for the model in the Test Manager. In the Model Testing Dashboard, in the **Artifacts** panel, expand the folder **Tests > Unit Tests** and double-click the test file `cc_DriverSwRequest_Tests.mldatx`.
- 2 In the Test Manager, click **Run**.
- 3 Select the results in the **Results and Artifacts** pane.
- 4 Save the test results as a file in the project. On the **Tests** tab, in the **Results** section, click **Export**. Name the results file `Results1.mldatx` and save the file under the project root folder.



The Model Testing Dashboard detects the results and automatically updates the **Artifacts** panel to include the new test results for the unit in the subfolder **Test Results > Model**.

The dashboard also detects that the metric results are now stale and shows a warning banner at the top of the dashboard.



The **Stale** icon **STALE** appears on the widgets in the **Simulation Test Result Analysis** section to indicate that they are showing stale data that does not include the changes.

Click the **Collect** button on the warning banner to re-collect the metric data and to update the stale widgets with data from the current artifacts.

Address Testing Failures and Gaps

For the unit `cc_DriverSwRequest`, the **Model Test Status** section of the dashboard indicates that one test failed and one test was disabled during the latest test run.

- 1 To view the disabled test, in the dashboard, click the **Disabled** widget. The table shows the disabled tests for the model.
- 2 Open the disabled test in the Test Manager. In the table, click the test artifact **Detect long decrement**.
- 3 Enable the test. In the **Test Browser**, right-click the test and click **Enabled**.
- 4 Re-run the test. In the **Test Browser**, right-click the test and click **Run** and save the test suite file.
- 5 View the updated number of disabled tests. In the dashboard, click the **Collect** button on the warning banner. Note that there are now zero disabled tests reported in the **Model Test Status** section of the dashboard.
- 6 View the failed test in the dashboard. Click the breadcrumb button for `cc_DriverSwRequest` to return to the **Model Testing** results and click the **Failed** widget.
- 7 Open the failed test in the Test Manager. In the table, click the test artifact **Detect set**.
- 8 Examine the test failure in the Test Manager. You can determine if you need to update the test or the model by using the test results and links to the model. For this example, instead of fixing the

failure, use the breadcrumbs in the dashboard to return to the **Model Testing** results and continue on to examine test coverage.

Check if the tests that you ran fully exercised the model design by using the coverage metrics. For this example, the **Model Coverage** section of the dashboard indicates that some conditions in the model were not covered. Place your cursor over the **Decision** bar in the widget to see what percent of condition coverage was achieved.

- 1 View details about the decision coverage by clicking one of the **Decision** bars. For this example, click the **Decision** bar for **Achieved** coverage.
- 2 In the table, expand the model artifact. The table shows the test results for the model and the results files that contains them. For this example, click on the hyperlink to the source file **Results1.mldatx** to open the results file in the Test Manager.
- 3 To see detailed coverage results, use the Test Manager to open the model in the Coverage perspective. In the Test Manager, in the **Aggregated Coverage Results** section, in the **Analyzed Model** column, click **cc_DriverSwRequest**.
- 4 Coverage highlighting on the model shows the points that were not covered by the tests. For this example, do not fix the missing coverage. For a point that is not covered in your project, you can add a test to cover it. You can find the requirement that is implemented by the model element or, if there is none, add a requirement for it. Then you can link the new test to the requirement. If the point should not be covered, you can justify the missing coverage by using a filter.

Once you have updated the unit tests to address failures and gaps in your project, run the tests and save the results. Then examine the results by collecting the metrics in the dashboard.

Iterative Requirements-Based Testing with the Model Testing Dashboard

In a project with many artifacts and traceability connections, you can monitor the status of the design and testing artifacts whenever there is a change to a file in the project. After you change an artifact, use the dashboard to check if there are downstream testing impacts by updating the tracing data and metric results. Use the **Metric Details** tables to find and fix the affected artifacts. Track your progress by updating the dashboard widgets until they show that the model testing quality meets the standards for the project.

See Also

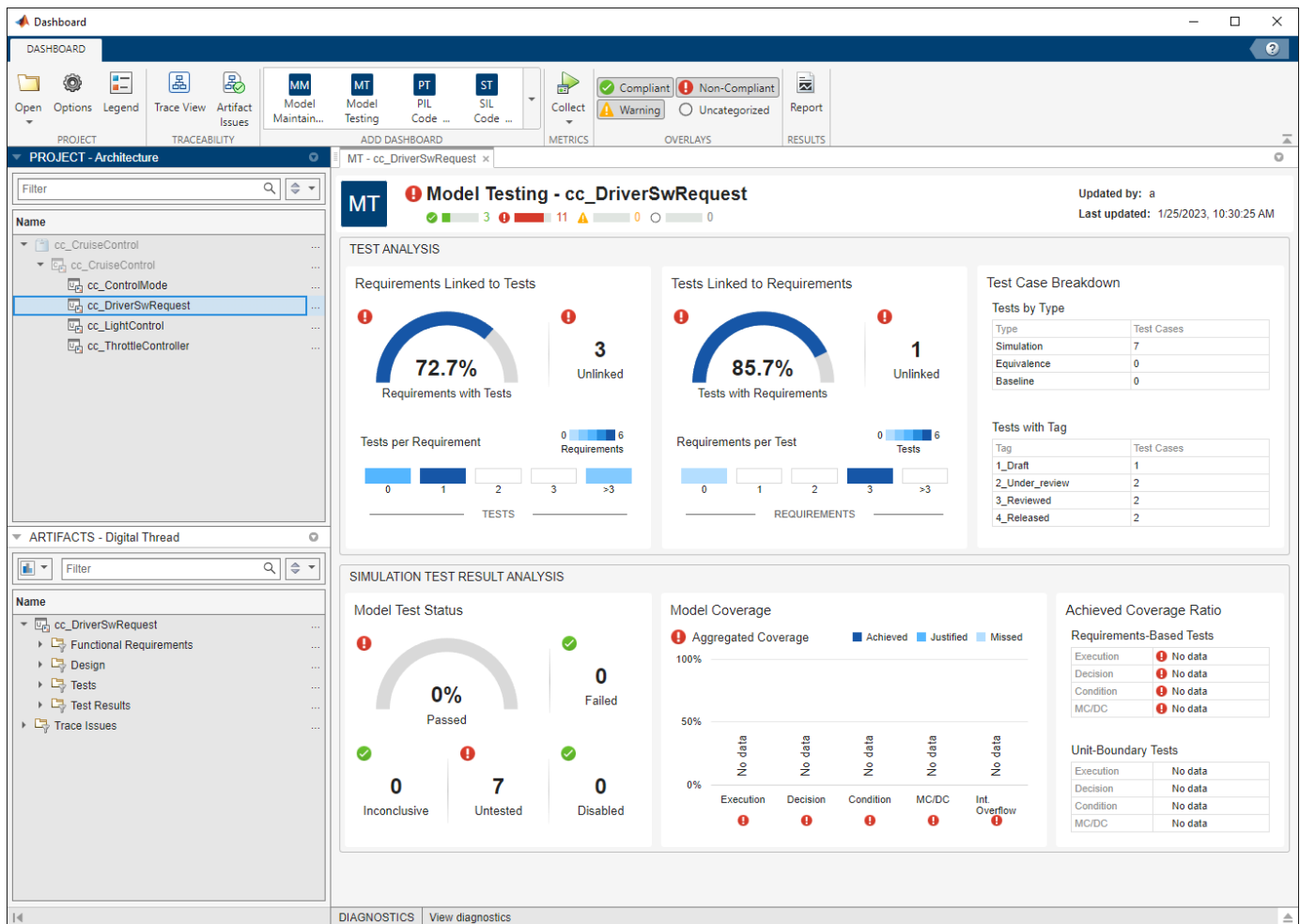
“Model Testing Metrics”

More About

- “Assess Requirements-Based Testing for ISO 26262” on page 5-102
- “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80
- “Resolve Missing Artifacts, Links, and Results” on page 5-128

Manage Project Artifacts for Analysis in Dashboard

When you develop and test software units using Model-Based Design, use the Model Testing Dashboard to assess the status and quality of your unit testing activities. Requirements-based testing is a central element of model verification. By establishing traceability links between your requirements, model design elements, and tests, you can measure the extent to which the requirements are implemented and verified. The Model Testing Dashboard analyzes this traceability information and provides detailed metric measurements on the traceability, status, and results of these testing artifacts.



Each metric in the dashboard measures a different aspect of the quality of your unit testing and reflects guidelines in industry-recognized software development standards, such as ISO 26262 and DO-178C. To monitor the requirements-based testing quality of your models in the Model Testing Dashboard, maintain your artifacts in a project and follow these considerations. For more information on using the Model Testing Dashboard, see “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80.

Manage Artifact Files in a Project

To analyze your requirements-based testing activities in the **Model Testing Dashboard**, store your design and testing artifacts in a MATLAB project. The artifacts that the testing metrics analyze include:

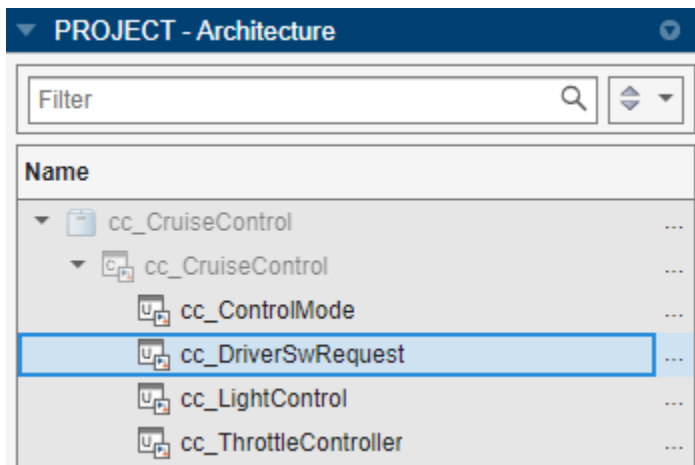
- Models
- Libraries that the models use
- Requirements that you create in Requirements Toolbox
- Tests that you create in Simulink Test
- Test results from the executed tests

For information on how the dashboard traces dependencies between project files, see “Digital Thread” on page 5-167.

When your project contains many models and model reference hierarchies, you can track your unit testing activities by configuring the dashboard to recognize the different testing levels of your models. You can specify which entities in your software architecture are units or higher-level components by labeling them in your project and configuring the Model Testing Dashboard to recognize the labels. The dashboard organizes your models in the **Artifacts** panel according to their testing levels and the model reference hierarchy. For more information, see “Categorize Models in a Hierarchy as Components or Units” on page 5-119.

Trace Artifacts to Units and Components

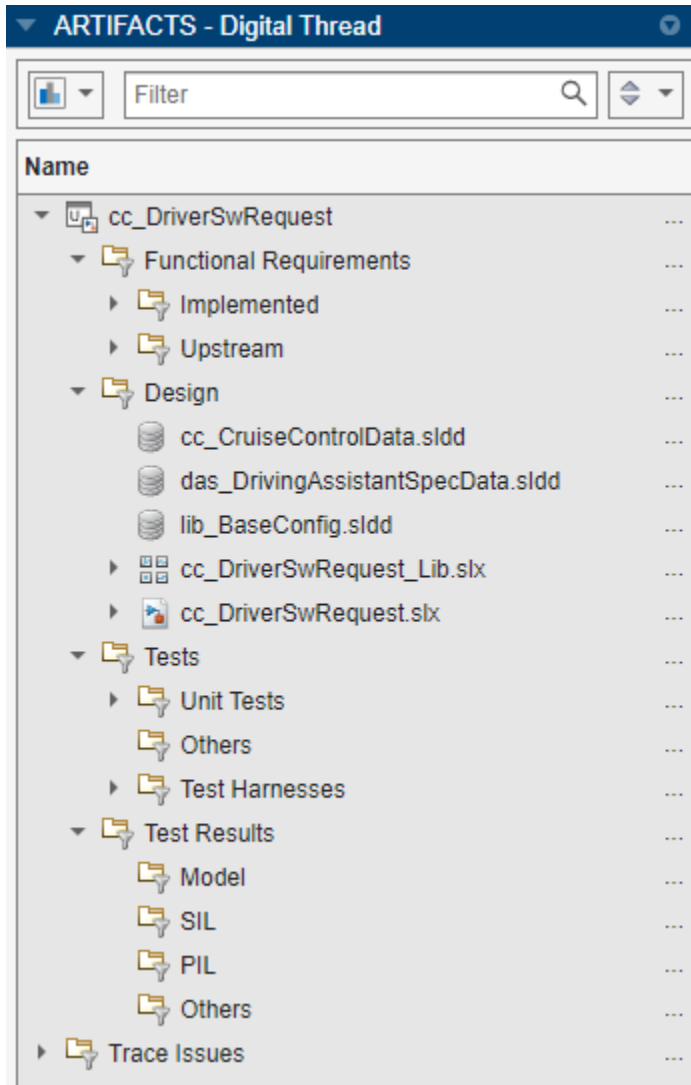
To determine which artifacts are in the scope of a unit or component, the dashboard analyzes the traceability links between the artifacts, software unit models, and component models in the project. The **Project** panel lists the units, organized by the components that reference them.



When you select a unit or component in the **Project** panel, the **Artifacts** panel shows the artifacts that trace to the selected unit or component. Traced artifacts include:

- Functional Requirements
- Design Artifacts

- Tests
- Test Results



To see the traceability path that the dashboard found between a unit or component and its artifacts, click **Trace View** in the toolstrip. Trace views are interactive diagrams that you can use to see how artifacts trace to units and components in your design and to view the traceability information for requirements, tests, and test results in the project. For more information, see “Explore Traceability Information for Units and Components” on page 5-205.

In the **Artifacts** panel, the folder **Trace Issues** contains unexpected requirement links, requirements links which are broken or not supported by the dashboard, and artifacts that the dashboard cannot trace to a unit or component. To help identify the type of tracing issue, the folder **Trace Issues** contains subfolders for **Unexpected Implementation Links**, **Unresolved and Unsupported Links**, **Untraced Tests**, and **Untraced Results**. For more information, see “Fix Requirements-Based Testing Issues” on page 5-89.

If an artifact returns an error during traceability analysis, the panel includes the artifact in an **Errors** folder. Use the traceability information in these sections to check if the artifacts trace to the units or components that you expect. To see details about the warnings and errors that the dashboard finds during artifact analysis, click **Artifact Issues** in the toolstrip.

Functional Requirements

The folder **Functional Requirements** shows requirements of **Type Functional** that are either implemented by or upstream of the unit or component.

When you collect metric results, the dashboard analyzes only the functional requirements that the unit or component directly implements. The folder **Functional Requirements** contains two subfolders to help identify which requirements are implemented by the unit or component, or are upstream of the unit or component:

- **Implemented** — Functional requirements that are directly linked to the unit or component with a link **Type of Implements**. The dashboard uses these requirements in the metrics for the unit or component.
- **Upstream** — Functional requirements that are indirectly or transitively linked to the implemented requirements. The dashboard does not use these requirements in the metrics for the unit or component.

Use the Requirements Toolbox to create or import the requirements in a requirements file (.slreqx). If a requirement does not trace to a unit or component, it appears in the “Trace Issues” folder. If a requirement does not appear in the **Artifacts** panel when you expect it to, see “Requirement Missing from Artifacts Panel” on page 5-130.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread” on page 5-167.

Design Artifacts

The folder **Design** shows project artifacts that trace to the current unit or component, including:

- The model file that contains the block diagram for the unit or component.
- Models that the unit or component references.
- Libraries that are partially or fully used by the model.
- Data dictionaries that are linked to the model.
- External MATLAB code that traces to the model. If you expect external MATLAB code to appear in the dashboard and it does not, see “External MATLAB Code Missing from Artifacts Panel” on page 5-155.

If an artifact does not appear in the **Design** folder when you expect it to, see “Resolve Missing Artifacts, Links, and Results” on page 5-128. For more information on how the dashboard traces dependencies between project files, see “Digital Thread” on page 5-167.

Tests

The folder **Tests** shows tests and test harnesses that trace to the selected unit. A *test* can be either:

- A test iteration
- A test case without iterations

When you collect metric results for a unit, the dashboard analyzes only the unit tests. The folder **Tests** contains subfolders to help identify whether a test is considered a unit test and which test harnesses trace to the unit:

- **Unit Tests** — Tests that the dashboard considers as unit tests. A unit test directly tests either the entire unit or lower-level elements in the unit, like subsystems. The dashboard uses these tests in the metrics for the unit.
- **Others** — Tests that trace to the unit but that the dashboard does not consider as unit tests. For example, the dashboard does not consider tests on a library or tests on a virtual subsystem to be unit tests. The dashboard does not use these tests in the metrics for the unit.
- **Test Harnesses** — Test harnesses that trace to the unit or lower-level elements in the unit. Double-click a test harness to open it.

Create tests by using Simulink Test. If a test does not trace to a unit, it appears in the “Trace Issues” folder. If a test does not appear in the **Artifacts** panel when you expect it to, see “Test Missing from Artifacts Panel” on page 5-130. For troubleshooting tests in metric results, see “Fix a test that does not produce metric results” on page 5-135.



For more information on how the dashboard traces dependencies between project files, see “Digital Thread” on page 5-167.

Test Results

When you collect metric results for a unit, the dashboard analyzes only the test results from unit tests. The folder **Test Results** contains subfolders to help identify which test results are from unit tests.

- The subfolders for **Model**, **SIL**, and **PIL** contain simulation results from normal, software-in-the-loop (SIL), and processor-in-the-loop (PIL) unit tests, respectively. The dashboard uses these results in the metrics for the unit.

The following types of test results are shown:

-  Saved test results — results that you have collected in the Test Manager and have exported to a results file.
-  Temporary test results — results that you have collected in the Test Manager but have not exported to a results file. When you export the results from the Test Manager the dashboard analyzes the saved results instead of the temporary results. Additionally, the dashboard stops recognizing the temporary results when you close the project or close the result set in the Simulink Test Result Explorer. If you want to analyze the results in a subsequent test session or project session, export the results to a results file.
- **Others** — Results that are not simulation results, are not from unit tests, or are only reports. For example, SIL results are not simulation results. The dashboard does not use these results in the metrics for the unit.

If a test result does not trace to a unit, it appears in the “Trace Issues” folder. If a test result does not appear in the **Artifacts** panel when you expect it to, see “Test Result Missing from Artifacts Panel” on page 5-130. For troubleshooting test results in dashboard metric results, see “Fix a test result that does not produce metric results” on page 5-135.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread” on page 5-167.

Trace Issues

The folder **Trace Issues** shows artifacts that the dashboard has not traced to any units or components. Use the folder **Trace Issues** to check if artifacts are missing traceability to the units or components. The folder **Trace Issues** contains subfolders to help identify the type of tracing issue:

- **Unexpected Implementation Links** — Requirement links of **Type Implements** for a requirement of **Type Container** or **Type Informational**. The dashboard does not expect these links to be of **Type Implements** because container requirements and informational requirements do not contribute to the Implementation and Verification status of the requirement set that they are in. If a requirement is not meant to be implemented, you can change the link type. For example, you can change a requirement of **Type Informational** to have a link of **Type Related to**.
- **Unresolved and Unsupported Links** — Requirements links that are either broken in the project or not supported by the dashboard. For example, if a model block implements a requirement, but you delete the model block, the requirement link is now unresolved. The dashboard does not support traceability analysis for some artifacts and some links. If you expect a link to trace to a unit or component and it does not, see the troubleshooting solutions in “Resolve Missing Artifacts, Links, and Results” on page 5-128.
- **Untraced Tests** — Tests that execute on models or lower-level elements, like subsystems, that are not on the project path.
- **Untraced Results** — Results that the dashboard cannot trace to a test. For example, if a test produces a result, but you delete the test, the dashboard cannot trace the results to the test.

The dashboard does not support traceability analysis for some artifacts and some links. If an artifact is untraced when you expect it to trace to a unit or component, see the troubleshooting solutions in “Trace Issues” on page 5-133.

Artifact Errors

The folder **Errors** appears if artifacts returned errors when the dashboard performed artifact analysis. These are some errors that artifacts might return during traceability analysis:

- An artifact returns an error if it has unsaved changes when traceability analysis starts.
- A test results file returns an error if it was saved in a previous version of Simulink.

Open these artifacts and fix the errors. The dashboard shows a banner at the top of the dashboard to indicate that the artifact traceability shown in the **Project** and **Artifacts** panels is outdated. Click the **Trace Artifacts** button on the banner to refresh the data in the **Project** and **Artifacts** panels.

Artifact Issues

To see details about artifacts that cause errors, warnings, and informational messages during analysis, click **Artifact Issues** in the toolstrip. The issues persist between MATLAB sessions and you can sort the messages by their severity, message, source, or message ID.

The messages show:

- Modeling constructs that the dashboard does not support
- Links that the dashboard does not trace
- Test harnesses or cases that the dashboard does not support
- Test results missing coverage or simulation results
- Artifacts that return errors when the dashboard loads them
- Information about model callbacks that the dashboard deactivates
- Artifacts that are not on the path and are not considered during tracing

Collect Metric Results

The Model Testing Dashboard can collect metric results for each unit listed in the **Project** panel. Each metric in the dashboard measures a different aspect of the quality of your model testing and reflects guidelines in industry-recognized software development standards, such as ISO 26262 and DO-178. For more information about the available metrics and the results that they return, see “Model Testing Metrics”.

As you edit and save the artifacts in your project, the dashboard detects changes to the artifacts. If the metric results might be affected by your artifact changes, the dashboard shows a warning banner at the top of the dashboard to indicate that the metric results are stale. Affected widgets have a gray



staleness icon **STALE**.

To update the results, click the **Collect** button on the warning banner to re-collect the metric data and to update the stale widgets with data from the current artifacts. If you want to collect metrics for each of the units and components in the project, click **Collect > Collect All**.

When you change a coverage filter file that your test results use, the coverage metrics in the dashboard do not indicate stale data or include the changes. After you save the changes to the filter file, re-run the tests and use the filter file for the new results.

See Also

“Model Testing Metrics”

Related Examples

- “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80
- “Resolve Missing Artifacts, Links, and Results” on page 5-128

Assess Requirements-Based Testing for ISO 26262

You can use the Model Testing Dashboard to assess the quality and completeness of your requirements-based testing activities in accordance with ISO 26262-6:2018. The dashboard facilitates this activity by monitoring the traceability between requirements, tests, and test results and by providing a summary of testing completeness and structural coverage. The dashboard analyzes the implementation and verification artifacts in a project and provides:

- Completeness and quality metrics for the requirements-based tests in accordance with ISO 26262-6:2018, Clause 9.4.3
- Completeness and quality metrics for the requirements-based test results in accordance with ISO 26262-6:2018, Clause 9.4.4
- A list of artifacts in the project, organized by the units

To assess the completeness of your requirements-based testing activities, follow these automated and manual review steps using the Model Testing Dashboard.

Open the Model Testing Dashboard and Collect Metric Results

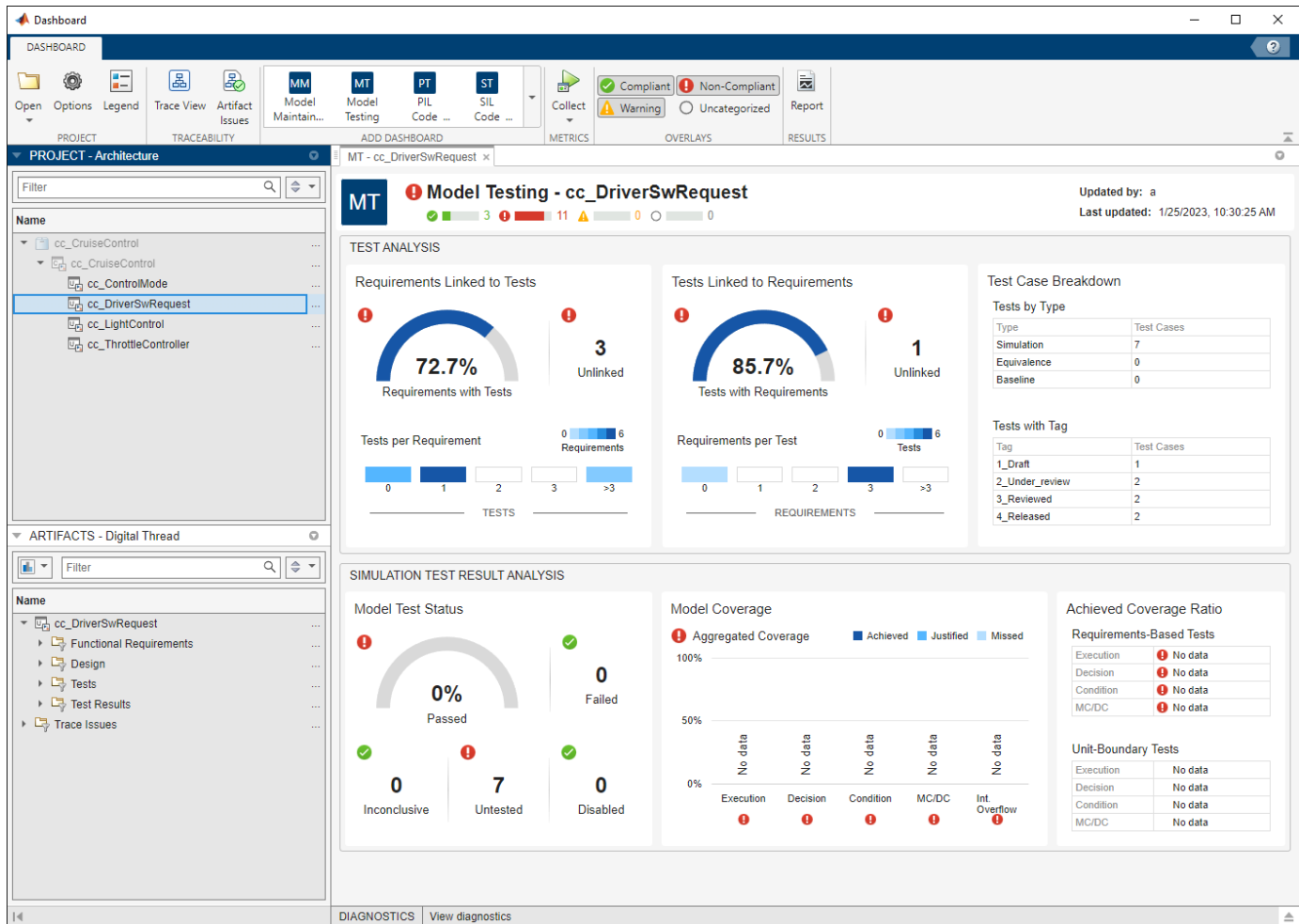
To analyze testing artifacts using the Model Testing Dashboard:


- 1 Open a project that contains your models and testing artifacts. Or to load an example project for the dashboard, in the MATLAB Command Window, enter:

```
dashboardCCProjectStart("incomplete")
```
- 2 Open the dashboard. To open the Model Testing Dashboard, use one of these approaches:
 - On the **Project** tab, click **Model Testing Dashboard**.
 - At the MATLAB command line, enter:

```
modelTestingDashboard
```
- 3 In the **Artifacts** panel, the dashboard organizes artifacts such as requirements, tests, and test results under the units that they trace to. To view the metric results for the unit `cc_DriverSwRequest` in the example project, in the **Project** panel, click **cc_DriverSwRequest**. The dashboard collects metric results and populates the widgets with the metric data for the unit.

Note If you do not specify the models that are considered units, then the Model Testing Dashboard considers a model to be a unit if it does not reference other models. You can control which models appear as units and components by labeling them in your project and configuring the Model Testing Dashboard to recognize the labels. For more information, see “Specify Models as Components and Units” on page 5-120.



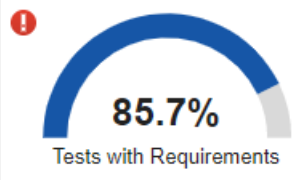
The dashboard widgets summarize the traceability and completeness measurements for the testing artifacts for each unit. The metric results displayed with the red **Non-Compliant** overlay icon  indicate issues that you may need to address to complete requirements-based testing for the unit. Results are compliant if they show full traceability, test completion, or model coverage. To see the compliance thresholds for a metric, point to the overlay icon. To explore the data in more detail, click an individual metric widget. For the selected metric, a table displays the artifacts and the metric value for each artifact. The table provides hyperlinks to open the artifacts so that you can get detailed metric results and fix the artifacts that have issues. For more information about using the Model Testing Dashboard, see “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80.

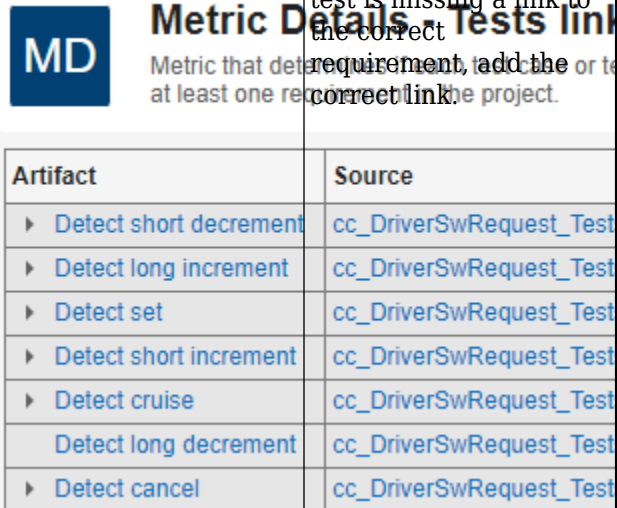
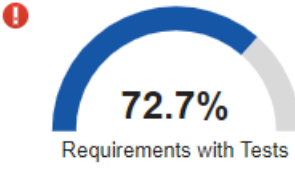
Test Review

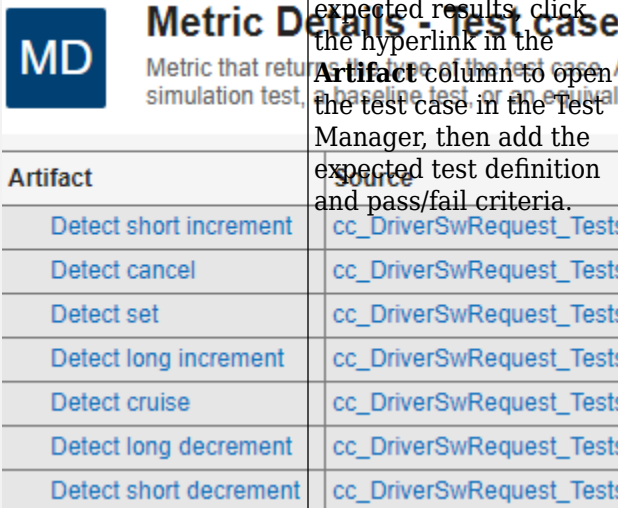
To verify that a unit satisfies its requirements, you create tests for the unit based on the requirements. ISO 26262-6, Clause 9.4.3 requires that tests for a unit are derived from the requirements. When you create a test for a requirement, you add a traceability link between the test and the requirement, as described in “Link Requirements to Tests” (Requirements Toolbox) and in “Establish Requirements Traceability for Testing” (Simulink Test). Traceability allows you to track which requirements have been verified by your tests and identify requirements that the model does not satisfy. Clause 9.4.3 requires traceability between requirements and tests, and review of the

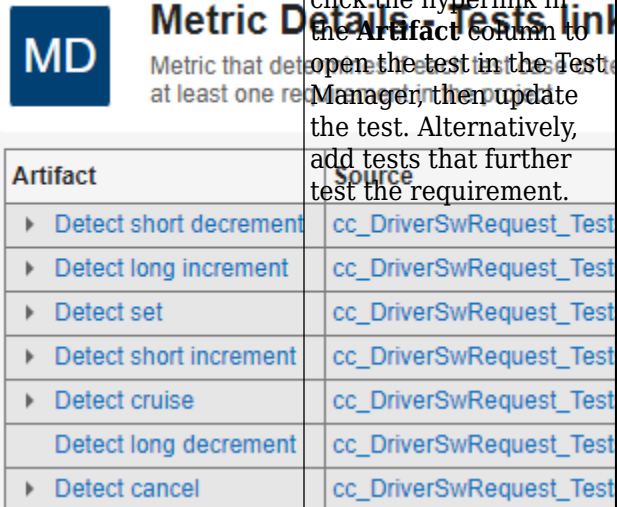
correctness and completeness of the tests. To assess the correctness and completeness of the tests for a unit, use the metrics in the **Test Analysis** section of the Model Testing Dashboard.

The following is an example checklist provided to aid in reviewing test correctness and completeness with respect to ISO 26262-6. For each question, perform the review activity using the corresponding dashboard metric and apply the corresponding fix. This checklist is provided as an example and should be reviewed and modified to meet your application needs.

Checklist Item	Review Activity	Dashboard Metric	Fix
<p>1 — Does each test trace to a requirement?</p>	<p>Check that 100% of the tests for the unit are linked to requirements by viewing the widgets in the Tests Linked to Requirements section.</p>	<p>Tests Linked to Requirements</p>  <p>Metric ID — TestCaseWithRequirementDistribution</p> <p>For more information, see “Test Case with Requirement Distribution” on page 5-247.</p>	<p>For each unlinked test, add a link to the requirement that the test verifies, as described in “Fix Requirements-Based Testing Issues” on page 5-89.</p>

Checklist Item	Review Activity	Dashboard Metric	Fix
<p>2 — Does each test trace to the correct requirements?</p>	<p>For each test, manually verify that the requirement it is linked to is correct. Click the Tests with Requirements widget to view a table of the tests. To see the requirements that a test traces to, in the Artifacts column, click the arrow to the left of the test name.</p>	<p>Tests Linked to Requirements</p>  <p>Metric ID — TestCaseWithRequirement</p> <p>For more information, see “Test Case with Requirement” on page 5-245.</p>	<p>For each link to an incorrect requirement, remove the link. If the test is missing a link to the correct requirement, add the correct link.</p>
<p>3 — Do the tests cover all requirements?</p>	<p>Check that 100% of the requirements for the unit are linked to tests by viewing the widgets in the Requirements Linked to Tests section.</p>	<p>Requirements Linked to Tests</p>  <p>Metric ID — RequirementWithTestCaseDistribution</p> <p>For more information, see “Requirement with Test Case Distribution” on page 5-240.</p>	<p>For each unlinked requirement, add a link to the test that verifies it, as described in “Fix Requirements-Based Testing Issues” on page 5-89.</p>


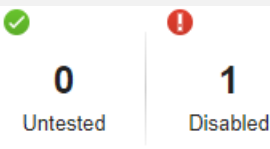

Checklist Item	Review Activity	Dashboard Metric	Fix
<p>4 — Do the test cases define the expected results including pass/fail criteria?</p>	<p>Manually review the test cases of each type. Click the widgets in the Tests by Type section to view a table of the test cases for each type: Simulation, Equivalence, and Baseline. Open each test case in the Test Manager by using the hyperlinks in the Artifact column. Baseline test cases must define baseline criteria. For simulation test cases, review that each test case defines pass/fail criteria by using assessments, as described in “Assess Simulation and Compare Output Data” (Simulink Test).</p>	<p>Tests by Type</p>  <p>Metric ID — TestCaseType</p> <p>For more information, see “Test Case Type” on page 5-253.</p>	<p>For each test case that does not define expected results, click the hyperlink in the Artifact column to open the test case in the Test Manager, then add the expected test definition and pass/fail criteria.</p>

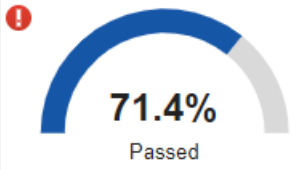
Checklist Item	Review Activity	Dashboard Metric	Fix																					
5 — Does each test properly test the requirement that it traces to?	Manually review the requirement links and content for each test. Click the Tests with Requirements widget to view a table of the tests. To see the requirements that a test traces to, in the Artifact column, click the arrow to the left of the test name. Use the hyperlinks to open the test and requirement and review that the test properly tests the requirement.	<p>Tests Linked to Requirements</p>  <p>Artifact</p> <table border="1"> <tr> <td>▶ Detect short decrement</td> <td>cc_DriverSwRequest_Tests.mldatx</td> <td>Lin</td> </tr> <tr> <td>▶ Detect long increment</td> <td>cc_DriverSwRequest_Tests.mldatx</td> <td>Lin</td> </tr> <tr> <td>▶ Detect set</td> <td>cc_DriverSwRequest_Tests.mldatx</td> <td>Lin</td> </tr> <tr> <td>▶ Detect short increment</td> <td>cc_DriverSwRequest_Tests.mldatx</td> <td>Lin</td> </tr> <tr> <td>▶ Detect cruise</td> <td>cc_DriverSwRequest_Tests.mldatx</td> <td>Lin</td> </tr> <tr> <td>▶ Detect long decrement</td> <td>cc_DriverSwRequest_Tests.mldatx</td> <td>Mis</td> </tr> <tr> <td>▶ Detect cancel</td> <td>cc_DriverSwRequest_Tests.mldatx</td> <td>Lin</td> </tr> </table> <p>Metric ID — TestCaseWithRequirement</p> <p>For more information, see “Test Case with Requirement” on page 5-245.</p>	▶ Detect short decrement	cc_DriverSwRequest_Tests.mldatx	Lin	▶ Detect long increment	cc_DriverSwRequest_Tests.mldatx	Lin	▶ Detect set	cc_DriverSwRequest_Tests.mldatx	Lin	▶ Detect short increment	cc_DriverSwRequest_Tests.mldatx	Lin	▶ Detect cruise	cc_DriverSwRequest_Tests.mldatx	Lin	▶ Detect long decrement	cc_DriverSwRequest_Tests.mldatx	Mis	▶ Detect cancel	cc_DriverSwRequest_Tests.mldatx	Lin	For each test that does not properly test the requirement it traces to, click the hyperlink in the Artifact column to open the test in the Test Manager, then update the test. Alternatively, add tests that further test the requirement.
▶ Detect short decrement	cc_DriverSwRequest_Tests.mldatx	Lin																						
▶ Detect long increment	cc_DriverSwRequest_Tests.mldatx	Lin																						
▶ Detect set	cc_DriverSwRequest_Tests.mldatx	Lin																						
▶ Detect short increment	cc_DriverSwRequest_Tests.mldatx	Lin																						
▶ Detect cruise	cc_DriverSwRequest_Tests.mldatx	Lin																						
▶ Detect long decrement	cc_DriverSwRequest_Tests.mldatx	Mis																						
▶ Detect cancel	cc_DriverSwRequest_Tests.mldatx	Lin																						

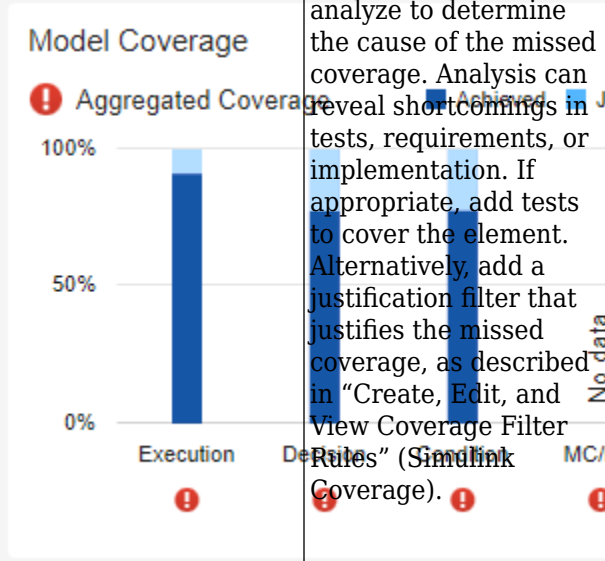
Test Results Review

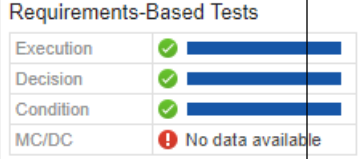
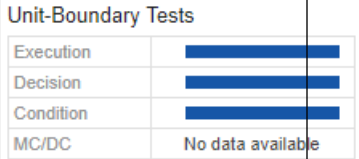
After you run tests on a unit, you must review the results to check that the tests executed, passed, and sufficiently tested the unit. Clause 9.4.4 in ISO 26262-6:2018 requires that you analyze the coverage of requirements for each unit. Check that each of the tests tested the intended model and passed. Additionally, measure the coverage of the unit by collecting model coverage results in the tests. To assess the testing coverage of the requirements for the unit, use the metrics in the **Simulation Test Result Analysis** section of the Model Testing Dashboard.

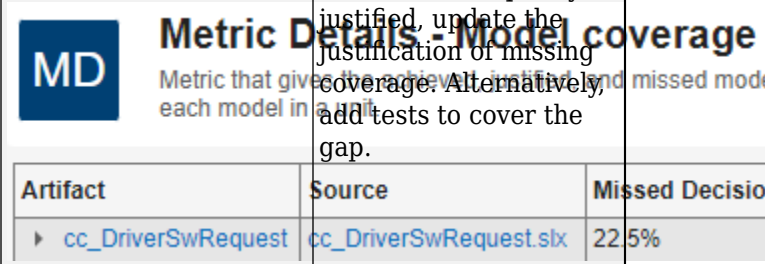
The following checklist is provided to facilitate test results analysis and review using the dashboard. For each question, perform the review activity using the corresponding dashboard metric and apply the corresponding fix. This checklist is provided as an example and should be reviewed and modified to meet your application needs.

Checklist Item	Review Activity	Dashboard Metric	Fix
1 — Does each test result trace to a test?	Use only test results that appear in the dashboard. Test results that do not trace to a test do not appear in the dashboard. Click a widget in the Model Test Status section to view a table of the tests and the results that trace to them.	<p>Model Test Status</p>  <p>Metric ID — slcomp.mt.TestStatusDistribution</p> <p>For more information, see “Model Test Status Distribution”.</p>	Open the metric details and click the hyperlink in the Artifacts column to open the test in the Test Manager. Re-run the tests that the results should trace to and export the new results.
2 — Does each test trace to a test result?	Check that zero tests are untested and zero tests are disabled.	<p>Model Test Status</p>  <p>Metric ID — slcomp.mt.TestStatusDistribution</p> <p>For more information, see “Model Test Status Distribution”.</p>	For each disabled or untested test, in the Test Manager, enable and run the test.
3 — Have all tests been executed?	Check that zero tests are untested and zero tests are disabled.	<p>Model Test Status</p>  <p>Metric ID — slcomp.mt.TestStatusDistribution</p> <p>For more information, see “Model Test Status Distribution”.</p>	For each disabled or untested test, in the Test Manager, enable, and run the test.

Checklist Item	Review Activity	Dashboard Metric	Fix
4 — Do all tests pass?	Check that 100% of the tests for the unit passed.	<p data-bbox="865 300 1105 331">Model Test Status</p>  <p data-bbox="865 569 1154 663">Metric ID — slcomp.mt.TestStat usDistribution</p> <p data-bbox="865 688 1146 783">For more information, see “Model Test Status Distribution”.</p>	For each test failure, review the failure in the Test Manager and fix the corresponding test or design element in the model.
5 — Do all test results include coverage results?	Manually review each test result in the Test Manager to check that it includes coverage results.	Not applicable	For each test result that does not include coverage, open the test in the Test Manager, and then enable coverage collection. Run the test again.

Checklist Item	Review Activity	Dashboard Metric	Fix
<p>6 — Were the required structural coverage objectives achieved for each unit?</p>	<p>Check that the tests achieved 100% model coverage for the coverage types that your unit testing requires. To determine the required coverage types, consider the safety level of your software unit and use table 9 in clause 9.4.4 of ISO 26262-6:2018.</p>	<p>Model Coverage</p>  <p>Metric ID — slcomp.mt.Coverage Breakdown</p> <p>For more information, see “Model Coverage Breakdown” on page 5-258.</p>	<p>For each design element that is not covered, analyze to determine the cause of the missed coverage. Analysis can reveal shortcomings in tests, requirements, or implementation. If appropriate, add tests to cover the element. Alternatively, add a justification filter that justifies the missed coverage, as described in “Create, Edit, and View Coverage Filter Rules” (Simulink Coverage).</p>

Checklist Item	Review Activity	Dashboard Metric	Fix
<p>7 — Does all of the overall achieved coverage come from requirements-based tests?</p>	<p>Check that 100% of the overall achieved coverage comes from requirements-based tests.</p>	<p>View the metric results in the Achieved Coverage Ratio subsection of the dashboard. Click the widgets under Requirements-Based Tests for information on the source of the overall achieved coverage for each coverage type.</p>  <p>For the associated metric IDs, see “Requirements-Based Tests”.</p>	<p>For any overall achieved coverage that does not come from requirements-based tests, add links to the requirements that the tests verify.</p>
<p>8 — Does the overall achieved coverage come from tests that properly test the unit?</p>	<p>Manually review the content for each test. Check the percentage of the overall achieved coverage that comes from unit-boundary tests.</p>	<p>View the metric results in the Achieved Coverage Ratio subsection of the dashboard. Click the widgets under Unit-Boundary Tests for information on the source of the overall achieved coverage for each coverage type.</p>  <p>For the associated metric IDs, see “Unit-Boundary Tests”.</p>	<p>For any overall achieved coverage that does not come from unit-boundary tests, either add a test that tests the whole unit or reconsider the unit-model definition.</p>

Checklist Item	Review Activity	Dashboard Metric	Fix
9 — Have shortcomings been acceptably justified?	Manually review coverage justifications. Click a bar in the Model Coverage section to view a table of the results for the corresponding coverage type. To open a test result in the Test Manager for further review, click the hyperlink in the Artifacts column.	<p>Model Coverage</p>  <p>Metric ID — slcomp.mt.Coverage Breakdown</p> <p>For more information, see “Model Coverage Breakdown” on page 5-258.</p>	For each coverage gap that is not acceptably justified, update the justification of missing coverage. Alternatively, add tests to cover the gap.

Unit Verification in Accordance with ISO 26262

The Model Testing Dashboard provides information about the quality and completeness of your unit requirements-based testing activities. To comply with ISO 26262-6:2018, you must also test your software at other architectural levels. ISO 26262-6:2018 describes compliance requirements for these testing levels:

- Software unit testing in Table 7, method 1j
- Software integration testing in Table 10, method 1a
- Embedded software testing in Table 14, method 1a

The generic verification process detailed in ISO 26262-8:2018, clause 9 includes additional information on how you can systematically achieve testing for each of these levels by using planning, specification, execution, evaluation, and documentation of tests. This table shows how the Model Testing Dashboard applies to the requirements in ISO 26262-8:2018, clause 9 for the unit testing level, and complementary activities required to perform to show compliance.

Requirement	Compliance Argument	Complementary Activities
9.4.1 — Scope of verification activity	The Model Testing Dashboard applies to all safety-related and non-safety-related software units.	Not applicable

Requirement	Compliance Argument	Complementary Activities
9.4.2 — Verification methods	The Model Testing Dashboard provides a summary on the completion of requirements-based testing (Table 7, method 1j) including a view on test results.	Where applicable, apply one or more of these other verification methods: <ul style="list-style-type: none"> • Manual review and analysis check list • Applying other tools, such as static code analysis, control flow analysis, and data flow analysis • Developing extra tests, such as interface tests, fault injection tests, and back-to-back comparisons
9.4.3 — Methods for deriving test cases	The Model Testing Dashboard provides several ways to traverse the software unit requirements and the relevant tests, which helps you to derive tests from the requirements.	You can also derive tests by using other tools, such as Simulink Design Verifier.
9.4.4 — Requirement and structural coverage	The Model Testing Dashboard aids in showing: <ul style="list-style-type: none"> • Completeness of requirement coverage • Branch/statement and MCDC model coverage achieved by testing • A rationale for the sufficiency of achieved coverage 	The dashboard provides structural coverage only at the model level. You can use other tools to track the structural coverage at the code level.
9.4.5 — Test environment	The Model Testing Dashboard aids in requirements-based testing at the model level.	Apply back-to-back comparison tests to verify that the behavior of the model is equivalent to the generated code.

References:

- ISO 26262-4:2018(en)Road vehicles — Functional safety — Part 4: Product development at the system level, International Standardization Organization
- ISO 26262-6:2018(en)Road vehicles — Functional safety — Part 6: Product development at the software level, International Standardization Organization
- ISO 26262-8:2018(en)Road vehicles — Functional safety — Part 8: Supporting processes, International Standardization Organization

See Also

“Model Testing Metrics”

Related Examples

- “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80
- “Fix Requirements-Based Testing Issues” on page 5-89

Collect Metrics on Model Testing Artifacts Programmatically

This example shows how to programmatically assess the status and quality of requirements-based testing activities in a project. When you develop software units by using Model-Based Design, you use requirements-based testing to verify your models. You can assess the testing status of one unit by using the metric API to collect metric data on the traceability between requirements and tests and on the status of test results. The metrics measure characteristics of completeness and quality of requirements-based testing that reflect industry standards such as ISO 26262 and DO-178. After collecting metric results, you can access the results or export them to a file. By running a script that collects these metrics, you can automatically analyze the testing status of your project to, for example, design a continuous integration system. Use the results to monitor testing completeness or to detect downstream testing impacts when you make changes to artifacts in the project.

Open the Project

Open a project that contains models and testing artifacts. For this example, in the MATLAB Command Window, enter:

```
dashboardCCProjectStart('incomplete')
```

The example project contains models, and requirements and tests for the models. Some of the requirements have traceability links to the models and tests, which help to verify that the functionality of the model meets the requirements.

The example project also has the project setting **Track tool outputs to detect outdated results** enabled. Before you programmatically collect metrics, make sure that the **Track tool outputs to detect outdated results** setting is enabled for your project. For information, see “Enable Artifact Tracing for the Project” on page 5-128.

Collect Metric Results

Create a `metric.Engine` object for the current project.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to reflect pending artifact changes and to track the test results.

```
updateArtifacts(metric_engine);
```

Create an array of metric identifiers for the metrics you want to collect. For this example, create a list of the metric identifiers used in the Model Testing Dashboard. For more information, see `getAvailableMetricIds`.

```
metric_Ids = getAvailableMetricIds(metric_engine,...  
'App', 'DashboardApp',...  
'Dashboard', 'ModelUnitTesting');
```

For a list of model testing metrics, see “Model Testing Metrics”.

When you collect metric results, you can collect results for one unit at a time or for each unit in the project.

Collect Results for One Unit

When you collect and view results for a unit, the metrics return data for the artifacts that trace to the model.

Collect the metric results for the `cc_DriverSwRequest`.

Create an array that identifies the path to the model file in the project and the name of the model.

```
unit = {fullfile(pwd, 'models', 'cc_DriverSwRequest.slx'), 'cc_DriverSwRequest'};
```

Execute the engine and use `'ArtifactScope'` to specify the unit for which you want to collect results. The engine runs the metrics on only the artifacts that trace to the model that you specify. Collecting results for these metrics requires a Simulink Test license, a Requirements Toolbox license, and a Simulink Coverage license.

```
execute(metric_engine, metric_Ids, 'ArtifactScope', unit)
```

Collect Results for Each Unit in the Project

To collect the results for each unit in the project, execute the engine without the argument for `'ArtifactScope'`.

```
execute(metric_engine, metric_Ids)
```

For more information on collecting metric results, see the function `execute`.

Access Results

Generate a report file that contains the results for all units in the project. For this example, specify the HTML file format, use `pwd` to provide the path to the current folder, and name the report `'MetricResultsReport.html'`.

```
reportLocation = fullfile(pwd, 'MetricResultsReport.html');
generateReport(metric_engine, 'Type', 'html-file', 'Location', reportLocation);
```

The HTML report opens automatically.

To open the table of contents and navigate to results for each unit, click the menu icon in the top-left corner of the report. For each unit in the report, there is an artifact summary table that displays the size and structure of that unit.

1.1. Artifact Summary

Artifact Group	Artifact Type	Number of Artifacts
Requirements	Functional requirement	41
Design	Block diagram	3
	Model reference	0
	Subsystem	17
	Stateflow chart	0
	MATLAB function	0
	Data dictionary file	1
Tests	Test case	37
	Test case result	0

Saving the metric results in a report file allows you to access the results without opening the project and the dashboard. Alternatively, you can open the Model Testing Dashboard to see the results and explore the artifacts.

```
modelTestingDashboard
```

To access the results programmatically, use the `getMetrics` function. The function returns the `metric.Result` objects that contain the result data for the specified unit and metrics. For this example, store the results for the metrics `slcomp.mt.TestStatus` and `TestCasesPerRequirementDistribution` in corresponding arrays.

```
results_TestCasesPerReqDist = getMetrics(metric_engine, 'TestCasesPerRequirementDistribution');
results_TestStatus = getMetrics(metric_engine, 'slcomp.mt.TestStatus');
```

View Distribution of Test Links per Requirement

The metric `TestCasesPerRequirementDistribution` returns a distribution of the number of tests linked to each functional requirement for the unit. Use the `disp` function to display the bin edges and bin counts of the distribution, which are fields in the `Value` field of the `metric.Result` object. The left edge of each bin shows the number of test links and the bin count shows the number of requirements that are linked to that number of tests. The sixth bin edge is `18446744073709551615`, which is the upper limit of the count of tests per requirement, which shows that the fifth bin contains requirements that have four or more tests.

```
disp(['Unit: ', results_TestCasesPerReqDist(1).Scope(1).Name])
disp([' Tests per Requirement: ', num2str(results_TestCasesPerReqDist(1).Value.BinEdges)])
disp([' Requirements: ', num2str(results_TestCasesPerReqDist(1).Value.BinCounts)])
```

```
Unit: cc_DriverSwRequest
Tests per Requirement: 0 1 2 3 4 18446744073709551615
Requirements: 3 6 0 0 2
```

This result shows that for the unit `cc_DriverSwRequest` there are 3 requirements that are not linked to tests, 6 requirements that are linked to one test, and 2 requirements that are linked to four or more tests. Each requirement should be linked to at least one test that verifies that the model meets the requirement. The distribution also allows you to check if a requirement has many more tests than the other requirements, which might indicate that the requirement is too general and that you should break it into more granular requirements.

View Test Status Results

The metric `slcomp.mt.TestStatus` assesses the testing status of each test for the unit and returns one of these numeric results:

- 0 — Failed
- 1 — Passed
- 2 — Disabled
- 3 — Untested

Display the name and status of each test.

```
for n=1:length(results_TestStatus)

    disp(['Test: ', results_TestStatus(n).Artifacts(1).Name])
    disp([' Status: ', num2str(results_TestStatus(n).Value)])

end
```

For this example, the tests have not been run, so each test returns a status of 3.

See Also

“Model Testing Metrics” | `metric.Engine` | `execute` | `generateReport` | `getAvailableMetricIds` | `updateArtifacts`

Related Examples

- “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80
- “Test Model Against Requirements and Report Results” (Requirements Toolbox)
- “Perform Functional Testing and Analyze Test Coverage” (Simulink Test)


Categorize Models in a Hierarchy as Components or Units

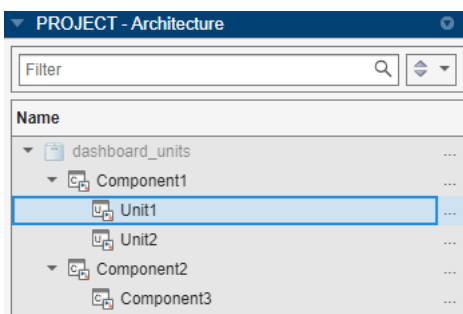
When testing your model-based software architecture, there are different testing requirements for different levels of the architecture. The dashboard helps you to focus on the models that require unit testing so you can assess their testing quality. You can use labels to classify the models in your projects as units or components, then use the dashboard to see the hierarchy. For more information on how to analyze the testing requirements for a unit, see “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80.

Units in the Dashboard

A *unit* is a functional entity in your software architecture that you can execute and test independently or as part of larger system tests. Software development standards, such as ISO 26262-6, define objectives for unit testing. Unit tests typically must cover each of the requirements for the unit and must demonstrate traceability between the requirements, the tests, and the unit. Unit tests must also meet certain coverage objectives for the unit, such as modified condition/decision coverage (MC/DC).

You can label models as units in the dashboard. If you do not specify the models that are considered units, then the dashboard considers a model to be a unit if it does not reference other models.

In the Dashboard window, in the **Project** panel, the unit dashboard icon  indicates a unit. If a unit is referenced by a component, it appears under the component in the **Project** panel. If a unit references one or more other models, those models are part of the unit. The referenced models appear in the **Design** folder under the unit and contribute to the metric results for the unit.



To specify which models are units, label them in your project and configure the dashboard to recognize the label, as shown in “Specify Models as Components and Units” on page 5-120.


Components in the Dashboard

A *component* is an entity that integrates multiple testable units together. For example:

- A model that references multiple unit models could be a component model.
- A System Composer architecture model could be a component. Supported architectures include System Composer architecture models, System Composer software architecture models, and AUTOSAR architectures.
- A component could also integrate other components.

The dashboard organizes components and units under the components that reference them in the **Project** panel.

If you do not specify the models that are considered components, then the dashboard considers a model to be a component if it references one or more other models.

In the Dashboard window, in the **Project** panel, the component icon  indicates a component. To see the units under a component, expand the component node by clicking the arrow next to the component icon.

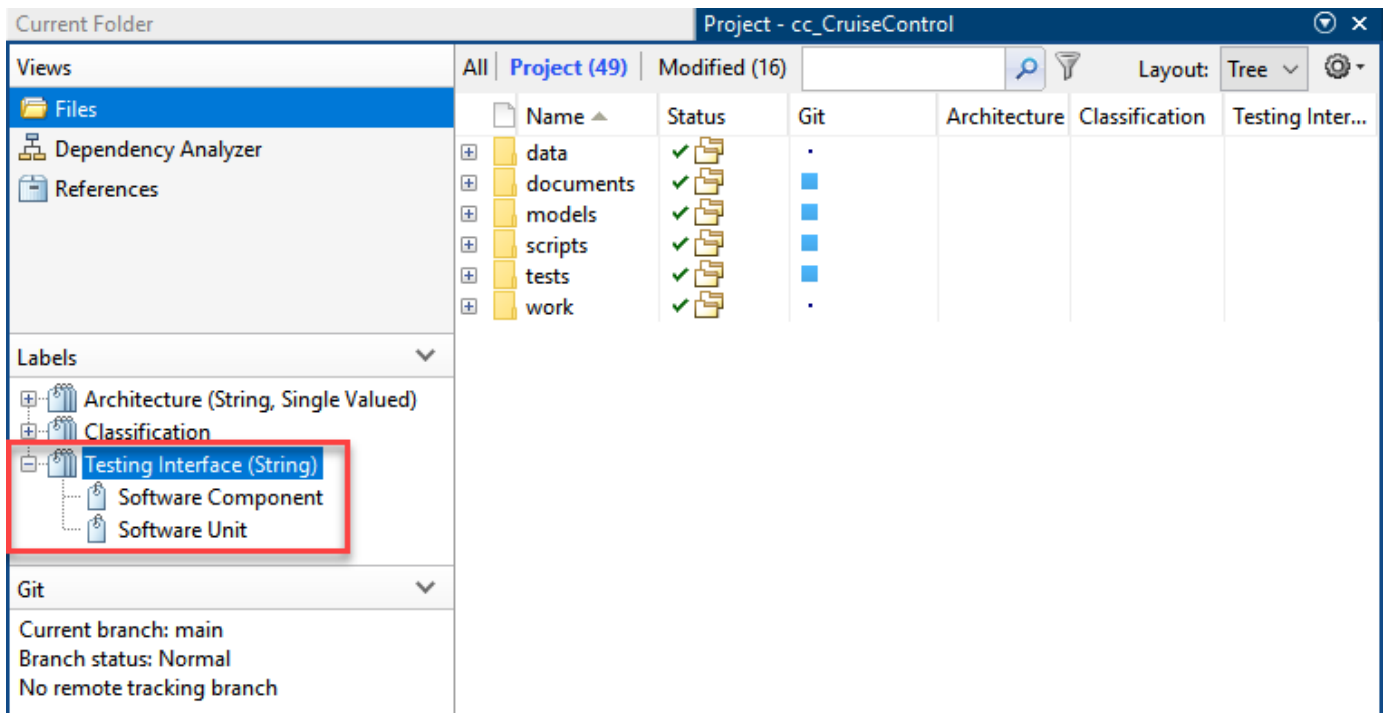
To specify the models that are considered components, label them in your project and configure the dashboard to recognize the label, as shown in “Specify Models as Components and Units” on page 5-120.

Specify Models as Components and Units

You can control which models appear as units and components by labeling them in your project and configuring the dashboard to recognize the labels.

- 1** Open a project. To open the dashboard example project, at the command line, enter `dashboardCCProjectStart`. This example project already has component and unit models configured.
- 2** In MATLAB, at the bottom left of the **Project** window, right-click in the **Labels** pane and click **Create New Category**. Type a name for the category that will contain your testing architecture labels, for example, `Testing Interface` and then click **Create**.
- 3** Create a label for the units. On the **Labels** pane, right-click the category that you created and click **Create New Label**. Type the label name `Software Unit` and click **OK**.
- 4** Create another label for component models and name the label `Software Component`.

The unit and component labels appear under the category in the **Labels** pane.




- 5 Label the models in the project as components and units. In the project pane, right-click a model and click **Add Label**. In the dialog box, select the label and click **OK**. For this example, apply these labels:
 - cc_CruiseControl — Software Component
 - cc_ControlMode — Software Unit
 - cc_DriverSwRequest — Software Unit
 - cc_LightControl — Software Unit
 - cc_TargetSpeedThrottle — Software Unit
- 6 Open the Dashboard window by using one of these approaches:
 - On the **Project** tab, in the **Tools** section, click **Model Testing Dashboard**.
 - On the **Project** tab, in the **Tools** section, click **Model Design Dashboard**.
- 7 In the **Dashboard** tab, click **Options**.
- 8 In the Project Options dialog box, in the **Classification** section, specify the category and labels that you created for the components and units. For the component interface, set **Category** to Testing Interface and **Label** to Software Component. For the unit interface, set **Category** to Testing Interface and **Label** to Software Unit.


Project Options ✕

Configure options for project "cc_CruiseControl". Options are saved in the project definition files and are shared with everyone using this project. ?

Classification

Classify models as **component interface** , if they have this label:

Category Label

Classify models as **unit interface** , if they have this label:

Category Label

- 9 Click **Apply**. The dashboard updates the traceability information in the **Project** panel and organizes the models under the component models that reference them. If a model is not referenced by a component, it appears at the top level with the components.

To open a dashboard for a unit or component, click the name of the unit or component in the **Project** panel. The dashboard shows the metric results for the unit or component you select.

See Also

Related Examples

- "Create Labels"
- "Add Labels to Files"

Monitor Low-Level Test Results in the Model Testing Dashboard

This example shows how you can use low-level tests to address gaps in your achieved coverage and how to monitor the percentage of the overall achieved coverage that comes from the unit-boundary tests.

- A *low-level test* is a model test that only tests a sub-element of a unit. Sub-elements include atomic subsystems, atomic subsystem references, atomic Stateflow® charts, atomic MATLAB® Function blocks, and referenced models. For example, if your test only tests an atomic subsystem inside a unit, the test is a low-level test.
- A *unit-boundary test* is a model test that tests the whole unit.

You can use both unit-boundary tests and low-level tests to achieve coverage. When you run a unit-boundary test, the test has access to the entire context of the unit design. When you run low-level tests, you only test part of the unit. If a high percentage of your overall achieved coverage comes from low-level tests, you may want to expand the scope of your testing by using unit-boundary tests instead.

This example uses Simulink® Check™, Simulink Test™, and Simulink Coverage™.

Open Project Dashboard

1. Open a project that contains models and testing artifacts. For this example, in the MATLAB Command Window, enter:

```
dashboardCCProjectStart
```

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

- On the **Project** tab, click **Model Testing Dashboard**.
- At the command line, enter:

```
modelTestingDashboard
```

The dashboard opens a **Model Testing** tab for the unit `cc_ControlMode`.

Run Unit-Boundary Test and Identify Model Coverage Gaps

Run a test on the whole unit and use the Model Testing Dashboard to collect the metric results for model coverage.

1. Open the test file associated with the unit `cc_ControlMode`. In the Model Testing Dashboard, in the **Artifacts** panel, expand the **Tests > Unit Tests** folder and double-click the file **cc_ControlMode_Tests.mldatx**.

The file contains three test suites:

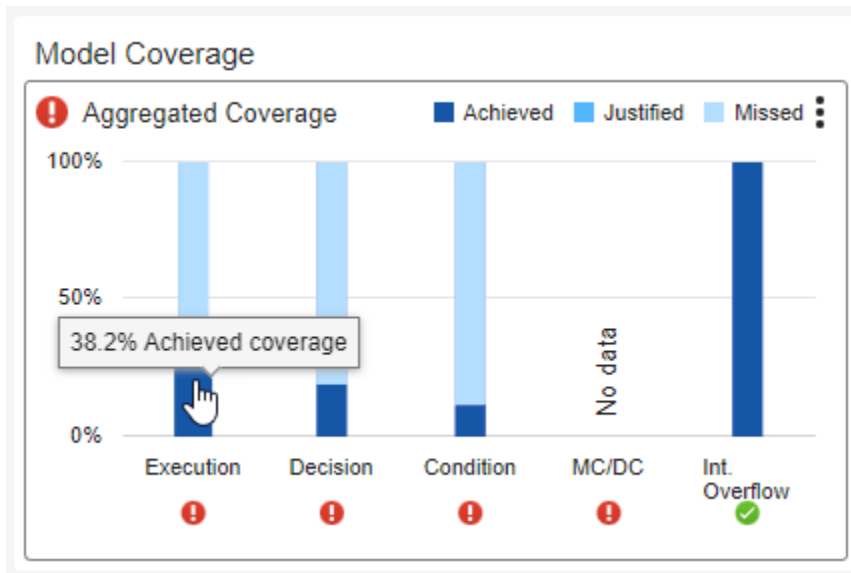
- `Control Mode Unit Tests` — Tests for the atomic subsystem `Control_Mode_StateMachine`
- `Target Speed Unit Tests` — Tests for the atomic subsystem `Target_Speed_Calculator`
- `Combined Tests` — A test for the unit `cc_ControlMode`

2. Run the test suite `Combined Tests` by right-clicking **Combined Tests** and clicking **Run**. The test suite contains the test case `Idle`, which is a unit-boundary test for the unit `cc_ControlMode`.

3. To refresh the **Model Coverage** widgets and view the model coverage results, click **Model Testing Dashboard** in the Test Manager toolstrip and click the **Collect** button on the Dashboard warning banner.

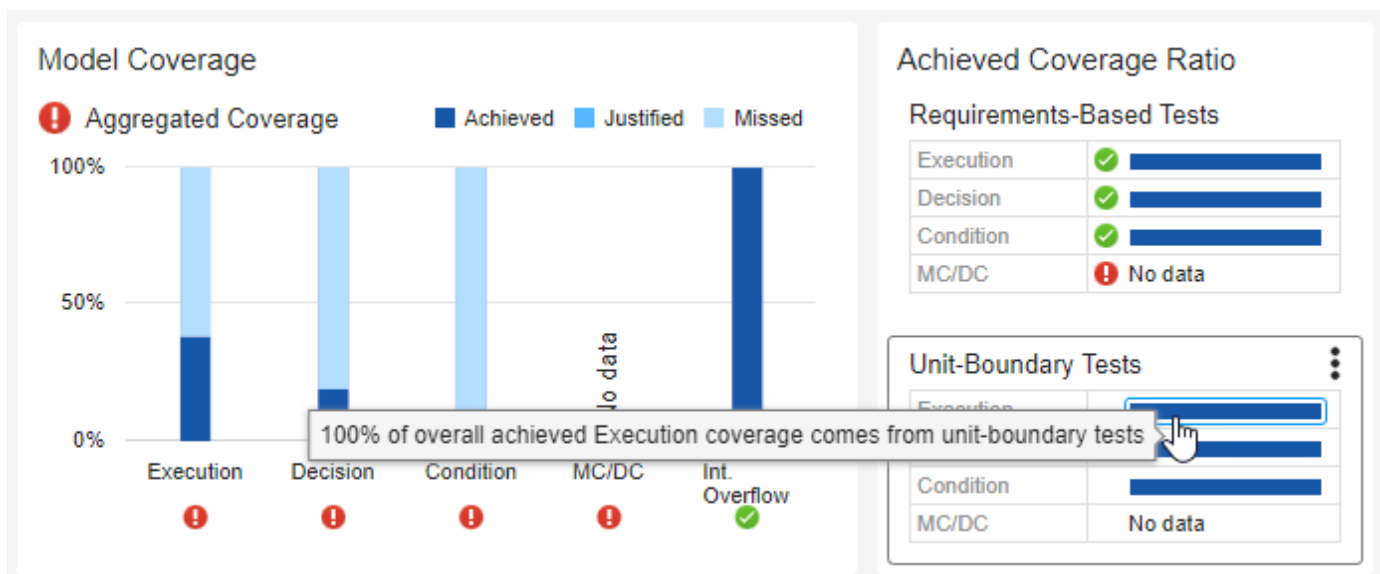
4. View the achieved execution coverage for the unit by pointing to the **Execution** bar in the **Model Coverage** subsection.

Based on the results from this unit-boundary test, the achieved execution coverage is 38.2%.



5. To view the overall achieved execution coverage from unit-boundary tests, point to the **Execution** bar in the **Unit-Boundary Tests** section under **Achieved Coverage Ratio**.

Since you only executed a unit-boundary test, Idle, 100% of the overall achieved execution coverage comes from unit-boundary tests.



Use Low-Level Testing to Address Coverage Gaps

Run low-level tests and use the Model Testing Dashboard to observe the increase in model coverage.

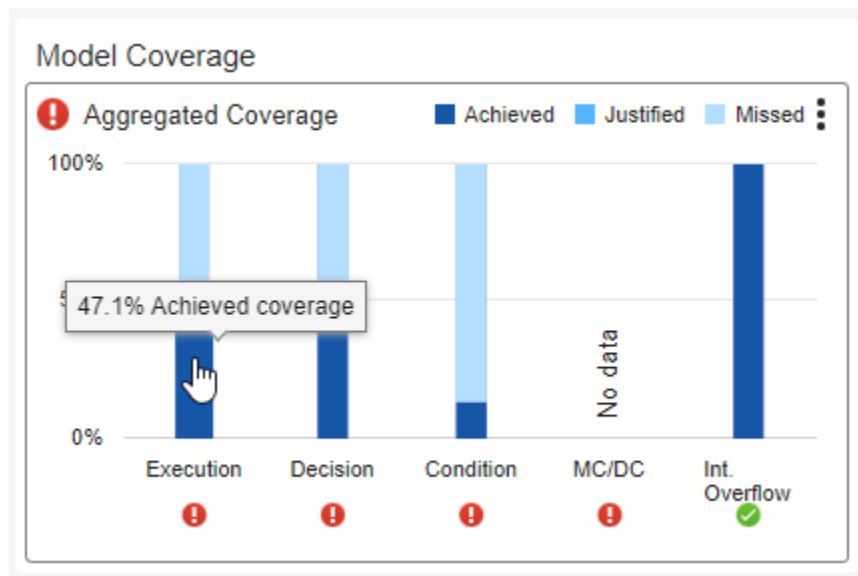
1. In the Test Manager, click **Test Browser**.
2. Run the test suite Target Speed Unit Tests by right-clicking **Target Speed Unit Tests** and clicking **Run**.
3. In the Model Testing Dashboard, click the **Collect** button on the warning banner or toolstrip to refresh the model coverage widgets. In the **Model Coverage** section, the aggregated model coverage now includes the results from the low-level tests in the test suite Target Speed Unit Tests.

Note that you can only collect aggregated coverage for low-level tests if you have previously run a unit-boundary test. The dashboard cannot calculate aggregated coverage for units that only have test results from low-level tests because lower-level tests do not test the whole unit. The unit-boundary test must be executed on the model, not by software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulations on the model code.

For this example, you could not calculate aggregated coverage without the test results from the test suite Combined Tests. The test suites Control Mode Unit Tests and Target Speed Unit Tests only contain low-level tests and therefore do not test the whole unit.

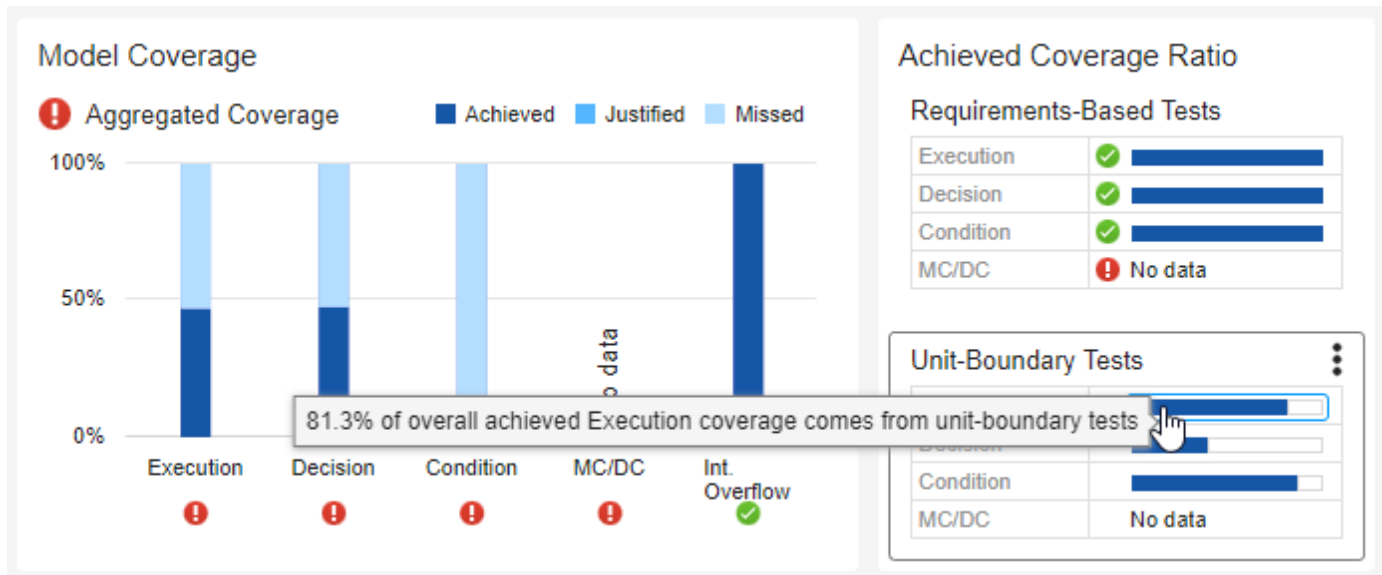
4. View the achieved execution coverage for the unit. In the **Model Coverage** section, point to the achieved coverage in the **Execution** bar.

The results from the unit-boundary and low-level tests in the test suite Target Speed Unit Tests increased the achieved execution coverage to 47.1%.

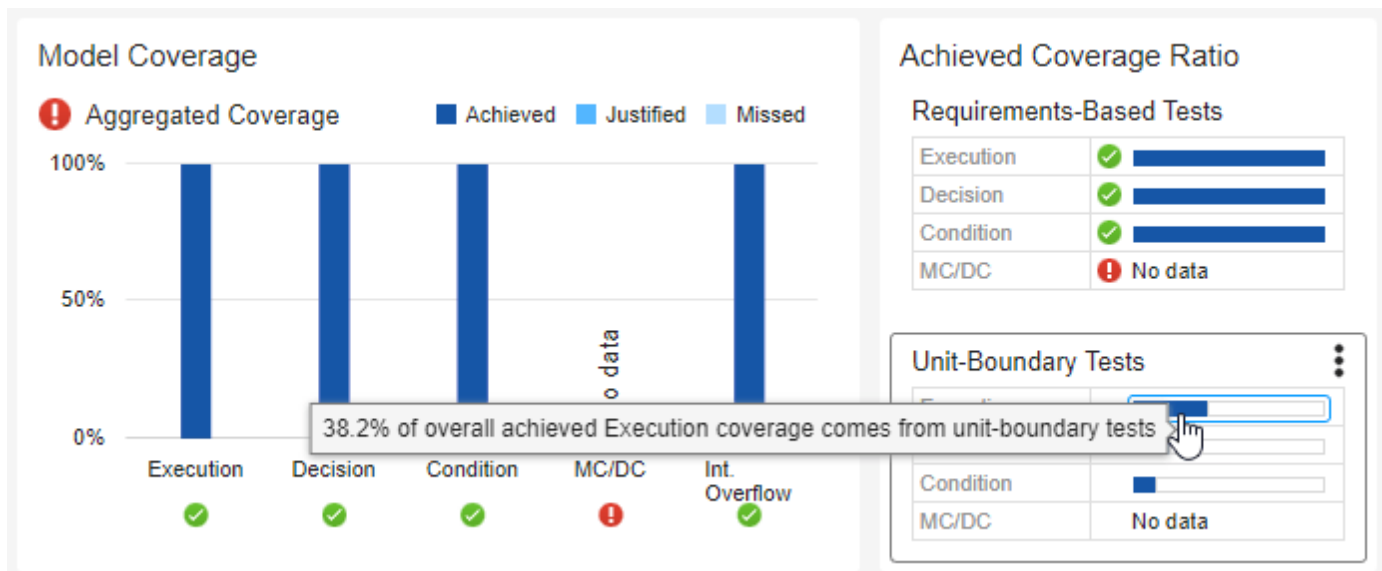


5. View the percentage of the overall achieved execution coverage that comes from unit-boundary tests. In the **Achieved Coverage Ratio** subsection, under **Unit-Boundary Tests**, point to the **Execution** bar.

Since you executed both a unit-boundary test, Idle, and low-level tests, now 81.3% of the overall achieved execution coverage comes from unit-boundary tests.



If you run the test suite Control Mode Unit Tests, the unit cc_ControlMode achieves 100% execution, decision, and condition coverage. But, for example, only 38.2% of the achieved execution coverage comes from unit-boundary tests. The remaining 61.8% of overall achieved execution coverage comes from low-level tests.



The low-level tests in the test suites Control Mode Unit Tests and Target Speed Unit Tests enable you to address the model coverage gaps. However, industry-recognized software development standards recommend using unit-boundary tests to confirm coverage completeness. If only a small percentage of your overall achieved coverage comes from unit-boundary tests, consider modifying

tests so that they test the functionality of the whole software unit and not just the low-levels like atomic subsystems.

See Also

Related Examples

- “Assess Requirements-Based Testing for ISO 26262” on page 5-102
- “Fix Requirements-Based Testing Issues” on page 5-89
- “Model Testing Metrics”

Resolve Missing Artifacts, Links, and Results

Issue

The dashboards analyze artifacts—models, requirements, tests, code, and results—that are part of the model design and testing workflow for software units. If an artifact or a link between artifacts is not part of the workflow, it might not appear in the dashboard or contribute to the analysis results. Additionally, some artifacts and links are not supported by the dashboard. If you expect a link or artifact to appear in the dashboard and it does not, try one of these solutions.

Possible Solutions

Try these solutions when you begin troubleshooting artifacts in the dashboard:

- Save changes to your artifact files.
- Check that your artifacts are saved in the project. The dashboard does not analyze files that are not saved in the project.
- If your project contains a referenced project, check that the referenced project has a unique project name. The dashboard only analyzes referenced projects that have unique project names.
- Check that your artifacts are on the MATLAB search path before you open the dashboard. When you change the MATLAB search path, the traceability information in the **Artifacts** panel is not updated. Do not change the search path while the dashboard is open.
- Open the “Artifact Issues” on page 5-100 pane and address errors or warnings. The **Artifact Issues** icon in the dashboard toolstrip changes based on the severity of the artifact issues in the project. For more information, see “View Artifact Issues in Project” on page 5-212.
- Use the dashboard to re-trace the artifacts and re-collect metric results.

Note Artifacts shown in the following folders in the **Artifacts** panel are not directly related to the unit and do not contribute to the metric results shown in the dashboard for the unit. If you expect an artifact to contribute to the metric results, check that the artifact is not in one of these folders:

- **Functional Requirements > Upstream**
 - **Tests > Others**
 - **Test Results > Others**
 - **Trace Issues**
-

Depending on the type of artifact or analysis issue that you are troubleshooting, try one of these solutions.

Enable Artifact Tracing for the Project

As you edit and save the artifacts in your project, the dashboard needs to track these changes to enable artifact tracing and to detect stale results.

By default, the dashboard requests that you enable artifact tracing the first time you open a project in the dashboard. Click **Enable and Continue** to allow the dashboard to track tool outputs to detect outdated metric results.

The dashboard needs to track tool outputs, such as test results from Simulink Test, to detect outdated metric results.

You can also enable artifact tracing from the **Startup and Shutdown** settings of the project. In the **Startup and Shutdown** settings for your project, select **Track tool outputs to detect outdated results**. For more information on the tool outputs and outdated metric results, see “Digital Thread” on page 5-167.

Cache Folder Artifact Tracking

By default, projects use the same root folder for both the simulation cache folder and the code generation folder. If possible, use different root folders for the simulation cache folder and code generation folder in your project. When you specify different root folders, the dashboard no longer needs to track changes to the simulation cache folder.

To view the cache folder settings for your project, on the Project tab, in the Environment section, click **Details**. The Project Details dialog shows the root folders specified for the **Simulation cache folder** and **Code generation folder**.

The behavior of change tracking only depends on the project settings. Custom manipulations do not impact the change tracking behavior. For example, the dashboard does not check root folders specified by `Simulink.fileGenControl`.

Project Requires Analysis by the Dashboard

The first time that you open the dashboard for the project, the dashboard identifies the artifacts in the project and collects traceability information. The dashboard must perform this first-time setup to establish the traceability data before it can monitor the artifacts. If you cancel the first-time setup, the artifacts in the project appear in the **Unanalyzed** folder in the **Artifacts** panel. To trace the unanalyzed artifacts, click **Collect > Trace Artifacts**.

Incorrect List of Models in Project Panel

The **Project** panel shows the models in your project that are either units or components. Models are organized under the components that reference them, according to the model reference hierarchy. If the list of units and components does not show the expected hierarchy of your models, try one of these solutions.

Check that your units and components are labeled

Label the units and components in your project and configure the dashboard to recognize the labeled models. Note that if a unit references one or more other models, the referenced models appear in the **Design** folder under the unit. For more information about labeling models and configuring the dashboard, see “Categorize Models in a Hierarchy as Components or Units” on page 5-119. Check that if you have Observer models, they are not labeled as units. The dashboard includes Observer models as units if they match the label requirements.

Check that your model was saved in a supported release

Check that your model was saved in R2012b or later. The dashboard does not support models that were saved before R2012b.

Block Skipped During Artifact Analysis

If a block has a mask and the mask hides the content of the block, the dashboard excludes the block from artifact analysis.

Check that your custom libraries do not contain blocks with self-modifiable masks. The dashboard does not analyze blocks that contain self-modifiable masks. Self-modifiable masks can change the structural content of a block, which is incompatible with artifact traceability analysis.

Library Missing from Artifacts Panel

Check that the library does not use a library forwarding table. The dashboard does not support library forwarding tables.

Requirement Missing from Artifacts Panel

If a requirement is missing from the **Artifacts** panel, try one of these solutions.

Check that the requirement is a functional requirement

Verify that the requirement is configured as a functional requirement. In the Requirement Editor, on the left pane, click the requirement. On the right pane, in the **Properties** section, set **Type** to **Functional**. Because the Model Testing Dashboard reports on requirements-based unit testing, only functional requirements appear in the **Artifacts** panel and are analyzed by the dashboard.

Check that the requirement is saved in a supported requirements file

Verify that the requirement is saved in a requirements file that has the `.slreqx` extension.

Test Missing from Artifacts Panel

Check that the test is supported by the dashboards. The Model Testing, SIL Code Testing, and PIL Code Testing dashboards do not support MATLAB-based Simulink tests.


Test Harness Missing from Artifacts Panel

Check that the test harness is not an internal test harness for a System Composer architecture model. The dashboard does not support internal test harnesses for System Composer architecture models. If your model already uses internal test harnesses, you can convert the internal test harnesses to externally stored test harnesses. Navigate to the top of the main model and open Simulink Test. On the **Tests** tab, click **Manage Test Harnesses > Convert to External Harnesses**. Click **OK** to convert the affected test harnesses. External test harnesses for System Composer architecture models appear in the **Artifacts** panel in the subfolder **Tests > Test Harnesses**.

Check that the test harness is not on a subsystem inside a library block instance. If a test harness is on a subsystem inside a library block inside a model, the dashboard cannot perform artifact traceability analysis on the test harness. The relationship between a model and a test harness on a subsystem inside a library block instance is incompatible with artifact traceability analysis. To enable artifact traceability analysis, move the test harness to the library.

Test Result Missing from Artifacts Panel

Check that either:

- The result is saved in a test results file. Save test results by exporting them from the Test Manager.
- You collected the results during the current project session and have not closed them. When you collect test results and do not export them, the dashboard recognizes the temporary results in the Test Manager, denoted by the  icon. The dashboard stops recognizing the temporary results when you close the project, close the test results set, or export the test results to a results file.

External MATLAB Code Missing from Artifacts Panel

The **Artifacts** panel shows external MATLAB code that the dashboard traced to the units and components in your project. If you expect external MATLAB code to appear in the dashboard and it does not, check if the construct is not supported:

A class method does not appear in the **Artifacts** panel if the method is:

- A nonstatic method that you call using dot notation. The dashboard shows the associated class definition in the **Artifacts** panel.
- A nonstatic method that you call using function notation. The dashboard shows the associated class definition in the **Artifacts** panel.
- A static method that you call from a Simulink model using dot notation. The dashboard shows the associated class definition in the **Artifacts** panel.
- A superclass method. The dashboard shows the associated superclass definition in the **Artifacts** panel.
- Defined in a separate file from the class definition file. Methods declared in separate files are not supported. For the dashboard to identify a method, you must declare a method in the class definition file. For example, if you have a class folder containing a class definition file and separate method files, the method files are not supported by the dashboard. The dashboard shows the associated class definition in the **Design** folder.

A class constructor does not appear in the **Artifacts** panel if the constructor is a superclass constructor. The dashboard shows the associated superclass definition in the **Design** folder, but not the method itself.

A class property does not appear in the **Artifacts** panel if the property is called from Simulink or Stateflow. The dashboard shows the associated class definition in the **Artifacts** panel.

An enumeration class does not appear in the **Artifacts** panel. For example, if you use an Enumerated Constant block in Simulink, the dashboard does not show the MATLAB class that defines the enum type.

Check that methods and local functions do not have the same name. If a class file contains a method and a local function that have the same name, calls that use dot notation call the method in the class definition, and calls that use function notation call the local function in the class file.

For example, if you have a class file containing the method `myAction` and the local function `myAction`, the code `obj.myAction` calls the method and the code `myAction(obj)` calls the local function.

```
classdef Class
    methods
        function myAction(~)
            % method in the class
            disp("Called method in the class.");
        end

        function myCall(obj)
            obj.myAction(); % dot notation calls the method in the class
            myAction(obj); % function notation calls the local function
        end
    end
end
```

```
function myAction(x)
    % local function
    disp("Called local function");
end
```

Artifact Returns a Warning

Check the details of the warning by clicking the **Artifact Issues** button in the toolbar.

Artifact Returns an Error

Check the details of the error by clicking the **Artifact Issues** button in the toolbar.

If the dashboard returns an error in the **Artifact Issues** tab, the metric data shown by the dashboard widgets may be incomplete. Errors indicate that the dashboard may not have been able to properly trace artifacts, analyze artifacts, or collect metrics.

Before using the metrics results shown in the dashboard, resolve the reported errors and retrace the artifacts.

Fix ambiguous links

Check that the links in your project define unambiguous relationships between project artifacts.

In requirements-based testing, projects often contain links between software requirements and:

- the design artifacts that implement the requirements
- the tests that test the implemented requirements
- the higher-level system requirements

The links in your project help to define the relationships between artifacts. The dashboard uses a digital thread to capture the traceability relationships between the artifacts in your project. To maintain the traceability relationships, the dashboard returns an error when the links to project artifacts are ambiguous. Ambiguous links are not supported in the dashboard.

If one of these conditions is met, the dashboard cannot establish unambiguous traceability:

- A link set shadows another loaded link set of the same name.
- A requirement set shadows another loaded requirement set of the same name.
- A link is not on the project path or is only temporarily on the project path.
- A link is not portable.

To avoid links that are not portable:

- Do not set the preference for a link path format to be an absolute path. Absolute paths are not portable. For information on how to set the preference for the path format of links, see `rmipref` and “Document Path Storage” (Requirements Toolbox).
- When you identify the source artifact of a link set, use the default link file name and location. Link source remapping persists in the MATLAB preferences directory and is not portable. For more information, see “Requirements Link Storage” (Requirements Toolbox).

Use the details and suggested actions in the dashboard error messages to fix the ambiguous links.

If you link a requirement to a MATLAB function, make sure you link to the first line of the function definition. For more information, see “Link Requirements to MATLAB or Plain Text Code” (Requirements Toolbox).

For more information on traceability relationships and the digital thread, see “Digital Thread” on page 5-167.

Trace Issues

If an artifact appears in the **Trace Issues** folder when you expect it to trace to a unit, depending on the type of artifact that is untraced, try one of these solutions.

Fix an untraced requirement

Check that the requirement traces to the unit using an implementation link.

The requirement must link to the model or to a library subsystem used by the model with a link where the **Type** is set to **Implements**.

Requirements-based testing verifies that your model fulfills the functional requirements that it implements. Because the Model Testing Dashboard reports on requirements-based testing quality, it analyzes only requirements that are specified as functional requirements and implemented in the unit. For each unit, the dashboard shows the functional requirements that are implemented in the unit in the folder **Functional Requirements > Implemented**.

Check that the requirement does not use an unsupported link. The Model Testing Dashboard does not trace these links:

- Downstream links. The Model Testing Dashboard traces only **Implements** links that link directly from the unit design to the requirement. Requirements that are not directly linked appear in the folder **Functional Requirements > Upstream**.
- Embedded links, which are requirements files that are saved directly in the model file.
- Links to requirements that are saved externally and linked using the Requirements Management Interface (RMI).
- Links to custom requirements that you defined by using stereotypes.
- Links inside:
 - Requirement Table blocks
 - Test Sequence blocks
 - Test Assessment blocks
- Links in deprecated requirement files, which have the extension `.req`. To analyze requirement links in the dashboard, save the links in an `.slmx` file or create them in the requirements file (`.slreqx`) that has the requirements.
- Links to models for which the model file extension changed. If a requirement is linked to a model with the file extension `.slx`, but the model file extension is changed to `.mdl`, the dashboard lists the requirement link as unresolved. Modify the requirement link to reference the expected model file and re-save the requirement link.
- Symbolic file links in a project, such as shortcuts.
- Links to modeling elements that are not supported by the Model Testing Dashboard, such as library forwarding tables.

- Custom link types that you defined by using stereotypes.
- A requirement link to a justification. If a requirement is linked with a justification and not linked to a test, it appears as unlinked in the metric results.
- A requirement link to a test harness.

For requirement links to data dictionary entries, the dashboard traces from the requirement to the data dictionary file associated with the data dictionary entry.

If a requirement links to a range of MATLAB code that contains multiple code constructs, the dashboard resolves the link to the first code construct that appears in the range. For example, if the linked line ranges contains MATLAB code for two functions, the dashboard generates a warning and resolves the requirement link to the first function.

Fix an untraced requirement link set

Check that the requirement link set does not use the legacy Requirements Management Interface (RMI) format. To allow the dashboard to analyze your requirement link set, pass your requirement link set as the input argument to the function `slreq.refreshSourceArtifactPath`.

Fix an untraced design artifact

Check that the design artifact does not rely on a model callback to be linked with the model. The dashboards do not execute model loading callbacks when they load the models for analysis. If a model relies on a callback to link a data dictionary, the data dictionary will not be linked when the dashboards run the traceability analysis.

Fix an untraced test

Check that the test runs on the model or runs on an atomic subsystem in the model by using a test harness.

Fix an untraced test result

Check that the project and test are set up correctly and re-run your tests. If one of these conditions is met when you run your test, the generated results are untraced because the dashboard cannot establish unambiguous traceability to the unit:

- No project is loaded.
- Artifact tracing is not enabled for the project. If artifact tracing is not enabled, the dashboard cannot track changes or trace from the tests to the generated test results. For more information, see “Enable Artifact Tracing for the Project” on page 5-128.
- You do not have a Simulink Check license.
- The test file is stored outside the project.
- The test file has unsaved changes.
- The tested model has unsaved changes.
- The test file returns an error during traceability analysis.
- The tested model returns an error during traceability analysis.
- The test result comes from a test that is not supported by the dashboards, such as a MATLAB-based Simulink test.

Check that the results and environment are set up correctly and re-export your test results. If one of these conditions is met when you export your test results, the generated results are untraced because the dashboard cannot establish unambiguous traceability to the unit:

- No project is loaded.
- Artifact tracing is not enabled for the project. For more information, see “Enable Artifact Tracing for the Project” on page 5-128.
- You do not have a Simulink Check license.
- The test result file returns an error during traceability analysis.

Metric Does Not Report Results for Requirement, Test, or Test Result

If an artifact traces to one of your units but does not appear in the metric results for that unit, depending on the type of artifact, try one of these solutions.

Fix a requirement that does not produce metric results

Check that the requirement directly links to the model with a link where the **Type** is set to **Implements**. The dashboard metrics analyze only implemented functional requirements. For each unit, the implemented functional requirements appear in the folder **Functional Requirements > Implemented**. Upstream requirements appear in the folder **Functional Requirements > Upstream**, but do not contribute to the metric results because upstream requirements are only indirectly or transitively linked to the implemented requirements.

Fix a test that does not produce metric results

Check that the test directly tests either the entire unit or atomic subsystems in the model. The dashboard metrics analyze only unit tests. For each unit, the unit tests appear in the folder **Tests > Unit Tests**. Other tests, that are not unit tests, appear in the folder **Tests > Others**, but do not contribute to the metric results because other tests do not directly test the unit or atomic model subsystems. For example, the dashboard does not consider tests on a library or tests on a virtual subsystem to be unit tests.

Fix a test result that does not produce metric results

Check that the results meet these criteria:

- The results are the most recent results generated from the tests.
- The results are from unit tests which appear in the folder **Tests > Unit Tests**. Tests in the folder **Tests > Others** do not contribute to the metric results.

The dashboard can only isolate outdated results to individual test cases or test suites if the test cases or test suites have revision numbers. If a test case or test suite was saved in a release that did not save revision numbers, use the function `sltest.testmanager.refreshTestRevisions` on the test file to refresh the revision information.

The coverage metrics do not aggregate coverage for external C code, such as S-functions and C Caller blocks, and the code coverage metrics do not include coverage results for shared code files.

For each unit, the test results that produce metric results appear in the **Test Results** folder in the subfolders **Model**, **SIL**, and **PIL**. The test results in the folder **Test Results > Others** do not contribute to the metric results.

Fix a test that does not produce simulation test result analysis metric results

Check that the test is a unit test and produces simulation results. For each unit, the metrics analyze the test results in the **Test Results > Model**. The test results in the folder **Test Results > Others** are results that are not model, software-in-the-loop (SIL), or processor-in-the-loop (PIL) results, are not from unit tests, or are only reports. The metrics in the **Simulation Test Result Analysis** section

count results from only simulation tests, whereas the metrics in the **Test Analysis** section count all unit tests.

If a test is not counted in the metrics in the **Simulation Test Result Analysis** section, check that the test meets these criteria for being a simulation test:

- The simulation mode is Normal, Accelerator, or Rapid Accelerator. If the test uses iterations to set a different simulation mode after testing one of these modes, the test is still considered a simulation test.
- The test is not a real-time test.
- If the test is an equivalence test, the first simulation meets one of the first two criteria.
- If the test contains multiple iterations, the test or at least one iteration meets one of the first two criteria.

Metric Results Show Unexpected Model or Code Coverage

Note that the model coverage metrics do not scope coverage to requirements. If you select the **Scope coverage results to linked requirements** check box in your test results, the dashboard ignores that selection and does not scope the model coverage metrics results that appear in the dashboard. For information on the **Scope coverage results to linked requirements** option, see “Scoping Coverage for Requirements-Based Tests” (Simulink Test).

Fix inconsistent model and code coverage from inlined external MATLAB functions

By default, the coverage metrics include external MATLAB function coverage in the overall unit coverage.

If you have external MATLAB functions in your project, either:

- Place the `coder.inline('never')` directive inside the function and use a project label to categorize the M file as a unit
- Place the `coder.inline('always')` directive inside the function, but do not use a project label to categorize the M file as a unit

For information on the `coder.inline` directive, see `coder.inline`. If possible, avoid using `coder.inline('default')`. `coder.inline('default')` uses internal heuristics to determine whether to inline the function, which can produce inconsistent coverage metric results in the dashboard.

Typically, you use a project label to categorize a *model* as a unit or component in the dashboard. When you add your unit label to an *external MATLAB function*, the function does not appear in the **Project** panel, but the dashboard is able to exclude the function coverage from the overall unit coverage. For information on how to use project labels to categorize units and components, see “Categorize Models in a Hierarchy as Components or Units” on page 5-119.

Metric Result Shows a Missing Link or Artifact

The metric results do not count all types of traceability links. If a metric shows that a test or requirement is missing links when you expect it to be linked, try one of these solutions.

Fix missing model coverage in test results

If the model coverage metrics report coverage gaps that you do not expect, re-run the tests and re-collect the metric results for the new test results. The Model Testing Dashboard might show coverage gaps if:

- You change the test results file or the coverage filter file after you collect the metrics, including if you re-import the test results file after you make changes.
- You collect accumulated coverage results and make changes to the model file after running one or more tests.

See Also

“Model Testing Metrics”

Related Examples

- “Fix Requirements-Based Testing Issues” on page 5-89
- “Manage Project Artifacts for Analysis in Dashboard” on page 5-95

Collect Requirements-Based Testing Metrics Using Continuous Integration

Overview

Requirements-based testing metrics allow you to assess the status and quality of your requirements-based testing activities. You can visualize the results by using the Model Testing Dashboard and integrate metric collection by using continuous integration workflows. Continuous collection of these metrics helps you to monitor the progression and quality of a project. This example uses GitLab® to host the project source and Jenkins® to build and test the project as well as archive the results.

Requirements

- Use of MATLAB® projects
- Use of the Simulink Test Manager test harness

Set Up the Project in Source Control

GitLab Setup

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

- 1 Install the Git Client.
- 2 Set up a branching workflow. Using GitLab, from the main branch, create a temporary branch for implementing changes to the model files. Integration engineers can use Jenkins test results to decide whether to merge a temporary branch into the main branch. For more information, see <https://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows>.
- 3 Under **Settings** > **Repository**, protect the main branch by enforcing the use of merge requests when developers want to merge their changes into the main branch.
- 4 Under **Settings** > **Integrations**, add a webhook to the URL of your Jenkins project. The webhook triggers a build job on Jenkins.

Add the Project

This example uses the example project for the dashboard. To create a working copy of the project, at the command line, enter:

```
dashboardCCProjectStart('incomplete')
```

Add all of the files in the project along with the files attached to this example to the main branch by using Git™. These scripts are used to run tests and collect the metrics.

Derived Artifact Filtering

Collecting metrics generates files that you typically do not want checked into source control. Git allows you to ignore files by adding filters to a text file named `.gitignore` located at the root directory. You can add the sample `.gitignore` file attached to this example which will filter the files generated by this example that do not need to be added to source control. For more information on `.gitignore` files, see <https://git-scm.com/docs/gitignore>.

Set Up the Project in the Continuous Integration Tool

The continuous integration tool automates the building and testing of the project. Many different tools can be used to automatically generate requirements-based testing results by following the same general steps. In this example, use Jenkins as the automation tool. To run the example, you must install the GitLab and MATLAB plugins for Jenkins.

Creating the Project

The CI tool will need integration to the source control repository of the project. The integration allows the CI tool to listen to changes and access the project to build. Jenkins provides a Freestyle project which serves as a generic template for projects which can work with any source control management (SCM). In the Freestyle project, add the source control information to enable the SCM to access the hosted project.

- 1 Click **New Item**, fill in the name, and choose Freestyle project. Or, for an existing Freestyle Jenkins project, click **Configure**.
- 2 Click the **Source Code Management** tab and specify the URL of your GitLab repository in the **Repository URL** field.
- 3 Click the **Build Triggers** tab and select **Build when a change is pushed to GitLab**.
- 4 Click the **Build Environment** tab, select **Use MATLAB version**, and provide the **MATLAB root** to specify the MATLAB version for the build.

Building and Testing

The MATLAB plugin for Jenkins allows integration by specifying MATLAB commands as well as configuring testing, bypassing the need to use the command line. In this example, a single build step will be used to open the project, initialize the metrics infrastructure, run tests, and then collect results.

In the **Build** section, select **Add build step > Run MATLAB Command**. In the **Command** field, enter:

```
openProject(pwd);collectModelTestingResults();runTests();collectModelTestingResults();
```

Archiving and Consuming Metric Results

Metrics results can be archived during the build step and then reimported into MATLAB when you want to review them. In this example, the result collection script stores the metric data in the **derived** directory. Because some of the metrics rely on exported Simulink Test results, include the exported **.mldatx** files in the archive.

To archive results for later review, configure the CI system to export these files:

- All files located in `<project-root>/derived` directory.
- All test results exported to `<project-root>/testresults/.mldatx` files.

For this example, use the Jenkins provided post-build action to archive artifacts produced during the build.

Click the **Post-build Actions** tab and click **Add post-build action**. Choose **Archive the artifacts**. Enter the path:

```
derived/**,testresults/*.mldatx
```

to archive all files saved to that directory.

Click the **Save** button to save and close the configuration.

Running a Build Job in Jenkins

Jenkins is now configured to execute a new build job each time new changes to the project have been committed to the GitLab repository. You can also manually run a build by clicking on **Build Now** in the Jenkins project page.

Reviewing the Archived Results in MATLAB

Jenkins will store all the files generated and archived for each successful build and these can be viewed individually or downloaded together in a single zip file. To view the results in MATLAB:

- 1** Get the version of the project that was used to generate the results from source control.
- 2** Get the archived metric results from the archived location.
- 3** Download and copy or extract the `derived` directory and all files into the root directory of the project.
- 4** Download the archived, exported, Simulink Test results files and copy or extract these files.
- 5** Open the project in MATLAB and open the Model Testing Dashboard. The dashboard displays the results generated from the CI build.

Alternative CI Integration Using Command Line

If you use a different automation tool, you can alternatively use the command line for testing integration. Run the tests and collect metrics by running the appropriate commands through the command line interface along using the `-batch` flag.

For example, when you use this command, MATLAB opens the project, initializes the model testing results, runs all of the tests, collects the model metrics, and then shuts down.

```
matlab -c %LICENSE_PATH% -nosplash -logfile output.log -batch  
"openProject(pwd);collectModelTestingResults();runTests();collectModelTestingResults(); exit;"
```

Hide Requirements Metrics in the Model Testing Dashboard and in API Results

If you do not want to track requirements metrics in the Model Testing Dashboard, or cannot because of a missing Requirements Toolbox license, you can hide the requirements metrics. When you hide the requirements metrics, the dashboard displays only the metrics in the **Test Case Breakdown**, **Model Test Status**, and **Model Coverage** sections. This example shows how to hide the requirements metrics when you use the Model Testing Dashboard or when you programmatically collect metrics.

If you want to collect requirements-based metrics, see “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80.

Open the Dashboard for the Project

- 1 Open a project that contains models and testing artifacts. For this example, at the MATLAB command line, enter:

```
dashboardCCProjectStart("incomplete")
```

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

- On the **Project** tab, click **Model Testing Dashboard**.
- At the command line, enter:

```
modelTestingDashboard
```

Hide Requirements Metrics

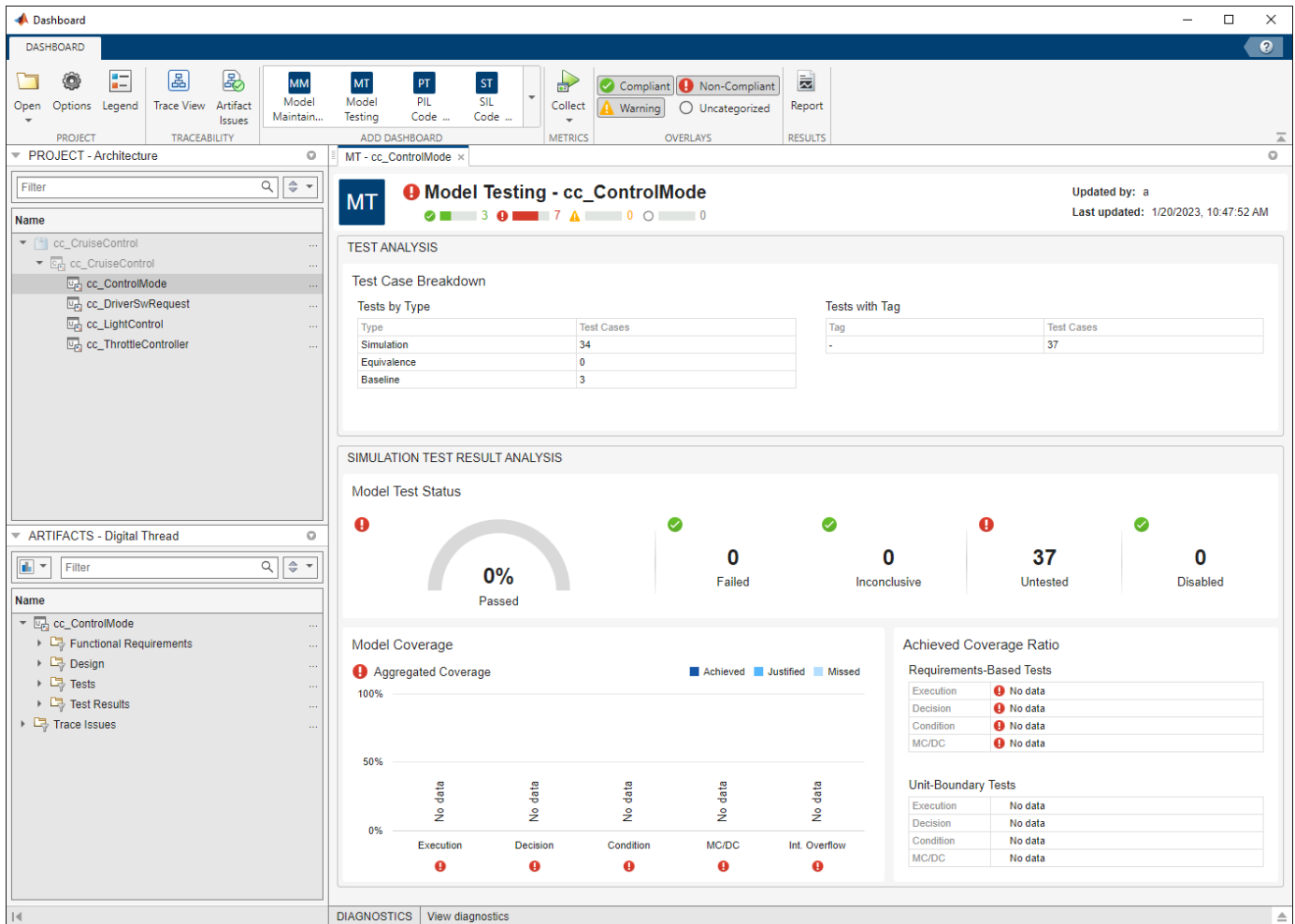
- 1 Open the Project Options dialog box by clicking **Options** in the toolbar.
- 2 In the **Layout** section, select **Hide requirements metrics** and click **Apply**.

The app closes any open dashboards.

Note Because the setting of the **Hide requirements metrics** property is saved in the project data, the dashboard displays or hides requirements-based metrics for any users who view the project.

- 3 In the **Add Dashboard** section of the toolbar, click **Model Testing** to re-open the Model Testing Dashboard.

The dashboard shows only the widgets for the **Test Case Breakdown**, **Model Test Status**, and **Model Coverage** sections.



View API Results with Requirements Metrics Hidden

When you select the **Hide requirements metrics** in the Project Options for the dashboard, the dashboard also hides requirements metrics from the API results.

- 1 Create a `metric.Engine` object for the current project.

```
metric_engine = metric.Engine();
```

- 2 Create a list of the available metric identifiers for the **Model Testing Dashboard**. Use the function `getAvailableMetricIds` and specify the 'App' as 'DashboardApp' and the 'Dashboard' as 'ModelUnitTesting'.

```
nonRequirementMetrics = getAvailableMetricIds(metric_engine, ...
'App', 'DashboardApp', ...
'Dashboard', 'ModelUnitTesting');
```

Since the setting **Hide requirements metrics** is selected, the API results do not include requirements-based metrics like the metric `RequirementsPerTestCase`. The list of metric identifiers still includes metrics like `RequirementsConditionCoverageBreakdown` because that metric only requires a Simulink Coverage license.

- 3 Use `nonRequirementsMetrics` to collect metric results from metric identifiers that are not associated with requirements metrics.

```
execute(metric_engine, nonRequirementMetrics);
```

- 4 Get the metric results that are not associated with requirements metrics.

```
results = getMetrics(metric_engine, nonRequirementMetrics)
```

For more information about the metric API, see “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115.

See Also

`metric.Engine` | `execute` | `getAvailableMetricIds` | `getMetrics`

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115
- “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80
- “Fix Requirements-Based Testing Issues” on page 5-89
- “Model Testing Metrics”

Monitor the Complexity of Your Design Using the Model Maintainability Dashboard

The Model Maintainability Dashboard collects metric data from the model design artifacts in a project to help you assess the size, architecture, and complexity of your design.

The dashboard analyzes different aspects of model maintainability from model design artifacts like Simulink® models, Stateflow® charts, and MATLAB® code. The model maintainability metrics help you determine if parts of a design are too complex and need to be refactored. A less complex design is easier to read, maintain, and test.

This example shows how to collect and explore the maintainability metric data for a project. As you update and development your design artifacts, use the dashboard to assess the impact on complexity and maintainability.

Open the Model Maintainability Dashboard for a Project

1. Open a project that contains design artifacts. For this example, in the MATLAB Command Window, enter:

```
dashboardCCProjectStart("incomplete")
```

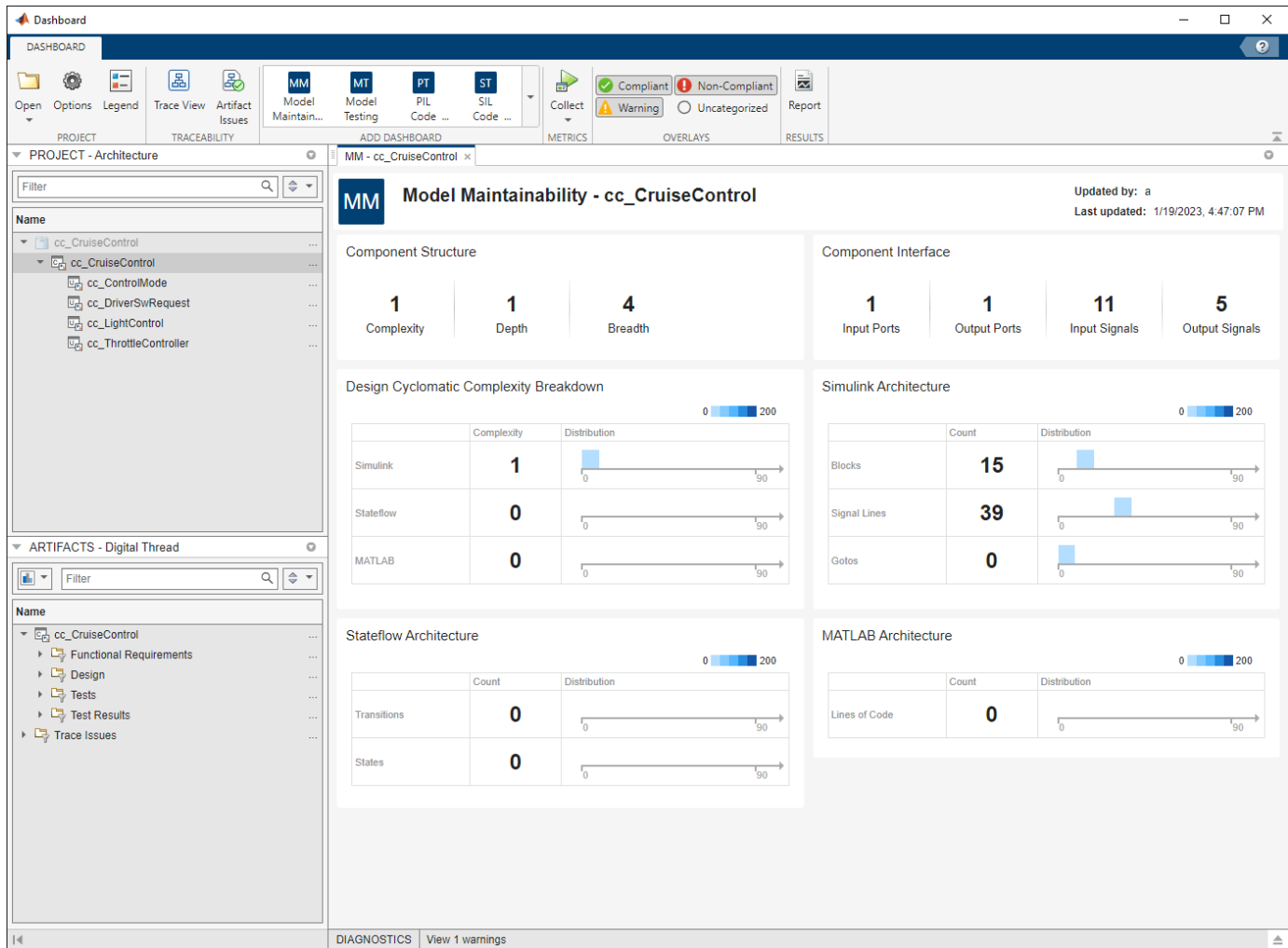
The `dashboardCCProjectStart` command creates a new instance of an example project for a cruise control system. The project includes several models, tests, and requirements files.

2. The Model Maintainability Dashboard is a model design dashboard that displays maintainability metrics. To open the Model Maintainability Dashboard, use one of these approaches:



- On the **Project** tab, click **Model Design Dashboard**.
- Open a model in the project and, in the **Apps** gallery, click **Model Design Dashboard**.
- In the Command Window, enter:

```
modelDesignDashboard
```

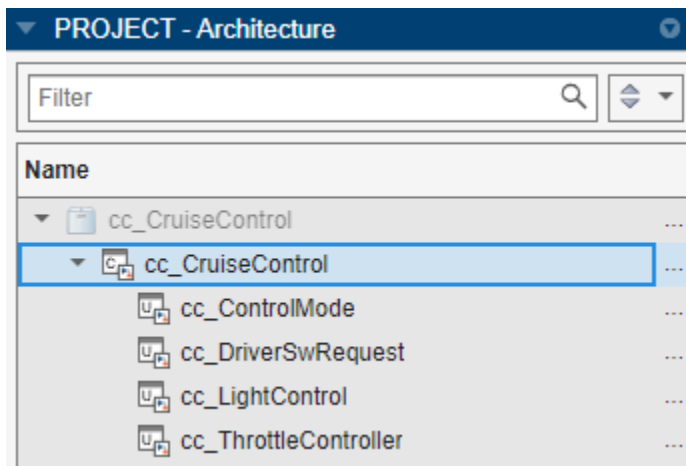
The dashboard app launches and opens a new **Model Maintainability** tab for the software component `cc_CruiseControl`.



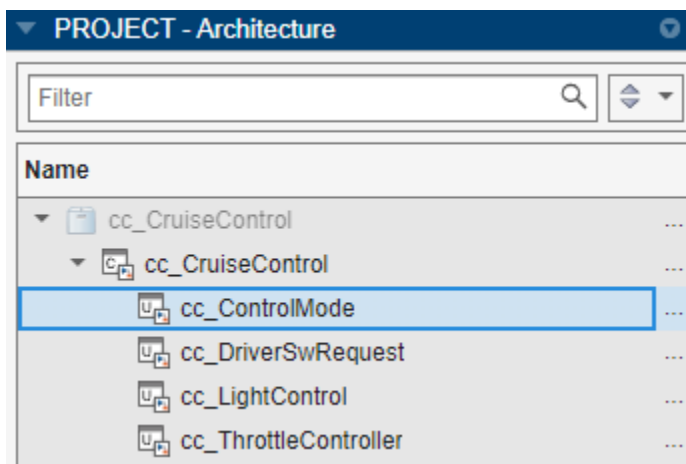
Explore the Project Architecture

On the left side of the dashboard app is the **Project** panel which displays the architecture of the units and components in the current project. Software components use the Component icon  and software units use the Unit icon .

3. In the **Project** panel, expand the component **cc_CruiseControl**. The component `cc_CruiseControl` contains 4 software units: `cc_ControlMode`, `cc_DriverSwRequest`, `cc_LightControl`, and `cc_ThrottleController`.



4. Select the unit **cc_ControlMode** to open a Model Maintainability Dashboard for that software unit. If a dashboard is not available for a specific unit or component, the name of the unit or component appears dimmed.



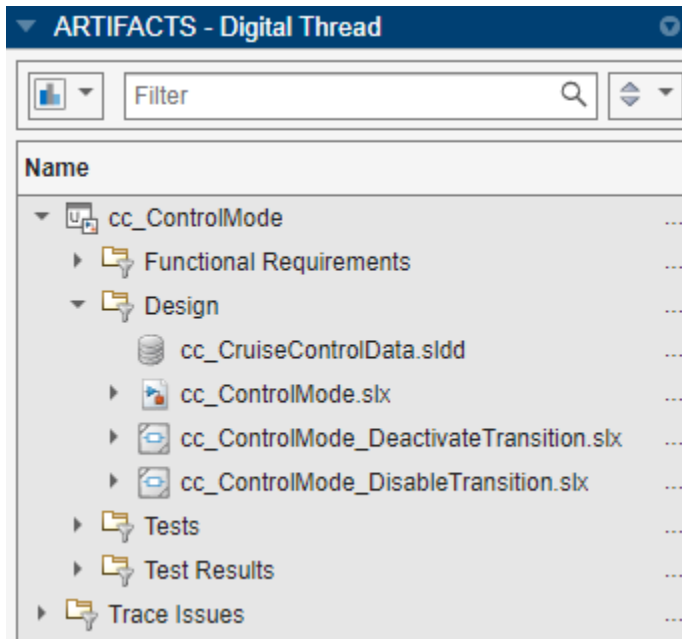
Use the **Project** panel to open the dashboard for the different units and components in your project architecture.

Explore the Design Artifacts

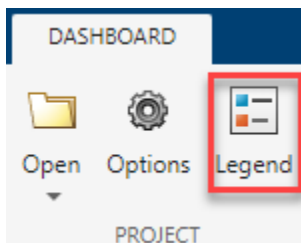
On the bottom left of the dashboard app is the Artifacts panel which displays the artifacts inside the selected unit or component. The **Functional Requirements**, **Design**, **Tests**, and **Test Results** folders contain the artifacts the dashboard traced to the current unit or component.

For example, when **cc_ControlMode** is selected in the **Project** panel, the **Artifacts** panel displays the artifacts associated with the unit **cc_ControlMode**.

5. In the **Artifacts** panel, expand the **Design** folder. For the unit **cc_ControlMode**, the **Design** folder shows the associated data dictionary, model, and subsystem references.



In the toolbar for the dashboard app, you can click the **Legend** button to view a list of the icons for artifacts that the dashboard traces.



Explore the Maintainability Metrics

When you open a **Model Maintainability Dashboard**, the dashboard automatically collects the metric data and displays the results in the six main sections of the dashboard:


- **Component Structure**
- **Component Interface**
- **Design Cyclomatic Complexity Breakdown**
- **Simulink Architecture**
- **Stateflow Architecture**
- **MATLAB Architecture**

Each section of the dashboard contains widgets that help you interact with the metric data.

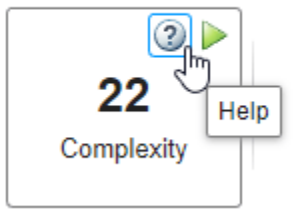
6. In the **Component Structure** section of the dashboard, point to the **Complexity** widget. Three dots appears in the top-right corner of the widget.

Component Structure



Point to the three dots and click the Help icon  to view more information about the metric and how the dashboard calculates the metric value.

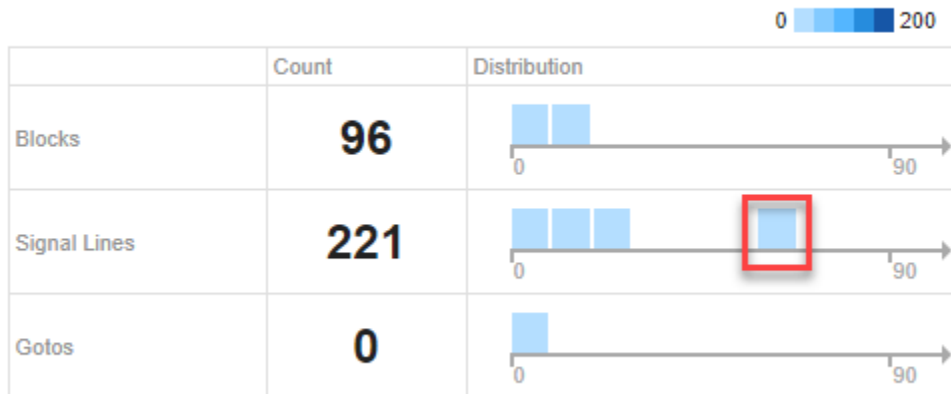
Component Structure



7. In the **Simulink Architecture** section, in the **Signal Lines** row and **Distribution** column, the metric results show an outlier in the signal line distribution.

The dashboard sorted most of the metric data into the three distribution bins on the left. The rightmost distribution bin contains metric data that is outside of that range.

Simulink Architecture

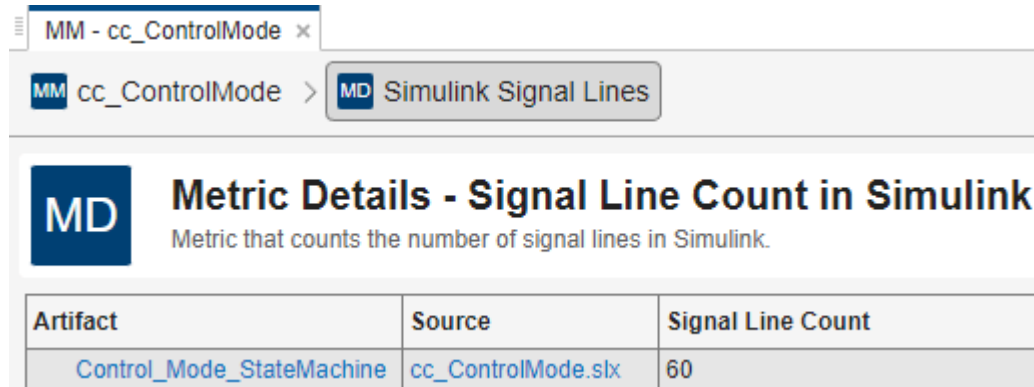


8. Point to the rightmost distribution bin. The tooltip indicates that there is one layer in the model hierarchy containing between 60 and 69 Simulink signals.

The other layers of the model contain fewer signal lines. For example, the leftmost distribution bin shows nine model layers containing fewer than 10 Simulink signals.

9. Click the rightmost distribution bin to explore the metric data in more detail.

The dashboard opens the **Metric Details** for the widget with a table of metric values and hyperlinks to each related artifact. The table shows that the artifact **Control_Mode_StateMachine** contains 60 signal lines.



The screenshot shows a breadcrumb trail: MM - cc_ControlMode > MD Simulink Signal Lines. Below this is a section titled "Metric Details - Signal Line Count in Simulink" with a sub-description: "Metric that counts the number of signal lines in Simulink." A table follows with the following data:

Artifact	Source	Signal Line Count
Control_Mode_StateMachine	cc_ControlMode.slx	60

Note that there is a breadcrumb trail from the **Metric Details** back to the main **Model Maintainability** results for **cc_ControlMode**.



The screenshot shows a breadcrumb trail: MM cc_ControlMode > MD Simulink Signal Lines.

10. In the **Artifact** column, click on **Control_Mode_StateMachine** to open the artifact in Simulink.

In this example, the artifact `Control_Mode_StateMachine` is a subsystem in the model `cc_ControlMode`. If the 60 signal lines at the top layer of the subsystem make the design hard to read, refactor the model layer to improve readability.

See Also

"Model Maintainability Metrics"

Related Examples

- "Assess Model Size and Complexity for ISO 26262" on page 5-158
- "Collect Model Maintainability Metrics Programmatically" on page 5-150

Collect Model Maintainability Metrics Programmatically

This example shows how to collect the metric data used in the Model Maintainability Dashboard, generate a report, and use the report to assess the model design artifacts.

If a model in your project is large, complex, or difficult to read, the design can be difficult to test and maintain. Use the metric results to determine if you want to refactor your design to identify smaller testable units, or restructure a model to improve readability.

Collect and Access Metric Results

Open Project

Open a project containing models that you want to analyze. For this example, in the MATLAB® Command Window, enter:

```
dashboardCCProjectStart
```

Collect Metric Results

Create a `metric.Engine` object. You can use the `metric.Engine` object to collect the metric data for the current project.

```
metric_engine = metric.Engine();
```

Create an array of the metric identifiers for model maintainability.

```
maintainabilityMetrics = getAvailableMetricIds(metric_engine,...
App="DashboardApp",...
Dashboard="ModelMaintainability");
```

You can collect results for one unit at a time or for each unit in the project.

For this example, suppose you want to collect results for the unit `db_ControlMode`. Create an array that identifies the path to the model file in the project and the name of the model.

```
unit = [fullfile(pwd,"models","db_ControlMode.slx"),"db_ControlMode"];
```

Use the `execute` function to collect the maintainability metric results for the unit `db_ControlMode`.

```
execute(metric_engine,maintainabilityMetrics,ArtifactScope=unit);
```

Access and View Metric Results

Access the metric results by using the `getMetrics` function. For this example, store the results for the metric `slcomp.SimulinkSignalLines` and for the unit `db_ControlMode`.

```
results_SignalLines = getMetrics(metric_engine,"slcomp.SimulinkSignalLines",...
ArtifactScope=unit);
```

The model maintainability metric `slcomp.SimulinkSignalLines` determines the number of Simulink® signal lines in each layer of a model. For more information on the model maintainability metrics, see “Model Maintainability Metrics”.

The `getMetrics` function returns a `metric.Result` object containing the metric results for the specified metric.

Use the `disp` function to display the model layer and number of signal lines, which are in the `Artifacts` and `Value` properties of the `metric.Result` object. The results may appear in a different order on your machine.

```
for n=1:length(results_SignalLines)
    disp([' Model Layer: ', results_SignalLines(n).Artifacts.Name])
    disp(['Signal Lines: ', num2str(results_SignalLines(n).Value)])
    disp(newline);
end
```

`end`

```
Model Layer: Switch Case Action
Subsystem1
```

```
Signal Lines: 1
```

```
Model Layer: Switch Case Action
Subsystem
```

```
Signal Lines: 1
```

```
Model Layer: Control_Mode_StateMachine
```

```
Signal Lines: 60
```

```
Model Layer: Transitions_From_ACTIVE
```

```
Signal Lines: 14
```

```
Model Layer: check_speed_range
```

```
Signal Lines: 6
```

```
Model Layer: Transitions_From_DISABLED
```

```
Signal Lines: 11
```

```
Model Layer: Switch Case Action
Subsystem2
```

```
Signal Lines: 1
```

```
Model Layer: Switch Case Action
Subsystem2
```

```
Signal Lines: 1
```

```
Model Layer: wasActiveOnce
```

```
Signal Lines: 12
```

```
Model Layer: Transitions_From_THROTTLE_OVERRIDE
```

```
Signal Lines: 14
```

```
Model Layer: db_ControlMode
```

```
Signal Lines: 14
```

```
Model Layer: Target_Speed_Calculator
```

Signal Lines: 18

Model Layer: Switch Case Action
Subsystem1

Signal Lines: 1

Model Layer: Switch Case Action
Subsystem

Signal Lines: 23

Model Layer: isFirstStep

Signal Lines: 2

Model Layer: Switch Case Action
Subsystem4

Signal Lines: 1

Model Layer: Switch Case Action
Subsystem3

Signal Lines: 1

Model Layer: Transitions_From_ENABLED

Signal Lines: 20

Model Layer: db_ControlMode_DisableTransition

Signal Lines: 10

Model Layer: db_ControlMode_DeactivateTransition

Signal Lines: 10

The code output shows that the model layer `Control_Mode_StateMachine` contains 60 Simulink signal lines. Based on these results, consider restructuring that model layer to improve readability.

Generate Report

Generate a report file containing the model maintainability results collected in `metric_engine`. By default, the report file type is a PDF. For this example, specify the report file type as HTML.

```
generateReport(metric_engine, ...  
App="DashboardApp", ...  
Dashboard="ModelMaintainability", ...  
Type="html-file");
```

Save the metric results in a report file to access them without opening the project or dashboard.

Alternatively, you can open the Model Maintainability Dashboard to see the results and explore the artifacts. To programmatically access the Model Maintainability Dashboard, enter:

```
modelDesignDashboard
```

See Also

“Model Maintainability Metrics”

Related Examples

- “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard” on page 5-144

Resolve Missing Artifacts and Results in the Model Maintainability Dashboard

Issue

The Model Maintainability Dashboard analyzes artifacts—Simulink models, Stateflow charts, and MATLAB code—that are part of the model design workflow for software units and components. If an artifact or a relationship between artifacts is not supported, it might not appear in the Model Maintainability Dashboard or contribute to the analysis results. If you expect a link or artifact to appear in the dashboard and it does not, try one of these solutions.

Possible Solutions

Try these solutions when you begin troubleshooting artifacts in the Model Maintainability Dashboard:

- Save changes to your artifact files.
- Check that your artifacts are saved in the project. The Model Maintainability Dashboard does not analyze files that are not saved in the project.
- If your project contains a referenced project, check that the referenced project has a unique project name. The dashboard only analyzes referenced projects that have unique project names.
- Check that your artifacts are on the MATLAB search path before you open the dashboard. When you change the MATLAB search path, the traceability information in the **Artifacts** panel is not updated. Do not change the search path while the dashboard is open.
- Open the “Artifact Issues” on page 5-100 pane and address errors or warnings.
- Use the dashboard to re-trace the artifacts and re-collect metric results.

Depending on the type of artifact or analysis issue that you are troubleshooting, try one of these solutions.

Enable Artifact Tracing for the Project

As you edit and save the artifacts in your project, the dashboard needs to track these changes to enable artifact tracing and to detect stale results.

By default, the Model Maintainability Dashboard requests that you enable artifact tracing the first time you open a project in the dashboard. Click **Enable and Continue** to allow the Model Maintainability Dashboard to track tool outputs to detect outdated metric results.

The dashboard needs to track tool outputs to detect outdated metric results.

You can also enable artifact tracing from the **Startup and Shutdown** settings of the project. In the **Startup and Shutdown** settings for your project, select **Track tool outputs to detect outdated results**. For more information on the tool outputs and outdated metric results, see “Digital Thread” on page 5-167.

Cache Folder Artifact Tracking

By default, projects use the same root folder for both the simulation cache folder and the code generation folder. If possible, use different root folders for the simulation cache folder and code generation folder in your project. When you specify different root folders, the dashboard no longer needs to track changes to the simulation cache folder.

To view the cache folder settings for your project, on the Project tab, in the Environment section, click **Details**. The Project Details dialog shows the root folders specified for the **Simulation cache folder** and **Code generation folder**.

The behavior of change tracking only depends on the project settings. Custom manipulations do not impact the change tracking behavior. For example, the dashboard does not check root folders specified by `Simulink.fileGenControl`.

Project Requires Analysis by the Dashboard

The first time that you open the dashboard for the project, the dashboard identifies the artifacts in the project and collects traceability information. The dashboard must perform this first-time setup to establish the traceability data before it can monitor the artifacts. If you cancel the first-time setup, the artifacts in the project appear in the **Unanalyzed** folder in the **Artifacts** panel. To trace the unanalyzed artifacts, click **Collect > Trace Artifacts**.

Incorrect List of Models in Project Panel

The **Project** panel shows the models in your project that are either units or components. Models are organized under the components that reference them, according to the model reference hierarchy. If the list of units and components does not show the expected hierarchy of your models, try one of these solutions.

Check that your units and components are labeled

Label the software units and components in your project and configure the Model Maintainability Dashboard to recognize the labeled models. Note that if a unit references one or more other models, the referenced models appear in the **Design** folder under the unit. For more information about labeling models and configuring the dashboard, see “Categorize Models in a Hierarchy as Components or Units” on page 5-119. Check that if you have Observer models, they are not labeled as units. The dashboard includes Observer models as units if they match the label requirements.

Check that your model was saved in a supported release

Check that your model was saved in R2012b or later. The Model Maintainability Dashboard does not support models that were saved before R2012b.

External MATLAB Code Missing from Artifacts Panel

The **Artifacts** panel shows external MATLAB code that the dashboard traced to the units and components in your project. If you expect external MATLAB code to appear in the dashboard and it does not, check if the construct is not supported:

A class method does not appear in the **Artifacts** panel if the method is:

- A nonstatic method that you call using dot notation. The dashboard shows the associated class definition in the **Artifacts** panel.
- A nonstatic method that you call using function notation. The dashboard shows the associated class definition in the **Artifacts** panel.
- A static method that you call from a Simulink model using dot notation. The dashboard shows the associated class definition in the **Artifacts** panel.
- A superclass method. The dashboard shows the associated superclass definition in the **Artifacts** panel.

- Defined in a separate file from the class definition file. Methods declared in separate files are not supported. For the dashboard to identify a method, you must declare a method in the class definition file. For example, if you have a class folder containing a class definition file and separate method files, the method files are not supported by the dashboard. The dashboard shows the associated class definition in the **Design** folder.

A class constructor does not appear in the **Artifacts** panel if the constructor is a superclass constructor. The dashboard shows the associated superclass definition in the **Design** folder, but not the method itself.

A class property does not appear in the **Artifacts** panel if the property is called from Simulink or Stateflow. The dashboard shows the associated class definition in the **Artifacts** panel.

An enumeration class does not appear in the **Artifacts** panel. For example, if you use an Enumerated Constant block in Simulink, the dashboard does not show the MATLAB class that defines the enum type.

Check that methods and local functions do not have the same name. If a class file contains a method and a local function that have the same name, calls that use dot notation call the method in the class definition, and calls that use function notation call the local function in the class file.

For example, if you have a class file containing the method `myAction` and the local function `myAction`, the code `obj.myAction` calls the method and the code `myAction(obj)` calls the local function.

```
classdef Class
    methods
        function myAction(~)
            % method in the class
            disp("Called method in the class.");
        end

        function myCall(obj)
            obj.myAction(); % dot notation calls the method in the class
            myAction(obj); % function notation calls the local function
        end
    end
end

function myAction(x)
    % local function
    disp("Called local function");
end
```

Block Skipped During Artifact Analysis

If a block has a mask and the mask hides the content of the block, the dashboard excludes the block from artifact analysis.

Check that your custom libraries do not contain blocks with self-modifiable masks. The Model Maintainability Dashboard does not analyze blocks that contain self-modifiable masks. Self-modifiable masks can change the structural content of a block, which is incompatible with artifact traceability analysis.

Library Missing from Artifacts Panel

Check that the library does not use a library forwarding table. The Model Maintainability Dashboard does not support library forwarding tables.

Artifact Returns a Warning

Check the details of the warning by clicking the **Artifact Issues** tab in the toolbar.

For more information, see “View Artifact Issues in Project” on page 5-212.

Artifact Returns an Error

Check the details of the error by clicking the **Artifact Issues** tab in the toolbar.

If the dashboard returns an error in the **Artifact Issues** tab, the metric data shown by the dashboard widgets may be incomplete. Errors indicates that the dashboard may not have been able to properly trace artifacts, analyze artifacts, or collect metrics.

Before using the metrics results shown in the dashboard, resolve any reported errors and retrace the artifacts.

For more information, see “View Artifact Issues in Project” on page 5-212.

Trace Issues

If an artifact appears in the **Trace Issues** folder when you expect it to trace to a unit or component, depending on the type of artifact that is untraced, try one of these solutions.

Fix an untraced design artifact

Check that the design artifact does not rely on a model callback to be linked with the model. The Model Maintainability Dashboard does not execute model loading callbacks when it loads the models for analysis. If a model relies on a callback to link a data dictionary, the data dictionary will not be linked when the dashboard runs the traceability analysis.

See Also

“Model Maintainability Metrics”

Assess Model Size and Complexity for ISO 26262

In this section...

“Open Model Maintainability Dashboard” on page 5-158

“Review Maintainability Metrics for Design Artifacts” on page 5-158

You can use the Model Maintainability Dashboard to assess the size and complexity of your models in accordance with ISO 26262-6:2018. The dashboard monitors the design artifacts in your project and provides an overview of the size, complexity, and architecture of each software unit and component. The dashboard provides:

- Quality metrics for complexity, at the model level, in accordance with the table in ISO 26262-6:2018, Clause 5.4.3
- Quality metrics for the structure, size, and complexity of software components, and the size of interfaces, in accordance with the table in ISO 26262-6:2018, Clause 7.4.3
- A list of the design artifacts in the project, organized by software units and components

To help you enforce low complexity and size in your design artifacts, follow these automated and manual review steps using the Model Maintainability Dashboard.

Open Model Maintainability Dashboard

To analyze design artifacts using the Model Maintainability Dashboard:

- 1 Open a project that contains your design artifacts. To load an example project for the dashboard, in the MATLAB Command Window, enter:

```
dashboardCCProjectStart
```

- 2 To open the Model Maintainability Dashboard, use one of these approaches:

- On the **Project** tab, click **Model Design Dashboard**.
- In the Command Window, enter:

```
modelDesignDashboard
```

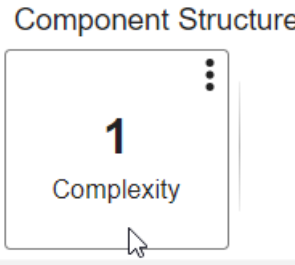
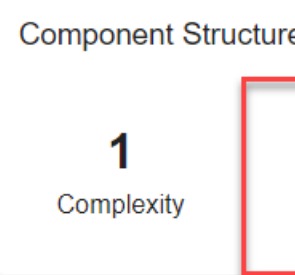
The Dashboard window launches and opens a new **Model Maintainability** tab for the software component `cc_CruiseControl`.

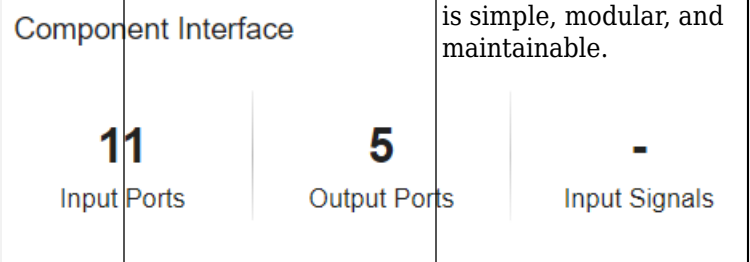
The dashboard widgets summarize the size, architecture, and complexity measurements for the design artifacts associated with each software unit and component in the design.

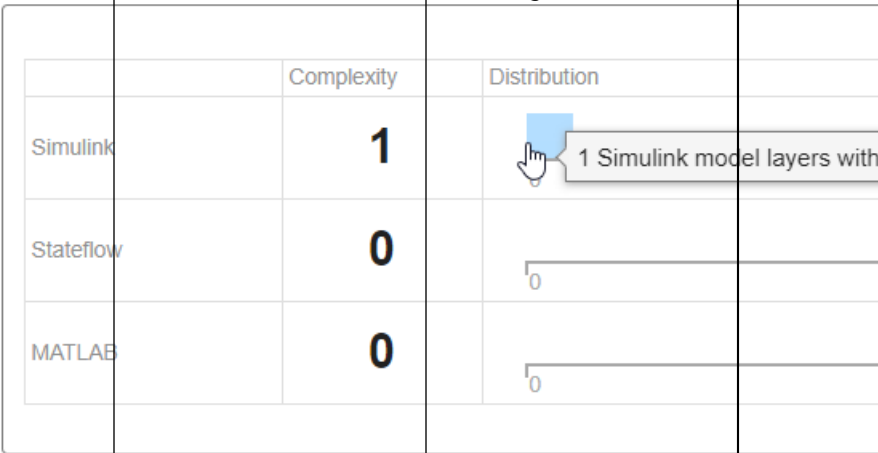
Review Maintainability Metrics for Design Artifacts

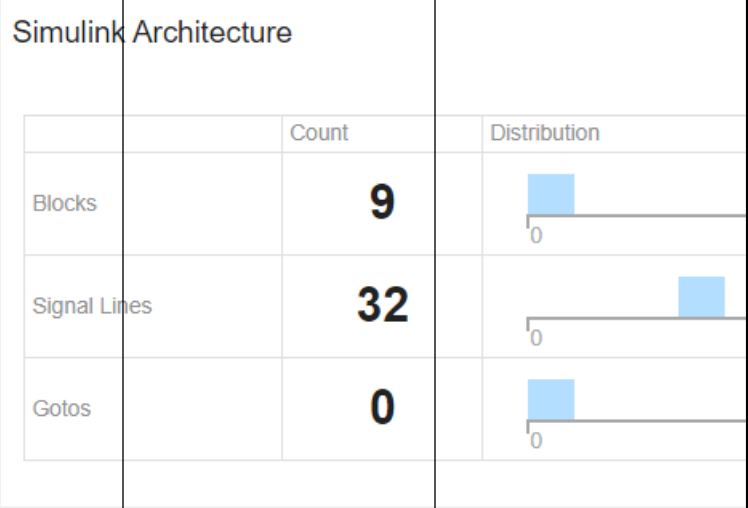
ISO 26262-6, Clause 5.4.3 requires that designs maintain a low complexity. ISO 26262-6, Clause 7.4.3 requires that designs demonstrate modularity and maintainability. To meet these requirements, use the appropriate software structure for your design and restrict the size and complexity of software components and interfaces.

The following is an example checklist to aid in reviewing design size, architecture, and complexity with respect to ISO 26262-6. For each checklist item, perform the review activity using the corresponding dashboard metric. Review and modify this example checklist to meet your application needs.

Checklist Item	Review Activity	Dashboard Metric	Rationale
<p>1 — Does the unit or component have low complexity at the model level?</p>	<p>Check that the overall design cyclomatic complexity is low.</p>	<p>In the Component Structure section, view the Complexity widget.</p>  <p>Metric ID — <code>slcomp.OverallCyclomaticComplexity</code></p> <p>For more information, see “Overall Design Cyclomatic Complexity” on page 5-294.</p>	<p>The overall design cyclomatic complexity is the total number of execution paths across the Simulink, Stateflow, and MATLAB design artifacts. In general, the more paths there are through a design, the more complex the design is.</p>
<p>2 — Does the unit or component use an appropriate model structure?</p>	<p>Check that the maximum layer breadth and depth are low. Click the Depth widget to view a table of the model layers that contribute to the depth of the unit or component. Click the Breadth widget to view a table of the model layers that contribute to the breadth of the unit or component.</p>	<p>In the Component Structure section, view the Depth and Breadth widgets.</p>  <p>Metric ID — <code>slcomp.MaxLayerDepth</code></p> <p>Metric ID — <code>slcomp.MaxLayerBreadth</code></p> <p>For more information, see “Maximum Layer Depth” on page 5-298 and “Maximum Layer Breadth” on page 5-302.</p>	<p>In a model with a hierarchy of child models, the maximum layer depth calculates the maximum depth of the hierarchical children. The maximum layer breadth calculates the maximum number of direct children on a model layer. In a hierarchical model structure, the maximum depth and maximum breadth are typically proportional to each other. If one metric result is significantly larger than the other, the structure is not properly hierarchical.</p>

Checklist Item	Review Activity	Dashboard Metric	Rationale
<p>3 — Does the unit or component have a reasonable number of interfaces?</p>	<p>Check the size of component interfaces. Click the widgets in the Component Interface section to view tables of the interfaces associated with the design.</p>	<p>In the Component Interface section, view the widgets for Input Ports, Output Ports, Input Signals, Output Signals.</p>  <p>Metric ID — slcomp.InterfacePorts</p> <p>Metric ID — slcomp.ComponentInterfaceSignals</p> <p>For more information, see “Input and Output Component Interface Ports” on page 5-304 and “Input and Output Component Interface Signals” on page 5-306.</p>	<p>The size of the component interfaces should be the minimum size that is appropriate for the design. The appropriate size of the interfaces is a size that is simple, modular, and maintainable.</p>

Checklist Item	Review Activity	Dashboard Metric	Rationale
<p>4 — Does each aspect of the design maintain a low complexity?</p>	<p>Check the design cyclomatic complexity values and decision count distributions associated with the Simulink, Stateflow, and MATLAB artifacts in the design. Click the widgets in the Design Cyclomatic Complexity Breakdown section to view tables that show the number of (associated with design artifacts</p>	<p>In the Design Cyclomatic Complexity Breakdown, view the values in the Complexity column and the distribution of decisions in the Distribution column. Point to a distribution bin to view more information.</p>  <p>Metric ID — slcomp.SLCyclomaticComplexity</p> <p>Metric ID — slcomp.SFCyclomaticComplexity</p> <p>Metric ID — slcomp.MatlabCyclomaticComplexity</p> <p>For more information, see “Design Cyclomatic Complexity Breakdown”.</p>	<p>The design cyclomatic complexity values for Simulink, Stateflow, and MATLAB are the total number of execution paths associated with those types of artifacts. In general, the more paths there are through a design, the more complex the design is. The Distribution column shows where decision paths occur in the design.</p>

Checklist Item	Review Activity	Dashboard Metric	Rationale												
5 — Is each aspect of the design appropriately modular?	Check the architecture of the Simulink, Stateflow, and MATLAB artifacts in the design. Click the widgets in the Simulink Architecture , Stateflow Architecture , and MATLAB Architecture sections to view tables that show the number of decisions associated with the design artifacts.	In the Simulink Architecture , Stateflow Architecture , and MATLAB Architecture sections, view the values in the Count column and the distribution of decisions in the Distribution column. Point to a distribution bin to view more information.	If parts of your design are large, make many decisions, or are difficult to read, the design can be difficult to test and maintain. Use the metric results to determine if you want to refactor your design to identify smaller testable units, or restructure a model to improve readability.												
		 <table border="1"> <thead> <tr> <th></th> <th>Count</th> <th>Distribution</th> </tr> </thead> <tbody> <tr> <td>Blocks</td> <td>9</td> <td></td> </tr> <tr> <td>Signal Lines</td> <td>32</td> <td></td> </tr> <tr> <td>Gotos</td> <td>0</td> <td></td> </tr> </tbody> </table>		Count	Distribution	Blocks	9		Signal Lines	32		Gotos	0		
	Count	Distribution													
Blocks	9														
Signal Lines	32														
Gotos	0														

Reference:

- ISO 26262-6:2018(en)Road vehicles — Functional safety — Part 6: Product development at the software level, International Standardization Organization

See Also

“Model Maintainability Metrics”

Related Examples

- “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard” on page 5-144
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

Analyze Your Project With Dashboards

The Dashboard window offers a centralized location where you can open dashboards for software units and components in your project, view metric results, analyze affected artifacts, and generate reports on the quality and compliance of the artifacts in your project.

Open a New Dashboard

To open a dashboard, use the Dashboards gallery in the **Add Dashboard** section of the toolstrip to select a type of dashboard:

- Click **Model Maintainability** to analyze the size, architecture, and complexity of the MATLAB, Simulink, and Stateflow artifacts in your project.

For more information, see “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard” on page 5-144.

- Click **Model Testing** to assess the traceability and completeness of the models, requirements, tests, and test results in your project.

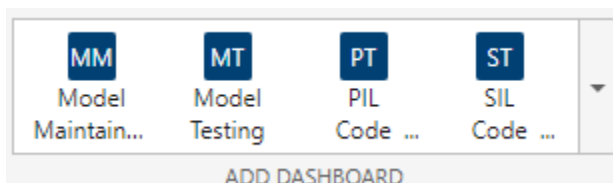
For more information, see “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80.

- Click **PIL Code Testing** to assess the status and quality of your processor-in-the-loop (PIL) code testing.

For more information, see “View Status of Code Testing Activities for Software Units in Project” on page 5-184.

- Click **SIL Code Testing** to assess the status and quality of your software-in-the-loop (SIL) code testing.

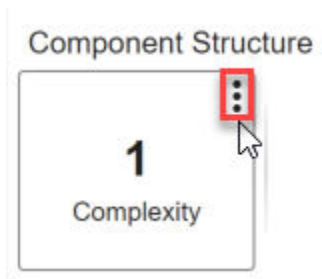
For more information, see “View Status of Code Testing Activities for Software Units in Project” on page 5-184.




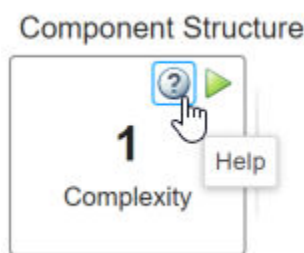
By default, a new dashboard tab opens for the first available unit or component in the **Project** panel. If a dashboard is not available for a specific unit or component, the name of the unit or component appears dimmed in the **Project** panel. Click the name of a unit or component that is not dimmed to view the dashboard results for that specific unit or component.

View Help for Metric Information

The dashboards use widgets to display the metric results. When you point to a widget, three dots appears in the top-right corner of the widget.





Point to the three dots and click the Help icon  to view more information about the metric and how the dashboard calculates the metric value.

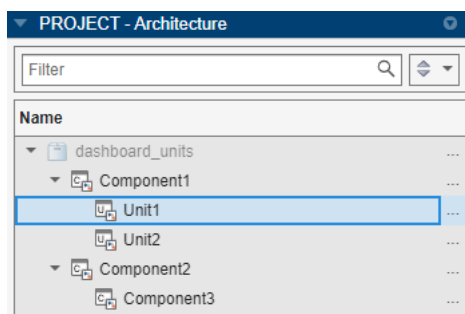


Navigate the Dashboard Window

The Dashboard window contains multiple sections:

- **Project** panel

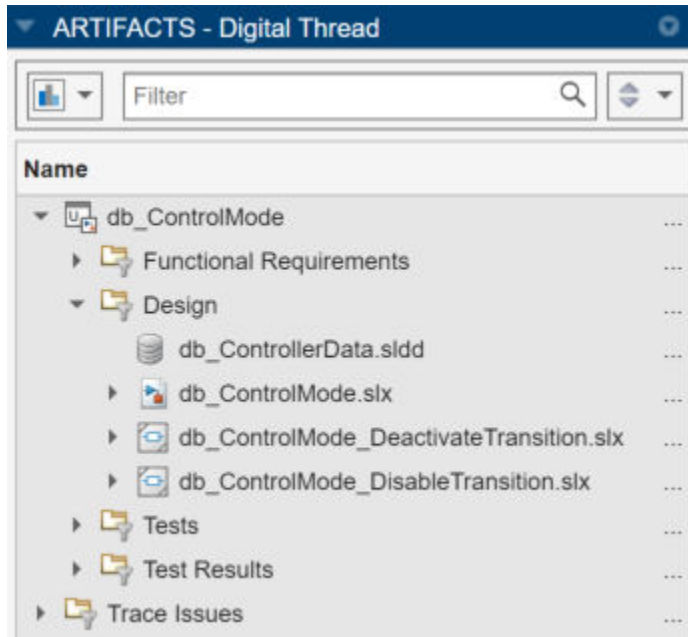
The **Project** panel shows the architecture of the software units and components in the current project. A unit is a functional entity in your software architecture that you can execute and test independently or as part of larger system tests. A component is an entity that integrates multiple testable units together. In the Dashboard window, the unit dashboard icon  indicates a unit and the component dashboard icon  indicates a component. Click the three dots to the right of each unit or component to view more information about that unit or component. By default, the dashboard considers a model to be a unit if it does not reference other models. For information on how to specify which entities the **Project** panel identifies as units and components, see “Specify Models as Components and Units”.



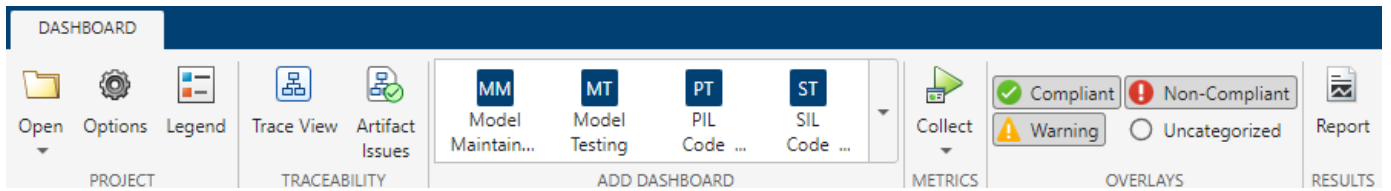
- **Artifacts** panel

Use the **Artifacts** panel to view the **Functional Requirements**, **Design**, **Tests**, and **Test Results** folders, which contain the artifacts the dashboard traced to the current unit or component

selected in the **Project** panel. For example, the **Design** folder for a unit might show the data dictionary, model, and subsystem references associated with the unit. The **Artifacts** panel uses a digital thread of project metadata information to capture and track information about the artifacts in your project and the relationship between artifacts. Click the three dots to the right of each folder or artifact to view more information.



- **Dashboard** toolstrip
 - Use the **Project** section of the toolstrip to open a project, specify how the dashboard identifies units and components, and change the layout that the dashboard uses. You can click the **Legend** button to view a list of the icons for artifacts that the dashboard traces.
 - Use the **Traceability** section of the toolstrip to explore traceability relationships and to troubleshoot artifact issues in the project. Click **Trace View** to open an interactive diagram that shows traceability information for artifacts in the project. Click **Artifact Issues** to troubleshoot warnings and errors associated with project artifacts. For more information, see "Explore Traceability Information for Units and Components" on page 5-205.
 - Use the **Add Dashboard** section to select a type of dashboard to open.
 - The **Metrics** section shows the options for tracing artifacts and collecting metric results. By default, the dashboards automatically trace the artifacts in your project and collect metrics when you initially select a dashboard. However, you can use the **Metrics** section if you need to manually trace artifacts, manually collect metric results, collect metric results for each of the units and components in the project, or turn off the automatic tracing and collecting settings.
 - The **Overlays** section shows which types of overlays are enabled. The dashboard uses overlays to show if the metric results are compliant with the metric thresholds, non-compliant with the metric thresholds, generate a warning, or are uncategorized because there are no metric thresholds specified. By default, the dashboard starts with the **Compliant**, **Non-Compliant**, and **Warning** buttons selected. To turn an overlay on or off, click the button for the overlay to select or de-select the overlay.
 - In the **Results** section, click the **Report** button to specify the report settings and to generate a report of metric results.



See Also

“Specify Models as Components and Units” | “Model Maintainability Metrics”

Related Examples

- “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard” on page 5-144
- “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80

Digital Thread

The dashboards create a *digital thread* to capture the attributes and unique identifiers of the artifacts in your project. The digital thread is a set of metadata information about the artifacts in a project, the artifact structure, and the traceability relationships between artifacts.

The dashboards monitor and analyze the digital thread to:

- Detect when project files move and maintain the same universal unique identifiers (UUIDs) for the artifact files and the elements inside the artifact files
- Capture traceability and listen to tools, such as the Test Manager in Simulink Test, to detect new tool outputs and the dependencies of the tool operations
- Identify outdated tool outputs by analyzing the traceability and checksums of inputs to the tool operations
- Create an index of your project and store a representation of each artifact, their inner structure, and their relationships with other artifacts
- Provide a holistic analysis of your project artifacts to help you maintain traceability and up-to-date information on the requirements, units, tests, and results impacting your design

The dashboard can store the results of the digital thread analysis and then perform traceability analysis across domains, tools, and artifacts, without needing to locally load or access the project artifacts.

As you modify your models and testing artifacts, check that you save the changes to the artifacts files and store the files that you want to analyze in your project.

To identify and maintain traceability relationships, the digital thread requires that your project is set up correctly. The digital thread requires that:

- Artifacts are saved in the current project.
- Artifact tracing is enabled for the project. You can view and change the artifact tracing setting in the **Startup and Shutdown** settings for your project. Select **Track tool outputs to detect outdated results** to enable artifact tracing.

If an artifact does not appear in the **Artifacts** panel when you expect it to, see “Resolve Missing Artifacts, Links, and Results” on page 5-128.

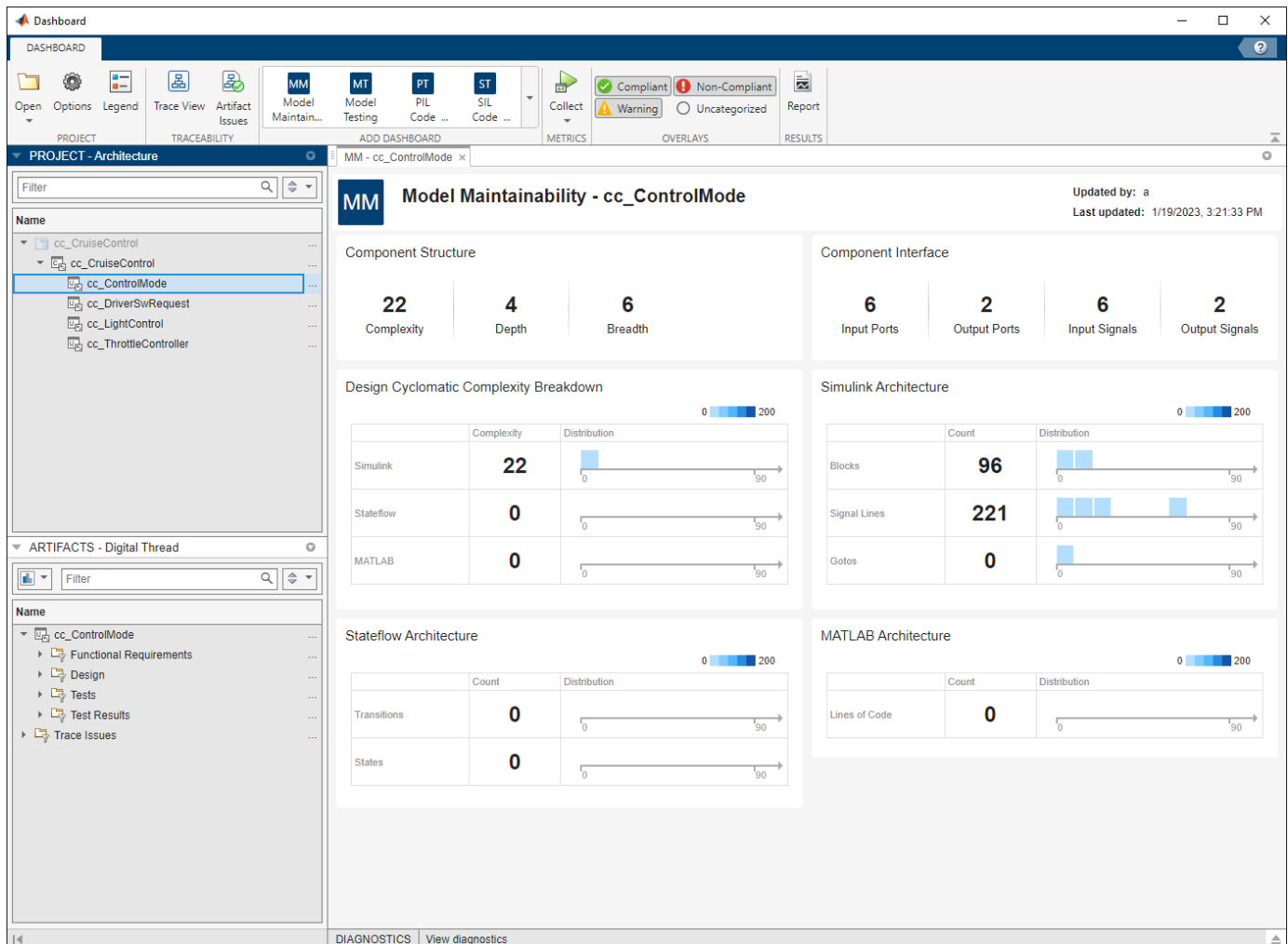
See Also

Related Examples

- “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard” on page 5-144
- “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80
- “View Status of Code Testing Activities for Software Units in Project” on page 5-184

Manage Design Artifacts for Analysis in the Model Maintainability Dashboard

When you develop software units and components using Model-Based Design, use the Model Maintainability Dashboard to assess the status and quality of your models. The Model Maintainability Dashboard analyzes the design artifacts in your project and provides detailed metric measurements on the size, architecture, and complexity of these design artifacts.



Each metric in the dashboard measures a different aspect of the quality of your design and reflects guidelines in industry-recognized software development standards, such as ISO 26262. To monitor the maintainability of your models in the Model Maintainability Dashboard, maintain your artifacts in a project and follow these considerations. For more information on using the Model Maintainability Dashboard, see “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard” on page 5-144.

Manage Artifact Files in a Project

To analyze your requirements-based testing activities in the Model Maintainability Dashboard, store your design and testing artifacts in a MATLAB project. The artifacts that the model design metrics analyze include:

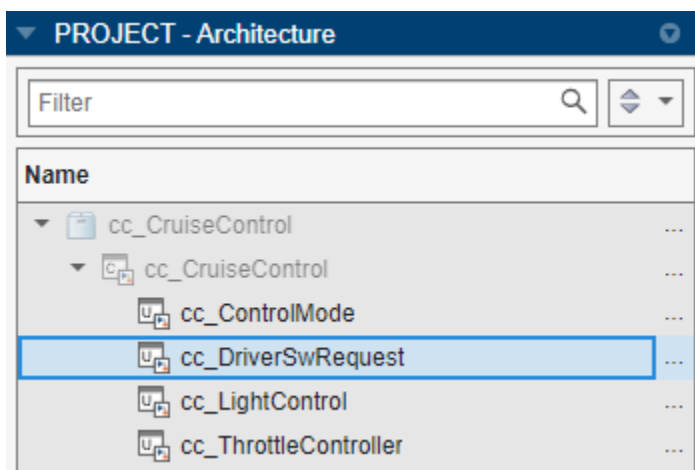
- Simulink models
- Libraries that the models use
- Stateflow charts
- MATLAB code

For information on how the dashboard traces dependencies between project files, see “Digital Thread” on page 5-167.

When your project contains many models and model reference hierarchies, you can configure the dashboard to recognize the different testing levels of your models. You can specify which entities in your software architecture are units or higher-level components by labeling them in your project and configuring the Model Maintainability Dashboard to recognize the labels. The dashboard organizes your models in the **Artifacts** panel according to their testing levels and the model reference hierarchy. For more information, see “Categorize Models in a Hierarchy as Components or Units” on page 5-119.

Trace Artifacts to Units and Components

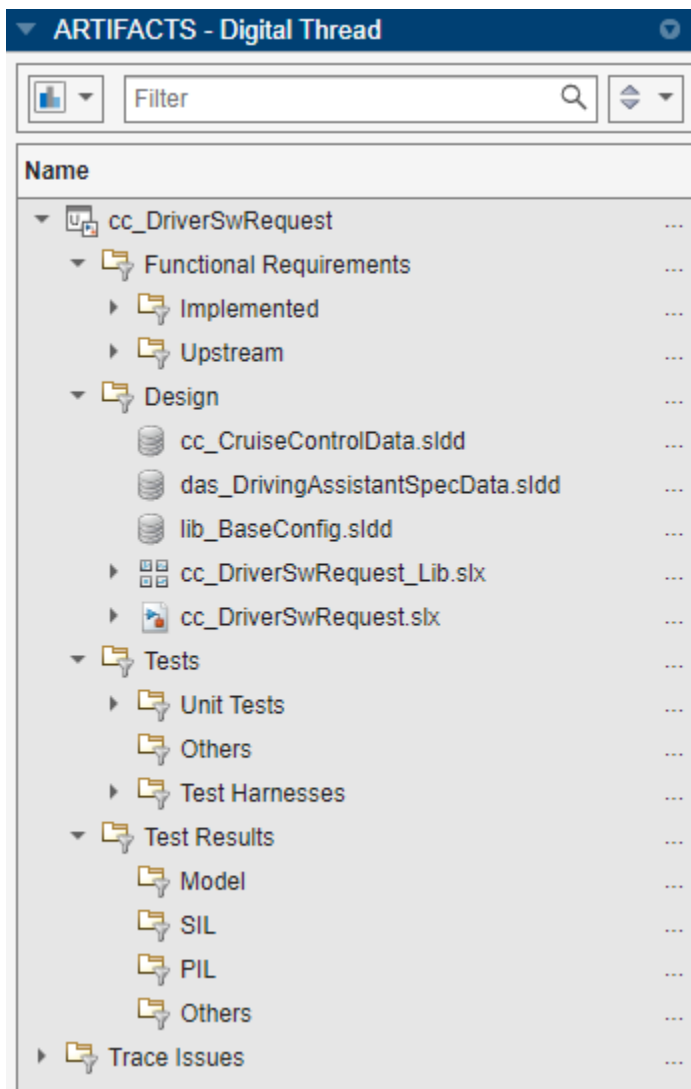
To determine which artifacts are in the scope of a unit or component, the dashboard analyzes the traceability links between the artifacts, software units, and components in the project. The **Project** panel lists the units, organized by the components that reference them.



When you select a unit or component in the **Project** panel, the **Artifacts** panel shows the artifacts that trace to the selected unit or component. Traced artifacts include:

- Functional Requirements
- Design Artifacts
- Tests

- Test Results



To see the traceability paths that the dashboard found, click the **Trace View** button in the toolbar. For more information, “Explore Traceability Information for Units and Components” on page 5-205.

In the **Artifacts** panel, the folder **Trace Issues** contains unexpected requirement links, requirements links which are broken or not supported by the dashboard, and artifacts that the dashboard cannot trace to a unit or component. To help identify the type of tracing issue, the folder **Trace Issues** contains subfolders for **Unexpected Implementation Links, Unresolved and Unsupported Links, Untraced Tests, and Untraced Results**. For more information, see “Resolve Missing Artifacts, Links, and Results” on page 5-128.

If an artifact returns an error during traceability analysis, the panel includes the artifact in an **Errors** folder. Use the traceability information in these sections to check if the artifacts trace to the units or components that you expect. To see details about the warnings and errors that the dashboard finds during artifact analysis, click **Artifact Issues** in the toolbar. For more information, see “View Artifact Issues in Project” on page 5-212.

Functional Requirements

The folder **Functional Requirements** shows requirements of **Type Functional** that are either implemented by or upstream of the unit or component.

When you collect metric results, the dashboard analyzes only the functional requirements that the unit or component directly implements. The folder **Functional Requirements** contains two subfolders to help identify which requirements are implemented by the unit or component, or are upstream of the unit or component:

- **Implemented** — Functional requirements that are directly linked to the unit or component with a link **Type of Implements**. The dashboard uses these requirements in the metrics for the unit or component.
- **Upstream** — Functional requirements that are indirectly or transitively linked to the implemented requirements. The dashboard does not use these requirements in the metrics for the unit or component.

If a requirement does not trace to a unit or component, it appears in the “Trace Issues” folder. If a requirement does not appear in the **Artifacts** panel when you expect it to, see “Requirement Missing from Artifacts Panel” on page 5-130.

Use the Requirements Toolbox to create or import the requirements in a requirements file (.slreqx).

Design Artifacts

The folder **Design** shows:

- The model file that contains the block diagram for the unit or component.
- Models that the unit or component references.
- Libraries that are partially or fully used by the model.
- Data dictionaries that are linked to the model.

Tests

The folder **Tests** shows tests and test harnesses that trace to the selected unit.

When you collect metric results for a unit, the dashboard analyzes only the tests for unit tests. The folder **Tests** contains subfolders to help identify whether a test is considered a unit test and which test harnesses trace to the unit:

- **Unit Tests** — Tests that the dashboard considers as unit tests. A unit test directly tests either the entire unit or lower-level elements in the unit, like subsystems. The dashboard uses these tests in the metrics for the unit.
- **Others** — Tests that trace to the unit but that the dashboard does not consider as unit tests. For example, the dashboard does not consider tests on a library to be unit tests. The dashboard does not use these tests in the metrics for the unit.
- **Test Harnesses** — Test harnesses that trace to the unit or lower-level elements in the unit. Double-click a test harness to open it.

If a test does not trace to a unit, it appears in the “Trace Issues” folder. If a test does not appear in the **Artifacts** panel when you expect it to, see “Test Missing from Artifacts Panel” on page 5-130. For troubleshooting tests in metric results, see “Fix a test that does not produce metric results” on page 5-135.



Create tests by using Simulink Test.

Test Results

When you collect metric results for a unit, the dashboard analyzes only the test results from unit tests. The folder **Test Results** contains subfolders to help identify which test results are from unit tests:

- The subfolders for **Model**, **SIL**, and **PIL** contain simulation results from normal, software-in-the-loop (SIL), and processor-in-the-loop (PIL) unit tests, respectively. The dashboard uses these results in the metrics for the unit.

The following types of test results are shown:

-  Saved test results — results that you have collected in the Test Manager and have exported to a results file.
-  Temporary test results — results that you have collected in the Test Manager but have not exported to a results file. When you export the results from the Test Manager the dashboard analyzes the saved results instead of the temporary results. Additionally, the dashboard stops recognizing the temporary results when you close the project or close the result set in the Simulink Test Result Explorer. If you want to analyze the results in a subsequent test session or project session, export the results to a results file.
- **Others** — Results that are not simulation results, are not from unit tests, or are only reports. For example, SIL results are not simulation results. The dashboard does not use these results in the metrics for the unit.

If a test result does not trace to a unit, it appears in the “Trace Issues” folder. If a test result does not appear in the **Artifacts** panel when you expect it to, see “Test Result Missing from Artifacts Panel” on page 5-130. For troubleshooting test results in dashboard metric results, see “Fix a test result that does not produce metric results” on page 5-135.

Trace Issues

The folder **Trace Issues** shows artifacts that the dashboard has not traced to any units or components. Use the folder **Trace Issues** to check if artifacts are missing traceability to the units or components. The folder **Trace Issues** contains subfolders to help identify the type of tracing issue:

- **Unexpected Implementation Links** — Requirement links of **Type Implements** for a requirement of **Type Container** or **Type Informational**. The dashboard does not expect these links to be of **Type Implements** because container requirements and informational requirements do not contribute to the Implementation and Verification status of the requirement set that they are in. If a requirement is not meant to be implemented, you can change the link type. For example, you can change a requirement of **Type Informational** to have a link of **Type Related to**.
- **Unresolved and Unsupported Links** — Requirements links that are either broken in the project or not supported by the dashboard. For example, if a model block implements a requirement, but you delete the model block, the requirement link is now unresolved. The dashboard does not support traceability analysis for some artifacts and some links. If you expect a link to trace to a unit or component and it does not, see the troubleshooting solutions in “Resolve Missing Artifacts, Links, and Results” on page 5-128.
- **Untraced Tests** — Tests that execute on models or lower-level elements, like subsystems, that are not on the project path.

- **Untraced Results** — Results that the dashboard cannot trace to a test. For example, if a test produces a result, but you delete the test, the dashboard cannot trace the results to the test.

The dashboard does not support traceability analysis for some artifacts and some links. If an artifact is untraced when you expect it to trace to a unit or component, see the troubleshooting solutions in “Trace Issues” on page 5-133.

Artifact Errors

The folder **Errors** appears if artifacts returned errors when the dashboard performed artifact analysis. These are some errors that artifacts might return during traceability analysis:

- An artifact returns an error if it has unsaved changes when traceability analysis starts.
- A test results file returns an error if it was saved in a previous version of Simulink.

Open these artifacts and fix the errors. The dashboard shows a banner at the top of the dashboard to indicate that the artifact traceability shown in the **Project** and **Artifacts** panels is outdated. Click the **Trace Artifacts** button on the banner to refresh the data in the **Project** and **Artifacts** panels.

Artifact Issues

To see details about artifacts that cause errors, warnings, and informational messages during analysis, click the **Artifact Issues** button in the toolstrip. You can sort the messages by their type: **Error**, **Warning**, and **Info**.

The messages show:

- Modeling constructs that the dashboard does not support
- Links that the dashboard does not trace
- Test harnesses or cases that the dashboard does not support
- Test results missing coverage or simulation results
- Artifacts that return errors when the dashboard loads them
- Information about model callbacks that the dashboard deactivates
- Files that have path traceability issues
- Artifacts that are not on the path and are not considered during tracing

Collect Metric Results

The Model Maintainability Dashboard can collect metric results for each unit and component listed in the **Project** panel. Each metric in the dashboard measures a different aspect of the quality of your model maintainability and reflects guidelines in industry-recognized software development standards, such as ISO 26262. For more information about the available metrics and the results that they return, see “Model Maintainability Metrics”.

As you edit and save the artifacts in your project, the dashboard detects changes to the artifacts. If the metric results might be affected by your artifact changes, the dashboard shows a warning banner at the top of the dashboard to indicate that the metric results are stale. Affected widgets have a gray



staleness icon

. To update the results, click the **Collect** button on the warning banner to re-collect the metric data and to update the stale widgets with data from the current artifacts. If

you want to collect metrics for each of the units and components in the project, click **Collect > Collect All**.

See Also

“Model Maintainability Metrics”

Related Examples

- “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard” on page 5-144
- “Resolve Missing Artifacts, Links, and Results” on page 5-128

Explore Metric Results, Monitor Progress, and Identify Issues

With Simulink Check, you can view metrics for your Model-Based Design artifacts, assess the current status of a project, and identify and fix non-compliant metric results. You can collect the metric results programmatically or use dashboards to visualize the metric results. You can also save reports for offline review of metric results.

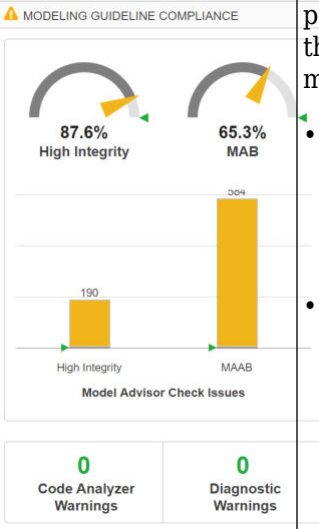
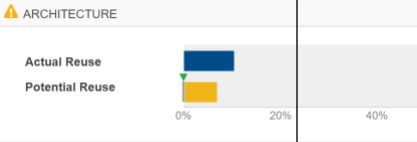

The dashboards and metric results provide an overview of model quality and the project verification status. Use the detailed metric results to identify specific artifacts that are not in compliance with industry standards and guidelines. The dashboard summarizes metric results so you can track progress and identify gaps. As you update, refactor, and fix artifacts, return to the dashboard to monitor your progress towards fixing non-compliant artifacts or gaps in requirements-based testing.

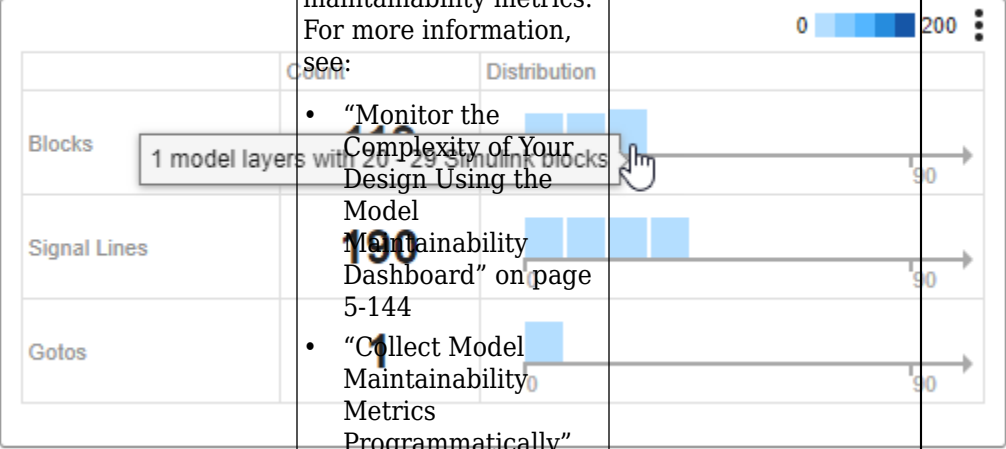
Metric Results in the Dashboards

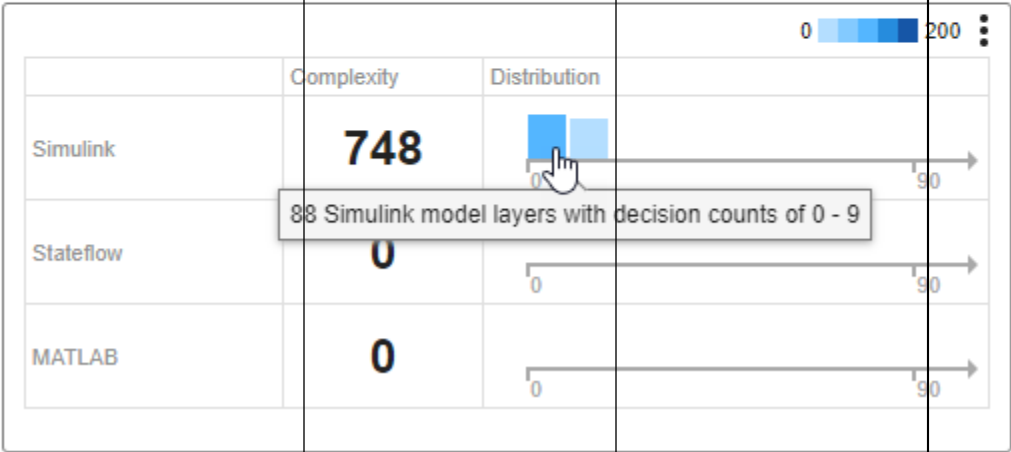
There are different types of dashboards:

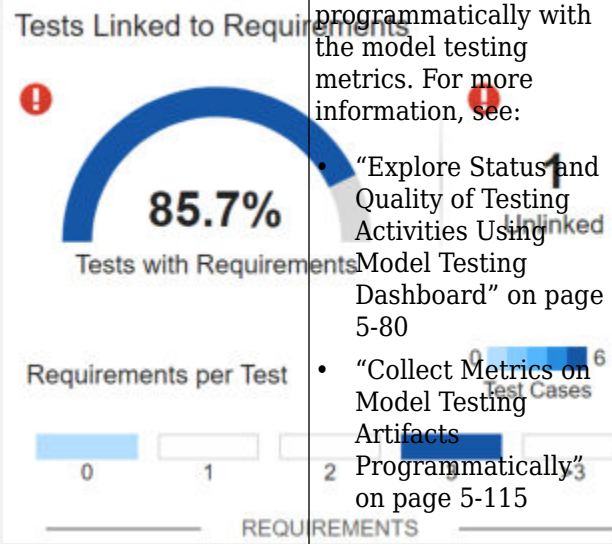
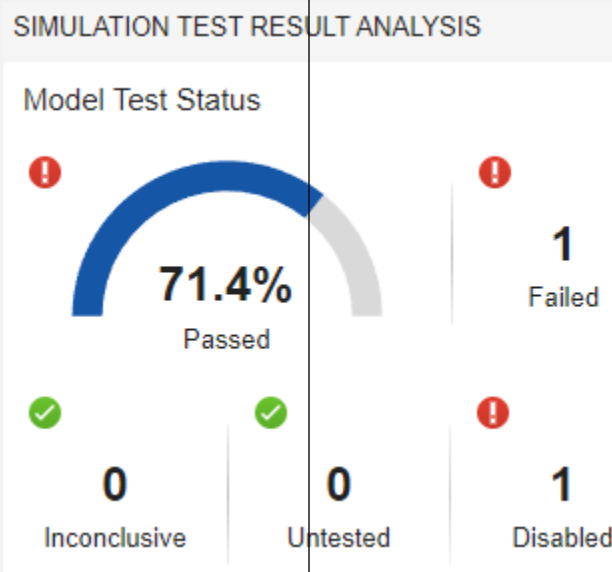
- **Metrics Dashboard** — To help you assess the status of your model against industry and custom modeling guidelines.
- **Model Maintainability Dashboard** — To help you assess the size, architecture, and complexity of the models in your project.
- **Model Testing Dashboard** — To help you assess the model testing artifacts in your project.
- **SIL Code Testing and PIL Code Testing Dashboards** — To help you assess the code testing artifacts in your project.

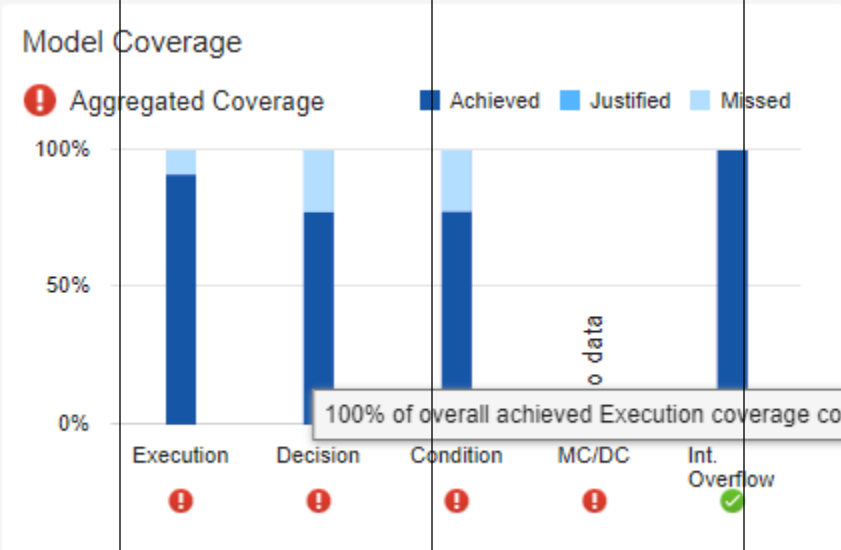
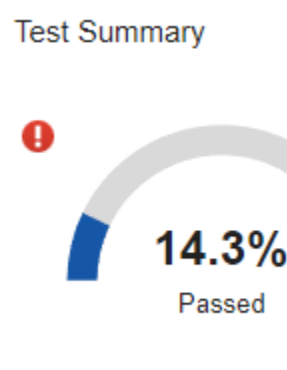

The following table shows examples of common dashboard results, the associated dashboard and metrics, and links to related examples.

Dashboard	Dashboard Results	Dashboard and Metric Examples	Guidelines and Standards
<p>Metrics Dashboard</p>	<p>Assess compliance to modeling standards and guidelines.</p>  <p>Identify parts of your model that use library content to reuse existing modeling components or parts of your model that you can potentially refactor to reuse modeling components.</p>  <p>Create custom model metrics and customize how to display metric results in the dashboard.</p> 	<p>Use the Metrics Dashboard to visualize the metric results or collect results programmatically with the model metrics. For more information, see:</p> <ul style="list-style-type: none"> • “Collect and Explore Metric Data by Using the Metrics Dashboard” on page 5-2 • “Collect Metric Data Programmatically and View Data Through the Metrics Dashboard” on page 5-28 	<p>Use the Metrics Dashboard to collect compliance data from checks on modeling guidelines. For more information, see “Collect Compliance Data and Explore Results in the Model Advisor” on page 5-25.</p> <p>By default, the Modeling Guideline Compliance section of the dashboard shows the results of high-integrity and MAAB Model Advisor checks, but you can customize the Metrics Dashboard to show compliance issues for other groups of checks. For more information, see “Customize Metrics Dashboard Layout and Functionality” on page 5-37.</p>

Dashboard	Dashboard Results	Dashboard and Metric Examples	Guidelines and Standards
<p>Model Maintainability Dashboard</p>	<p>View the overall architecture of the Simulink, Stateflow, and MATLAB artifacts in your project.</p> <p>Simulink Architecture</p> 	<p>Use the Model Maintainability Dashboard to visualize the metric results or collect results programmatically with the model maintainability metrics. For more information, see:</p> <ul style="list-style-type: none"> • “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard” on page 5-144 • “Collect Model Maintainability Metrics Programmatically” on page 5-150 	<p>“Assess Model Size and Complexity for ISO 26262” on page 5-158</p>

Dashboard	Dashboard Results	Dashboard and Metric Examples	Guidelines and Standards												
	<p>Pinpoint areas in the design that have high complexity.</p> <p>Component Structure</p> <p>748 Complexity</p>	<p>4 Depth</p>	<p>87 Breadth</p>												
	<p>Design Cyclomatic Complexity Breakdown</p>  <table border="1" data-bbox="558 793 1563 1241"> <thead> <tr> <th></th> <th>Complexity</th> <th>Distribution</th> </tr> </thead> <tbody> <tr> <td>Simulink</td> <td>748</td> <td></td> </tr> <tr> <td>Stateflow</td> <td>0</td> <td></td> </tr> <tr> <td>MATLAB</td> <td>0</td> <td></td> </tr> </tbody> </table>				Complexity	Distribution	Simulink	748		Stateflow	0		MATLAB	0	
	Complexity	Distribution													
Simulink	748														
Stateflow	0														
MATLAB	0														

Dashboard	Dashboard Results	Dashboard and Metric Examples	Guidelines and Standards
<p>Model Testing Dashboard</p>	<p>Trace the relationships between models, requirements, and tests.</p>  <p>View the current status of model testing and identify gaps in test results.</p> 	<p>Use the Model Testing Dashboard to visualize the metric results or collect results programmatically with the model testing metrics. For more information, see:</p> <ul style="list-style-type: none"> • “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80 • “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115 	<p>“Assess Requirements-Based Testing for ISO 26262” on page 5-102</p>

Dashboard	Dashboard Results	Dashboard and Metric Examples	Guidelines and Standards
	<p>View the overall achieved model coverage and monitor the sources of coverage.</p> 		<p>Achieved Coverage Rat</p> <p>Requirements-Based Tests</p> <ul style="list-style-type: none"> Execution ✔ Decision ✔ Condition ✔ MC/DC ❗ No data <p>Unit-Boundary Tests</p> <ul style="list-style-type: none"> Condition █ MC/DC █ No data
<p>SIL Code Testing and PIL Code Testing Dashboards</p>	<p>View the current status of code testing and identify gaps in test results.</p> 	<p>Use the SIL Code Testing and PIL Code Testing dashboards to visualize the metric results or collect results programmatically with the code testing metrics. For more information, see:</p> <ul style="list-style-type: none"> • “View Status of Code Testing Activities for Software Units in Project” on page 5-184 • “Collect Code Testing Metrics Programmatically” on page 5-201 	<p>“Identify and Troubleshoot Gaps in Code Testing Results and Coverage” on page 5-192</p> 

Dashboard	Dashboard Results	Dashboard and Metric Examples	Guidelines and Standards																														
	<p>View the overall achieved code coverage and monitor the sources of coverage.</p>  <p>Identify issues across model and code testing results.</p> <table border="1" data-bbox="560 1123 1380 1470"> <caption>Coverage Recap</caption> <thead> <tr> <th></th> <th>Justified SIL</th> <th>Justified Model</th> <th>Completed SIL</th> <th>Completed Model</th> </tr> </thead> <tbody> <tr> <td>Integer Overflow</td> <td>N/A</td> <td>0%</td> <td>N/A</td> <td>100%</td> </tr> <tr> <td>Statement / Execution</td> <td>0%</td> <td>0%</td> <td>69.5%</td> <td>79.4%</td> </tr> <tr> <td>Decision</td> <td>0%</td> <td>0%</td> <td>48.5%</td> <td>42.5%</td> </tr> <tr> <td>Condition</td> <td>0%</td> <td>0%</td> <td>54.2%</td> <td>41.7%</td> </tr> <tr> <td>MC/DC</td> <td>No data</td> <td>No data</td> <td>No data</td> <td>No data</td> </tr> </tbody> </table>		Justified SIL	Justified Model	Completed SIL	Completed Model	Integer Overflow	N/A	0%	N/A	100%	Statement / Execution	0%	0%	69.5%	79.4%	Decision	0%	0%	48.5%	42.5%	Condition	0%	0%	54.2%	41.7%	MC/DC	No data	No data	No data	No data		
	Justified SIL	Justified Model	Completed SIL	Completed Model																													
Integer Overflow	N/A	0%	N/A	100%																													
Statement / Execution	0%	0%	69.5%	79.4%																													
Decision	0%	0%	48.5%	42.5%																													
Condition	0%	0%	54.2%	41.7%																													
MC/DC	No data	No data	No data	No data																													

See Also
metric.Engine

Related Examples

- “Identify Modeling Clones with the Metrics Dashboard” on page 5-23
- “Explore Traceability Information for Units and Components” on page 5-205
- “Fix Requirements-Based Testing Issues” on page 5-89
- “Identify and Troubleshoot Gaps in Code Testing Results and Coverage” on page 5-192

Use the Model Maintainability Dashboard to Collect Size, Architecture, and Complexity Metrics for a Model

In the example “Collect Model Metric Data by Using the Metrics Dashboard” on page 1-9, the Metrics Dashboard collects metric results for size, architecture, complexity, and modeling guideline compliance. The Metrics Dashboard will be removed in a future release. For size, architecture, and complexity metrics, you can use the Model Maintainability Dashboard to collect metric results, create reports, and detect stale metric results as you update and refactor your model.

The Metrics Dashboard can only run on individual models and requires a computationally intensive, compile-based analysis for many metrics. Instead, use the Model Maintainability Dashboard to collect size, architecture, and complexity metrics for a model.

- 1 Create a MATLAB project. The Model Maintainability Dashboard runs on each model in the project and can aggregate metrics across software units and components.

```
matlab.project.createProject
```

- 2 Save a model to the project. For this example, open the example model `sldemo_fuelsys` and save the model as `sldemo_fuelsys_copy` inside the current project.

```
open_system('sldemo_fuelsys');
save_system('sldemo_fuelsys', 'sldemo_fuelsys_copy');
```

- 3 In the Simulink window for `sldemo_fuelsys_copy`, on the **Apps** tab, open the Model Maintainability Dashboard by clicking **Model Design Dashboard**.
- 4 Click **Enable and Continue** to allow the dashboard to track changes to the project and automatically detect stale results.

The Dashboard window opens and shows a new **Model Maintainability** tab for the model `sldemo_fuelsys_copy`. The Model Maintainability Dashboard contains widgets that show metric results for the size, architecture, and complexity of software units and components in the project. By default, the dashboard categorizes `sldemo_fuelsys_copy` as a software unit because the model does not reference other models.

- 5 In the **Component Structure** section, locate the **Complexity** widget. The overall design cyclomatic complexity for `sldemo_fuelsys_copy` is 65.

To calculate the design cyclomatic complexity, the Model Maintainability Dashboard determines the number of possible execution paths through the design. The dashboard analyzes and includes Simulink, Stateflow, and MATLAB artifacts in the calculation. The overall design cyclomatic complexity, shown in the **Complexity** widget, is the sum of the Simulink, Stateflow, and MATLAB decision counts, plus one for the default execution path.

- 6 In the **Design Cyclomatic Complexity Breakdown** section, locate the widgets in the **Complexity** column.

The **Design Cyclomatic Complexity Breakdown** shows the design cyclomatic complexity associated with the Simulink, Stateflow, and MATLAB artifacts.

For the `sldemo_fuelsys_copy` model, the design cyclomatic complexity is 18 for Simulink-based execution paths, 48 for Stateflow-based execution paths, and one for MATLAB-based execution paths. The majority of the complexity comes from Stateflow artifacts, like charts, states, or truth tables.

- 7 In the **Stateflow** row and **Distribution** column, click the blue distribution bin to explore the Stateflow complexity metric results in more detail.

The dashboard opens the **Metric Details** for the widget. In the table, you can see that the **Artifact** `control_logic` contributes 47 graphical decisions and zero MATLAB code decisions.

- 8 In the **Artifact** column, click **control_logic** to open the artifact in the model.

For this example, `control_logic` is a single Stateflow chart that contributes 47 Stateflow decisions. To reduce the complexity of `control_logic`, you could consider refactoring the chart by moving the logic into individual atomic subcharts.

- 9 Modify `control_logic`. For this example, move the **O2** subchart and re-save the model.

The Model Maintainability Dashboard automatically detects the change to the model.

- 10 On the warning banner in the dashboard, click **Trace Artifacts**.

The dashboard identifies that the previously collected metric results are stale because the change to the model may have invalidated the metric results.

- 11 On the warning banner, click **Collect** to re-collect the metric results.

See Also

“Model Maintainability Metrics”

Related Examples

- “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard” on page 5-144
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

View Status of Code Testing Activities for Software Units in Project

The SIL Code Testing and PIL Code Testing dashboards analyze artifacts in a project, like models and test results, and collect metrics that help you see the status of code testing activities for your project. The SIL Code Testing dashboard provides an overview of software-in-the-loop (SIL) testing and the PIL Code Testing dashboard provides an overview of processor-in-the-loop (PIL) testing. Each metric in the dashboards measures a different aspect of the quality of your code testing and reflects guidelines in industry-recognized software development standards, such as ISO 26262 and DO-178C. Use the code testing dashboards after completing model testing with the Model Testing Dashboard to help perform equivalence testing between software units and their associated code.

With the metric results in the dashboard, you can:

- See a summary of the code testing results and code coverage for each of the software units in your project.
- View code testing results in context with relevant model testing results.
- View detailed information about the testing artifacts in a project.
- Monitor progress towards compliance with industry standards like ISO 26262 and DO-178C.
- Identify and troubleshoot issues in the project to reach compliance with code testing requirements.

You can click on widgets in the dashboards to see detailed information about the current results and use hyperlinks to open the affected artifacts. You can view the metric results directly in the dashboard, generate a report for offline review, or collect the metrics programmatically.

This page shows how to assess the status of SIL testing by using the SIL Code Testing dashboard, but you follow the same steps to analyze PIL testing in the PIL Code Testing dashboard. The code testing dashboards use the same layout, but the metric results come from either SIL or PIL tests, respectively.

Note Review and fix non-compliant model testing results before you review code testing results. Since models are upstream of the generated code, there are some code testing issues that you can only address during model testing. Note that *model testing results* refers to results from running tests in normal simulation mode. For information on how to review and fix model testing results, see “Fix Requirements-Based Testing Issues” on page 5-89.

View Code Testing Status

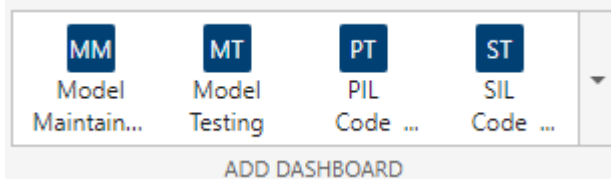
You can view the status of code testing for each of the software units in a project by using the code testing dashboards.

To open a SIL Code Testing or PIL Code Testing dashboard:

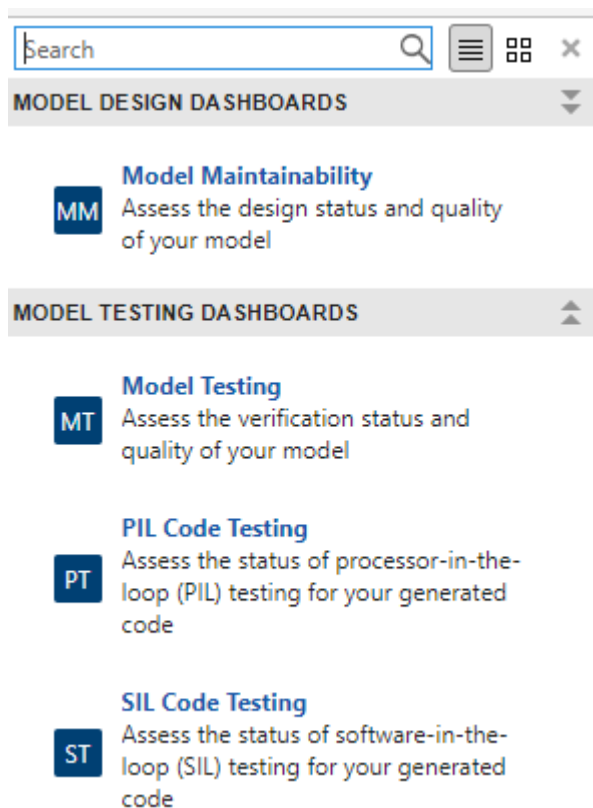
- 1 On the **Project** tab, click **Model Testing Dashboard** or, in the MATLAB Command Window, enter:

```
modelTestingDashboard
```

- View the available dashboards by expanding the dashboards gallery in the **Add Dashboard** section of the toolstrip. Click **SIL Code Testing** to open the dashboard for SIL results. You can also click **PIL Code Testing** to open the dashboard for PIL results.

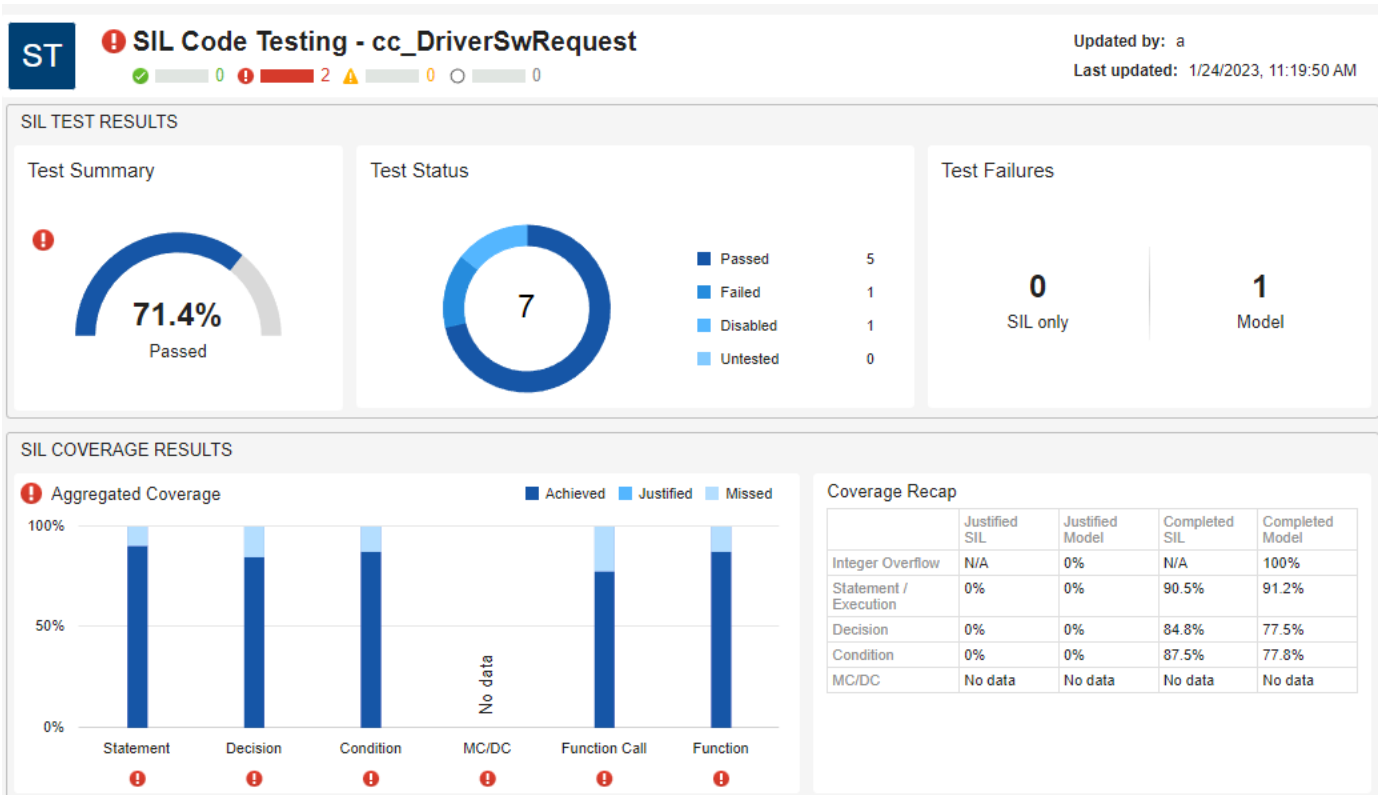


The SIL Code Testing and PIL Code Testing dashboards are two types of model testing dashboards.



The dashboard widgets show a summary of the metric results for the selected software unit. By default, the dashboard opens for the first software unit listed in the **Project** panel. You can click on a different unit in the **Project** panel to view the model testing results for that unit. For this example, click the unit **cc_DriverSwRequest**. For more information about the **Project** and **Artifacts** panels, see “Analyze Your Project With Dashboards” on page 5-163.

The following images show metric results for a project that already has model testing and SIL testing results. For information on how to collect the model and code testing results and how to address testing issues, see “Identify and Troubleshoot Gaps in Code Testing Results and Coverage” on page 5-192.

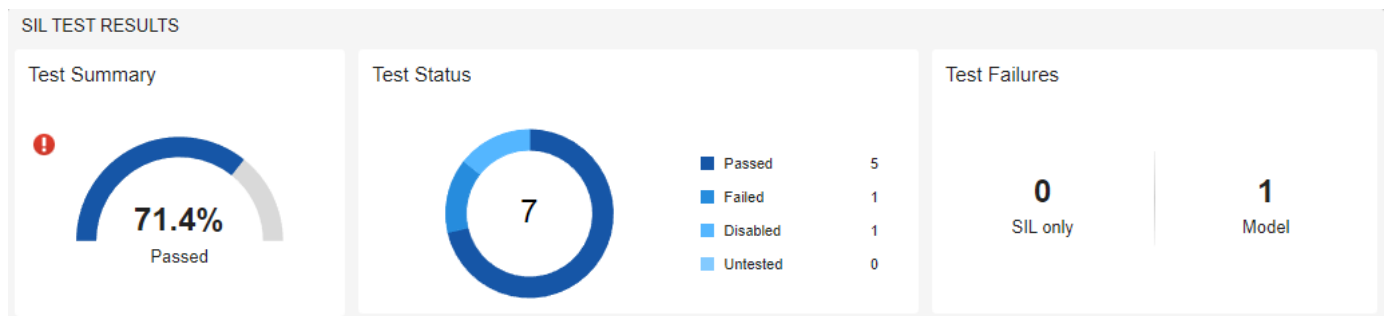


Review Test Status and Coverage Results

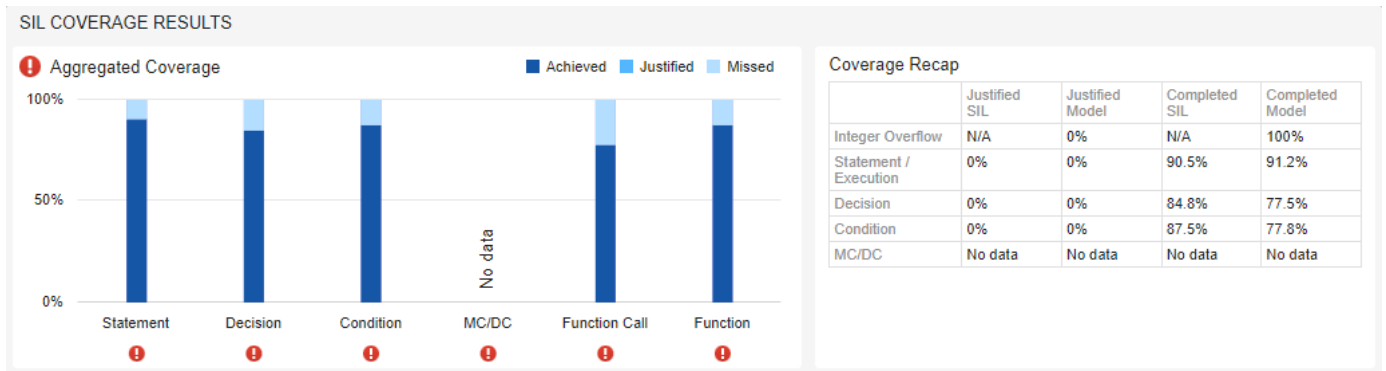
These images show example results in the SIL Code Testing dashboard for the unit `cc_DriverSwRequest` after running the test cases in both Normal and Software-in-the-loop simulation modes.

The dashboard is split into two main sections that contain individual widgets showing metric results:

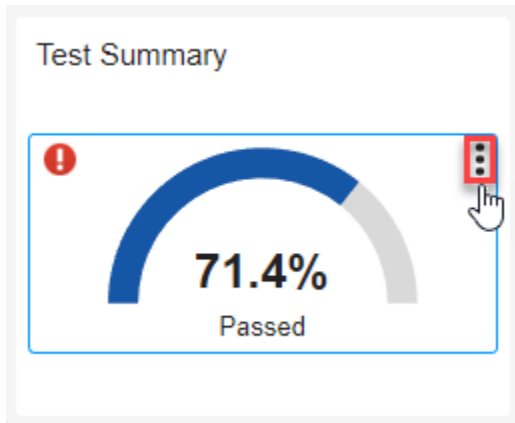
- **Test Results**




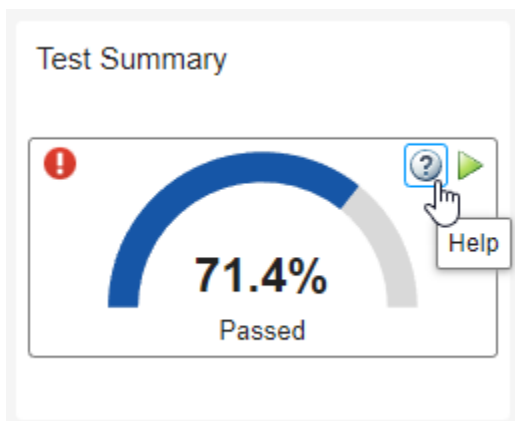
- **Coverage Results**



To view information about the metric results in the widget, point to the widget. Three dots appear in the top-right corner of the widget.



Point to the three dots and click the Help  icon to view information about the metric and how the dashboard calculates the metric results.



To explore metric results in more detail, click on the widget itself. A table lists the artifacts in the unit and their results for the metric. The table provides hyperlinks to open each artifact so that you can view details about the artifact and address testing quality issues.

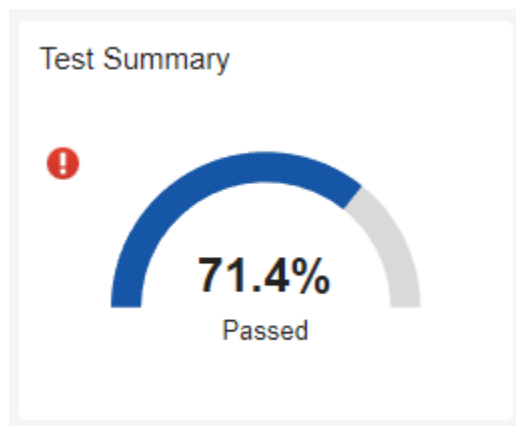
Assess Test Results


The **Test Results** section shows metric results for code testing results. For information on how to collect and fix code testing results, see “Identify and Troubleshoot Gaps in Code Testing Results and Coverage” on page 5-192.

Test Summary

For code testing results to be compliant, 100% of code tests must pass.

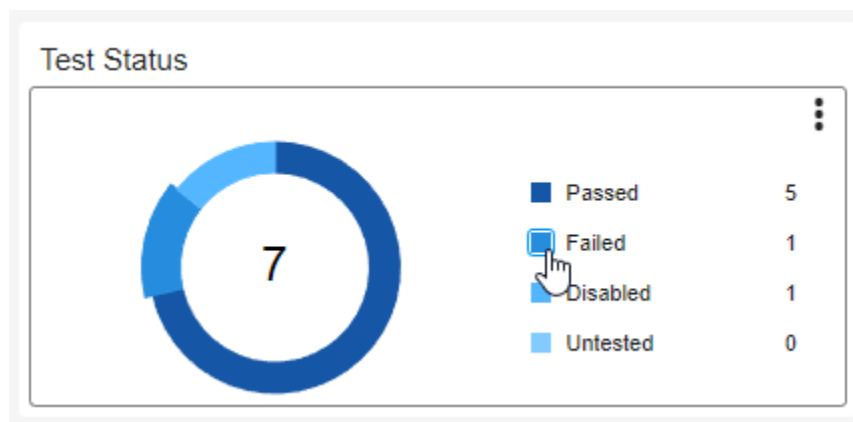
You can view the percentage of tests that pass in the **Test Summary** widget.




If less than 100% of tests pass, the widget shows the red **Non-Compliant** overlay icon . Use the **Test Status**, **SIL only**, and **Model** widgets to identify and fix testing issues.

Test Status

Use the **Test Status** widget to identify why specific tests did not pass. The **Test Status** widget shows the number of tests that passed, failed, are disabled, or are untested.



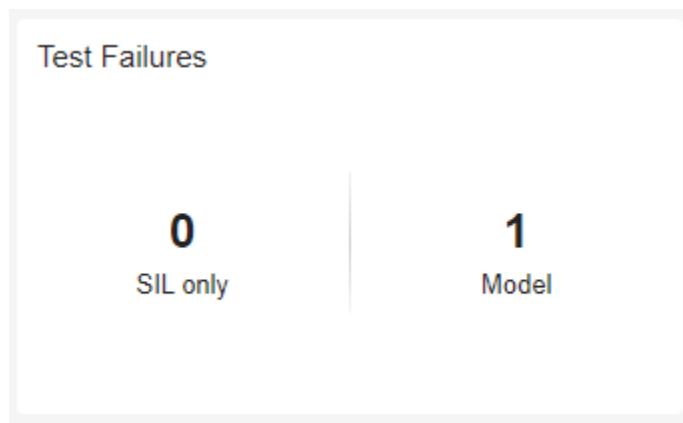
You can click on the icons in the widget to explore the metric results in more detail. For example, if you click on the icon next to **Untested**  you can view detailed metric results. The results include a table that lists the untested tests in the unit and provides hyperlinks to open the associated test files.

To address these issues, run the untested tests, confirm that any disabled tests can remain disabled, and address any failed tests.

Test Failures

Use the **Test Failures** group to identify which failures only occurred in either model testing or code testing. The group contains two widgets:

- **SIL only** — Number of tests that passed during model testing but failed during SIL testing. If the **SIL only** widget shows a number greater than 0, investigate and fix the code testing failures before analyzing coverage results.
- **Model** — Number of tests that failed during model testing. If the **Model** widget shows a number greater than 0, investigate and fix the issue using the Model Testing Dashboard. You should try to address model-only failures through model testing and not code testing.



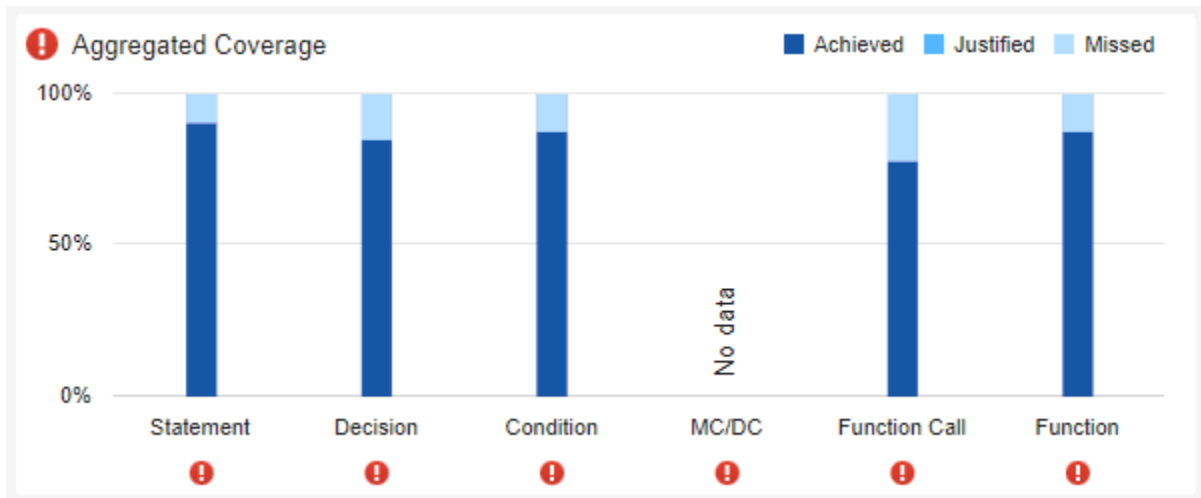
Analyze Coverage Results

The **Coverage Results** section shows metric results for code coverage results. For information on how to collect and fix code coverage results, see “Identify and Troubleshoot Gaps in Code Testing Results and Coverage” on page 5-192.

Aggregated Coverage

For code coverage to be compliant, 100% of the aggregated coverage must be completed for the coverage types that your unit testing requires. To determine your required coverage types, consider the safety level of your software unit. The dashboard considers coverage completed if the coverage is either achieved by code testing or justified using coverage filters.

You can view the aggregated statement, decision, condition, MC/DC, function call, and function coverages for the unit in the **Aggregated Coverage** widget.



If any coverage type has less than 100% coverage, the widget shows the red **Non-Compliant** overlay icon . Use the **Coverage Recap** widget to identify and fix coverage issues.

Coverage Recap

Use the **Coverage Recap** widget to see the justified and completed coverage for both model and code testing. The dashboard considers coverage to be *completed* if the coverage was either achieved or justified.

	Justified SIL	Justified Model	Completed SIL	Completed Model
Integer Overflow	N/A	0%	N/A	100%
Statement / Execution	0%	0%	90.5%	91.2%
Decision	0%	0%	84.8%	77.5%
Condition	0%	0%	87.5%	77.8%
MC/DC	No data	No data	No data	No data

Use this widget to help you identify issues across model and code testing results. For example:

- You can identify if there were tests that you justified during model testing but did not justify for SIL code testing.
- If there was a gap in integer overflow coverage during model testing, you need to go back and address that gap in the model testing. Use the Model Testing Dashboard to identify and fix the source of the missing coverage. You can see integer overflow coverage in the **Aggregated Coverage** widget in the Model Testing Dashboard.

See Also

“Code Testing Metrics”

Related Examples

- “Identify and Troubleshoot Gaps in Code Testing Results and Coverage” on page 5-192
- “Collect Code Testing Metrics Programmatically” on page 5-201

Identify and Troubleshoot Gaps in Code Testing Results and Coverage

You can identify and troubleshoot common issues in code testing results and coverage by using the SIL Code Testing and PIL Code Testing dashboards. After you complete model testing, you can use the dashboards to assess the status of your software-in-the-loop (SIL) and processor-in-the-loop (PIL) testing. The dashboards use metrics to measure the quality and completeness of test results and coverage and summarize the metric results in widgets. When you click on a widget, you can view detailed metric results and use hyperlinks to directly open artifacts with issues.

Use the code testing dashboards after completing model testing with the Model Testing Dashboard to help demonstrate equivalence between software units and their associated code.

This example shows how to assess the status of SIL testing by using the SIL Code Testing dashboard, but you follow the same steps to analyze PIL testing in the PIL Code Testing dashboard. The PIL Code Testing dashboard uses the same layout as the SIL Code Testing dashboard, but the metric results come from PIL tests.

Check Completeness of Model Testing Results

1. Open a project that contains the models and testing artifacts that you want to analyze. Open an example project by entering this code in the MATLAB® Command Window:

```
dashboardCCProjectStart("incomplete")
```

This command creates a copy of the dashboard example project in your example projects directory. The example project contains several models and tests, but does not include any test results.

2. Get model and SIL code testing results by running a test in Normal and Software-in-the-Loop (SIL) mode, respectively. Typically, you run each of your tests in both simulation modes. For this example, run this code:

```
% get test case "Detect set"
cp = currentProject;
rf = cp.RootFolder;
tf = fullfile(rf,"tests","cc_DriverSwRequest_Tests.mldatx");
tfObj = sltest.testmanager.load(tf);
tsObj = getTestSuites(tfObj);
tcObj = getTestCases(tsObj);
tc3 = tcObj(3);

% run as model test (SimulationMode = 'Normal')
run(tc3,SimulationMode='Normal');

% run as code test (SimulationMode = 'Software-in-the-Loop (SIL)')
run(tc3,SimulationMode='Software-in-the-loop (SIL)');
```

The code gets the test case `Detect set` from the example project, runs the test using the Normal simulation mode, and then runs the same test again using the Software-in-the-Loop (SIL) simulation mode. For equivalence testing (also called back-to-back testing), it is important to use the same test in both the model testing and code testing environments, which is why the code only changes the value of the `SimulationMode` argument when running the test case. For more information about the `SimulationMode` argument, see `run (Simulink Test)`.

3. Check the status of model testing results by using the Model Testing Dashboard. To open the dashboard: on the **Project** tab, click **Model Testing Dashboard** or enter:

```
modelTestingDashboard
```

You can use the Model Testing Dashboard to view an overview of the model testing results and the compliance status for each software unit in your project.

The Model Testing Dashboard has four main sections: the toolstrip, the **Project** panel, the **Artifacts** panel, and the dashboard tab. In the **Project** panel, the dashboard organizes software units under the components that contain them in the model hierarchy. You can click on a unit in the **Project** panel to view the metric results for that unit in the dashboard.

4. For this example, view the metric results for the software unit `cc_DriverSwRequest`. In the **Project** panel, click **cc_DriverSwRequest**.

The Model Testing Dashboard shows the metric results for `cc_DriverSwRequest`. In the **Simulation Test Result Analysis** section of the dashboard, under **Model Test Status**, you can see that the 1 test has the status **Failed** and 6 tests have the status **Untested**. Before analyzing the code testing results, you typically run these untested model tests and fix the failing results. But for this example, do not fix the failing test and continue to the code testing results. For information on how to run tests and fix failing test results using the Model Testing Dashboard, see “Fix Requirements-Based Testing Issues” on page 5-89.

View Status of Code Testing and Address Non-Compliant Results

1. View an overview of the SIL code testing results for the software unit `cc_DriverSwRequest` by opening the SIL Code Testing dashboard. In the toolstrip, in the **Add Dashboard** section, click **SIL Code Testing**.

The SIL Code Testing dashboard opens in a new tab next to the tab for the Model Testing Dashboard. The SIL Code Testing dashboard contains two main sections: **SIL Test Results** and **SIL Coverage Results**.

For this example, notice that in the **SIL Test Results** section, the **Test Summary** widget shows that 0% of SIL tests have the status **Passed**. If you look at the **Test Status** widget, you can see that 1 SIL test has the status **Failed** and 6 SIL tests have the status **Untested**. Under **Test Failures**, the **SIL only** and **Model** widgets show that 0 SIL tests failed only in SIL code testing and that 1 SIL test failed during both model testing and SIL code testing. The **Model** widget in the SIL Code Testing dashboard shows the same metric information as the **Failed** widget in the Model Testing Dashboard, so if there are any model testing failures, you can also see that information directly from the SIL Code Testing dashboard.

The **Test Summary** widget shows the **Non-Compliant** icon  if fewer than 100% of SIL tests pass.

2. Fix the source of the test failure. In this example project, the `Detect` set test failed both model and SIL code testing because the model `cc_DriverSwRequest.slx` uses the wrong enumerated value. To fix the issue, run this code:

```
load_system(fullfile(rf, "models", "cc_DriverSwRequest.slx"));
set_param("cc_DriverSwRequest/Const_reqMode_Set", "Value", "db_Request_Enum.SET")
close_system("cc_DriverSwRequest.slx", 1);
```

The code loads the model, updates the value, and closes the model.

3. Retest the model by rerunning this code:

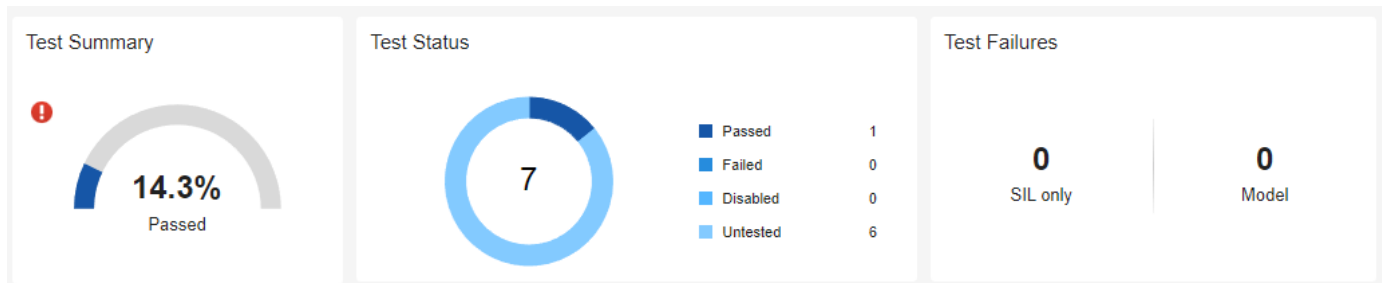
```
% get test case "Detect set"
cp = currentProject;
rf = cp.RootFolder;
tf = fullfile(rf,"tests","cc_DriverSwRequest_Tests.mldatx");
tfObj = sltest.testmanager.load(tf);
tsObj = getTestSuites(tfObj);
tcObj = getTestCases(tsObj);
tc3 = tcObj(3);

% run as model test (SimulationMode = 'Normal')
run(tc3,SimulationMode='Normal');

% run as code test (SimulationMode = 'Software-in-the-Loop (SIL)')
run(tc3,SimulationMode='Software-in-the-loop (SIL)');
```

4. In the SIL Code Testing dashboard, click **Collect** and inspect the updated metric results in the dashboard.

The **Test Summary** widget shows that now 14.3% of SIL tests have the status **Passed**. Under **Test Failures**, the **Model** widget shows that there are currently 0 failures in the model testing results.



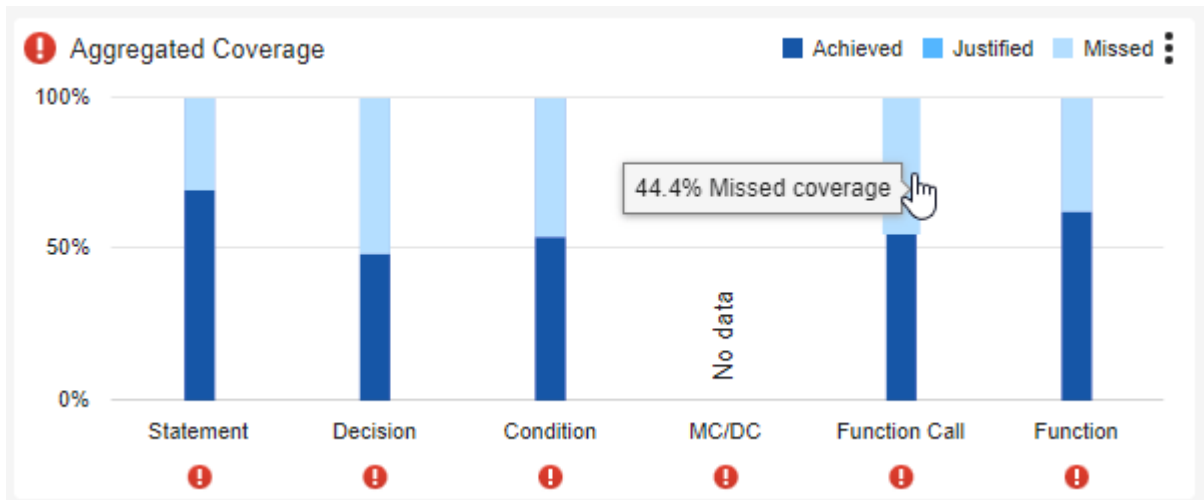
Before analyzing the SIL coverage results, you typically run any untested tests and fix any failing test results. But for this example, leave the tests untested and continue to analyze the results in the dashboard.

Identify Non-Compliant Coverage Results

In the **SIL Coverage Results** section, the **Aggregated Coverage** widget shows statement, decision, condition, MC/DC, function call, and function coverage aggregated from across the unit. For compliance, 100% of aggregated coverage must be completed, either by achieving the coverage through code testing or justifying the coverage using coverage filters.

Identify what percentage of each coverage type was achieved, justified, or missed. If the **Aggregated Coverage** widget shows that you are missing coverage, consider adding more tests or justifying the missing coverage.

For this example, point to the two **Function Call** bars in the bar chart. The unit achieved 55.6% of function call coverage, but missed 44.4% of function call coverage during SIL code testing.



If the **Aggregated Coverage** widget shows that you are missing **Function Call** or **Function** coverage, you need to address those coverage gaps by running any untested tests, adding additional code tests, or justifying the missing coverage. **Function Call** and **Function** coverage are specific to code testing and there is no direct model testing equivalent that you can change to correct this missing coverage.

Investigate Gaps in Coverage

To investigate coverage gaps in more detail, use the **Coverage Recap** widget to identify gaps and find unexpected differences in model and code coverage results. Note that the dashboard considers coverage to be *completed* if the coverage was either achieved or justified.

When you read the table in the **Coverage Recap** widget:

- Check that 100% of integer overflow coverage completed during model testing.
- See coverage types that have better code coverage than the model coverage.
- Check that 100% of model coverage completed during model testing.
- See coverage types that have better model coverage than code coverage.
- Check that 100% of code coverage completed during code testing.

Identify Missing Integer Overflow Coverage

Inspect the integer overflow coverage for the model by using the **Integer Overflow** row.

The first row of the **Coverage Recap** table shows the **Integer Overflow** coverage that is justified or completed during model and code testing. **Integer Overflow** coverage is always **N/A** for code testing results because integer overflow coverage is specific to model testing. In this example, the **Completed Model** column shows 100% integer overflow coverage.

	Justified SIL	Justified Model	Completed SIL	Completed Model
Integer Overflow	N/A	0%	N/A	100%
Statement / Execution	0%	0%	69.5%	79.4%
Decision	0%	0%	48.5%	42.5%
Condition	0%	0%	54.2%	41.7%
MC/DC	No data	No data	No data	No data

If the **Integer Overflow** row shows less than 100% **Completed Model** coverage, you need to run any untested tests, add model tests, or justify the missing model coverage. Address gaps in **Integer Overflow** coverage through model testing and not code testing. You can review the model testing results in the Model Testing Dashboard and troubleshoot missing integer overflow coverage using the **Int. Overflow** bar in the **Model Coverage** widget. For more information, see “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80.

Identify Other Sources of Missing Model Coverage

Identify any coverage types that have better code coverage than the model coverage.

The **Completed SIL** column shows completed code coverage.

	Justified SIL	Justified Model	Completed SIL	Completed Model
Integer Overflow	N/A	0%	N/A	100%
Statement / Execution	0%	0%	69.5%	79.4%
Decision	0%	0%	48.5%	42.5%
Condition	0%	0%	54.2%	41.7%
MC/DC	No data	No data	No data	No data

If the **Completed SIL** percentage is greater than the **Completed Model** percentage for a coverage type, the code might contain functionalities that you can test by code testing but you did not fully test during model testing. For example, when you generate code, the model can create reused functions that the existing code tests can test more fully than the model tests. In those instances, you might need to justify the model coverage or add tests that more fully test the upstream modeling constructs in the code.

Inspect the state of model coverage by using the **Completed Model** column.

The **Completed Model** column shows the overall status of model coverage for each coverage type. In this example, the **Statement/Execution**, **Decision**, and **Condition** rows show that there was less than 100% coverage of these coverage types achieved or justified during model testing. The tests did not collect MC/DC coverage results, so the **MC/DC** row shows **No data**.

	Justified SIL	Justified Model	Completed SIL	Completed Model
Integer Overflow	N/A	0%	N/A	100%
Statement / Execution	0%	0%	69.5%	79.4%
Decision	0%	0%	48.5%	42.5%
Condition	0%	0%	54.2%	41.7%
MC/DC	No data	No data	No data	No data

If the **Completed Model** column shows less than 100% coverage for any coverage type, you need to review and address the missing coverage through model testing. You can review the model testing results in the Model Testing Dashboard and troubleshoot missing coverage by using the **Model Coverage** widget. For more information, see “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80.

For this example, leave the model coverage incomplete and continue to analyze the results in the dashboard.

Identify Where Code Coverage Is Less Than Model Coverage

Suppose you add tests to address the missing model coverage and the **Model Coverage** column shows improved model coverage results.

Inspect the rows where the code coverage in the **Completed SIL** column is less than the model coverage in the **Completed Model** column.

	Justified SIL	Justified Model	Completed SIL	Completed Model
Integer Overflow	N/A	0%	N/A	100%
Statement / Execution	0%	0%	69.5%	79.4%
Decision	0%	0%	48.5%	42.5%
Condition	0%	0%	54.2%	41.7%
MC/DC	No data	No data	No data	No data

If the **Completed SIL** percentage is less than the **Completed Model** percentage for a coverage type, there might be functionalities in the generated code that your existing tests do not cover. For example, if your generated code uses a code replacement library, lookup table, or other code-specific functionality, your existing tests might cover your model, but do not fully test those coding constructs. You might need to add tests that directly test these generated coding constructs or justify the missing coverage.

Identify Other Sources of Missing Code Coverage

Inspect the rows where code coverage is less than 100%.

The **Completed SIL** column shows the overall status of code coverage for each coverage type.

	Justified SIL	Justified Model	Completed SIL	Completed Model
Integer Overflow	N/A	0%	N/A	100%
Statement / Execution	0%	0%	69.5%	79.4%
Decision	0%	0%	48.5%	42.5%
Condition	0%	0%	54.2%	41.7%
MC/DC	No data	No data	No data	No data

If the **Completed SIL** column still shows less than 100% coverage, check the **Justified SIL** and **Justified Model** columns to see if you have more justifications for model coverage than for code coverage.

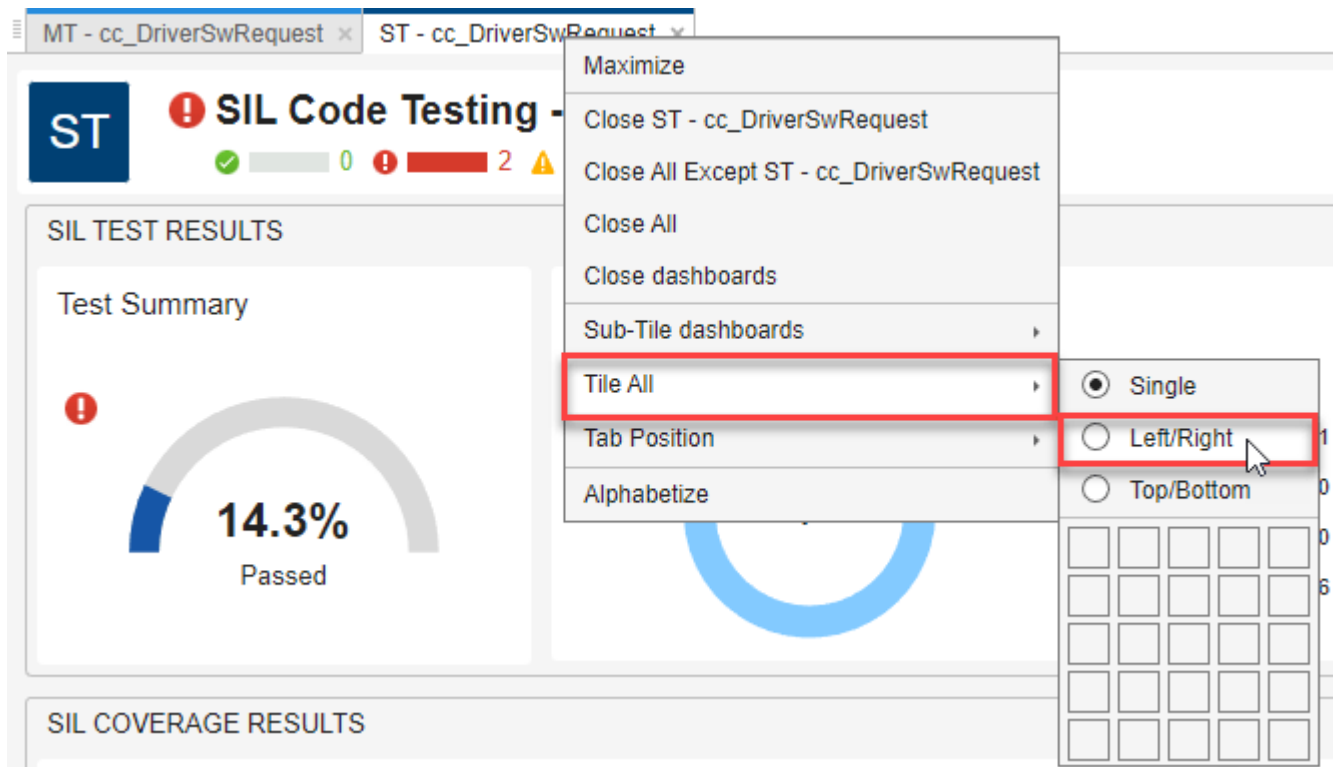
	Justified SIL	Justified Model	Completed SIL	Completed Model
Integer Overflow	N/A	0%	N/A	100%
Statement / Execution	0%	0%	69.5%	79.4%
Decision	0%	0%	48.5%	42.5%
Condition	0%	0%	54.2%	41.7%
MC/DC	No data	No data	No data	No data

If **Justified Model** percentage is greater than **Justified SIL** percentage for a coverage type, you might need to update your coverage filter to apply the justifications you made in the model coverage to your code coverage.

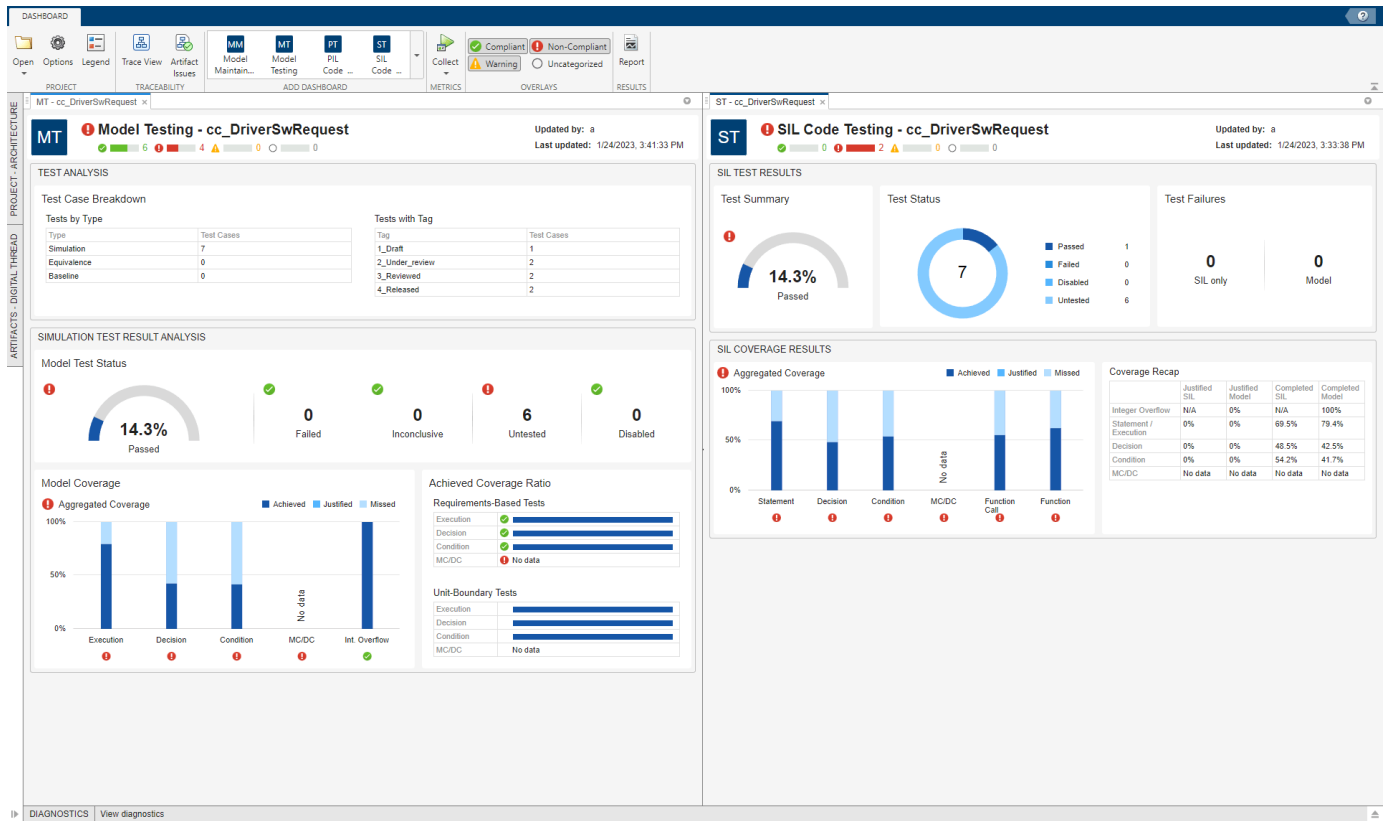
See Results for Multiple Dashboards

When you review the model and code testing dashboards, it is helpful to have the Model Testing dashboard tab open next to the SIL Code Testing dashboard tab. You can reduce the amount of information shown in the Model Testing tab and rearrange the dashboard tabs to see the Model Testing dashboard tab and SIL Code Testing dashboard tab are side-by-side.

1. In the toolstrip, click **Options** and select **Hide requirements metrics**. Click **Apply** to hide the **Test Analysis** section for the Model Testing dashboard. Any open dashboards automatically close.
2. Re-open the Model Testing dashboard for `cc_DriverSwRequest` by clicking **Model Testing** in the dashboards gallery and selecting `cc_DriverSwRequest` in the **Project** panel.
3. Re-open the SIL Code Testing dashboard for `cc_DriverSwRequest` by clicking **SIL Code Testing** in the dashboards gallery and selecting `cc_DriverSwRequest` in the **Project** panel.
4. To create more space for the dashboard tabs, right-click a tab and select **Maximize** to hide the **Project** and **Artifact** panels.
5. Right-click the **SIL Code Testing** tab (shown as **ST - cc_DriverSwRequest**) and select **Tile All > Left/Right**.



Now you can see the model testing and code testing dashboards side-by-side.



See Also

“Code Testing Metrics”

Related Examples

- “View Status of Code Testing Activities for Software Units in Project” on page 5-184
- “Collect Code Testing Metrics Programmatically” on page 5-201

Collect Code Testing Metrics Programmatically

This example shows how to programmatically assess the status and quality of code testing activities in a project. You can assess the testing status of a unit by using the metric API to collect metric data on the test status and coverage results. After collecting metric results, you can access the results or export them to a file. By running a script that collects these metrics, you can automatically analyze the testing status of your project to, for example, design a continuous integration system. Use the results to monitor testing completeness or to detect downstream testing impacts when you make changes to artifacts in the project.

Run Model and Code Tests

Open a project that contains models and testing artifacts. For this example, in the MATLAB Command Window, enter:

```
dashboardCCProjectStart("incomplete")
```

The example project contains models and tests for the models. The example project also has the project setting **Track tool outputs to detect outdated results** enabled. Before you programmatically collect metrics, make sure that the **Track tool outputs to detect outdated results** setting is enabled for your project. For information, see “Enable Artifact Tracing for the Project” on page 5-128.

For this example, run a test case in Normal mode and then in Software-in-the-loop (SIL) mode by entering:

```
cp = currentProject;
rf = cp.RootFolder;
tf = fullfile(rf,"tests","cc_DriverSwRequest_Tests.mldatx");
tfObj = sltest.testmanager.load(tf);
tsObj = getTestSuites(tfObj);
tcObj = getTestCases(tsObj);
tc3 = tcObj(3);
run(tc3,SimulationMode='Normal');
run(tc3,SimulationMode='Software-in-the-loop (SIL)');
```

Collect Metric Results For Software Units in Project

Create a `metric.Engine` object for the current project.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to reflect pending artifact changes and to track the test results.

```
updateArtifacts(metric_engine);
```

Create an array of metric identifiers for the metrics you want to collect. For this example, create a list of the metric identifiers used in the SIL Code Testing dashboard by specifying the Dashboard as "ModelUnitSILTesting". For more information, see `getAvailableMetricIds`.

```
metric_Ids = getAvailableMetricIds(metric_engine,...
App = "DashboardApp",...
Dashboard="ModelUnitSILTesting");
```

For a list of code testing metrics, see “Code Testing Metrics”.

Execute the metric engine to collect the metric results.

```
execute(metric_engine, metric_Ids);
```

By default, the metric engine collects results for each unit in the project. If you only want to collect results for a single unit, specify the unit using the `ArtifactScope` argument. For more information, see `execute`.

Access Results

After you collect metric results, you can access the results programmatically or generate a report for offline review.

View Test and Coverage Results Programmatically

To access the results programmatically, use the `getMetrics` function. The function returns the `metric.Result` objects that contain the result data for the specified unit and metrics. For this example, store the results for the metrics `slcomp.sil.TestStatusDistribution` and `slcomp.sil.CoverageBreakdown` in corresponding arrays.

```
results_silTests = getMetrics(metric_engine, "slcomp.sil.TestStatusDistribution");
results_silCoverage = getMetrics(metric_engine, "slcomp.sil.CoverageBreakdown");
```

The metric `slcomp.sil.TestStatusDistribution` returns a distribution of the number of tests that passed, failed, were disabled, or were untested. For this example, the distribution values for the unit `cc_DriverSwRequest` were in `results_silTests(4).Value`. The order of the units in your results may be different on your machine.

```
results_silTests(4).Value
```

```
ans =
```

```
struct with fields:
    BinCounts: [4×1 double]
    BinEdges: [4×1 double]
    OverallCount: 7
    Ratios: [4×1 double]
```

Use the `disp` function to display the bin counts of the distribution, which are fields in the `Value` field of the `metric.Result` object.

```
disp(newline)
disp(['Unit: ', results_silTests(4).Scope(1).Name])
disp([' ', num2str(results_silTests(4).Value.BinCounts(1)), ' SIL test FAILED'])
disp([' ', num2str(results_silTests(4).Value.BinCounts(2)), ' SIL tests PASSED'])
disp([' ', num2str(results_silTests(4).Value.BinCounts(3)), ' SIL tests DISABLED'])
disp([' ', num2str(results_silTests(4).Value.BinCounts(4)), ' SIL tests UNTESTED'])
```

```
Unit: cc_DriverSwRequest
 1 SIL test FAILED
 0 SIL tests PASSED
 0 SIL tests DISABLED
 6 SIL tests UNTESTED
```

This result shows that for the unit `cc_DriverSwRequest`, one SIL test failed and six SIL tests remain untested. For code testing results to be compliant, each of the tests should have passed. If one

or more tests are untested, disabled, or failed, those issues should be addressed before you analyze the code coverage results. For information on how to address the test failure in this example project, see “Identify and Troubleshoot Gaps in Code Testing Results and Coverage” on page 5-192.

The metric `slcomp.sil.CoverageBreakdown` returns the percentage of coverage achieved, justified, completed, or missed for each coverage type. For this example, the coverage results for the unit `cc_DriverSwRequest` were in `results_silCoverage(4)`:

```
results_silCoverage(4).Value
```

```
ans =
```

```
struct with fields:
    Statement: [1x1 struct]
    Decision: [1x1 struct]
    Condition: [1x1 struct]
    MCDC: [1x1 struct]
    Function: [1x1 struct]
    FunctionCall: [1x1 struct]
```

You can access the results for each coverage type using the fields in the `Value` field of the `metric.Result` object. For example, to access the SIL decision coverage results for the unit:

```
decisionCoverage = results_silCoverage(4).Value.Decision
```

```
decisionCoverage =
```

```
struct with fields:
    Achieved: 48.4848
    Justified: 0
    Missed: 51.5152
    AchievedOrJustified: 48.4848
```

This result shows that for the unit `cc_DriverSwRequest`, so far the SIL tests achieved 48% of decision coverage and missed 52% of decision coverage. For code coverage results to be compliant, 100% of test results for each coverage type should either be achieved or justified. When the coverage results for each coverage type have either been achieved or justified, the dashboard considers the coverage to be *completed*.

Generate Report For Offline Review

Generate a report file that contains the SIL testing results for each of the units in the project. For this example, specify the HTML file format, use `pwd` to provide the path to the current folder, and name the report `"SILResultsReport.html"`.

```
reportLocation = fullfile(pwd, "SILResultsReport.html");
generateReport(metric_engine, Type="html-file", Location=reportLocation, ...
App = "DashboardApp", Dashboard = "ModelUnitSILTesting");
```

The generated SIL Code Testing Report opens automatically.

To open the table of contents and navigate to results for each unit, click the menu icon in the top-left corner of the report. For each unit in the report, there is a summary of the SIL test results and coverage results.

2.2.2. Test Status

- 0 tests with status Passed
- 1 tests with status Failed
- 0 tests with status Disabled
- 6 tests with status Untested

2.2.3. Test Failures

0 of unit tests failed in SIL only

1 of unit tests failed during model testing

2.3. SIL Coverage Results

2.3.1. Aggregated Coverage

	Achieved	Justified	Missed
Statement	69.52%	0%	30.48%
Decision	48.48%	0%	51.52%
Condition	54.17%	0%	45.83%
MC/DC	No data	No data	No data
Function Call	55.56%	0%	44.44%
Function	62.50%	0%	37.50%

For more information on the report, see `generateReport`.

Saving the metric results in a report file allows you to access the results without opening the project and the dashboard. Alternatively, you can open the dashboard to see the results and explore the artifacts. In the MATLAB Command Window, enter:

```
modelTestingDashboard
```

In the **Add Dashboard** section of the toolstrip, click **SIL Code Testing** to open the SIL Code Testing dashboard.

See Also

`metric.Engine` | `execute` | `generateReport` | `getAvailableMetricIds` | `updateArtifacts`

Related Examples

- “View Status of Code Testing Activities for Software Units in Project” on page 5-184
- “Identify and Troubleshoot Gaps in Code Testing Results and Coverage” on page 5-192

Explore Traceability Information for Units and Components

Explore traceability relationships for design artifacts, requirements, tests, and results in your software units and components by using trace views in the dashboard. A *trace view* is an interactive diagram that shows a certain preset of traceability information for artifacts in a project. Trace views provide a detailed, tree-like structure of project artifacts and show trace relationships, individual artifact information, and a hierarchical view of trace relationships between the artifacts in a unit or component.

This example shows how to use trace views to see how artifacts trace to units and components in your design and view the traceability information for requirements, tests, and test results in the project. You can access these trace views from any open Dashboard window in the **View** section of the toolbar:

- **Design Dependency**
- **Requirement to Design**
- **Tests and Results**

The trace views provide traceability information within the context of software units and components in a project. To find required files for a whole project or to find required products and add-ons, use the Dependency Analyzer. For more information, see “Analyze Project Dependencies”.

Access Trace Views

- 1 Open the MATLAB Project that you want to analyze. For example, to open a copy of the dashboard example project, in the MATLAB Command Window, enter:

```
dashboardCCProjectStart
```

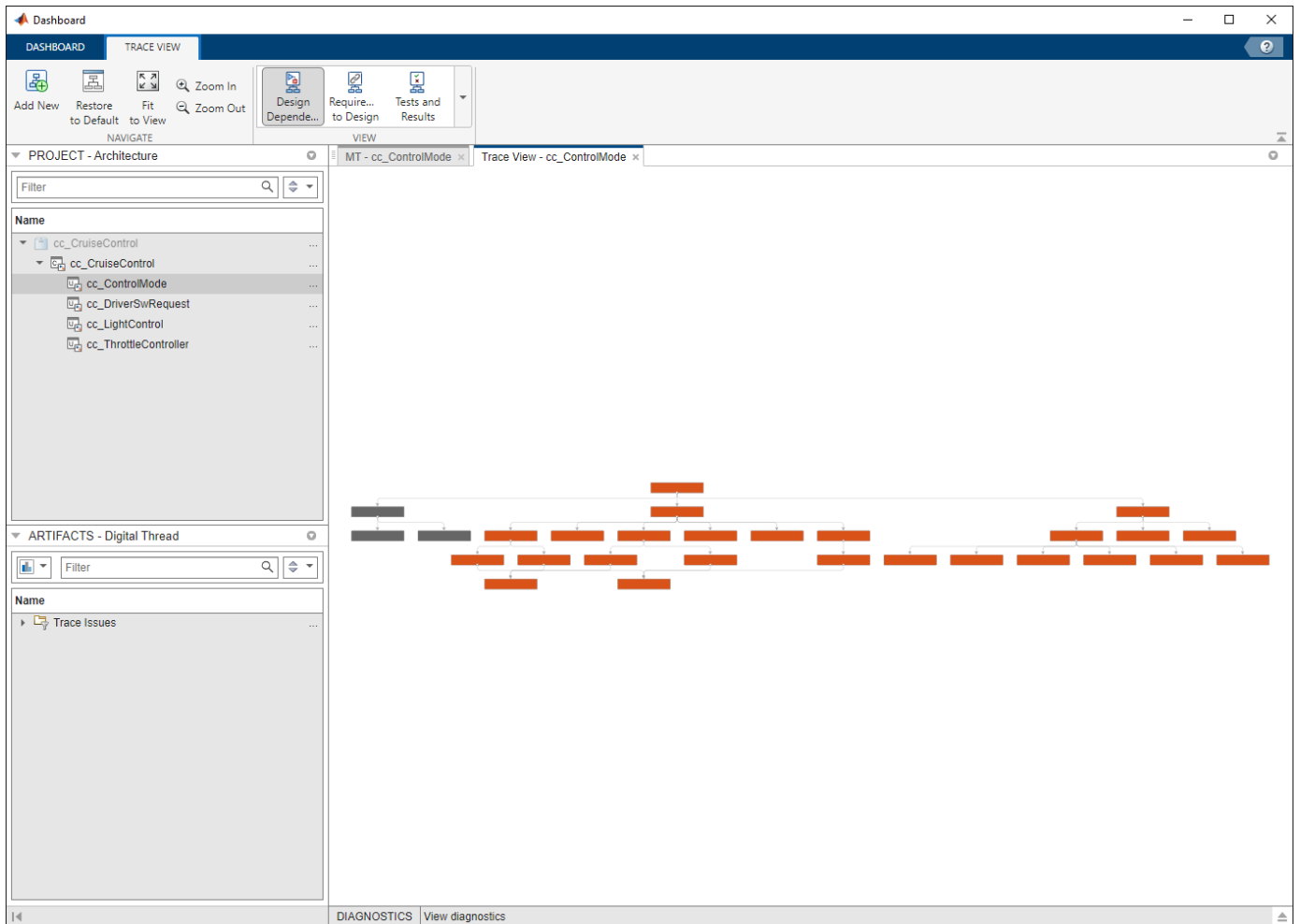
- 2 Open a dashboard. On the **Project** tab, click either:

- **Model Testing Dashboard**
- **Model Design Dashboard**

When you open a dashboard, the dashboard analyzes the project and collects information about the project artifacts, the artifact structure, and the traceability relationships between artifacts. The dashboard creates a digital thread that stores these attributes and unique identifiers for your project artifacts. The dashboard automatically populates the **Artifacts** panel with the artifacts that the digital thread traces to the software units and components in your project. For more information on the artifact analysis and tracing that the dashboard performs, see “Digital Thread” on page 5-167.

- 3 To open a trace view, first select a unit or component in the **Project** pane. For this example, select `cc_ControlMode`. Then click **Trace View** in the **Traceability** section of the toolbar.

If you do not see the units and components that you expect, see “Categorize Models in a Hierarchy as Components or Units” on page 5-119.

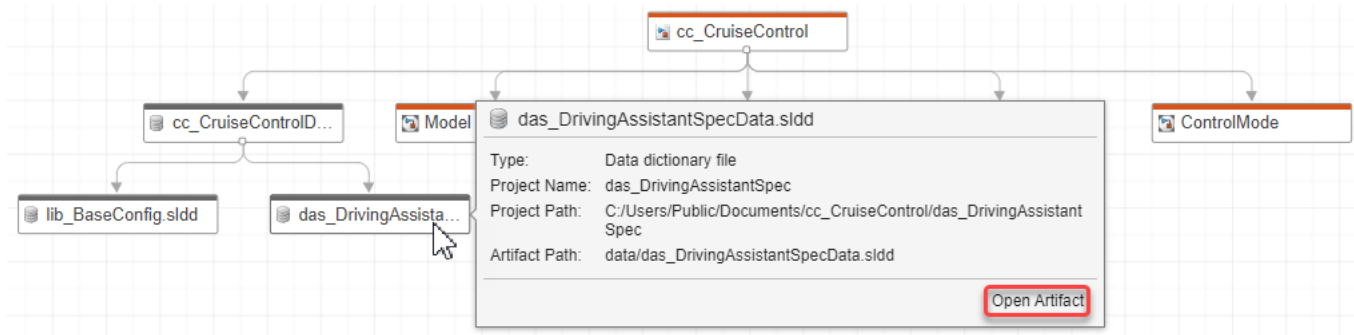


To navigate and explore the trace view:

- Zoom in or out by using the scroll wheel on your mouse or by using the **Zoom In** and **Zoom Out** buttons in the **Navigate** section of the toolbar.
- Pan vertically and horizontally by holding the space bar and clicking and dragging the mouse.
- Highlight the connection between artifacts by clicking the arrow between the artifacts. You can click an artifact or arrow to select it. You can also move the artifacts and arrows in the trace view to rearrange the diagram.
- Open a new trace view tab for the next unit or component listed in the **Project** panel by clicking **Add New** in the toolbar.
- Reset the diagram to the original layout and zoom level by clicking **Restore to Default** in the toolbar.

To view a tooltip with detailed information about an artifact, point to the artifact. You also see an **Open Artifact** button that you can use to open the artifact directly from the trace view.

By default, the dashboard opens the **Design Dependency** trace view, but you can use the other trace views to explore the traceability information for your project.

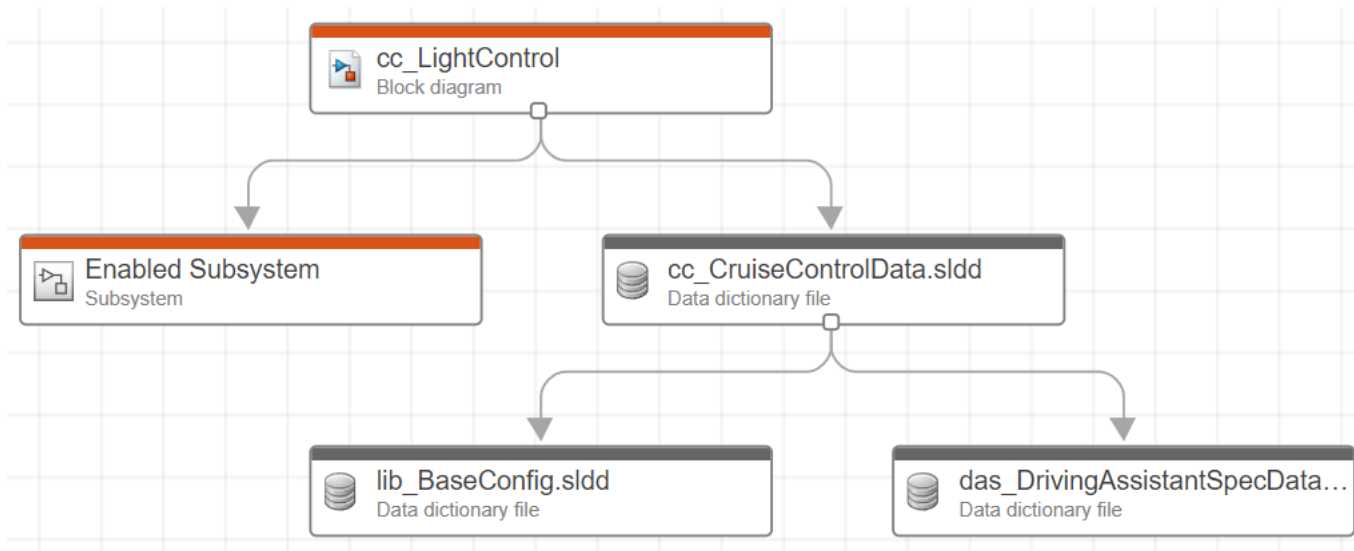


View Different Trace Views

View different traceability information by using the trace views gallery in the **View** section of the toolstrip. The gallery shows the names and descriptions of the trace views. Open a different trace view by clicking the name of a trace view in the trace views gallery. If a unit or component does not trace to any of the artifacts supported by the trace view that you select, select a different trace view from the trace views gallery.

Design Dependency Trace View

The **Design Dependency** trace view shows the library blocks, data dictionaries, model references, and MATLAB files that trace to the selected unit or component.

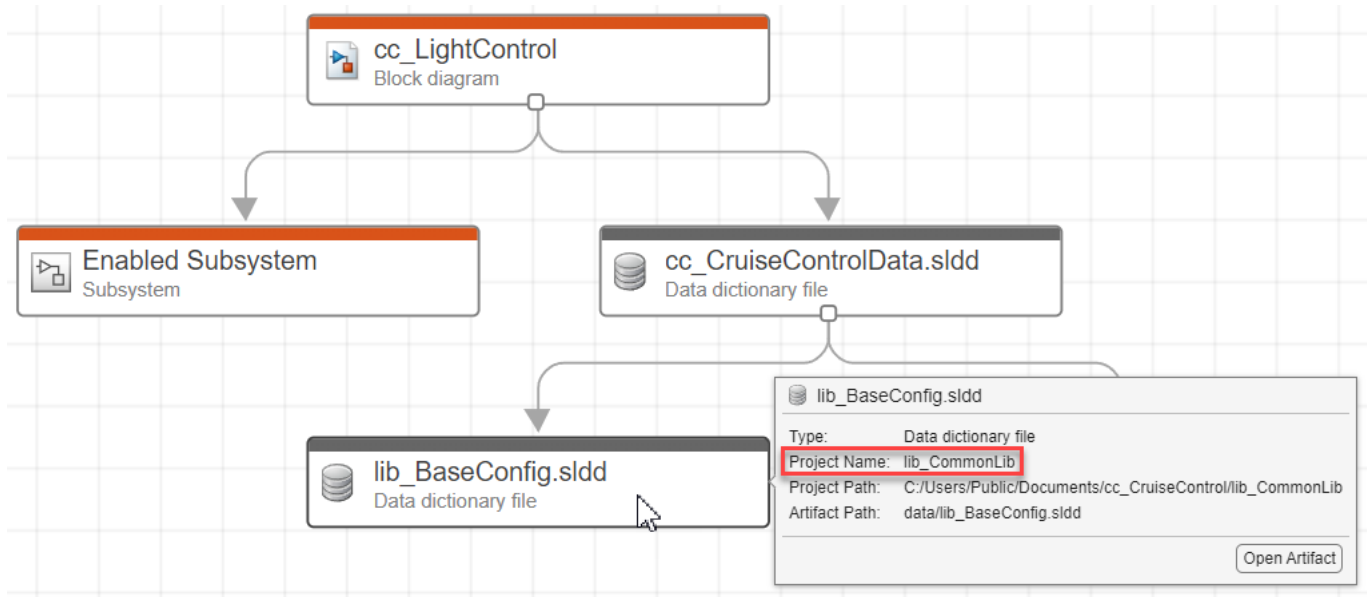


For example, the **Design Dependency** trace view for the example software unit `cc_LightControl` shows that the unit:

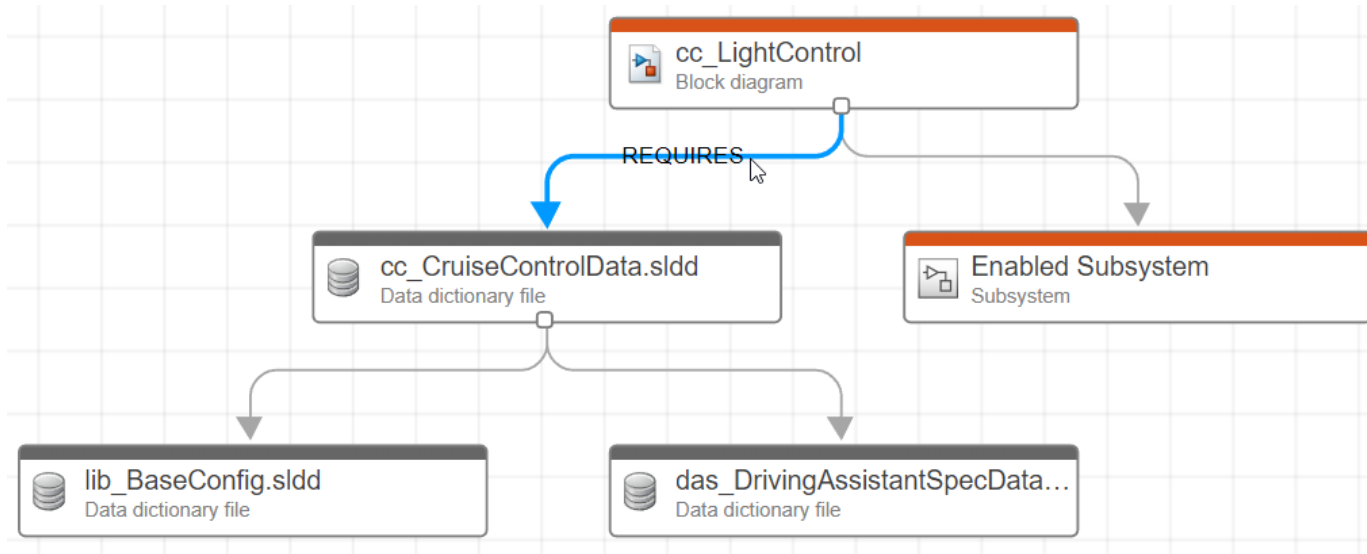
- Contains the subsystem `Enabled Subsystem`
- Requires the three data dictionary files: `cc_CruiseControlData.sldd`, `lib_BaseConfig.sldd`, and `das_DrivingAssistantSpecData.sldd`

You can use the trace view to examine the relationships that the dashboard identifies between design artifacts and to inspect component and unit dependencies. If you point to **`lib_BaseConfig.sldd`** in the

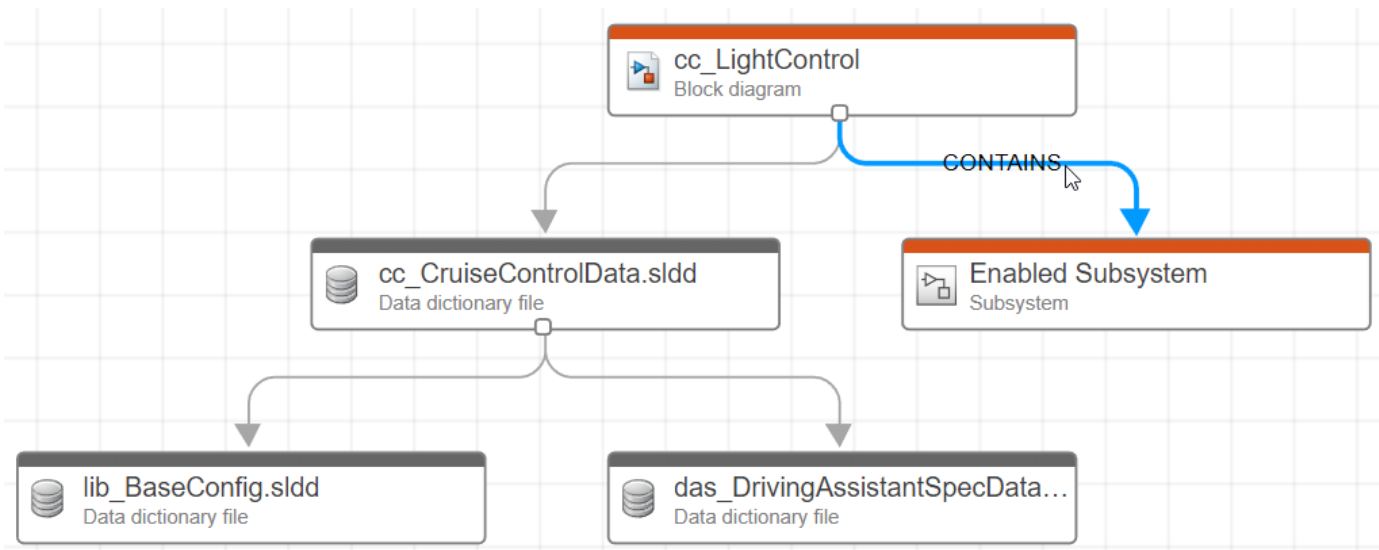
trace view, the tooltip shows that this data dictionary file belongs to the referenced project `lib_CommonLib`.



You can point to the arrow between artifacts to view the relationship type. For example, the unit `cc_LightControl` requires the data dictionary file `cc_CruiseControlData.sldd`.

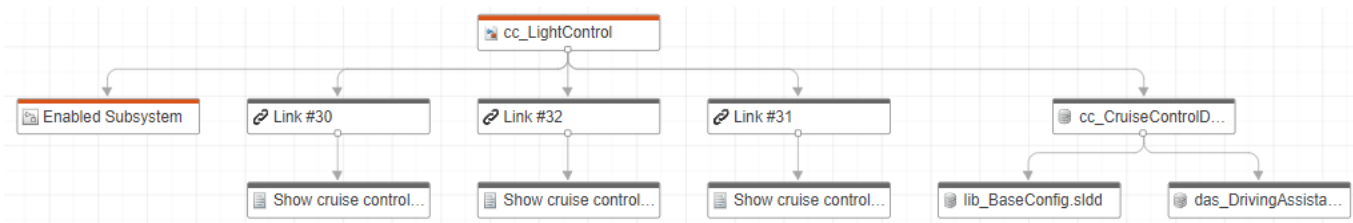


The unit `cc_LightControl` contains the subsystem `Enabled Subsystem`.

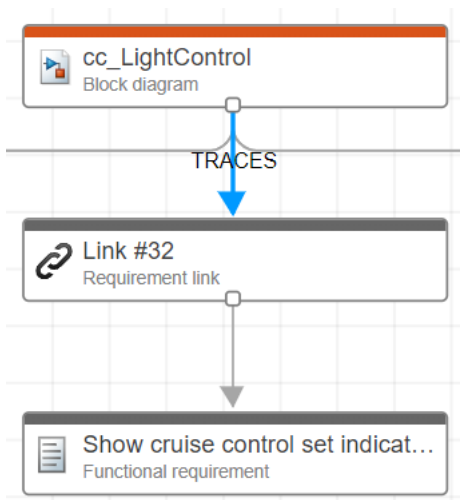


Requirement to Design Trace View

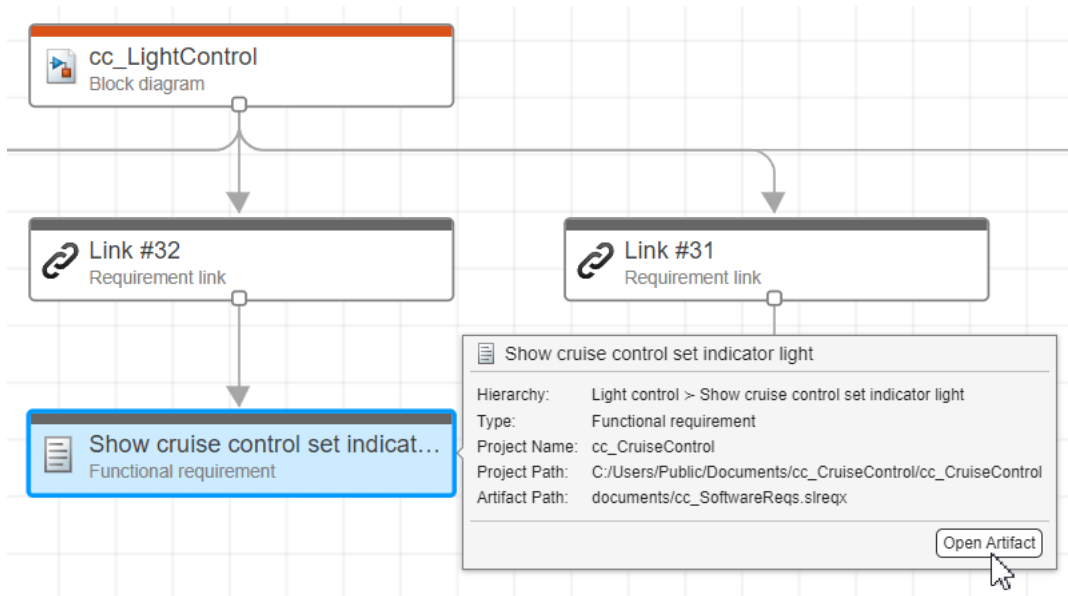
The **Requirement to Design** trace view shows the functional requirements that trace to the selected unit or component.



For example, the **Requirement to Design** trace view for the example software component `cc_LightControl` shows that 3 requirement links trace to 3 functional requirements: Show cruise control light indicator, Show cruise control set indicator light, and Show cruise control light when ignition switch is turned. The trace view shows that the dashboard traced the relationship from the block diagram for the unit, to the requirement link, and then to the functional requirement.

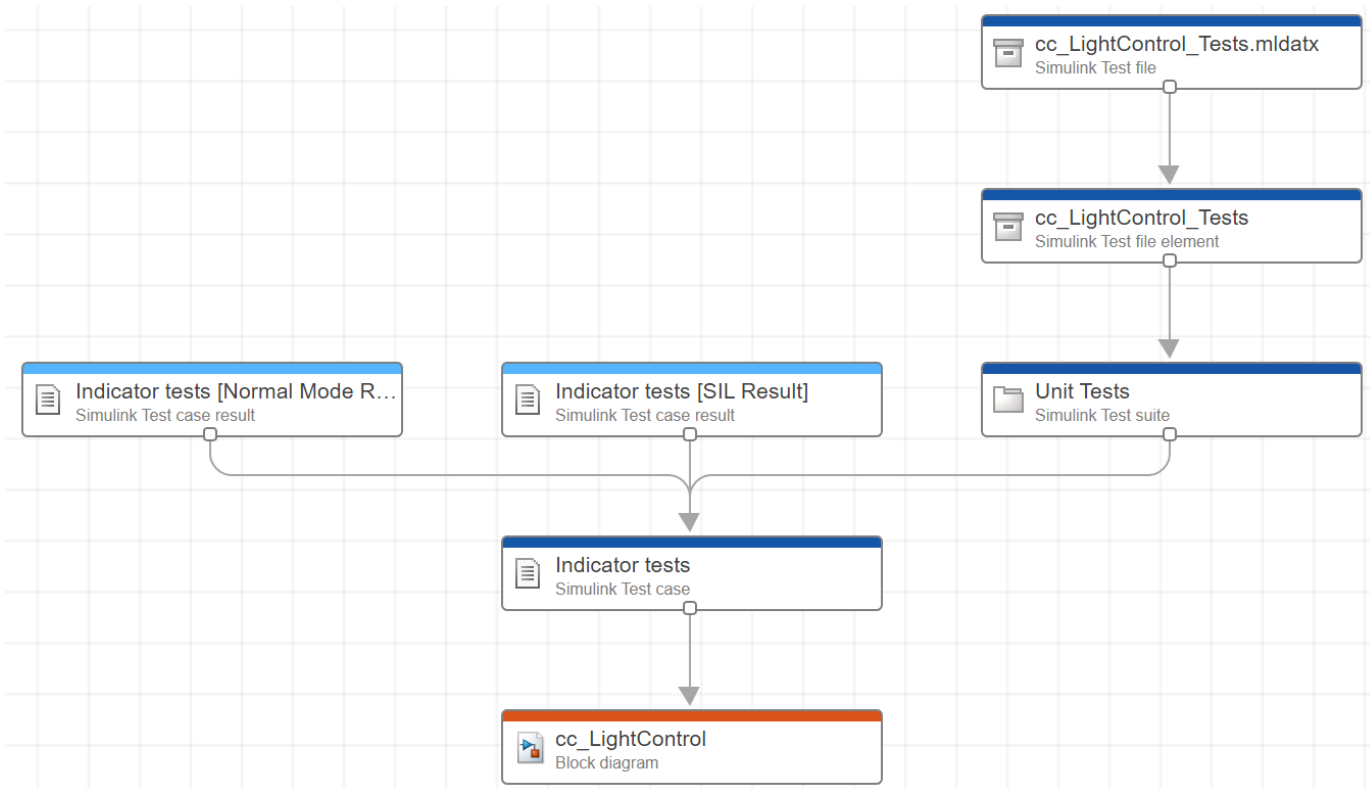


You can point to a functional requirement to see where the requirement is in the requirements hierarchy and use the **Open Artifact** button to open the requirement directly in the Requirements Editor.

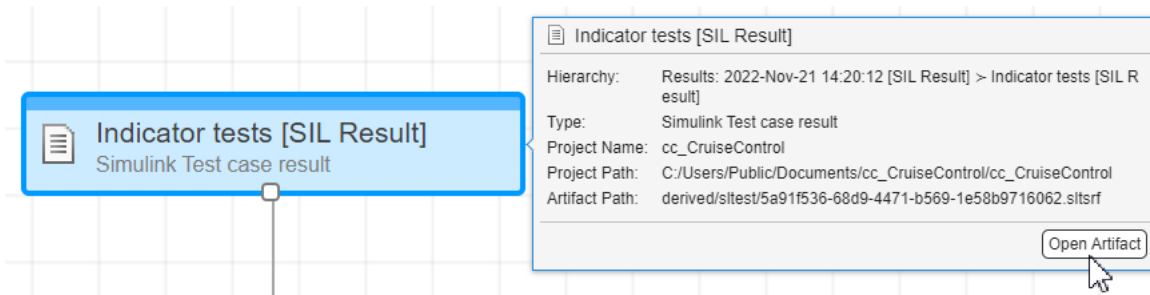


Tests and Results Trace View

The **Tests and Results** trace view shows the test cases and test results that trace to the selected unit or component. The trace view shows the Simulink Test files and test suites containing test cases that trace to the current unit or component. If you have test results associated with the test cases, the trace view traces those results to the test case and identifies whether the results are from model tests, software-in-the-loop (SIL) tests, or processor-in-the-loop (PIL) tests.



You can point to a Simulink Test case result and click **Open Artifact** to open the results directly in Test Manager.



Troubleshoot Missing Artifacts and Relationships

If you expect an artifact to appear in the trace view for a unit or component and it does not, see “Resolve Missing Artifacts, Links, and Results” on page 5-128.

See Also

Related Examples

- “Manage Project Artifacts for Analysis in Dashboard” on page 5-95
- “Identify and Troubleshoot Gaps in Code Testing Results and Coverage” on page 5-192




View Artifact Issues in Project

When you open a Model Design or Model Testing Dashboard, the dashboard analyzes the project and collects information about the project artifacts, the artifact structure, and the traceability relationships between artifacts. For information about traceability relationships, see “Explore Traceability Information for Units and Components” on page 5-205. This example shows how to identify and troubleshoot the errors and warnings the dashboard generates during artifact analysis by using the **Artifact Issues** tab and at the command line. The **Artifact Issues** tab displays three types of messages: errors, warnings, and informational messages.

If an artifact generates a warning or error during artifact analysis, the dashboard considers that an *artifact issue*. Artifact issues in a project can lead to incorrect metric results.

Open Artifact Issues Tab

The **Artifact Issues** icon in the dashboard toolstrip changes based on the severity of the artifact issues in the project:

- If you do not have warnings or errors, the icon shows a green check mark: 
- If you have warnings but do not have errors, the icon shows a yellow triangle: 
- If you have any errors, the icon shows a red octagon: 

You can investigate artifact issues by clicking the **Artifact Issues** button in the toolstrip. The dashboard opens an **Artifact Issues** tab that you can use to troubleshoot the warnings and errors and to identify and fix artifact issues in your project. The issues appear in the **Artifact Issues** tab and persist between MATLAB sessions.

If you have errors, warnings, or informational messages, the **Artifact Issues** tab displays detailed information for each message. If not, you see an empty table.

View Details About Artifact Issues

When you point to the icon in the **Severity** column, the tooltip shows the severity of the artifact issue. The severity is either **Error**, **Warning**, or **Info** (for informational messages). The **Message** column shows detailed information about the issue. The **Source** column provides a hyperlink to open the affected artifact so that you can investigate and fix the artifact issue. To sort the artifact issues by issue type, click the **Message ID** column header.

The **Artifact Issues** tab displays a table with columns showing the severity type of an artifact issue and additional information about the issue.

- **Severity** — When you point to the icon in the **Severity** column, the tooltip shows the severity of the artifact issue with color-coded icons. The different severity types are:
 - **Info**: Informational message about the artifact.
 - **Warning**: The dashboard does not support that specific artifact, modeling construct, or relationship. For example, the dashboard does not execute callbacks on model loading or

closing. If a model relies on a callback to link to another artifact, that artifact does not link to the model when the dashboard runs the artifact traceability analysis.

- **Error**: The dashboard may not have been able to properly trace artifacts, analyze artifacts, or collect metrics. For example, the dashboard cannot analyze a dirty test case until you resave the test case.
- **Message** — Detailed information about the issue or the informational message.
- **Source** — Hyperlink to the affected artifact. Open the artifact so that you can investigate and fix the artifact issue.
- **Message ID** — Artifact issue type. To sort the artifact issues by type, click the **Message ID** column header.

For example, suppose a model in your project uses callbacks. When the dashboard performs artifact analysis, the artifact returns a warning because the dashboards do not execute model loading callbacks when loading the model for analysis. Fix this artifact issue to avoid incorrect metric results.

Severity	Message	Source	Message ID
Warning	Model Loading and Closing Callbacks have been deactivated while loading 'cc_CruiseControl' for analysis.	cc_CruiseControl	alm:simulink_trace_plugins:ModelCallbacksDeacti

Get Artifact Issues Programmatically

Alternatively, you can use the function `getArtifactIssues` on a metric engine object to return a list of the artifact issues the dashboard detects in the project. In this example model that uses callbacks, the function `getArtifactIssues` returns a warning that model loading and closing callbacks are deactivated during analysis.

```
metric_engine = metric.Engine;
getArtifactIssues(metric_engine)
```

```
ans =
```

```
struct with fields:
```

```
    IssueId: "alm:simulink_trace_plugins:ModelCallbacksDeactivated"
    IssueMessage: "Model Loading and Closing Callbacks have been deactivated while loading 'cc_CruiseControl'"
    Severity: "WARNING"
    Address: "cc_CruiseControl :: models/cc_CruiseControl.slx :: cc_CruiseControl"
    UUID: "172e0964-ac10-46ed-a3ae-e6f170fe92c0"
```

Fix Artifact Issues

For information on how to resolve artifact issues, see “Resolve Missing Artifacts, Links, and Results” on page 5-128.

See Also

`getArtifactErrors` | `getArtifactIssues`

Related Examples

- “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard” on page 5-144

- “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” on page 5-80
- “View Status of Code Testing Activities for Software Units in Project” on page 5-184
- “Resolve Missing Artifacts, Links, and Results” on page 5-128

PIL Coverage Recap

The **Coverage Recap** shows the justified and completed coverage for model and PIL testing. The dashboard considers coverage to be *completed* if the coverage is either achieved or justified. Use the table to help you identify issues across model and code testing results.

For example:

- You can identify tests that you justified during model testing but did not justify for PIL code testing.
- If there was a gap in integer overflow coverage during model testing, you need to go back and address that gap in the model testing. In the **Add Dashboard** section of the toolstrip, click **Model Testing** and use the Model Testing Dashboard to identify and fix the source of the missing coverage. The Model Testing Dashboard shows integer overflow coverage in the **Aggregated Coverage** section.

For more information, see “Identify and Troubleshoot Gaps in Code Testing Results and Coverage” on page 5-192.

SIL Coverage Recap

The **Coverage Recap** shows the justified and completed coverage for model and SIL testing. The dashboard considers coverage to be *completed* if the coverage is either achieved or justified. Use the table to help you identify issues across model and code testing results.

For example:

- You can identify tests that you justified during model testing but did not justify for SIL code testing.
- If there was a gap in integer overflow coverage during model testing, you need to go back and address that gap in the model testing. In the **Add Dashboard** section of the toolstrip, click **Model Testing** and use the Model Testing Dashboard to identify and fix the source of the missing coverage. The Model Testing Dashboard shows integer overflow coverage in the **Aggregated Coverage** section.

For more information, see “Identify and Troubleshoot Gaps in Code Testing Results and Coverage” on page 5-192.

PIL Coverage Breakdown

Metric ID

`slcomp.pil.CoverageBreakdown`

Description

This metric returns the coverage measured in the processor-in-the-loop (PIL) test results, aggregated across the unit. The metric result includes the percentage of coverage achieved by the PIL tests, the percentage of coverage justified in coverage filters, and the percentage of coverage missed by the PIL tests.

Computation Details

The metric:

- Returns aggregated coverage results.
- Does not include coverage from tests that run in simulation (model testing) or software-in-the-loop (SIL) mode.
- Returns 100% coverage for models that do not have coverage points.

Collection

To collect data for this metric, use `getMetrics` with the metric ID `slcomp.pil.CoverageBreakdown`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return `Value` as a `struct` that contains fields for:

- `Statement` — Aggregated statement coverage
- `Decision` — Aggregated decision coverage
- `Condition` — Aggregated condition coverage
- `MCDC` — Aggregated modified condition/decision coverage (MC/DC)
- `Function` — Aggregated function coverage
- `FunctionCall` — Aggregated function call coverage

Each field contains a `struct` that contains these fields:

- `Achieved` — Percentage of coverage achieved by PIL tests
- `Justified` — Percentage of coverage justified by PIL tests
- `Missed` — Percentage of coverage missed by PIL tests

- `AchievedOrJustified` — Percentage of coverage completed by PIL tests. The dashboard considers coverage *completed* if the coverage is either achieved or justified.

Compliance Thresholds

The default compliance thresholds for this metric are:

- `Compliant` — Test results return 0% missed coverage
- `Non-Compliant` — Test results return missed coverage
- `Warning` — None

See Also

“PIL Coverage Fragment” | “PIL Coverage Recap”

PIL Coverage Fragment

Metric ID

`slcomp.pil.CoverageFragment`

Description

This metric returns the coverage measured in the processor-in-the-loop (PIL) test results for each model in the unit. The metric result includes the percentage of coverage achieved by the PIL tests, the percentage of coverage justified in coverage filters, and the percentage of coverage missed by the PIL tests.

Computation Details

The metric:

- Does not include coverage from tests that run in simulation (model testing) or software-in-the-loop (SIL) mode.
- Returns 100% coverage for models that do not have coverage points.

Collection

To collect data for this metric, use `getMetrics` with the metric ID `slcomp.pil.CoverageFragment`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return `Value` as a `struct` that contains fields for:

- `Statement` — Statement coverage
- `Decision` — Decision coverage
- `Condition` — Condition coverage
- `MCDC` — Modified condition/decision coverage (MC/DC)
- `Function` — Function coverage
- `FunctionCall` — Function call coverage

Note that the metric returns instances of `metric.Result` for each model in the unit.

Each field contains a `struct` that contains these fields:

- `Achieved` — Percentage of coverage achieved by PIL tests
- `Justified` — Percentage of coverage justified by PIL tests
- `Missed` — Percentage of coverage missed by PIL tests

- **AchievedOrJustified** — Percentage of coverage completed by PIL tests. The dashboard considers coverage *completed* if the coverage is either achieved or justified.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

“PIL Coverage Breakdown” | “PIL Coverage Recap”

PIL and Model Test Statuses

Metric ID

`slcomp.pil.PilMtTestStatus`

Description

This metric returns the status of the processor-in-the-loop (PIL) and simulation (model testing) test results.

Computation Details

The metric:

- Includes only tests in the project that test the model or subsystems in the unit for which you collect metric data.
- Shows tests as untested if the test only ran in software-in-the-loop (SIL) mode.

Collection

To collect data for this metric, use `getMetrics` with the metric ID `slcomp.pil.PilMtTestStatus`.

Collecting data for this metric loads the model file and test result files and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as one of these integer outputs:

- 0 — The test results are incomplete.
- 1 — Only the PIL test results failed.
- 2 — Only the model test results failed.
- 3 — Both the PIL and model test results failed.
- 4 — Both the PIL and model test results passed.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

“PIL and Model Test Status Distributions”

PIL and Model Test Status Distributions

Metric ID

`slcomp.pil.PilMtTestStatusDistribution`

Description

This metric returns the status of the processor-in-the-loop (PIL) and simulation (model testing) test results.

Computation Details

The metric:

- Includes only tests in the project that test the model or subsystems in the unit for which you collect metric data.
- Shows tests as untested if the test only ran in normal mode or software-in-the-loop (SIL) mode.

Collection

To collect data for this metric, use `getMetrics` with the metric ID `slcomp.pil.PilMtTestStatusDistribution`.

Collecting data for this metric loads the model file and test result files and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of tests in each bin, returned as an integer vector.
- `BinEdges` — The outputs of the `sil.PilMtTestStatus` metric, returned as an integer vector. The integer outputs represent the test result statuses:
 - 0 — The test results are incomplete.
 - 1 — Only the PIL test results failed.
 - 2 — Only the model test results failed.
 - 3 — Both the PIL and model test results failed.
 - 4 — Both the PIL and model test results passed.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

“PIL and Model Test Statuses”

PIL Test Status

Metric ID

`slcomp.pil.TestStatus`

Description

This metric returns the status of the processor-in-the-loop (PIL) test result. A PIL test can have a status of either:

- Passed
- Failed
- Disabled
- Untested

Computation Details

The metric:

- Includes only tests in the project that test the model or subsystems in the unit for which you collect metric data.
- Shows tests as untested if the test only ran in normal mode or software-in-the-loop (SIL) mode.

Collection

To collect data for this metric, use `getMetrics` with the metric ID `slcomp.pil.TestStatus`.

Collecting data for this metric loads the model file and test result files and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as one of these integer outputs:

- 0 — The test failed.
- 1 — The test passed.
- 2 — The test is disabled.
- 3 — The test is untested.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

“PIL Test Status Distribution”

PIL Test Status Distribution

Metric ID

`slcomp.pil.TestStatusDistribution`

Description

This metric returns the status of the processor-in-the-loop (PIL) test results. A PIL test can have a status of either:

- Passed
- Failed
- Disabled
- Untested

Computation Details

The metric:

- Includes only tests in the project that test the model or subsystems in the unit for which you collect metric data.
- Shows tests as untested if the test only ran in normal mode or software-in-the-loop (SIL) mode.

Collection

To collect data for this metric, use `getMetrics` with the metric ID `slcomp.pil.TestStatusDistribution`.

Collecting data for this metric loads the model file and test result files and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of tests in each bin, returned as an integer vector.
- `BinEdges` — The outputs of the test status metric, returned as an integer vector. The integer outputs represent the test result statuses:
 - 0 — The test failed.
 - 1 — The test passed.
 - 2 — The test is disabled.
 - 3 — The test is untested.
- `OverallCount` — The total number of tests. The metric calculates `OverallCount` as the sum of the integers in `BinCounts`.

- **Ratios** — The ratio of a test result status to the total number of tests, returned as an integer vector that contains these elements:
 - **Ratios(1)** — Percentage of tests that failed.
 - **Ratios(2)** — Percentage of tests that passed.
 - **Ratios(3)** — Percentage of tests that are disabled.
 - **Ratios(4)** — Percentage of tests that are untested.

The percentages are in decimal form. For example, if 10% of unit tests passed and the remaining unit tests are untested, **Ratios** returns an integer vector with the percentages in decimal form: `[0; 0.1000; 0; 0.9000]`.

Compliance Thresholds

The default compliance thresholds for this metric are:

- **Compliant** — Each of the tests passed.
- **Non-Compliant** — 1 or more tests are untested, disabled, or have failed.
- **Warning** — None

See Also

“PIL Test Status”

SIL Coverage Breakdown

Metric ID

`slcomp.sil.CoverageBreakdown`

Description

This metric returns the coverage measured in the software-in-the-loop (SIL) test results, aggregated across the unit. The metric result includes the percentage of coverage achieved by the SIL tests, the percentage of coverage justified in coverage filters, and the percentage of condition coverage missed by the SIL tests.

Computation Details

The metric:

- Returns aggregated coverage results.
- Does not include coverage from tests that run in simulation (model testing) or processor-in-the-loop (PIL) mode.
- Returns 100% coverage for models that do not have coverage points.

Collection

To collect data for this metric, use `getMetrics` with the metric ID `slcomp.sil.CoverageBreakdown`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return `Value` as a `struct` that contains fields for:

- `Statement` — Aggregated statement coverage
- `Decision` — Aggregated decision coverage
- `Condition` — Aggregated condition coverage
- `MCDC` — Aggregated modified condition/decision coverage (MC/DC)
- `Function` — Aggregated function coverage
- `FunctionCall` — Aggregated function call coverage

Each field contains a `struct` that contains these fields:

- `Achieved` — Percentage of coverage achieved by SIL tests
- `Justified` — Percentage of coverage justified by SIL tests
- `Missed` — Percentage of coverage missed by SIL tests

- `AchievedOrJustified` — Percentage of coverage completed by SIL tests. The dashboard considers coverage *completed* if the coverage is either achieved or justified.

Compliance Thresholds

The default compliance thresholds for this metric are:

- `Compliant` — Test results return 0% missed coverage
- `Non-Compliant` — Test results return missed coverage
- `Warning` — None

See Also

“SIL Coverage Fragment” | “SIL Coverage Recap”

SIL Coverage Fragment

Metric ID

`slcomp.sil.CoverageFragment`

Description

This metric returns the coverage measured in the software-in-the-loop (SIL) test results for each model in the unit. The metric result includes the percentage of coverage achieved by the SIL tests, the percentage of coverage justified in coverage filters, and the percentage of coverage missed by the SIL tests.

Computation Details

The metric:

- Does not include coverage from tests that run in simulation (model testing) or processor-in-the-loop (PIL) mode.
- Returns 100% coverage for models that do not have coverage points.

Collection

To collect data for this metric, use `getMetrics` with the metric ID `slcomp.sil.CoverageFragment`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return `Value` as a `struct` that contains fields for:

- `Statement` — Statement coverage
- `Decision` — Decision coverage
- `Condition` — Condition coverage
- `MCDC` — Modified condition/decision coverage (MC/DC)
- `Function` — Function coverage
- `FunctionCall` — Function call coverage

Note that the metric returns instances of `metric.Result` for each model in the unit.

Each field contains a `struct` that contains these fields:

- `Achieved` — Percentage of coverage achieved by SIL tests
- `Justified` — Percentage of coverage justified by SIL tests
- `Missed` — Percentage of coverage missed by SIL tests

- **AchievedOrJustified** — Percentage of coverage completed by SIL tests. The dashboard considers coverage *completed* if the coverage is either achieved or justified.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

“SIL Coverage Breakdown” | “SIL Coverage Recap”

SIL and Model Test Statuses

Metric ID

`slcomp.sil.SilMtTestStatus`

Description

This metric returns the status of the software-in-the-loop (SIL) and simulation (model testing) test results.

Computation Details

The metric:

- Includes only tests in the project that test the model or subsystems in the unit for which you collect metric data.
- Shows tests as untested if the test only ran in normal mode or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric, use `getMetrics` with the metric ID `slcomp.sil.SilMtTestStatus`.

Collecting data for this metric loads the model file and test result files and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as one of these integer outputs:

- 0 — The test results are incomplete.
- 1 — Only the SIL test results failed.
- 2 — Only the model test results failed.
- 3 — Both the SIL and model test results failed.
- 4 — Both the SIL and model test results passed.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

“SIL and Model Test Status Distributions”

SIL and Model Test Status Distributions

Metric ID

`slcomp.sil.SilMtTestStatusDistribution`

Description

This metric returns a distribution of the statuses of the software-in-the-loop (SIL) and simulation (model testing) test results.

Computation Details

The metric:

- Includes only tests in the project that test the model or subsystems in the unit for which you collect metric data.
- Shows tests as untested if the test only ran in processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric, use `getMetrics` with the metric ID `slcomp.sil.SilMtTestStatusDistribution`.

Collecting data for this metric loads the model file and test result files and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of tests in each bin, returned as an integer vector.
- `BinEdges` — The outputs of the `sil.SilMtTestStatus` metric, returned as an integer vector. The integer outputs represent the test result statuses:
 - 0 — The test results are incomplete.
 - 1 — Only the SIL test results failed.
 - 2 — Only the model test results failed.
 - 3 — Both the SIL and model test results failed.
 - 4 — Both the SIL and model test results passed.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

“SIL and Model Test Statuses”

SIL Test Status

Metric ID

`slcomp.sil.TestStatus`

Description

This metric returns the status of the software-in-the-loop (SIL) test result. A SIL test can have a status of either:

- Passed
- Failed
- Disabled
- Untested

Computation Details

The metric:

- Includes only tests in the project that test the model or subsystems in the unit for which you collect metric data.
- Shows tests as untested if the test only ran in normal mode or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric, use `getMetrics` with the metric ID `slcomp.sil.TestStatus`.

Collecting data for this metric loads the model file and test result files and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as one of these integer outputs:

- 0 — The test failed.
- 1 — The test passed.
- 2 — The test is disabled.
- 3 — The test is untested.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

“SIL Test Status Distribution”

SIL Test Status Distribution

Metric ID

`slcomp.sil.TestStatusDistribution`

Description

This metric returns a distribution of the statuses of the software-in-the-loop (SIL) test results. A SIL test can have a status of either:

- Passed
- Failed
- Disabled
- Untested

Computation Details

The metric:

- Includes only tests in the project that test the model or subsystems in the unit for which you collect metric data.
- Shows tests as untested if the test only ran in normal mode or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric, use `getMetrics` with the metric ID `slcomp.sil.TestStatusDistribution`.

Collecting data for this metric loads the model file and test result files and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of tests in each bin, returned as an integer vector.
- `BinEdges` — The outputs of the test status metric, returned as an integer vector. The integer outputs represent the test result statuses:
 - 0 — The test failed.
 - 1 — The test passed.
 - 2 — The test is disabled.
 - 3 — The test is untested.
- `OverallCount` — The total number of tests. The metric calculates `OverallCount` as the sum of the integers in `BinCounts`.

- **Ratios** — The ratio of a test result status to the total number of tests, returned as an integer vector that contains these elements:
 - **Ratios(1)** — Percentage of tests that failed.
 - **Ratios(2)** — Percentage of tests that passed.
 - **Ratios(3)** — Percentage of tests that are disabled.
 - **Ratios(4)** — Percentage of tests that are untested.

The percentages are in decimal form. For example, if 10% of unit tests passed and the remaining unit tests are untested, **Ratios** returns an integer vector with the percentages in decimal form: `[0; 0.1000; 0; 0.9000]`.

Compliance Thresholds

The default compliance thresholds for this metric are:

- **Compliant** — Each of the tests passed.
- **Non-Compliant** — 1 or more tests are untested, disabled, or have failed.
- **Warning** — None

See Also

“SIL Test Status”

Requirement with Test Case

Metric ID

RequirementWithTestCase

Description

Use this metric to determine whether a requirement is linked to a test with a link where the **Type** is set to `Verifies`. The metric analyzes only requirements where the **Type** is set to `Functional` and that are linked to the unit with a link where the **Type** is set to `Implements`.

Computation Details

The metric:

- Analyzes only requirements where the **Type** is set to `Functional` and that are linked to the unit with a link where the **Type** is set to `Implements`.
- Counts links to tests in the project where the link type is set to `Verifies`, including links to tests that test other models or subsystems. For each requirement that is linked to tests, check that the links are to tests that run on the unit that implements the requirement.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, click a metric in the **Requirements Linked to Tests** section and, in the table, see the **Test Link Status** column.
- Use `getMetrics` with the metric ID `RequirementWithTestCase`.

Collecting data for this metric loads the model file and requires a Requirements Toolbox license.

Results

For this metric, instances of `metric.Result` return `Value` as one of these logical outputs:

- 0 — The requirement is not linked to tests in the project.
- 1 — The requirement is linked to at least one test with a link where the **Type** is set to `Verifies`.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Requirement with Test Case Distribution

Metric ID

RequirementWithTestCaseDistribution

Description

Use this metric to count the number of requirements that are linked to tests and the number of requirements that are missing links to tests. The metric analyzes only requirements where the **Type** is set to `Functional` and that are linked to a unit with a link where the **Type** is set to `Implements`. A requirement is linked to a test if it has a link where the **Type** is set to `Verifies`.

This metric returns the result as a distribution of the results of the `RequirementWithTestCase` metric.

Computation Details

The metric:

- Analyzes only requirements where the **Type** is set to `Functional` and that are linked to a unit with a link where the **Type** is set to `Implements`.
- Counts links to tests in the project where the link type is set to `Verifies`, including links to tests that test other models or subsystems. For each requirement that is linked to tests, check that the links are to tests that run on the unit that implements the requirement.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, place your cursor over the **Requirements with Tests** widget.
- Use `getMetrics` with the metric ID `RequirementWithTestCaseDistribution`.

Collecting data for this metric loads the model file and requires a Requirements Toolbox license.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of requirements in each bin, returned as an integer vector. The first bin includes requirements that are not linked to tests. The second bin includes requirements that are linked to at least one test.
- `BinEdges` — The logical output results of the `RequirementWithTestCase` metric, returned as a vector with entries `0` (`false`) and `1` (`true`).
- `OverallCount` — The total number of functional requirements implemented in the unit with a link where the **Type** is set to `Implements`. `OverallCount` is calculated as the sum of the integers in `BinCounts`.

- **Ratios** — The ratio of requirements missing links to tests and the ratio of requirements with links to tests, returned as an integer vector that contains these elements:
 - **Ratios(1)** — Percentage of requirements missing links to model tests.
 - **Ratios(2)** — Percentage of requirements with links to model tests.

Each ratio is calculated as the **BinCounts** value divided by the **OverallCount** value. For example, if 27.27% of unit requirements are missing links to tests and 72.73% of unit requirements have links to tests, **Ratios** returns an integer vector with the percentages in decimal form: `[0.2727; 0.7273]`.

Compliance Thresholds

The default compliance thresholds for this metric are:

- **Compliant** — 0 requirements are missing links to tests
- **Non-Compliant** — 1 or more requirements are missing links to tests
- **Warning** — None

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Test Cases per Requirement

Metric ID

TestCasesPerRequirement

Description

Use this metric to count the number of tests linked to each requirement. The metric analyzes only requirements where the **Type** is set to `Functional` and that are linked to the unit with a link where the **Type** is set to `Implements`. A test is linked to a requirement if it has a link where the **Type** is set to `Verifies`.

Computation Details

The metric:

- Analyzes only requirements where the **Type** is set to `Functional` and that are linked to the unit with a link where the **Type** is set to `Implements`.
- Counts links to tests in the project where the link type is set to `Verifies`, including links to tests that test other models or subsystems. For each requirement that is linked to tests, check that the links are to tests that run on the unit that implements the requirement.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, click a metric in the section **Tests per Requirement** to display the results in a table.
- Use `getMetrics` with the metric ID `TestCasesPerRequirement`.

Collecting data for this metric loads the model file and requires a Requirements Toolbox license.

Results

For this metric, instances of `metric.Result` return `Value` as an integer.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Test Cases per Requirement Distribution

Metric ID

TestCasesPerRequirementDistribution

Description

This metric returns a distribution of the number of tests linked to each requirement. Use this metric to determine if requirements are linked to a disproportionate number of tests. The metric analyzes only requirements where the **Type** is set to `Functional` and that are linked to the unit with a link where the **Type** is set to `Implements`. A test is linked to a requirement if it has a link where the **Type** is set to `Verifies`.

This metric returns the result as a distribution of the results of the `tests per requirement` metric.

Computation Details

The metric:

- Analyzes only requirements where the **Type** is set to `Functional` and that are linked to the unit with a link where the **Type** is set to `Implements`.
- Counts links to tests in the project where the link type is set to `Verifies`, including links to tests that test other models or subsystems. For each requirement that is linked to tests, check that the links are to tests that run on the unit that implements the requirement.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, view the **Tests per Requirement** widget.
- Use `getMetrics` with the metric ID `TestCasesPerRequirementDistribution`.

Collecting data for this metric loads the model file and requires a Requirements Toolbox license.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of requirements in each bin, returned as an integer vector.
- `BinEdges` — Bin edges for the number of tests linked to each requirement, returned as an integer vector. `BinEdges(1)` is the left edge of the first bin, and `BinEdges(end)` is the right edge of the last bin. The length of `BinEdges` is one more than the length of `BinCounts`.

The bins in the result of this metric correspond to the bins **0**, **1**, **2**, **3**, and **>3** in the **Tests per Requirement** widget.

Compliance Thresholds

This metric does not have predefined thresholds. Consequently, the compliance threshold overlay icon appears when you click **Uncategorized** in the **Overlays** section of the toolstrip.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Test Case with Requirement

Metric ID

TestCaseWithRequirement

Description

Use this metric to determine whether a test is linked to a requirement with a link where the **Type** is set to **Verifies**. The metric analyzes only tests that run on the model or subsystems in the unit for which you collect metric data.

Computation Details

The metric:

- Analyzes only tests in the project that test:
 - Unit models
 - Atomic subsystems
 - Atomic subsystem references
 - Atomic Stateflow charts
 - Atomic MATLAB Function blocks
 - Referenced models
- Counts only links where the **Type** is set to **Verifies** that link to requirements where the **Type** is set to **Functional**. This includes links to requirements that are not linked to the unit or are linked to other units. For each test that is linked to requirements, check that the links are to requirements that are implemented by the unit that the test runs on.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, click a metric in the **Tests Linked to Requirements** section and, in the table, see the **Requirement Link Status** column.
- Use `getMetrics` with the metric ID `TestCaseWithRequirement`.

Collecting data for this metric loads the model file and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as one of these logical outputs:

- 0 — The test is not linked to requirements that are implemented in the unit.
- 1 — The test is linked to at least one requirement with a link where the **Type** is set to **Verifies**.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Test Case with Requirement Distribution

Metric ID

TestCaseWithRequirementDistribution

Description

Use this metric to count the number of tests that are linked to requirements and the number of tests that are missing links to requirements. The metric analyzes only tests that run on the model or subsystems in the unit for which you collect metric data. A test is linked to a requirement if it has a link where the **Type** is set to *Verifies*.

This metric returns the result as a distribution of the results of the `TestCaseWithRequirement` metric.

Computation Details

The metric:

- Analyzes only tests in the project that test:
 - Unit models
 - Atomic subsystems
 - Atomic subsystem references
 - Atomic Stateflow charts
 - Atomic MATLAB Function blocks
 - Referenced models
- Counts only links where the **Type** is set to *Verifies* that link to requirements where the **Type** is set to *Functional*. This includes links to requirements that are not linked to the unit or are linked to other units. For each test that is linked to requirements, check that the links are to requirements that are implemented by the unit that the test runs on.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, place your cursor over the **Tests with Requirements** widget.
- Use `getMetrics` with the metric ID `TestCaseWithRequirementDistribution`.

Collecting data for this metric loads the model file and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return the `Value` as a distribution structure that contains these fields:

- **BinCounts** — The number of tests in each bin, returned as an integer vector. The first bin includes tests that are not linked to requirements. The second bin includes tests that are linked to at least one requirement.
- **BinEdges** — The logical output results of the `TestCaseWithRequirement` metric, returned as a vector with entries `0` (`false`) and `1` (`true`).
- **OverallCount** — The total number of tests. `OverallCount` is calculated as the sum of the integers in `BinCounts`.
- **Ratios** — The ratio of tests missing links to requirements and the ratio of tests with links to requirements, returned as an integer vector that contains these elements:
 - `Ratios(1)` — Percentage of model tests missing links to requirements.
 - `Ratios(2)` — Percentage of model tests with links to requirements.

Each ratio is calculated as the `BinCounts` value divided by the `OverallCount` value. For example, if 27.27% of unit tests are missing links to requirements and 72.73% of unit tests have links to requirements, `Ratios` returns an integer vector with the percentages in decimal form: `[0.2727; 0.7273]`.

Compliance Thresholds

The default compliance thresholds for this metric are:

- **Compliant** — 0 unit tests are missing links to requirements
- **Non-Compliant** — 1 or more unit tests are missing links to requirements
- **Warning** — None

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Requirements per Test Case

Metric ID

RequirementsPerTestCase

Description

Use this metric to count the number of requirements linked to each test. The metric analyzes only tests that run on the model or subsystems in the unit for which you collect metric data. A test is linked to a requirement if it has a link where the **Type** is set to *Verifies*.

Computation Details

The metric:

- Analyzes only tests in the project that test:
 - Unit models
 - Atomic subsystems
 - Atomic subsystem references
 - Atomic Stateflow charts
 - Atomic MATLAB Function blocks
 - Referenced models
- Counts only links where the **Type** is set to *Verifies* that link to requirements where the **Type** is set to *Functional*. This includes links to requirements that are not linked to the unit or are linked to other units. For each test that is linked to requirements, check that the links are to requirements that are implemented by the unit that the test runs on.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, click a metric in the section **Requirements per Test** to display the results in a table.
- Use `getMetrics` with the metric ID `RequirementsPerTestCase`.

Collecting data for this metric loads the model file and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as an integer.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Requirements per Test Case Distribution

Metric ID

RequirementsPerTestCaseDistribution

Description

This metric returns a distribution of the number of requirements linked to each test. Use this metric to determine if tests are linked to a disproportionate number of requirements. The metric analyzes only tests that run on the model or subsystems in the unit for which you collect metric data. A test is linked to a requirement if it has a link where the **Type** is set to **Verifies**.

This metric returns the result as a distribution of the results of the Requirements per test metric.

Computation Details

The metric:

- Analyzes only tests in the project that test:
 - Unit models
 - Atomic subsystems
 - Atomic subsystem references
 - Atomic Stateflow charts
 - Atomic MATLAB Function blocks
 - Referenced models
- Counts only links where the **Type** is set to **Verifies** that link to requirements where the **Type** is set to **Functional**. This includes links to requirements that are not linked to the unit or are linked to other units. For each test that is linked to requirements, check that the links are to requirements that are implemented by the unit that the test runs on.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, view the **Requirements per Test** widget.
- Use `getMetrics` with the metric ID `RequirementsPerTestCaseDistribution`.

Collecting data for this metric loads the model file and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- **BinCounts** — The number of tests in each bin, returned as an integer vector.
- **BinEdges** — Bin edges for the number of requirements linked to each test, returned as an integer vector. **BinEdges(1)** is the left edge of the first bin, and **BinEdges(end)** is the right edge of the last bin. The length of **BinEdges** is one more than the length of **BinCounts**.

The bins in the result of this metric correspond to the bins **0**, **1**, **2**, **3**, and **>3** in the **Requirements per Test** widget.

Compliance Thresholds

This metric does not have predefined thresholds. Consequently, the compliance threshold overlay icon appears when you click **Uncategorized** in the **Overlays** section of the toolstrip.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Test Case Type

Metric ID

TestCaseType

Description

This metric returns the type of the test case. A test case is either a baseline, equivalence, or simulation test.

- Baseline tests compare outputs from a simulation to expected results stored as baseline data.
- Equivalence tests compare the outputs from two different simulations. Simulations can run in different modes, such as normal simulation and software-in-the-loop.
- Simulation tests run the system under test and capture simulation data. If the system under test contains blocks that verify simulation, such as Test Sequence and Test Assessment blocks, the pass/fail results are reflected in the simulation test results.

Computation Details

The metric includes only test cases in the project that test the model or subsystems in the unit for which you collect metric data.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, click a widget in the section **Tests by Type** to display the results in a table.
- Use `getMetrics` with the metric ID `TestCaseType`.

Collecting data for this metric loads the model file and test files and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as one of these integer outputs:

- 0 — Simulation test
- 1 — Baseline test
- 2 — Equivalence test

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Test Case Type Distribution

Metric ID

TestCaseTypeDistribution

Description

This metric returns a distribution of the types of test cases that run on the unit. A test case is either a baseline, equivalence, or simulation test. Use this metric to determine if there is a disproportionate number of test cases of one type.

- Baseline tests compare outputs from a simulation to expected results stored as baseline data.
- Equivalence tests compare the outputs from two different simulations. Simulations can run in different modes, such as normal simulation and software-in-the-loop.
- Simulation tests run the system under test and capture simulation data. If the system under test contains blocks that verify simulation, such as Test Sequence and Test Assessment blocks, the pass/fail results are reflected in the simulation test results.

This metric returns the result as a distribution of the results of the `Test case type` metric.

Computation Details

The metric includes only test cases in the project that test the model or subsystems in the unit for which you collect metric data.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, view the **Tests by Type** widget.
- Programmatically, use `getMetrics` with the metric ID `TestCaseTypeDistribution`.

Collecting data for this metric loads the model file and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of test cases in each bin, returned as an integer vector.
- `BinEdges` — The outputs of the `Test case type` metric, returned as an integer vector. The integer outputs represent the three test case types:
 - 0 — Simulation test
 - 1 — Baseline test
 - 2 — Equivalence test

Compliance Thresholds

This metric does not have predefined thresholds. Consequently, the compliance threshold overlay icon appears when you click **Uncategorized** in the **Overlays** section of the toolbar.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Test Case Tag

Metric ID

TestCaseTag

Description

This metric returns the tags for a test case. You can add custom tags to a test case by using the Test Manager.

Computation Details

The metric includes only test cases in the project that test the model or subsystems in the unit for which you collect metric data.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, click a widget in the **Tests with Tag** section to display the results in a table.
- Use `getMetrics` with the metric ID `TestCaseTag`.

Collecting data for this metric loads the model file and test files and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as a string.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Test Case Tag Distribution

Metric ID

TestCaseTagDistribution

Description

This metric returns a distribution of the tags on the test cases that run on the unit. For a test case, you can specify custom tags in a comma-separated list in the Test Manager. Use this metric to determine if there is a disproportionate number of test cases that have a particular tag.

This metric returns the result as a distribution of the results of the `Test case tag` metric.

Computation Details

The metric includes only test cases in the project that test the model or subsystems in the unit for which you collect metric data.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, view the **Tests with Tag** widget.
- Use `getMetrics` with the metric ID `TestCaseTagDistribution`.

Collecting data for this metric loads the model file and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of test cases in each bin, returned as an integer vector.
- `BinEdges` — The bin edges for the tags that are specified for the test cases, returned as a string array.

Compliance Thresholds

This metric does not have predefined thresholds. Consequently, the compliance threshold overlay icon appears when you click **Uncategorized** in the **Overlays** section of the toolstrip.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Model Coverage Breakdown

Metric ID

`slcomp.mt.CoverageBreakdown`

Description

This metric returns the coverage measured in the model test results, aggregated across each of the models in the unit. The metric result includes the percentage of coverage achieved by the model tests, the percentage of model coverage justified in coverage filters, and the percentage of coverage missed by the model tests.

Computation Details

The metric:

- Returns aggregated coverage results.
- Does not include coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
- Returns 100% coverage for models that do not have coverage points.

Collection

Use `getMetrics` with the metric ID `slcomp.mt.CoverageBreakdown`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a `struct` that contains fields for:

- `Execution` — Aggregated execution coverage for the unit
- `Decision` — Aggregated decision coverage for the unit
- `Condition` — Aggregated condition coverage for the unit
- `MCDC` — Aggregated modified condition and decision coverage (MC/DC) for the unit
- `OverflowSaturation` — Aggregated saturate on integer overflow coverage for the unit

Each field contains a `struct` that contains these fields:

- `Achieved` — The percentage of coverage achieved by the model tests.
- `Justified` — The percentage of model coverage justified by coverage filters.
- `Missed` — The percentage of coverage missed by the model tests.
- `AchievedOrJustified` — The percentage of coverage achieved by the model tests or justified by coverage filters.

Compliance Thresholds

The default compliance thresholds for this metric are:

- **Compliant** — Test results return 0% missed coverage
- **Non-Compliant** — Test results return missed coverage
- **Warning** — None

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Model Coverage Fragment

Metric ID

`slcomp.mt.CoverageFragment`

Description

This metric returns the coverage measured in the model test results for each model in the unit. The metric result includes the percentage of coverage achieved by the model tests, the percentage of model coverage justified in coverage filters, and the percentage of coverage missed by the model tests.

Computation Details

The metric:

- Does not include coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
- Returns 100% coverage for models that do not have coverage points.

Collection

Use `getMetrics` with the metric ID `slcomp.mt.CoverageFragment`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a `struct` that contains fields for:

- `Execution` — Execution coverage
- `Decision` — Decision coverage
- `Condition` — Condition coverage
- `MCDC` — Modified condition and decision coverage (MC/DC)
- `OverflowSaturation` — Saturate on integer overflow coverage

Note that there are instances of `metric.Result` returned for each model in the unit.

Each field contains a `struct` that contains these fields:

- `Achieved` — The percentage of coverage achieved by the model tests.
- `Justified` — The percentage of model coverage justified by coverage filters.
- `Missed` — The percentage of coverage missed by the model tests.
- `AchievedOrJustified` — The percentage of coverage achieved by the model tests or justified by coverage filters.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Model Test Status

Metric ID

`slcomp.mt.TestStatus`

Description

This metric returns the status of the model test result. A model test can have a status of either:

- Passed
- Failed
- Disabled
- Untested

Computation Details

The metric:

- Includes only simulation tests in the project that test the model or subsystems in the unit for which you collect metric data.
- Does not count the status of tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. The metric shows these tests as untested.

Collection

To collect data for this metric, use the function `getMetrics` with the metric ID `slcomp.mt.TestStatus`.

Collecting data for this metric loads the model file and test result files and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as one of these integer outputs:

- 0 — The test failed.
- 1 — The test passed.
- 2 — The test is disabled.
- 3 — The test is untested.

Compliance Thresholds

The default compliance thresholds for this metric are:

- `Compliant` — Each of the tests passed.

- Non-Compliant — 1 or more tests are untested, disabled, or have failed.
- Warning — None

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Model Test Status Distribution

Metric ID

`slcomp.mt.TestStatusDistribution`

Description

This metric returns a distribution of the status of the results of model tests that run on the unit. A model test can have a status of either:

- Passed
- Failed
- Disabled
- Untested

This metric returns the result as a distribution of the results of the `slcomp.mt.TestStatus` metric.

Computation Details

The metric:

- Includes only simulation tests in the project that test the model or subsystems in the unit for which you collect metric data.
- Does not count the status of tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. The metric shows these tests as untested.

Collection

To collect data for this metric, use `getMetrics` with the metric ID `slcomp.mt.TestStatusDistribution`.

Collecting data for this metric loads the model file and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of tests in each bin, returned as an integer vector.
- `BinEdges` — The outputs of the test status metric, returned as an integer vector. The integer outputs represent the test result statuses:
 - 0 — The test failed.
 - 1 — The test passed.
 - 2 — The test is disabled.
 - 3 — The test is untested.

- **OverallCount** — The total number of tests. The metric calculates **OverallCount** as the sum of the integers in **BinCounts**.
- **Ratios** — The ratio of a test result status to the total number of tests, returned as an integer vector that contains these elements:
 - **Ratios(1)** — Percentage of tests that failed.
 - **Ratios(2)** — Percentage of tests that passed.
 - **Ratios(3)** — Percentage of tests that are disabled.
 - **Ratios(4)** — Percentage of tests that are untested.

The percentages are in decimal form. For example, if 10% of unit tests passed and the remaining unit tests are untested, **Ratios** returns an integer vector with the percentages in decimal form: `[0; 0.1000; 0; 0.9000]`.

Compliance Thresholds

The default compliance thresholds for this metric are:

- **Compliant** — Each of the tests passed.
- **Non-Compliant** — 1 or more tests are untested, disabled, or have failed.
- **Warning** — None

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Test Case Verification Status

Metric ID

TestCaseVerificationStatus

Description

Use this metric to determine whether a test has pass/fail criteria.

A test has pass/fail criteria if it has at least one of the following:

- at least one executed verify statement
- at least one executed temporal or logical assessment
- custom criteria that has a pass/fail status in Simulink Test Manager
- baseline criteria which determine the pass/fail criteria of the test

Computation Details

The metric:

- Includes only tests in the project that test the model or subsystems in the unit for which you collect metric data.
- Does not count the pass/fail criteria of tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. The metric shows these tests as **Missing Pass/Fail Criteria**.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Model Test Status** section, click the **Inconclusive** widget to view the `TestCaseVerificationStatus` results in a table.
- Use `getMetrics` with the metric ID `TestCaseVerificationStatus`.

Collecting data for this metric loads the model file and test result files and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as one of these integer outputs:

- 0 — The test is missing pass/fail criteria.
- 1 — The test has pass/fail criteria.
- 2 — The test was not run.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Test Case Verification Status Distribution

Metric ID

TestCaseVerificationStatusDistribution

Description

Use this metric to count the number of tests that do not have pass/fail criteria and the number of tests that do have pass/fail criteria.

A test has pass/fail criteria if it has at least one of the following:

- at least one executed verify statement
- at least one executed temporal or logical assessment
- custom criteria that has a pass/fail status in Simulink Test Manager
- baseline criteria which determine the pass/fail criteria of the test

This metric returns the result as a distribution of the results of the `TestCaseVerificationStatusDistribution` metric.

Computation Details

The metric:

- Includes only tests in the project that test the model or subsystems in the unit for which you collect metric data.
- Does not count the pass/fail criteria of tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. The metric shows these tests as **Missing Pass/Fail Criteria**.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Model Test Status** section, place your cursor over the **Inconclusive** widget.
- Use `getMetrics` with the metric ID `TestCaseVerificationStatusDistribution`.

Collecting data for this metric loads the model file and test files and requires a Simulink Test license.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of tests in each bin, returned as an integer vector.
- `BinEdges` — The outputs of the `TestCaseVerificationStatus` metric, returned as an integer vector. The integer outputs represent the three test verification statuses:

- 0 — The test is missing pass/fail criteria.
- 1 — The test has pass/fail criteria.
- 2 — The test was not run.

Compliance Thresholds

The default compliance thresholds for this metric are:

- Compliant — 0 unit tests are missing pass/fail criteria
- Non-Compliant — 1 or more unit tests do not have pass/fail criteria
- Warning — None

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Requirements Execution Coverage Breakdown

Metric ID

RequirementsExecutionCoverageBreakdown

Description

This metric returns the fraction of overall achieved execution coverage that comes from requirements-based tests.

Computation Details

The metric:

- Analyzes the overall aggregated coverage results.
- Does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Requirements-Based Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `RequirementsExecutionCoverageBreakdown`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of requirements-based tests that contribute to the overall achieved execution coverage.
- `Denominator` — The total number of tests (requirements-based and non-requirements-based) that contribute to the overall achieved execution coverage.

Compliance Thresholds

The default compliance thresholds for this metric are:

- `Compliant` — 100% of the overall achieved execution coverage comes from requirements-based tests
- `Non-Compliant` — Less than 100% of the overall achieved execution coverage comes from requirements-based tests

- Warning — None

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Requirements Decision Coverage Breakdown

Metric ID

RequirementsDecisionCoverageBreakdown

Description

This metric returns the fraction of overall achieved decision coverage that comes from requirements-based tests.

Computation Details

The metric:

- Analyzes the overall aggregated coverage results.
- Does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Requirements-Based Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `RequirementsDecisionCoverageBreakdown`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of requirements-based tests that contribute to the overall achieved decision coverage.
- `Denominator` — The total number of tests (requirements-based and non-requirements-based) that contribute to the overall achieved decision coverage.

Compliance Thresholds

The default compliance thresholds for this metric are:

- `Compliant` — 100% of the overall achieved decision coverage comes from requirements-based tests
- `Non-Compliant` — Less than 100% of the overall achieved decision coverage comes from requirements-based tests

- Warning — None

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Requirements Condition Coverage Breakdown

Metric ID

RequirementsConditionCoverageBreakdown

Description

This metric returns the fraction of overall achieved condition coverage that comes from requirements-based tests.

Computation Details

The metric:

- Analyzes the overall aggregated coverage results.
- Does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Requirements-Based Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `RequirementsConditionCoverageBreakdown`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of requirements-based tests that contribute to the overall achieved condition coverage.
- `Denominator` — The total number of tests (requirements-based and non-requirements-based) that contribute to the overall achieved condition coverage.

Compliance Thresholds

The default compliance thresholds for this metric are:

- `Compliant` — 100% of the overall achieved condition coverage comes from requirements-based tests
- `Non-Compliant` — Less than 100% of the overall achieved condition coverage comes from requirements-based tests

- Warning — None

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Requirements MDCDC Coverage Breakdown

Metric ID

RequirementsMDCDCoverageBreakdown

Description

This metric returns the fraction of overall achieved MC/DC coverage that comes from requirements-based tests.

Computation Details

The metric:

- Analyzes the overall aggregated coverage results.
- Does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Requirements-Based Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `RequirementsMDCDCoverageBreakdown`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of requirements-based tests that contribute to the overall achieved MC/DC coverage.
- `Denominator` — The total number of tests (requirements-based and non-requirements-based) that contribute to the overall achieved MC/DC coverage.

Compliance Thresholds

The default compliance thresholds for this metric are:

- `Compliant` — 100% of the overall achieved MC/DC coverage comes from requirements-based tests
- `Non-Compliant` — Less than 100% of the overall achieved MC/DC coverage comes from requirements-based tests

- Warning — None

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Requirements Execution Coverage Fragment

Metric ID

RequirementsExecutionCoverageFragment

Description

This metric returns the fraction of overall achieved execution coverage that comes from requirements-based tests.

Computation Details

The metric does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Requirements-Based Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `RequirementsExecutionCoverageFragment`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of requirements-based tests that contribute to the overall achieved execution coverage.
- `Denominator` — The total number of tests (requirements-based and non-requirements-based) that contribute to the overall achieved execution coverage.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Requirements Decision Coverage Fragment

Metric ID

RequirementsDecisionCoverageFragment

Description

This metric returns the fraction of overall achieved decision coverage that comes from requirements-based tests.

Computation Details

The metric does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Requirements-Based Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `RequirementsDecisionCoverageFragment`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of requirements-based tests that contribute to the overall achieved decision coverage.
- `Denominator` — The total number of tests (requirements-based and non-requirements-based) that contribute to the overall achieved decision coverage.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Requirements Condition Coverage Fragment

Metric ID

RequirementsConditionCoverageFragment

Description

This metric returns the fraction of overall achieved condition coverage that comes from requirements-based tests.

Computation Details

The metric does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Requirements-Based Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `RequirementsConditionCoverageFragment`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of requirements-based tests that contribute to the overall achieved condition coverage.
- `Denominator` — The total number of tests (requirements-based and non-requirements-based) that contribute to the overall achieved condition coverage.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Requirements MCDCCoverage Fragment

Metric ID

RequirementsMCDCCoverageFragment

Description

This metric returns the fraction of overall achieved MC/DC coverage that comes from requirements-based tests.

Computation Details

The metric does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Requirements-Based Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `RequirementsMCDCCoverageFragment`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- **Numerator** — The number of requirements-based tests that contribute to the overall achieved MC/DC coverage.
- **Denominator** — The total number of tests (requirements-based and non-requirements-based) that contribute to the overall achieved MC/DC coverage.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Unit Boundary Execution Coverage Breakdown

Metric ID

UnitBoundaryExecutionCoverageBreakdown

Description

This metric returns the fraction of overall achieved execution coverage that comes from unit-boundary tests.

Computation Details

The metric:

- Analyzes the overall aggregated coverage results.
- Does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Unit-Boundary Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `UnitBoundaryExecutionCoverageBreakdown`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of unit-boundary tests that contribute to the overall achieved execution coverage.
- `Denominator` — The total number of tests (unit-boundary and non-unit-boundary) that contribute to the overall achieved execution coverage.

Compliance Thresholds

This metric does not have predefined thresholds. Consequently, the compliance threshold overlay icon appears when you click **Uncategorized** in the **Overlays** section of the toolstrip.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Unit Boundary Decision Coverage Breakdown

Metric ID

UnitBoundaryDecisionCoverageBreakdown

Description

This metric returns the fraction of overall achieved decision coverage that comes from unit-boundary tests.

Computation Details

The metric:

- Analyzes the overall aggregated coverage results.
- Does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Unit-Boundary Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `UnitBoundaryDecisionCoverageBreakdown`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of unit-boundary tests that contribute to the overall achieved decision coverage.
- `Denominator` — The total number of tests (unit-boundary and non-unit-boundary) that contribute to the overall achieved decision coverage.

Compliance Thresholds

This metric does not have predefined thresholds. Consequently, the compliance threshold overlay icon appears when you click **Uncategorized** in the **Overlays** section of the toolbar.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Unit Boundary Condition Coverage Breakdown

Metric ID

UnitBoundaryConditionCoverageBreakdown

Description

This metric returns the fraction of overall achieved condition coverage that comes from unit-boundary tests.

Computation Details

The metric:

- Analyzes the overall aggregated coverage results.
- Does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Unit-Boundary Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `UnitBoundaryConditionCoverageBreakdown`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of unit-boundary tests that contribute to the overall achieved condition coverage.
- `Denominator` — The total number of tests (unit-boundary and non-unit-boundary) that contribute to the overall achieved condition coverage.

Compliance Thresholds

This metric does not have predefined thresholds. Consequently, the compliance threshold overlay icon appears when you click **Uncategorized** in the **Overlays** section of the toolstrip.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Unit Boundary MCDCCoverage Breakdown

Metric ID

UnitBoundaryMCDCCoverageBreakdown

Description

This metric returns the fraction of overall achieved MC/DC coverage that comes from unit-boundary tests.

Computation Details

The metric:

- Analyzes the overall aggregated coverage results.
- Does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Unit-Boundary Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `UnitBoundaryMCDCCoverageBreakdown`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of unit-boundary tests that contribute to the overall achieved MC/DC coverage.
- `Denominator` — The total number of tests (unit-boundary and non-unit-boundary) that contribute to the overall achieved MC/DC coverage.

Compliance Thresholds

This metric does not have predefined thresholds. Consequently, the compliance threshold overlay icon appears when you click **Uncategorized** in the **Overlays** section of the toolstrip.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Unit Boundary Execution Coverage Fragment

Metric ID

UnitBoundaryExecutionCoverageFragment

Description

This metric returns the fraction of overall achieved execution coverage that comes from unit-boundary tests.

Computation Details

The metric does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Unit-Boundary Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `UnitBoundaryExecutionCoverageFragment`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of unit-boundary tests that contribute to the overall achieved execution coverage.
- `Denominator` — The total number of tests (unit-boundary and non-unit-boundary) that contribute to the overall achieved execution coverage.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Unit Boundary Decision Coverage Fragment

Metric ID

UnitBoundaryDecisionCoverageFragment

Description

This metric returns the fraction of overall achieved decision coverage that comes from unit-boundary tests.

Computation Details

The metric does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Unit-Boundary Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `UnitBoundaryDecisionCoverageFragment`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of unit-boundary tests that contribute to the overall achieved decision coverage.
- `Denominator` — The total number of tests (unit-boundary and non-unit-boundary) that contribute to the overall achieved decision coverage.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Unit Boundary Condition Coverage Fragment

Metric ID

UnitBoundaryConditionCoverageFragment

Description

This metric returns the fraction of overall achieved condition coverage that comes from unit-boundary tests.

Computation Details

The metric does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Unit-Boundary Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `UnitBoundaryConditionCoverageFragment`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of unit-boundary tests that contribute to the overall achieved condition coverage.
- `Denominator` — The total number of tests (unit-boundary and non-unit-boundary) that contribute to the overall achieved condition coverage.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Unit Boundary MCDC Coverage Fragment

Metric ID

UnitBoundaryMCDCCoverageFragment

Description

This metric returns the fraction of overall achieved MC/DC coverage that comes from unit-boundary tests.

Computation Details

The metric does not analyze coverage from tests that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Collection

To collect data for this metric:

- In the Model Testing Dashboard, in the **Achieved Coverage Ratio** section, point to the widget for **Unit-Boundary Tests**, point to the three dots, and click the run button.
- Use `getMetrics` with the metric ID `UnitBoundaryMCDCCoverageFragment`.

Collecting data for this metric loads the model file and test results files and requires a Simulink Coverage license.

Results

For this metric, instances of `metric.Result` return the `Value` as a structure that contains these fields:

- `Numerator` — The number of unit-boundary tests that contribute to the overall achieved MC/DC coverage.
- `Denominator` — The total number of tests (unit-boundary and non-unit-boundary) that contribute to the overall achieved MC/DC coverage.

Compliance Thresholds

This metric does not have predefined thresholds.

See Also

Related Examples

- “Collect Metrics on Model Testing Artifacts Programmatically” on page 5-115

Overall Design Cyclomatic Complexity

Metric ID

slcomp.OverallCyclomaticComplexity

Description

Use this metric to determine the overall design cyclomatic complexity for a unit or component.

The *design cyclomatic complexity* is the number of possible execution paths through a design. In general, the more paths there are through a design, the more complex the design is. When you keep the design cyclomatic complexity low, the design typically is easier to read, maintain, and test. The design cyclomatic complexity is calculated as the number of decision paths plus one. The metric adds one to account for the execution path represented by the default path. The design cyclomatic complexity includes the default path because the metric identifies each possible outcome from an execution path, including the default outcome.

For example, the design cyclomatic complexity of an `if-else` statement is two because the `if` statement represents one decision and the `else` statement represents the default path. The default path is not included in the number of decisions because no decision is made to reach a default state.

```
function y = fcn(u)
    if u < 0
        % one decision
        y = -1*u;
    else
        % default path
        % zero decisions
        y = u;
    end
end
```

The overall design cyclomatic complexity metric counts the total number of execution paths in a unit or component. The default path is only counted once per unit or component.

The overall design cyclomatic complexity is equal to the sum of the:

- “Simulink Decision Count” on page 5-310
- “Stateflow Decision Count” on page 5-326
- “MATLAB Decision Count” on page 5-331
- plus one (for the default path)

Computation Details

The metric has a minimum value of 1 because every design has at least one default execution path, even if no decisions are made within the design.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.OverallCyclomaticComplexity`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the overall design cyclomatic complexity for a unit or component.

Examples

Suppose your unit contains four Simulink decisions, five Stateflow decisions, and six MATLAB decisions. The overall design cyclomatic complexity for your unit is 16 because the sum of four Simulink decisions, plus five Stateflow decisions, plus six MATLAB decisions, plus one default outcome equals 16.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150

Layer Depth

Metric ID

`slcomp.LayerDepth`

Description

Use this metric to count how many layers deep a model component is in the model hierarchy for a unit or component. A *layer* is a parent or child model component in the model hierarchy.

This metric analyzes these model components:

- Simulink Subsystems
- Stateflow states
- MATLAB Function blocks
- External MATLAB functions and classes
- MATLAB methods

Computation Details

The metric:

- Does not count relationships to data dictionaries.
- Does not count relationships to other units. For example, if you have a component with a required relationship to another model that is a unit, that relationship is not included in the metric count. The metric only counts the relationship if the other model is not a unit.

Collection

To collect data for this metric:

- In the Model Maintainability Dashboard, in the **Component Structure** section, point to the **Depth** widget and click the **Run metrics for widget** icon. The **Depth** widget shows the maximum depth found in the current unit or component. If you click on the **Depth** widget, you can view a table that shows the depth for each model component.
- Use `getMetrics` with the metric identifier `slcomp.LayerDepth`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the depth of the model component in the model hierarchy of a unit or component.

Examples

Suppose you have a unit, `u1`, that contains a subsystem, `s1`, and subsystem `s1` contains subsystem `s2`. `u1` is at layer depth 1, `s1` is at layer depth 2, and `s2` is at layer depth 3.

See Also

Related Examples

- “Maximum Layer Depth” on page 5-298
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

Maximum Layer Depth

Metric ID

`slcomp.MaxLayerDepth`

Description

Use this metric to determine the maximum number of layers in the model hierarchy of a unit or component. A *layer* is a parent or child model component in the model hierarchy.

This metric analyzes these model components:

- Simulink Subsystems
- Stateflow States
- MATLAB Functions
- MATLAB Methods

Computation Details

The metric:

- Does not count model references or subsystem references.
- Does not count relationships to data dictionaries.
- Does not count relationships to other units. For example, if you have a component with a required relationship to another model that is a unit, that relationship is not included in the metric count. The metric only analyzes the relationship if the other model is not a unit.

Collection

To collect data for this metric:

- In the Model Maintainability Dashboard, in the **Component Structure** section, point to the **Depth** widget and click the **Run metrics for widget** icon. The **Depth** widget shows the maximum depth found in the current unit or component. If you click on the **Depth** widget, you can view a table that shows the depth for each model component.
- Use `getMetrics` with the metric identifier `slcomp.MaxLayerDepth`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the maximum number of layers deep a model component is in the model hierarchy for a unit or component.

Examples

Suppose you have a unit, `u1`, that contains only a subsystem, `s1`, and subsystem `s1` contains only subsystem `s2`. `u1` is at layer depth 1, `s1` is at layer depth 2, and `s2` is at layer depth 3. The maximum

layer depth is 3 because 3 is the maximum of the layer depth values associated with the unit. You can view the layer depth associated with each child model component in the **Metric Details** table by clicking on the **Depth** widget.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150

Layer Breadth

Metric ID

`slcomp.LayerBreadth`

Description

Use this metric to count the number of child model components that each artifact contains. A *layer* is a parent or child model component in the model hierarchy.

This metric counts the following model components:

- Simulink Subsystems
- Stateflow states
- MATLAB Function blocks
- External MATLAB functions and classes
- MATLAB methods

Computation Details

The metric:

- Includes model references as child model components.
- Does not count relationships to data dictionaries. For example, suppose that you have a block diagram that contains a subsystem and requires a Simulink Data Dictionary (SLDD) file. The metric returns a layer breadth of one, not two.
- Does not count relationships to other units. For example, if you have a component with a required relationship to another model that is a unit, that relationship is not included in the metric count. The metric only counts the relationship if the other model is not a unit.

Collection

To collect data for this metric:

- In the Model Maintainability Dashboard, in the **Component Structure** section, point to the **Breadth** widget and click the **Run metrics for widget** icon. The **Breadth** widget shows the maximum breadth found in the current unit or component. If you click on the **Breadth** widget, you can view a table that shows the breadth for each artifact.
- Use `getMetrics` with the metric identifier `slcomp.LayerBreadth`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the number of child model components that each artifact contains.

Examples

Suppose you have a unit, *u2*, that contains two subsystems: *c1* and *c2*. The layer breadth for the unit *u2* is 2 because there are 2 child model components directly associated with the unit.

See Also

Related Examples

- “Maximum Layer Breadth” on page 5-302
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

Maximum Layer Breadth

Metric ID

`slcomp.MaxLayerBreadth`

Description

Use this metric to determine the maximum number of child model components that a single model layer in the unit or component contains.

This metric analyzes these model components:

- Simulink Block Diagrams
- Simulink Subsystems
- Stateflow Charts
- MATLAB Functions

Computation Details

The metric:

- Includes model references as child model components.
- Does not count relationships to data dictionaries. For example, suppose that you have a block diagram that contains a subsystem and requires a Simulink Data Dictionary (SLDD) file. The layer breadth for the model component is one, not two.
- Does not count relationships to other units. For example, if you have a component with a required relationship to another model that is a unit, that relationship is not included in the metric count. The metric only analyzes the relationship if the other model is not a unit.

Collection

To collect data for this metric:

- In the Model Maintainability Dashboard, in the **Component Structure** section, point to the **Breadth** widget and click the **Run metrics for widget** icon. The **Breadth** widget shows the maximum breadth found in the current unit or component. If you click on the **Breadth** widget, you can view a table that shows the breadth for each artifact.
- Use `getMetrics` with the metric identifier `slcomp.MaxLayerBreadth`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the maximum number of child model components for a unit or component.

Examples

Suppose you have a unit, `u3`, that contains three subsystems: `e1`, `e2`, and `e3`. `e1` has a layer breadth of 2, `e2` has a layer breadth of three, and `e3` has a layer breadth of four. The maximum layer breadth for the unit is four because that is the maximum of the layer breadth values associated with the child model components for the unit. You can view the layer breadth associated with each child model component in the **Metric Details** table by clicking on the **Breadth** widget.

See Also

Related Examples

- “Layer Breadth” on page 5-300
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

Input and Output Component Interface Ports

Metric ID

`slcomp.InterfacePorts`

Description

Use this metric to count the number of input ports and output ports to the component interface.

This metric counts:

- Simulink Inport blocks
- Simulink Outport blocks
- Input ports for Stateflow charts
- Output ports for Stateflow charts
- Inputs and outputs to MATLAB code

Computation Details

The metric returns the number of component interfaces.

Collection

To collect data for this metric:

- In the Model Maintainability Dashboard, in the **Component Interface** section, point to the **Input Ports** or **Output Ports** widgets and click the **Run metrics for widget** icon.
- Use `getMetrics` with the metric identifier `slcomp.InterfacePorts`.

Results

For this metric, instances of `metric.Result` return the `Value` as an integer vector that contains these elements:

- `Value(1)` — The number of input component interface ports. This corresponds to the **Input Ports** widget in the **Component Interface** section.
- `Value(2)` — The number of output component interface ports. This corresponds to the **Output Ports** widget in the **Component Interface** section.

The results of this metric correspond to the **Input Ports** and **Output Ports** widgets in the **Component Interface** section.

Examples

Suppose your component is a model that has eleven Inport blocks and five Outport blocks, this metric returns a `Result` array with a `Value` property of `[11, 5]`.

See Also

Related Examples

- “Input and Output Component Interface Signals” on page 5-306
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

Input and Output Component Interface Signals

Metric ID

`slcomp.ComponentInterfaceSignals`

Description

Use this metric to count the number of root-level, input signals and output signals that connect to the component interface.

This metric counts signals to or from:

- Simulink Inport blocks
- Simulink Outport blocks
- Input ports for Stateflow charts
- Output ports for Stateflow charts
- Inputs and outputs to MATLAB code

Computation Details

The metric:

- Returns the number of component interface signals. If the data type of an input or output is `Bus`, the metric counts the individual elements in the bus.
- Displays a warning if the data type of an input or output is set to `'Inherit: auto'`. The metric is unable to resolve inherited data types. Specify valid data types, other than `'Inherit: auto'`, for root outports and inports.
- Displays a warning if the `'HasAccessToBaseWorkspace'` property for a model is set to `true`. Data type definitions accessed from the base workspace are not considered by traceability analysis. Changes to base workspace variables will not cause the metric to be recollected.

Collection

To collect data for this metric:

- In the Model Maintainability Dashboard, in the **Component Interface** section, point to the **Input Signals** or **Output Signals** widgets and click the **Run metrics for widget** icon.
- Use `getMetrics` with the metric identifier `slcomp.ComponentInterfaceSignals`.

Results

For this metric, instances of `metric.Result` return the `Value` as an integer vector that contains these elements:

- `Value(1)` — The number of input component interface signals. This corresponds to the **Input Signals** widget in the **Component Interface** section.

- **Value(2)** — The number of output component interface signals. This corresponds to the **Output Signals** widget in the **Component Interface** section.

The results of this metric correspond to the **Input Signals** and **Output Signals** widgets in the **Component Interface** section.

Examples

Suppose your unit is a model that has six Inport blocks and two Outport blocks that connect to your component, this metric returns a **Result** array with a **Value** property of [6, 2].

See Also

Related Examples

- “Input and Output Component Interface Ports” on page 5-304
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

Simulink Design Cyclomatic Complexity

Metric ID

`slcomp.SLCyclomaticComplexity`

Description

The *design cyclomatic complexity* is the number of possible execution paths through a design. In general, the more paths there are through a design, the more complex the design is. When you keep the design cyclomatic complexity low, the design typically is easier to read, maintain, and test. The design cyclomatic complexity is calculated as the number of decision paths plus one. The metric adds one to account for the execution path represented by the default path. The design cyclomatic complexity includes the default path because the metric identifies each possible outcome from an execution path, including the default outcome.

Use the Simulink design cyclomatic complexity metric to determine the design cyclomatic complexity of the Simulink model components in your design.

This metric counts the total number of Simulink-based execution paths through a unit or component. The number of execution paths is calculated as the number of Simulink decisions, “Simulink Decision Count” on page 5-310, plus one for the default path. The default path is only counted once per unit or component.

To see the number of Simulink decisions associated with different block types, see “Decision Counts for Common Block Types” on page 5-310.

Computation Details

The metric does not exclude decisions associated with short-circuiting logical operations because Simulink runs a block regardless of whether or not a previous input alone determined the block output.

For example, in Simulink an AND block with repeating inputs runs regardless of whether or not the previous input alone determined the block output.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.SLCyclomaticComplexity`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the Simulink design cyclomatic complexity of a unit or component.

Examples

Suppose you have a unit that contains only an Abs block with an input `u`.

The Abs block can be treated as an `if-else` statement:

- If the input to the Abs block is less than zero, the output of the Abs block is the input multiplied by negative one.
- Otherwise, by default, the output of the Abs block is equal to the input of the Abs block.

```
if (input < 0)
    % one decision
    output = -1*input;
else
    % default path
    % zero decisions
    output = input;
end
```

For an `if-else` statement, the number of decisions is one because the `if` statement represents one decision and the `else` statement represents the default behavior. The execution path follows the default path if no decisions are made.

In this example, the number of Simulink decisions is 1 and therefore the Simulink design cyclomatic complexity of the unit is 2.

The value of the Simulink design cyclomatic complexity represents the two possible execution paths through the unit, either:

- $u < 0$ and the output of the Abs block is the input multiplied by negative 1
- $u \geq 0$ and the output of the Abs block is equal to the input of the Abs block

See Also

Related Examples

- “Simulink Decision Count” on page 5-310
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

Simulink Decision Count

Metric ID

slcomp.SimulinkDecisions

Description

Use this metric to count the number of Simulink decisions in the Simulink blocks in each layer of a unit or component.

Decision Counts for Common Block Types

The following table shows how the metric calculates the decision count for common Simulink blocks.

Decision Counts Table

Block	Decision Count	Description
Abs	1	<p>For the decision count, the Abs block can be treated as an if-else statement:</p> <ul style="list-style-type: none"> • If the input to the Abs block is less than zero, the output of the Abs block is the input multiplied by negative 1. • Otherwise, by default, the output of the Abs block is equal to the input of the Abs block. <pre> if (input < 0) % one decision output = -1*input; else % default path % zero decisions output = input; end </pre> <p>An if-else statement has a decision count of 1 because the if statement represents one decision and the else statement represents the default behavior (no decisions made).</p>

Block	Decision Count	Description
Cominatorial Logic	Is equal to the number of rows in the Truth table parameter) minus 1	<p>The decision count depends on the number of rows in the truth table.</p> <p>If a truth table contains five rows, the decision count is 4. One row in the truth table contains the default output and the other four rows contain potential outputs that depend on a decision being made.</p>
Dead Zone	2	<p>The output of the Dead Zone block depends on the block input (U) and the values of the Start of dead zone(lower limit, LL) and End of dead zone (upper limit, UL) parameters.</p> <p>For the decision count, the Dead Zone block can be treated as an if-elseif-else statement:</p> <ul style="list-style-type: none"> • If $U \geq LL$ and $U \leq UL$, the output is 0. • If $U > UL$, the output is $U - UL$. • Otherwise, the output is $U - LL$. <pre> if ((U >= LL) & (U <= UL)) % one decision output = 0; elseif (U > UL) % one decision output = U - UL; else % default path % zero decisions output = U - LL; end </pre> <p>An if-elseif-else statement has a decision count of 2 because the if statement represents one decision, the elseif statement represents one decision, and the else statement represents the default behavior (no decisions made).</p>

Block	Decision Count	Description
Discrete-Time Integrator	Is equal to either 0, 2, 3, or 5	<p>The decision count depends on the External reset parameter and the Limit output parameter.</p> <ul style="list-style-type: none"> • If External reset is set to none and the Limit output check box is unselected, the decision count is 0. This is the default behavior for the Discrete-Time Integrator block. • If External reset is set to a value other than none and the Limit output check box is unselected, the decision count is 2. • If External reset is set to none and the Limit output check box is selected, the decision count is 3. • If External reset is set to a value other than none and the Limit output check box is selected, the decision count is 5.
Enabled Subsystem	1	<p>For the decision count, the Enabled Subsystem block can be treated as an if-else statement:</p> <ul style="list-style-type: none"> • If the subsystem is enabled, the subsystem executes. • Otherwise, by default, the subsystem does not execute. <p>An if-else statement has a decision count of 1 because the if statement represents one decision and the else statement represents the default behavior (no decisions made).</p>

Block	Decision Count	Description
Enabled and Triggered Subsystem	1	<p>For the decision count, the Enabled and Triggered Subsystem block can be treated as an <code>if-else</code> statement:</p> <ul style="list-style-type: none">• If the subsystem is enabled and triggered, the subsystem executes.• Otherwise, by default, the subsystem does not execute. <p>An <code>if-else</code> statement has a decision count of 1 because the <code>if</code> statement represents one decision and the <code>else</code> statement represents the default behavior (no decisions made).</p>
For Iterator Subsystem	1	<p>For the decision count, the For Iterator block can be treated as an <code>if-else</code> statement:</p> <ul style="list-style-type: none">• If the iteration value is less than or equal to the iteration limit, iterate over blocks in the For Iterator Subsystem block.• Otherwise, the subsystem does not execute. <p>An <code>if-else</code> statement has a decision count of 1 because the <code>if</code> statement represents one decision and the <code>else</code> statement represents the default behavior (no decisions made).</p>

Block	Decision Count	Description
If	Is equal to the number of <code>if</code> and <code>elseif</code> expressions	<p>Each <code>if</code> statement represents a decision and each <code>elseif</code> statement represents a decision.</p> <p>The decision count is equal to the number of <code>if</code> and <code>elseif</code> statements.</p> <p>For example, the following code contains one <code>if</code> statement and one <code>elseif</code> statement and therefore has a decision count of 2:</p> <pre data-bbox="1057 724 1468 930"> if (u > 0) % one decision y = 1; elseif (u < 0) % one decision y = 2; else % default path, zero decisions y = 0; end </pre>
Index Vector	1	<p>The Index Vector block is associated with a decision count of 1.</p> <p>The Index Vector block is a special configuration of the Multiport Switch block in which you specify one data input and the control input is zero-based.</p> <p>For example, if the input vector is [18 15 17 10] and the control input is 3 (zero-based index), the output is 10. Using the control input to index into the input vector is considered a single decision.</p>

Block	Decision Count	Description
Multiport Switch	Is equal to the number of data ports minus 1	<p>The decision count for the Multiport Switch block depends on the number of data ports that are inputs to the block. The number of data ports represents the number of possible outcomes.</p> <p>The decision count equals the number of data ports minus 1 because one of the possible outcomes is the default path. For the default path, no decision was made.</p> <p>For example, if the number of data inputs is 3, then the decision count is 2.</p> <p>If there is only one data port, the Multiport Switch block acts as an Index Vector block and has a decision count of 1.</p>

Block	Decision Count	Description
Rate Limiter	2	<p>The output of the Rate Limiter block is determined by comparing <i>rate</i> to the Rising slew rate (R) and Falling slew rate (F) parameters.</p> <p>For the decision count, the Rate Limiter block can be treated as an if-elseif-else statement:</p> <ul style="list-style-type: none">• If <i>rate</i> is greater than R, the output is calculated using the rising slew rate.• If <i>rate</i> is less than F, the output is calculated using the falling slew rate.• Otherwise, the <i>rate</i> is between the bounds of R and F and the change in output is equal to the change in input. <p>An if-elseif-else statement has a decision count of 2 because the if statement represents one decision, the elseif statement represents one decision, and the else statement represents the default behavior (no decisions made).</p>

Block	Decision Count	Description
Relay	2	<p>The output of the Relay block is determined by the Switch off point and the Switch on point parameters.</p> <p>For the decision count, the Relay block can be treated as an if-elseif-else statement:</p> <ul style="list-style-type: none">• If the relay is on, the output remains on until the input drops below the value of the Switch off point parameter.• If the relay is off, the output remains off until the input exceeds the value of the Switch on point parameter.• Otherwise, the input falls between the Switch off point and the Switch on point and the output is unchanged. <p>An if-elseif-else statement has a decision count of 2 because the if statement represents one decision, the elseif statement represents one decision, and the else statement represents the default behavior (no decisions made).</p>

Block	Decision Count	Description
Saturation	2	<p>The output of the Saturation block is determined by comparing the input to the Upper limit and Lower limit parameters.</p> <p>For the decision count, the Saturation block can be treated as an <code>if-elseif-else</code> statement:</p> <ul style="list-style-type: none">• If the input is less than the Lower limit, the output is equal to the value of the Lower limit.• If the input is greater than the Upper limit, the output is equal to the value of the Upper limit.• Otherwise, the output is equal to the value of the input. <p>An <code>if-elseif-else</code> statement has a decision count of 2 because the <code>if</code> statement represents one decision, the <code>elseif</code> statement represents one decision, and the <code>else</code> statement represents the default behavior (no decisions made).</p>

Block	Decision Count	Description
Switch	1	<p>The Switch block is associated with a decision count of 1:</p> <ul style="list-style-type: none"> • If input 2 (the control input) satisfies the selected criterion, the output is equal to input 1. • Otherwise, the output is equal to input 3. <p>For example, the following code shows a decision count of 1:</p> <pre>switch (u2 > 0) case 1 % one decision y = u1; otherwise % default path, zero decision y = u3; end</pre>
Switch Case	Is equal to the number of outputs minus 1	<p>The default case represents the default behavior (no decisions made). But each subsequent case represents a decision.</p> <p>The decision count is equal to the number of outputs minus 1.</p> <p>For example, the following code represents a Switch Case block with 3 outputs and has a decision count of 2:</p> <pre>switch u1 case 1 % one decision y = port1; case 2 % one decision y = port2; otherwise % default path, zero decision y = port3; end</pre>

Block	Decision Count	Description
Triggered Subsystem	1	<p>For the decision count, the Triggered Subsystem block can be treated as an <code>if-else</code> statement:</p> <ul style="list-style-type: none"> • If the subsystem is triggered, the subsystem executes. • Otherwise, by default, the subsystem does not execute. <p>An <code>if-else</code> statement has a decision count of 1 because the <code>if</code> statement represents one decision and the <code>else</code> statement represents the default behavior (no decisions made).</p>

Computation Details

The metric does not exclude decisions associated with short-circuiting logical operations because Simulink runs a block regardless of whether or not a previous input alone determined the block output.

For example, in Simulink an AND block with repeating inputs runs regardless of whether or not the previous input alone determined the block output.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.SimulinkDecisions`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the number of Simulink decisions in the Simulink blocks in each layer of a unit or component.

Examples

To see the number of Simulink decisions associated with different block types, see “Decision Counts for Common Block Types”.

See Also

Related Examples

- “Simulink Design Cyclomatic Complexity” on page 5-308
- “Simulink Decision Distribution” on page 5-322

- “Collect Model Maintainability Metrics Programmatically” on page 5-150

Simulink Decision Distribution

Metric ID

`slcomp.SimulinkDecisionDistribution`

Description

Use this metric to determine the distribution of the number of Simulink decisions in a unit or component. For information on how the metric calculates the number of Simulink decisions, see “Simulink Decision Count”.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.SimulinkDecisionDistribution`.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of artifacts in each bin, returned as an integer vector.
- `BinEdges` — Bin edges for the number of Simulink decisions, returned as an integer vector. `BinEdges(1)` is the left edge of the first bin and `BinEdges(end)` is the right edge of the last bin. The length of `BinEdges` is one more than the length of `BinCounts`.

The bins in this metric result correspond to the bins in the **Simulink** row and **Distribution** column in the **Design Cyclomatic Complexity Breakdown** section.

Examples

Suppose your unit has:

- 16 model layers that each make between zero and nine Simulink decisions
- 41 model layers that each make between 10 and 19 Simulink decisions
- 100 model layers that each make between 20 and 29 Simulink decisions

The `Value` structure contains:

```
ans =  
  
    struct with fields:  
  
    BinCounts: [16 41 100 0 0 0 0 0 0 0]  
    BinEdges: [0 10 20 30 40 50 60 70 80 90 18446744073709551615]
```

There are 10 bins in the Simulink decision distribution. `BinCounts` shows the number of model layers in each bin and `BinEdges` shows the edges of each bin. The last bin edge is 18446744073709551615, which is the upper limit of the decision count.

For this example, there are 16 model layers in the first bin, 41 model layers in the second bin, and 100 model layers in the third bin, and no model layers in the other seven bins.

You can view the distribution bins in the **Model Maintainability Dashboard**. Point to a distribution bin to see tooltip information on the number of model layers and decisions associated with the bin.

See Also

Related Examples

- “Simulink Decision Count” on page 5-310
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

Stateflow Design Cyclomatic Complexity

Metric ID

`slcomp.SFCyclomaticComplexity`

Description

The *design cyclomatic complexity* is the number of possible execution paths through a design. In general, the more paths there are through a design, the more complex the design is. When you keep the design cyclomatic complexity low, the design typically is easier to read, maintain, and test. The design cyclomatic complexity is calculated as the number of decision paths plus one. The metric adds one to account for the execution path represented by the default path. The design cyclomatic complexity includes the default path because the metric identifies each possible outcome from an execution path, including the default outcome.

Use the Stateflow design cyclomatic complexity metric to determine the design cyclomatic complexity for the Stateflow components in your design.

This metric counts the total number of Stateflow-based execution paths through a unit or component. The total number of execution paths is calculated as the number of Stateflow decisions, “Stateflow Decision Count” on page 5-326, plus one for the default path. The default path is only counted once per unit or component.

To see the number of Stateflow decisions associated with different Stateflow components, see “Decision Counts for Common Stateflow Components” on page 5-326.

Computation Details

The metric:

- Only supports Stateflow objects that use MATLAB as the action language.
- Analyzes Stateflow charts that use C as the action language, but does not represent decisions from short-circuiting in logical operations.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.SFCyclomaticComplexity`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the Stateflow design cyclomatic complexity of a unit or component.

Examples

Suppose you have a unit that contains only a Truth Table block from the Stateflow library. By default, the Truth Table block contains a **Condition Table** with two condition rows and three decision columns and an **Action Table** with two actions.

To see the number of Stateflow decisions associated with different Stateflow components, see “Decision Counts for Common Stateflow Components” on page 5-326.

For the Truth Table block, the number of graphical decisions is calculated as the number of conditions (rows) multiplied by the number of decisions (columns) in the **Condition Table**. In this example, the number of graphical decisions is equal to 2 multiplied by 3.

For the Truth Table block, the number of MATLAB code decisions is calculated as the number of decisions in the **Action Table**. In this example, there are no decisions made in the code of the **Action Table**.

The Stateflow design cyclomatic complexity is equal to the sum of the number of graphical Stateflow decisions, plus the number of MATLAB code decisions in the Stateflow component, plus one (for the default path). In this case, the Stateflow design cyclomatic complexity is calculated as $6 + 0 + 1$ which equals seven.

See Also

Related Examples

- “Stateflow Decision Count” on page 5-326
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

Stateflow Decision Count

Metric ID

slcomp.StateflowDecisions

Description

Use this metric to count the number of Stateflow decisions in the Stateflow model components in each layer of a unit or component.

Decision Counts for Common Stateflow Components

The metric returns both graphical decisions and MATLAB code decisions. *Graphical decisions* are decisions based on the connections drawn graphically in a Stateflow chart. *MATLAB code decisions* are decisions based on code written in the MATLAB action language used in a Stateflow chart.

The following table shows how the metric calculates the decision count for common Stateflow components.

Decision Counts Table

Component	Graphical Decision Count	MATLAB Code Decision Count
Transitions	Is equal to the number of transitions with conditions	Is equal to the number of short-circuit operations (&& and) in conditions
States	Is equal to the number of states minus the default state	Is equal to the number of MATLAB decisions in entry, during, and exit actions For more information, see "MATLAB Decision Count".
Truth Tables	Is equal to the number of conditions multiplied by the number of decisions in the Condition Table	Is equal to the number of code decisions in the Action Table

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.StateflowDecisions`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the number of Stateflow decisions in each layer of a unit or component.

Examples

To see the number of Stateflow decisions associated with different Stateflow components, see “Decision Counts for Common Stateflow Components”.

See Also

Related Examples

- “Stateflow Design Cyclomatic Complexity” on page 5-324
- “Stateflow Decision Distribution” on page 5-328
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

Stateflow Decision Distribution

Metric ID

`slcomp.StateflowDecisionsDistribution`

Description

Use this metric to determine the distribution of the number of Stateflow decisions in the Stateflow model components in each layer of a unit or component. For information on how the number of Stateflow decisions is calculated, see “Stateflow Decision Count” on page 5-326.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.StateflowDecisionsDistribution`.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of artifacts in each bin, returned as an integer vector.
- `BinEdges` — Bin edges for the number of Stateflow decisions, returned as an integer vector. `BinEdges(1)` is the left edge of the first bin and `BinEdges(end)` is the right edge of the last bin. The length of `BinEdges` is one more than the length of `BinCounts`.

The bins in this metric result correspond to the bins in the **Stateflow** row and **Distribution** column in the **Design Cyclomatic Complexity Breakdown** section.

See Also

Related Examples

- “Stateflow Decision Count” on page 5-326
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

MATLAB Design Cyclomatic Complexity

Metric ID

slcomp.MatlabCyclomaticComplexity

Description

The *design cyclomatic complexity* is the number of possible execution paths through a design. In general, the more paths there are through a design, the more complex the design is. When you keep the design cyclomatic complexity low, the design typically is easier to read, maintain, and test. The design cyclomatic complexity is calculated as the number of decision paths plus one. The metric adds one to account for the execution path represented by the default path. The design cyclomatic complexity includes the default path because the metric identifies each possible outcome from an execution path, including the default outcome.

Use this metric to determine the design cyclomatic complexity of the MATLAB code in your design.

The MATLAB design cyclomatic complexity is the total number of execution paths through the MATLAB code in a unit or component. The number of execution paths is calculated as the number of MATLAB decisions, “MATLAB Decision Count” on page 5-331, plus one for the default path. The default path is only counted once per unit or component.

Computation Details

The metric includes the decisions associated with logical operations that might be short-circuited during code execution.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.MatlabCyclomaticComplexity`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the MATLAB design cyclomatic complexity of a unit or component.

Examples

Suppose you have a unit that contains only a MATLAB Function block with this code:

```
function y = fcn(u)
    if u < 0
        % one decision
        y = -1*u;
    else
        % default path
        % zero decisions
```

```
        y = u;  
    end  
end
```

For an `if-else` statement, the number of decisions is one because the `if` statement represents one decision and the `else` statement represents the default behavior. The execution path follows the default path if no decisions are made.

- If the input to the MATLAB Function block is less than zero, the output of the MATLAB Function block is the input multiplied by negative one.
- Otherwise, by default, the output of the MATLAB Function block is equal to the input of the MATLAB Function block.

In this example, the number of MATLAB decisions is one and therefore the MATLAB design cyclomatic complexity of the unit is two. The default path is not included in the number of decisions because no decision is made to reach a default state, but the default path is included in the design cyclomatic complexity.

The value of the MATLAB design cyclomatic complexity represents the two possible execution paths through the unit, either:

- $u < 0$ and the output of the MATLAB Function block is the input multiplied by negative one
- $u \geq 0$ and the output of the MATLAB Function block is equal to the input of the MATLAB Function block

See Also

Related Examples

- “MATLAB Decision Count” on page 5-331
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

MATLAB Decision Count

Metric ID

slcomp.MATLABDecisions

Description

Use this metric to count the number of decisions in MATLAB code associated with a unit or component.

This metric analyzes:

- MATLAB functions in Simulink, including MATLAB Function blocks and Interpreted MATLAB Function blocks
- MATLAB functions in Stateflow objects
- Functions and methods in MATLAB files

Decision Counts for Common MATLAB Keywords

The following table shows how the metric calculates the decision count for common MATLAB keywords.

Decision Counts Table

Keyword	Decision Count	Description
case	Is equal to the number of case statements	Each case statement represents a decision in the code.
catch	0	The catch statement represents a default behavior. The default path is not included in the number of decisions because no decision is made to reach a default state.
else	0	The else statement represents a default behavior. The default path is not included in the number of decisions because no decision is made to reach a default state.
elseif	Is equal to the number of elseif statements	Each elseif statement represents a decision in the code.
if	Is equal to the number of if statements	Each if statement represents a decision in the code.

Keyword	Decision Count	Description
otherwise	0	The <code>otherwise</code> statement represents a default behavior. The default path is not included in the number of decisions because no decision is made to reach a default state.
parfor	Is equal to the number of <code>parfor</code> statements	Each <code>parfor</code> statement represents a decision in the code.
try	Is equal to the number of <code>try</code> statements	Each <code>try</code> statement represents a decision in the code.
while	Is equal to the number of <code>while</code> statements	Each <code>while</code> statement represents a decision in the code.

Computation Details

The metric:

- Calculates the number of decisions in MATLAB code by taking the cyclomatic complexity calculated by the Code Analyzer and subtracting one to exclude the default path. For more information, see `checkcode` and “Measure Code Complexity Using Cyclomatic Complexity”.
- Counts decisions from `if`, `elseif`, `while`, `for`, `parfor`, `try`, and `case` statements.
- Ignores `else`, `otherwise`, and `catch` statements because they form the default path.

Collection

To collect data for this metric:

- In the Model Maintainability Dashboard, in the **Design Cyclomatic Complexity Breakdown** section, click the **Run metrics for widget** icon. The distribution of the decisions appears in the **MATLAB** row and **Distribution** column. To view a table that shows the MATLAB decision count for each model component, click one of the bins in the distribution.
- Use `getMetrics` with the metric identifier `slcomp.MATLABDecisions`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the number of MATLAB decisions in the MATLAB code associated with each layer of a unit or component.

Examples

Suppose you have a unit that contains only a MATLAB Function block with this code:

```
function y = fcn(u)
    if u < 0
        % one decision
        y = -1*u;
```

```

else
    % default path
    % zero decisions
    y = u;
end
end

```

For an `if-else` statement, the number of decisions is one because the `if` statement represents one decision and the `else` statement represents the default behavior. The execution path follows the default path if no decisions are made.

- If the input to the MATLAB Function block is less than zero, the output of the MATLAB Function block is the input multiplied by negative one.
- Otherwise, by default, the output of the MATLAB Function block is equal to the input of the MATLAB Function block.

In this example, the number of MATLAB decisions is one.

Suppose the code includes `elseif` statements:

```

function y = fcn(u)
    if u < 0
        % one decision
        y = -1*u;
    elseif u == 1
        % one decision
        y = 1;
    elseif u == 2
        % one decision
        y = 2;
    else
        % default path
        % zero decisions
        y = u;
    end
end

```

The number of MATLAB decisions increases by one for each additional `elseif` statement in the code. MATLAB code that contains one `if` statement, two `elseif` statements, and one `else` statement contains three decisions. The `else` statement does not contribute to the number of decisions because the `else` statement is part of the default path. The default path is not included in the number of decisions because no decision is made to reach a default state.

See Also

Related Examples

- “MATLAB Design Cyclomatic Complexity” on page 5-329
- “MATLAB Decision Distribution” on page 5-334
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

MATLAB Decision Distribution

Metric ID

`slcomp.MATLABDecisionsDistribution`

Description

Use this metric to determine the distribution of the number of decisions in MATLAB code.

This metric analyzes the MATLAB functions and methods in:

- MATLAB files
- MATLAB Function blocks

For information on how the number of MATLAB decisions is calculated, see “MATLAB Decision Count” on page 5-331.

Collection

To collect data for this metric:

- In the Model Maintainability Dashboard, in the **Design Cyclomatic Complexity Breakdown** section, click the **Run metrics for widget** icon. The distribution of decisions appears in the **MATLAB** row and **Distribution** column.
- Use `getMetrics` with the metric identifier `slcomp.MATLABDecisionsDistribution`.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of artifacts in each bin, returned as an integer vector.
- `BinEdges` — Bin edges for the number of MATLAB decisions, returned as an integer vector. `BinEdges(1)` is the left edge of the first bin and `BinEdges(end)` is the right edge of the last bin. The length of `BinEdges` is one more than the length of `BinCounts`.

The bins in this metric result correspond to the bins in the **MATLAB** row and **Distribution** column in the **Design Cyclomatic Complexity Breakdown** section.

See Also

Related Examples

- “MATLAB Decision Count” on page 5-331
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

Overall Blocks

Metric ID

`slcomp.OverallBlocks`

Description

Use this metric to count the total number of blocks in a unit or component.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.OverallBlocks`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the number of overall blocks that a unit or component contains.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Simulink Blocks” on page 5-336
- “Simulink Blocks Distribution” on page 5-337

Simulink Blocks

Metric ID

`slcomp.SimulinkBlocks`

Description

Use this metric to count the of the number of Simulink blocks, excluding Inport, Outport, and Goto blocks, in each layer of the model hierarchy for a unit or component.

Computation Details

The metric:

- Does not include Inport, Outport, and Goto blocks in the block count.
- Counts the blocks, excluding Inport, Outport, and Goto blocks, in each variant.
- Does not count blocks that are commented out.

Collection

To collect data for this metric:

- In the Model Maintainability Dashboard, point to the **Simulink Architecture** section and click the **Run metrics for widget** icon. If you click on the widget in the **Blocks** row and **Count** column, you can view a table that shows the number of Simulink blocks, excluding Inport, Outport, and Goto blocks, in each layer of a unit or component.
- Use `getMetrics` with the metric identifier `slcomp.SimulinkBlocks`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the number of Simulink blocks, excluding Inport, Outport, and Goto blocks, in each layer of a unit or component.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Overall Blocks” on page 5-335
- “Simulink Blocks Distribution” on page 5-337

Simulink Blocks Distribution

Metric ID

slcomp.BlocksDistribution

Description

Use this metric to determine the distribution of the number of Simulink blocks, excluding Inport, Outport, and Goto blocks, in each layer of the model hierarchy for a unit or component.

Computation Details

The metric:

- Does not include Inport, Outport, and Goto blocks in the block count.
- Counts the blocks, excluding Inport, Outport, and Goto blocks, in each variant.
- Does not count blocks that are commented out.

Collection

To collect data for this metric:

- In the Model Maintainability Dashboard, point to the **Simulink Architecture** section and click the **Run metrics for widget** icon. See the results in the **Blocks** row and **Distribution** column.
- Use `getMetrics` with the metric identifier `slcomp.BlocksDistribution`.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of artifacts in each bin, returned as an integer vector.
- `BinEdges` — Bin edges for the number of blocks, excluding Inport, Outport, and Goto blocks, in each layer of a unit or component, returned as an integer vector. `BinEdges(1)` is the left edge of the first bin and `BinEdges(end)` is the right edge of the last bin. The length of `BinEdges` is one more than the length of `BinCounts`.

The bins in this metric result correspond to the bins in the **Blocks** row and **Distribution** column in the **Simulink Architecture** section.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Overall Blocks” on page 5-335

- “Simulink Blocks” on page 5-336

Overall Signal Lines

Metric ID

`slcomp.OverallSignalLines`

Description

Use this metric to count the total number of signal lines in a unit or component.

The metric uses `find_system` to find the signal lines.

The metric includes:

- Simulink signal lines that the model uses
- Commented lines
- Each individual branch line
- Unterminated lines

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.OverallSignalLines`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the overall number of signal lines in a unit or component.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Simulink Signal Lines” on page 5-340
- “Simulink Signal Lines Distribution” on page 5-341

Simulink Signal Lines

Metric ID

`slcomp.SimulinkSignalLines`

Description

Use this metric to count the number of signal lines in each layer of a unit or component.

The metric uses `find_system` to find the signal lines.

The metric includes:

- Simulink signal lines that the model uses
- Commented lines
- Each individual branch line
- Unterminated lines

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.SimulinkSignalLines`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the number of signal lines in each layer of a unit or component.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Overall Signal Lines” on page 5-339
- “Simulink Signal Lines Distribution” on page 5-341

Simulink Signal Lines Distribution

Metric ID

`slcomp.SignalLinesDistribution`

Description

Use this metric to determine the distribution of the number of signal lines in a unit or component.

The metric uses `find_system` to find the signal lines.

The metric includes:

- Simulink signal lines that the model uses
- Commented lines
- Each individual branch line
- Unterminated lines

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.SignalLinesDistribution`.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of artifacts in each bin, returned as an integer vector.
- `BinEdges` — Bin edges for the number of signal lines, returned as an integer vector. `BinEdges(1)` is the left edge of the first bin and `BinEdges(end)` is the right edge of the last bin. The length of `BinEdges` is one more than the length of `BinCounts`.

The bins in this metric result correspond to the bins in the **Signal Lines** row and **Distribution** column in the **Simulink Architecture** section.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Overall Signal Lines” on page 5-339
- “Simulink Signal Lines” on page 5-340

Overall Goto Blocks

Metric ID

`slcomp.OverallGotos`

Description

Use this metric to count the total number of Goto blocks in a unit or component.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.OverallGotos`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the overall number of Goto blocks in a unit or component.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Simulink Goto Blocks” on page 5-343
- “Simulink Goto Blocks Distribution” on page 5-344

Simulink Goto Blocks

Metric ID

slcomp.SimulinkGotos

Description

Use this metric to count the number of Simulink Goto blocks in each layer of a unit or component.

Collection

To collect data for this metric:

- In the Model Maintainability Dashboard, point to the **Simulink Architecture** section and click the **Run metrics for widget** icon. If you click on the widget in the **Gotos** row and **Count** column, you can view a table that shows the number of Goto blocks in each layer of a unit or component.
- Use `getMetrics` with the metric identifier `slcomp.SimulinkGotos`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the number of Goto blocks in each layer of a unit or component.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Overall Goto Blocks” on page 5-342
- “Simulink Goto Blocks Distribution” on page 5-344

Simulink Goto Blocks Distribution

Metric ID

slcomp.GotosDistribution

Description

Use this metric to determine the distribution of the number of Simulink Goto blocks in each layer of a unit or component.

Collection

To collect data for this metric:

- In the Model Maintainability Dashboard, point to the **Simulink Architecture** section and click the **Run metrics for widget** icon.
- Use `getMetrics` with the metric identifier `slcomp.GotosDistribution`.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of artifacts in each bin, returned as an integer vector.
- `BinEdges` — Bin edges for the number of Goto blocks in each layer of a unit or component, returned as an integer vector. `BinEdges(1)` is the left edge of the first bin and `BinEdges(end)` is the right edge of the last bin. The length of `BinEdges` is one more than the length of `BinCounts`.

The bins in this metric result correspond to the bins in the **Gotos** row and **Distribution** column in the **Simulink Architecture** section.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Overall Goto Blocks” on page 5-342
- “Simulink Goto Blocks” on page 5-343

Overall Transitions

Metric ID

`slcomp.OverallTransitions`

Description

Use this metric to count the total number of transitions in a unit or component.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.OverallTransitions`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the overall number of transitions in a unit or component.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Stateflow Transitions” on page 5-346
- “Stateflow Transitions Distribution” on page 5-347

Stateflow Transitions

Metric ID

`slcomp.StateflowTransitions`

Description

Use this metric to count the number of Stateflow transitions in the Stateflow model components in each layer of a unit or component.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.StateflowTransitions`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the number of Stateflow transitions in each layer of a unit or component.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Overall Transitions” on page 5-345
- “Stateflow Transitions Distribution” on page 5-347

Stateflow Transitions Distribution

Metric ID

`slcomp.TransitionsDistribution`

Description

Use this metric to determine the distribution of the number of Stateflow transitions in the Stateflow model components in each layer of a unit or component.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.TransitionsDistribution`.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of artifacts in each bin, returned as an integer vector.
- `BinEdges` — Bin edges for the number of Stateflow transitions, returned as an integer vector. `BinEdges(1)` is the left edge of the first bin and `BinEdges(end)` is the right edge of the last bin. The length of `BinEdges` is one more than the length of `BinCounts`.

The bins in this metric result correspond to the bins in the **Transitions** row and **Distribution** column in the **Stateflow Architecture** section.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Overall Transitions” on page 5-345
- “Stateflow Transitions” on page 5-346

Overall States

Metric ID

`slcomp.OverallStates`

Description

Use this metric to count the total number of states in a unit or component.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.OverallStates`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the overall number of states in a unit or component.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Stateflow States” on page 5-349
- “Stateflow States Distribution” on page 5-350

Stateflow States

Metric ID

`slcomp.StateflowStates`

Description

Use this metric to count the number of Stateflow states in the Stateflow model components in each layer of a unit or component.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.StateflowStates`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the number of Stateflow states in each layer of a unit or component.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Overall States” on page 5-348
- “Stateflow States Distribution” on page 5-350

Stateflow States Distribution

Metric ID

`slcomp.StatesDistribution`

Description

Use this metric to determine the distribution of the number of Stateflow states in the Stateflow model components in each layer of a unit or component.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.StatesDistribution`.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of artifacts in each bin, returned as an integer vector.
- `BinEdges` — Bin edges for the number of Stateflow states, returned as an integer vector. `BinEdges(1)` is the left edge of the first bin and `BinEdges(end)` is the right edge of the last bin. The length of `BinEdges` is one more than the length of `BinCounts`.

The bins in this metric result correspond to the bins in the **States** row and **Distribution** column in the **Stateflow Architecture** section.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Overall States” on page 5-348
- “Stateflow States” on page 5-349

Overall MATLAB Executable Lines of Code (eLOC)

Metric ID

`slcomp.OverallMATLABeLOC`

Description

Use this metric to count the total number of executable lines of MATLAB code in a unit or component. Effective lines of MATLAB code are lines of executable code.

The metric does not consider the following to be effective lines of code:

- empty lines
- lines that contain only comments
- lines that contain only white spaces
- lines that contain only an end statement

For example, suppose a unit contains only two MATLAB Function blocks: `f1` and `f2`. If `f1` contains 100 effective lines of code and `f2` contains 50 effective lines of code, the overall number of executable lines of MATLAB code in the unit is 150.

Collection

To collect data for this metric, use `getMetrics` with the metric identifier `slcomp.OverallMATLABeLOC`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the overall number of executable lines of MATLAB code in a unit or component.

See Also

Related Examples

- “MATLAB Effective Lines of Code (eLOC)” on page 5-352
- “Collect Model Maintainability Metrics Programmatically” on page 5-150

MATLAB Effective Lines of Code (eLOC)

Metric ID

`slcomp.MATLABeLOC`

Description

Use this metric to count the number of effective lines of MATLAB code. Effective lines of MATLAB code are lines of executable code.

The metric does not consider the following to be effective lines of code:

- empty lines
- lines that only contain comments
- lines that only contain white spaces
- lines that only contain an end statement

To view the total number of lines of MATLAB code associated with a unit or component, see “Overall MATLAB Executable Lines of Code (eLOC)” on page 5-351.

Collection

To collect data for this metric:

- In the Model Maintainability Dashboard, in the **MATLAB Architecture** section, click the **Run metrics for widget** icon. The distribution of the effective lines of MATLAB code appears in the **Lines of Code** row and **Distribution** column. To view a table that shows the effective lines of MATLAB-based code for each model component, click one of the bins in the distribution.
- Use `getMetrics` with the metric identifier `slcomp.MATLABeLOC`.

Results

For this metric, instances of `metric.Result` return `Value` as the integer value for the number of lines of effective MATLAB code in a unit or component.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150
- “Overall MATLAB Executable Lines of Code (eLOC)” on page 5-351
- “MATLAB Effective Lines of Code (eLOC) Distribution” on page 5-353

MATLAB Effective Lines of Code (eLOC) Distribution

Metric ID

slcomp.MATLABeLOCdistribution

Description

Use this metric to determine the distribution of the number of effective lines of MATLAB code. Effective lines of MATLAB code are lines of executable code.

The metric does not consider the following to be effective lines of code:

- empty lines
- lines that only contain comments
- lines that only contain white spaces
- lines that only contain an end statement

To view the number of lines of MATLAB code associated with each model component, see “MATLAB Effective Lines of Code (eLOC)” on page 5-352.

Collection

To collect data for this metric:

- In the Model Maintainability Dashboard, in the **MATLAB Architecture** section, click the **Run metrics for widget** icon. The distribution of effective lines of code appears in the **Lines of Code** row and **Distribution** column.
- Use `getMetrics` with the metric identifier `slcomp.MATLABeLOCdistribution`.

Results

For this metric, instances of `metric.Result` return `Value` as a distribution structure that contains these fields:

- `BinCounts` — The number of artifacts in each bin, returned as an integer vector.
- `BinEdges` — Bin edges for the number of effective lines of MATLAB code, returned as an integer vector. `BinEdges(1)` is the left edge of the first bin and `BinEdges(end)` is the right edge of the last bin. The length of `BinEdges` is one more than the length of `BinCounts`.

The bins in this metric result correspond to the bins in the **Lines of Code** row and **Distribution** column in the **MATLAB Architecture** section.

See Also

Related Examples

- “Collect Model Maintainability Metrics Programmatically” on page 5-150

- “Overall MATLAB Executable Lines of Code (eLOC)” on page 5-351
- “MATLAB Effective Lines of Code (eLOC)” on page 5-352

Simulink Block Metric

Metric Information

Metric ID: `mathworks.metrics.SimulinkBlockCount`

Description

Metric Type: Size

Model Advisor Check ID: `mathworks.metricchecks.SimulinkBlockCount`

Use this metric to calculate the number of blocks in the model. The results provide the number of blocks at the model and subsystem level. This metric counts Simulink—based blocks, but does not include underlying blocks used to implement the block. This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.SimulinkBlockCount`.

The `slmetric.metric.AggregationMode` property setting is `Sum`.

Computation Details

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

Collection

To collect data for this metric using the Model Advisor, run the check, **Simulink block metric** in **By Task > Model Metrics > Count Metrics**. The Model Advisor check displays the number of blocks in the model or the subsystem. The check does not analyze referenced models or return aggregated results.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Number of blocks.
- `AggregatedValue`: Number of blocks for component and its subcomponents.
- `Measures`: Not applicable.

Note The results from metric analysis of **Simulink block metric** can differ from calling `sliagnostics`. The result of the Simulink block metric:

- Includes referenced models.

- Does not include any underlying blocks used to implement a MathWorks block that you used from the Simulink Library Browser.
 - Does not include links into MathWorks libraries, which means that MathWorks library blocks that are masked subsystems are counted as one block. The inner content of those blocks is not counted.
 - Does not include hidden content under Stateflow Charts or MATLAB Function blocks.
 - Does not include requirements blocks.
-

Subsystem Metric

Metric Information

Metric ID: `mathworks.metrics.SubSystemCount`

Description

Metric Type: Size

Model Advisor Check ID: `mathworks.metricchecks.SubSystemCount`

Use this metric to calculate the number of subsystems in the model. The results provide the number of subsystems at the model and subsystem level.

This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.SubSystemCount`.

The `slmetric.metric.AggregationMode` property setting is `Sum`.

Computation Details

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- Does not count subsystems linked to MathWorks libraries.
- If specified, analyzes the content of library-linked blocks or referenced models.

Collection

To collect data for this metric using the Model Advisor, run the check, **Subsystem metric in By Task > Model Metrics > Count Metrics**. The Model Advisor check displays the number of subsystems in the model or the subsystem. The check does not analyze referenced models or return aggregated results.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Number of subsystems.
- `AggregatedValue`: Number of subsystems for a component and its subcomponent.
- `Measures`: Not applicable.

Library Link Metric

Metric Information

Metric ID: `mathworks.metrics.LibraryLinkCount`

Description

Metric Type: Size

Model Advisor Check ID: `mathworks.metricchecks.LibraryLinkCount`

Use this metric to calculate the number of library-linked blocks in the model. The results provide the number of library-linked blocks at the model and subsystem level.

This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.LibraryLinkCount`.

The `slmetric.metric.AggregationMode` property setting is `Sum`.

Computation Details

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- Does not count subsystems linked to MathWorks libraries.
- If specified, analyzes the content of library-linked blocks or referenced models.

Collection

To collect results for this metric using the Model Advisor, run the check, **Library link metric** in **By Task > Model Metrics > Count Metrics**. The Model Advisor check displays the number of library links in the model or the subsystem. The check does not analyze referenced models or return aggregated results.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Number of library linked blocks.
- `AggregatedValue`: Number of library linked blocks for a component and its subcomponents.
- `Measures`: Not applicable.

Effective Lines of MATLAB Code Metric

Metric Information

Metric ID: `mathworks.metrics.MatlabLOCCount`

Description

Metric Type: Size

Model Advisor Check ID: `mathworks.metricchecks.MatlabLOCCount`

Run this metric to calculate the number of effective lines of MATLAB code. Effective lines of MATLAB code are lines of executable code. Empty lines, lines that contain only comments, and lines that contain only an end statement are not considered effective lines of code. The results provide the number of effective lines of MATLAB code for each MATLAB Function block and for MATLAB functions in Stateflow charts.

This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.MatlabLOCCount`.

The `slmetric.metric.AggregationMode` property setting is `Sum`.

Computation Details

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- Does not analyze the content of MATLAB code in external files.
- If specified, analyzes the content of library-linked blocks or referenced models.

Collection

To collect results for this metric using the Model Advisor, run the check, **Effective lines of MATLAB code metric** in **By Task > Model Metrics > Count Metrics**. The Model Advisor check displays the number of effective lines of MATLAB code for each MATLAB Function block and for MATLAB functions in Stateflow charts in the model. The check does not analyze referenced models or return aggregated results.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- **Value:** Number of effective lines of MATLAB code.
- **AggregatedValue:** Number of effective lines of MATLAB code for a component and its subcomponents.

- Measures: Not applicable.

Stateflow Chart Objects Metric

Metric Information

Metric ID: `mathworks.metrics.StateflowChartObjectCount`

Description

Metric Type: Size

Model Advisor Check ID: `mathworks.metricchecks.StateflowChartObjectCount`

Run this metric to calculate the number of Stateflow objects. For each chart in the model, the results provide the number of the following Stateflow objects:

- Atomic subcharts
- Boxes
- Data objects
- Events
- Graphical functions
- Junctions
- Linked charts
- MATLAB functions
- Notes
- Simulink functions
- States
- Transitions
- Truth tables

This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.StateflowChartObjectCount`.

The `slmetric.metric.AggregationMode` property setting is `Sum`.

Computation Details

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

Collection

To collect results for this metric using the Model Advisor, run the check, **Stateflow chart objects metric** in **By Task > Model Metrics > Count Metrics**. The Model Advisor check displays the

number of Stateflow objects in each chart in the model. The check does not analyze charts in referenced models or return aggregated results.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Number of Stateflow objects.
- `AggregatedValue`: Number of Stateflow objects for a component and its subcomponents.
- `Measures`: Not applicable.

Lines of Code for Stateflow Blocks Metric

Metric Information

Metric ID: `mathworks.metrics.StateflowLOCCount`

Description

Metric Type: Size

Model Advisor Check ID: `mathworks.metricchecks.StateflowLOCCount`

Use this metric to calculate the number of effective lines of code in Stateflow. Effective lines of MATLAB code are lines of executable code. Empty lines, lines that contain only comments, and lines that contain only an end statement are not considered effective lines of code. This metric calculates the lines of code for the following Stateflow blocks in the model:

- Chart, counting the code on Transitions and inside States
- State Transition Table block
- Truth Table block

This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.StateflowLOCCount`.

The `slmetric.metric.AggregationMode` property setting is `Sum`.

Computation Details

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

Collection

To collect results for this metric using the Model Advisor, run the check, **Lines of code for Stateflow blocks metric** in **By Task > Model Metrics > Count Metrics**. The Model Advisor check displays the number of code lines for Stateflow blocks in the model. The check does not analyze referenced models or return aggregated results.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Number of Stateflow block code lines.
- `AggregatedValue`: Number of Stateflow block code lines for a component and its subcomponents.

- **Measures:** Vector with two entries: number of effective lines of code in MATLAB action language and number of effective lines of code in C action language.

Subsystem Depth Metric

Metric Information

Metric ID: `mathworks.metrics.SubSystemDepth`

Description

Metric Type: Size

Model Advisor Check ID: `mathworks.metricchecks.SubSystemDepth`

Use this metric to count the relative depth of all hierarchical children for a given subsystem or model starting from the given component, or root of analysis. Depth traversal analysis stops when it reaches a referenced model or a library. Depth is restarted with 0 for each of these components.

This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.SubSystemDepth`.

The `slmetric.metric.AggregationMode` property setting is `None`.

Computation Details

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

Collection

To collect results for this metric using the Model Advisor, run the check, **Subsystem depth metric** in **By Task > Model Metrics > Count Metrics**. The Model Advisor check displays the depth of the subsystems in the model. The check does not analyze referenced models or return aggregated results.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: subsystem depth for each component in the hierarchy.
- `AggregatedValue`: Not applicable.
- `Measures`: Not applicable.

Input Output Metric

Metric Information

Metric ID: `mathworks.metrics.IOCount`

Description

Metric Type: Size

Use this metric to calculate the number of inputs and outputs in the model, which include:

- Inputs: Inport blocks, Trigger ports, Enable ports, chart input data and events.
- Outputs: Outport blocks, chart output data and events.
- Implicit inputs: From block, where the matching Goto block is outside of the component.
- Implicit outputs: Goto block, where the matching From block is outside of the component.

The `slmetric.metric.AggregationMode` property setting is Max.

Computation Details

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- **Value:** total interface size or sum of the elements of **Measures**.
- **AggregatedValue:** Number of inputs and outputs for a component and its subcomponents.
- **Measures:** Array consisting of number of inputs, number of outputs, number of implicit inputs, and number of implicit outputs, which are local to the component.
- **AggregatedMeasures:** Maximum number of inputs, outputs, implicit inputs, and implicit outputs for a component and subcomponents.

Explicit Input Output Metric

Metric Information

Metric ID: `mathworks.metrics.ExplicitIOCount`

Description

Metric Type: Size

Use this metric to calculate the number of inputs and outputs in the model, which include:

- Inputs: Inport blocks, Trigger ports, Enable ports, chart input data and events.
- Outputs: Outport blocks, chart output data and events.

This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.ExplicitIOCount`.

The `slmetric.metric.AggregationMode` property setting is Max.

Computation Details

The metric:

- Excludes From and Goto blocks.
- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- **Value:** Total interface size or sum of the elements of **Measures**.
- **AggregatedValue:** Number of inputs and outputs for a component and its subcomponents.
- **Measures:** Array consisting of number of inputs and number of outputs which are local to the component.
- **AggregatedMeasures:** Maximum number of inputs and outputs for a component and subcomponents.

File Metric

Metric Information

Metric ID: `mathworks.metrics.FileCount`

Description

Metric Type: Size

Use this metric to count the number of model and library files used by a specific component and its subcomponents. This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.FileCount`.

The `slmetric.metric.AggregationMode` property setting is `None`.

Computation Details

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Number of model and library files.
- `AggregatedValue`: Not applicable.
- `Measures`: Not applicable.

MATLAB Function Metric

Metric Information

Metric ID: `mathworks.metrics.MatlabFunctionCount`

Description

Metric Type: Size

Use this metric to count the number of MATLAB Function blocks inside a component. This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.MatlabFunctionCount`.

The `slmetric.metric.AggregationMode` property setting is `Sum`.

Computation Details

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Number of MATLAB Function blocks.
- `AggregatedValue`: Number of MATLAB Function blocks for component and its subcomponents.
- `Measures`: Not applicable.

Model File Count

Metric Information

Metric ID: `mathworks.metrics.ModelFileCount`

Description

Metric Type: Size

Use this metric to count the number of model files. This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.ModelFileCount`.

The `slmetric.metric.AggregationMode` property setting is `None`.

Computation Details

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Number of files reference by a component and its subcomponents.
- `AggregatedValue`: Not applicable.
- `Measures`: Not applicable.

Parameter Metric

Metric Information

Metric ID: `mathworks.metrics.ParameterCount`

Description

Metric Type: Size

Use this metric to calculate the number of instances of parameter data inside a Simulink system.

A parameter is a variable used by a Simulink block or object of a basic type (which includes `single`, `double`, `uint8`, `uint16`, `uint32`, `int8`, `int16`, `int32`, `boolean`, `logical`, `struct`, `char`, `cell`), `Simulink.Parameter` object, `Simulink.Variant` object, or enum value. This metric returns every instance of a parameter in a model, which means that the metric counts each instance of a parameter separately. The parameter data must be located in the base workspace, the model workspace, or a data dictionary.

For example, the model `f14` uses two instances of the parameter `Zw`. One instance is in block `f14/Gain` at the root level of the model. One instance is in block `f14/Aircraft Dynamics Model/Transfer Fcn.2` in the `Aircraft Dynamics Model` subsystem. The metric `mathworks.metrics.ParameterCount` includes both of these instances of parameter `Zw` when it calculates the number of parameter instances in the `f14` model and its subsystems.

This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.ParameterCount`.

The `slmetric.metric.AggregationMode` property setting is `Sum`.

Computation Details

This metric:

- Filters results from the `Simulink.findVars` function and inherits the limitations of this function.
- Counts the parameter instances in a component rather than unique parameters.
- Does not include parameters in masked workspaces.
- Does not include data type and signal objects.
- If specified, analyzes the content of library-linked blocks or referenced models.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Number of parameter instances used inside a component.
- `AggregatedValue`: Number of parameter instances for a component and its subcomponents.
- `Measures`: Not applicable.

Stateflow Chart Metric

Metric Information

Metric ID: `mathworks.metrics.StateflowChartCount`

Description

Metric Type: Size

Use this metric to count the number of Stateflow charts at any component level. This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.StateflowChartCount`.

The `slmetric.metric.AggregationMode` property setting is `Sum`.

Computation Details

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Number of Stateflow charts at the model level.
- `AggregatedValue`: Number of charts for component and its subcomponents.
- `Measures`: Not applicable.

Cyclomatic Complexity Metric

Metric Information

Metric ID: `mathworks.metrics.CyclomaticComplexity`

Description

Metric Type: Architecture

Model Advisor Check ID: `mathworks.metricchecks.CyclomaticComplexity`

Use this metric to calculate the cyclomatic complexity of the model. Cyclomatic complexity is a measure of the structural complexity of a model. The complexity measure can be different for the generated code than for the model due to code features that this analysis does not consider, such as consolidated logic and error checks. To compute the cyclomatic complexity of an object (such as a block, chart, or state), Simulink Check uses this formula:

$$c = \sum_{n=1}^N (o_n - 1)$$

N is the number of decision points that the object represents and o_n is the number of outcomes for the n th decision point. The calculation considers a vectorized operation or a Multiport switch block as a single decision point. The tool adds 1 to the complexity number for models, atomic subsystems, and Stateflow charts.

The results provide the local and aggregated cyclomatic complexity for the:

- Model
- Subsystems
- Charts
- MATLAB functions

Local complexity is the cyclomatic complexity for objects at their hierarchical level. Aggregated cyclomatic complexity is the cyclomatic complexity of an object and its descendants

This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.CyclomaticComplexity`.

The `slmetric.metric.AggregationMode` property setting is `Sum`.

Computation Details

The metric:

- Does not run on library models.
- Analyzes content in masked subsystems.
- Does not analyze inactive variants.

- If specified, analyzes the content of library-linked blocks or referenced models.
- Does not analyze referenced models in accelerated mode.

Collection

To collect data for this metric using the Model Advisor, run the check, **Cyclomatic complexity metric** in **By Task > Model Metrics > Complexity Metrics**. The Model Advisor check displays the local cyclomatic complexity for the root model and for Simulink and Stateflow objects in the system. The check does not analyze referenced models or return aggregated results.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Local cyclomatic complexity.
- `AggregatedValue`: Aggregated cyclomatic complexity.
- `Measures`: Not applicable.

For more information on complexity, see:

- “Compare Model Complexity and Code Complexity Metrics” on page 5-76
- “Cyclomatic Complexity for Stateflow Charts” (Simulink Coverage)
- “Specify Coverage Options” (Simulink Coverage)

Clone Content Metric

Metric Information

Metric ID: `mathworks.metrics.CloneContent`

Description

Metric Type: Architecture

Use this metric to calculate the fraction of the total number of subcomponents that are clones. Clones must have identical block types and connections but they can have different parameter values. For more information on clone detection, see “Enable Component Reuse by Using Clone Detection” on page 3-29.

This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.CloneContent`.

The `slmetric.metric.AggregationMode` property setting is `None`.

Computation Details

- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- **Value:** Fraction of total number of subcomponents that are clones
- **AggregatedValue:** Not applicable.
- **Measures:** Vector containing number of clones, total number of components, and clone group number.

Clone Detection Metric

Metric Information

Metric ID: `mathworks.metrics.CloneDetection`

Description

Metric Type: Architecture

Use this metric to count the number of clones in a model. Clones must have identical block types and connections but they can have different parameter values. This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.CloneDetection`.

The `slmetric.metric.AggregationMode` property setting is `Sum`.

Computation Details

- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Number of clones.
- `AggregatedValue`: Number of clones for component and its subcomponents.
- `Measures`: Not applicable.

Library Content Metric

Metric Information

Metric ID: `mathworks.metrics.LibraryContent`

Description

Metric Type: Architecture

Use this metric to calculate the fraction of total number of components that are linked library blocks. This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.LibraryContent`.

The `slmetric.metric.AggregationMode` property setting is `None`.

Computation Details

- If specified, analyzes the content of library-linked blocks or referenced models.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Fraction of the total number of subcomponents that are linked library blocks.
- `AggregatedValue`: Not applicable.
- `Measures`: Vector containing the number of linked library blocks and total number of components

MATLAB Code Analyzer Warnings

Metric Information

Metric ID: `mathworks.metrics.MatlabCodeAnalyzerWarnings`

Description

Metric Type: Compliance

Use this metric to calculate the number of MATLAB code analyzer warnings from MATLAB code in the model. This metric is available with Simulink Check.

The `slmetric.metric.AggregationMode` property setting is Sum.

Computation Details

The metric:

- Analyzes MATLAB code in MATLAB Function blocks
- Analyzes MATLAB functions in Stateflow charts
- Runs on library models
- Analyzes content in masked subsystems
- If specified, analyzes content of library-linked blocks and referenced models
- Does not analyze external MATLAB code files

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Number of MATLAB code analyzer warnings
- `AggregatedValue`: Number of MATLAB code analyzer warnings aggregated for a component and subcomponents.
- `Measures`: Not applicable.

For more information on code analyzer warnings, see “Check Code for Errors and Warnings Using the Code Analyzer”.

Diagnostic Warnings Metric

Metric Information

Metric ID: `mathworks.metrics.DiagnosticWarningsCount`

Description

Metric Type: Compliance

Use this metric to calculate the number of Simulink diagnostic warnings reported during a model update for simulation. This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.DiagnosticWarningsCount`.

The `slmetric.metric.AggregationMode` property setting is `Sum`.

Computation Details

- If specified, analyzes the content of library-linked blocks or referenced models.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Number of diagnostic warnings reported.
- `AggregatedValue`: Number of diagnostic warnings reported for component and its subcomponents.
- `Measure`: Not applicable.

Model Advisor Check Compliance for High-Integrity Systems

Metric Information

Metric ID: `mathworks.metrics.ModelAdvisorCheckCompliance.hisl_do178`

Description

Metric Type: Compliance

Family ID: `mathworks.metrics.ModelAdvisorCheckCompliance`

Use this metric to calculate the fraction of Model Advisor checks that pass for the **High-Integrity Systems** subgroups. This metric is available with Simulink Check.

The `slmetric.metric.AggregationMode` property setting is Percentile.

Computation Details

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.
- Analyzes content in Stateflow objects.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- **Value:** Fraction of total number of checks passed in **High-Integrity Systems** subgroups.
- **AggregatedValue:** Fraction of total number of checks passed in **High-Integrity Systems** subgroups aggregated for a component and all of its subcomponents.
- **Measures:** Vector containing: number of checks passed in subgroups and number of checks in subgroups.
- **AggregatedMeasures:** Vector containing: number of checks passed in subgroups and number of checks in subgroup, for a component and all its subcomponents.

Results Details

For this metric, instances of the `slmetric.metric.ResultDetail` Value property provides these results:

- A value of 0 indicates that a check did not run.
- A value of 1 indicates that a check passed.
- A value of 2 indicates a check warning.

- A value of 3 indicates a failure.

For more information on the checks, see “Model Advisor Checks for DO-178C/DO-331 Standards Compliance” on page 3-45.

Model Advisor Check Compliance for Modeling Standards for MAB

Metric Information

Metric ID: `mathworks.metrics.ModelAdvisorCheckCompliance.maab`

Description

Metric Type: Compliance

Family ID: `mathworks.metrics.ModelAdvisorCheckCompliance`

Use this metric to calculate the fraction of Model Advisor checks that pass for the group **Modeling Standards for MAB**. This metric is available with Simulink Check.

The `slmetric.metric.AggregationMode` property setting is Percentile.

Computation Details

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.
- Analyzes content in Stateflow objects.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- **Value:** Fraction of total number of checks passed in MAB.
- **AggregatedValue:** Fraction of total number of checks passed in MAB aggregated for a component and all of its subcomponents.
- **Measures:** Vector containing: number of checks passed in group and number of checks in group.
- **AggregatedMeasures:** Vector containing: number of checks passed in group and number of checks in group, for a component and all its subcomponents.

Results Details

For this metric, instances of the `slmetric.metric.ResultDetail` Value property provides these results:

- A value of 0 indicates that a check did not run.
- A value of 1 indicates that a check passed.
- A value of 2 indicates a check warning.

- A value of 3 indicates a failure.

For more information on the checks, see “Using Model Advisor Checks for JMAAB Modeling Guidelines”.

Model Advisor Check Issues for High-Integrity Systems

Metric Information

Metric ID: `mathworks.metrics.ModelAdvisorCheckIssues.hisl_do178`

Description

Metric Type: Compliance

Family ID: `mathworks.metrics.ModelAdvisorCheckIssues`

Use this metric to calculate number of issues reported by the subgroups of Model Advisor checks for **High-Integrity Systems**. This metric counts each Model Advisor check that produces a warning or failure. If a check contains links to blocks, this metric counts one issue for each linked block. Checks with links to the model are highlighted in the Simulink Editor. If a check does not contain links to blocks, this metric counts one issue. This metric is available with Simulink Check.

The `slmetric.metric.AggregationMode` property setting is Sum.

Computation Details

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.
- Analyzes content in Stateflow objects.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- **Value:** Number of issues reported by the **High-Integrity Systems** checks
- **AggregatedValue:** Number of issues reported by the **High-Integrity Systems** checks aggregated for a component and all of its subcomponents.
- **Measures:** Not applicable.

For more information on the checks, see “Model Advisor Checks for DO-178C/DO-331 Standards Compliance” on page 3-45.

Model Advisor Check Issues for MAB Standards

Metric Information

Metric ID: `mathworks.metrics.ModelAdvisorCheckIssues.maab`

Description

Metric Type: Compliance

Family ID: `mathworks.metrics.ModelAdvisorCheckIssues`

Use this metric to calculate number of issues reported by the group of Model Advisor checks for **Modeling Standards for MAB**. This metric counts each Model Advisor check that produces a warning or failure. If a check contains links to blocks, this metric counts one issue for each linked block. Checks with links to the model are highlighted in the Simulink Editor. If a check does not contain links to blocks, this metric counts one issue. This metric is available with Simulink Check.

The `slmetric.metric.AggregationMode` property setting is Sum.

Computation Details

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.
- Analyzes content in Stateflow objects.
- Adds check issues on the configuration set or issues with data objects to the issue count at the model root level.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- **Value:** Number of issues reported by the Model Advisor for MAB checks.
- **AggregatedValue:** Number of issues reported by the Model Advisor for MAB checks aggregated for a component and its subcomponents.
- **Measures:** Not applicable.

For more information on the checks, see “Using Model Advisor Checks for JMAAB Modeling Guidelines”.

Nondescriptive Block Name Metric

Metric Information

Metric ID: `mathworks.metrics.DescriptiveBlockNames`

Description

Metric Type: Readability

Model Advisor Check ID: `mathworks.metricchecks.DescriptiveBlockNames`

Run this metric to determine nondescriptive Inport, Outport, and Subsystem block names. Default names appended with an integer are nondescriptive block names. The results provide the nondescriptive block names at the model and subsystem levels.

This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.DescriptiveBlockNames`.

The `slmetric.metric.AggregationMode` property setting is `Sum`.

Computation Details

The metric:

- Does not run on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

Collection

To collect results for this metric using the Model Advisor, run the check, **Nondescriptive block name metric** in **By Task > Model Metrics > Readability Metrics**. The Model Advisor check displays the number of nondescriptive Inport, Outport, and Subsystem block names in the model or subsystem. The check does not display the result for each type of block separately. The check does not analyze referenced models or return aggregated results.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- **Value:** Number of nondescriptive Inport, Outport, and Subsystem block names.
- **AggregatedValue:** Number of nondescriptive Inport, Outport, and Subsystem block names for a component and its subcomponents.
- **Measures:** 1-D vector containing:
 - Total number of Inport blocks
 - Number of Inport blocks with nondescriptive names

- Total number of Outport blocks
- Number of Outport blocks with nondescriptive names
- Total number of Subsystem blocks
- Number of Subsystem blocks with nondescriptive names
- **AggregatedMeasures:** 1-D vector containing sum of:
 - Total number of Inport blocks
 - Number of Inport blocks with nondescriptive names
 - Total number of Outport blocks
 - Number of Outport blocks with nondescriptive names
 - Total number of Subsystem blocks
 - Number of Subsystem blocks with nondescriptive names

Data and Structure Layer Separation Metric

Metric Information

Metric ID: `mathworks.metrics.LayerSeparation`

Description

Metric Type: Readability

Model Advisor Check ID: `mathworks.metricchecks.LayerSeparation`

Run this metric to calculate the data and structure layer separation. The results provide the separation at the model and subsystem level.

This metric is available with Simulink Check. To collect data for this metric, use `getMetrics` with the metric identifier, `mathworks.metrics.LayerSeparation`.

For guidelines about blocks on model levels, see the MAB guideline `db_0143: Usable block types in model hierarchy`.

The `slmetric.metric.AggregationMode` property setting is `Sum`.

Computation Details

The metric:

- Does not run on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

Collection

To collect results for this metric using the Model Advisor, run the check, **Data and structure layer separation metric** in **By Task > Model Metrics > Readability Metrics**. The Model Advisor check displays the separation for the model or subsystem. The check does not analyze referenced models or return aggregated results.

Results

For this metric, instances of `slmetric.metric.Result` provide the following results:

- `Value`: Number of basic blocks on a structural level.
- `AggregatedValue`: Number of basic blocks on a structural level for a component and its subcomponents.
- `Measures`: Not applicable.

Create Model Advisor Checks

Overview of the Customization File for Custom Checks

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

Function	Description	When Required
<code>sl_customization()</code>	Registers custom checks, tasks, folders, and callbacks with the Simulink customization manager at start-up. See “Define Custom Model Advisor Checks” on page 6-45.	Required for programmatic customizations to the Model Advisor.
One or more check definitions	Defines custom checks. See “Define Custom Model Advisor Checks” on page 6-45.	Required for custom checks and to add custom checks to the By Product folder. If the By Product folder is not displayed in the Model Advisor window, select Show By Product Folder from the Settings > Preferences dialog box.
Check callback functions	Defines the actions of the custom checks. See “Define Custom Model Advisor Checks” on page 6-45.	Required for custom checks. You must write one callback function for each custom check
One or more calls to check input parameters	Specifies input parameters to custom checks. See “Define Check Input Parameters” on page 6-48.	Optional
One or more calls to checklist views	Specifies calls to the Model Advisor Result Explorer for custom checks.	Optional
One or more calls to check actions	Specifies actions the software performs for custom checks. See “Define Custom Model Advisor Checks” on page 6-45.	Optional

This example shows a custom configuration of the Model Advisor that has custom checks defined in custom folders and procedures. The selected check includes input parameters, list view parameters, and actions.

The screenshot displays the Model Advisor interface. On the left is a tree view under 'Model Advisor' with the following structure:

- Model Advisor
 - By Product
 - Demo
 - Check Simulink
 - Check Simulink
 - Check model o
 - By Task
 - Demo Factory Gro
 - Check Simulink
 - Check Simulink
 - Check model o
 - My Group
 - Example task with
 - Example task 2
 - Example task 3
 - My Procedure

The right pane is titled 'Example task with input parameter and auto-fix ability'. It contains the following sections:

- Analysis**
 - Example style three callback
 - Input Parameters
 - Skip font checks.
 - Standard font size: 12 Valid font: Arial
 - Run This Check** button
 - Result: Not Run
 - Click **Run This Check**.
- Action**
 - Click the button to update all blocks with specified font
 - Fix block fonts** button

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.ID.getSID	Identifying Simulink blocks, model annotations or Stateflow objects.
Stateflow API (Stateflow)	Programmatic access to Stateflow objects.

Review a Model Against Conditions that You Specify with the Model Advisor

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

Create an `sl_customization` Function

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

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

% Copyright 2019 The MathWorks, Inc.

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

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

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

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

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

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

```
function definePassFailCheck
mdlAdvRoot = ModelAdvisor.Root;
rec = ModelAdvisor.Check('simplePassFailCheck');
rec.Title = 'Check Constant block usage';
rec.TitleTips = ['Warn if Constant block value is a number; Pass if' ...
    ' Constant block value is a letter'];
```

```

rec.setCallbackFcn(@simplePassFailCheck, 'None', 'DetailStyle')

mdladvRoot.publish(rec, 'Demo');

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

```

This check identifies Constant block values that are numbers and produces a warning, but it does not provide a fix action. For more information on how to create a check definition function with a fix, see “Fix a Model to Comply with Conditions that You Specify with the Model Advisor” on page 6-21.

Create the Check Definition Function for an Informational Check

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

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

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

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

```

function defineInformationCheck

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

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

end

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

```



```

function modelVersionChecksumCallbackUsingFT_Detail(system,CheckObj)

model = bdroot(system);
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
ElementResults = ModelAdvisor.ResultDetail;
ElementResults.Description = 'Display model configuration and checksum information';

% If running the Model Advisor on a subsystem, add note to description.
if strcmp(system, model) == false
    ElementResults.IsInformer = true;
    ElementResults.Status = 'NOTE: The Model Advisor is reviewing a subsystem, but these results are based on root-level';
    ElementResults(end + 1) = ModelAdvisor.ResultDetail;
end

% If error is encountered, use these values.
mdlver = 'Error - could not retrieve Version';
mdlauthor = 'Error - could not retrieve Author';
mdldate = 'Error - could not retrieve Date';
mdlsum = 'Error - could not retrieve CheckSum';

% Get model configuration and checksum information.
try
    mdlver = get_param(model, 'ModelVersion');
    mdlauthor = get_param(model, 'LastModifiedBy');
    mdldate = get_param(model, 'LastModifiedDate');
    mdlsum = Simulink.BlockDiagram.getChecksum(model);
    mdlsum = [num2str(mdlsum(1)) ' ' num2str(mdlsum(2)) ' ' ...
              num2str(mdlsum(3)) ' ' num2str(mdlsum(4))];
    ElementResults(end).IsInformer = true;
    mdladvObj.setCheckResultStatus(true);
catch
    ElementResults(end).IsViolation = true;
    mdladvObj.setCheckResultStatus(false);
end

lbStr = '<br>';
resultStr = ['Model Version: ' mdlver lbStr 'Author: ' mdlauthor lbStr ...
             'Date: ' mdldate lbStr 'Model Checksum: ' mdlsum];
ElementResults(end).Status = resultStr;
CheckObj.setResultDetails(ElementResults);

end

```

Run the Custom Checks in the Model Advisor

- 1 In the Command Window, enter:

```
Advisor.Manager.refresh_customizations
```
- 2 Open the model `sldemo_fuelsys` by typing this command in the MATLAB command prompt:

```
sldemo_fuelsys
```
- 3 In the **Modeling** tab, select **Model Advisor**. A System Selector dialog opens. Click **OK**.
- 4 In the left pane, select **By Product > Demo > Identify model configuration and checksum information**.
- 5 Click **Run Checks**.

The check passes and displays the information.

- 6 In the left pane, select **By Product > Demo > Check Constant block usage**.
- 7 Click **Run Checks**.

The check produces a warning because several blocks contain values that are numbers. The results contain links to these blocks. The result displays a **Recommended Action**.

- 8** Follow the **Recommended Action** to fix the Constant blocks.

See Also

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

More About

- “Define Custom Model Advisor Checks” on page 6-45
- “Fix a Model to Comply with Conditions that You Specify with the Model Advisor” on page 6-21
- “Create and Deploy a Model Advisor Custom Configuration” on page 7-24

Define Edit-Time Checks to Comply with Conditions That You Specify with the Model Advisor

In this example, you create three custom edit-time checks that check for compliance with certain software design standards. Custom edit-time checks help you catch issues earlier in the model design review process, but these checks also report issues in the Model Advisor.

The first check checks that Inport and Outport blocks have certain colors depending on their output data types.

The second check checks whether a Trigger block is higher than other blocks in a subsystem. This check must check edited blocks and other blocks in the same subsystem as the Trigger block.

The third check checks whether signals that connect to Outport blocks have labels.

Register and Define the Custom Edit-Time Checks

1. To register a custom edit-time check, create an `sl_customization` function. For this example, copy the `sl_customization_cec.m` file to the `sl_customization.m` file by entering this command at the command prompt:

```
copyfile sl_customization_cec.m sl_customization.m f
```

The `sl_customization` function accepts one argument, a customization manager object. To register the custom check, use the `addModelAdvisorCheckFcn` method. The input to this method is a handle to the check definition function. For this example, `defineCheck` is the check definition function. Open and inspect the `sl_customization.m` file.

```
function sl_customization(cm)
cm.addModelAdvisorCheckFcn(@defineCheck);
```

2. Create the check definition function. For this example, open and inspect the `defineCheck` function. This function contains three `ModelAdvisor.Check` objects with their check IDs as input arguments. The `CallbackHandle` property is the name of the class that you create to define the edit-time check. For this example, `MyEditTimeChecks` is the package name and `PortColor`, `TriggerBlockPosition`, and `SignalLabel` are the class names. The `mdladvRoot.publish` function publishes the checks to a new folder in the Model Advisor. For this example, the folder name is **DEMO: Edit Time Checks**.

```
function defineCheck

% Check the background color of Inport and Outport blocks.
rec = ModelAdvisor.Check("advisor.edittimecheck.PortColor");
rec.Title = 'Check color of Inport and Outport blocks';
rec.CallbackHandle = 'MyEditTimeChecks.PortColor';
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.publish(rec, 'DEMO: Edit Time Checks');

% Check that determines whether Trigger block is the top-most block in a subsystem.
rec= ModelAdvisor.Check("advisor.edittimecheck.TriggerBlock");
rec.Title = 'Check that Trigger block position is higher than other blocks';
rec.CallbackHandle = 'MyEditTimeChecks.TriggerBlockPosition';
mdladvRoot.publish(rec, 'DEMO: Edit Time Checks');
```

```

%% Check that determines whether signals with SignalPropagation 'on' have labels.
rec = ModelAdvisor.Check("advisor.edittimecheck.SignalLabel");
rec.Title = 'Check that signals have labels if they are to propagate those labels';
rec.CallbackHandle = 'MyEditTimeChecks.Signallabels';
mdladvRoot.publish(rec, 'DEMO: Edit Time Checks');

```

3. Create an edit-time check by creating a class that derives from the `ModelAdvisor.EdittimeCheck` abstract base class. Inspect the first edit-time check by opening the `PortColor.m` file.

```

classdef PortColor < ModelAdvisor.EdittimeCheck
    % Check that ports conform to software design standards for background color.
    %
    % Background Color          Data Types
    % orange                   Boolean
    % green                     all floating-point
    % cyan                      all integers
    % Light Blue               Enumerations and Bus Objects
    % white                     auto
    %

    methods
        function obj=PortColor(checkId)
            obj=obj@ModelAdvisor.EdittimeCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.BLKITER;
        end

        function violation = blockDiscovered(obj, blk)
            violation = [];
            if strcmp(get_param(blk, 'BlockType'), 'Inport') || strcmp(get_param(blk, 'BlockType'),

                dataType = get_param(blk, 'OutDataTypeStr');
                currentBgColor = get_param(blk, 'BackgroundColor');

                if strcmp(dataType, 'boolean')
                    if ~strcmp(currentBgColor, 'orange')
                        % Create a violation object using the ModelAdvisor.ResultDetail class
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation, 'SID', Simulink.ID.getSID);
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Output blocks with Boolean outputs';
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
                elseif any(strcmp({'single', 'double'}, dataType))
                    if ~strcmp(currentBgColor, 'green')
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation, 'SID', Simulink.ID.getSID);
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Output blocks with floating-point outputs';
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
                elseif any(strcmp({'uint8', 'uint16', 'uint32', 'int8', 'int16', 'int32'}, dataType))
                    if ~strcmp(currentBgColor, 'cyan')
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation, 'SID', Simulink.ID.getSID);
                    end
                end
            end
        end
    end
end

```

```

        violation.CheckID = obj.checkId;
        violation.Description = 'Inport/Outport blocks with integer outputs should have integer outputs';
        violation.title = 'Port Block Color';
        violation.ViolationType = 'Warning';
    end
elseif contains(dataType, 'Bus:')
    if ~strcmp(currentBgColor, 'lightBlue')
        violation = ModelAdvisor.ResultDetail;
        ModelAdvisor.ResultDetail.setData(violation, 'SID', Simulink.ID.getSID(blk));
        violation.CheckID = obj.checkId;
        violation.Description = 'Inport/Outport blocks with bus outputs should have integer outputs';
        violation.title = 'Port Block Color';
        violation.ViolationType = 'Warning';
    end
elseif contains(dataType, 'Enum:')
    if ~strcmp(currentBgColor, 'lightBlue')
        violation = ModelAdvisor.ResultDetail;
        ModelAdvisor.ResultDetail.setData(violation, 'SID', Simulink.ID.getSID(blk));
        violation.CheckID = obj.checkId;
        violation.Description = 'Inport/Outport blocks with enumeration outputs should have integer outputs';
        violation.title = 'Port Block Color';
        violation.ViolationType = 'Warning';
    end
elseif contains(dataType, 'auto')
    if ~strcmp(currentBgColor, 'white')
        violation = ModelAdvisor.ResultDetail;
        ModelAdvisor.ResultDetail.setData(violation, 'SID', Simulink.ID.getSID(blk));
        violation.CheckID = obj.checkId;
        violation.Description = 'Inport/Outport blocks with auto outputs should have integer outputs';
        violation.title = 'Port Block Color';
        violation.ViolationType = 'Warning';
    end
end
end
end
end

function violation = finishedTraversal(obj)
    violation = [];
end

function success = fix(obj, violation)
    success = true;
    dataType = get_param(violation.Data, 'OutDataTypeStr');
    if strcmp(dataType, 'boolean')
        set_param(violation.Data, 'BackgroundColor', 'orange');
    elseif any(strcmp({'single', 'double'}, dataType))
        set_param(violation.Data, 'BackgroundColor', 'green');
    elseif any(strcmp({'uint8', 'uint16', 'uint32', 'int8', 'int16', 'int32'}, dataType))
        set_param(violation.Data, 'BackgroundColor', 'cyan');
    elseif contains(dataType, 'Bus:') || contains(dataType, 'Enum:')
        set_param(violation.Data, 'BackgroundColor', 'lightBlue');
    elseif contains(dataType, 'auto')
        set_param(violation.Data, 'BackgroundColor', 'white');
    end
end
end
end
end
end

```

The PortColor class defines three methods: PortColor, blockDiscovered, and fix. The PortColor method sets the CheckId and TraversalType properties. This check has a traversal type of edittimecheck.TraversalTypes.BLKITER because the check must check newly added and edited blocks, but it does not have to check for affected blocks in the same subsystem or model as the edited or newly added blocks. The blockDiscovered method contains an algorithm that checks the color of Inport and Outport blocks. Then, because the violation is on a block, the algorithm highlights a violating block by creating a ModelAdvisor.ResultDetail violation object with the Type property set to the default value of SID. The fix method updates blocks that do not have correct colors.

5. Inspect the second edit-time check by opening the TriggerBlockPosition.m file.

```
classdef TriggerBlockPosition < ModelAdvisor.EdittimeCheck
    properties
        TriggerBlock = [];
        position = [];
    end

    methods
        function obj=TriggerBlockPosition(checkId)
            obj=obj@ModelAdvisor.EdittimeCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.ACTIVEGRAPH;
        end

        function violation = blockDiscovered(obj, blk)
            violation = [];
            if strcmp(get_param(blk,'BlockType'),'TriggerPort')
                obj.TriggerBlock = blk;
            else
                h = get_param(blk,'Position');
                obj.position = [obj.position, h(2)];
            end
        end

        function violation = finishedTraversal(obj)
            violation = [];
            if isempty(obj.TriggerBlock)
                return;
            end
            triggerPosition = get_param(obj.TriggerBlock,'Position');
            if min(obj.position) < triggerPosition(2)
                violation = ModelAdvisor.ResultDetail;
                ModelAdvisor.ResultDetail.setData(violation,'SID',...
                    Simulink.ID.getSID(obj.TriggerBlock));
                violation.CheckID = obj.checkId;
                violation.title = 'Trigger Block Position';
                violation.Description = 'Trigger Block should be top block in subsystem';
                violation.ViolationType = 'Warning';
            end
            obj.TriggerBlock = [];
            obj.position =[];
        end
    end
end
```

The TriggerBlockPosition class defines three methods: TriggerBlockPosition, blockDiscovered, and finishedTraversal. The TriggerBlockPosition method sets the

CheckId and TraversalType properties. This check has a traversal type of `edittimecheck.TraversalTypes.ACTIVEGRAPH` because it must check other blocks in the same subsystem as the Trigger block. The `blockDiscovered` method checks the position of Trigger blocks within subsystems. The `finishedTraversal` method checks whether the position of these Trigger blocks are higher than other blocks in a subsystem. Then, because the violation is on a block, the algorithm highlights a violating block by creating a `ModelAdvisor.ResultDetail` violation object with the Type property set to the default value of SID.

6. Inspect the third edit-time check by opening the `SignalLabels.m` file.

```
classdef SignalLabels < ModelAdvisor.EdittimeCheck
    methods
        function obj=SignalLabels(checkId)
            obj=obj@ModelAdvisor.EdittimeCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.BLKITER;
        end

        function violation = blockDiscovered(obj, blk)
            violation = [];
            ports = get_param(blk,'Ports');
            lh = get_param(blk, 'LineHandles');
            if strcmp(get_param(blk,'BlockType'),'Outport')
                for j = 1 : ports(1)
                    if lh.Inport(j) ~= -1 % failure case: no connection
                        allsources = get_param(lh.Inport(j),'SrcPortHandle');
                        hiliteHandle = get_param(lh.Inport(j), 'DstPortHandle');
                        if (isempty(allsources) ~= 0) || (isempty(find(allsources==-1,1)) ~= 0)
                            lh_obj = get_param(lh.Inport(j),'Object');
                            if isempty(lh_obj.Name)
                                if strcmp(lh_obj.signalPropagation,'off') == 1
                                    allsources_parent = get_param(allsources,'Parent');
                                    if strcmp(get_param(allsources_parent,'BlockType'),'Inport')
                                        buscreator_outputs = get_param(allsources_parent,'IsBusE
                                    else
                                        buscreator_outputs = 'off';
                                    end
                                    if ~strcmp(buscreator_outputs,'on')
                                        violation = ModelAdvisor.ResultDetail;
                                        ModelAdvisor.ResultDetail.setData(violation,'Signal',hil
                                        violation.Description = 'This signal should have a label.
                                        violation.CheckID = obj.checkId;
                                        violation.Title = 'Signal Label Missing';
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end
```

The `SignalLabels` class defines two methods: `SignalLabels` and `blockDiscovered`. The `SignalLabels` method sets the `CheckId` and `TraversalType` properties. This check has a traversal type of `edittimecheck.TraversalTypes.BLKITER` because the check must check newly added and edited blocks, but it does not have to check for affected blocks in the same subsystem or

model as the edited or newly added blocks. Because this check is for signals, the `blockDiscovered` method must use the parameters on the line handles, `LineHandles`, of blocks to find signals with violations. Specifically, for signals that connect to Output blocks, this algorithm checks whether the `Name` signal parameter has a value. Then, because the violation is on a signal, the algorithm highlights the signal by creating a violation object with the `Type` property value set to `Signal`.

7. Create a folder named `+MyEditTimeChecks` and save the three class definition files in that folder. Classes must be in a folder that has the same name as their package name. Enter these commands at the command prompt:

```
copyfile PortColor.m* +MyEditTimeChecks f
copyfile TriggerBlockPosition.m +MyEditTimeChecks
copyfile SignalLabels.m +MyEditTimeChecks
```

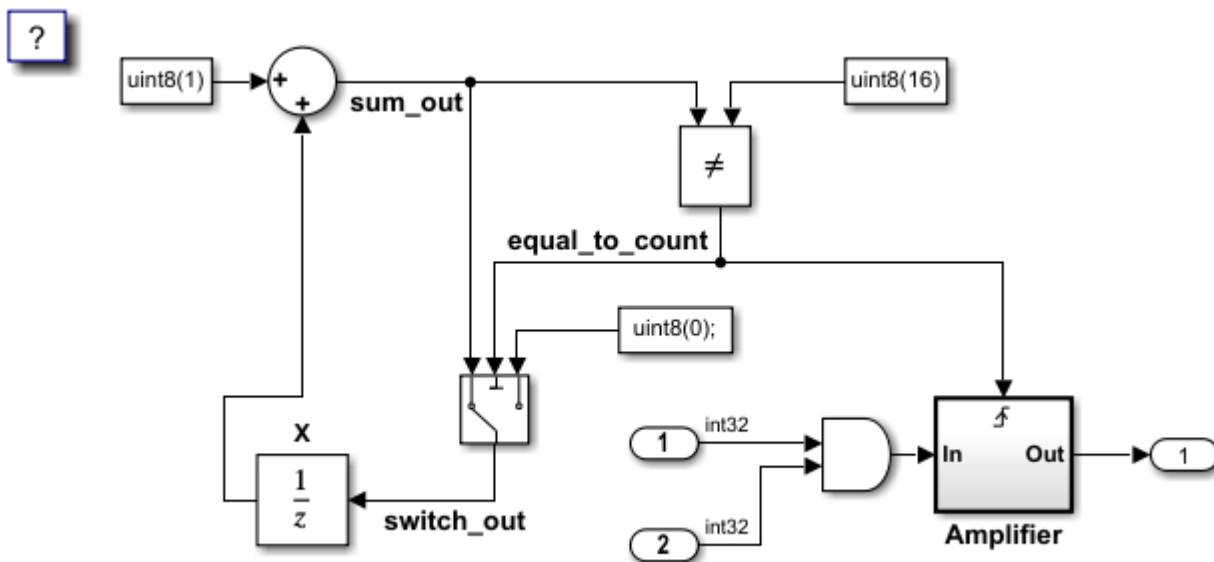
Run the Edit-Time Checks on a Model

1. In order for your custom checks to be visible in the Model Advisor, you must refresh the Model Advisor check information cache. Enter this command at the command prompt:

```
Advisor.Manager.refresh_customizations
```

2. Open the model by entering this command at the command prompt.

```
open_system('AdvisorCustomizationExample.slx');
```



3. To create a custom configuration of checks, open the Model Advisor Configuration Editor. On the Modeling tab, click **Model Advisor > Configuration Editor**.

4. You can create a custom configuration consisting of just the three edit-time checks in this example by deleting every folder except for the **DEMO: Edit Time Checks** folder. The `my_config.json` file that ships with this example contains this configuration. Close the Model Advisor Configuration Editor.

5 Set the custom configuration to the `my_config.json` file by clicking the **Modeling** tab and selecting **Model Advisor > Edit-Time Checks**. In the Configuration Parameters dialog box, specify the path to the configuration file in the **Model Advisor configuration file** parameter. Alternatively, enter this command at the command prompt:

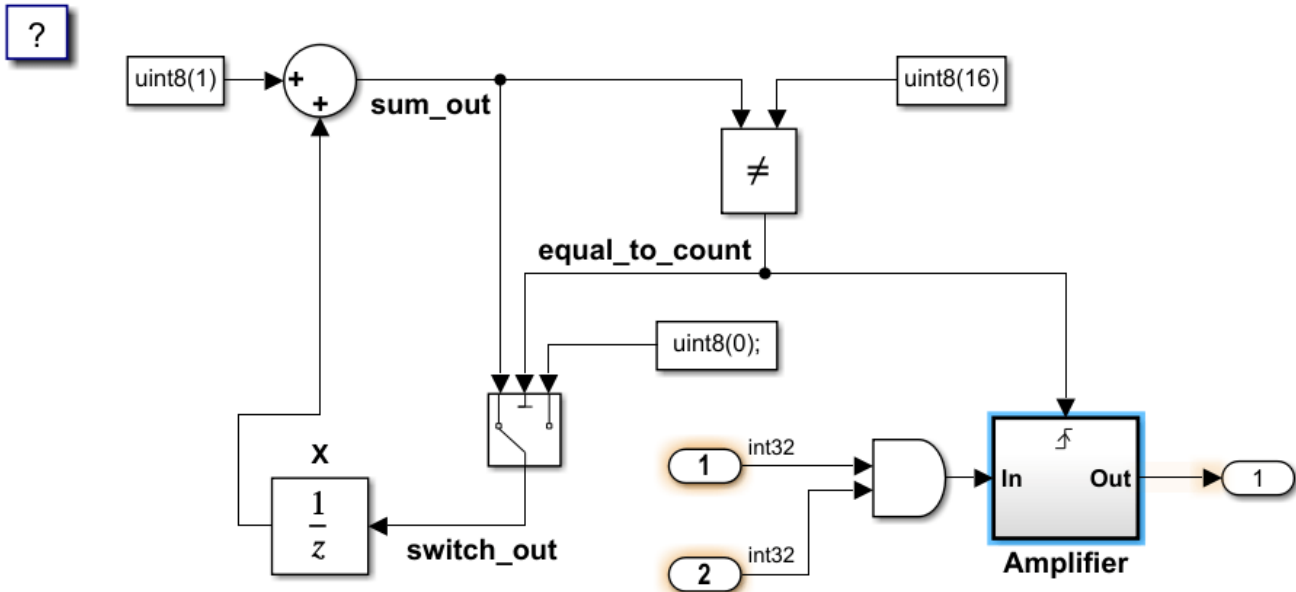
```
ModelAdvisor.setModelConfiguration('AdvisorCustomizationExample', 'my_config.json');
```

6. Turn on edit-time checking by clicking the **Modeling** tab and selecting **Model Advisor > Edit-Time Checks**. In the Configuration Parameters dialog box, select the **Edit-Time Checks** parameter. Alternatively, you can enter this command at the command prompt:

```
edittime.setAdvisorChecking('AdvisorCustomizationExample', 'on');
```

7. Open the Model Advisor by clicking the **Modeling** tab and selecting **Model Advisor**. Observe that the three edit-time checks are the only ones in the Model Advisor. Close the Model Advisor.

8. To view the edit-time warnings, click the blocks and signals highlighted in yellow.



At the top level of the model, the two Inport blocks have an output data type of `int32`. They produce edit-time warnings because they should be cyan. The Outport block does not produce a violation because it has an `auto` data type and is white. In the **Amplifier** subsystem, the Inport and Outport blocks do not produce edit-time warnings because they have a data type of `auto` and are white. The Trigger block does not produce an edit-time warning because it is the top-most block in the model. If you move the Trigger block below another block, the Trigger block has an edit-time warning. The signal connecting to the Outport block produces a warning because it does not have a label.

8. To fix the edit-time warnings for the two Inport blocks, in the edit-time check warning window, click **Fix**.

9. Close the model and the Model Advisor.

```
bdclose;
```

10. Remove the files from your working directory. Refresh the Model Advisor check information cache by entering this command:

```
Advisor.Manager.refresh_customizations
```

Performance Considerations for Custom Edit-Time Checks

To help prevent custom edit-time checks from negatively impacting performance as you edit your model, the Model Advisor automatically disables custom edit-time checks if, in the current MATLAB® session, the check takes longer than 500 milliseconds to execute in at least three different Simulink® models.

If the Model Advisor disables a custom edit-time check, you will see a warning on the Simulink canvas. You can re-enable the edit-time check by either:

- Clicking the hyperlink text in the warning.
- Passing the check identifier, `checkID`, to the function `edittime.enableCheck`:

```
edittime.enableCheck(checkID)
```

To prevent a custom edit-time check from being disabled, author the check so that the check executes in less than 500 milliseconds on your models.

See Also

[ModelAdvisor.EdittimeCheck](#) | [ModelAdvisor.Check](#) | [ModelAdvisor.ResultDetail](#)

Related Examples

- “Define Custom Model Advisor Checks” on page 6-45
- “Define Custom Edit-Time Checks that Fix Issues in Architecture Models” on page 6-17

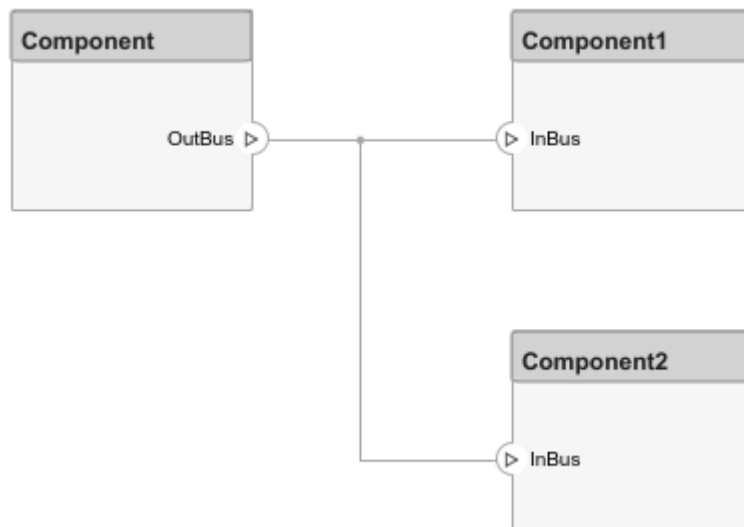
Define Custom Edit-Time Checks that Fix Issues in Architecture Models

This example shows how to create a custom edit-time check that runs on architecture models that you create using System Composer. Edit-time checks help you catch issues earlier in the model design review process. The custom edit-time check in this example produces a warning if the names of connecting block port interface names do not match. For more information on the process for authoring custom checks, see “Define Custom Model Advisor Checks” on page 6-45.

Create a Simple Architecture Model

Create a simple architecture model with mismatched data interface names in ports that share a connector.

- 1 Create a temporary working directory.
- 2 In MATLAB, on the **Home** tab, click **Simulink**.
- 3 In the Simulink Start page, click **System Composer** and select **Architecture Model**.
- 4 Add three Component blocks.
- 5 Connect the output of one Component block to the inport ports of the other two Component blocks as shown in this image.



- 6 On the **Modeling** tab, click **Interface Editor**.
- 7 Create two data interfaces with the names `interface0` and `interface1`.
- 8 Open the Property Inspector.
- 9 For **Component**, click the **OutBus** port. In the **Interface** section of the Property Inspector, for the **Name** field, select `interface0`.
- 10 For **Component1**, click the **InBus** port. In the **Interface** section of the Property Inspector, for the **Name** field, select `interface1`.
- 11 For **Component2**, click the **InBus** port. In the **Interface** section of the Property Inspector, for the **Name** field, select `interface0`.

- 12 Save the model to your working directory. For this example, the model name is `myModel.slx`.

Create the Custom Edit-Time Check

Create a check that detects the mismatched data interface names in ports that share the same connector while a user is editing a model.

- 1 To register the custom edit-time check, create an `sl_customization` function. The `sl_customization` function accepts one argument, a customization manager object. To register the custom check, use the `addModelAdvisorCheckFcn` method. The input to this method is a handle to the check definition function. For this example, `defineCheck` is the check definition function. Create the `sl_customization` function and save it to your working folder.

```
function sl_customization(cm)
cm.addModelAdvisorCheckFcn(@defineCheck);
```

- 2 Create the check definition function. Inside the function, create a `ModelAdvisor.Check` object and specify the Check ID as an input argument. Then, specify the `ModelAdvisor.Check` Title and `CallbackHandle` properties. The `CallbackHandle` property is the name of the class that you create to define the edit-time check. For this example, `MyEditTimeChecks` is the package name and `PortMismatch` is the class name. Then, publish the check to a new folder in the Model Advisor. For this example, the folder name is **System Composer Edit-time Check**. For this example, create a `defineCheck` function and include the code below in it. Save the `defineCheck` function to your working folder.

```
function defineCheck
rec = ModelAdvisor.Check("advisor.edittimecheck.SystemComposerPortMismatch");
rec.Title = 'Check port mismatch for system composer components';
rec.CallbackHandle = 'MyEditTimeChecks.PortMismatch';
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.publish(rec, 'System Composer Edit-time Check');
end
```

- 3 Create a class that derives from the `ModelAdvisor.EditableCheck` abstract base class. For this example, create a class file named `PortMismatch`. Copy the code below into the `PortMismatch.m` file. Then, create a folder named `+MyEditTimeChecks` and save the `PortMismatch.m` file in that folder. The class must be in a folder that has the same name as the package name.

The `PortMismatch` class defines two methods: `PortMismatch` and `blockDiscovered`. The `PortMismatch` method sets the `CheckId` and `TraversalType` properties. This check has a traversal type of `edittimecheck.TraversalTypes.ACTIVEGRAPH` because the check must check newly added and edited blocks and affected blocks in the same subsystem or model. The `blockDiscovered` method contains an algorithm that checks whether the port interface names match.

```
classdef PortMismatch < ModelAdvisor.EditableCheck

    methods
        function obj=PortMismatch(checkId)
            obj=obj@ModelAdvisor.EditableCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.ACTIVEGRAPH;
        end

        function violationArray = blockDiscovered(obj,blk)
            violationArray = [];
            blkHdl = get_param(blk, 'Handle');
            archMdl = systemcomposer.arch.Model(bdroot(blk));
            comp = archMdl.lookup('SimulinkHandle',blkHdl);
            if isa(comp, 'systemcomposer.arch.Component')
                for i = 1:length(comp.Ports)

```

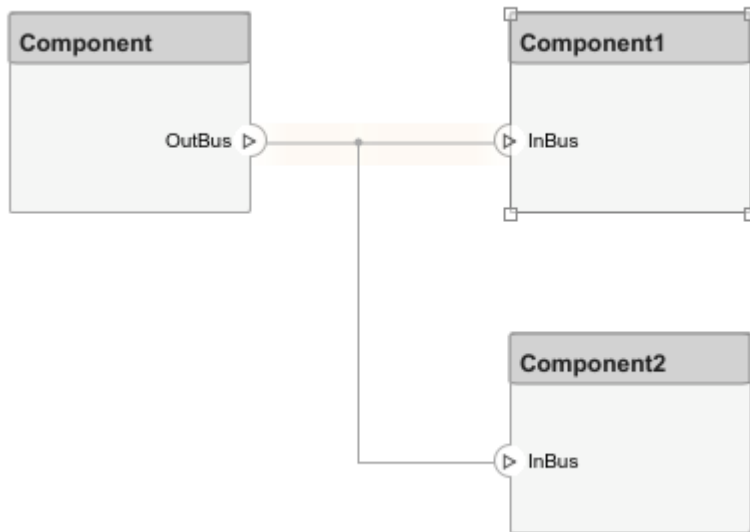


```
ModelAdvisor.setModelConfiguration('myModel', 'sc_config.json');
```

- 5 Turn on edit-time checking by selecting the **Model Advisor > Edit Time** configuration parameter. Alternatively, you can enter this command at the command prompt:

```
edittime.setAdvisorChecking('myModel', 'on');
```

- 6 To view the edit-time warnings, click the signal highlighted in yellow.



The connector between the Component and Component1 blocks produces a warning because the data interface names in each port do not match.

See Also

[ModelAdvisor.EdittimeCheck](#) | [ModelAdvisor.Check](#) | [ModelAdvisor.ResultDetail](#)

Related Examples

- “Define Custom Model Advisor Checks” on page 6-45
- “Run Custom Model Advisor Checks on Architecture Models” on page 3-99
- “Define Edit-Time Checks to Comply with Conditions That You Specify with the Model Advisor” on page 6-9

Fix a Model to Comply with Conditions that You Specify with the Model Advisor

This example shows how to create a customized Model Advisor pass/fail check with a fix action. When a model does not contain a check violation, the results contain the check description and result status. When a model contains a check violation, the results contain the check description, result status, and the recommended action to fix the issue. This example adds a custom check to a Model Advisor **By Product > Demo** subfolder.

For this example, the custom check identifies blocks whose names do not appear below the blocks. The fix action is to make the block names appear below the blocks.

When a check does not pass, the results include a hyperlink to each model element that violates the check. Use these hyperlinks to easily locate areas in your model or subsystem. The code for this example consists of an `sl_customization.m` file and a `defineDetailStyleCheck.m` file.

Create the `sl_customization` File

- 1 In your working folder, create an `sl_customization.m` file.
- 2 To register the custom checks, create an `sl_customization(cm)` function as shown here. This function accepts one argument, a customization manager object. The customization manager object includes the `addModelAdvisorCheckFcn` method for registering the custom check. The input to this method is a handle to the function `defineModelAdvisorChecks`. `defineModelAdvisorChecks` contains a call to the check definition function for custom Model Advisor pass/fail check.

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

% Copyright 2019 The MathWorks, Inc.

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

% -----
% defines Model Advisor Checks
% -----
function defineModelAdvisorChecks
    defineDetailStyleCheck;
```

Create the Check Definition File

The check definition function defines the check and fix actions that the Model Advisor takes when you run the check. For this example, the completed check definition function file is `defineDetailStyleCheck.m`, and it contains this code:

```
function defineDetailStyleCheck
mdladvRoot = ModelAdvisor.Root;

% Create ModelAdvisor.Check object and set properties.
rec = ModelAdvisor.Check('com.mathworks.sample.detailStyle');
rec.Title = 'Check whether block names appear below blocks';
rec.TitleTips = 'Check position of block names';
rec.setCallbackFcn(@DetailStyleCallback,'None','DetailStyle');
```

```

% Create ModelAdvisor.Action object for setting fix operation.
myAction = ModelAdvisor.Action;
myAction.setCallbackFcn(@ActionCB);
myAction.Name='Make block names appear below blocks';
myAction.Description='Click the button to place block names below blocks';
rec.setAction(myAction);

mdladvRoot.publish(rec, 'Demo'); % publish check into Demo group.

end

% -----
% This callback function uses the DetailStyle CallbackStyle type.
% -----
function DetailStyleCallback(system, CheckObj)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system); % get object

% Find all blocks whose name does not appear below blocks
violationBlks = find_system(system, 'Type','block',...
    'NamePlacement','alternate',...
    'ShowName', 'on');
if isempty(violationBlks)
    ElementResults = ModelAdvisor.ResultDetail;
    ElementResults.IsInformer = true;
    ElementResults.Description = 'Identify blocks where the name is not displayed below the block.';
    ElementResults.Status = 'All blocks have names displayed below the block.';
    mdladvObj.setCheckResultStatus(true);
else
    for i=1:numel(violationBlks)
        ElementResults(1,i) = ModelAdvisor.ResultDetail;
    end
    for i=1:numel(ElementResults)
        ModelAdvisor.ResultDetail.setData(ElementResults(i), 'SID',violationBlks{i});
        ElementResults(i).Description = 'Identify blocks where the name is not displayed below the block.';
        ElementResults(i).Status = 'The following blocks have names that do not display below the blocks.';
        ElementResults(i).RecAction = 'Change the location such that the block name is below the block.';
    end
    mdladvObj.setCheckResultStatus(false);
    mdladvObj.setActionEnable(true);
end
CheckObj.setResultDetails(ElementResults);
end

% -----
% This action callback function changes the location of block names.
% -----
function result = ActionCB(taskobj)
mdladvObj = taskobj.MAObj;
checkObj = taskobj.Check;
resultDetailObjs = checkObj.ResultDetails;
for i=1:numel(resultDetailObjs)
    % take some action for each of them
    block=Simulink.ID.getHandle(resultDetailObjs(i).Data);
    set_param(block,'NamePlacement','normal');
end

result = ModelAdvisor.Text('Changed the location such that the block name is below the block.');
```

The following steps explain how to create the `defineDetailStyleCheck.m` file.

- 1 Create a `ModelAdvisor.Root` object.

```
mdladvRoot = ModelAdvisor.Root;
```

- 2 Create a `ModelAdvisor.Check` object and define the unique check ID. For this check, the ID is `com.mathworks.sample.detailStyle`.

```
rec = ModelAdvisor.Check('com.mathworks.sample.detailStyle');
```


- Specify the `ModelAdvisor.Check.Title` and `ModelAdvisor.Check.TitleTips` properties.

```
rec.Title = 'Check whether block names appear below blocks';
rec.TitleTips = 'Check position of block names';
```

- Use the `setCallbackFcn` method to call the callback function. The `setCallbackFcn` method arguments are a handle to the callback function and the `ModelAdvisor.Check.CallbackStyle` property value. For this example, the `CallbackStyle` property value is `DetailStyle`. This style allows you to view results by block, subsystem, or recommended action. Applying this style produces default formatting, so that you do not have to use the `ModelAdvisor.FormatTemplate` class or the other Model Advisor formatting APIs to format the results that appear in the Model Advisor.

```
rec.setCallbackFcn(@DetailStyleCallback, 'None', 'DetailStyle');
```

- To set the fix operation, create a `ModelAdvisor.Action` object and define its properties. Use the `setCallback` method to call the action callback function. The input to this method is a handle to the action callback function.

```
myAction = ModelAdvisor.Action;
myAction.setCallbackFcn(@ActionCB);
myAction.Name='Make block names appear below blocks';
myAction.Description='Click the button to place block names below blocks';
```

- Use the `setAction` method to set the action for the check.

```
rec.setAction(myAction);
```

- Use the `publish` method to publish the check to a folder within the **By Product** folder. For this example, the folder name is **Demo**.

```
mdladvRoot.publish(rec, 'Demo'); % publish check into Demo group.
```

Create the Check Callback Definition Function

- In the `defineDetailStyleCheck.m` file, create the check callback function. In this example, the function name is `DetailStyleCallback`. The inputs to this function are a `ModelAdvisor.CheckObject` and the path to the model or system that the Model Advisor analyzes.

```
function DetailStyleCallback(system, CheckObj)
```

- To create a `Simulink.ModelAdvisor` object, use the function `Simulink.ModelAdvisor.getModelAdvisor`.

```
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system); % get object
```

- To identify blocks that violate the check, use the `find_system` function. For each model element, this function creates a `ModelAdvisor.ResultDetail` object.

```
violationBlks = find_system(system, 'Type', 'block', ...
                             'NamePlacement', 'alternate', ...
                             'ShowName', 'on');
```

- Write code for the case when the `find_system` function does not identify blocks whose names do not appear below the block. In this case, `ElementResults` is one instance of a `ModelAdvisor.ResultDetail` object and provides information content only. The method specifies that there is no check violation and displays **Passed** in the Model Advisor.

```
if isempty(violationBlks)
    ElementResults = ModelAdvisor.ResultDetail;
    ElementResults.IsInformer = true;
    ElementResults.Description = 'Identify blocks where the name is not displayed below the block.';
    ElementResults.Status = 'All blocks have names displayed below the block.';
    mdladvObj.setCheckResultStatus(true);
end
```

- Write code for the case when the `find_system` function returns a list of blocks whose names do not appear below the block (`violationBlks`). `ElementResults` includes each `ModelAdvisor.ResultDetail` object that violates the check and provides a recommended action message for fixing the check violation.

For this case, the `setCheckResultStatus` method specifies the check violation and displays **Warning** or **Failed** in the Model Advisor. The `Simulink.ModelAdvisor.setActionEnable(true)` method enables the ability to fix the check violation issue from the Model Advisor.

```
else
    for i=1:numel(violationBlks)
        ElementResults(1,i) = ModelAdvisor.ResultDetail;
    end
    for i=1:numel(ElementResults)
        ModelAdvisor.ResultDetail.setData(ElementResults(i), 'SID',violationBlks{i});
        ElementResults(i).Description = 'Identify blocks where the name is not displayed below the block.';
        ElementResults(i).Status = 'The following blocks have names that do not display below the blocks:.';
        ElementResults(i).RecAction = 'Change the location such that the block name is below the block.';
    end
    mdladvObj.setCheckResultStatus(false);
    mdladvObj.setActionEnable(true);
end
```

- To associate the results with a check object, use the `setResultDetails` method.

```
CheckObj.setResultDetails(ElementResults);
end
```

Create the Action Callback Definition Function

- In the `defineDetailStyleCheck.m` file, create the action callback function. In this example, the function name is `sampleActionCB`. The input to this function is a `ModelAdvisor.Task` object.

```
function result = ActionCB(taskobj)
```

- Create handles to `Simulink.ModelAdvisor` and `ModelAdvisor.Check` objects.

```
mdladvObj = taskobj.MAObj;
checkObj = taskobj.Check;
```

- Create an array of `ModelAdvisor.ResultDetail` objects for storing the information for blocks that violate the check.

```
resultDetailObjs = checkObj.ResultDetails;
```

- Write code that changes the block name location to below the block.

```
for i=1:numel(resultDetailObjs)
    % take some action for each of them
    block=Simulink.ID.getHandle(resultDetailObjs(i).Data);
    set_param(block, 'NamePlacement', 'normal');
end
result = ModelAdvisor.Text('Changed the location such that the block name is below the block.');
```

- Disable the **Action** box.

```
mdladvObj.setActionEnable(false);
```

Run the Check

- Save the `sl_customization.m` and `defineDetailStyleCheck.m` files.

- 2 In the MATLAB command window, enter:

```
Advisor.Manager.refresh_customizations
```
- 3 Open the model `sldemo_fuelsys` by typing this command in the MATLAB command prompt:

```
sldemo_fuelsys
```
- 4 In the top model, select the block named `Engine Speed`. In the toolstrip, on the **Format** tab, click **Flip Name**.
- 5 Open the `fuel_rate_control` subsystem. Select the block named `validate_sample_time`. In the toolstrip, on the **Format** tab, click **Flip Name**.

Return to the top model and save as `example_sldemo_fuelsys`.

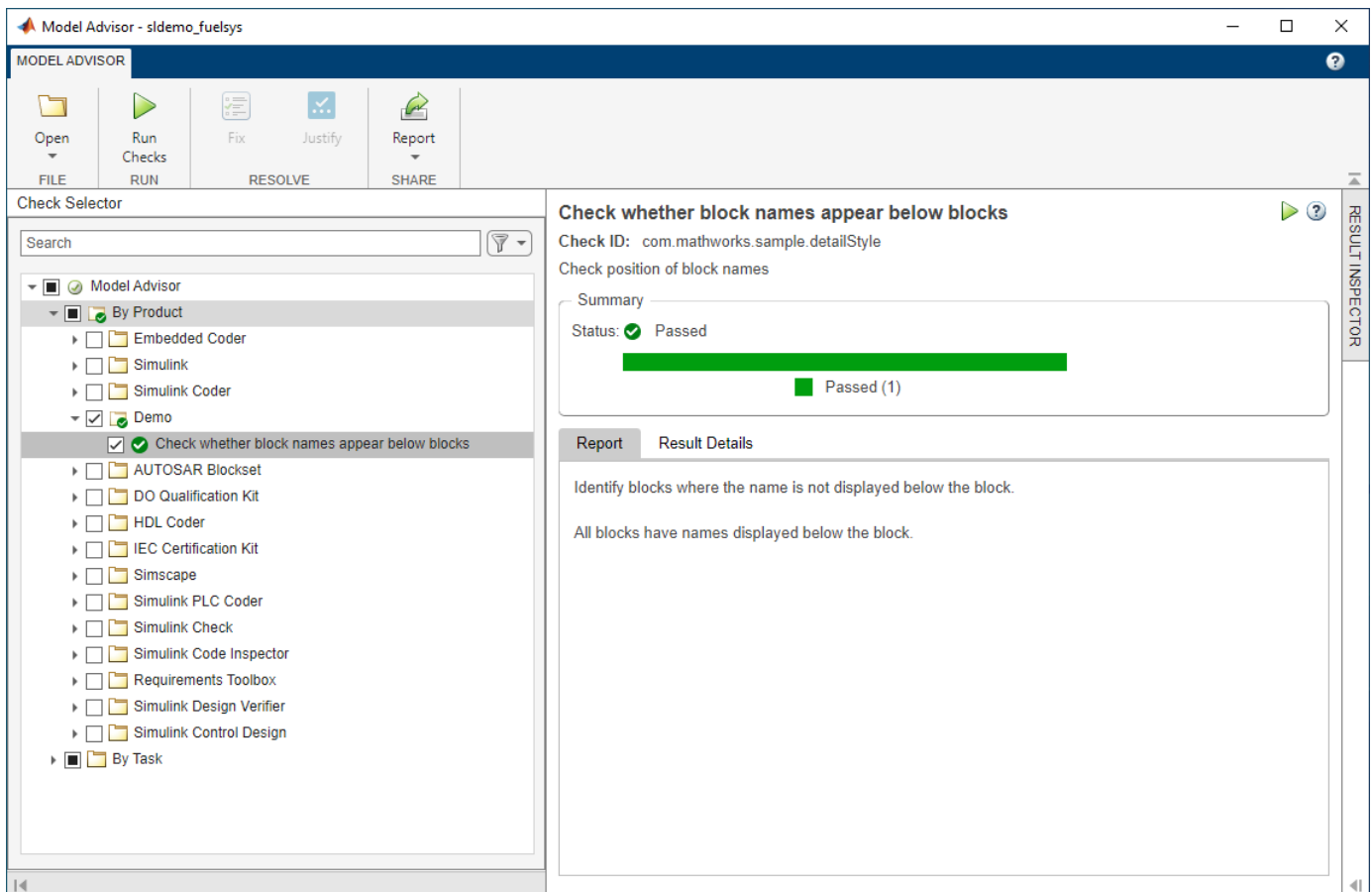
- 6 In the **Modeling** tab, select **Model Advisor**. A **System Selector — Model Advisor** dialog box opens. Click **OK**. The Model Advisor opens.
- 7 In the left pane, select **By Product > Demo > Check whether block names appear below blocks**.
- 8 Select **Run Checks**. The Model Advisor check produces a warning for the blocks that you changed.
- 9 Review the results by selecting either the **Report** or **Result Detail** tabs.

Both tabs provide a recommended action for each block that violates the check. You can click the hyperlink path to open the block in the model editor. For example:

The screenshot displays the Model Advisor window for the `sldemo_fuelsys` model. The interface is divided into several sections:

- Toolbar:** Contains icons for 'Open', 'Run Checks', 'Fix', 'Justify', and 'Report'.
- Check Selector:** A tree view on the left showing the navigation path: **Model Advisor** > **By Product** > **Demo** > **Check whether block names appear below blocks**.
- Check Summary:**
 - Check ID:** `com.mathworks.sample.detailStyle`
 - Check position of block names**
 - Status:** Warning (indicated by a red triangle icon)
 - Warning (2):** A progress bar shows the warning level.
- Report / Result Details:**
 - Warning:** The following blocks have names that do not display below the blocks:
 - [sldemo_fuelsys/Engine Speed](#)
 - [sldemo_fuelsys/fuel_rate_control/validate_sample_time](#)
 - Recommended Action:** Change the location such that the block name is below the block.
- Justifications:** A text area for entering rationale to justify the check, with an **Add Justification** button.

- 10 Follow the recommended action for fixing the violating blocks by using one of these methods:
 - Update each violation individually by double-clicking the hyperlink to open the block. Select the block. In the toolbar, on the **Format** tab, select **Flip Name**.
 - In the toolbar, click **Fix**. The Model Advisor automatically fixes the issues in the model. Notice that the button is dimmed after the violations are fixed.
- 11 Rerun the Model Advisor check. The check passes.



See Also

[ModelAdvisor.Check](#) | [ModelAdvisor.FormatTemplate](#) | [ModelAdvisor.FormatTemplate](#) | [ModelAdvisor.Check.CallbackContext](#) | [Simulink.ModelAdvisor](#) | [Simulink.ModelAdvisor.getModelAdvisor](#) | [Simulink.ModelAdvisor.openConfigUI](#) | [Simulink.ModelAdvisor.reportExists](#)

More About

- “Define Custom Model Advisor Checks” on page 6-45
- “Review a Model Against Conditions that You Specify with the Model Advisor” on page 6-5
- “Create and Deploy a Model Advisor Custom Configuration” on page 7-24

Create Model Advisor Check for Model Configuration Parameters

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

Decide which configuration parameter settings to use for your model. If desired, review the modelling guidelines:

- MathWorks Advisory Board (MAB) Modeling Guidelines
 - High-Integrity System Modeling Guidelines
 - Code Generation Modeling Guidelines
- 1 Create an XML data file containing the configuration parameter settings you want to check. You can use `Advisor.authoring.generateConfigurationParameterDataFile` or manually create the file yourself.
 - 2 Register the model configuration parameter check using an `sl_customization.m` file.
 - 3 Run the check on your models.

Create a Data File for a Configuration Parameter Check

This example shows how to create a data file that specifies configuration parameter values in the **Diagnostics** pane. A custom check warns when the configuration parameters values do not match the values defined in the data file.

At the command prompt, type `vdp` to open the van der Pol Equation model.

Right-click in the model window and select **Model Configuration Parameters**. In the **Diagnostics** pane, set the configuration parameters as follows:

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

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

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

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

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

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

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

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

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

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

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

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

```

    disp(msg);
end

```

Create Check for Diagnostics Pane Model Configuration Parameters

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

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

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

The check uses the `ex_DataFile.xml` data file created in “Create a Data File for a Configuration Parameter Check” on page 6-27.

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

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

```

function sl_customization(cm)

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

%% defineModelAdvisorChecks
function defineModelAdvisorChecks

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

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

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

mdladvRoot = ModelAdvisor.Root;
mdladvRoot.publish(rec, 'Demo');

```

Note that model configuration parameter settings checks must use the `setCallbackFcn` type of `StyleOne`.

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

```
Advisor.Manager.refresh_customizations
```

At the command prompt, type `vdp` to open the van der Pol Equation model.

Right-click in the model window and select **Model Configuration Parameters**. In the **Diagnostics** pane, set the configuration parameters as follows:

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

In the **Modeling** tab, select **Model Advisor** to open the Model Advisor.

In the left pane, select **DemoExample: Check model configuration parameters**. In the right pane, **Data File** is set to `ex_DataFile.xml`.

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

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

In the toolbar, click **Fix**. The Model Advisor updates the configuration parameters for **Block priority violation** and **Minimize algebraic loop**.

Run the **Demo > Example: Check model configuration parameters** check again. The check warns because **Algebraic loop** is set to none.

In the right pane of the Model Advisor window, use the `Algebraic loop` (`AlgebraicLoopMsg`) link to edit the configuration parameter. Set **Algebraic loop** to warning or error.

Run the check a final time. The check passes.

Data File for Configuration Parameter Check

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

```
<?xml version="1.0" encoding="utf-8"?>
<customcheck xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```



```

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

```

The `<messages>` tag contains:

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

Note The `<messages>` tag is optional.

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

In the `<checkdata>` tag, the data file specifies two types of constraints:

- **PositiveModelParameterConstraint** - Specifies the configuration parameter setting that you want.
- **NegativeModelParameterConstraint** - Specifies the configuration parameter setting that you do not want.

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

- **parameter** - Specifies the configuration parameter that you want to check. The tagging uses the command line name for the configuration parameter. For example:

```

<PositiveModelParameterConstraint>
  <parameter>BlockPriorityViolationMsg</parameter>
</PositiveModelParameterConstraint>
<NegativeModelParameterConstraint>
  <parameter>AlgebraicLoopMsg</parameter>
</NegativeModelParameterConstraint>

```

- `value` - Specifies the setting(s) for the configuration parameter. You can specify more than one `value` tag.

When using `PositiveModelParameterConstraint`, `value` specifies the setting(s) that you want for the configuration parameter. For `NegativeModelParameterConstraint`, `value` specifies the setting(s) you that do not want for the configuration parameter.

You can specify the `value` using a format in this table.

Type	Format	Example
Scalar value	<code><value>xyz</value></code>	In this example, constraint <code>NegativeModelParameterConstraint</code> warns when the configuration parameter settings for configuration parameter is not error or none. <pre><NegativeModelParameterConstraint> <value>error</value> <value>none</value> </NegativeModelParameterConstraint></pre>
Structure or object	<code><value> <param1>xyz</param1> <param2>yza</param2> </value></code>	In this example, constraints <code>PositiveModelParameterConstraint</code> warns when the configuration parameter settings are not a valid structure: <pre><PositiveModelParameterConstraint> <value> <double>a</double> <single>b</single> </value> </PositiveModelParameterConstraint></pre>
Array	<code><value> <element>value</element> <element>value</element> </value></code>	In this example, constraint <code>NegativeModelParameterConstraint</code> warns when the configuration parameter settings are an invalid array: <pre><NegativeModelParameterConstraint> <value> <element>A</element> <element>B</element> </value> </NegativeModelParameterConstraint></pre>

Type	Format	Example
Structure Array	<pre><value> <element> <param1>xyz</param1> <param2>yza</param2> </element> <element> <param1>xyz</param1> <param2>yza</param2> </element> </value></pre>	<p>In this example, constraint <code>NegativeModelParameterConstraint</code> warns when the configuration parameter settings are an invalid structure array:</p> <pre><NegativeModelParameterConstraint> <value> <element> <double>a</double> <single>b</single> </element> <element> <double>a</double> <single>b</single> </element> </value> </NegativeModelParameterConstraint></pre>

- `fixvalue` - (Optional) Specifies the setting to use when applying the Model Advisor fix action.

You can specify the `fixvalue` using a format in this table.

Type	Format	Example
Scalar value	<pre><fixvalue>xyz</fixvalue></pre>	<p>In this example, the fix action tag specifies the new configuration parameter setting as warning.</p> <pre><PositiveModelParameterConstraint> <value>error</value> <fixaction>warning</fixaction> </PositiveModelParameterConstraint></pre>
Structure or object	<pre><fixvalue> <param1>xyz</param1> <param2>yza</param2> </fixvalue></pre>	<p>In this example, the fix action tag specifies the new configuration parameter setting for a structure.</p> <pre><PositiveModelParameterConstraint> <value> <double>a</double> <single>b</single> </value> <fixvalue> <double>c</double> <single>d</single> </fixvalue> </PositiveModelParameterConstraint></pre>

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

- `dependson` - (Optional) Specifies a prerequisite subcheck.

In this example, `dependson` specifies that configuration parameter subcheck `ID_B` must pass before configuration parameter subcheck `ID_A` runs.

```
<PositiveModelParameterConstraint id="ID_A">
  <dependson>ID_B</value>
</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 Model Advisor Checks for Supported and Unsupported Blocks and Parameters

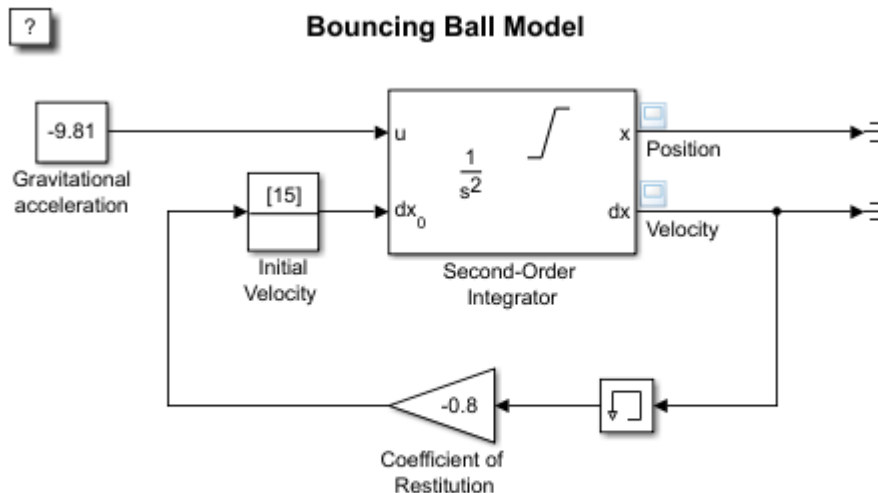
You can create Model Advisor checks that check whether blocks use specific block or parameter values. You can specify constraints for:

- Supported or unsupported block parameter values
- Supported or unsupported model parameter values
- Supported or unsupported blocks
- Whether blocks or parameters meet a combination of constraints

You can also use the `addPreRequisiteConstraintID` function to add prerequisite constraints that must pass before Model Advisor checks the actual constraint. You can check your model against these constraints as you edit or by running the checks from the Model Advisor.

Example

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



Copyright 2004-2013 The MathWorks, Inc.

Create a Check for Supported or Unsupported Block Parameters

First, create a Model Advisor check that contains three block parameter constraints, `c1`, `c2`, and `c3`, that specify the supported and unsupported block parameter values.

1. Define a new function.

```
function constraints = createConstraints_Check1()
end
```

2. Inside the function, create two block parameter constraints, `c1` and `c2`.


```
function constraints = createConstraints_Check1()

    c1=Advisor.authoring.PositiveBlockParameterConstraint;
    c1.ID='ID_1';
    c1.BlockType='Gain';
    c1.ParameterName='Gain';
    c1.SupportedParameterValues={'-0.7'};
    c1.ValueOperator='eq'; % equal to

    c2=Advisor.authoring.NegativeBlockParameterConstraint;
    c2.ID='ID_2';
    c2.BlockType='InitialCondition';
    c2.ParameterName='Value';
    c2.UnsupportedParameterValues={'0'};
    c2.ValueOperator='le'; % less than or equal to

end
```

Constraint c1 specifies that a Gain block must have a value equal to -0.7. Constraint c2 specifies that an Initial Condition block with a value less than or equal to zero is unsupported.

3. Create a positive block constraint, c3, and set constraints equal to a cell array of constraints c1, c2, and c3.

```
function constraints = createConstraints_Check1()

    c1=Advisor.authoring.PositiveBlockParameterConstraint;
    c1.ID='ID_1';
    c1.BlockType='Gain';
    c1.ParameterName='Gain';
    c1.SupportedParameterValues={'-0.7'};
    c1.ValueOperator='eq'; % equal to

    c2=Advisor.authoring.NegativeBlockParameterConstraint;
    c2.ID='ID_2';
    c2.BlockType='InitialCondition';
    c2.ParameterName='Value';
    c2.UnsupportedParameterValues={'0'};
    c2.ValueOperator='le'; % less than or equal to

    c3=Advisor.authoring.PositiveBlockTypeConstraint;
    c3.ID='ID_3';
    s1=struct('BlockType','Constant','MaskType','');
    s2=struct('BlockType','SubSystem','MaskType','');
    s3=struct('BlockType','InitialCondition','MaskType','');
    s4=struct('BlockType','Gain','MaskType','');
    s5=struct('BlockType','Memory','MaskType','');
    s6=struct('BlockType','SecondOrderIntegrator','MaskType','');
    s7=struct('BlockType','Terminator','MaskType','');
    c3.SupportedBlockTypes={s1;s2;s3;s4;s5;s6;s7};

    constraints = {c1,c2,c3};

end
```

Constraint c3 specifies the supported blocks. constraints is a cell array of the block constraints.

4. Define a new Model Advisor check by creating another function, `check1`. Use the function `Advisor.authoring.createBlockConstraintCheck` to create a Model Advisor check, `rec`, with these block constraints. Then use `mdladvRoot.register(rec)` to register the block constraints check with the Model Advisor.

```
function check1()

    rec = Advisor.authoring.createBlockConstraintCheck('mathworks.check_0001',...
                                                    'Constraints',@createConstraints_Check1);

    rec.Title = 'Example 1: Check three block parameter constraints';
    rec.TitleTips = 'Example check three block parameter constraints';

    mdladvRoot = ModelAdvisor.Root;
    mdladvRoot.publish(rec,'Example: My Group')
```

end

```
function constraints = createConstraints_Check1()

    c1=Advisor.authoring.PositiveBlockParameterConstraint;
    c1.ID='ID_1';
    c1.BlockType='Gain';
    c1.ParameterName='Gain';
    c1.SupportedParameterValues={'-0.7'};
    c1.ValueOperator='eq'; % equal to

    c2=Advisor.authoring.NegativeBlockParameterConstraint;
    c2.ID='ID_2';
    c2.BlockType='InitialCondition';
    c2.ParameterName='Value';
    c2.UnsupportedParameterValues={'0'};
    c2.ValueOperator='le'; % less than or equal to

    c3=Advisor.authoring.PositiveBlockTypeConstraint;
    c3.ID='ID_3';
    s1=struct('BlockType','Constant','MaskType','');
    s2=struct('BlockType','SubSystem','MaskType','');
    s3=struct('BlockType','InitialCondition','MaskType','');
    s4=struct('BlockType','Gain','MaskType','');
    s5=struct('BlockType','Memory','MaskType','');
    s6=struct('BlockType','SecondOrderIntegrator','MaskType','');
    s7=struct('BlockType','Terminator','MaskType','');
    c3.SupportedBlockTypes={s1;s2;s3;s4;s5;s6;s7};

    constraints = {c1,c2,c3};
```

end

Create a Check for a Composite Constraint

Next, create a Model Advisor check that contains three block parameter constraints `cc1`, `cc2`, and `cc`. Constraints `cc1` and `cc2` specify which block parameters are supported and constraint `cc` is a composite constraint which contains `cc1` and `cc2`.

1. Define a new function.

```
function constraints = createConstraints_Check2()
end
```

2. Create two block parameter constraints, cc1 and cc2, and a composite constraint, cc. Set constraints equal to a cell array of constraints cc1, cc2, and cc.

```
function constraints = createConstraints_Check2()

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

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

    cc=Advisor.authoring.CompositeConstraint;
    cc.addConstraintID('ID_cc1');
    cc.addConstraintID('ID_cc2');
    cc.CompositeOperator='and'; % Model Advisor checks multiple constraints

    constraints = {cc1,cc2,cc};

end
```

Constraint cc1 specifies that for a Second-Order Integrator block, the **Upper limit x** parameter must have a value equal to `inf`. Constraint cc2 additionally specifies that the **Lower limit x** parameter must have a value equal to zero. Constraint cc specifies that for this check to pass, both cc1 and cc2 must pass. `constraints` is a cell array of the block constraints.

3. Define a new Model Advisor check in a new function, `check2`. Use the function `Advisor.authoring.createBlockConstraintCheck` to create a Model Advisor check for the block constraints defined by the function `createConstraints_Check2`.

```
function check2()

    rec = Advisor.authoring.createBlockConstraintCheck('mathworks.check_0002',...
        'Constraints',@createConstraints_Check2);

    rec.Title = 'Example 2: Check three block parameter constraints';
    rec.TitleTips = 'Example check three block parameter constraints';

    mdladvRoot = ModelAdvisor.Root;
    mdladvRoot.publish(rec,'Example: My Group')

end
```

```
function constraints = createConstraints_Check2()

    cc1=Advisor.authoring.PositiveBlockParameterConstraint;
```

```

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

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

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

constraints = {cc1,cc2,cc};

```

end

Create and Run Model Advisor Checks

1. To register the new checks, use an `sl_customization.m` file. For this example, rename the `sl_customization_DefineChecks` function and file to `sl_customization`.

```

function sl_customization(cm)

    % register custom checks.
    cm.addModelAdvisorCheckFcn(@check1);
    cm.addModelAdvisorCheckFcn(@check2);

```

2. At the command prompt, create the **Example 1: Check block parameter constraints** and **Example 2: Check block parameter constraints** checks by typing this command:

```
Advisor.Manager.refresh_customizations
```

3. At the command prompt, open the model `sldemo_bounce`.

```
open_system('sldemo_bounce')
```

4. In the **Modeling** tab, select **Model Advisor** to open the Model Advisor.

5. In the left pane, select **By Product > Example: My Group**.

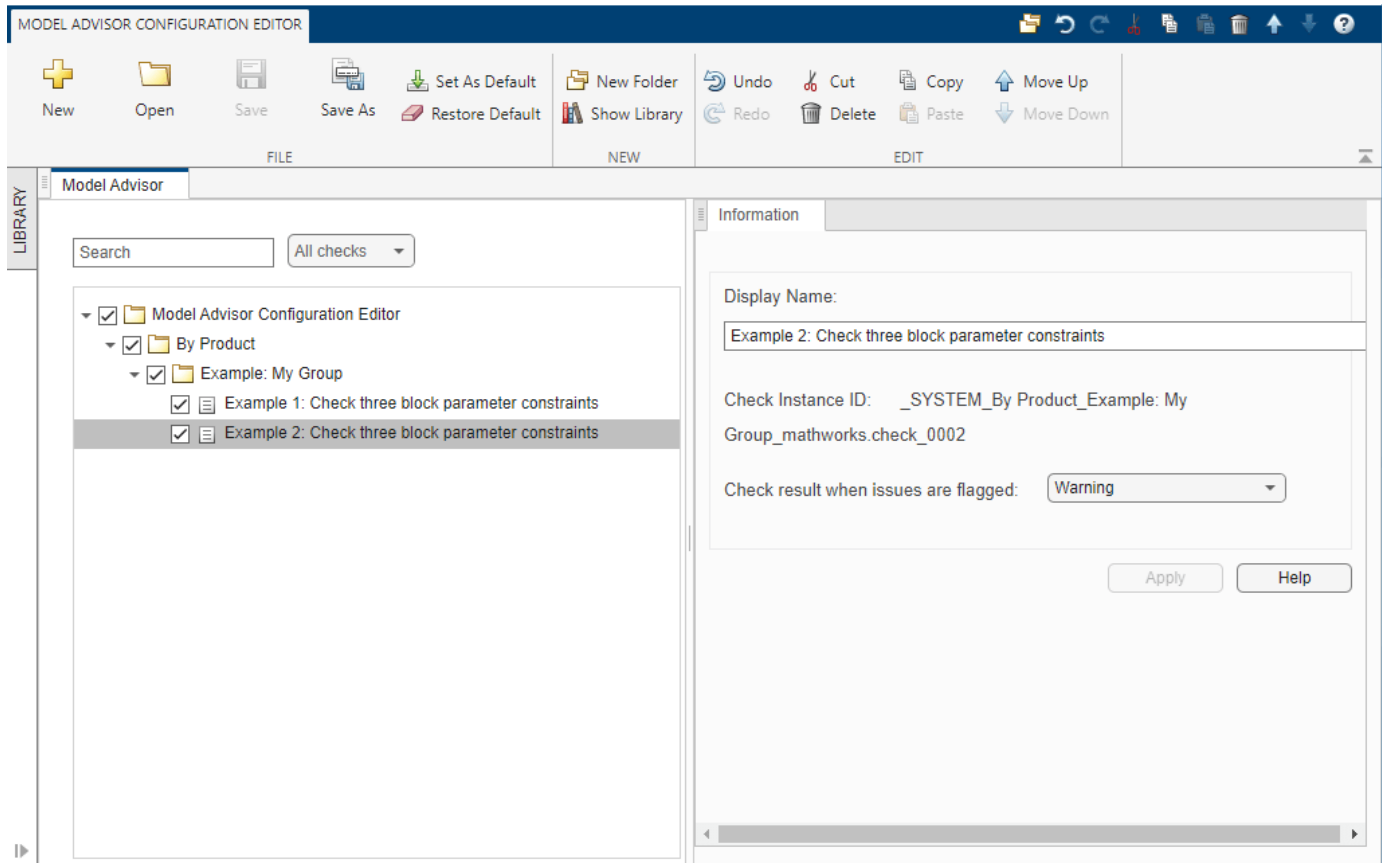
6. Click **Run Checks**.

The **Example 1: Check three block parameter constraints** check produces a warning because the Gain block has a value of -0.8 . The **Example 2: Check three block parameter constraints** check passes because the Second-Order Integrator block meets both constraints.

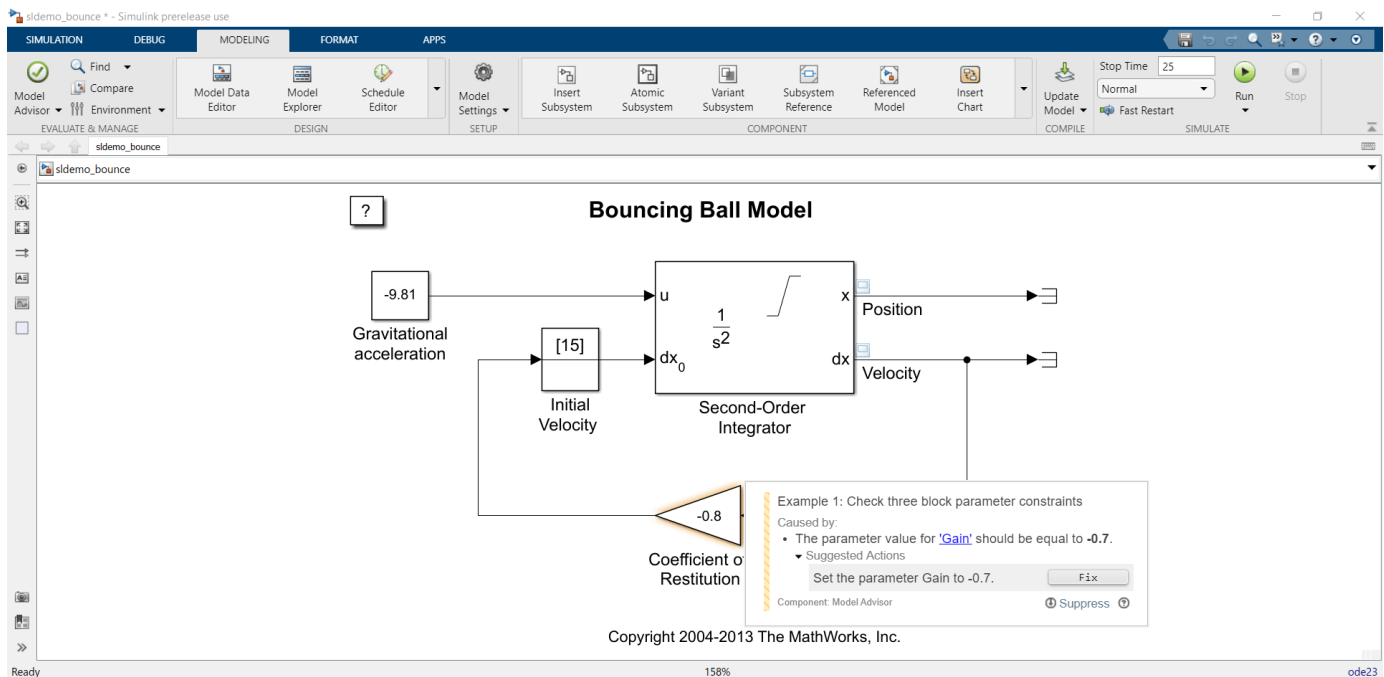
Create Model Advisor Edit-Time Checks using Constraints

You can use edit-time checking to highlight blocks with block constraint violations in the model canvas. You can choose which Model Advisor checks evaluate during edit-time checking by selecting the desired checks in the Model Advisor Configuration Editor and saving a custom configuration.

1. To open the Model Advisor Configuration Editor, open the Model Advisor and select **Open > Open Configuration Editor**.
2. The checks you created appear in the **By Product > Example: My Group**.
3. For this example, delete all folders except for the **Example: My Group** folder.



4. Click **Save** and save the configuration as `my_config.json`.
5. In the dialog box, click **No** because you do not want to set this configuration as the default configuration.
6. In the Simulink model editor, on the **Modeling** tab, click **Model Advisor > Edit-Time Checks**.
7. For the **Model advisor configuration file** parameter, click **Browse**, and select `my_config.json`.
8. Select the **Edit-Time Checks** parameter and close the Configuration Parameters dialog box.
9. Click on the highlighted block in the model to view the edit-time warning.



You can edit the block parameter values from the edit-time check diagnostics window by clicking the **Fix** button or by clicking the hyperlinks of the unsupported parameter to open the **Block Parameters** window.

See Also

[PositiveBlockParameterConstraint](#) | [NegativeBlockParameterConstraint](#) | [PositiveModelParameterConstraint](#) | [NegativeModelParameterConstraint](#) | [PositiveBlockTypeConstraint](#) | [NegativeBlockTypeConstraint](#) | [Advisor.authoring.generateBlockConstraintsDataFile](#)

More About

- “Check Model Compliance Using Edit-Time Checking” on page 3-6
- “Define Edit-Time Checks to Comply with Conditions That You Specify with the Model Advisor” on page 6-9
- “Define Custom Edit-Time Checks that Fix Issues in Architecture Models” on page 6-17

Define Custom Model Advisor Checks

You can create your own conditions and model configuration settings for the Model Advisor to review by defining custom checks. You can create custom checks that run during edit-time and in the Model Advisor or only in the Model Advisor.

Custom edit-time checks help you identify issues earlier in the model design process, but they look only at blocks and signals at the same level of the model or subsystem that a user is editing. However, these checks do aggregate over the levels of a model hierarchy and report issues in the Model Advisor. If your check must check for impacted blocks at other levels of the model, create a custom check that runs only in the Model Advisor. For example, if your check must check for mismatched From and Goto blocks across a model hierarchy, define this check to run only in the Model Advisor.

These steps show the process for creating checks that run during edit-time or only in the Model Advisor.

- 1 “Create `sl_customization` Function” on page 6-45
- 2 “Register Custom Checks” on page 6-45
- 3 “Create Check Definition Function” on page 6-46
 - a “Create an Instance of the `ModelAdvisor.Check` Class” on page 6-46
 - b Define your custom check by following the steps in either “Define Custom Model Advisor Checks” on page 6-46 or “Define Custom Edit-Time Checks” on page 6-47.
 - c “Define Check Input Parameters” on page 6-48
 - d “Publish Custom Check” on page 6-49

Create `sl_customization` Function

To define a custom check, begin by creating an `sl_customization.m` file on the MATLAB path. In the `sl_customization.m` file, create an `sl_customization` function. The `sl_customization` function accepts one argument, a customization manager object:

```
function sl_customization(cm)
```

Tip

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

Register Custom Checks

To register custom checks, use the `addModelAdvisorCheckFcn` method, which is part of the customization manager object that you input to the `sl_customization` function. This code shows a sample `sl_customization.m` function:

```
function sl_customization(cm)
% register custom checks
```

```

cm.addModelAdvisorCheckFcn(@defineModelAdvisorChecks);

% -----
% defines Model Advisor Checks
% -----
function defineModelAdvisorChecks
defineDetailStyleCheck;
defineConfigurationParameterCheck;
defineNewBlockConstraintCheck;
defineEditTimeChecks;

```

The `addModelAdvisorCheckFcn` method registers the checks to the **By Product** folder of the Model Advisor. The `defineModelAdvisorChecks` argument is a handle to the function that contains calls to the functions that define the custom checks. For each custom Model Advisor check that you create, you should create a check definition function. You can create one check definition function for your edit-time checks because each edit-time check contains its own class definition.

Create Check Definition Function

The check definition function defines the actions that the Model Advisor takes when you run the check. These sections describe the key components of the check definition function for custom edit-time checks and checks that run only in the Model Advisor.

Create an Instance of the `ModelAdvisor.Check` Class

For each custom check, create one instance of the `ModelAdvisor.Check` class. Use the `ModelAdvisor.Check` properties and methods to define the check user interface and actions. This table describes some key check components.

Contents	Description
Check ID (required)	Uniquely identifies the check. The Model Advisor uses this ID to access the check.
<i>(Custom Model Advisor check only)</i> Handle to the check callback function (required)	Function that specifies the contents of a check.
<i>(Custom Model Advisor check only)</i> Handle to action callback function (optional)	Adds a fixing action.
<i>(Custom Edit-time check only)</i> Handle to class (required)	Derived class that defines the actions for the edit-time check. Optionally, this class can also define a fix for the edit-time check.
Check name (recommended)	Specifies a name for the check in the Model Advisor.
Model compiling (optional)	Specifies whether the model is compiled for check analysis. The <code>PostCompileForCodegen</code> value of the <code>CallbackContext</code> property is not supported for edit-time checks.
Input parameters (optional)	Adds input parameters that request input from the user. The Model Advisor uses the input to perform the check.

Define Custom Model Advisor Checks

For a custom check that only appears in the Model Advisor, the check definition function contains a check callback function that specifies the actions that you want the Model Advisor to perform on a

model or subsystem. Define the check callback function and pass a handle to it to the `setCallbackFcn` method. The Model Advisor executes the callback function when you run the check. Callback functions provide one or more return arguments that display the results after executing the check. The Model Advisor executes the callback function when you run the check.

If you are specifying a custom check fix, the check definition function should also contain an action callback function. In the check definition function, create an instance of the `ModelAdvisor.Action` class. Define the action callback function and pass a handle to it to the `setCallbackFcn` method. In the Model Advisor, the check user clicks **Fix** to apply the custom fix to their model.

Callback and action callback functions provide one or more return arguments for displaying the results after executing the check. See “Create the Check Callback Definition Function” on page 6-23 and “Create the Action Callback Definition Function” on page 6-24.

To use default formatting for Model Advisor results, specify the callback function type as `DisplayStyle` in the `setCallbackFcn` method. If the default formatting does not meet your needs, use either the `ModelAdvisor.FormatTemplate` class or these other Model Advisor formatting classes:

Class	Description
<code>ModelAdvisor.Text</code>	Create a Model Advisor text output.
<code>ModelAdvisor.List</code>	Create a list.
<code>ModelAdvisor.Table</code>	Create a table.
<code>ModelAdvisor.Paragraph</code>	Create and format a paragraph.
<code>ModelAdvisor.LineBreak</code>	Insert a line break.
<code>ModelAdvisor.Image</code>	Include an image in the Model Advisor output.

Define Custom Edit-Time Checks

To create a custom edit-time check, create a MATLAB class that derives from the `ModelAdvisor.EdittimeCheck` class. In the check definition function, specify this class as the value of the `ModelAdvisor.CheckCallbackHandle` property. Inside the derived class, define these methods:

- Define a method that specifies the check ID and the `ModelAdvisor.EdittimeCheck.TraversalType` properties of the check. The `TraversalType` property specifies how the Model Advisor runs the check.
- Define a `blockDiscovered` method that looks for blocks that violate your edit-time algorithm.
- If the violation is on a block, highlight the block during edit-time by creating a `ModelAdvisor.ResultDetail` violation object with the `Type` property set to the default value of `SID`. If the violation is on a signal, highlight the signal by creating a violation object with the `Type` property set to `Signal`.
- If you specify a `TraversalType` property of `edittimecheck.TraversalTypes.ACTIVEGRAPH`, define a `finishedTraversal` method that specifies what the edit-time check does with the data the check collects as part of the `blockDiscovered` method.
- Optionally, define a `fix` method for edit-time check violations.

For an example, see “Define Edit-Time Checks to Comply with Conditions That You Specify with the Model Advisor” on page 6-9.

To help prevent custom edit-time checks from negatively impacting performance as you edit your model, the Model Advisor automatically disables custom edit-time checks if, in the current MATLAB session, the check takes longer than 500 milliseconds to execute in at least three different Simulink models.

If the Model Advisor disables a custom edit-time check, you will see a warning on the Simulink canvas. You can re-enable the edit-time check by either:

- Clicking the hyperlink text in the warning.
- Passing the check identifier, `checkID`, to the function `edittime.enableCheck`:
`edittime.enableCheck(checkID)`.

To prevent a custom edit-time check from being disabled, author the check so that the check executes in less than 500 milliseconds on your models.

Define Check Input Parameters

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

Specify the layout of input parameters in the Model Advisor by using these methods.

Purpose	Method
Specifies the size of the input parameter grid	<code>.setInputParametersLayoutGrid</code>
Specifies the number of rows the parameter occupies in the input parameter layout grid.	<code>setRowSpan</code>
Specifies the number of columns the parameter occupies in the input parameter layout grid.	<code>setColSpan</code>

The Model Advisor displays input parameters in the **Input Parameters** box.

Display and Enable Check

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

- If the `Visible` property is `false`, the check or task is not displayed in the Model Advisor and the `Enable` and `Value` properties are ignored.
- If the `Visible` property is `true` and the `Enable` property is `false`:
 - The check is displayed in the Model Advisor.
 - The initial status of the check is `Value`.
 - The check box appears dimmed.
- If the `Visible` property is `true` and the `Enabled` property is `true`, the check or task is displayed in the Model Advisor and the check box is active.

Publish Custom Check

Create a folder for custom checks in the **By Product** folder by using the `publish` method. Then, use the Model Advisor Configuration Editor to customize the folders within the Model Advisor tree. For more information, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.

See Also

`ModelAdvisor.Check` | `ModelAdvisor.EdittimeCheck` | `ModelAdvisor.InputParameter` | `ModelAdvisor.Action` | `publish`

Related Examples

- “Create and Deploy a Model Advisor Custom Configuration” on page 7-24
- “Define Edit-Time Checks to Comply with Conditions That You Specify with the Model Advisor” on page 6-9
- “Define Custom Edit-Time Checks that Fix Issues in Architecture Models” on page 6-17
- “Fix a Model to Comply with Conditions that You Specify with the Model Advisor” on page 6-21
- “Create Model Advisor Check for Model Configuration Parameters” on page 6-27
- “Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters” on page 6-38

Define the Compile Option for Custom Model Advisor Checks

Depending on the implementation of your model and what you want your custom check to achieve, it is important that you specify the correct compile option. You specify the compile option for the check definition function of a `ModelAdvisor.Check` object by setting the `CallbackContext` property as follows:

- `None` specifies that the Model Advisor does not have to compile your model before analysis by your custom check. `None` is the default setting of the `CallbackContext` property.
- `PostCompile` specifies that the Model Advisor must compile the model to update the model diagram and then simulate the model to execute your custom check. The Model Advisor does not flag modeling issues that fail during code generation because these issues do not affect the simulated model.
- `PostCompileForCodegen` specifies that the Model Advisor must compile and update the model diagram specifically for code generation, but does not simulate the model. Use this option for Model Advisor checks that analyze the code generation readiness of the model. This option is not supported for custom edit-time checks.

Checks that Evaluate the Code Generation Readiness of the Model

You can create custom Model Advisor checks that enable the Model Advisor engine to identify code generation setup issues in a model at an earlier stage so you can avoid unexpected errors during code generation. For example, in this model, the `Red` enumeration in `BasicColors` and `OtherColors` are OK for use in a simulated model. In the generated code, however, these `Red` enumerations result in an enumeration clash. By using the `'PostCompileForCodegen'` option, your custom Model Advisor check can identify this type of code generation setup issue.

```

classdef BasicColors < Simulink.IntEnumType
    enumeration
        Red (0)
        Blue (1)
        Green (2)
    end
    methods (Static = true)
        function retVal = addClassNameToEnumNames()
            retVal = false;
        end
    end
end

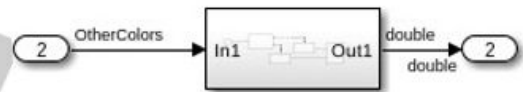
```



```

classdef OtherColors < Simulink.IntEnumType
    enumeration
        Red (0)
        Yellow (1)
        Magenta (2)
    end
    methods (Static = true)
        function retVal = addClassNameToEnumNames()
            retVal = false;
        end
    end
end

```



The 'PostCompileForCodegen' option compiles the model for all variant choices. This compilation enables you to analyze possible issues present in the generated code for active and inactive variant paths in the model. An example is provided in “Create Custom Check to Evaluate Active and Inactive Variant Paths from a Model” on page 6-51.

Create Custom Check to Evaluate Active and Inactive Variant Paths from a Model

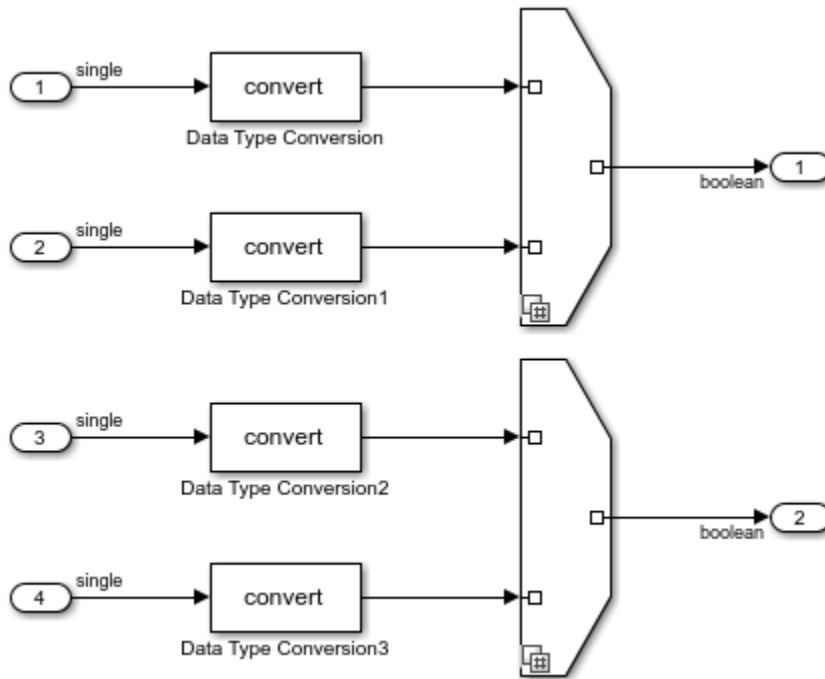
This example shows the creation of a custom Model Advisor check that evaluates active and inactive variant paths from a variant system model. The example provides Model Advisor results that demonstrate why you use `PostCompileForCodegen` instead of `PostCompile` as the value for the `ModelAdvisor.Check.CallbackContext` property when generating code from the model is the final objective.

Update Model to Analyze All Variant Choices

For the Model Advisor to evaluate active and inactive paths in a variant system, you must set the **Variant activation time** parameter to `Code compile` or `startup` for the variant blocks (Variant Sink, Variant Source, and Variant Subsystem, Variant Model, Variant Assembly Subsystem). You must also set the **System target file** configuration parameter to `ert.tlc`.

Note: Selecting this option can affect the execution time and increase the time it takes for the Model Advisor to evaluate the model.

- 1 Open the example model `ex_check_compile_code_gen`.
- 2 For each Variant Source block, open the block parameters and set the **Variant activation time** parameter to `Code compile`.
- 3 Save the model to your local working folder.



Create an `sl_customization` Function

In your working folder, create this `sl_customization` function and save it.

```
function sl_customization(cm)

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

% -----
% defines Model Advisor Checks
% -----
function defineModelAdvisorCheck
CheckSingleToBoolConversion;
```

The `sl_customization` function accepts a customization manager object. The customization manager object includes the `addModelAdvisorCheckFcn` method for registering custom checks. The input to this method is a handle to a function (`defineModelAdvisorCheck`). This function contains a call to the check definition function that corresponds to the custom check.

Open and inspect the check definition function, `CheckSingleToBoolConversion.m`:

```
function CheckSingleToBoolConversion
mdladvRoot = ModelAdvisor.Root;
```

```

rec = ModelAdvisor.Check('exampleCheck1');
rec.Title = 'Check to identify Single to Bool conversions';
rec.TitleID = 'custom.dtcCheck.CompileForCodegen1';
rec.TitleTips = 'Custom check to identify Single to Bool conversions';
rec.setCallbackFcn(@DetailStyleCallback,'None','DetailStyle');
rec.CallbackContext = 'PostCompileForCodegen'; % Compile for Code Generation

mdladvRoot.publish(rec, 'Demo');

end

function DetailStyleCallback(system, CheckObj)

mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);

violationBlks = find_system(system, 'BlockType', 'DataTypeConversion');
for ii = numel(violationBlks):-1:1
    dtcBlk = violationBlks{ii};
    compDataTypes = get_param(dtcBlk, 'CompiledPortDataTypes');
    if isempty(compDataTypes)
        violationBlks(ii) = [];
        continue;
    end
    if ~(strcmp(compDataTypes.Inport, 'single') && strcmp(compDataTypes.Outport, 'boolean'))
        violationBlks(ii) = [];
        continue;
    end
end

if isempty(violationBlks)
    ElementResults = ModelAdvisor.ResultDetail;
    ElementResults(1,numel(violationBlks))=ModelAdvisor.ResultDetail;
    ElementResults.IsInformer = true;
    ElementResults.Description = 'This check looks for data type conversion blocks that convert single to boolean';
    ElementResults.Status = 'Check has passed. No data type conversion blocks that convert single to boolean';
    mdladvObj.setCheckResultStatus(true);
else
    for i=1:numel(violationBlks)
        ElementResults(1,i) = ModelAdvisor.ResultDetail;
    end
    for i=1:numel(ElementResults)
        ModelAdvisor.ResultDetail.setData(ElementResults(i), 'SID',violationBlks{i});
        ElementResults(i).Description = 'This check looks for data type conversion blocks that convert single to boolean';
        ElementResults(i).Status = 'Check has failed. The following data type conversion blocks convert single to boolean';
        ElementResults(i).RecAction = 'Modify the model to avoid converting data type from single to boolean';
    end
    mdladvObj.setCheckResultStatus(false);
    mdladvObj.setActionEnable(true);
end
CheckObj.setResultDetails(ElementResults);
end

```

For more information on creating custom checks, see “Define Custom Model Advisor Checks” on page 6-45.

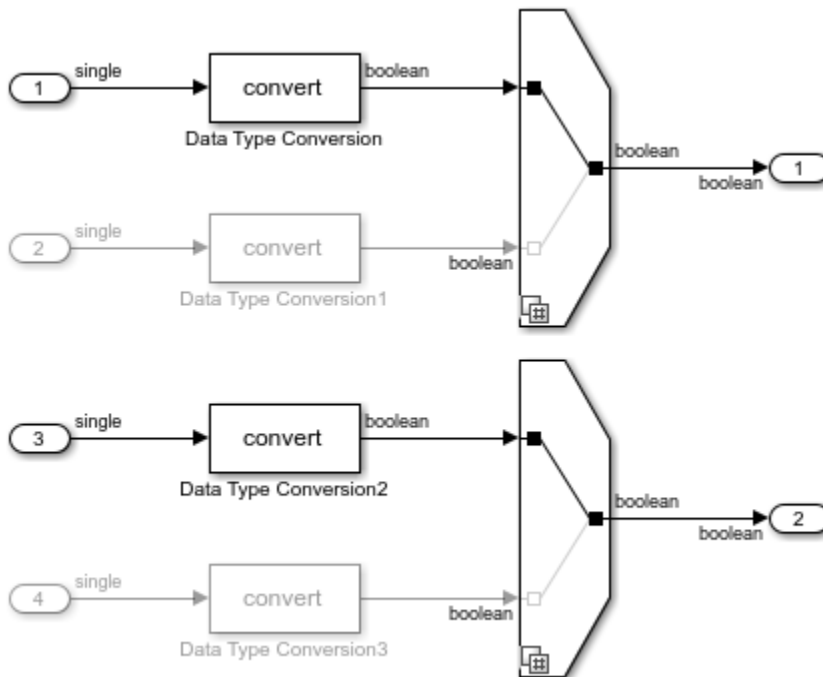
Open Model Advisor and Execute Custom Check

Before opening the Model Advisor and running the custom check, you must refresh the Model Advisor check information cache. In the MATLAB Command Window, enter:

```
Advisor.Manager.refresh_customizations
```

To open the Model Advisor and execute the custom check:

- 1 Open your saved model.
- 2 In the **Modeling** tab, select **Model Advisor**. A **System Selector - Model Advisor** dialog box opens. Click **OK**. The Model Advisor opens.
- 3 In the left pane, select **By Product > Demo > Check to identify Single to Bool conversion**.
- 4 Right-click the check and select **Run this Check**. The Model Advisor compiles the model and executes the check. The Model Advisor updates the model diagram. The inactive variant paths appear dimmed.



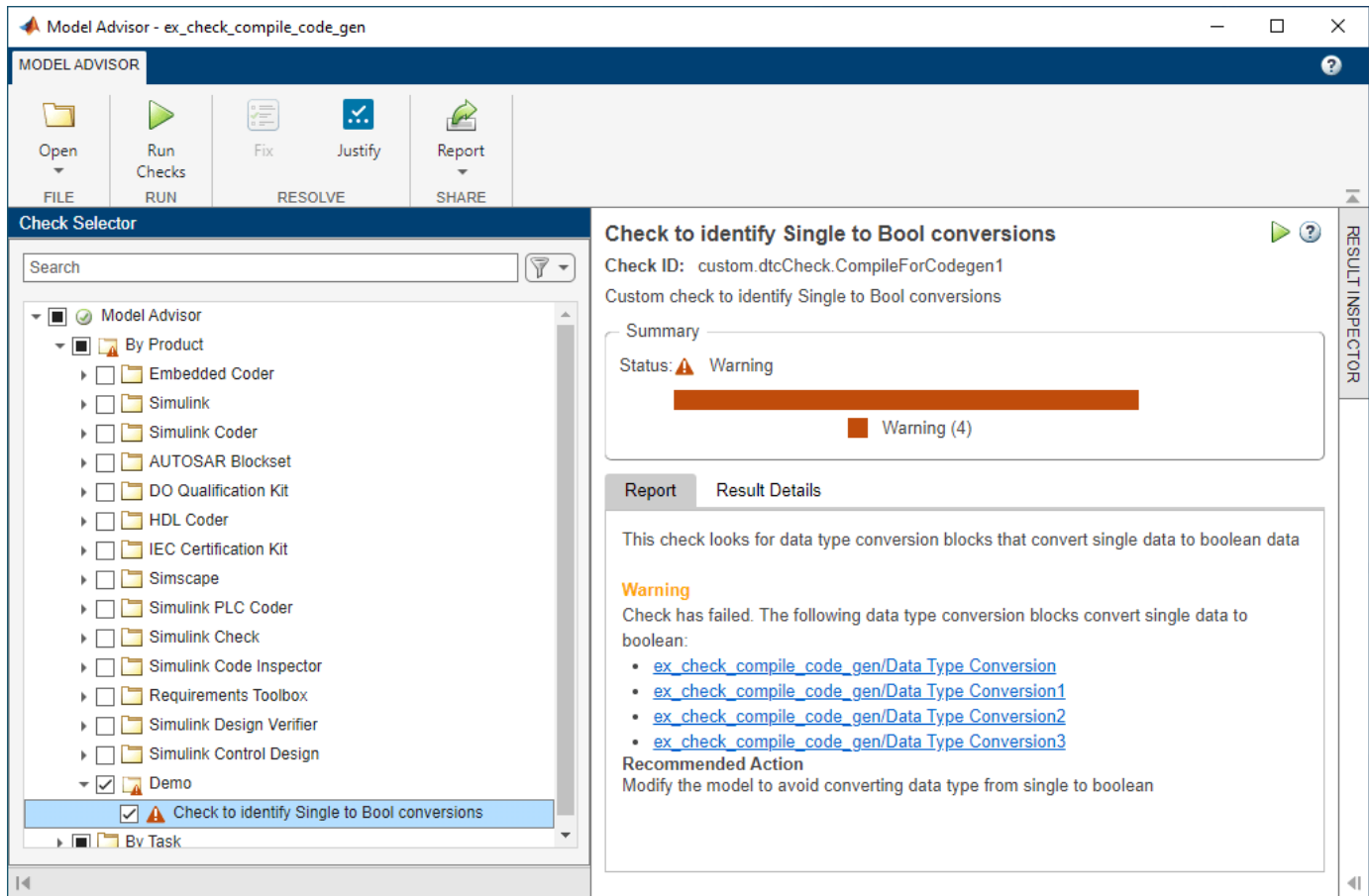
Review the Model Advisor Results

Review the check analysis results in the Model Advisor. Click the hyperlinks to open the violating block in the model editor.

In this example, because you defined the compile option in the `sl_customization.m` file as

```
rec.CallbackContext = 'PostCompileForCodegen';
```

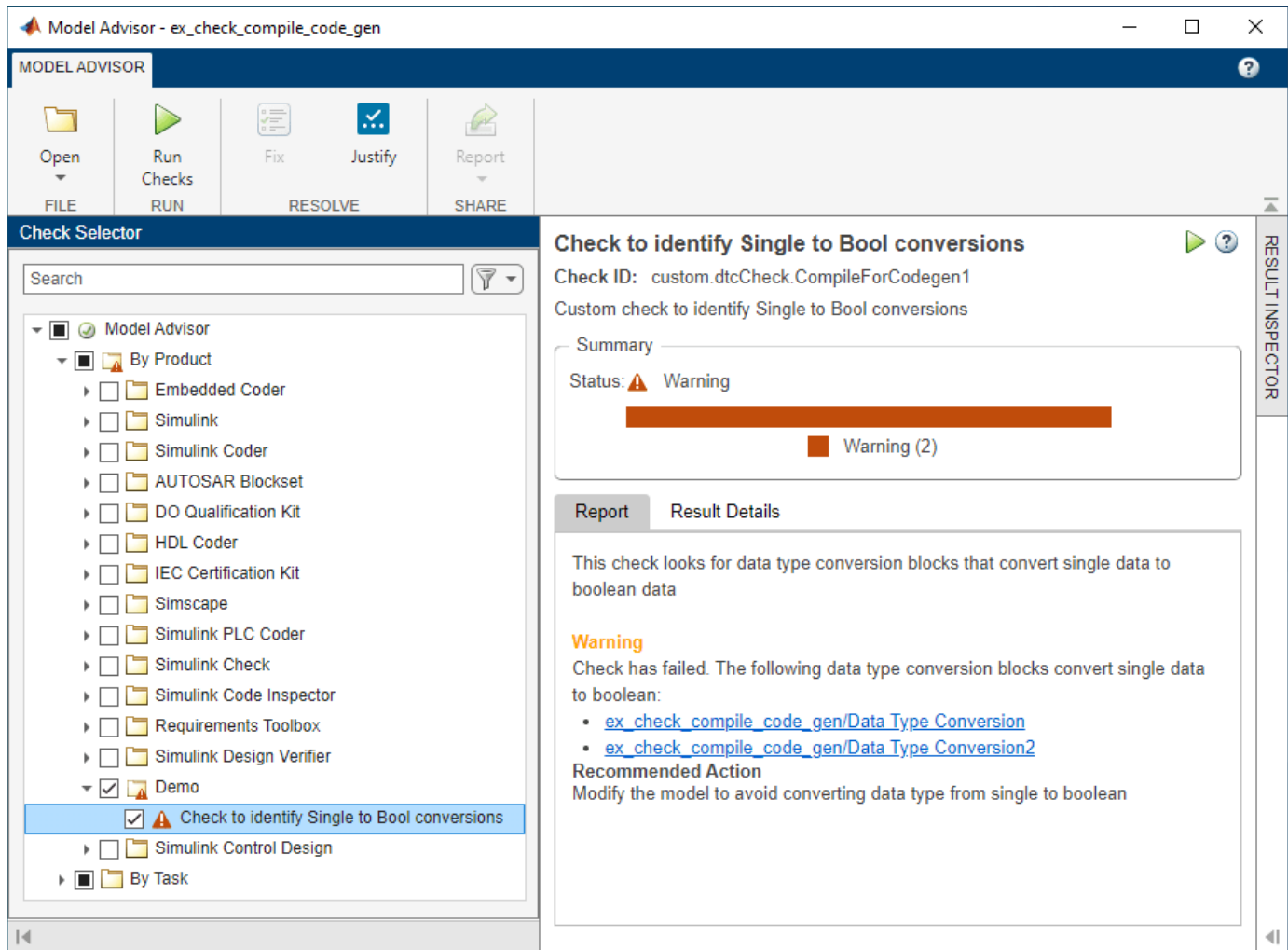
the Model Advisor generates warnings for the Data Type Conversion blocks in the active paths and the inactive paths of the variant systems.



If you defined the compile option in the `sl_customization.m` file as

```
rec.CallbackContext = 'PostCompile';
```

the results include only the Data Type Conversion blocks in the active path.



See Also

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

More About

- “Define Custom Model Advisor Checks” on page 6-45
- “Variant Systems”

Exclude Blocks From Custom Checks

This example shows how to exclude blocks from custom checks. To save time during model development and verification, you can exclude individual blocks from custom checks during a Model Advisor analysis. To exclude custom checks from Simulink blocks and Stateflow charts, use the `ModelAdvisor.Check.supportExclusion` and `Simulink.ModelAdvisor.filterResultWithExclusion` functions in the check definition file.

Update the Check Definition File

- 1 Open the example that contains the supporting files for this example:

```
openExample('slcheck/CreateAndDeployAModelAdvisorCustomConfigurationExample')
```

- 2 Delete the supporting files from your working directory except for the `AdvisorCustomizationExample` model and the `defineDetailStyleCheck.m` and `sl_customization.m` files.

- 3 Open the `sl_customization` file and modify it as follows:

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

% Copyright 2019 The MathWorks, Inc.

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

% -----
% defines Model Advisor Checks
% -----
function defineModelAdvisorChecks
defineDetailStyleCheck;
```

- 4 Open the `defineDetailStyleCheck` file.

- 5 To update the **Check whether block names appear below blocks** check to exclude blocks during Model Advisor analysis, make two modifications to the `defineDetailStyleCheck` file.

- a Enable the **Check whether block names appear below blocks** check to support check exclusions by using the `ModelAdvisor.Check.supportExclusion` property. After `rec.setCallbackFcn(@DetailStyleCallback, 'None', 'DetailStyle');`, add `rec.supportExclusion = true;`. The first section of the function `defineDetailStyleCheck` now looks like:

```
% Create ModelAdvisor.Check object and set properties.
rec = ModelAdvisor.Check('com.mathworks.sample.detailStyle');
rec.Title = 'Check whether block names appear below blocks';
rec.TitleTips = 'Check position of block names';
rec.setCallbackFcn(@DetailStyleCallback, 'None', 'DetailStyle');
rec.supportExclusion = true;
```

- b Use the `Simulink.ModelAdvisor.filterResultWithExclusion` function to filter model objects causing a check warning or failure with checks that have exclusions enabled. To do this, modify the `DetailStyleCallback(system, CheckObj)` function as follows:

```
% Find all blocks whose name does not appear below blocks
violationBlks = find_system(system, 'Type', 'block', ...
    'NamePlacement', 'alternate', ...
```

```
'ShowName', 'on');  
violationBlks = mdladvObj.filterResultWithExclusion(violationBlks);
```

- 6 Save the DefineDetailStyleCheck file. If you are asked if it is OK to overwrite the file, click **OK**.

Create and Save Exclusions

- 1 In order for your customizations to be visible in the Model Advisor, you must refresh the Model Advisor check information cache. At the MATLAB command prompt, type this command:

```
Advisor.Manager.refresh_customizations();
```
- 2 To open the model, double-click `AdvisorCustomizationExample.slx`.
- 3 In the **Modeling** tab, select **Model Advisor** to open the Model Advisor.
- 4 In the left pane of the Model Advisor window, select the **By Product > Demo > Check whether block names appear below blocks** check. In the right pane, select **Run Checks**. The check fails.
- 5 In the model window, right-click the X block and select **Model Advisor > Exclude block only > Select Checks**. Navigate to the **Demo** folder and select the **Check whether block names appear below blocks** check.
- 6 In the Model Advisor Exclusion Editor, click **Save** to create an exclusion file.
- 7 In the model window, open the **Amplifier** subsystem and right-click the **GainBlock** block and select **Model Advisor > Exclude block only > Select Checks**. Navigate to the **Demo** folder and select the **Check whether block names appear below blocks** check.
- 8 In the Model Advisor Exclusion Editor, click **Save** to update the exclusion file.

Review Exclusions

- 1 In the left pane of the Model Advisor window, select the **By Product > Demo > Check whether block names appear below blocks** check. In the right pane, select **Run Checks**. The check now passes. In the right-pane of the Model Advisor window, you can see the **Check Exclusion Rules** that the Model Advisor applies during the analysis.
- 2 Close the model and the Model Advisor.

See Also

[supportExclusion](#) | [Simulink.ModelAdvisor](#)

Related Examples

- [Excluding Blocks From Model Advisor Checks](#) on page 3-9

More About

- [“Run Model Advisor Checks and Review Results”](#) on page 3-4

Create Help for Custom Model Advisor Checks

You can define help files for your custom Model Advisor checks to make the checks easier to use. Custom help files allow you to verify the check capabilities and avoid potential warnings in the model.

You can define custom help for:

- Custom checks - To Add help for custom Model Advisor checks, use `ModelAdvisor.Check.setHelp`. For more information, see `setHelp`.
- Folders that have custom checks - To add help for folders that contain custom Model Advisor checks, use `ModelAdvisor.Group.setHelp`. For more information, see `setHelp`.

To point the custom check help to a PDF or an HTML page of your choice:

- 1 Open the `sl_customization.m` file.
- 2 Use `setHelp()` on the check or group object created in the `sl_customization.m` file.

```
setHelp('format','webpage','path','custom_path');
```

The supported name-value arguments are:

Format - "webpage" , "pdf"

Path - Path of the user-defined help page or document

Example:

```
checkObj = ModelAdvisor.Check('SimplePassFailCheck');
checkObj.setHelp('format','webpage','path','custom_path');
```

- 3 Close the `sl_customization.m` file.
- 4 Refresh the customizations by entering:

```
Advisor.Manager.refresh_customizations
```

To view the custom help, right-click the custom checks or the folder and click **What's This?**.

See Also

“Define Custom Model Advisor Checks” on page 6-45

“Create and Deploy a Model Advisor Custom Configuration” on page 7-24

Model Advisor Customization

Customize the Configuration of the Model Advisor Overview

You can use Model Advisor API and the Model Advisor Configuration Editor to customize the configuration of the Model Advisor, including:

- Define which built-in (shipped) and custom Model Advisor checks are available in the Model Advisor and their order of execution.
- Create custom folders and organize checks.
- Designate the default configuration file for the Model Advisor.
- Associate a configuration file with a model.
- Suppress the warning about missing checks when loading the Model Advisor configuration.
- Upgrade old configuration files to be compatible with newer versions of MATLAB.

To customize the Model Advisor to include custom checks and a custom configuration, perform the following tasks:

- 1** (Optional) Author custom checks in a customization file. For more information, see “Create Model Advisor Checks”.
- 2** Use the Model Advisor Configuration Editor to specify the folders and checks that you want to include in your custom Model Advisor configuration. For more information, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.
- 3** Update your Simulink environment so that the Model Advisor uses your configuration files. For more information, see “Update the Environment to Include Your Custom Configuration” on page 7-20.
- 4** Open the Model Advisor, load the configuration, and verify that the Model Advisor is using the correct configuration and that your checks are available. Optionally, you can associate the configuration with your model. For more information, see “Load and Associate a Custom Configuration with a Model” on page 7-21.
- 5** (Optional) Deploy the custom configurations to your users. For more information, see “Deploy Custom Configurations” on page 7-23.
- 6** Verify that models comply with modeling guidelines. For more information, see “Run Model Advisor Checks and Review Results” on page 3-4.

Use the Model Advisor Configuration Editor to Customize the Model Advisor

Overview of the Model Advisor Configuration Editor

The Model Advisor Configuration Editor provides a way for you to specify the checks that you want to use for edit-time checking, as well as the checks included in the Model Advisor. This organizational hierarchy is saved as a configuration file, which is loaded when you initiate the Model Advisor. You can use the Model Advisor Configuration Editor to modify existing configurations, create new Model Advisor configurations, and specify the default configuration.

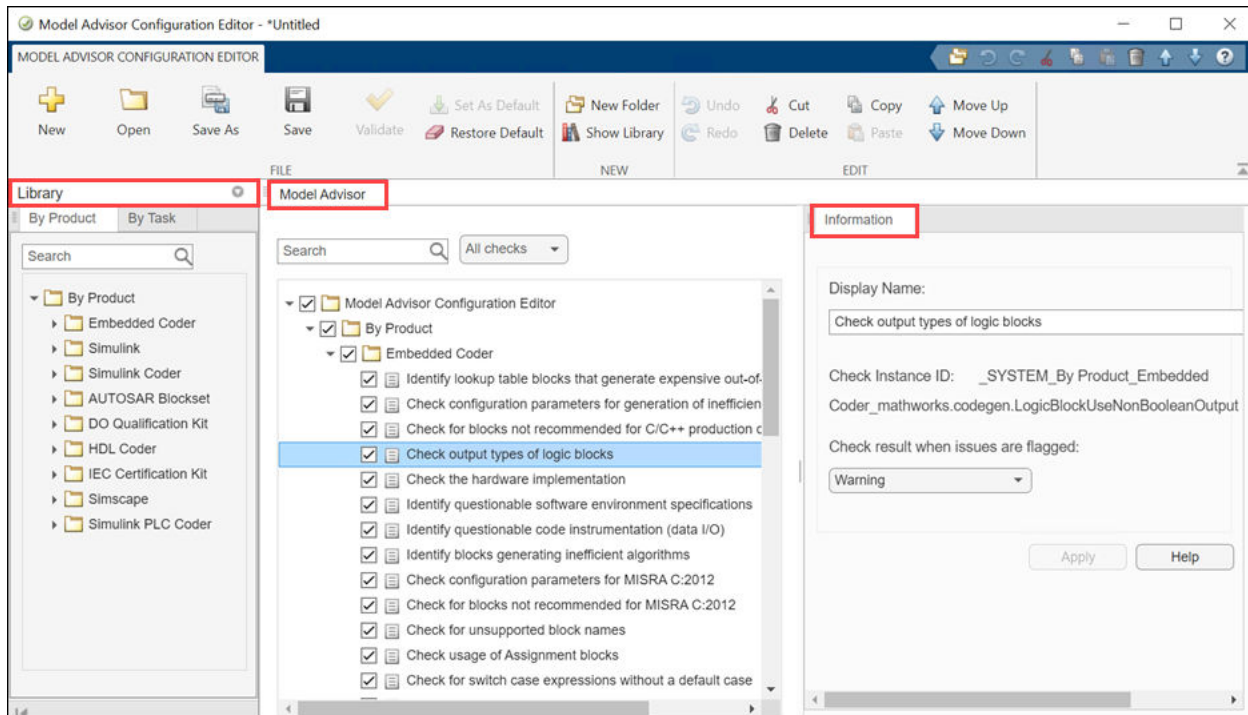
The Model Advisor Configuration Editor gives you the flexibility to customize the Model Advisor analysis to meet the needs of your organization by allowing you to:

- Review all available Model Advisor checks.
- Add, remove, and organize built-in checks and folders in the Model Advisor tree.
- Integrate custom Model Advisor checks in your verification and validation workflow.
- Disable and enable checks and folders.
- Rename checks and folders.
- Specify whether a check is marked as a warning or failure when it is flagged during a Model Advisor analysis.
- Suppress the warning about missing checks when loading the Model Advisor configuration.
- Upgrade old check configuration files to be compatible with newer versions of MATLAB.

The Model Advisor Configuration Editor includes:

- The **Library** pane — A read-only pane that lists all checks and folders that are available for use in the configuration, delineated by the **By Product** and **By Task** tabs. To permanently display the Library tab, click **Show Library** on the toolbar.
- The **Model Advisor** pane — This pane lists the checks and folders in the current Model Advisor configuration, filtered by:
 - **All checks** — Lists all Model Advisor checks included in the current configuration
 - **Edit time supported checks** — Lists only the Model Advisor checks that are supported as edit-time checks
- **Information** tab — This tab provides:
 - Details about the check or folder, such as the **Display Name**, **Check Instance ID** or **Check Group ID**, and the **Check result when issues are flagged**.
 - The **Fix** button, to fix outdated checks.
 - The **Delete** button to delete checks that are no longer supported.

Use the search functionality in the **Library** and **Model Advisor** panes to locate specific checks and folders.



Open the Model Advisor Configuration Editor

Before opening the Model Advisor Configuration Editor, verify that the current folder is writable. If the folder is not writable, you see an error message when you start the Model Advisor Configuration Editor.

When implementing custom checks or Model Advisor customizations by using the Model Advisor API, you must first update the Simulink environment to include your `sl_customization.m` file. At the MATLAB command line, enter

```
Advisor.Manager.refresh_customizations
```

Use one of these methods to open the Model Advisor Configuration Editor:

- Programmatically — At the MATLAB command line, enter `Simulink.ModelAdvisor.openConfigUI`.
- From the Simulink editor — In the **Modeling** tab, select **Model Advisor > Configuration Editor**.
- From the Model Advisor — Select **Open > Open Configuration Editor**.

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

To create a new configuration, click the **New** button on the toolbar. Use **Save As** to rename the configuration file. Model Advisor configuration files are saved in `.json` format.

Note If your configuration file contains checks that are incompatible with the newer version of MATLAB you use, fix the issues in the configuration as described in “Upgrade Incompatible Checks in Model Advisor Configuration Files” on page 7-7.

Specify a Default Configuration File

You can use the Model Advisor Configuration Editor to specify a default configuration that loads automatically when the Model Advisor opens. To set the default configuration, open the configuration file in the Model Advisor Configuration Editor and click the **Set As Default** button on the toolbar.

Note If you have previously designated a default configuration, you can use **Clear default configuration setting** to clear the setting that designates the current default configuration file. Clicking the button does not modify the configuration that is currently displayed in the Model Advisor Configuration Editor. When you do not specify a default configuration, the Model Advisor uses the standard configuration that is defined by your system administrator.

If you do not specify the configuration file as the default, when you save the file, you are prompted as to whether to make the file the default configuration. To make this file the default configuration, click **Yes**.

To associate a custom configuration with a model, so that the Model Advisor uses that configuration each time you open that model, see “Load and Associate a Custom Configuration with a Model” on page 7-21.

Customize the Model Advisor Configuration

You can use the Model Advisor Configuration Editor to customize the Model Advisor configuration tree, including adding and removing checks and folders and specifying the order in which checks are executed. You can also disable the ability for users to select whether to include or exclude a check from an analysis. You can also use the Model Advisor Configuration Editor to define the input parameters for a check.

Note Checks that are copied from the **Library** tab retain their default parameter settings. When they are pasted into your custom configuration folder, the box beside each check is not selected.

Checks that are copied or cut from a folder in the **Model Advisor** tab retain their user-defined parameter settings. When a check is included in multiple folders, you can specify different parameters for each check individually.

Organize the Hierarchy

You can customize the layout of the checks and folders in the Model Advisor configuration tree by using:

- **New Folder** to create a folder.
- **Copy**, **Cut**, and **Paste** to add, copy, and move checks and folders.
- **Delete** to remove checks and folders.

- **Move Up** or **Move Down** to shift the position of the check or folder in the configuration tree. The folders and checks that are higher in the configuration tree are executed first in the analysis.

Note, you can customize the Model Advisor by using the `ModelAdvisor.Group` and `ModelAdvisor.FactoryGroup` classes instead of the Model Advisor Configuration Editor. However, these APIs are a less flexible and more time-consuming way of customizing the Model Advisor. To place customized checks in custom folders at the top-level of the Model Advisor tree (the Model Advisor root), use the `ModelAdvisor.Group` class. To place customized checks in new folders in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class. You must include methods that register these tasks and folders in the `sl_customization` function.

Enable or Disable Checks

You can use the Model Advisor Configuration Editor to disable the check box control for checks and folders in the Model Advisor. By doing so, the check is still listed in the Model Advisor configuration tree, but it is dimmed and you do not have the ability to add or remove the check from the analysis.

In the **Model Advisor** pane, right-click on a folder or check and select **Disable**. Depending on the check box selection in the Model Advisor Configuration Editor, the following results occur in the Model Advisor:

- If the box beside check is selected in the Model Advisor Configuration Editor, then in the Model Advisor, the check is automatically selected. Because you selected **Disable**, the check is dimmed and you cannot choose to remove the check from the analysis.

If the box beside the check is *not* selected and the **Disable** option is applied in the Model Advisor Configuration Editor, then in the Model Advisor, the check is not selected and you cannot include it in the analysis.

- If the box beside folder is selected in the Model Advisor Configuration Editor, then in the Model Advisor, the checks within the folder are automatically selected. Because you selected **Disable**, the folder and its checks are dimmed and you cannot choose to remove the checks from the analysis.

If the box beside the folder is *not* selected and the **Disable** option is applied in the Model Advisor Configuration Editor, then in the Model Advisor, none of the checks within the folder are selected and you cannot include it in the analysis.

When a check or folder is disabled, you can use the **Enable** option to allow users to determine whether to include the check(s) in an analysis.

Note **Enable** and **Disable** affects the execution of checks in the analysis for both the Model Advisor user interface and edit-time checking.

Specify Parameters for Check Customization

You can use the Model Advisor Configuration Editor to customize a Model Advisor check, such as the display name and input parameters for the check.

In the **Information** tab, review the content that you can customize for the check:

- **Display Name** — Provide a new name for the check, which is displayed in the Model Advisor. Note that changing the display name does not change the check title.

- **Check result when issues are flagged** — Specify whether you want the check to be marked as a warning or failure in the results when the check flags an issue in your model. The default value is `Warning`. Select `Fail` to mark a flagged check as failed in the results.
- **Input Parameters** — Specify additional characteristics and functionality for the check. The Model Advisor uses these parameters to further define the emphasis of the analysis. For example, you can choose to include only subcheck `jc_0736_b` and specify the acceptable number of single-byte spaces in the analysis for Model Advisor check “Check indentation of code in Stateflow states”.

Suppress Warning Message for Missing Checks

The Model Advisor automatically warns you of checks that are missing when loading a Model Advisor configuration. You can use the Model Advisor Configuration Editor to suppress this message. Select the **Model Advisor Configuration Editor** root node and, in the **Information** tab, select **Suppress warning message for missing checks when loading configuration**.

Alternatively, you can programmatically suppress the Model Advisor warning by entering this command at the MATLAB command line:

```
warning('off','Simulink:tools:MALoadConfigMissCorrespondCheck')
```

Upgrade Incompatible Checks in Model Advisor Configuration Files

You can use the Model Advisor Configuration Editor to upgrade your Model Advisor configurations to the newer version of MATLAB. Upgrading configurations enables you to view and use newly introduced or updated input parameters and check IDs. You can also delete the checks that are no longer supported. Model Advisor Configuration Editor facilitates you to automatically bulk update or delete the incompatible checks in your configuration. You can also update or delete incompatible checks individually and then validate the configuration for compatibility.

Update or Delete Checks in Configuration Files Automatically Using Model Advisor

- 1 In Model Advisor Configuration Editor, when you load an old configuration file which contains checks that are incompatible with the newer version of MATLAB in use, you get a dialog box asking whether to automatically fix issues in the configuration. To learn how to load a configuration, see “Open the Model Advisor Configuration Editor” on page 7-4.
- 2 To fix the issues automatically, click **Yes**. Model Advisor alters the fixable checks and deletes the nonfixable checks from the configuration file.
 - A check is fixable if it contains outdated input parameters or check IDs. Or else, it does not have the input parameters or check IDs introduced in the newer version of MATLAB in use.
 - A check is nonfixable if it is no longer supported in the newer version of MATLAB in use.



The Validation Summary window displays the list of updated and deleted checks. For more information, see “View Updated and Deleted Checks in Validation Summary” on page 7-8.

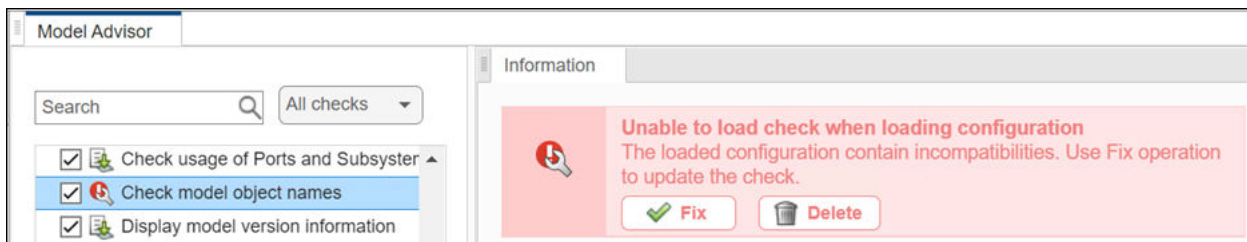
- 3 Click **Save** to save the upgraded configuration file in JSON format. You can now use the upgraded configuration file to review your models in the newer version of MATLAB.

Update or Delete Checks in Configuration Files Individually

- 1 In Model Advisor Configuration Editor, when you load an old configuration file which contains checks that are incompatible with the newer version of MATLAB in use, you get a dialog box

asking whether to automatically fix issues in the configuration. To learn how to load a configuration, see “Open the Model Advisor Configuration Editor” on page 7-4.

- 2 To open the Model Advisor Configuration Editor with check issues highlighted and fix the issues individually, click **No**. The **Model Advisor** pane lists the folders of the loaded configuration. The red cross mark over the folder icon  indicates that the folder contains incompatible checks.
- 3 Expand a folder that contains incompatible checks. The wrench exclamation icon  denotes incompatible checks. When you click an incompatible check on the list, the **Information** tab displays a banner. If the check is a fixable check, you can update it or delete it. If the check is a nonfixable check, you can only delete it.





- 4 If you want Model Advisor to automatically update or delete the rest of the incompatible checks within the configuration file, click the **Validate** button in the toolbar. Model Advisor alters the fixable checks and deletes the nonfixable checks from the configuration file. The Validation Summary window displays the list of updated and deleted checks. For more information, see “View Updated and Deleted Checks in Validation Summary” on page 7-8.
- 5 Click **Save** to save the upgraded configuration file in JSON format. You can now use the upgraded configuration to review your models in the newer version of MATLAB.

View Updated and Deleted Checks in Validation Summary

After you upgrade an old configuration file to verify your models in a newer version of MATLAB, Model Advisor generates a Validation Summary. The Validation Summary window lists the updated and the deleted checks.

This table explains the sections in the Validation Summary window.

Icon	Section
	<p>Checks with updated input parameters — The checks that are altered to include input parameters which are introduced or updated in the newer version of MATLAB.</p> <p>Checks with updated Check IDs — The checks that are altered to include check IDs which are introduced or updated in the newer version of MATLAB.</p>
	<p>Checks which are deleted — The checks that are no longer supported in the newer version of MATLAB.</p>

Use the Model Advisor Configuration Editor to Create a Custom Model Advisor Configuration

You can use the Model Advisor Configuration Editor to organize the hierarchy of the Model Advisor and specify checks that are included in check analyses. This example shows how to create a new configuration file, specify checks for the Model Advisor and edit-time checking, define check parameters, and load the configuration to the Model Advisor.

Create a Model Advisor Configuration

In this example, you will create a custom configuration file named `custom_Configuration.json`. This configuration will consist of MathWorks Advisory Board (MAB) modeling guidelines checks and industry standard checks that you want to execute by using the Model Advisor.

1. Open the Model Advisor Configuration editor by entering this command at the command prompt:

```
Simulink.ModelAdvisor.openConfigUI
```

2. In the toolstrip, select **Show Library** to display the **Library** pane. In the **By Product** tab search field, enter **ISO 26262**.

3. Right-click on the **Simulink Check > Modeling Standards > IEC 61508, IEC 62304, ISO 26262, ISO 25119, EN 50128, and EN 50657 Checks** folder and select **Copy**. Right-click on the Model Advisor Configuration Editor root folder and click **Paste**. Verify that the folder and checks have been copied to the root folder.

Note: Checks that are copied from the **Library** pane retain their default parameter settings. When they are pasted into your custom configuration folder, the box beside each check is not selected.

4. In the **By Task** folder on the **Model Advisor** pane, right-click on the **Modeling Standards for MAB** folder and select **Cut**. Click on the **Model Advisor Configuration Editor** root folder and click **Paste**. The folder is removed from the **By Task** folder and is added as a new subfolder in the **Model Advisor Configuration Editor** root folder.

Note: Checks that are copied or cut from a folder in the **Model Advisor** pane retain their user-defined parameter settings. When a check is included in multiple folders, you can specify different parameters for each check individually.

5. Select the **IEC 61508, IEC 62304, ISO 26262, ISO 25119, EN 50128, and EN 50657 Checks** folder and use the **Move Down** button to change the position this folder in the hierarchy. The Model Advisor will execute the checks in the **Modeling Standards for MAB** folder first.

6. Select the **By Product** and **By Task** folders and select **Delete**.

7. In the **Model Advisor** pane, set the configuration focus option to *Edit-Time supported checks*. The *Edit-Time supported checks* option displays the checks in this configuration that support edit-time checking.

8. Click **Save As** and name the configuration file to `custom_Configuration`. Select **Yes** at the prompt to save the configuration as the default configuration. The file automatically saves in `.json` format.

Note: If you have previously designated a default configuration, you can use **Clear default configuration** setting to clear the flag that specifies the current default configuration file. Clicking the button does not modify the configuration that is currently displayed in the Model Advisor Configuration Editor.

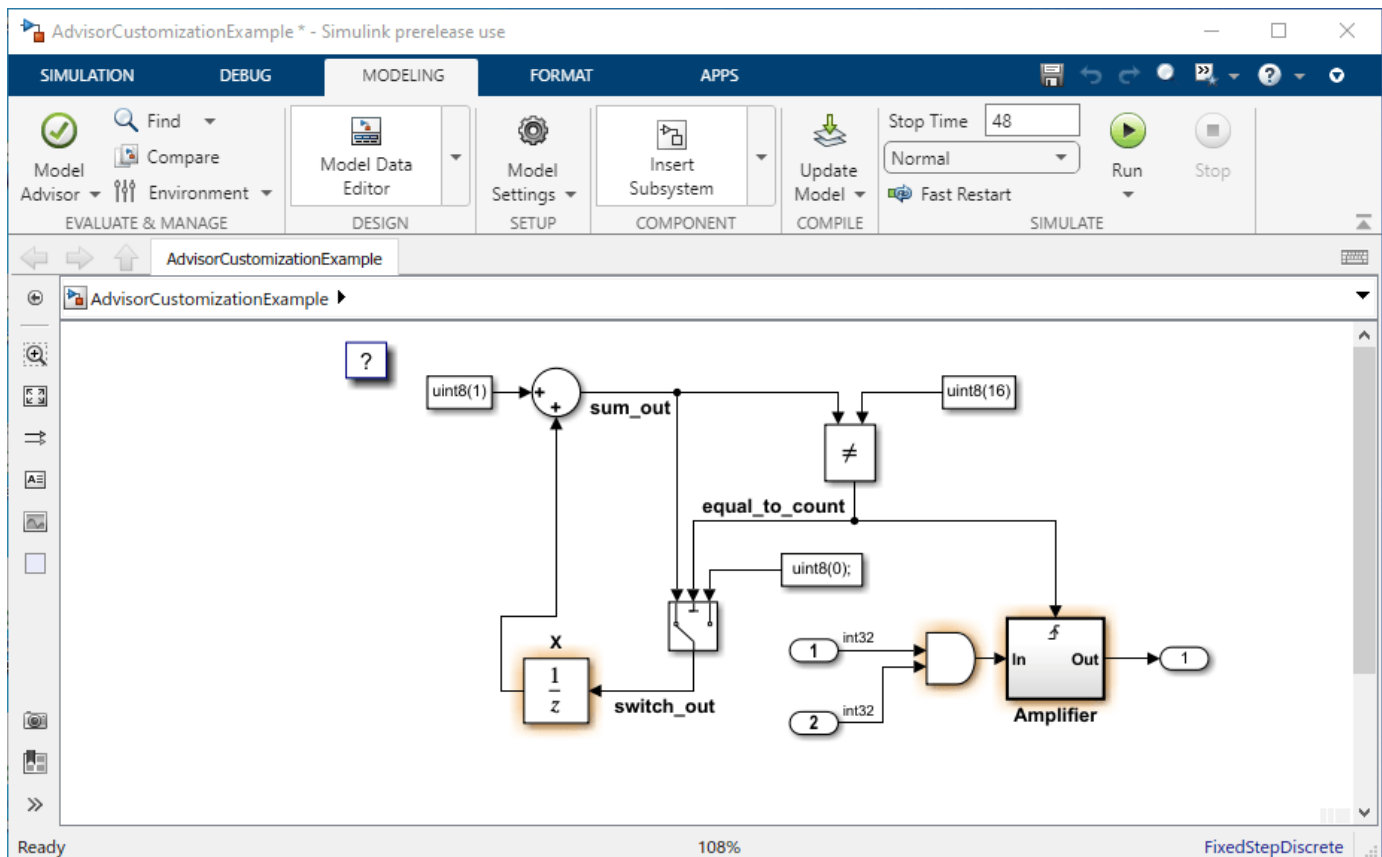
9. Close the Model Advisor Configuration Editor.

10. Open the `AdvisorCustomizationExample.slx` model by entering the following in the MATLAB command line:

```
open_system('AdvisorCustomizationExample.slx');
```

11. On the **Modeling** tab, click **Model Advisor > Edit-Time Checks**. In the Configuration Parameters dialog box, select **Edit-Time Checks** and **Apply**. Close the Configuration Parameters dialog box.

12. In the model, notice that three blocks are highlighted. These blocks contain edit-time check violations for this configuration. Place your cursor over a warning and click the block to discover the issue.



13. Open the Model Advisor and confirm that the Model Advisor displays the folders **Modeling Standards for MAB** and **IEC 61508, IEC 62304, ISO 26262, ISO 25119, EN 50128, and EN 50657 Checks**.

14. Close the Model Advisor.

Update a Model Advisor Configuration

You will now customize the checks in your custom configuration file, `custom_configuration.json` file and review the effect that your customizations have on the Model Advisor analysis of the `AdvisorCustomizationExample.slx` model.

1. On the **Modeling** tab, click **Model Advisor > Configuration Editor**.

2. Clear the check box the box beside the **Model Advisor Configuration Editor** root node folder. (This step is optional. However, deselecting the checks allows you to more easily view the results of using the Model Advisor Configuration Editor to specify checks for display in the Model Advisor.)
 3. To allow you to use the Model Advisor to specify which checks to include in the Model Advisor analysis, right-click on the **Model Advisor Configuration Editor** root folder and click **Enable**. (Note: Enable is the default setting. This option is dimmed when none of the checks are disabled.)
 4. Check the box beside the **Modeling Standards for MAB > Naming Conventions > Content > Check character usage in block name** check.
 5. Right-click on each of these checks and select **Disable**:
 - **Modeling Standards for MAB > Naming Conventions > Content > Check character usage in block names**
 - **Modeling Standards for MAB > Naming Conventions > Content > Check length of subsystem name**
 6. Click the **Modeling Standards for MAB > Simulink > Diagram Appearance > Check whether block names appear below blocks** check and, in the **Information** tab, select **Fail** for the **Check result when issues are flagged** option. Click **Apply**.
- Note:** The default for the **Check result when issues are flagged** option is **Warning**.
7. Click **Save** to save the configuration. Close the Model Advisor Configuration Editor and the model.
 8. Refresh the Model Advisor cache and open model `AdvisorCustomizationExample.slx` by entering the following in the MATLAB command line:


```
Advisor.Manager.refresh_customizations();
open_system('AdvisorCustomizationExample.slx');
```
 9. Open the Model Advisor.
- Observe these checks, which reflect the settings that you chose in the Model Advisor Configuration Editor:
- The **Modeling Standards for MAB > Naming Conventions > Content > Check character usage in block names check** is dimmed and the check box is selected. This check will always execute in a Model Advisor analysis and, because it is dimmed, you cannot choose to exclude it from the analysis.
 - The **Modeling Standards for MAB > Naming Conventions > Content > Check length of subsystem names check** is dimmed and the check box is not selected. This check will not be included in the analysis and, because it is dimmed, you cannot select it for inclusion in the analysis.
10. Check the box beside the **Modeling Standards for MAB > Simulink > Diagram Appearance > Check whether block names appear below blocks** check.
 11. To run the Model Advisor analysis, right-click on the **Model Advisor Standards for MAB** root node and select **Run Selected Checks**.
 12. Click on the following checks and review the Model Advisor analysis results:

- The **Modeling Standards for MAB > Naming Conventions > Content > Check character usage in block names** check is marked with a warning icon and the results specify the check violation is in the Gain block.
- There are no results for the **Modeling Standards for MAB > Naming Conventions > Content > Check length of subsystem** check because it could not be selected for the analysis.
- The **Modeling Standards for MAB > Simulink > Diagram Appearance > Check whether block names appear below blocks** check is marked with a fail icon. This behavior is intended; you specified this check settings by using the Model Advisor Configuration Editor.

See Also

`ModelAdvisor.setDefaultConfiguration` | `ModelAdvisor.Check`

More About

- “Define Custom Model Advisor Checks” on page 6-45
- “Customize the Configuration of the Model Advisor Overview” on page 7-2
- “Update the Environment to Include Your Custom Configuration” on page 7-20
- “Load and Associate a Custom Configuration with a Model” on page 7-21

Programmatically Customize Tasks and Folders for the Model Advisor

Customization File Overview

The `sl_customization.m` file contains a set of functions for registering and defining custom checks, tasks, and groups. To set up the `sl_customization.m` file, follow the guidelines in this table.

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

Function	Description	Required or Optional
<code>sl_customization()</code>	Registers custom checks and tasks, folders with the Simulink customization manager at startup. See “Define Custom Model Advisor Checks” on page 6-45.	Required for customizations to the Model Advisor.
One or more check definitions	Defines custom checks. See “Define Custom Model Advisor Checks” on page 6-45.	Required for custom checks and to add custom checks to the By Product folder.
One or more task definitions	Defines custom tasks. See “Define Custom Tasks” on page 7-14.	Required to add custom checks to the Model Advisor, except when adding the checks to the By Product folder. Write one task for each check that you add to the Model Advisor.
One or more groups	Defines custom groups. See “Define Custom Tasks” on page 7-14.	Required to add custom tasks to new folders in the Model Advisor, except when adding a new subfolder to the By Product folder. Write one group definition for each new folder.

Register Tasks and Folders

Create `sl_customization` Function

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

Tip

- You can have more than one `sl_customization.m` file on your MATLAB path.

- Do not place an `sl_customization.m` file that customizes the Model Advisor in your root MATLAB folder or its subfolders, except for the `matlabroot/work` folder. Otherwise, the Model Advisor ignores the customizations that the file specifies.
-

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

```
function sl_customization(cm)
```

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

Register Tasks and Folders

The customization manager provides the following methods for registering custom tasks and folders:

- `addModelAdvisorTaskFcn (@factorygroupDefinitionFcn)`

Registers the tasks that you define in `factorygroupDefinitionFcn` to the **By Task** folder of the Model Advisor.

The `factorygroupDefinitionFcn` argument is a handle to the function that defines the checks to add to the Model Advisor as instances of the `ModelAdvisor.FactoryGroup` class.

- `addModelAdvisorTaskAdvisorFcn (@taskDefinitionFcn)`

Registers the tasks and folders that you define in `taskDefinitionFcn` to the folder in the Model Advisor that you specify using the `ModelAdvisor.Root.publish` method or the `ModelAdvisor.Group` class.

The `taskDefinitionFcn` argument is a handle to the function that defines custom tasks and folders. Simulink adds the checks and folders to the Model Advisor as instances of the `ModelAdvisor.Task` or `ModelAdvisor.Group` classes.

The following example shows how to register custom tasks and folders:

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

Define Custom Tasks

Add Check to Custom or Multiple Folders Using Tasks

You can use custom tasks for adding checks to the Model Advisor, either in multiple folders or in a single, custom folder. You define custom tasks in one or more functions that specify the properties of each instance of the `ModelAdvisor.Task` class. Define one instance of this class for each custom task that you want to add to the Model Advisor. Then register the custom task. The following sections describe how to define custom tasks.

To add a check to multiple folders or a single, custom folder:

- 1 Create a check using the `ModelAdvisor.Check` class.
- 2 Register a task wrapper for the check.
- 3 If you want to add the check to folders that are not already present, register and create the folders using the `ModelAdvisor.Group` class.
- 4 Add a check to the task using the `ModelAdvisor.Task.setCheck` method.
- 5 Add the task to each folder using the `ModelAdvisor.Group.addTask` method and the task ID.

Create Custom Tasks Using MathWorks Checks

You can add MathWorks checks to your custom folders by defining the checks as custom tasks. When you add the checks as custom tasks, you identify checks by the check ID.

To find MathWorks check IDs:

- 1 In the hierarchy, navigate to the folder that contains the MathWorks check.
- 2 In the left pane of the Model Advisor, select the check.
- 3 Right-click the check name and select **Send Check ID to Workspace**. The ID is displayed in the Command Window and sent to the base workspace.
- 4 Select and copy the **Check ID** of the check that you want to add from the **Command Window** as a task.

Display and Enable Tasks

The `Visible`, `Enable`, and `Value` properties interact the same way for tasks as they do for checks.

Define Where Tasks Appear

You can specify where the Model Advisor places tasks within the Model Advisor using the following guidelines:

- To place a task in a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class.
- To place a task in a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class.

Task Definition Function

The following example shows a task definition function. This function defines three tasks.

Define Custom Folders

About Custom Folders

Use folders to group checks in the Model Advisor by functionality or usage. You define custom folders in:

- A factory group definition function that specifies the properties of each instance of the `ModelAdvisor.FactoryGroup` class.
- A task definition function that specifies the properties of each instance of the `ModelAdvisor.Group` class.

Define one instance of the group classes for each folder that you want to add to the Model Advisor.

Add Custom Folders

To add a custom folder:

- 1 Create the folder using the `ModelAdvisor.Group` or `ModelAdvisor.FactoryGroup` classes.
- 2 Register the folder.

Define Where Custom Folders Appear

You can specify the location of custom folders within the Model Advisor using the following guidelines:

- To define a new folder in the **Model Advisor Task Manager**, use the `ModelAdvisor.Group` class.
- To define a new folder in the **By Task** folder, use the `ModelAdvisor.FactoryGroup` class.

Note To define a new folder in the **By Product** folder, use the `ModelAdvisor.Root.publish` method within a custom check. If the **By Product** folder is not displayed in the Model Advisor window, select **Show By Product Folder** from the **Settings > Preferences** dialog box.

Group Definition

The following examples shows a group definition. The definition places the tasks inside a folder called **My Group** under the **Model Advisor** root. The task definition function includes this group definition.

The following example shows a factory group definition function. The definition places the checks into a folder called **Demo Factory Group** inside of the **By Task** folder.

See Also

`ModelAdvisor.Check` | `ModelAdvisor.FactoryGroup` | `ModelAdvisor.Group` | `ModelAdvisor.Task` | `ModelAdvisor.Procedure` | `publish`

More About

- “Define Custom Model Advisor Checks” on page 6-45
- “Customize the Configuration of the Model Advisor Overview” on page 7-2

Programmatically Create Procedural-Based Configurations

You can create a procedural-based configuration that allows you to specify the order in which you make changes to your model. You organize checks into procedures using the procedures API. A check in a procedure does not run until the previous check passes. A procedural-based configuration runs until a check fails, requiring you to modify the model to pass the check and proceed to the next check. Changes you make to your model to pass the checks therefore follow a specific order.

To create a procedural-based configuration, perform the following tasks:

- 1 Review the information in “Customize the Configuration of the Model Advisor Overview” on page 7-2.
- 2 Decide on order of changes to your model.
- 3 Identify checks that provide information about the modifications you want to make to your model. For example, if you want to modify your model optimization settings, the Check optimization settings check provides information about the settings. You can use custom checks and checks provided by MathWorks.
- 4 (Optional) Author custom checks in a customization file. See “Create Model Advisor Checks”.
- 5 Organize the checks into procedures for a procedural-based configuration:
 - a Create procedures by using the procedure API. For detailed information, see “Create Procedural-Based Configurations” on page 7-17.
 - b Create the custom configuration “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.
- 6 (Optional) Deploy the custom configurations to your users. For detailed information, see “Deploy Custom Configurations” on page 7-23.
- 7 Verify that models comply with modeling guidelines. For detailed information, see “Run Model Advisor Checks and Review Results” on page 3-4.

Create Procedural-Based Configurations

Create Procedures Using the Procedures API

You create procedures with the `ModelAdvisor.Procedure` class API. You first add the checks to tasks, which are wrappers for the checks. The tasks are added to procedures.

Note When creating procedural checks, be aware of potential conflicts with the checks. Verify that it is possible to pass both checks.

You use the `ModelAdvisor.Procedure` class to create procedural checks.

- 1 Add each check to a task using the `ModelAdvisor.Task.setCheck` method. The task is a wrapper for the check. You cannot add checks directly to procedures.
- 2 Add each task to a procedure using the `ModelAdvisor.Procedure.addTask` method.

Define Procedures

You define procedures in a procedure definition function that specifies the properties of each instance of the `ModelAdvisor.Procedure` class. Define one instance of the procedure class for each

procedure that you want to add to the Model Advisor. Then register the procedure using the `ModelAdvisor.Root.register` method.

You can add subprocedures or tasks to a procedure. The tasks are wrappers for checks.

- Use the `ModelAdvisor.Procedure.addProcedure` method to add a subprocedure to a procedure.
- Use the `ModelAdvisor.Procedure.addTask` method to add a task to a procedure.

The following code example adds subprocedures to a procedure:

```
%Create a procedure
MAP = ModelAdvisor.Procedure('com.mathworks.example.Procedure');

%Create 3 sub procedures
MAP1=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub1');
MAP2=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub2');
MAP3=ModelAdvisor.Procedure('com.mathworks.example.procedure_sub3');

%Add sub procedures to procedure
addProcedure(MAP, MAP1);
addProcedure(MAP, MAP2);
addProcedure(MAP, MAP3);

%register the procedures
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(MAP);
mdladvRoot.register(MAP1);
mdladvRoot.register(MAP2);
mdladvRoot.register(MAP3);
```

The following code example adds tasks to a procedure:

```
%Create three tasks
MAT1=ModelAdvisor.Task('com.mathworks.tasksample.myTask1');
MAT2=ModelAdvisor.Task('com.mathworks.tasksample.myTask2');
MAT3=ModelAdvisor.Task('com.mathworks.tasksample.myTask3');

%Create a procedure
MAP = ModelAdvisor.Procedure('com.mathworks.tasksample.myProcedure');

%Add the three tasks to the procedure
addTask(MAP, MAT1);
addTask(MAP, MAT2);
addTask(MAP, MAT3);

%register the procedure and tasks
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(MAP);
mdladvRoot.register(MAT1);
mdladvRoot.register(MAT2);
mdladvRoot.register(MAT3);
```

You can specify where the Model Advisor places a procedure using the `ModelAdvisor.Group.addProcedure` method. The following code example adds procedures to a group:


```
%Create three procedures
MAP1=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure1');
MAP2=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure2');
MAP3=ModelAdvisor.Procedure('com.mathworks.sample.myProcedure3');

%Create a group
MAG = ModelAdvisor.Group('com.mathworks.sample.myGroup');

%Add the three procedures to the group
addProcedure(MAG, MAP1);
addProcedure(MAG, MAP2);
addProcedure(MAG, MAP3);

%register the group and procedures
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.register(MAG);
mdladvRoot.register(MAP1);
mdladvRoot.register(MAP2);
mdladvRoot.register(MAP3);
```

See Also

[ModelAdvisor.Check](#) | [ModelAdvisor.Procedure](#)

More About

- “Define Custom Model Advisor Checks” on page 6-45
- “Customize the Configuration of the Model Advisor Overview” on page 7-2

Update the Environment to Include Your Custom Configuration

To make custom configuration available for use by the Model Advisor, you need to first update your Simulink environment to refresh the Model Advisor cache. This includes the creation of new or modifications to existing:

- `.json` files by using the Model Advisor Configuration Editor. See “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.
- `sl_customization.m` files for custom Model Advisor checks. See “Create Model Advisor Checks”.

To update your environment:

- 1** If you previously started the Model Advisor:
 - a** Close the model from which you started the Model Advisor
 - b** Clear the data associated with the previous Model Advisor session by removing the `slprj` folder from your “Code generation folder”.
- 2** In the MATLAB command line, enter:
`Advisor.Manager.refresh_customizations`
- 3** Open your model
- 4** In the **Modeling** tab, select **Model Advisor** to open the Model Advisor. If you have customized the configuration by using the Model Advisor Configuration Editor, load and verify the configuration as described in “Load and Associate a Custom Configuration with a Model” on page 7-21.

Load and Associate a Custom Configuration with a Model

Custom configurations allow you to specify which checks run during Model Advisor analysis. When you load a custom configuration, the Model Advisor uses the folders and checks specified by the configuration. You can also associate a custom configuration with your model so that the Model Advisor uses that configuration each time you open that model.

For example, to create a custom configuration and associate the configuration with a model `newModel`:

- 1 Set your current folder to a writeable directory.
- 2 Create and save a new model called `newModel`. In the MATLAB Command Window, enter:

```
new_system("newModel");
save_system("newModel");
```

- 3 Open the Model Advisor by entering:

```
modeladvisor("newModel")
```

- 4 In the Model Advisor, click **Open > Open Configuration Editor** to open the Model Advisor Configuration Editor.

You can use the Model Advisor Configuration Editor to specify the folders and checks that you want to include in the Model Advisor. For information on how to create a custom configuration, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.

- 5 In the Model Advisor Configuration Editor, click **Save As** to save a new, custom configuration file. For this example, save the configuration file as `customConfig.json`. The Model Advisor Configuration Editor prompts you to save the configuration as the default configuration. For this example, click **No**.
- 6 Update your Simulink environment to include the custom configuration file. At the MATLAB command line, enter:

```
Advisor.Manager.refresh_customizations
```

For more information, see “Update the Environment to Include Your Custom Configuration” on page 7-20.

- 7 Re-open the Model Advisor for the model `newModel`. In the MATLAB Command Window, enter:

```
modeladvisor("newModel")
```

- 8 In the Model Advisor, click **Open > Load Configuration** and select the custom configuration file `customConfig.json` to load the configuration file into the Model Advisor.

If the configuration file that you loaded contains checks that are incompatible with the newer version of MATLAB you use, you get a dialog box asking whether to automatically fix issues in the configuration. To fix the issues automatically, click **Yes**, and then perform the steps as described in “Update or Delete Checks in Configuration Files Automatically Using Model Advisor” on page 7-7.

- 9 In the Model Advisor Check Selector pane, you can verify that you see the folders and checks specified in the custom configuration. If you expect a folder or check to appear in the Model Advisor and it does not, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3 and “Update the Environment to Include Your Custom Configuration” on page 7-20.
- 10 If you want the Model Advisor to use this configuration each time you open the model `newModel`, associate the loaded configuration file with the model by using one of these approaches:

- In the Model Advisor, click **Open > Associate Configuration to Model**. The Model Advisor opens the Configuration Parameters dialog box for the model. The **Model Advisor configuration file** parameter lists the configuration file associated with the model. Click **OK**.
- In the MATLAB Command Window, provide the model name and configuration file name as inputs to the function `ModelAdvisor.setModelConfiguration`.

```
ModelAdvisor.setModelConfiguration("newModel", "customConfig.json");
```

- 11** You can view the Model Advisor configuration file associated with the model by entering:

```
ModelAdvisor.getModelConfiguration("newModel")
```

- 12** To return to your default configuration, click **Open > Restore Default Configuration**.

For information on how to restore the default shipping configuration or set a new default configuration, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.

Note The Model Advisor looks for configuration files in the following order:

- 1** The configuration that you specify by using the `configfile` input argument of the `modeladvisor` function.
- 2** The configuration associated with the model.
- 3** The default configuration. The *default configuration* is either the default shipping configuration or the default configuration that you set. For more information on the default configuration, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor” on page 7-3.

The Model Advisor loads the first configuration file that it finds and ignores other configuration files.

Deploy Custom Configurations

When you create a custom configuration, often you *deploy* the custom configuration to your development group. Deploying the custom configuration allows your development group to review models using the same checks. You can deploy custom configurations whether you created the configuration using the Model Advisor Configuration Editor or within the customization file.

To deploy a custom configuration:

- 1 Determine which files to distribute. You might need to distribute more than one file.

If You...	Using the...	Distribute...
Created custom Model Advisor checks	Customization file	<ul style="list-style-type: none"> • <code>sl_customization.m</code> • Files containing check and action callback functions (if separate)
Created custom Model Advisor configuration files	Model Advisor Configuration Editor	Configuration <code>.json</code> file

- 1 Distribute the files and tell the user to include these files on the MATLAB path.
- 2 Instruct the user to load the custom configuration.

Create and Deploy a Model Advisor Custom Configuration

To check that a model meets the standards and modeling guidelines of your company, you can customize the Model Advisor. This example shows you how to add custom checks to the Model Advisor and remove shipping checks that you do not require. You can save the custom check configuration and deploy it to others in your development group. Deploying a custom configuration allows your development group to review models using the same set of checks.

Define Custom Checks

This example defines four types of custom checks:

- An edit-time check that provides a fix action.
- A check that runs in only the Model Advisor and groups results by blocks and subsystems and provides a fix action.
- A check that runs only in the Model Advisor and verifies model configuration parameter settings.
- An edit-time check that specifies a constraint for a block parameter setting and provides a fix action.

The example files include the `sl_customization.m` file. This file contains the `sl_customization` function, which contains calls to functions that define the custom checks. Open and inspect the `sl_customization.m` file.

```
function sl_customization(cm)
% SL_CUSTOMIZATION - Model Advisor customization demonstration.
% Copyright 2019 The MathWorks, Inc.
% register custom checks
cm.addModelAdvisorCheckFcn(@defineModelAdvisorChecks);
% -----
% defines Model Advisor Checks
% -----
function defineModelAdvisorChecks
defineEditTimeCheck
defineDetailStyleCheck;
defineConfigurationParameterCheck;
defineNewBlockConstraintCheck;
```

The `sl_customization` function accepts a customization manager object that includes the `addModelAdvisorCheckFcn` method for registering custom checks. The input to this method is a handle to a function, `defineModelAdvisorChecks`, which contains calls to the four check definition functions that correspond to the four custom checks.

Edit-Time Check with Fix

The `defineEditTimeCheck.m` file contains the `defineEditTimeCheck` check definition function, which defines a check that checks whether Inport and Outport blocks have certain colors depending on their output data types. This check must check other edited blocks, but it does not have to check for affected blocks at the same level or across the entire model hierarchy. This check provides a fix that updates the color of the blocks that do not have the correct colors. The name of this check is **Check color of Inport and Outport blocks**. This check runs at edit-time and in the Model Advisor. Open and inspect the `defineEditTimeCheck.m` file.

```
function defineEditTimeCheck

% Check the background color of Inport and Outport blocks.
rec = ModelAdvisor.Check("advisor.edittimecheck.PortColor");
rec.Title = 'Check color of Inport and Outport blocks';
rec.CallbackHandle = 'MyEditTimeChecks.PortColor';
mdladvRoot = ModelAdvisor.Root;
mdladvRoot.publish(rec, 'Demo');
```

The edit-time check has a class definition, `PortColor`, that derives from the `ModelAdvisor.EdittimeCheck` base class. For more information on how to create this type of check, see “Define Edit-Time Checks to Comply with Conditions That You Specify with the Model Advisor” on page 6-9. Create a folder named `+MyEditTimeChecks` and save `PortColor.m` to this folder.

```
copyfile PortColor.m* +MyEditTimeChecks
```

Open and inspect the `PortColor.m` file.

```
classdef PortColor < ModelAdvisor.EdittimeCheck
    % Check that ports conform to software design standards for background color.
    %
    % Background Color          Data Types
    % orange                    Boolean
    % green                      all floating-point
    % cyan                       all integers
    % Light Blue                Enumerations and Bus Objects
    % white                      auto
    %

    methods
        function obj=PortColor(checkId)
            obj=obj@ModelAdvisor.EdittimeCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.BLKITER;
        end

        function violation = blockDiscovered(obj, blk)
            violation = [];
            if strcmp(get_param(blk, 'BlockType'), 'Inport') || strcmp(get_param(blk, 'BlockType'),

                dataType = get_param(blk, 'OutDataTypeStr');
                currentBgColor = get_param(blk, 'BackgroundColor');

                if strcmp(dataType, 'boolean')
                    if ~strcmp(currentBgColor, 'orange')
                        % Create a violation object using the ModelAdvisor.ResultDetail class
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation, 'SID', Simulink.ID.getSID);
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with Boolean outputs';
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
                elseif any(strcmp({'single', 'double'}, dataType))
                    if ~strcmp(currentBgColor, 'green')
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation, 'SID', Simulink.ID.getSID);
```

```

        violation.CheckID = obj.checkId;
        violation.Description = 'Inport/Outport blocks with floating-point outputs should be integer';
        violation.title = 'Port Block Color';
        violation.ViolationType = 'Warning';
    end
elseif any(strcmp({'uint8','uint16','uint32','int8','int16','int32'}, dataType))
    if ~strcmp(currentBgColor, 'cyan')
        violation = ModelAdvisor.ResultDetail;
        ModelAdvisor.ResultDetail.setData(violation, 'SID', Simulink.ID.getSID(blk));
        violation.CheckID = obj.checkId;
        violation.Description = 'Inport/Outport blocks with integer outputs should be floating-point';
        violation.title = 'Port Block Color';
        violation.ViolationType = 'Warning';
    end
elseif contains(dataType, 'Bus:')
    if ~strcmp(currentBgColor, 'lightBlue')
        violation = ModelAdvisor.ResultDetail;
        ModelAdvisor.ResultDetail.setData(violation, 'SID', Simulink.ID.getSID(blk));
        violation.CheckID = obj.checkId;
        violation.Description = 'Inport/Outport blocks with bus outputs should be integer';
        violation.title = 'Port Block Color';
        violation.ViolationType = 'Warning';
    end
elseif contains(dataType, 'Enum:')
    if ~strcmp(currentBgColor, 'lightBlue')
        violation = ModelAdvisor.ResultDetail;
        ModelAdvisor.ResultDetail.setData(violation, 'SID', Simulink.ID.getSID(blk));
        violation.CheckID = obj.checkId;
        violation.Description = 'Inport/Outport blocks with enumeration outputs should be integer';
        violation.title = 'Port Block Color';
        violation.ViolationType = 'Warning';
    end
elseif contains(dataType, 'auto')
    if ~strcmp(currentBgColor, 'white')
        violation = ModelAdvisor.ResultDetail;
        ModelAdvisor.ResultDetail.setData(violation, 'SID', Simulink.ID.getSID(blk));
        violation.CheckID = obj.checkId;
        violation.Description = 'Inport/Outport blocks with auto outputs should be integer';
        violation.title = 'Port Block Color';
        violation.ViolationType = 'Warning';
    end
end
end
end

function violation = finishedTraversal(obj)
    violation = [];
end

function success = fix(obj, violation)
    success = true;
    dataType = get_param(violation.Data, 'OutDataTypeStr');
    if strcmp(dataType, 'boolean')
        set_param(violation.Data, 'BackgroundColor', 'orange');
    elseif any(strcmp({'single','double'}, dataType))
        set_param(violation.Data, 'BackgroundColor', 'green');
    elseif any(strcmp({'uint8','uint16','uint32','int8','int16','int32'}, dataType))
        set_param(violation.Data, 'BackgroundColor', 'cyan');
    end
end

```



```

elseif contains(dataType,'Bus:') || contains(dataType,'Enum:')
    set_param(violation.Data,'BackgroundColor','lightBlue');
elseif contains(dataType,'auto')
    set_param(violation.Data,'BackgroundColor','white');
end
end
end
end
end
end

```

Model Advisor Check with Fix

The `defineDetailStyleCheck.m` file contains the `defineDetailStyleCheck` check definition function, which defines a check that lists blocks whose names are not displayed below the blocks. This check provides a fix that moves those names below the blocks. The name of this check is **Check whether block names appear below blocks**. This check authoring style is for checks that only run in the Model Advisor. Open and inspect the `defineDetailStyleCheck.m` file.

```

function defineDetailStyleCheck

mdladvRoot = ModelAdvisor.Root;

% Create ModelAdvisor.Check object and set properties.
rec = ModelAdvisor.Check('com.mathworks.sample.detailStyle');
rec.Title = 'Check whether block names appear below blocks';
rec.TitleTips = 'Check position of block names';
rec.setCallbackFcn(@DetailStyleCallback,'None','DetailStyle');
% Create ModelAdvisor.Action object for setting fix operation.
myAction = ModelAdvisor.Action;
myAction.setCallbackFcn(@ActionCB);
myAction.Name='Make block names appear below blocks';
myAction.Description='Click the button to place block names below blocks';
rec.setAction(myAction);
mdladvRoot.publish(rec, 'Demo'); % publish check into Demo group.

end

% -----
% This callback function uses the DetailStyle CallbackStyle type.
% -----
function DetailStyleCallback(system, CheckObj)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system); % get object

% Find all blocks whose name does not appear below blocks
violationBlks = find_system(system, 'Type','block',...
    'NamePlacement','alternate',...
    'ShowName', 'on');
if isempty(violationBlks)
    ElementResults = ModelAdvisor.ResultDetail;
    ElementResults.IsInformer = true;
    ElementResults.Description = 'Identify blocks where the name is not displayed below the block.';
    ElementResults.Status = 'All blocks have names displayed below the block.';
    mdladvObj.setCheckResultStatus(true);
else
    for i=1:numel(violationBlks)
        ElementResults(1,i) = ModelAdvisor.ResultDetail;
    end
    for i=1:numel(ElementResults)

```

```

        ModelAdvisor.ResultDetail.setData(ElementResults(i), 'SID',violationBlks{i});
        ElementResults(i).Description = 'Identify blocks where the name is not displayed below the block.';
        ElementResults(i).Status = 'The following blocks have names that do not display below the block.';
        ElementResults(i).RecAction = 'Change the location such that the block name is below the block.';
    end
    mdladvObj.setCheckResultStatus(false);
    mdladvObj.setActionEnable(true);
end
CheckObj.setResultDetails(ElementResults);
end

% -----
% This action callback function changes the location of block names.
% -----
function result = ActionCB(taskobj)
mdladvObj = taskobj.MAObj;
checkObj = taskobj.Check;
resultDetailObjs = checkObj.ResultDetails;
for i=1:numel(resultDetailObjs)
    % take some action for each of them
    block=Simulink.ID.getHandle(resultDetailObjs(i).Data);
    set_param(block, 'NamePlacement', 'normal');
end

result = ModelAdvisor.Text('Changed the location such that the block name is below the block. ');
mdladvObj.setActionEnable(false);
end

```

This check uses the `setCallbackFcn` type of `DetailStyle`, which produces default formatting, so that you do not have to use the `ModelAdvisor.FormatTemplate` or the other Model Advisor formatting APIs to format the results that appear in the Model Advisor. For more information on how to create this type of check definition function, see “Fix a Model to Comply with Conditions that You Specify with the Model Advisor” on page 6-21.

Model Configuration Parameter Settings Check

The `defineConfigurationParameterCheck.m` file contains the `defineConfigurationParameterCheck` check definition function, which defines a check that identifies model configuration parameter settings that might impact MISRA C:2012 compliant code generation. The name of this check is **Check model configuration parameters**.

This check requires a supporting XML data file that must be on the MATLAB path and contain the model configuration parameter settings that you want to check. For this example, that file is `configurationParameterDataFile.xml`. For more information on how to create this check type, see “Create Model Advisor Check for Model Configuration Parameters” on page 6-27.

Open and inspect the `defineConfigurationParameterCheck.m` file.

```

function defineConfigurationParameterCheck

% Create ModelAdvisor.Check object and set properties.
rec = ModelAdvisor.Check('com.mathworks.sample.configurationParameter');
rec.Title = 'Check model configuration parameters';
rec.setCallbackFcn(@(system) (Advisor.authoring.CustomCheck.checkCallback...
    (system)), 'None', 'StyleOne');
rec.TitleTips = 'Identify configuration parameters that might impact MISRA C:2012 compliant code

```

```

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

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

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

end

```

Block Parameter Constraint Check

The `defineNewBlockConstraintCheck.m` file contains the `defineNewBlockConstraintCheck` check definition function, which defines a check that identifies Logical Operator blocks that do not have a rectangular shape. The name of this check is **Check icon shape of Logical Operator blocks**.

A block parameter constraint check supports edit-time checking. For more information on this check type, see “Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters” on page 6-38.

Open and inspect the `defineNewBlockConstraintCheck.m` file.

```

function defineNewBlockConstraintCheck

rec = Advisor.authoring.createBlockConstraintCheck('com.mathworks.sample.blockConstraint',...
    'Constraints',@createBlockConstraints); % constraint creation is part of block constraint che
rec.Title = 'Check icon shape of Logical Operator blocks';
rec.TitleTips = 'Checks icon shape of Logical Operator blocks. Icon shape of Logical Operator sho

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

end

function constraints = createBlockConstraints()

% Create block constraints.
c1 = Advisor.authoring.PositiveBlockParameterConstraint;
c1.ID = 'ID_c1';
c1.BlockType = 'Logic';
c1.ParameterName = 'IconShape';

```

```
c1.SupportedParameterValues = {'rectangular'};  
c1.ValueOperator = 'eq';
```

```
constraints = {c1};
```

```
end
```

The `createBlockConstraints` function defines the block constraint `c1`. The `Advisor.authoring.createBlockConstraintCheck` function has a 'Constraints' name-value argument that calls the constraints creation function `createBlockConstraints`.

View Custom Checks in the Model Advisor

To confirm that your custom checks are available, open the Model Advisor.

1. In order for your custom checks to be visible in the Model Advisor, you must refresh the Model Advisor check information cache. At the MATLAB command prompt, enter:

```
Advisor.Manager.refresh_customizations();
```

2. Open the example model.

```
open_system('AdvisorCustomizationExample.slx');
```

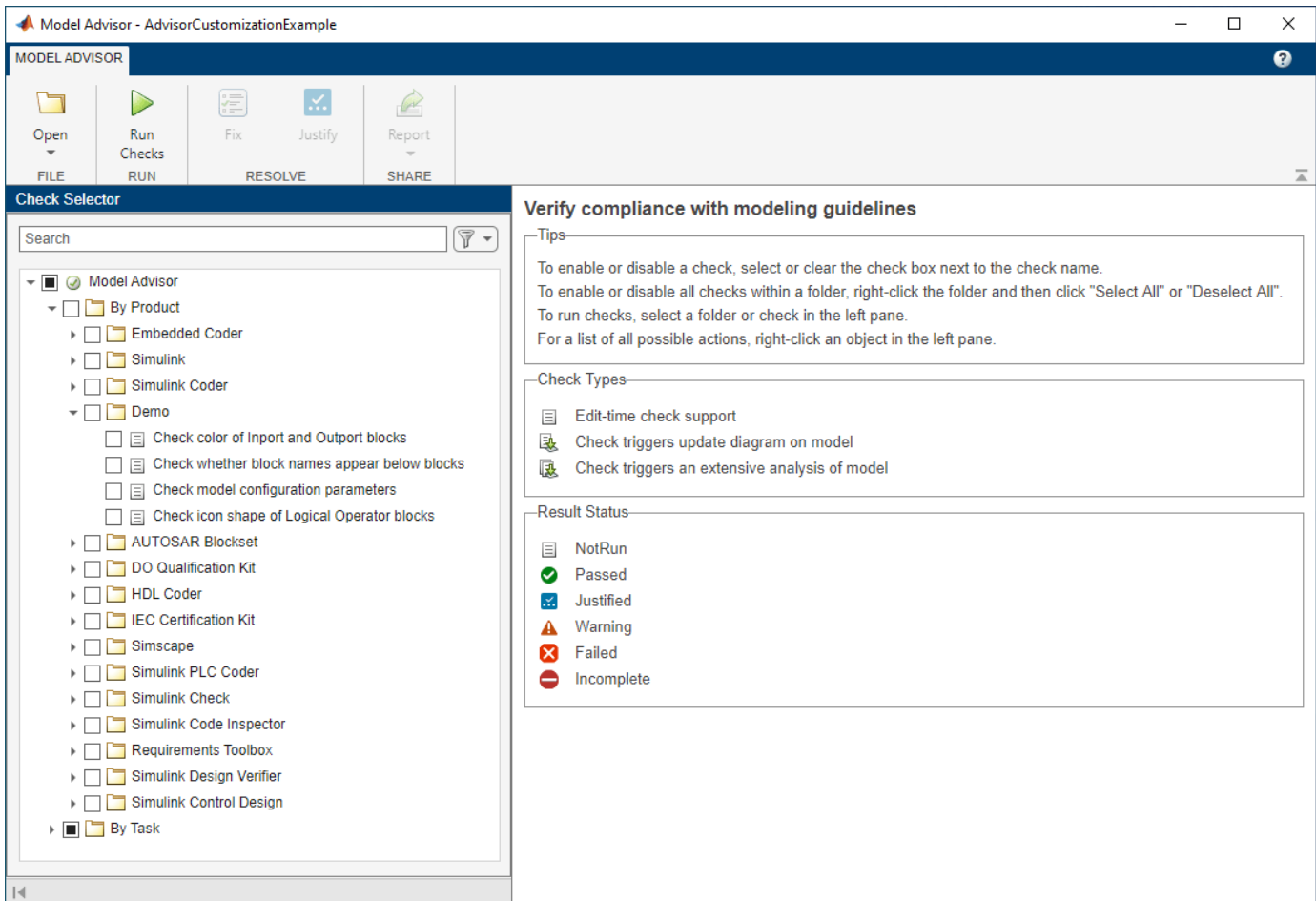
3. On the **Modeling** tab, open the **Model Advisor**. You can also open the Model Advisor by entering this command at the MATLAB command prompt:

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

```
Updating Model Advisor cache...
```

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

4. Expand the **By Product > Demo** folder. In the check definition functions, the `publish` command adds the checks to the **By Product > Demo** folder.



Specify and Deploy a Model Advisor Custom Configuration

To specify which checks to include in the Model Advisor and which checks to use during edit-time checking, use the Model Advisor Configuration Editor.

1. To open the Configuration Editor, in the Model Advisor, click **Open > Open Configuration Editor**.
2. To add or remove checks and folders, select from the options in the **Edit** section of the Model Advisor Configuration Editor.
3. To save a configuration, select **Save**. A window opens and prompts you to save the configuration as a JSON file. For this example, you do not have to save the configuration, because the file `demoConfiguration.json` file contains the four custom checks for this example.
4. Close the model and the Model Advisor Configuration Editor.

```
bdclose;
```

Associate a Custom Check Configuration with a Model and Address Check Issues

To address check issues, first associate the configuration with a model. Then, you can address issues during edit-time and in the Model Advisor.

1. Open the example model.

```
open_system('AdvisorCustomizationExample.slx');
```

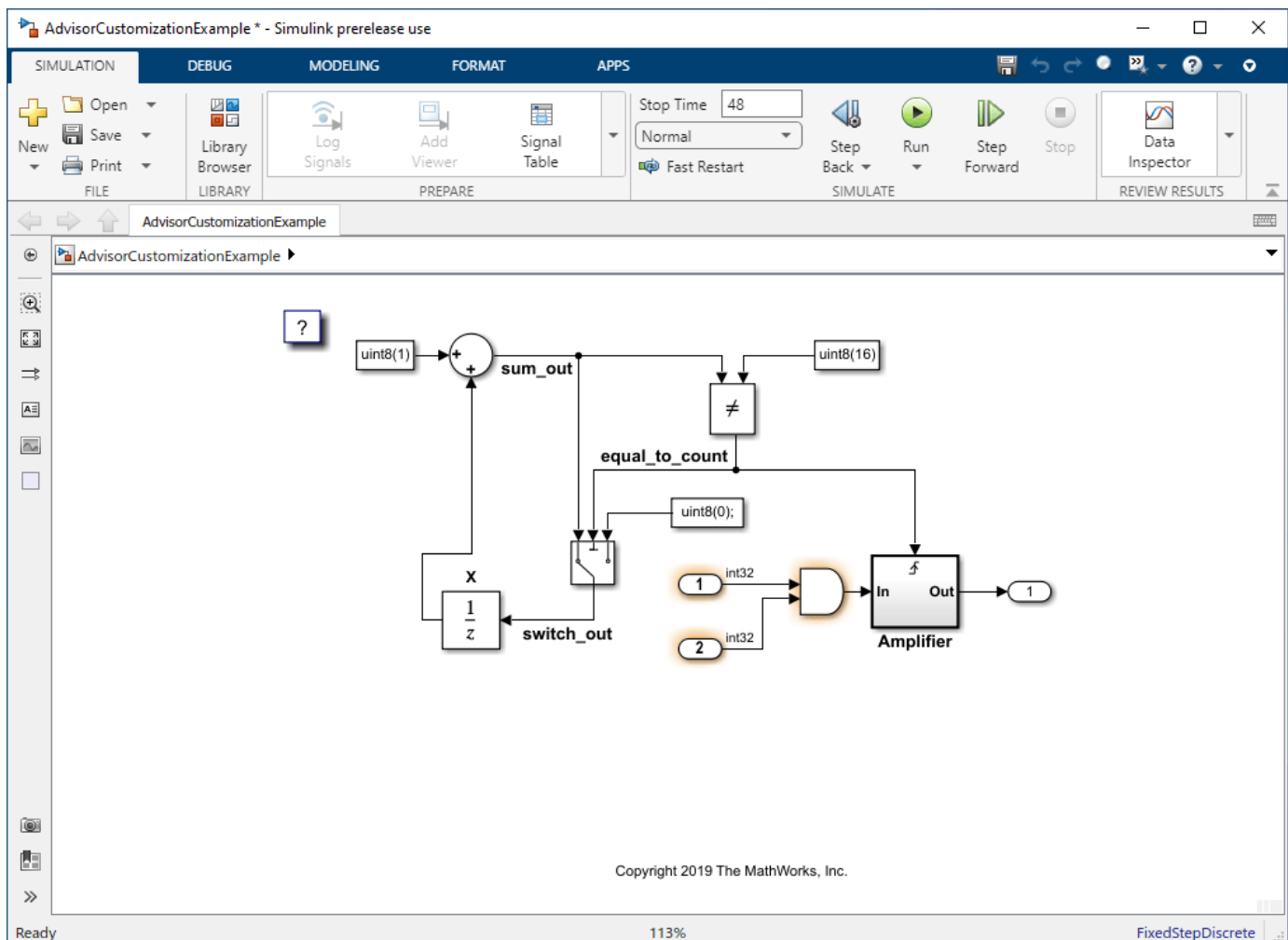
2. Associate the custom configuration, `demoConfiguration.json`, with the model. When you associate a custom configuration with a model, the model uses the same check configuration every time you open a model. Click the **Modeling** tab and select **Model Advisor > Edit-Time Checks**. In the Configuration Parameters dialog box, specify the path to the configuration file for the **Model Advisor** configuration file parameter. Alternatively, enter this command at the command prompt:

```
ModelAdvisor.setModelConfiguration('AdvisorCustomizationExample', 'demoConfiguration.json');
```

3. Turn on edit-time checking by clicking the **Modeling** tab and selecting **Model Advisor > Edit-Time Checks**. The Configuration Parameters dialog box opens. Select the **Edit-Time Checks** parameter. Alternatively, you can enter this command at the command prompt:

```
edittime.setAdvisorChecking('AdvisorCustomizationExample', 'on');
```

At the top level of the model, the two Inport blocks have an output data type of `int32`. The blocks produce edit-time warnings because they should be cyan. The Outport block does not produce a warning because it has an auto data type and is white.



4. For each Inport block, click the edit-time warning window. Then click **Fix**. The color of the blocks changes to cyan and the warning goes away.

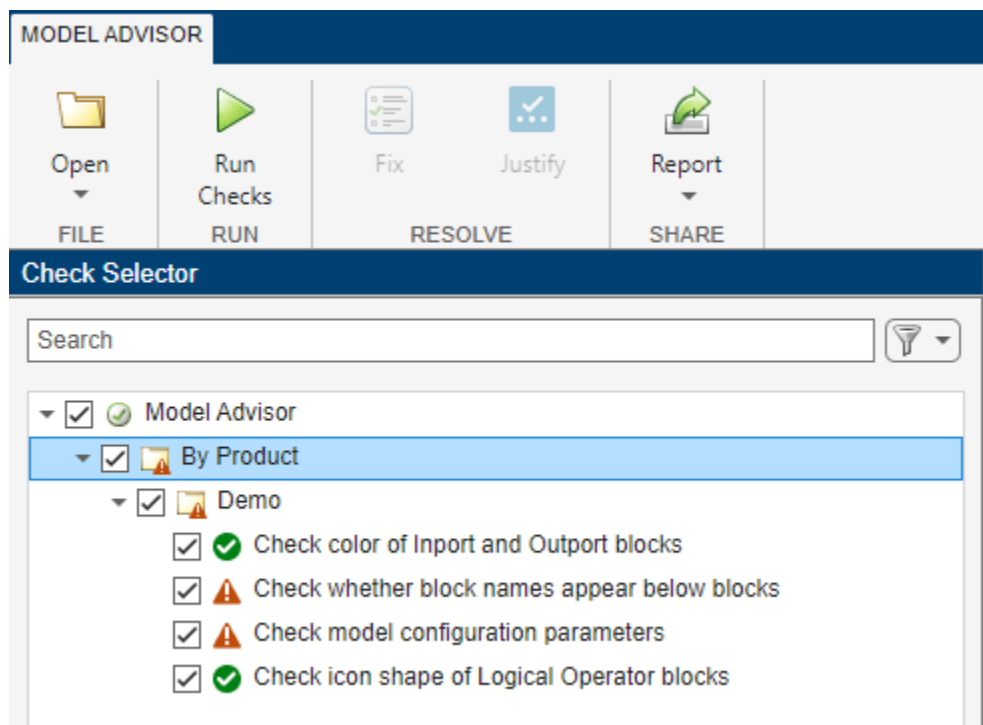
5. The Logical Operator block produces a warning because it should have a rectangular shape. Click the edit-time warning window and then click **Fix**. The shape of the Logical Operator block changes to a rectangle, and the warning goes away.

6. Now that you have addressed the edit-time check warnings, open the Model Advisor to address any remaining check issues.

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

Model Advisor is removing the existing report.

7. The Model Advisor contains the four checks in the custom configuration. Click **Run Checks**. The two checks that you addressed during edit-time pass. The other two checks produce warnings.



8. Click the **Check whether block names appear below blocks** check. To apply a fix and resolve the warnings, in the right pane, click **Fix**.

9. Click the **Check model configuration parameters** check. To apply a fix and resolve the warnings, click **Fix**.

10. Rerun the checks. They now pass.

11. Close the model and the Model Advisor.

```
bdclose;
```

12. Remove the files from your working directory. Refresh the Model Advisor check information cache by entering this command:

```
Advisor.Manager.refresh_customizations
```

Programmatically Run a Model Advisor Custom Configuration and View Results

You can programmatically run a Model Advisor configuration and then open the results in the Model Advisor.

1. Call the `ModelAdvisor.run` function.

```
SysResultObjArray = ModelAdvisor.run({'AdvisorCustomizationExample'}, ...  
'Configuration', 'demoConfiguration.json');
```

2. View the results in the Model Advisor:

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

3. Click **Continue** in the dialog box. You can now apply fixes and resolve warnings.

4. Close the model and the Model Advisor.

```
bdclose;
```

5. Remove the files from your working directory. Refresh the Model Advisor check information cache by entering this command:

```
Advisor.Manager.refresh_customizations
```

See Also

`ModelAdvisor.Check` | `ModelAdvisor.EditTimeCheck`

More About

- “Define Custom Model Advisor Checks” on page 6-45
- “Justify Model Advisor Violations from Check Analysis” on page 3-102
- “Define Custom Edit-Time Checks that Fix Issues in Architecture Models” on page 6-17
- “Run Custom Model Advisor Checks on Architecture Models” on page 3-99

Model Slicer

- “Highlight Functional Dependencies” on page 8-2
- “Highlight Dependencies for Multiple Instance Reference Models” on page 8-8
- “Refine Highlighted Model” on page 8-12
- “Refine Dead Logic for Dependency Analysis” on page 8-23
- “Create a Simplified Standalone Model” on page 8-29
- “Highlight Active Time Intervals by Using Activity-Based Time Slicing” on page 8-30
- “Simplify a Standalone Model by Inlining Content” on page 8-37
- “Workflow for Dependency Analysis” on page 8-39
- “Configure Model Highlight and Sliced Models” on page 8-41
- “Model Slicer Considerations and Limitations” on page 8-44
- “Using Model Slicer with Stateflow” on page 8-50
- “Isolating Dependencies of an Actuator Subsystem” on page 8-52
- “Isolate Model Components for Functional Testing” on page 8-56
- “Refine Highlighted Model by Using Existing .slsicex or Dead Logic Results” on page 8-63
- “Simplification of Variant Systems” on page 8-65
- “Programmatically Resolve Unexpected Behavior in a Model with Model Slicer” on page 8-67
- “Refine Highlighted Model Slice by Using Model Slicer Data Inspector” on page 8-75
- “Debug Slice Simulation by Using Fast Restart Mode” on page 8-82
- “Isolate Referenced Model for Functional Testing” on page 8-88
- “Analyze the Dead Logic” on page 8-92
- “Investigate Highlighted Model Slice by Using Model Slicer Data Inspector” on page 8-97
- “Programmatically Generate I/O Dependency Matrix” on page 8-103
- “Observe Impact of Simulink Parameters Using Model Slicer” on page 8-105
- “Analyze Models Containing Simulink Functions Using Model Slicer” on page 8-107

Highlight Functional Dependencies

Large models often contain many levels of hierarchy, complicated signals, and complex mode logic. You can use Model Slicer to understand which parts of your model are significant for a particular behavior. This example shows how to use Model Slicer to explore the behavior of the `sldvSliceClimateControlExample` model. You first select an area of interest, and then highlight the related blocks in the model. In this example, you trace the dependency paths upstream of `Out1` to highlight which portions of the model affect.

Open the model

Open the model and highlight the functional dependencies of a signal in the system.

1. Open the `sldvSliceClimateControlExample` model.

```
open_system("sldvSliceClimateControlExample");
```

2. To open the Model Slicer, on the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

When you open the Model Slicer, Model Slicer compiles the model. You then configure the model slice properties.

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

4. Set the slice properties:

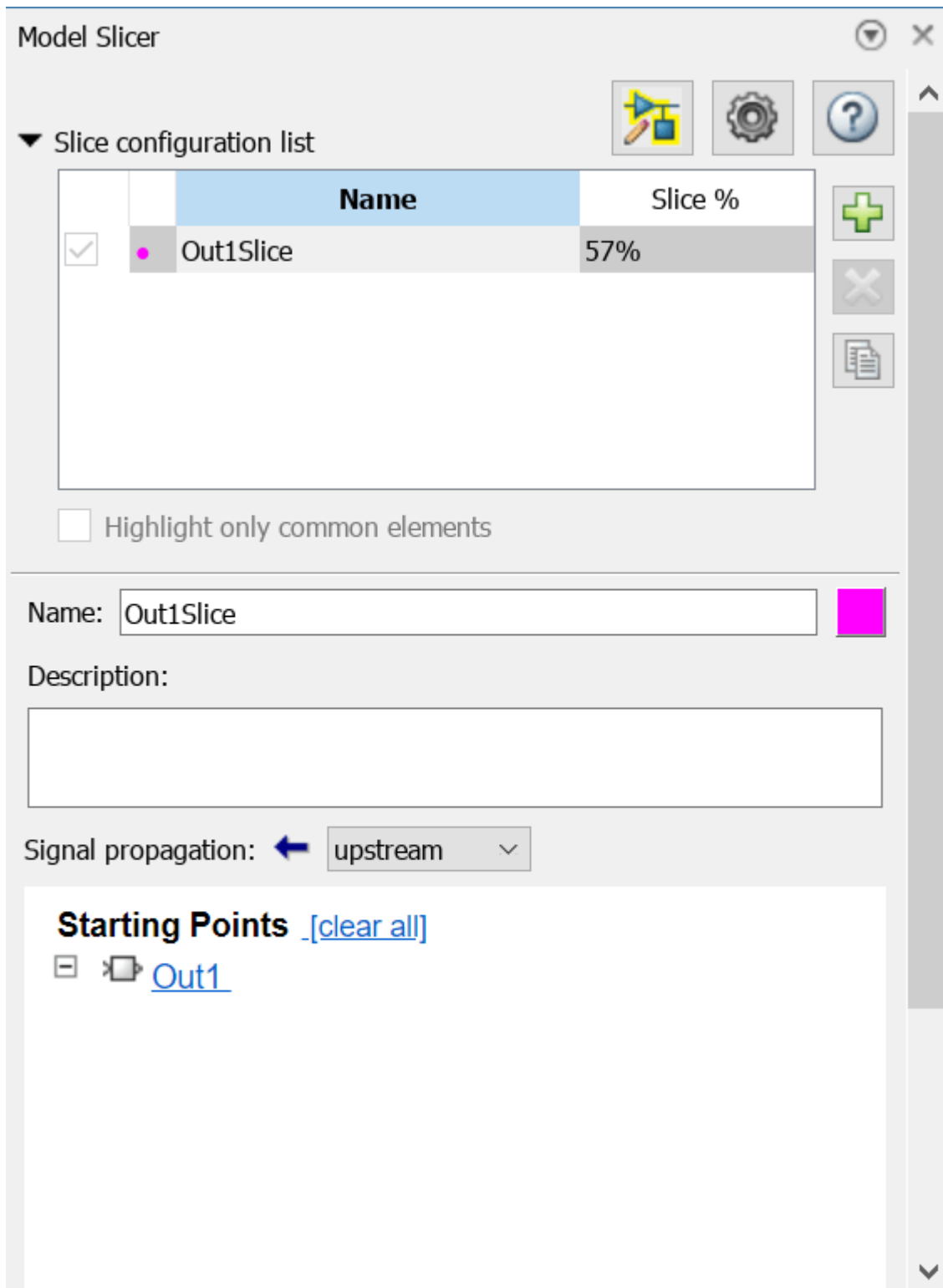
(a) **Name:** `Out1Slice`

(b) **Color:**  (magenta)

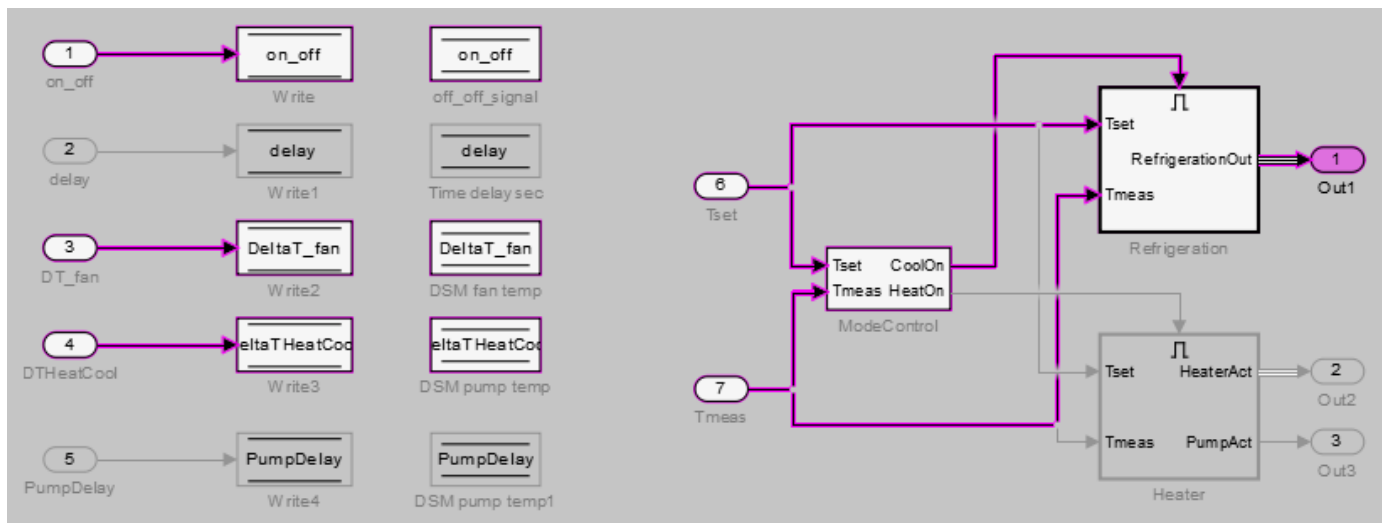
(c) **Signal Propagation:** `upstream`

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

5. Add `Out1` as a starting point. In the model, right-click `Out1` and select **Model Slicer > Add as Starting Point**.



The Model Slicer now highlights the upstream constructs that affect Out1.



If you create two slice configurations, you can highlight the intersecting portions of their highlights. Create a new slice configuration and view the intersecting portions of the slice configuration you created above and the new slice configuration:

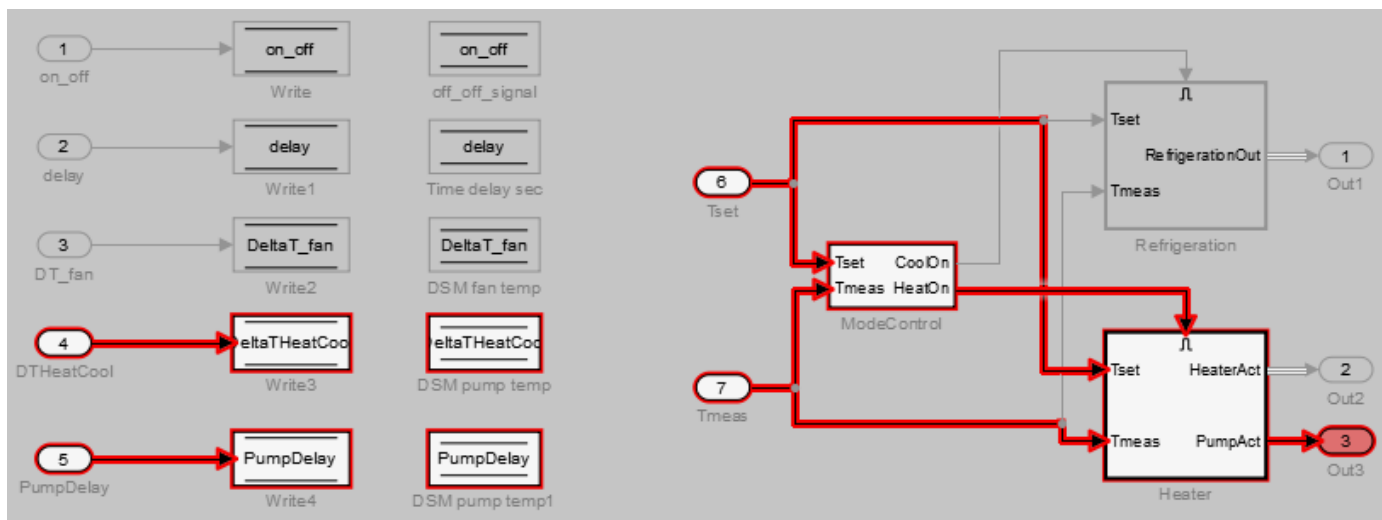
1. Create a new slice configuration with the following properties

(a) **Name:** Out3Slice

(b) **Color:** ■ (red)

(c) **Signal Propagation:** upstream

(d) **Starting Point:** Out3




2. In the Model Slicer, select both the Out1Slice slice configuration and the Out3Slice slice configuration.

Model Slicer


▼ Slice configuration list

	Name	Slice %
<input checked="" type="checkbox"/>	● Out1Slice	57%
<input checked="" type="checkbox"/>	● Out3Slice	34%



Highlight only common elements

Name: 

Description:

Signal propagation:  ▼

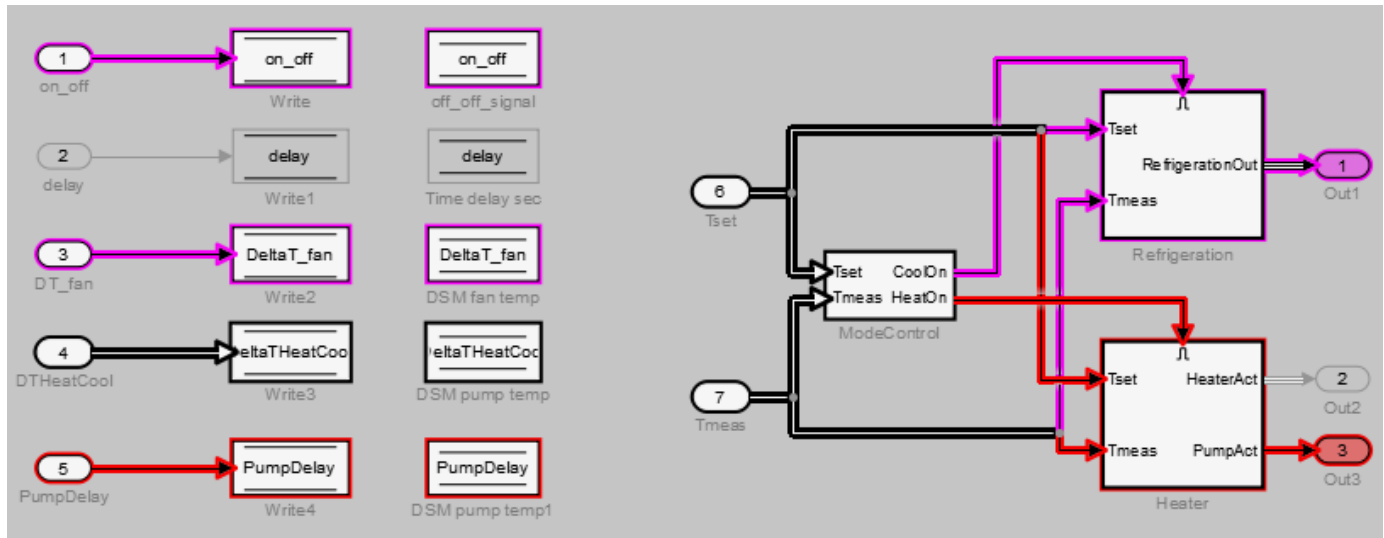
Starting Points [\[clear all\]](#)

  [Out3](#)

Model Slicer highlights portions of the model as follows:

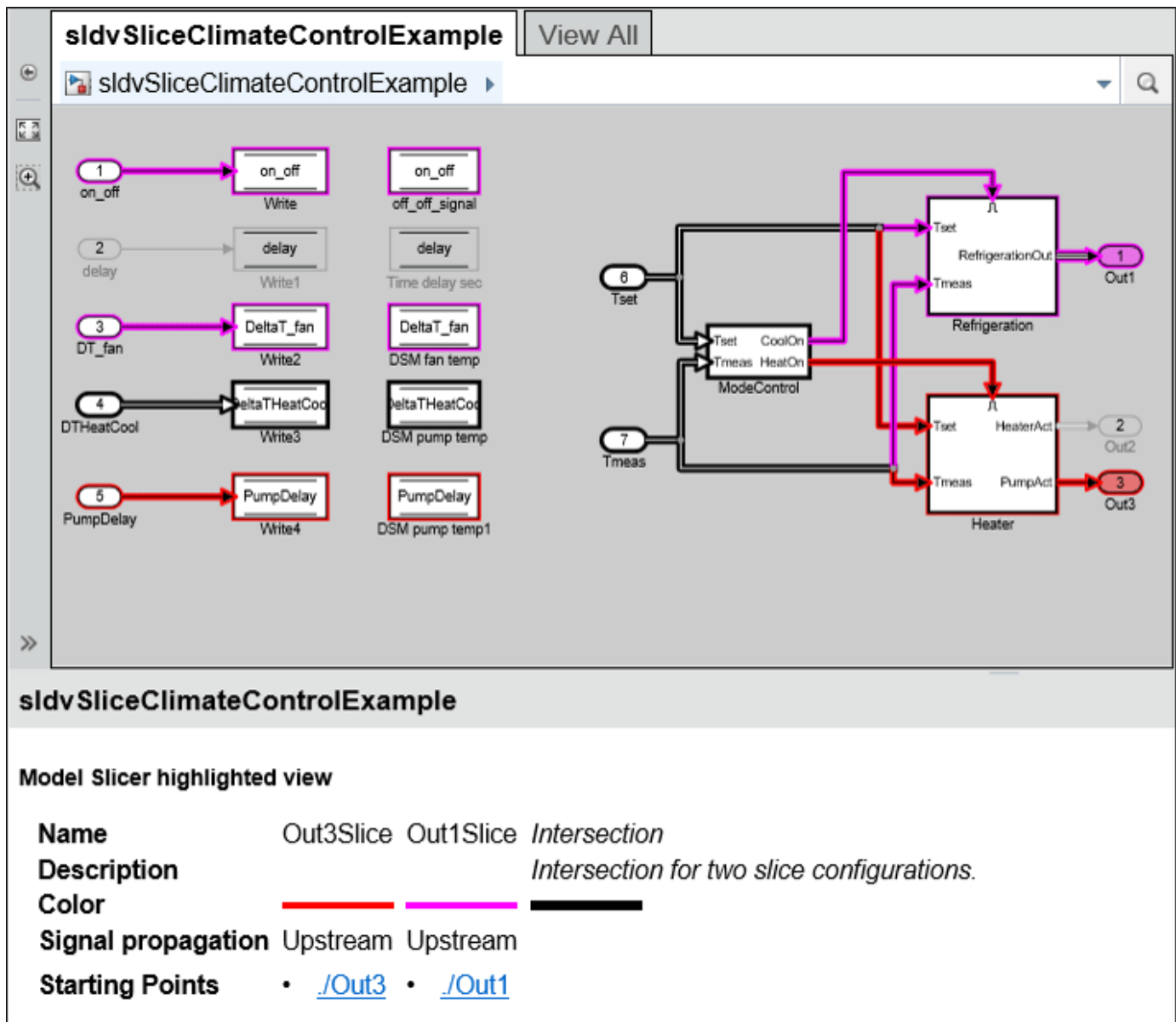
- The portions of the model that are exclusively upstream of Out1 are highlighted in cyan.

- The portions of the model that are exclusively upstream of Out3 are highlighted in red.
- The portions of the model that are upstream of both Out1 and Out3 are highlighted in black.



After you highlight a portion of your model, you can then refine the highlighted model to an area of interest. Or, you can create a simplified standalone model containing only the highlighted portion of your model.

To view the details of the highlighted model in web view, click **Export to Web** in the Model Slicer. The web view HTML file is stored in `<current folder>\<model_name>\webview.html`



Related Topics

- “Refine Highlighted Model” on page 8-12
- “Create a Simplified Standalone Model” on page 8-29
- “Model Slicer Considerations and Limitations” on page 8-44

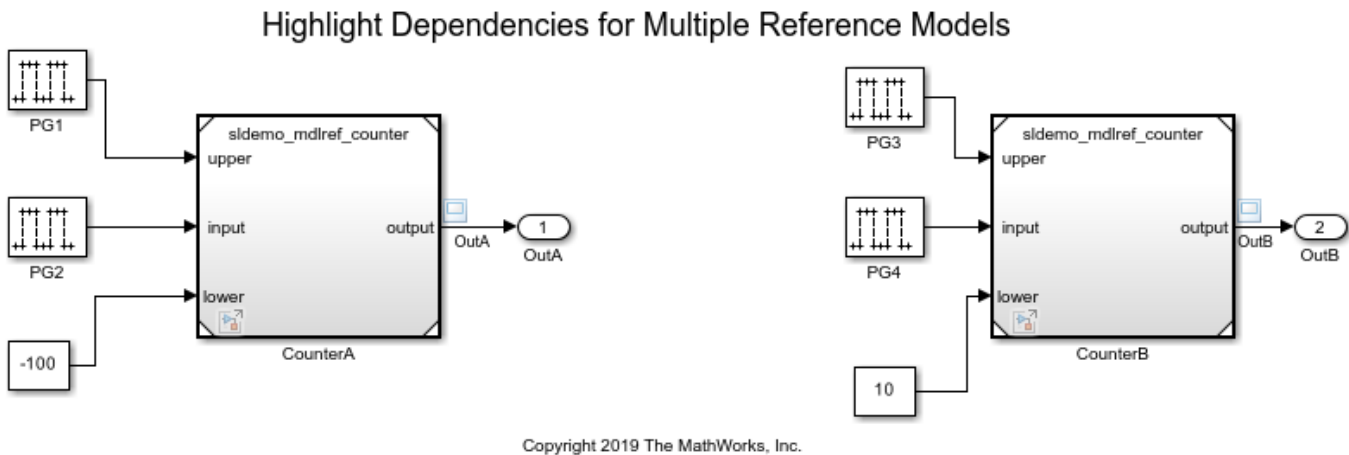
Highlight Dependencies for Multiple Instance Reference Models

To highlight the functional dependencies in a Simulink model with multiple instances of a referenced model, use Model Slicer. You can use Model Slicer on a Simulink model that contains single or multiple references to a same model in normal simulation mode.

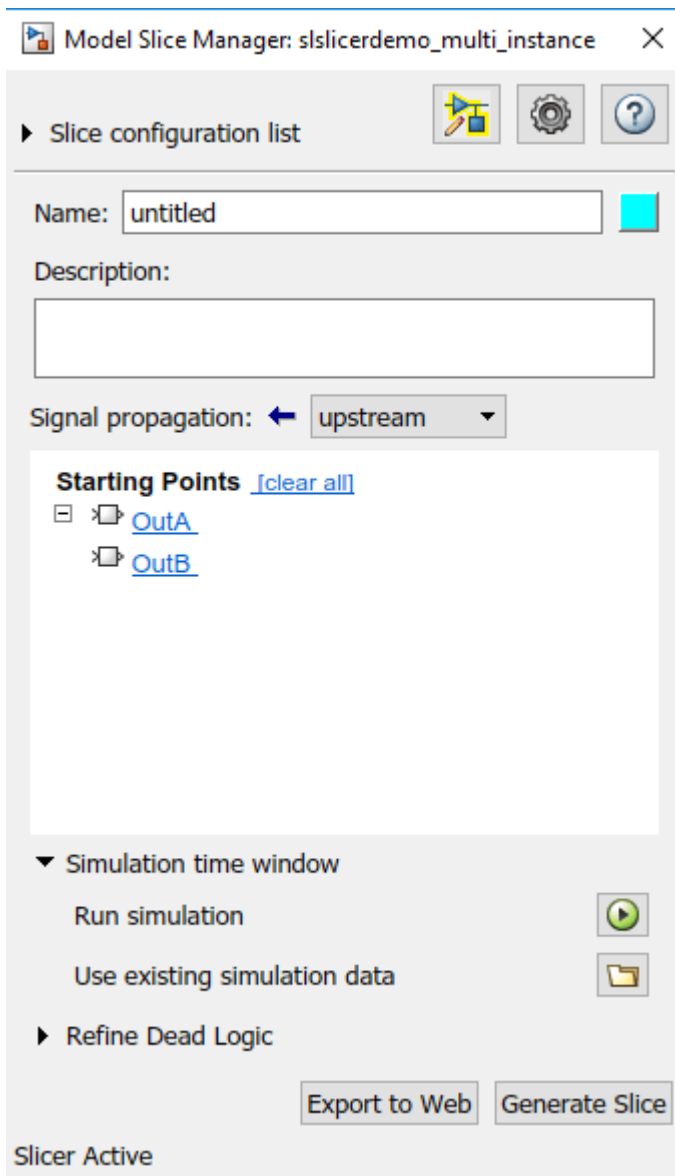
This example shows the behavior of Model Slicer when there are multiple instances of the referenced model. The `slslicerdemo_multi_instance` model consists of `sldemo_mdref_counter` referenced two times with different inputs during the course of the signal flow transition.

1. Open the model `slslicerdemo_multi_instance.slx`.

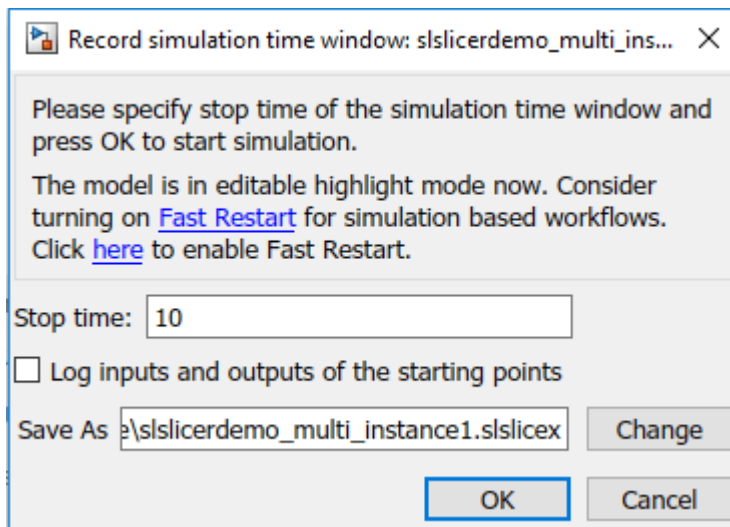
```
open_system('slslicerdemo_multi_instance');
```



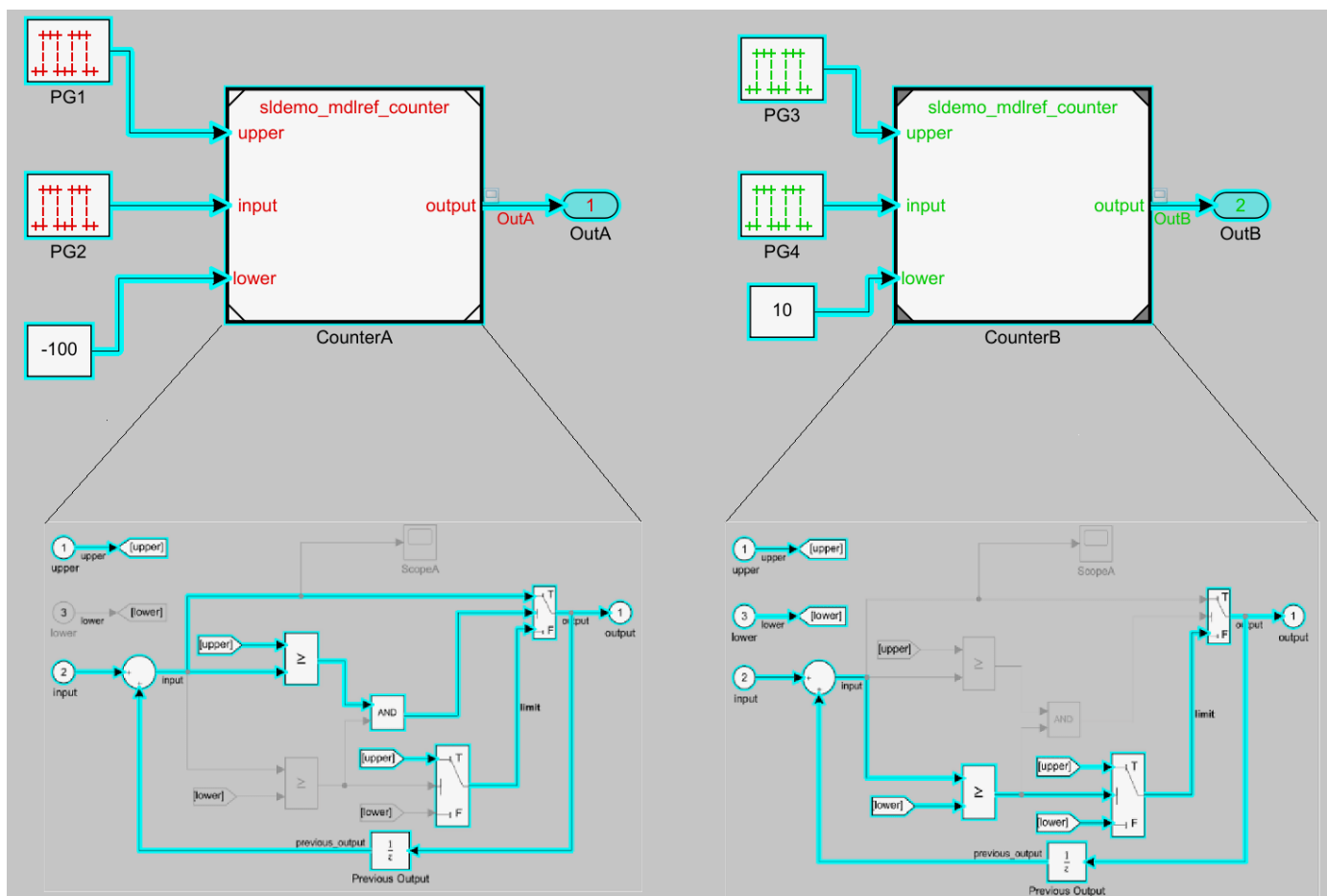
2. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.
3. In the Model Slicer window, click **Add all outports**. This sets **OutA** and **OutB** as starting points.
4. Ensure that the **Signal Propagation** is set to **upstream**.
5. In the **Simulation time window** section, click **Run simulation**.



6. In the simulation time window, click **OK**. The model simulation starts.



7. The simulated model highlights the upstream dependency of the outputs **OutA** and **OutB**.



You can notice that the referenced model in both the instances shows different signal propagations highlighted by Simulink Slicer for which the signal travels.

8. To generate the slice, click **Generate Slice**.

More About

- “Highlight Functional Dependencies” on page 8-2
- “Model Slicer Considerations and Limitations” on page 8-44

Refine Highlighted Model

After you highlight a model using Model Slicer, you can refine the dependency paths in the highlighted portion of the model. Using Model Slicer, you can refine a highlighted model by including only those blocks used in a portion of a simulation time window, or by excluding blocks or certain inputs of switch blocks. By refining the highlighted portion of your model, you can include only the relevant parts of your model.

In this section...

“Define a Simulation Time Window” on page 8-12

“Exclude Blocks” on page 8-17

“Exclude Inputs of a Switch Block” on page 8-20

Define a Simulation Time Window

You can refine a highlighted model to include only those blocks used in a portion of a simulation time window. Defining the simulation time window holds some switch blocks constant, and as a result removes inactive inputs.

1. Open the `sldvSliceClimateControlExample` model.

```
open_system("sldvSliceClimateControlExample");
```

2. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

When you open the Model Slicer, Model Slicer compiles the model. You then configure the model slice properties.

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

4. Set the slice properties:

(a) **Name:** Out1Simulation

(b) **Color:**  (cyan)

(c) **Signal Propagation:** upstream

Model Slicer

▼ Slice configuration list

	Name	Slice %
<input checked="" type="checkbox"/>	Out1Simulation	

Highlight only common elements

Name:

Description:

Signal propagation:


Starting Points [\[Add all outputs\]](#)
Right-click model items to select.

▶ Simulation time window
▶ Refine Dead Logic

5. In the top level of the model, select the **Out1** block as the slice starting point. Right-click the **Out1** block and select **Model Slicer > Add as Starting Point**.

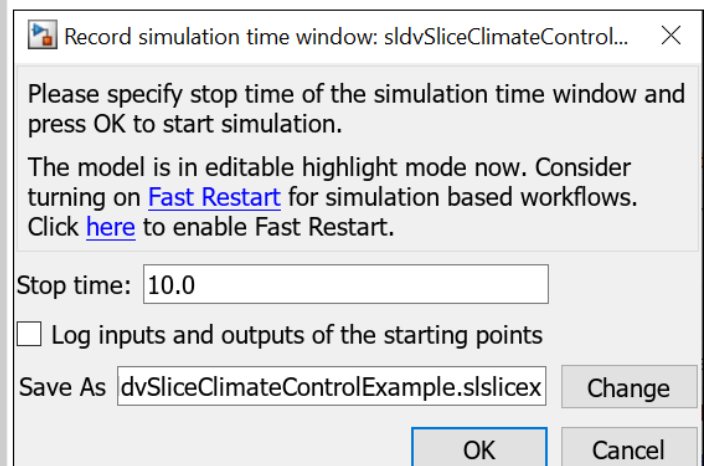
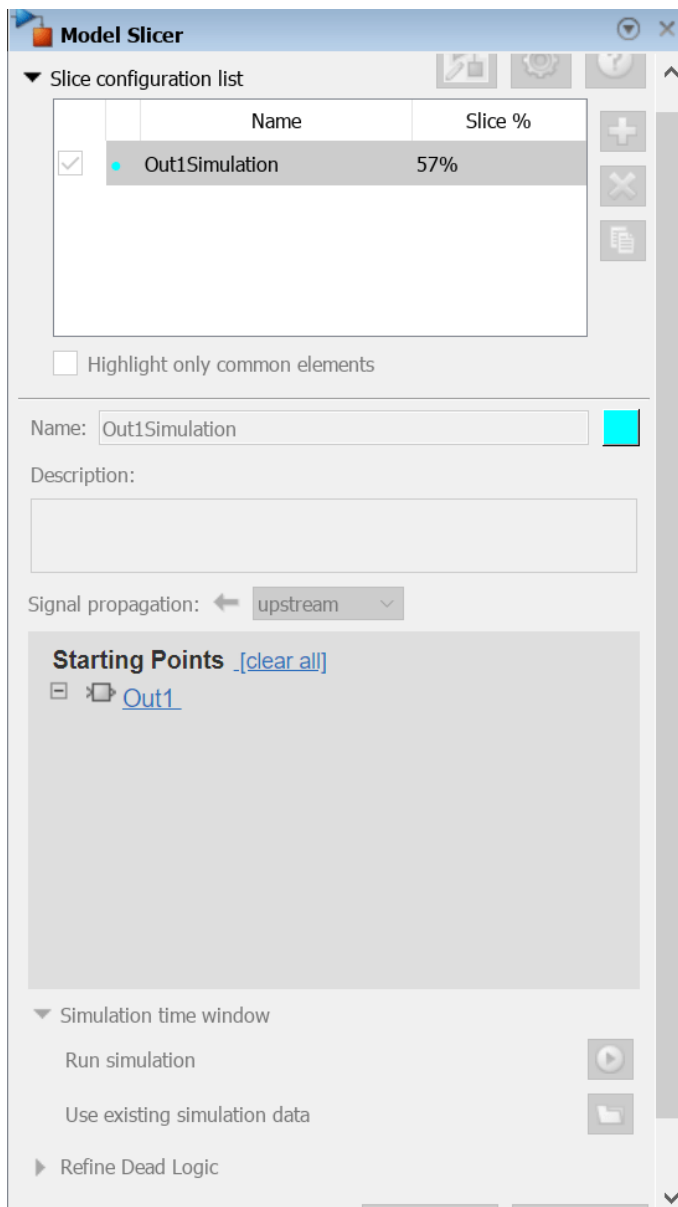
The model is highlighted.

6. In the Model Slicer, select **Simulation time window**.

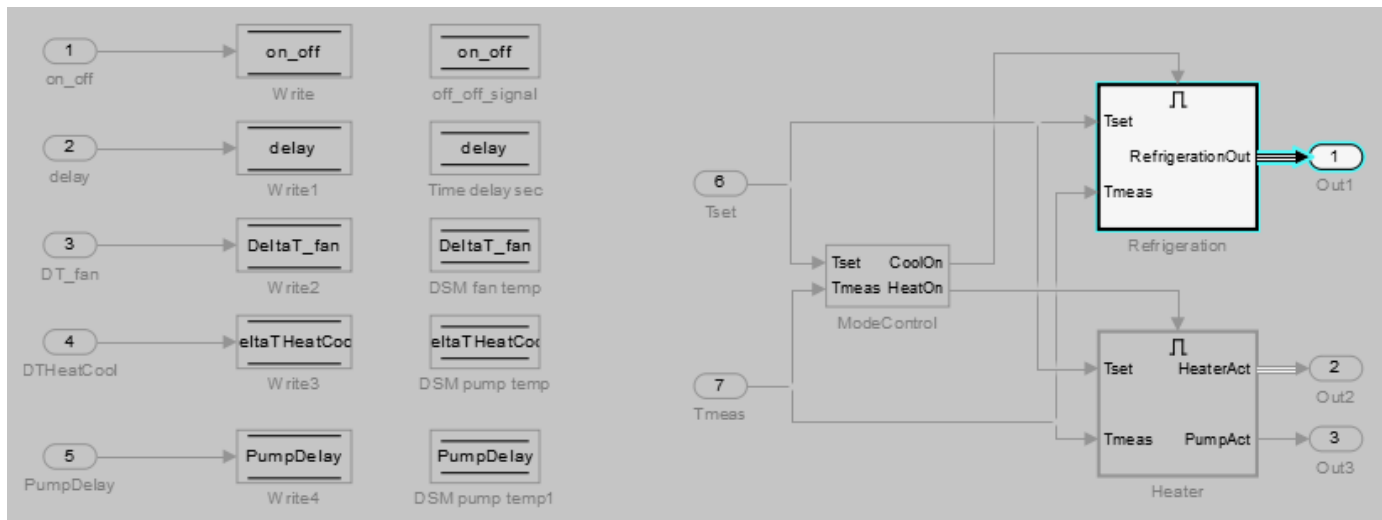
7. To specify the stop time of the simulation time window, click the run simulation button  in the Model Slicer.

8. Set the **Stop time** to 10.

9. Click **OK** to start the simulation.



The path is restricted to only those blocks that are active until the stop time that you entered.




10. To highlight the model for a defined simulation time window, set the **Stop time** to 5. Click **Highlight**.

Model Slicer


▼ Slice configuration list

	Name	Slice %
<input checked="" type="checkbox"/>	Out1Simulation	14%



Highlight only common elements

Name: 

Description:

Signal propagation:  ▼

Starting Points [\[clear all\]](#)

  [Out1](#)

▼ Simulation time window (Enabled)

Simulation data:

`sldvSliceClimateControlExample.slslicex`
0 to 10 seconds

Time window

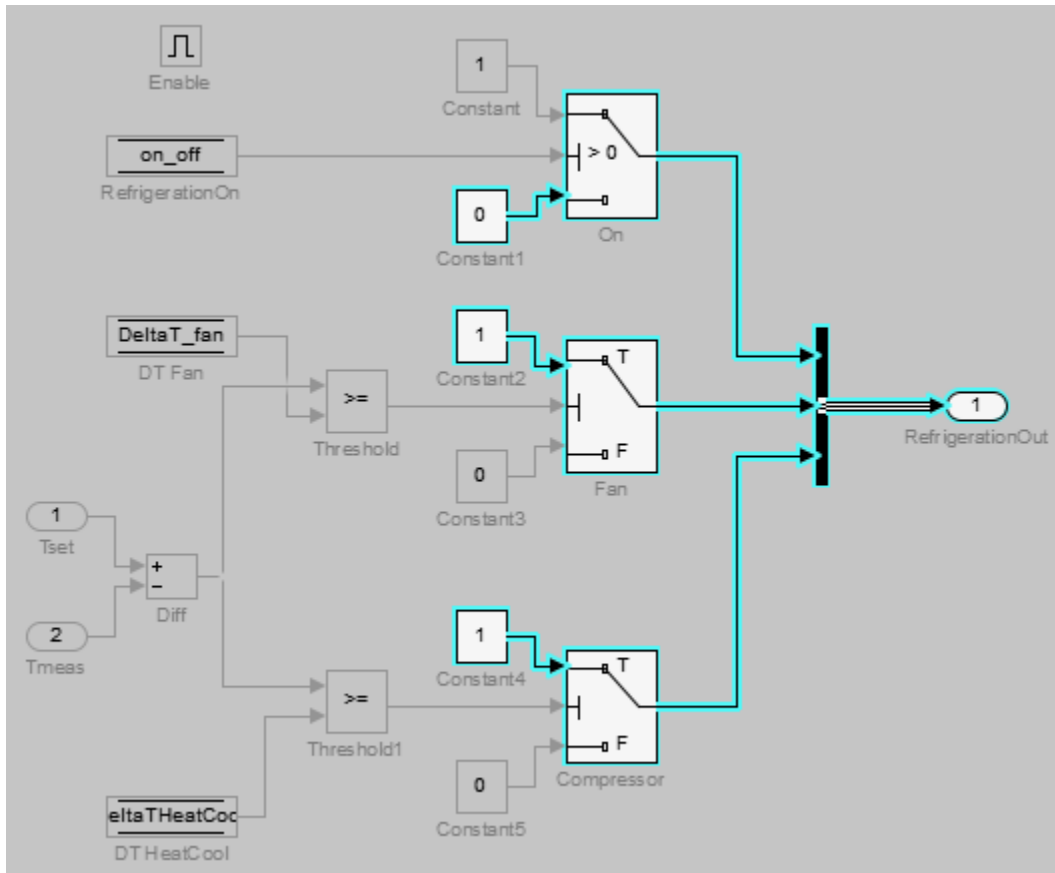
Start Stop

Actual simulation time: 0 to 10 seconds

► Refine Dead Logic

11. To see how this constraint affects the highlighted portion of the model, open the Refrigeration subsystem.

The highlighted portion of the model includes only the input ports of switches that are active in the simulation time window that you specified.





After you refine your highlighted model to include only those blocks used in a portion of a simulation time window, you can then “Create a Simplified Standalone Model” on page 8-29 incorporating the highlighted portion of your model.

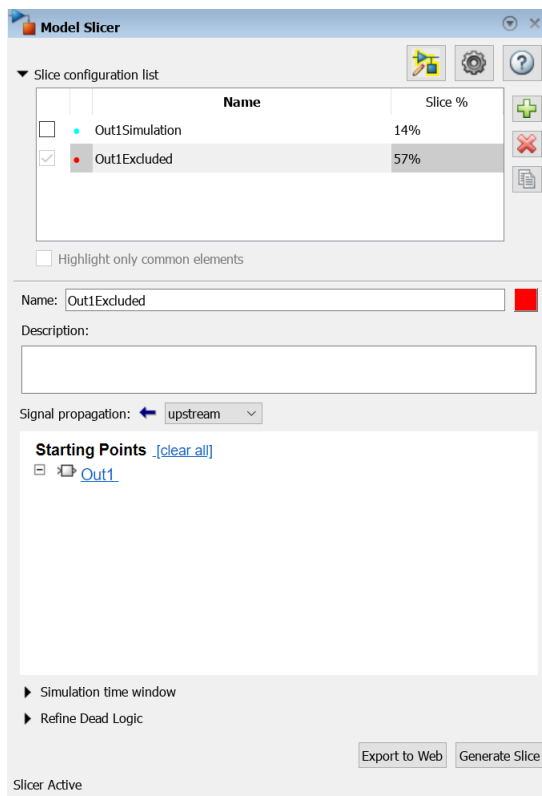
Exclude Blocks

You can refine a highlighted model to exclude blocks from the analysis. Excluding a block halts the propagation of dependencies, so that signals and model items beyond the excluded block in the analysis direction are ignored.

Exclusion points are useful for viewing a simplified set of model dependencies. For example, control feedback paths create wide dependencies and extensive model highlighting. You can use an exclusion point to restrict the analysis, particularly if your model has feedback paths.

Note Simplified standalone model creation is not supported for highlighted models with exclusion points.

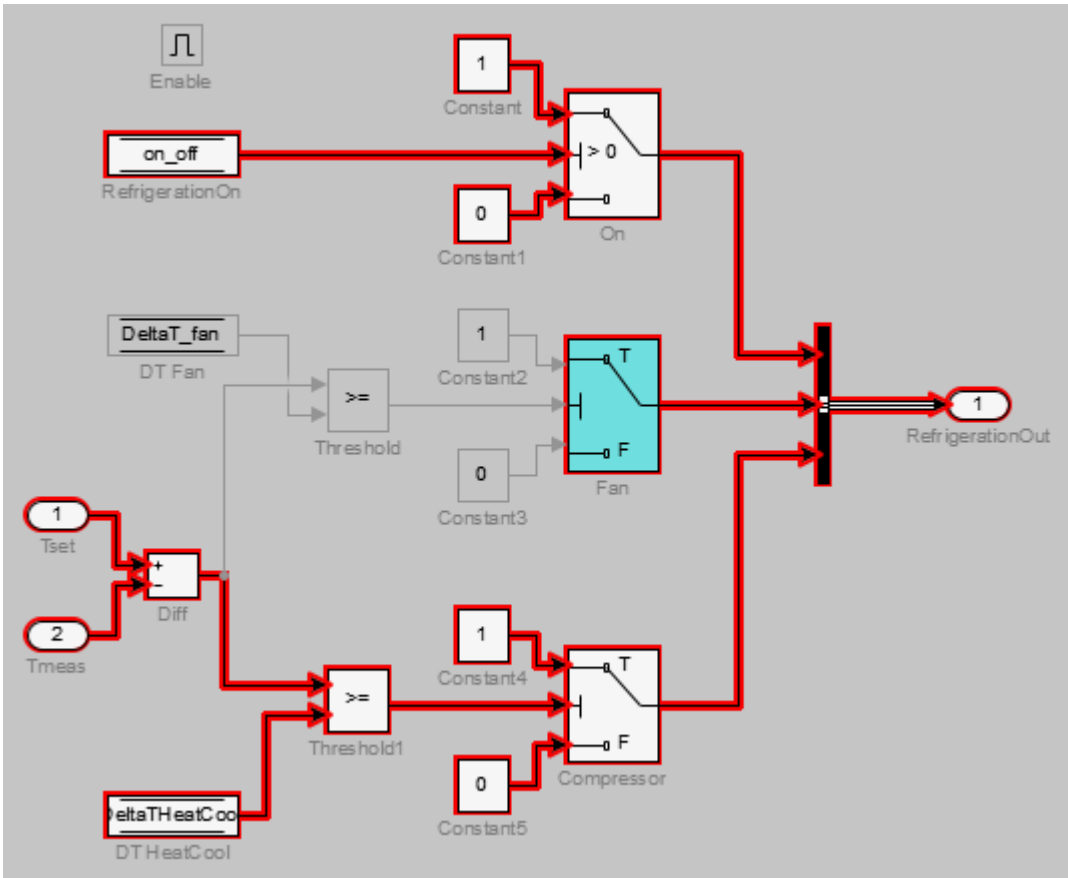
- 1 In the Model Slicer, click the arrow to expand the **Slice configuration list**.
- 2 To add a new slice configuration, click the add new button .
- 3 Set the slice properties:
 - **Name:** Out1Excluded
 - **Color:**  (red)
 - **Signal Propagation:** upstream
- 4 In the top level of the model, select the **Out1** block as the slice starting point. Right-click the **Out1** block and select **Model Slicer > Add as Starting Point**.



The model is highlighted.

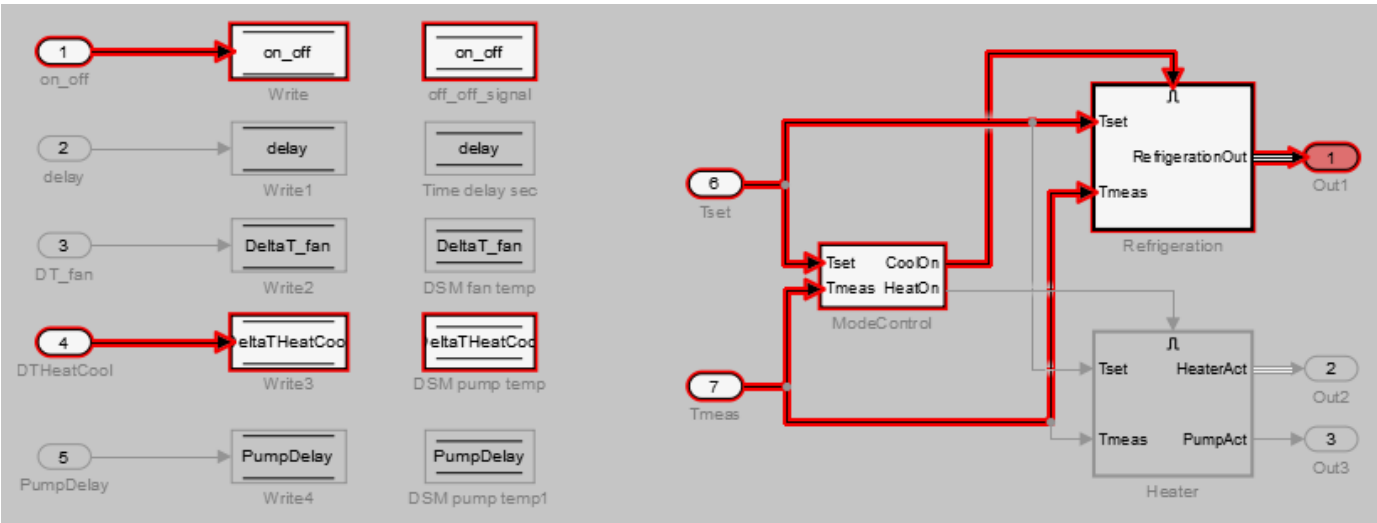
- 5 To open the subsystem, double-click Refrigeration.
- 6 Right-click the Fan switch block, and then select **Model Slicer > Add as Exclusion Point**.

The blocks that are exclusively upstream of the Fan switch block are no longer highlighted. The DT Fan Data Store Read block is no longer highlighted.



7 To see how this constraint affects the highlighted portion of the model, view the parent system.

The DSM fan temp Data Store Memory block and the Write2 Data Store Write block are no longer highlighted, because the DT Fan Data Store Read in the Refrigeration subsystem no longer accesses them.

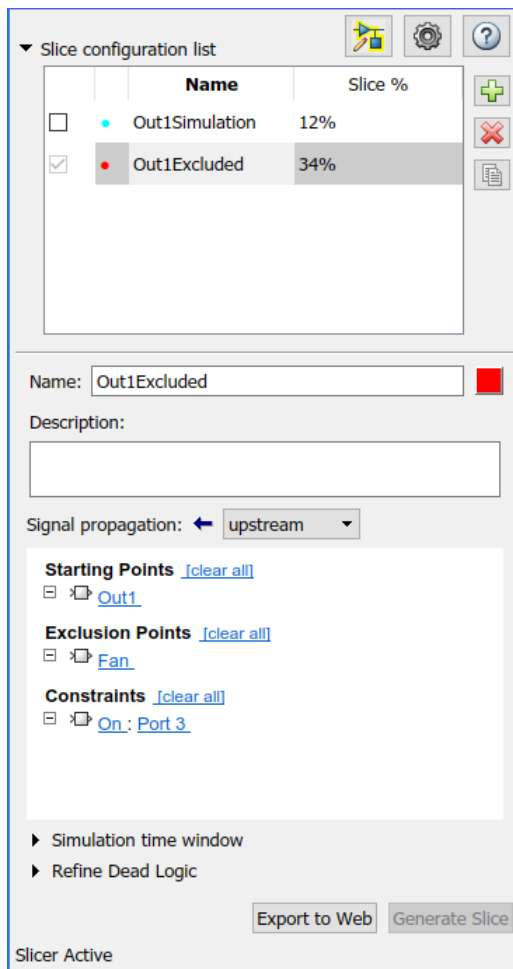


Exclude Inputs of a Switch Block

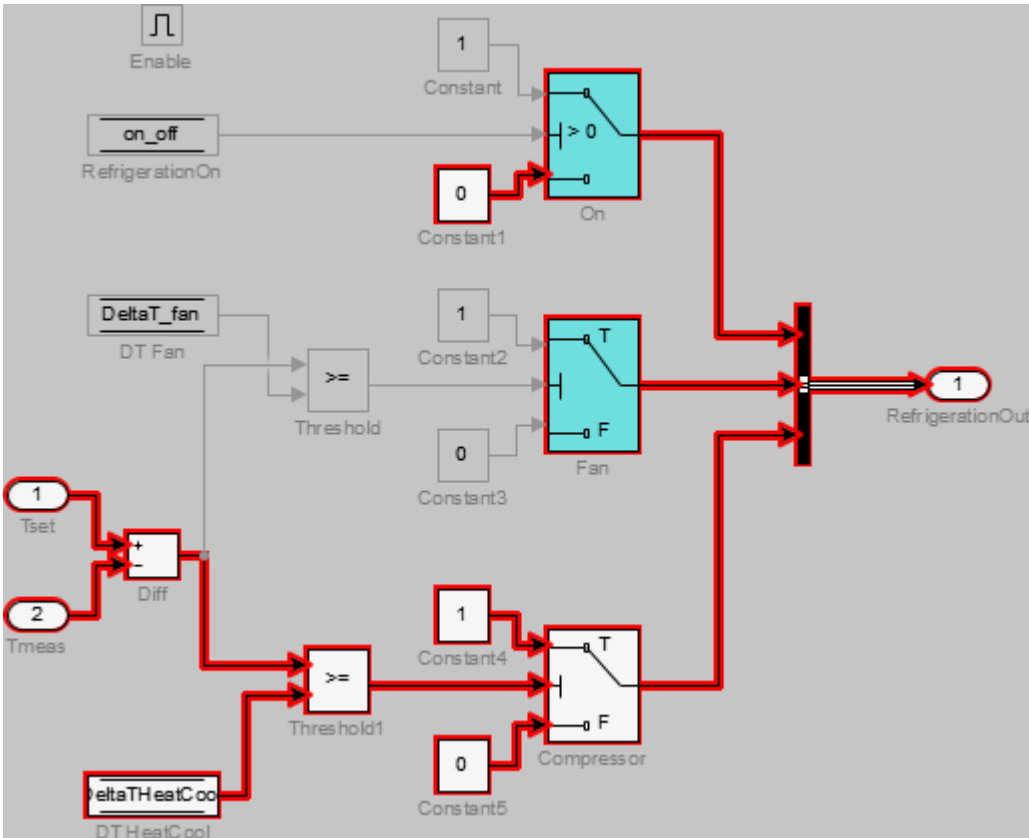
For complex signal routing, you can constrain the dependency analysis paths to a subset of the available paths through switch blocks. Constraints appear in the Model Slicer.

Note Simplified standalone model creation is not supported for highlighted models with constrained switch blocks.

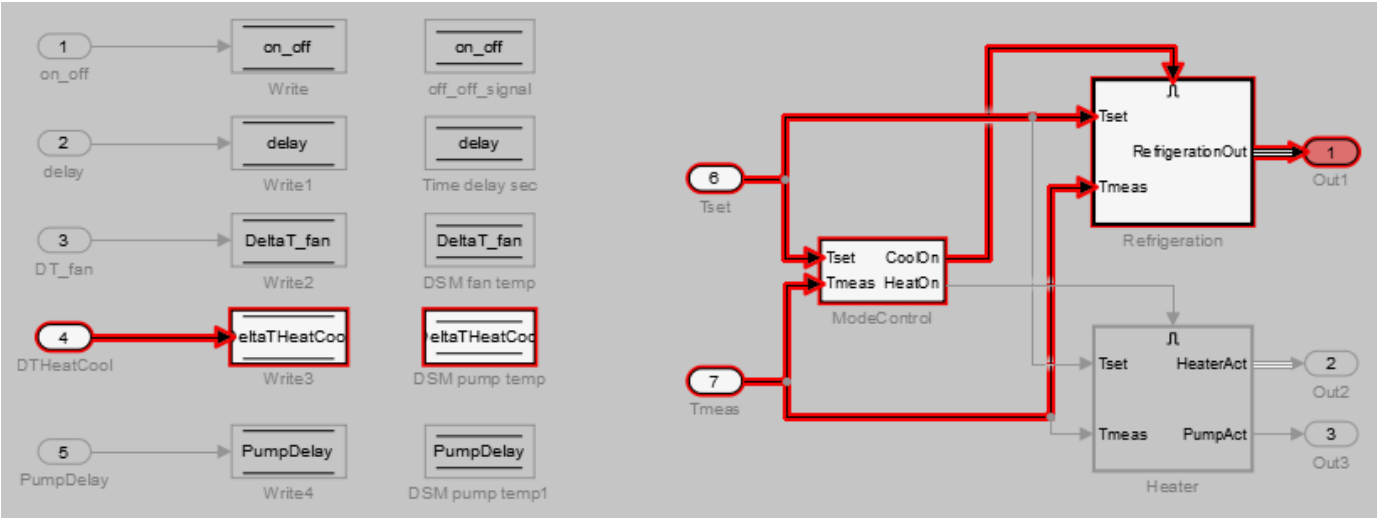
- 1 Double-click Refrigeration to open the subsystem.
- 2 Constrain the On switch block:
 - Right-click the switch block and select **Model Slicer > Add Constraint**.
 - In the Constraints dialog box, select **Port 3**.
 - Click **OK**.



The path is restricted to the Constant1 port on the switch. The blocks that are upstream of **Port 1** and **Port 2** of the constrained switch are no longer highlighted. Only the blocks upstream of **Port 3** are highlighted.



3 To see how this constraint affects the highlighted portion of the model, view the parent system.



See Also

More About

- “Create a Simplified Standalone Model” on page 8-29

- “Model Slicer Considerations and Limitations” on page 8-44

Refine Dead Logic for Dependency Analysis

To refine the dead logic in your model for dependency analysis, use the Model Slicer. To provide an accurate slice, Model Slicer leverages Simulink Design Verifier dead logic analysis to remove the unreachable paths in the model. Model Slicer identifies the dead logic and refines the model slice for dependency analysis. For more information on Dead logic, see “Dead Logic Detection” (Simulink Design Verifier).

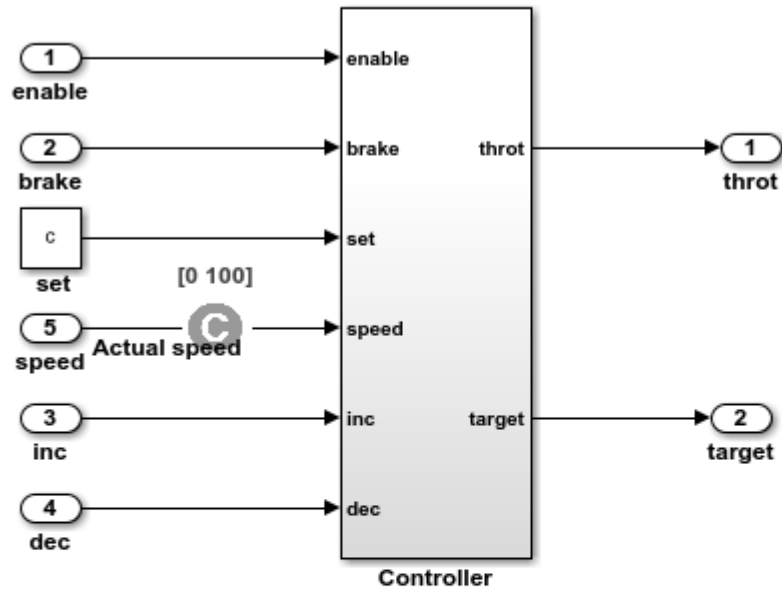
Analyze the Dead Logic

This example shows how to refine the model for dead logic. The `sldvSlicerdemo_dead_logic` model consists of dead logic paths that you refine for dependency analysis.

1. Open the `sldvSlicerdemo_dead_logic` model.
2. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

```
open_system('sldvSlicerdemo_dead_logic');
```

Simulink Design Verifier Cruise Control Test Generation



This example shows how to refine the model for dead logic. The model consists of a Controller subsystem that has a set value equal to 1. Dead logic refinement analysis identifies the dead logic in the model. The inactive elements are removed from the slice.

Run
(double-click)

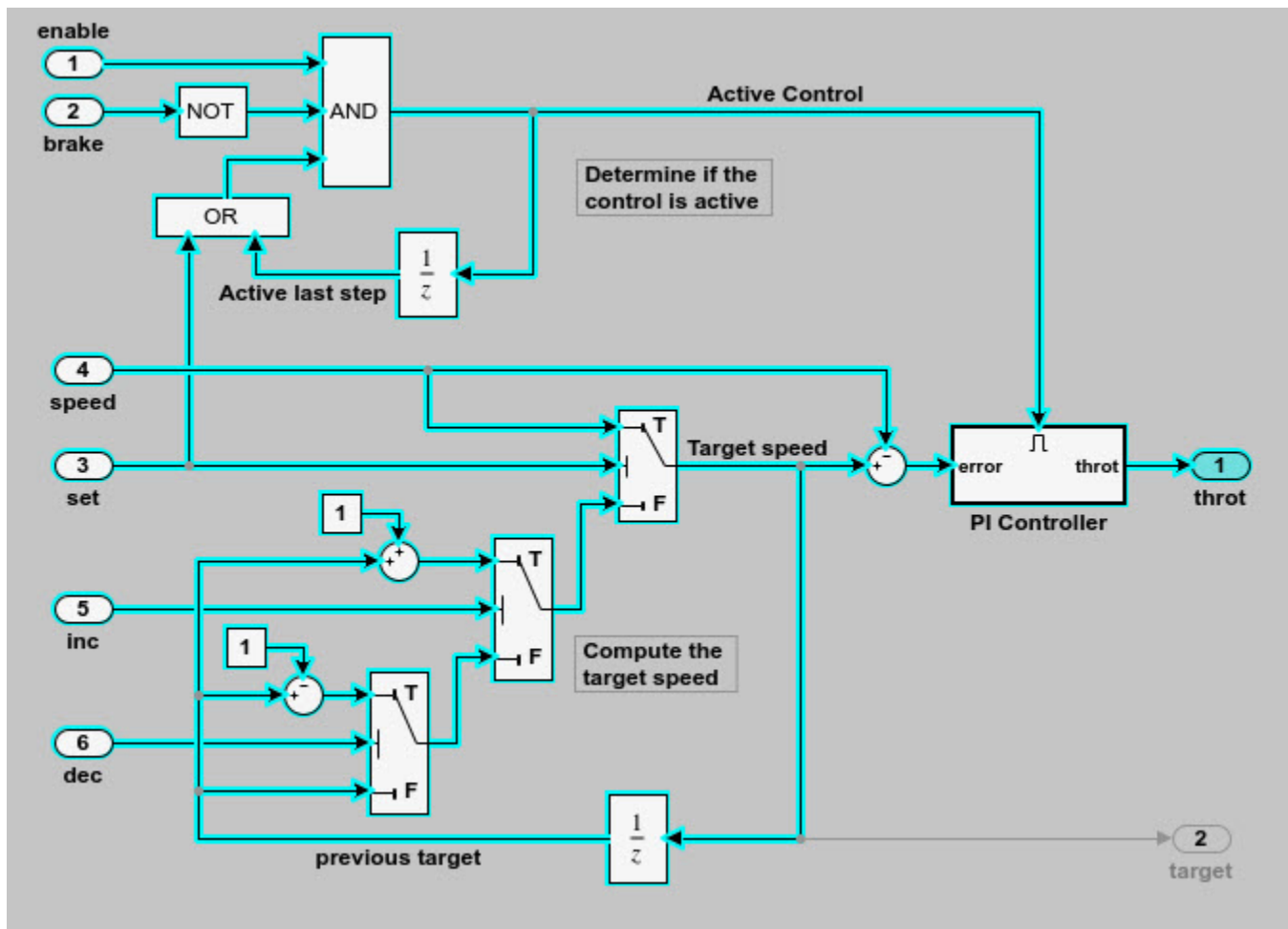
**Toggle Speed
Constraint**
(double-click)

Toggle Constraint

View Options
(double-click)

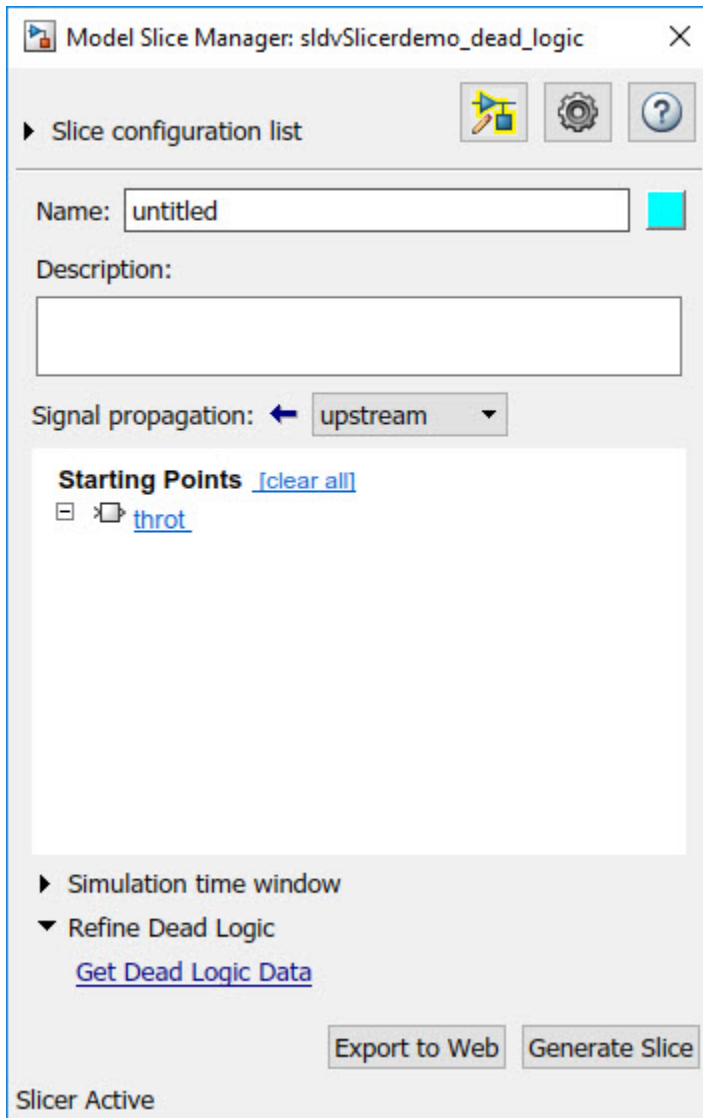
Copyright 2006-2018 The MathWorks, Inc.

Open the Controller subsystem and add the output `throt` as the starting point.

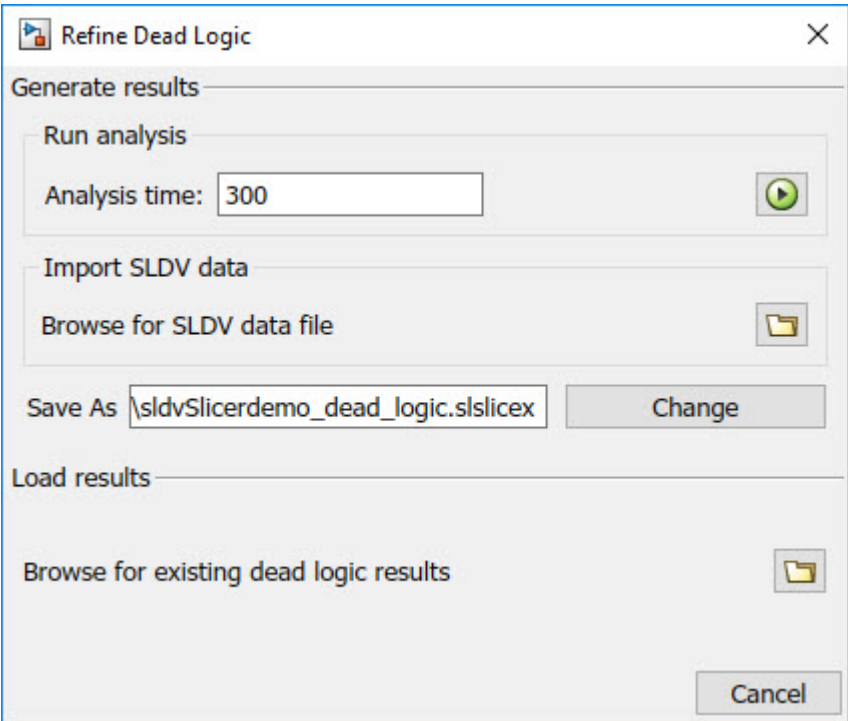


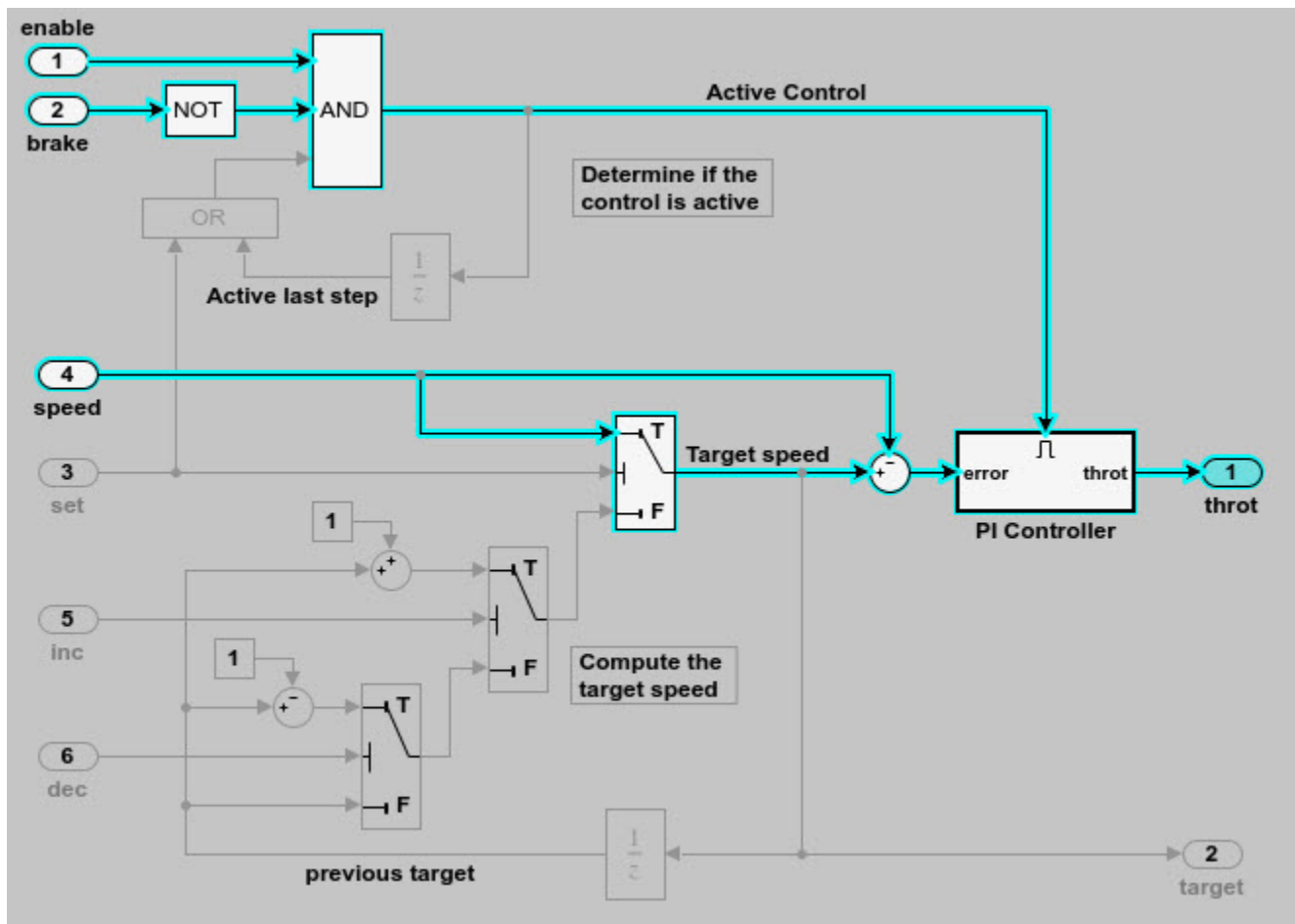
The Model Slicer highlights the upstream dependency of the throt output.

2. In the Model Slice Manager, select **Refine Dead Logic**.
3. Click **Get Dead Logic Data**.



4. Specify the **Analysis time** and run the analysis. You can import existing dead logic results from the `sldvData` file or load existing `.slicex` data for analysis. For more information, see “Refine Highlighted Model by Using Existing `.slicex` or Dead Logic Results” on page 8-63.





As the set input is equal to `true`, the `False` input to switch is removed for dependency analysis. Similarly, the output of block OR is always `true` and removed from the model slice.

See Also

More About

- “Refine Highlighted Model” on page 8-12
- “Refine Highlighted Model by Using Existing .slicex or Dead Logic Results” on page 8-63

Create a Simplified Standalone Model

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

Note Simplified standalone model creation is not supported for highlighted models with exclusion points or constrained switch blocks. If you want to view the effects of exclusion points or constrained switch blocks on a simplified standalone model, first create the simplified standalone model, and then add exclusion points or constrained switch blocks.

- 1 Highlight a portion of your model using Model Slicer.

See “Highlight Functional Dependencies” on page 8-2 and “Refine Highlighted Model” on page 8-12.
- 2 In the Model Slicer, click **Generate slice**.
- 3 In the **Select File to Write** dialog box, select the save location and enter a model name.

The simplified standalone model contains the highlighted model items.
- 4 To remove highlighting from the model, close the Model Slicer.

When generating a simplified standalone model from a model highlight, you might need to refine the highlighted model before the simplified standalone model can compile. See the “Model Slicer Considerations and Limitations” on page 8-44 for compilation considerations.

See Also

More About

- “Configure Model Highlight and Sliced Models” on page 8-41

Highlight Active Time Intervals by Using Activity-Based Time Slicing

Stateflow states and transitions can be active, inactive, or sleeping during model simulation. You can use Model Slicer to constrain model highlighting to only highlight the time intervals in which certain Stateflow “Represent Operating Modes by Using States” (Stateflow) and “Transition Between Operating Modes” (Stateflow) are active. Therefore, you are able to refine your area of interest to only those portions of your model that affect model simulation during the operation of the selected states and transitions. You can also constrain model highlighting to the intersection of the time intervals of two or more states or transitions.

In this section...

“Highlighting the Active Time Intervals of a Stateflow® State or Transition” on page 8-30

“Activity-Based Time Slicing Limitations and Considerations” on page 8-36

“Stateflow State and Transition Activity” on page 8-36

Highlighting the Active Time Intervals of a Stateflow® State or Transition

The `slslicer_fuelsys_activity_slicing` model contains a fault-tolerant fuel control system. In this tutorial, you use activity-based time slicing to refine a model highlight to only those time intervals in which certain states and transitions are active. You must be familiar with how to “Highlight Functional Dependencies” on page 8-2 by using Model Slicer.

Create a Dynamic Slice Highlight for an Area of Interest

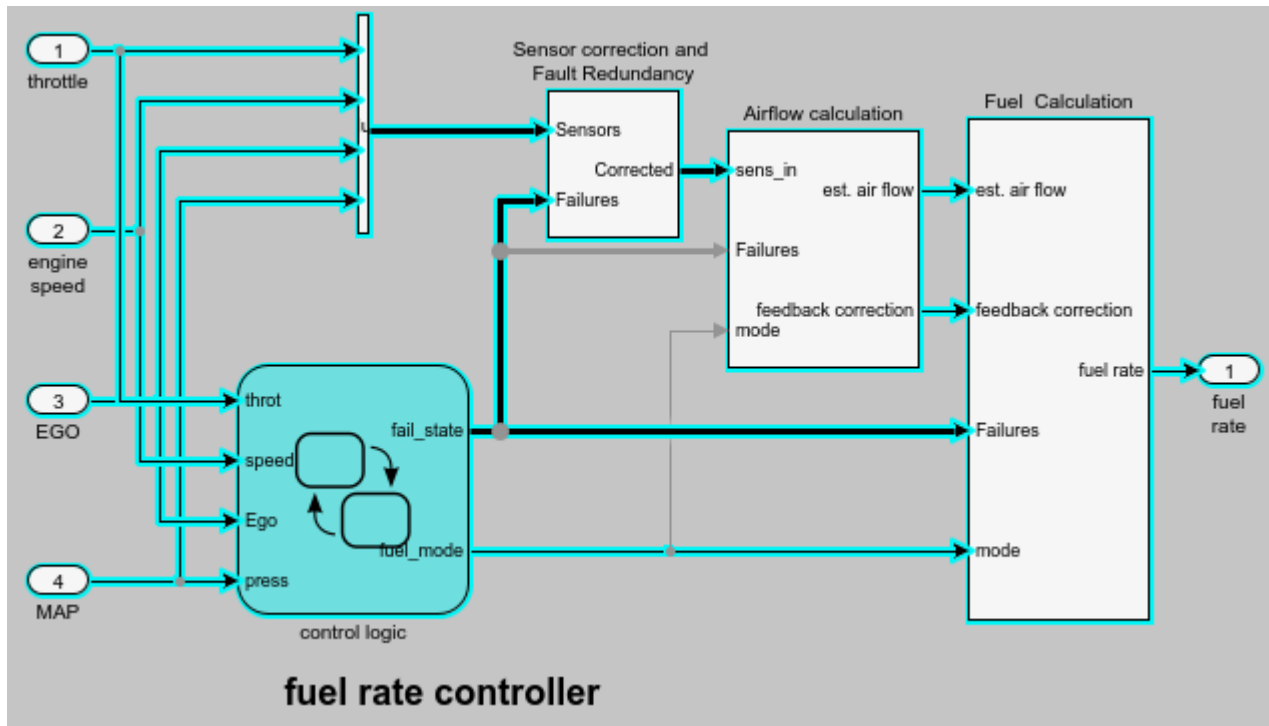
1. Open the `slslicer_fuelsys_activity_slicing` model.

```
open_system('slslicer_fuelsys_activity_slicing');
```

2. Open Model Slicer and add the `control_logic` Stateflow chart in the fuel rate controller subsystem as a Model Slicer starting point.

3. Highlight the portions of the model that are upstream of the `control_logic` Stateflow chart.

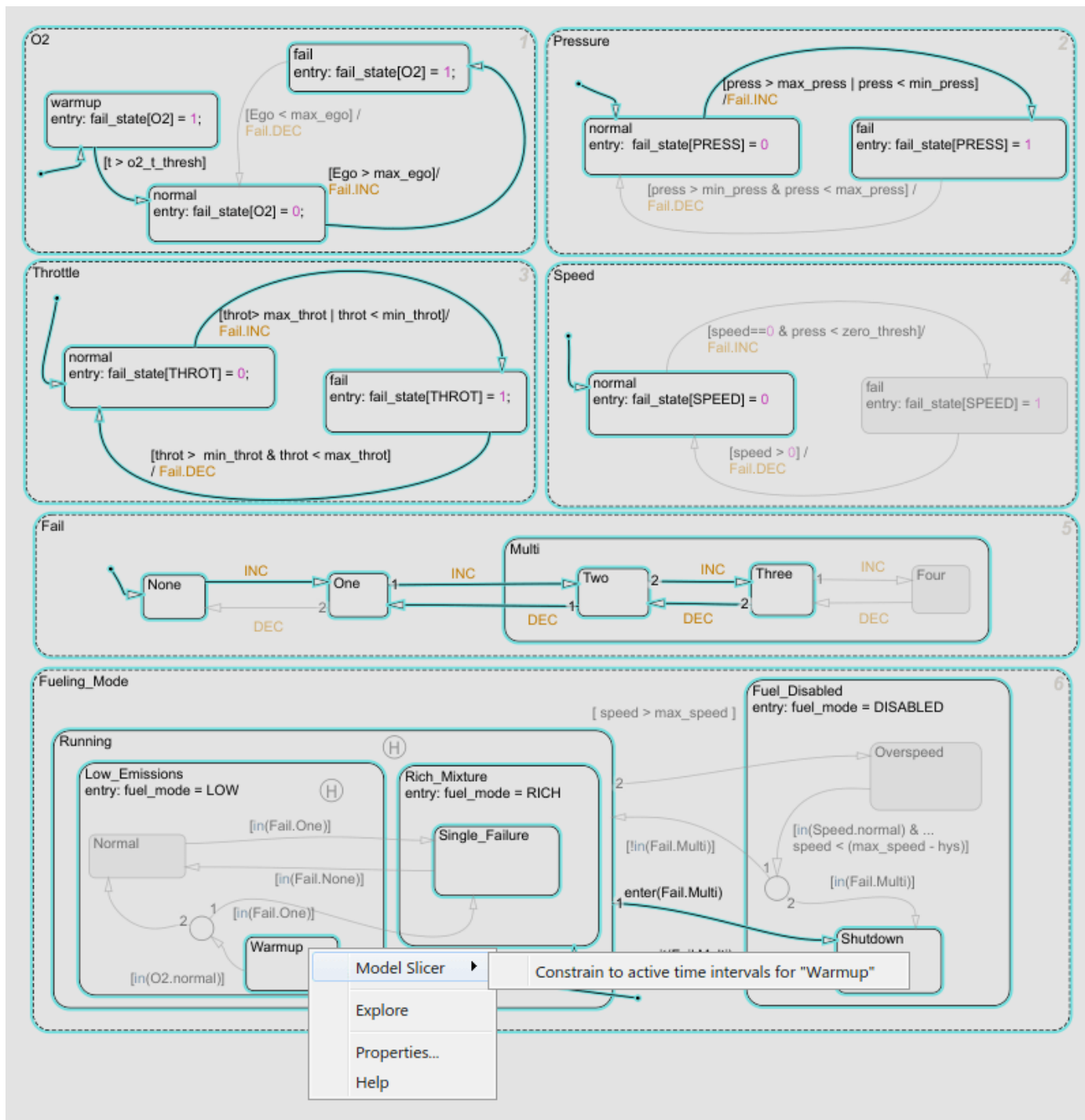
4. Simulate the model within a restricted simulation time window (maximum 20 seconds) to highlight only the areas of the model upstream of the starting point and active during the time window of interest.



Constrain the Model Highlight to the Active Time Interval of a Stateflow State

1. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.
2. Navigate to the `control logic` Stateflow chart in the `fuel rate controller` subsystem.
`open_system('slslicer_fuelsys_activity_slicing/fuel rate controller/control logic');`

3. To constrain the model highlight to only those time intervals in which the **Fueling_Mode > Running > Low_Emissions > Warmup** state is active, right-click the `Warmup` state and select **Model Slicer > Constrain to active time intervals for Warmup**.



Model Slicer is updated to highlight only those portions of the model that are active during the time intervals in which the Warmup state is active.



The Model Slicer is also updated to show the time interval in which the Warmup state is active:

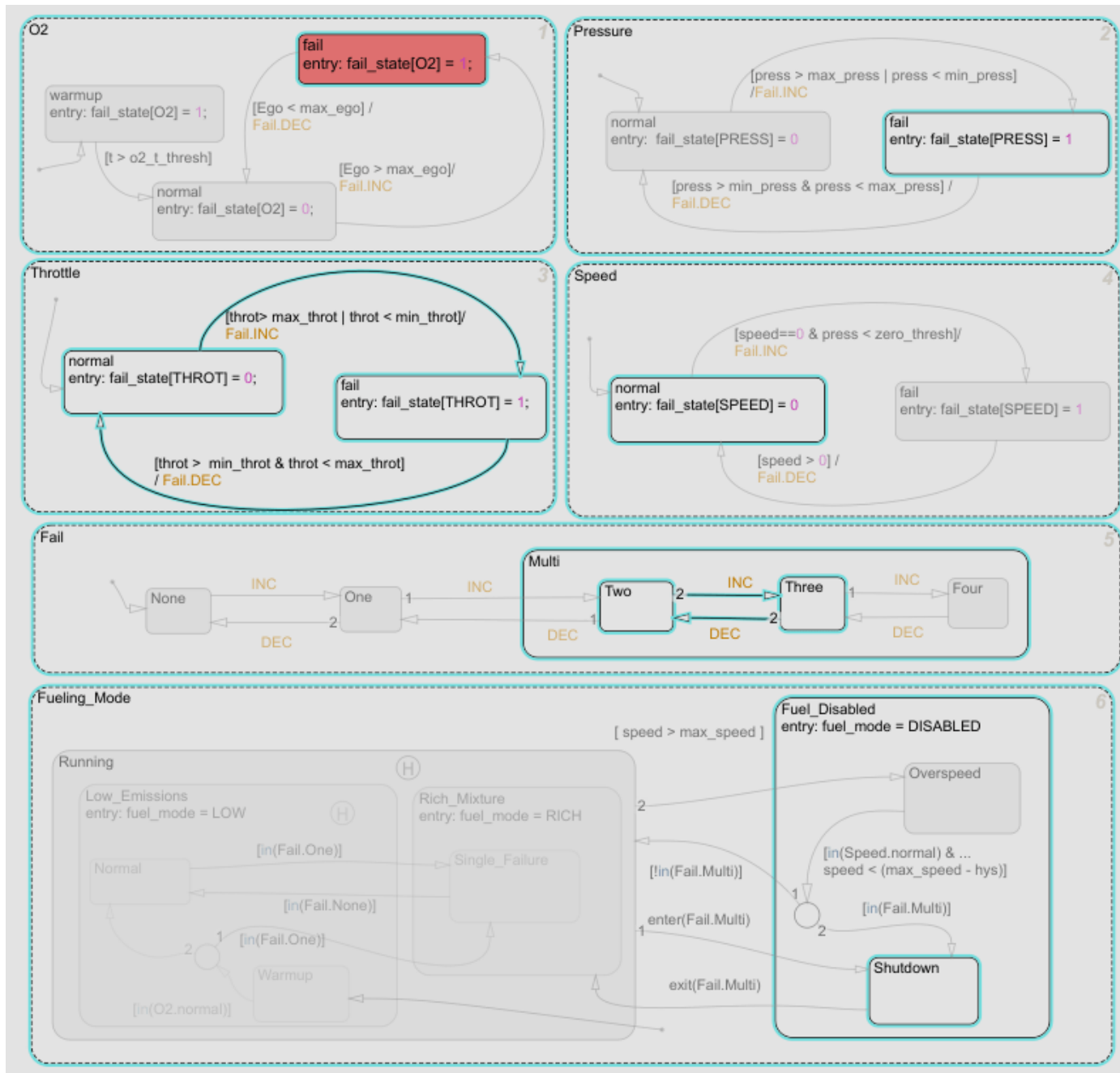
Actual simulation time: 0.01 to 3.86 seconds : 1 active interval.

The highlight shows a normal to fail transition in the Pressure state, showing that a pressure failure occurred during the time interval in which the Warmup state was active.

Constrain the Model Highlight to the Intersection of the Active Time Intervals of a Stateflow State and Transition

1. Clear any time interval constraints from the Model Slicer.

2. Constrain the model highlight to only those time intervals in which the **O2 > fail** state is active.

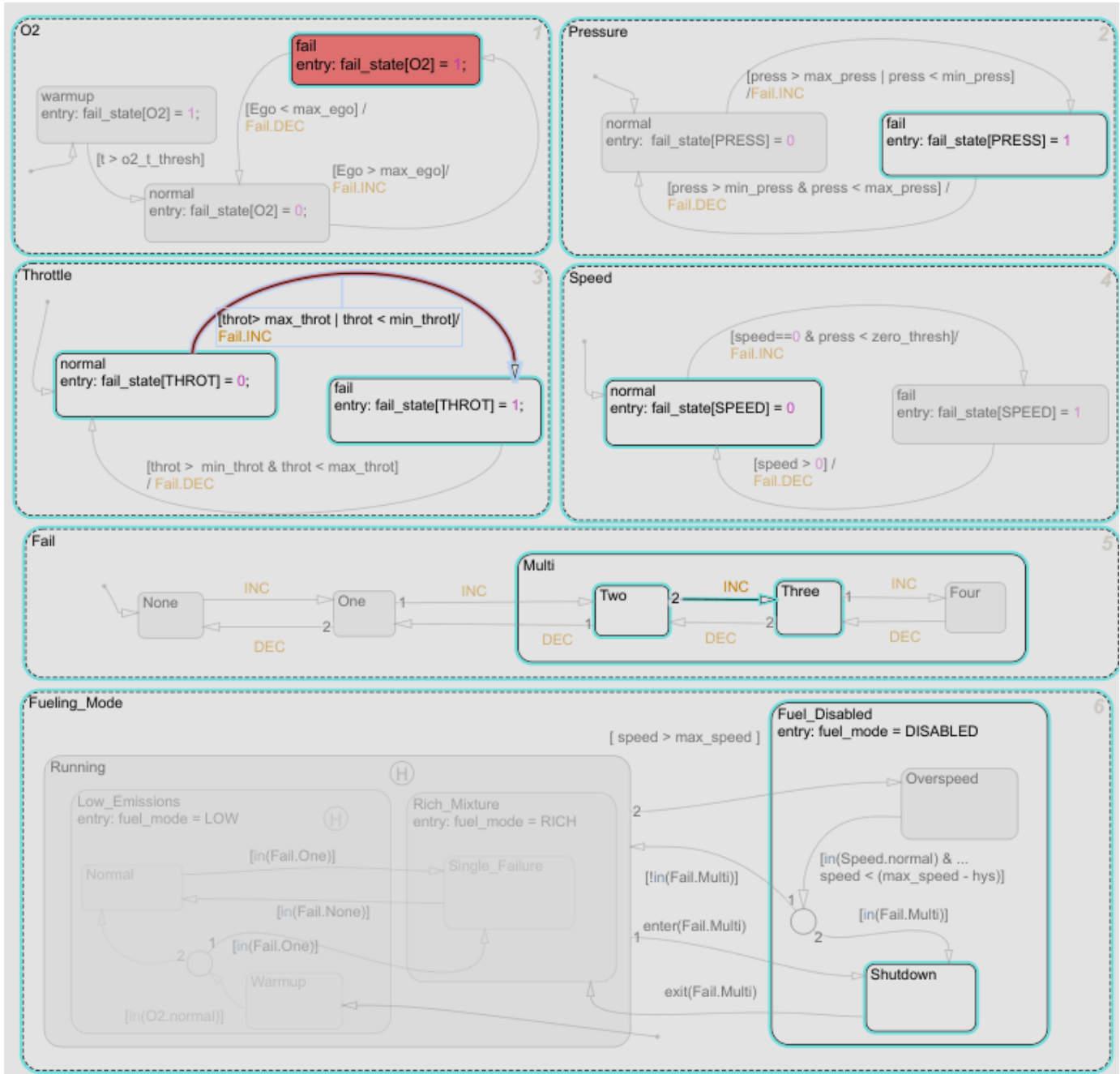


Model Slicer is updated to highlight only those portions of the model that are active during the time intervals in which the **O2 > fail** state is active. The Model Slicer is also updated to show the time interval in which the **O2 > fail** state is active:

Actual simulation time: 4.83 to 20 seconds : 1 active interval

3. To constrain the highlighting to the time interval in which the **O2 > fail** state is active and the normal to fail transition occurs for the Throttle chart, right-click the normal to fail transition

and add it as a constraint. Model Slicer is updated to highlight only those portions of the model that are active during the intersection of the time intervals in which the **O2 > fail** state is active and the normal to fail transition occurs for the Throttle chart.



The Model Slicer is also updated to show the time interval in which the **O2 > fail** state is active and the normal to fail transition occurs for the Throttle chart:

Actual simulation time: 13.87 to 13.87 seconds : 1 active interval

Activity-Based Time Slicing Limitations and Considerations

For limitations and considerations of activity-based time slicing, see “Model Slicer Considerations and Limitations” on page 8-44.

Stateflow State and Transition Activity

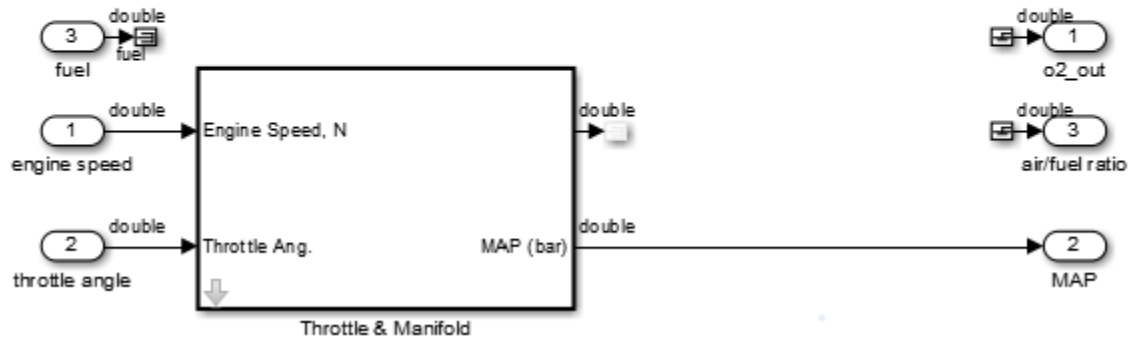
For more information on Stateflow state and transition activity, see “Chart Simulation Semantics” (Stateflow), “Types of Chart Execution” (Stateflow), and “Syntax for States and Transitions” (Stateflow).

See Also

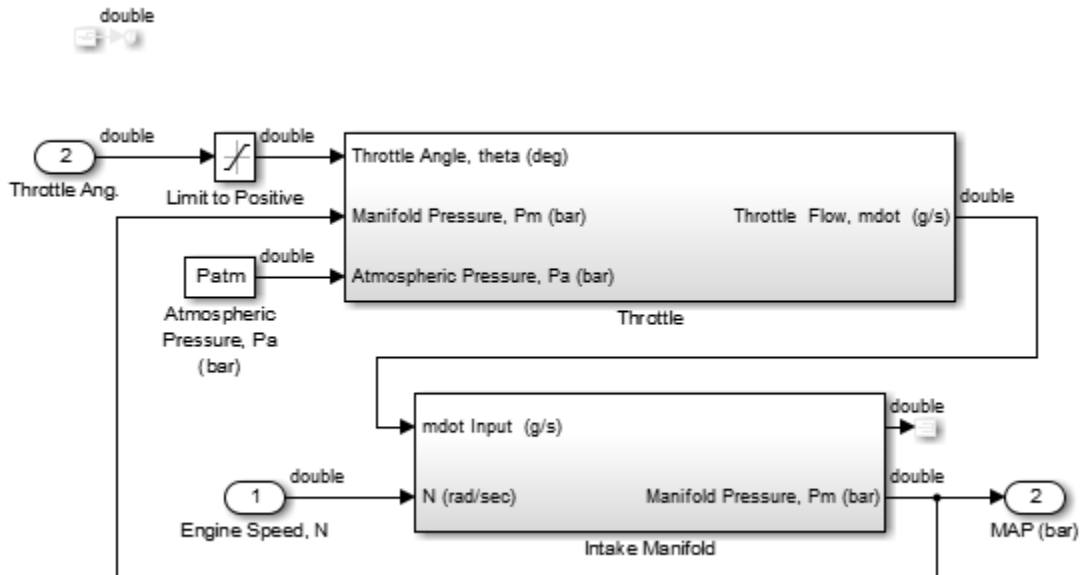
More About

- “Using Model Slicer with Stateflow” on page 8-50
- “Represent Operating Modes by Using States” (Stateflow)
- “Transition Between Operating Modes” (Stateflow)

Engine Gas Dynamics



7. Click the arrow to look under the mask of the `ThrottleAndManifold` subsystem. The content from the referenced model is inlined into the model in the masked subsystem.



Workflow for Dependency Analysis

In this section...

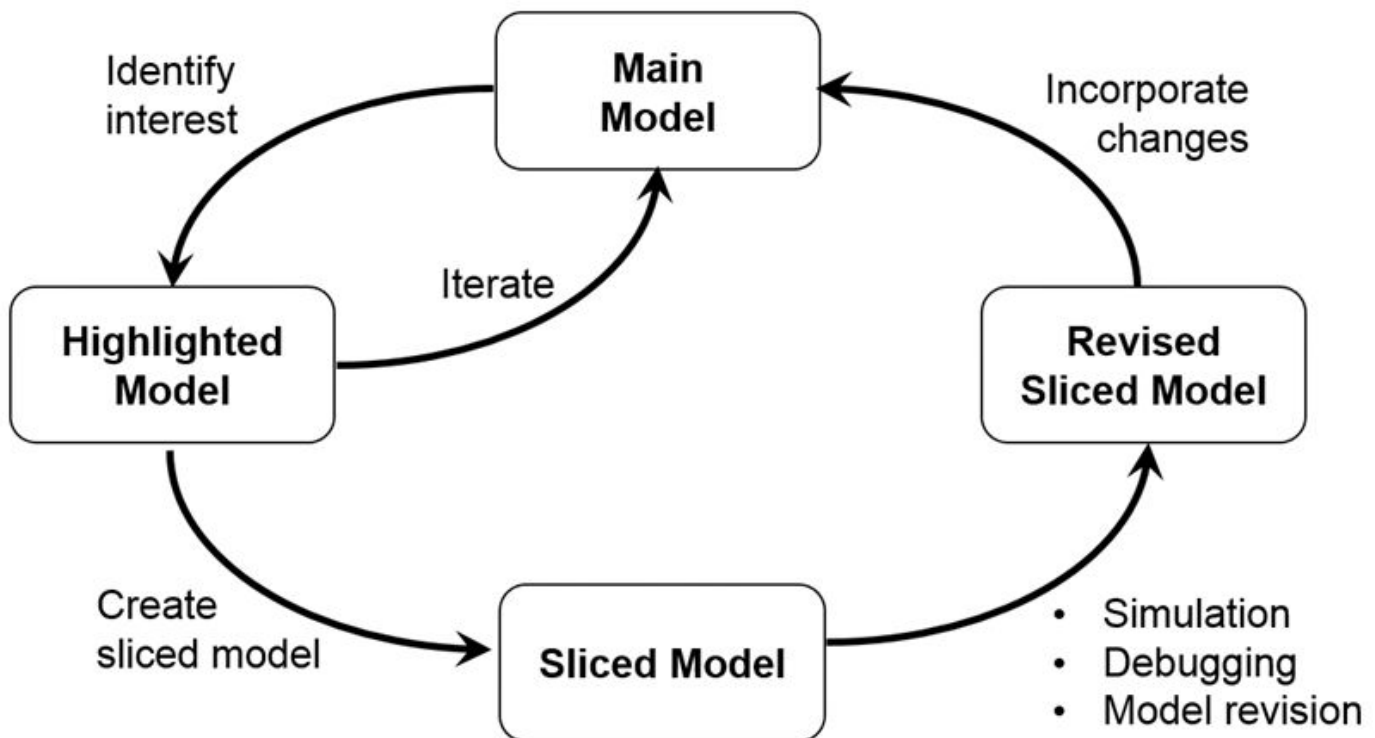
“Dependency Analysis Workflow” on page 8-39

“Dependency Analysis Objectives” on page 8-39

Model analysis includes determining dependencies of blocks, signals, and model components. For example, to view blocks affecting a subsystem output, or trace a signal path through multiple switches and logic. Determining dependencies can be a lengthy process, particularly for large or complex models. Use Model Slicer as a simple way to understand functional dependencies in large or complex models. You can also use Model Slicer to create simplified standalone models that are easier to understand and analyze, yet retain their original context.

Dependency Analysis Workflow

The dependency analysis workflow identifies the area of interest in your model, generates a sliced model, revises the sliced model, and incorporates those revisions in the main model.



Dependency Analysis Objectives

To identify the area of interest in your model, determine objectives such as:

- What item or items are you analyzing? Analysis begins with at least one starting point.
- In what direction does the analysis propagate? The dependency analysis propagates upstream, downstream, or bidirectionally from the starting points.

- What model items or paths do you want to exclude from analysis?
- What paths do you want to constrain? If your model has switches, you can constrain the switch positions for analysis.
- Is your model a closed-loop system? If so, the highlighted portion of the model can include model dependencies from the feedback loop. Consider excluding blocks from the feedback loop to refine the highlighted portion of the model.
- Do you want to analyze static dependencies, or include simulation effects? Static analysis considers model dependencies for possible simulation paths. Simulation-based analysis highlights only paths active during simulation.

See Also

Related Examples

- “Highlight Functional Dependencies” on page 8-2
- “Refine Highlighted Model” on page 8-12
- “Create a Simplified Standalone Model” on page 8-29


Configure Model Highlight and Sliced Models


In this section...

“Model Slicer” on page 8-41
 “Model Slicer Options” on page 8-41
 “Storage Options” on page 8-41
 “Refresh Highlighting Automatically” on page 8-42
 “Sliced Model Options” on page 8-42
 “Trivial Subsystems” on page 8-42
 “Inline Content Options” on page 8-43


Model Slicer

Set the properties of your model highlight and standalone sliced model using the Model Slicer configuration window.

Click the toggle mode button  to switch between model edit mode and model highlight mode.

If automatic highlighting is disabled in the slice settings, refresh the model highlight using the refresh button . Refresh the highlight after changing the slice configuration.

Model Slicer Options

You can customize the slice behavior using the options dialog box, which is accessed with the options button .

Storage Options

Changes you make to a model slice configuration are saved automatically. You can store the slice configuration in the model SLX file, or in an external SLMS file. Saving the configuration externally can be useful if your SLX file is restricted by a change control system.

To set the storage location, click the options  button in the Model Slicer and set the location in the **Storage options** pane.

Settings

Store in <model_name>.slx

Saves the model slice configuration in your model’s SLX file

Store in external file

Saves the model slice configuration in a separate SLMS file you specify by clicking the **Save As** button. The model slice configuration filename is shown in **File**.

Refresh Highlighting Automatically

Enables automatic refresh of a model highlight after changing the slice configuration.

Settings

on (default)

Model highlighting refreshes automatically.

off

Model highlighting must be refreshed manually. Click the refresh button  in the Model Slicer to refresh the highlight.

Sliced Model Options

You can control what items are retained when you create a sliced model from a model highlight using the options in the **Sliced model options** pane.

Option	On (selected)	Off (cleared)
Retain signal observers	Signal observers, such as scopes, displays, and test condition blocks, are retained in the sliced model.	Signal observers are not retained in the sliced model (default).
Retain root-level inports and outports	Root-level ports are retained in the sliced model (default).	Root-level ports are not retained in the sliced model.
Expand trivial subsystems	Trivial subsystems are expanded in the sliced model and the subsystem boundary is removed (default).	Trivial subsystems are not expanded in the sliced model and the subsystem boundary is retained. See “Trivial Subsystems” on page 8-42.

Trivial Subsystems

If a subsystem has all of these characteristics, Model Slicer considers the subsystem trivial:

- If the subsystem is virtual, it contains three or fewer nonvirtual blocks.
- If the subsystem is atomic, it contains one or fewer nonvirtual blocks.
- The subsystem has two or fewer inports.
- The subsystem has two or fewer outports.
- The active inport or outport blocks of the subsystem have default block parameters.
- The system does not contain Goto Tag Visibility blocks.

- In the Block Properties dialog box, the subsystem **Priority** is empty.
- The data type override parameter (if applicable) is set to use local settings.

Note If you generate a sliced model which does not remove contents of a particular subsystem, the subsystem remains intact in the sliced model.

Inline Content Options

When you create a sliced model from a highlight, model items can be inlined into the sliced model. The **Inline content options** pane controls which model components are inlined in generating a sliced model.

Model Component	Inlining on (selected)	Inlining off (cleared)
Libraries	Model items inside sliced libraries are inlined in the sliced model and the library link is removed. (default)	Model items inside sliced libraries are not inlined in the sliced model and library link remains in place.
Masked subsystems	Model items inside sliced masked subsystems are inlined in the sliced model. (default) The mask is retained in the sliced model.	Model items inside sliced masked subsystems are not inlined in the sliced model and the mask is retained.
Model blocks	Model items are inlined to the sliced model from the model referenced by the Model block. The Model block is removed. (default) Note Model Slicer cannot inline model blocks that are not in Normal mode.	Model items are not inlined to the sliced model from the model referenced by the Model block. The Model block is retained.
Variants	Model items are inlined to the sliced model from the active variant. Variants are removed. (default)	Model items are not inlined to the sliced model from the variant. The variant is retained.

See Also

Related Examples

- “Highlight Functional Dependencies” on page 8-2
- “Refine Highlighted Model” on page 8-12
- “Simplify a Standalone Model by Inlining Content” on page 8-37

Model Slicer Considerations and Limitations

When you work with the Model Slicer, consider these behaviors and limitations:

In this section...

“Model Compilation” on page 8-44

“Model Highlighting and Model Editing” on page 8-44

“Standalone Sliced Model Generation” on page 8-44

“Sliced Model Considerations” on page 8-44

“Port Attribute Considerations” on page 8-45

“Simulation Time Window Considerations” on page 8-46

“Simulation-based Sliced Model Simplifications” on page 8-46

“Model Slicer Support Limitations for Simulink Software Features” on page 8-47

“Model Slicer Support Limitations for Simulation Stepper” on page 8-47

“Model Slicer Support Limitations for Simulink Blocks” on page 8-47

“Model Slicer Support Limitations for Stateflow” on page 8-48

Model Compilation

When you open Model Slicer, the model is compiled. To avoid a compilation error, before you open Model Slicer, make sure that the model is compilable.

Model Highlighting and Model Editing

When a slice highlight is active, you cannot edit the model. You can switch to model edit mode and preserve the highlights. When you switch back to slice mode, the slice configuration is recomputed and the highlight is updated.

Standalone Sliced Model Generation

Sliced model generation requires one or more starting points for highlighting your model. Sliced model generation is not supported for:

- Forward-propagating (including bidirectional) dependencies
- Constraints
- Exclusion points present in active highlight

Sliced model generation requires a writable working folder in MATLAB.

Sliced Model Considerations

When you generate a sliced model from a model highlight, simplifying your model can change simulation behavior or prevent the sliced model from compiling. For example:

- Model simplification can change the sorted execution order in a sliced model compared to the original model, which can affect the sliced model simulation behavior.

- If you generate a sliced model containing a bus, but not the source signal of that bus, the sliced model can contain unresolved bus elements.
- If you generate a sliced model that inlines a subset of the contents of a masked block, make sure that the subsystem contents resolve to the mask parameters. If the contents and mask do not resolve, it is possible that the sliced model does not compile.
- If the source model uses a bus signal, ensure that the sliced model signals are initialized correctly. Before you create the sliced model, consider including an explicit copy of the bus signal in the source model. For example, you can include a Signal Conversion block with the **Output** option set to **Signal Copy**.
- For solver step sizes set to **auto**, Simulink calculates the maximum time step in part based on the blocks in the model. If the sliced model removes blocks that affect the time step determination, the time step of the sliced model can differ from the source model. The time step difference can cause simulation differences. Consider setting step sizes explicitly to the same values calculated in the source model.

Port Attribute Considerations


You can use blocks that the Model Slicer removes during model simplification to determine compiled attributes, such as inherited sample times, signal dimensions, and data types. The Model Slicer can change sliced model port attributes during model simplification to resolve underspecified model port attributes. If the Model Slicer cannot resolve these inconsistencies, you can resolve some model port attribute inconsistencies by:

- Explicitly specifying attributes in the source model instead of relying on propagation rules.
- Including in the sliced model the blocks that are responsible for the attribute propagation in your source model. Before you slice the model, add these blocks as additional starting points in the source model highlighting.
- Not inlining the model blocks that are responsible for model port attributes into the sliced model. For more information on model items that you can inline into the sliced model, see “Inline Content Options” on page 8-43.

Because of the way Simulink handles model references, you cannot simultaneously compile two models that both contain a model reference to the same model. When you generate a sliced model, the Model Slicer enters the **Slicer Locked (for attribute checking)** mode if these conditions are true:

- The parent model contains a referenced model.
- The highlighted portion of the parent model contains the referenced model.
- The referenced model is not inlined in the sliced model due to one of the following
 - You choose not to inline model blocks in the **Inline content options** pane of the **Model Slicer options**.
 - The Model Slicer cannot inline the referenced model. For more information on model items that Model Slicer cannot inline, see “Inline Content Options” on page 8-43.

To continue refining the highlighted portion of the parent model, you must first activate the slice

highlight mode .

Simulation Time Window Considerations

Depending on the step size of your model and the values that you enter for the start time and stop time of the simulation time window, Model Slicer might alter the actual simulation start time and stop time.

- If you enter a stop or start time that falls between time steps for your model solver, the Model Slicer instead uses a stop or start time that matches the time step previous to the value that you entered. For more information on step sizes in Simulink, see “Compare Solvers”.
- The stop time for the simulation time window cannot be greater than the total simulation time.

Simulation-based Sliced Model Simplifications

When you slice a model by using a simulation time window, some blocks in the source model, such as switch blocks, logical operator blocks, and others, can be replaced when creating the simplified standalone model. For example, a switch block that always passes one input is removed, and the active input is directly connected to the output destination. The unused input signal is also removed from the standalone model.

This table describes the blocks that the Model Slicer can replace during model simplification.

Block in Source Model	Simplification
Switch	If only one input port is active, the switch is replaced by a signal connecting the active input to the block output.
Multiport Switch	
Enabled Subsystem or Model	<p>If the subsystem or model is always enabled, remove the control input and convert to a standard subsystem or model.</p> <p>If the subsystem is never enabled, replace the subsystem with a constant value defined by the initial condition.</p>
Triggered Subsystem or Model	<p>If the subsystem or model is always triggered, remove the trigger input and convert to a standard subsystem or model.</p> <p>If the subsystem is never triggered, replace the subsystem with a constant value defined by the initial condition.</p>
Enabled and Triggered Subsystem or Model	<p>If the subsystem is always executed, convert to a standard subsystem or model</p> <p>If the subsystem is never executed, replace the subsystem with a constant value defined by the initial condition.</p>
Merge	If only one input port is active, the merge is replaced by a signal connecting the active input to the block output.

Block in Source Model	Simplification
If If Action	If only one action subsystem is active, convert to a standard subsystem or model and remove the If block.
Switch Case Switch Case Action	If only one action subsystem is active, convert to a standard subsystem or model and remove the Switch Case block.
Logical operator	Replace with constant when the block always outputs true or always outputs false. Replace the input signal with a constant if the input signal is always true or always false.

Model Slicer Support Limitations for Simulink Software Features

The Model Slicer does not support these features:

- Analysis of Simulink Test test harnesses
- Models that contain Simscape physical modeling blocks
- Models that contain algebraic loops
- Loading initial states from the source model for sliced model generation, such as data import/export entries. Define initial states explicitly for the sliced model in the sliced model configuration parameters.
- Component slicing of the subsystems and referenced models that have multiple rates.
- Component based slice generation of Function call triggered subsystems and model blocks.

Model Slicer Support Limitations for Simulation Stepper

When using Model Slicer with Simulation Stepper, the slice highlight after a Step Back may not be limited to a single step. The highlight can be influenced by the **Simulation Stepping Options > Interval between stored back steps**. For more information, see “Interval between stored back steps”.

Model Slicer Support Limitations for Simulink Blocks

The table lists the Model Slicer support limitations for Simulink Blocks.

Block	Limitation
For Each Subsystem block	The simulation impact is ignored for blocks in a For Each subsystem. Therefore, applying a simulation time window returns the same dependency analysis result as a dependency analysis that does not use a simulation time window.
MATLAB Function block	Model Slicer assumes that any output depends on all inputs in the upstream direction and any input affects all outputs in the downstream direction.

Block	Limitation
Merge block	If you generate a slice by using a simulation time window, Merge blocks are removed in the standalone model if only a single path is exercised.
Model block	<p>Model Slicer does not resolve data dependencies generated by global data store memory in Model blocks with Simulation mode set to Accelerator.</p> <p>Model Slicer does not analyze the contents within a reference to a "Reference Protected Models from Third Parties". When you slice a model that contains a protected model reference, the Model Slicer includes the entire model reference in the sliced model.</p>
S-function block	<p>Model Slicer assumes that any output depends on all inputs in the upstream direction and any input affects all outputs in the downstream direction.</p> <p>Model Slicer does not determine dependencies that result from an S-function block accessing model information dependent on a simulation time window.</p>
Simulink Functions	<p>Sliced model generation is not supported for models that contain Simulink Functions.</p> <p>The dependency propagates from a Simulink function definition to all its Function Callers.</p> <p>The dependency propagates from a Function Callers to its Simulink function definition and all the other Function Callers.</p>

Model Slicer Support Limitations for Stateflow

- When you highlight models containing a Stateflow chart or state transition table, Model Slicer assumes that any output from the Chart block or State Transition Table block depends on all inputs to the Chart block or State Transition Table block.
- When you slice a model with a Stateflow chart or a state transition table, Model Slicer does not simplify the chart or table. The chart or table is included in its entirety in the sliced model.
- If you do not "Define a Simulation Time Window" on page 8-12 when you highlight functional dependencies in a Stateflow chart or state transition table, Model Slicer assumes that all elements of the chart or table are active. Model Slicer highlights the entire contents of such charts and tables.
- When you highlight functional dependencies in a Stateflow chart or state transition table for a defined simulation time window, Model Slicer does not highlight only the states and transitions that affect the selected starting point. Instead, the Model Slicer highlights elements that are active in the time window that you specify.
- The Model Slicer does not determine dependencies between Stateflow graphical functions and function calls in other Stateflow charts.
- Graphical functions and their contents that were not active during the selected time window can potentially remain highlighted in some cases.

- Entry into states that are preempted due to events can potentially remain highlighted in some cases. For example, after a parent state is entered, an event action can exit the state and preempt entry into the child state. In such a case, the Model Slicer highlights the entry into the child state.

Activity-Based Time Slicing Considerations for Stateflow

As measured by the 'Executed Substate' decision coverage, state activity refers to these during/exit actions:

- Entry into a state does not constitute activity.
- The active time interval for a state or transition includes the moment in which the selected state exits and the subsequent state is entered.
- Indirect exits from a state or transition do not constitute activity. For example, if a state *C* exits because its parent state *P* exits, state *C* is not considered active.

For more information on decision coverage for Stateflow charts, see “Decision Coverage for Stateflow Charts” (Simulink Coverage).

When you “Highlight Active Time Intervals by Using Activity-Based Time Slicing” on page 8-30, you can select states and transitions only as activity constraints. You cannot select these Stateflow objects as constraints:

- Parallel states
- Transitions without conditions, such as unlabeled transitions which do not receive decision coverage
- States or transitions within library-linked charts
- XOR states without siblings. For example, if a state *P* has only one child state *C*, you cannot select state *C* as an activity constraints because state *P* does not receive decision coverage for the executed substate

See Also

“Algebraic Loop Concepts” | “Solver Pane”

Using Model Slicer with Stateflow

In this section...

“Model Slicer Highlighting Behavior for Stateflow Elements” on page 8-50

“Using Model Slicer with Stateflow State Transition Tables” on page 8-50

“Support Limitations for Using Model Slicer with Stateflow” on page 8-50

You can use Model Slicer highlighting to visually verify the logic in your Stateflow charts or tables. After you “Define a Simulation Time Window” on page 8-12, you use Model Slicer to highlight and slice Stateflow elements that are active within the selected time window.

Note If you do not “Define a Simulation Time Window” on page 8-12 when you highlight functional dependencies in a Stateflow chart or table, Model Slicer assumes that all elements of the chart or table are active. Model Slicer highlights the entire contents of such charts and tables.

In this section...

“Model Slicer Highlighting Behavior for Stateflow Elements” on page 8-50

“Using Model Slicer with Stateflow State Transition Tables” on page 8-50

“Support Limitations for Using Model Slicer with Stateflow” on page 8-50

Model Slicer Highlighting Behavior for Stateflow Elements

Model Slicer highlights a Stateflow element if it was executed in the specified time window. Some examples include:

- A chart, if it is activated in the specified a time window.
- A state, if its entry, exit, or during actions are executed in the specified a time window.
- A parent state, if its child state is highlighted in the specified a time window.
- A transition, if it is taken in the specified time window, such as inner, outer, and default. If the conditions of a transition are evaluated, but the transition is not taken, Model Slicer does not highlight the transition.

Using Model Slicer with Stateflow State Transition Tables

Model Slicer does not directly highlight the contents of Stateflow state transition tables. To view highlighted functional dependencies in a state transition table, you must view the auto-generated diagram for the state transition table. In the **Debug** tab, click **Show Auto Chart**. For more information, see “Inspect the Design of State Transition Tables” (Stateflow).

Support Limitations for Using Model Slicer with Stateflow

For support limitations when you use Model Slicer with Stateflow, see “Model Slicer Support Limitations for Stateflow” on page 8-48.

See Also

More About

- “Highlight Functional Dependencies” on page 8-2
- “Refine Highlighted Model” on page 8-12
- “Chart Simulation Semantics” (Stateflow)

Isolating Dependencies of an Actuator Subsystem

This example demonstrates highlighting model items that a subsystem depends on. It also demonstrates generating a standalone model slice from the model highlight.

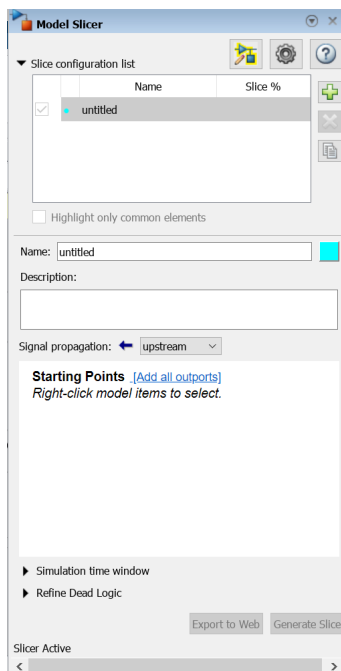
In this section...


“Choose Starting Points and Direction” on page 8-52

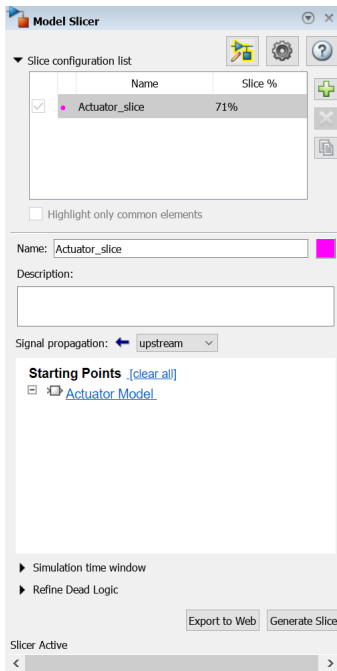
“View Precedents and Generate Model Slice” on page 8-53

Choose Starting Points and Direction

- 1 Open the f14 example model.
f14
- 2 On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

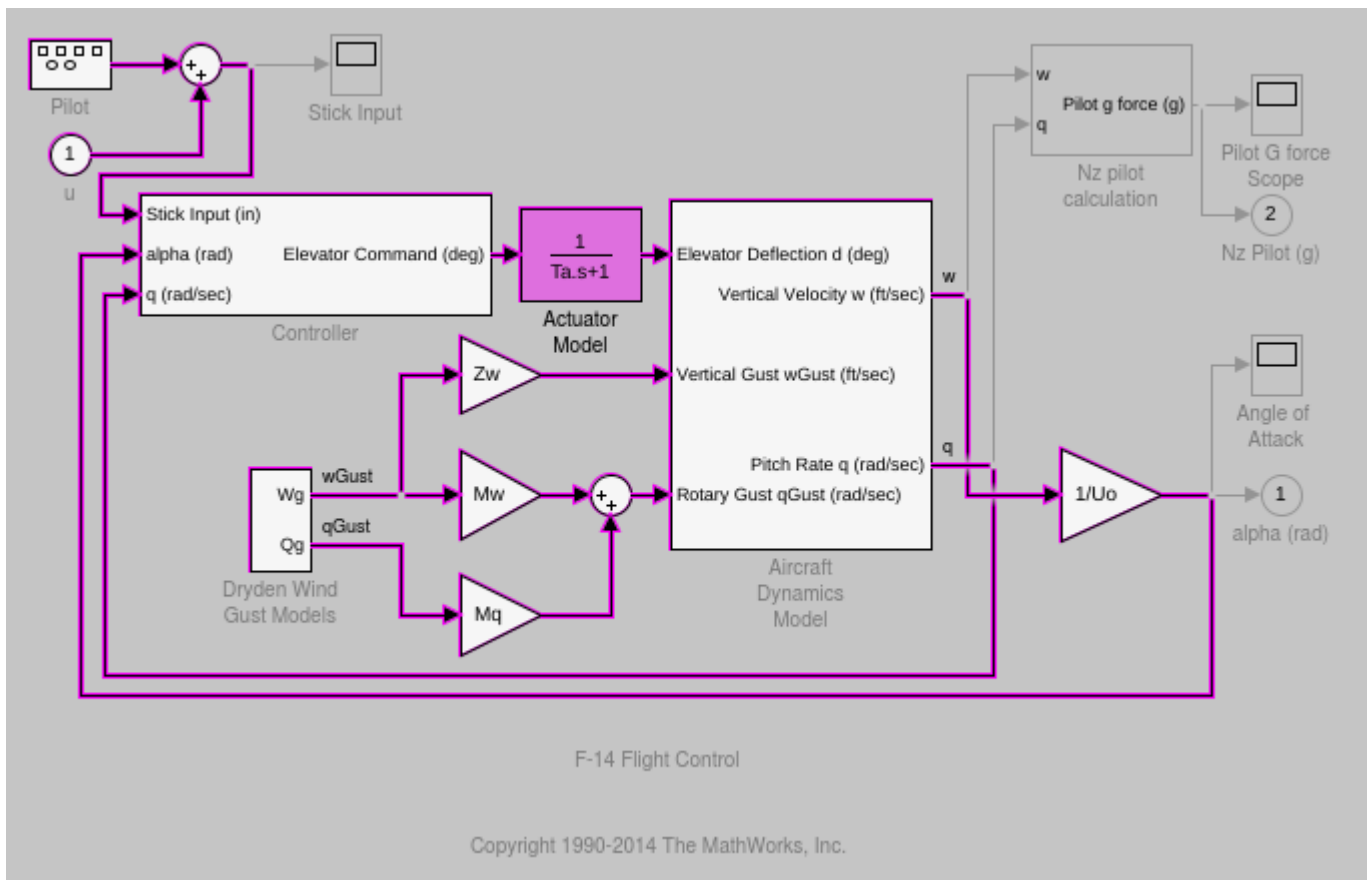


- 3 In the Model Slicer, click the arrow to expand the **Slice configuration list** list. Set the slice properties:
 - **Name:** Actuator_slice
 - To the right of **Name**, click the colored square to set the highlight color. Choose magenta  from the palette.
 - **Signal Propagation:** upstream.
- 4 Add the Actuator Model subsystem as a starting point. In the model, right-click the Actuator Model subsystem and select **Model Slicer > Add as Starting Point**.



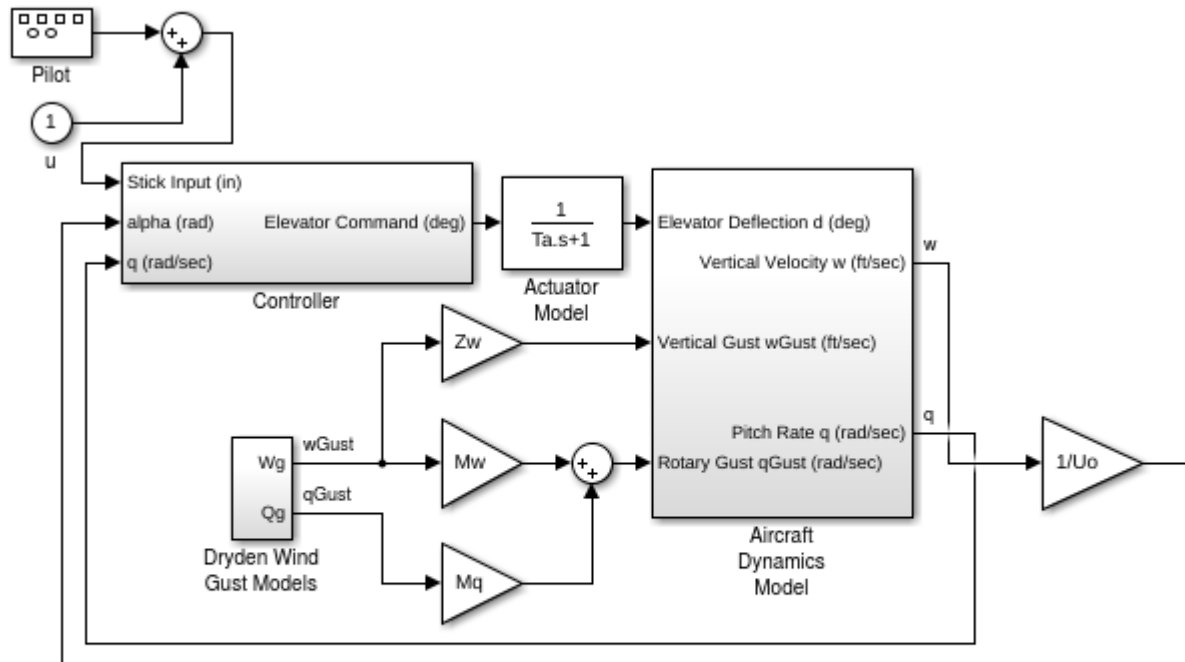
View Precedents and Generate Model Slice

- 1 The model highlights the upstream dependencies of the Actuator Model subsystem.



Trace the following dependency path. Aircraft Dynamics Model is highlighted via the Pitch Rate q signal, which is an input to Controller, the output of which feeds Actuator Model.

- 2 Generate a standalone model containing the highlighted model items:
 - a In the Model Slicer, click **Generate slice**.
 - b In the **Select File to Write** dialog box, select the save location and enter `actuator_slice_model`.
 - c Click **Save**.
- 3 The sliced model contains the highlighted model items.



F-14 Flight Control

Copyright 1990-2014 The MathWorks, Inc.

- 4 To remove highlighting from the model, close the Model Slicer.

Isolate Model Components for Functional Testing

You can create a standalone model for the model designed using “Design Model Architecture”. The model slice isolates the model components and relevant signals for debugging and refinement.

Isolate Subsystems for Functional Testing

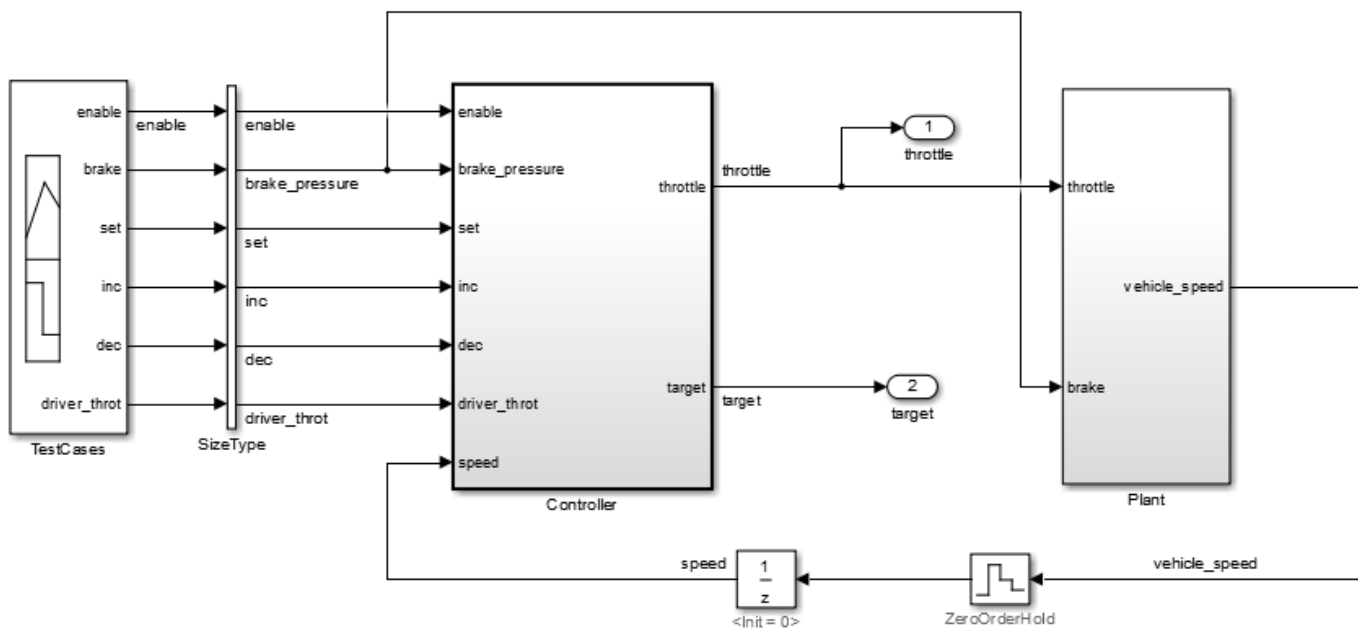
To debug and refine a subsystem of your model, create a standalone model. The standalone model isolates the subsystem and relevant signals. You can observe the subsystem behavior without simulating the entire source model.

Note You cannot slice virtual subsystems. To isolate a virtual subsystem, first convert it to an atomic subsystem.

Isolate a Subsystem with Simulation-Based Inputs

To observe the simulation behavior of a subsystem, include logged signal inputs in the standalone model. When you configure the model slice, specify a simulation time window. For large models, observing subsystem behavior in a separate model can save time compared to compiling and running the entire source model.

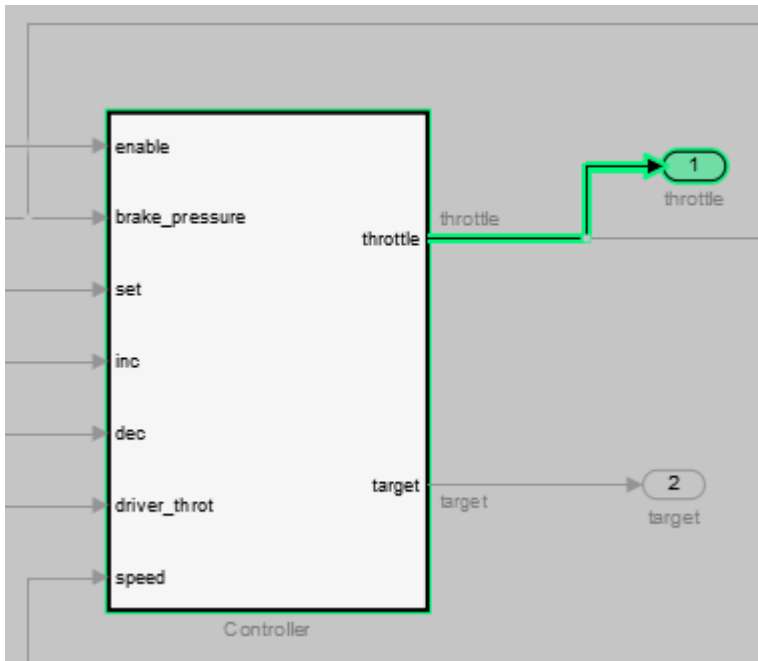
This example shows how to include simulation effects for the Controller subsystem of a cruise control system.




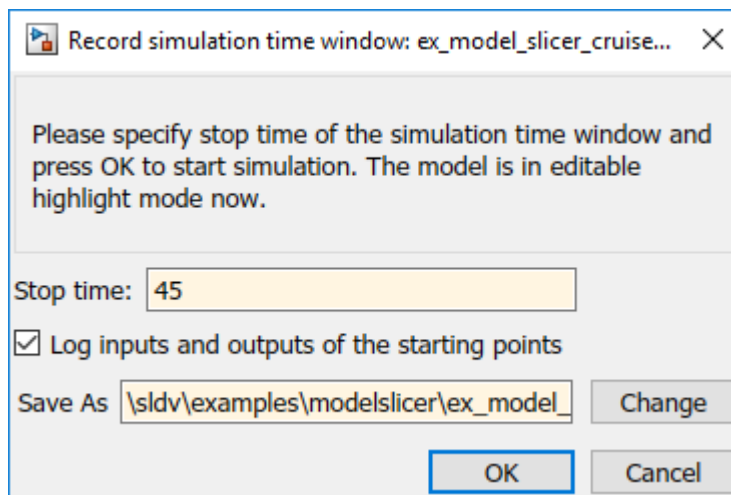
- 1 To open the Model Slicer, on the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

- 2 To select the starting point for dependency analysis, right-click a block, signal, or a port, and select **Model Slicer > Add as Starting point**.
- 3 To isolate the subsystem in the sliced model, right-click the subsystem, and select **Model Slicer > Slice component**.

In the example model, selecting **Slice component** for the Controller subsystem limits the dependency analysis to the path between the starting point (the throttle output) and the Controller subsystem.



- 4 To specify the simulation time window:
 - a In the Model Slicer, select **Simulation time window**.
 - b Click the run simulation button .
 - c Enter the simulation stop time, and click **OK**.

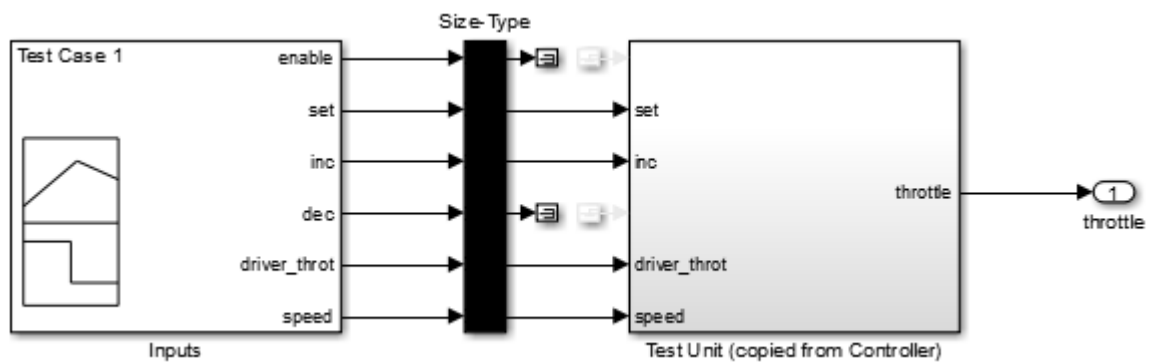


The Model slicer analyzes the model dependencies for the simulation interval.

- 5 To extract the subsystem and logged signals, click **Generate slice**. Enter a file name for the sliced model.

Based on the dependency analysis, a Signal Builder block supplies the signal inputs to the subsystem.

In the sliced model shown, the sliced model Signal Builder block contains one test case representing the signal inputs to the Controller subsystem for simulation time 0–45 seconds.



Isolate Referenced Model for Functional Testing

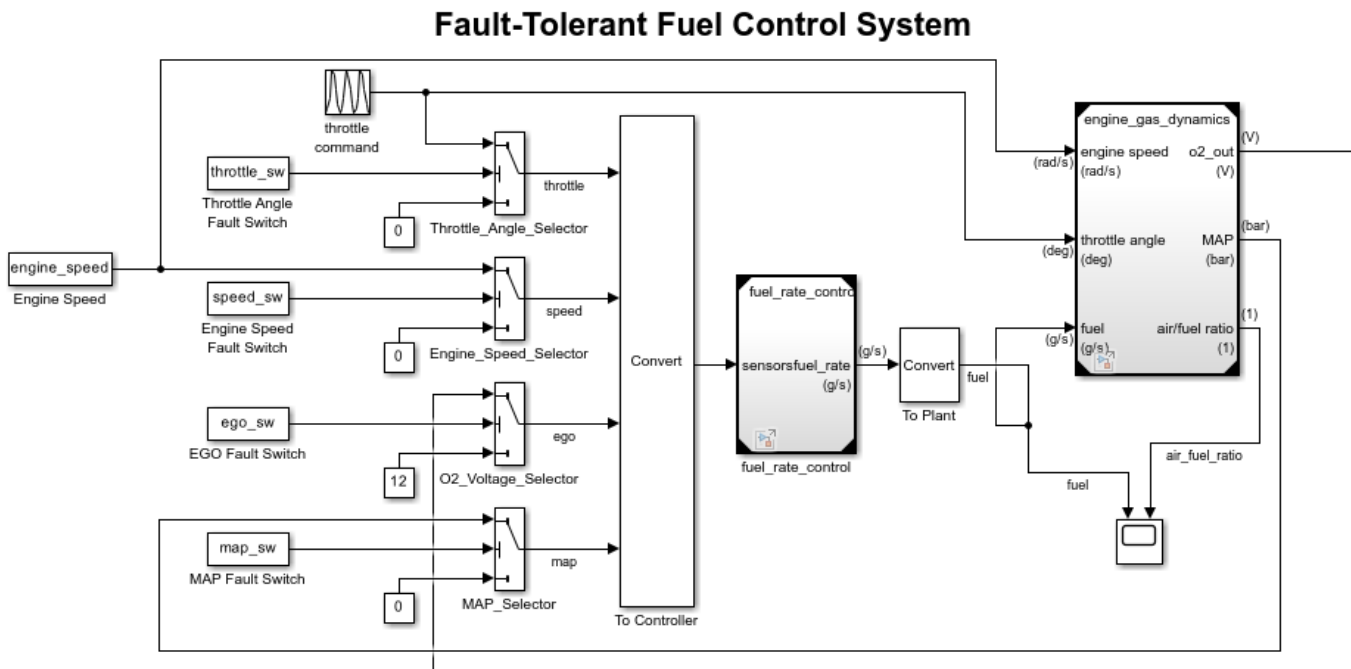
To functionally test a referenced model, you can create a slice of a referenced model treating it as an open-loop model. You can isolate the simplified open-loop referenced model with the inputs generated by simulating the close-loop system.

This example shows how to slice the referenced model controller of a fault-tolerant fuel control system for functional testing. To create a simplified open-loop referenced model for debugging and refinement, you generate a slice of the referenced controller.

Step 1: Open the Model

The fault-tolerant fuel control system model contains a referenced model controller `fuel_rate_control`.

```
open_system('sldvSlicerdemo_fuelsys');
```



Copyright 1990-2017 The MathWorks, Inc.

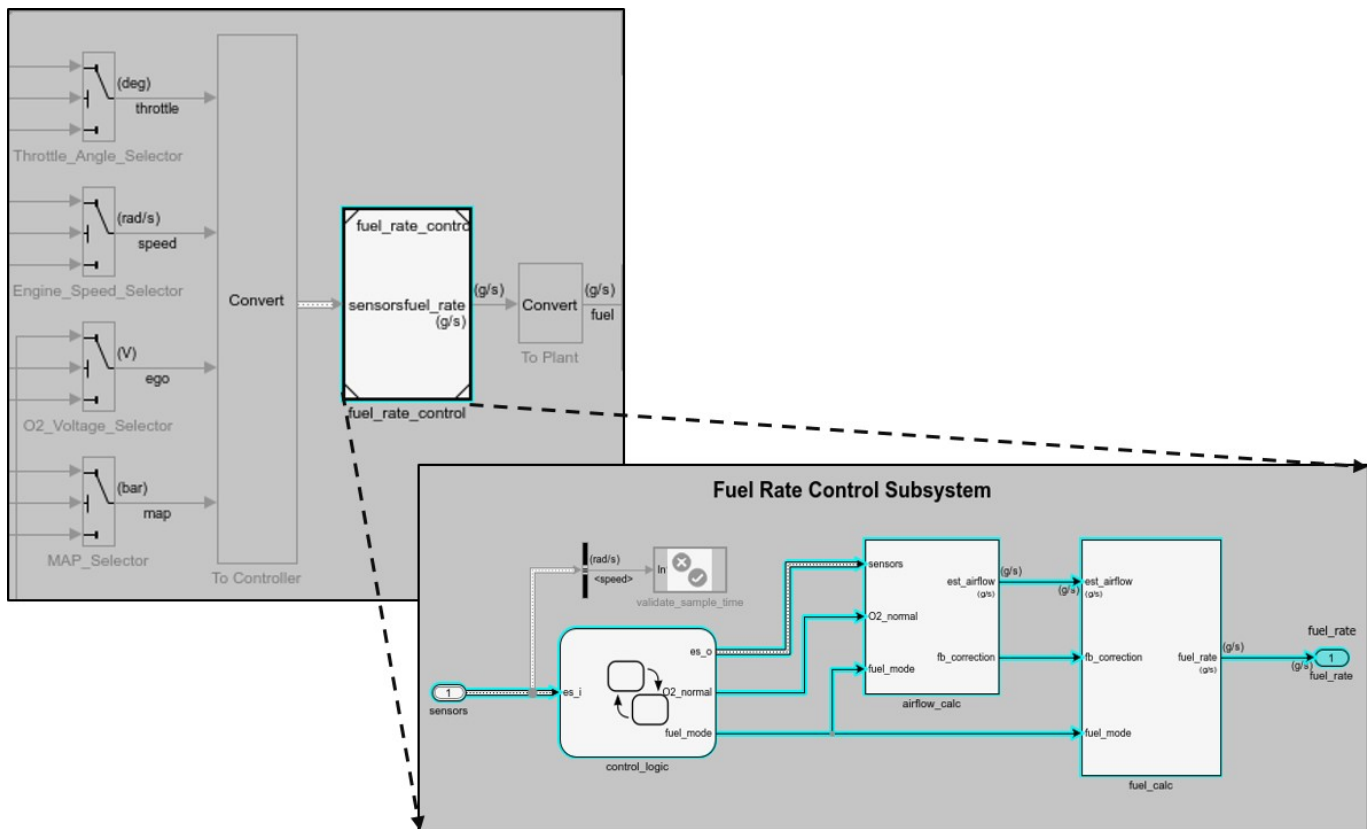
Step 2: Slice the Referenced Model

To analyze the `fuel_rate_control` referenced model, you slice it to create a standalone open-loop model. To open the Model Slice Manager, select **Apps > Model Verification, Validation, and Test > Model Slicer**, or right-click the `fuel_rate_control` model and select **Model Slicer > Slice component**. When you open the Model Slice Manager, the Model Slicer compiles the model. You then configure the model slice properties.

Note: The simulation mode of the `sldvSlicerdemo_fuelsys` model is Accelerator mode. When you slice the referenced model, the software configures the simulation mode to Normal mode and sets it back to its original simulation mode while exiting the Model Slicer.

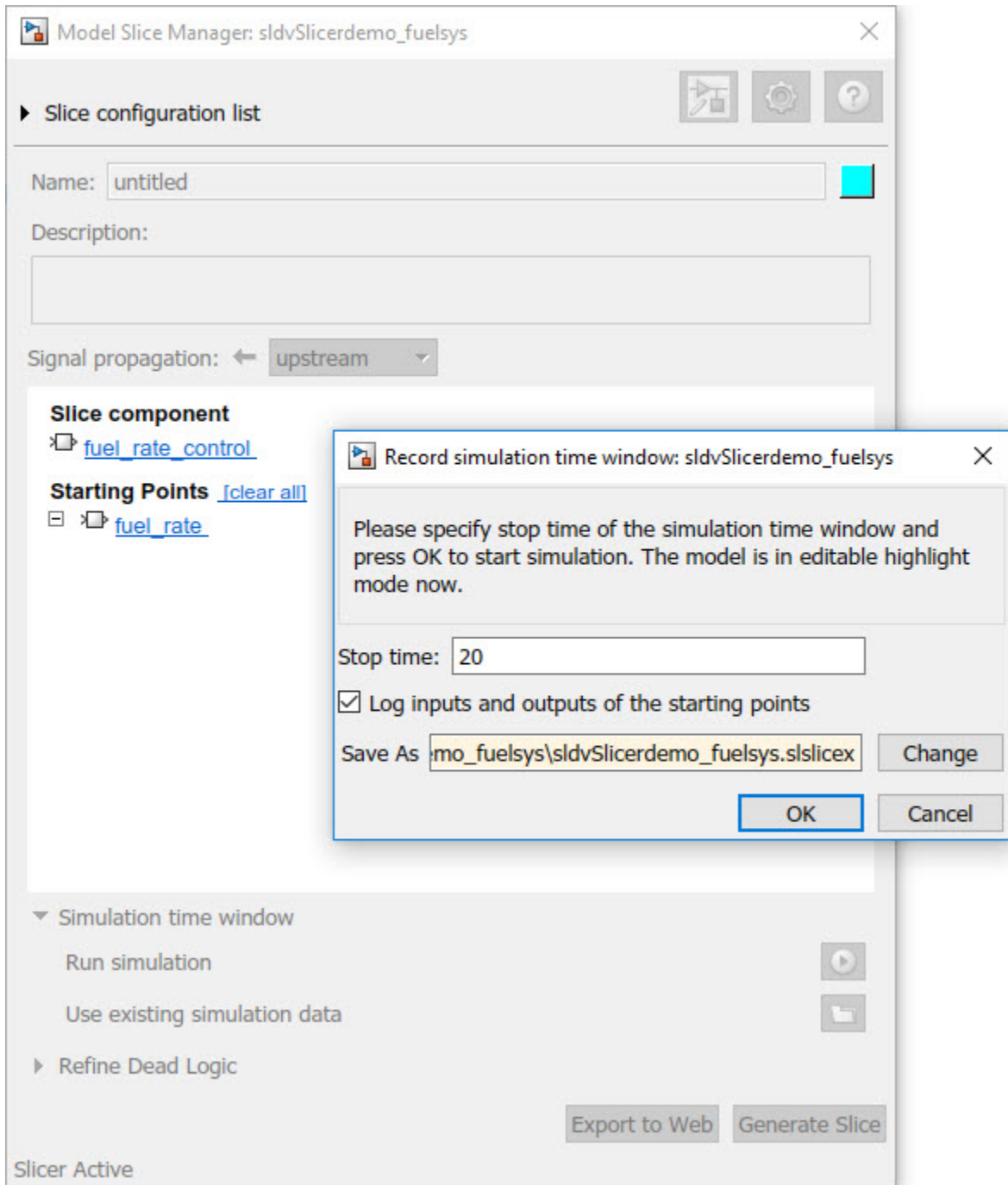
Step 3: Select Starting Point

Open the `fuel_rate_control` model, right-click the `fuel` - rate port, and select **Model Slicer > Add as starting point**. The Model Slicer highlights the upstream constructs that affect the `fuel_rate`.

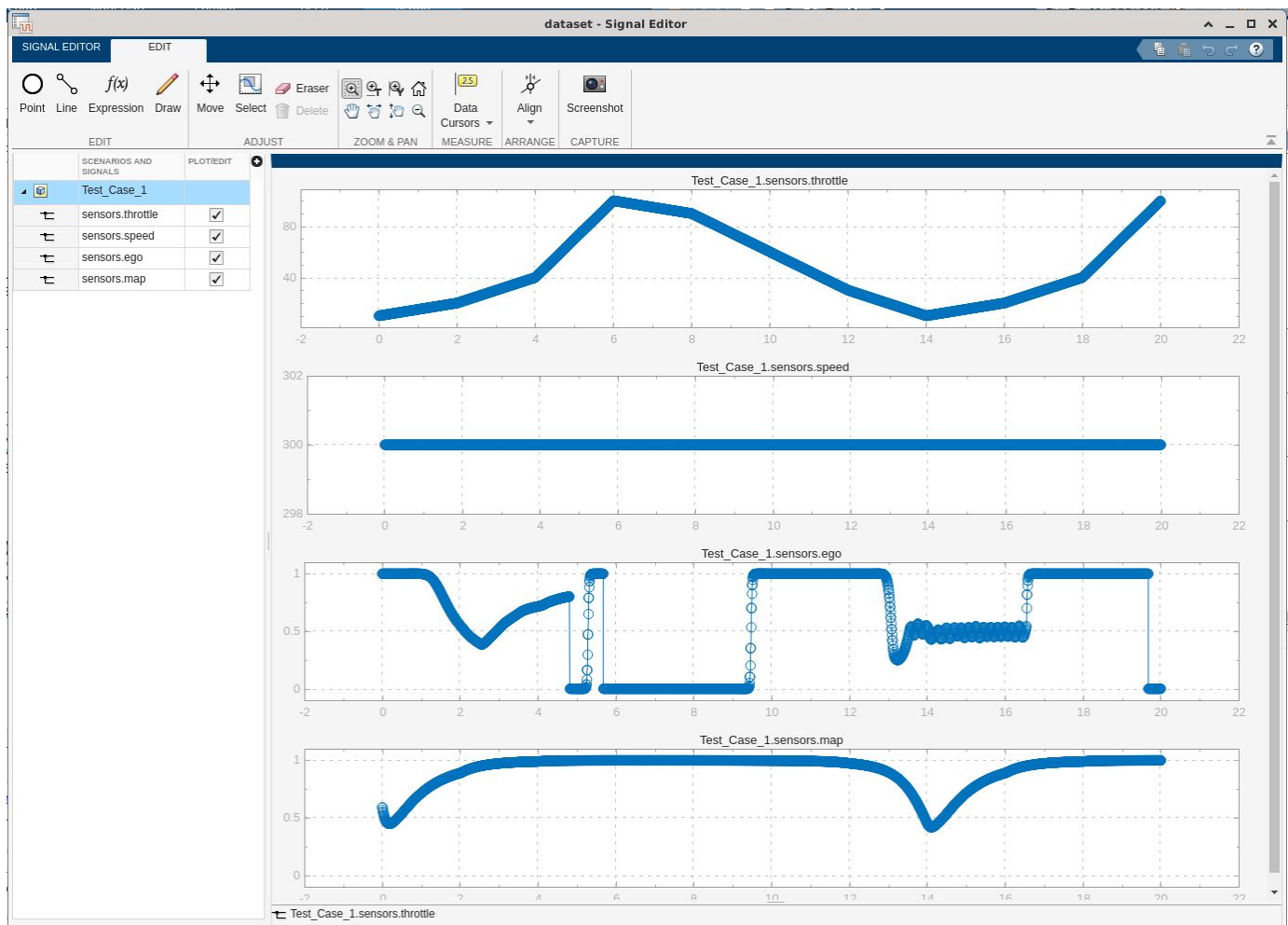
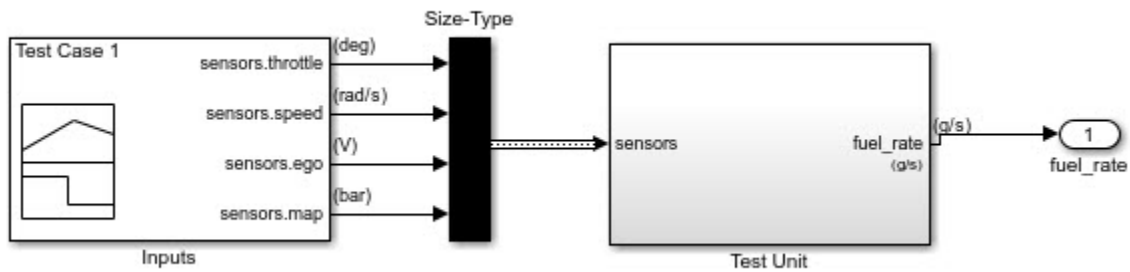


Step 4: Generate Slice

- In the Model Slice Manager dialog box, select the **Simulation time window**.
- Click **Run simulation**.
- For the **Stop time**, enter 20. Click **OK**.
- Click **Generate Slice**. The software simulates the sliced referenced model by using the inputs of the close-loop `slvSlicerdemo_fuelsys` model.



For the sliced model, in the Signal Editor window, one test case is displayed that represents the signals input to the referenced model for simulation time 0-20 seconds.




See Also

“Model Slicer Considerations and Limitations” on page 8-44 | “Highlight Functional Dependencies” on page 8-2



Refine Highlighted Model by Using Existing .slicex or Dead Logic Results

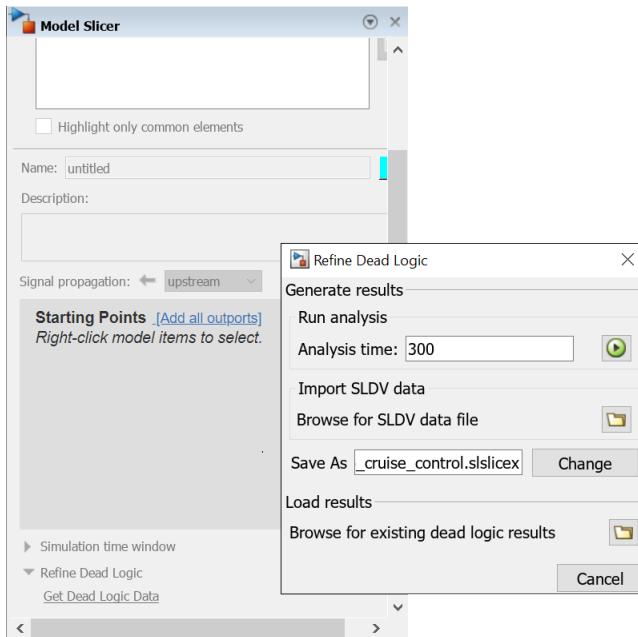
When you run simulation or refine dead logic, Model Slicer saves your simulation results at the default location `<current_folder>\modelslicer\<<model_name>\<model_name>.slicex`. For large or complex models, the simulation time can be lengthy. To refine the highlighted slice, you can use the existing Model Slicer simulation data or dead logic results.

If you want to highlight functional dependencies in the model again at another time, you can use the existing .slicex simulation time window data without needing to resimulate the model. Model Slicer then uses the existing simulation data to highlight the model.

- 1 Open the Simulink model.
- 2 To open the Model Slicer, On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.
- 3 Select **Simulation time window**.
- 4 Click **Use existing simulation data** .
- 5 Navigate to the existing .slicex data and click **Open**.

To refine the dead logic for dependency analysis, you can import the existing Simulink Design Verifier data file or use the existing .slicex dead logic results. For more information see, “Dead Logic Detection” (Simulink Design Verifier) and “Manage Simulink Design Verifier Data Files” (Simulink Design Verifier).

- 1 In Model Slicer, select **Refine Dead Logic** and click **Get Dead Logic Data**.
- 2 To import the Simulink Design Verifier data file, click **Browse for SLDV data file** .
To load the existing dead logic results, click **Browse for existing dead logic results** .
- 3 Navigate to the existing data and click **Open**.



See Also

More About

- “Highlight Functional Dependencies” on page 8-2
- “Configure Model Highlight and Sliced Models” on page 8-41
- “Refine Dead Logic for Dependency Analysis” on page 8-23

Simplification of Variant Systems

In this section...

“Use the Variant Reducer to Simplify Variant Systems” on page 8-65

“Use Model Slicer to Simplify Variant Systems” on page 8-65

If your model contains “Variant Systems”, you can reduce the model to a simplified, standalone model containing only selected variant configurations.

Use the Variant Reducer to Simplify Variant Systems

After you Add and Validate Variant Configurations, you can reduce the model from the Variant Manager:

- 1 Open a model containing at least one valid variant configuration.
- 2 Right-click a variant system and select **Variant >> Open in Variant Manager**.
- 3 Click **Reduce model...**
- 4 Select one or more variant configurations.
- 5 Set the **Output directory**.
- 6 Click **Reduce** to create a simplified, standalone model containing only the selected variant configurations.

The Variant Reducer creates a simplified, standalone model in the output directory you specified containing only the variant configurations you selected. For more information, see “Reduce Variant Models Using Variant Reducer”.

Use Model Slicer to Simplify Variant Systems

After you Add and Validate Variant Configurations, you can use Model Slicer to create a simplified, standalone model containing only the active variant configuration. When you “Highlight Functional Dependencies” on page 8-2 in a model containing variant systems, only active variant choices are highlighted. When you “Create a Simplified Standalone Model” on page 8-29 from a model highlight that includes variant systems, Model Slicer removes the variant systems and replaces them with the active variant configurations.

For instructions on how to change the active variant configuration and how to set default variant choices, see “Working with Variant Choices”.

See Also

More About

- “Create a Simple Variant Model”
- “Implement Variations in Separate Hierarchy Using Variant Subsystems”
- “Introduction to Variant Controls”

- “Reduce Variant Models Using Variant Reducer”

Programmatically Resolve Unexpected Behavior in a Model with Model Slicer

In this example, you evaluate a Simulink® model, detect unexpected behavior, and use Model Slicer to programmatically isolate and resolve the unexpected behavior. When you plan to reuse your API commands and extend their use to other models, a programmatic approach is useful.

Prerequisites

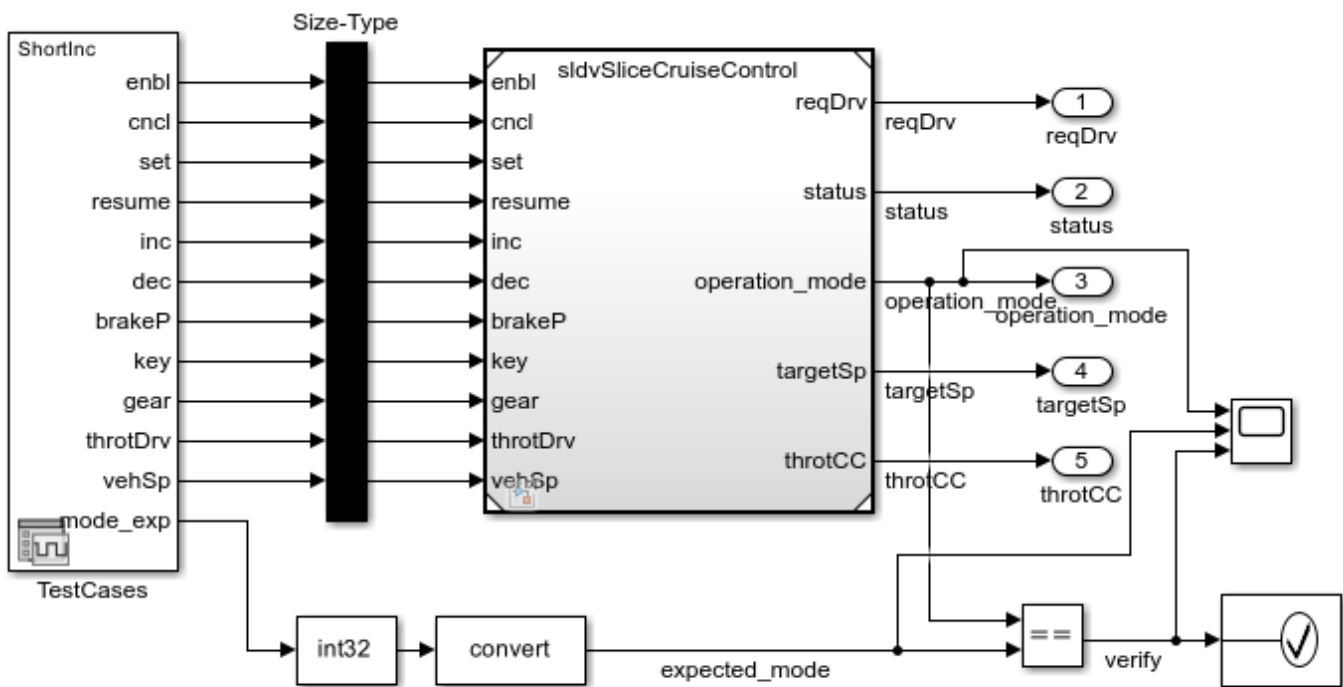
Be familiar with the behavior and purpose of Model Slicer and the functionality of the Model Slicer API. “Highlight Functional Dependencies” on page 8-2 outlines how to use Model Slicer user interface to explore models. The `slslicer`, `slsliceroptions`, and `slslicertrace` function reference pages contain the Model Slicer API command help.

Find the Area of the Model Responsible for Unexpected Behavior

The `sldvSliceCruiseControlHarness` test harness model contains a cruise controller subsystem `sldvSliceCruiseControl` and a block, **TestCases**, containing a test case for this subsystem. You first simulate the model to execute the test case. You then evaluate the behavior of the model to find and isolate areas of the model responsible for unexpected behavior:

1. Open the `sldvSliceCruiseControlHarness` test harness for the cruise control model.

```
open_system('sldvSliceCruiseControlHarness')
```



Note: The Assertion block is set to **Stop simulation when assertion fails** when the actual operation mode is not the same as the expected operation mode.

The TestCases block contains several test inputs for sldvSliceCruiseControl.

2. In the TestCases Signal Editor click the **Run all** button to run all of the included test cases. You receive an error during the ResumeWO test case. The Assertion block halted simulation at 27 seconds, because the actual operation mode was not the same as the expected operation mode. Click **OK** to close this error message.

3. In the sldvSliceCruiseControlHarness model, double-click the Assertion block, clear **Enable assertion**, and click **OK**.

```
set_param('sldvSliceCruiseControlHarness/Assertion','Enabled','off')
```

4. Set the **Active Group** of the TestCases Signal Editor block to the test case containing the error and run the simulation again.

```
set_param('sldvSliceCruiseControlHarness/TestCases','ActiveScenario','ResumeWO')
sim('sldvSliceCruiseControlHarness')
```

The Scope block in the model contains three signals:

- *operation_mode* - displays the actual operation mode of the subsystem.
- *expected_mode* - displays the expected operation mode of the subsystem that the test case provides.
- *verify* - displays a Boolean value comparing the operation mode and the expected mode.

The scope shows a disparity between the expected operation mode and the actual operation mode beginning at time **27**. Now that you know the output displaying the unexpected behavior and the time window containing the unexpected behavior, use Model Slicer to isolate and analyze the unexpected behavior.

Isolate the Area of the Model Responsible for Unexpected Behavior

1. Create a Model Slicer configuration object for the model using `slslicer`. The Command Window displays the slice properties for this Model Slicer configuration.

```
obj = slslicer('sldvSliceCruiseControlHarness')
```

```
obj =
```

```
SLSlicer with properties:
```

```
Configuration: [1x1 SLSlicerAPI.SLSlicerConfig]
ActiveConfig: 1
DisplayedConfig: []
StorageOptions: [1x1 struct]
AnalysisOptions: [1x1 struct]
SliceOptions: [1x1 struct]
InlineOptions: [1x1 struct]
```

```
Contents of active configuration:
    Name: 'untitled'
    Description: ''
```

```

        Color: [0 1 1]
SignalPropagation: 'upstream'
    StartingPoint: [1x0 struct]
    ExclusionPoint: [1x0 struct]
        Constraint: [1x0 struct]
SliceComponent: [1x0 struct]
    UseTimeWindow: 0
    CoverageFile: ''
    UseDeadLogic: 0
    DeadLogicFile: ''
    
```

2. Activate the slice highlighting mode of Model Slicer to compile the model and prepare it for dependency analysis.

```
activate(obj);
```

Consider turning on [matlab:helpview\(fullfile\(docroot, 'simulink/ug/fast-restart-workflow'\)\)](matlab:helpview(fullfile(docroot, 'simulink/ug/fast-restart-workflow')))

3. Add the **operation_mode** outpost block as a starting point and highlight it.

```
addStartingPoint(obj, 'sldvSliceCruiseControlHarness/operation_mode')
highlight(obj)
```

The area of the model upstream of the starting point and active during simulation is highlighted.

4. Simulate the model within a restricted simulation time window (maximum 30 seconds) to highlight only the area of the model upstream of the starting point and active during the time window of interest.

```
simulate(obj, 0, 30)
```

Only the portion of the model upstream of the starting point and active during the simulation time window is highlighted.

5. You can further narrow the simulation time window by changing the start time to 20 seconds.

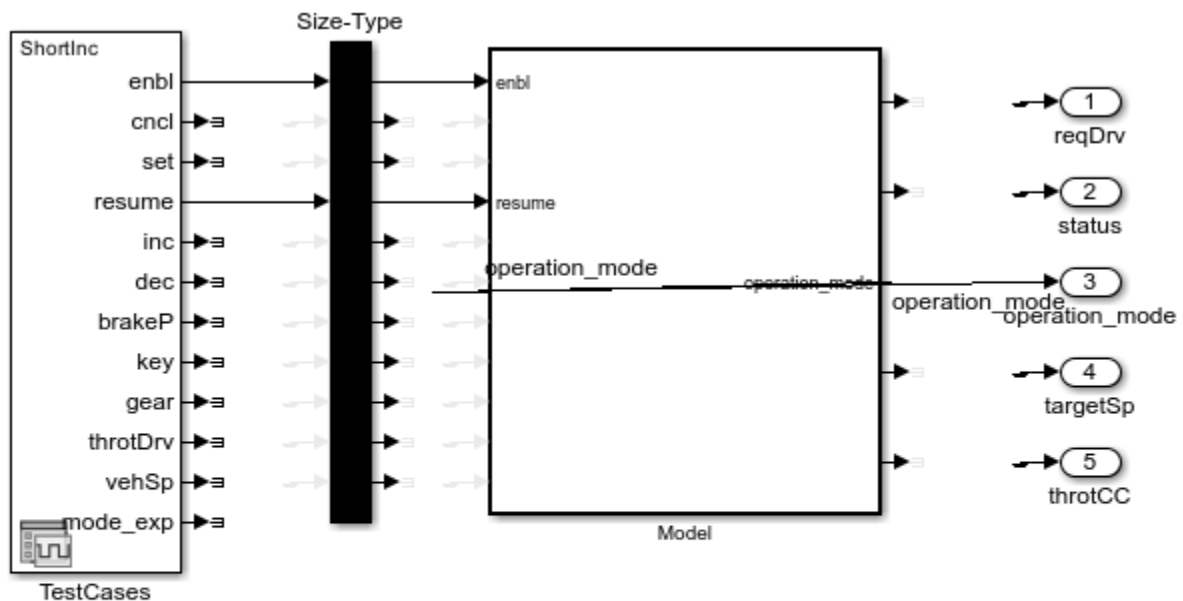
```
setTimeWindow(obj, 20, 30)
```

6. Create a sliced model **sldvSliceCruiseControlHarness_sliced** containing only the area of interest.

```
slicedModel = slice(obj, 'sldvSliceCruiseControlHarness_sliced')
open_system('sldvSliceCruiseControlHarness_sliced')
```

```
slicedModel =
```

```
    'sldvSliceCruiseControlHarness_sliced'
```



Copyright 2015 The MathWorks, Inc.

The sliced model **sldvSliceCruiseControlHarness_sliced** now contains a simplified version of the source model **sldvSliceCruiseControlHarness**. The simplified standalone model contains only those parts of the model that are upstream of the specified starting point and active during the time window of interest.

Investigate the Sliced Model and Debug the Source Model

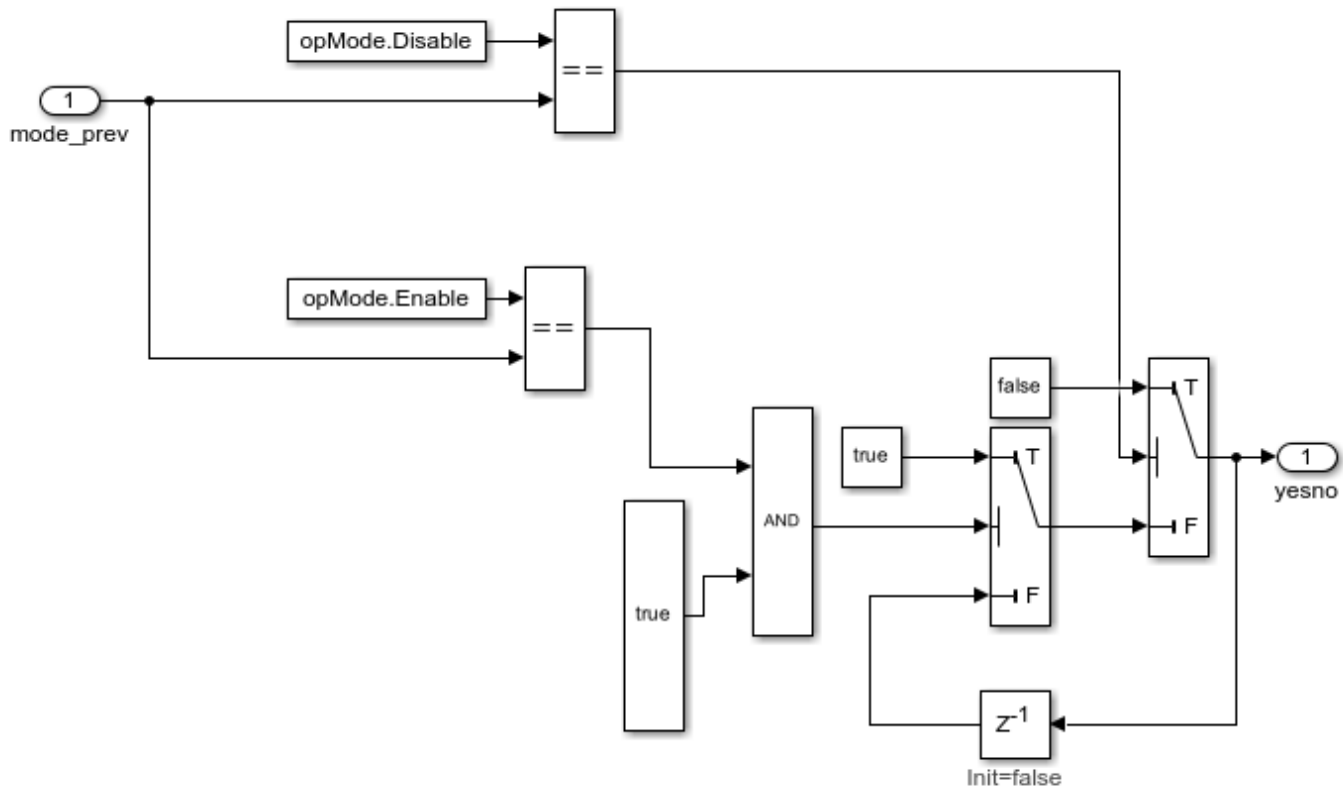
You can now debug the unexpected behavior in the simplified standalone model and then apply changes to the source model.

1. To enable editing the model again, terminate the Model Slicer mode.

```
terminate(obj)
```

2. Navigate to the area of the sliced model that contains the unexpected behavior.

```
open_system('sldvSliceCruiseControlHarness_sliced/Model/CruiseControlMode/opMode/resumeCondition')
```



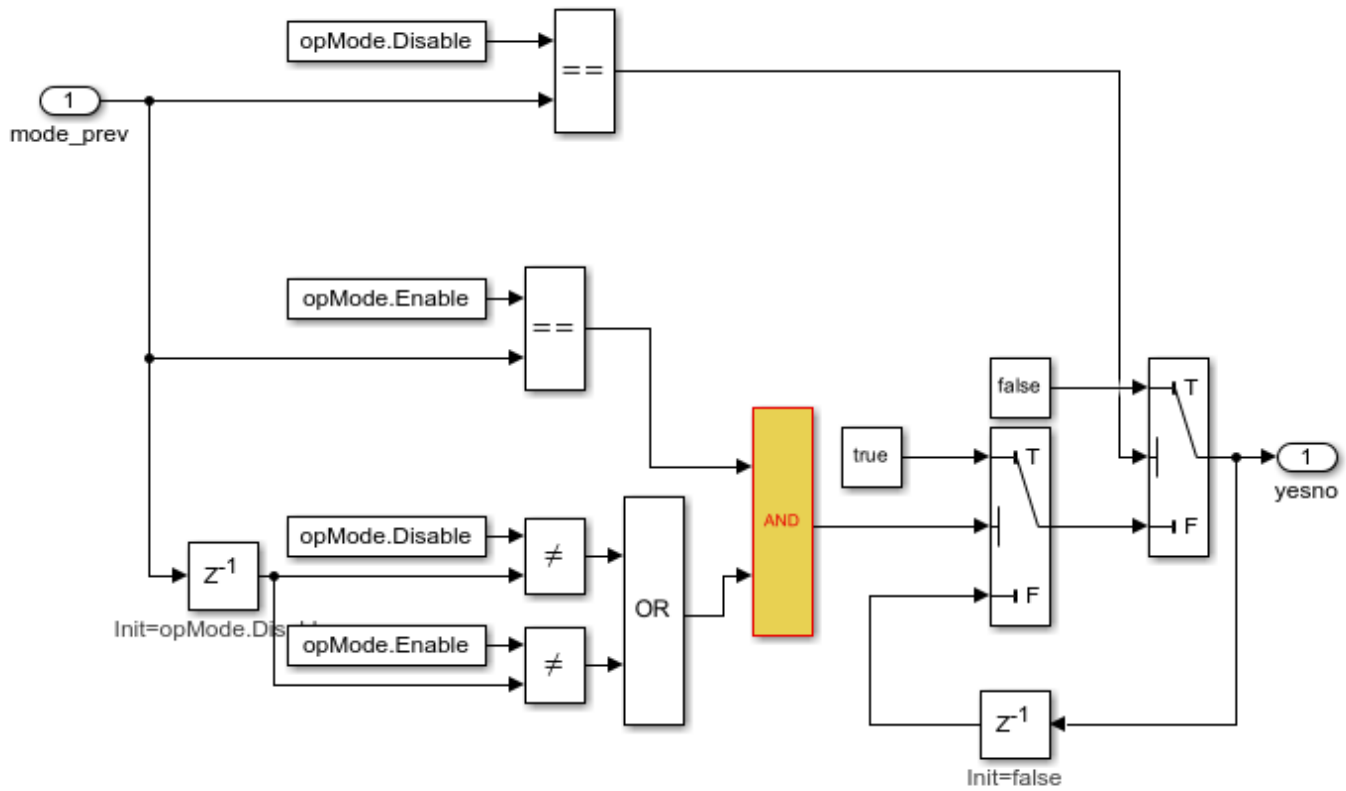
The *AND* Logical Operator block in this subsystem has a truncated *true* constant attached to its second input port. This *true* constant indicates that the second input port is always *true* during the restricted time window for this sliced model, causing the cruise control system not to enter the "has canceled" state.

3. Navigate to the equivalent *AND* Logical Operator block in the source system by using `slslicertrace` to view the blocks connected to the second input port.

```
h = slslicertrace('SOURCE',...
    'sldvSliceCruiseControlHarness_sliced/Model/CruiseControlMode/opMode/resumeCondition/hasCanceled',...
    hilite_system(h))
```

h =

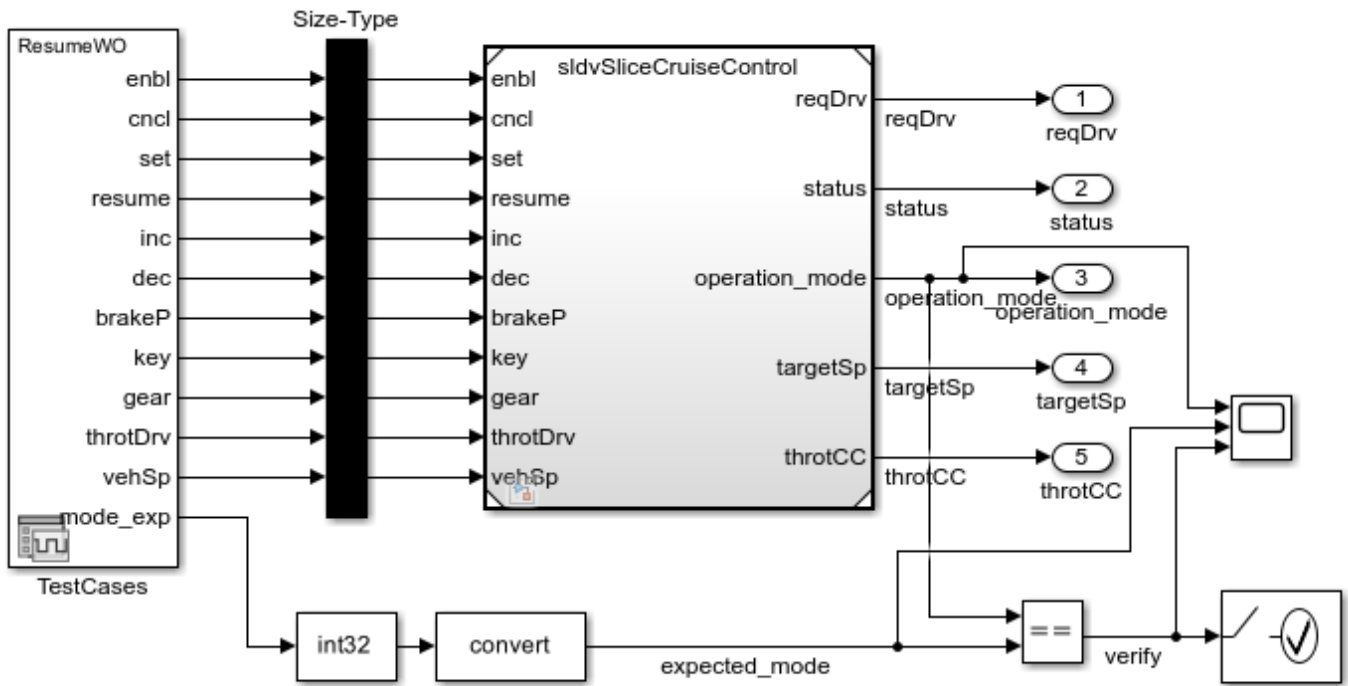
1.2100e+03



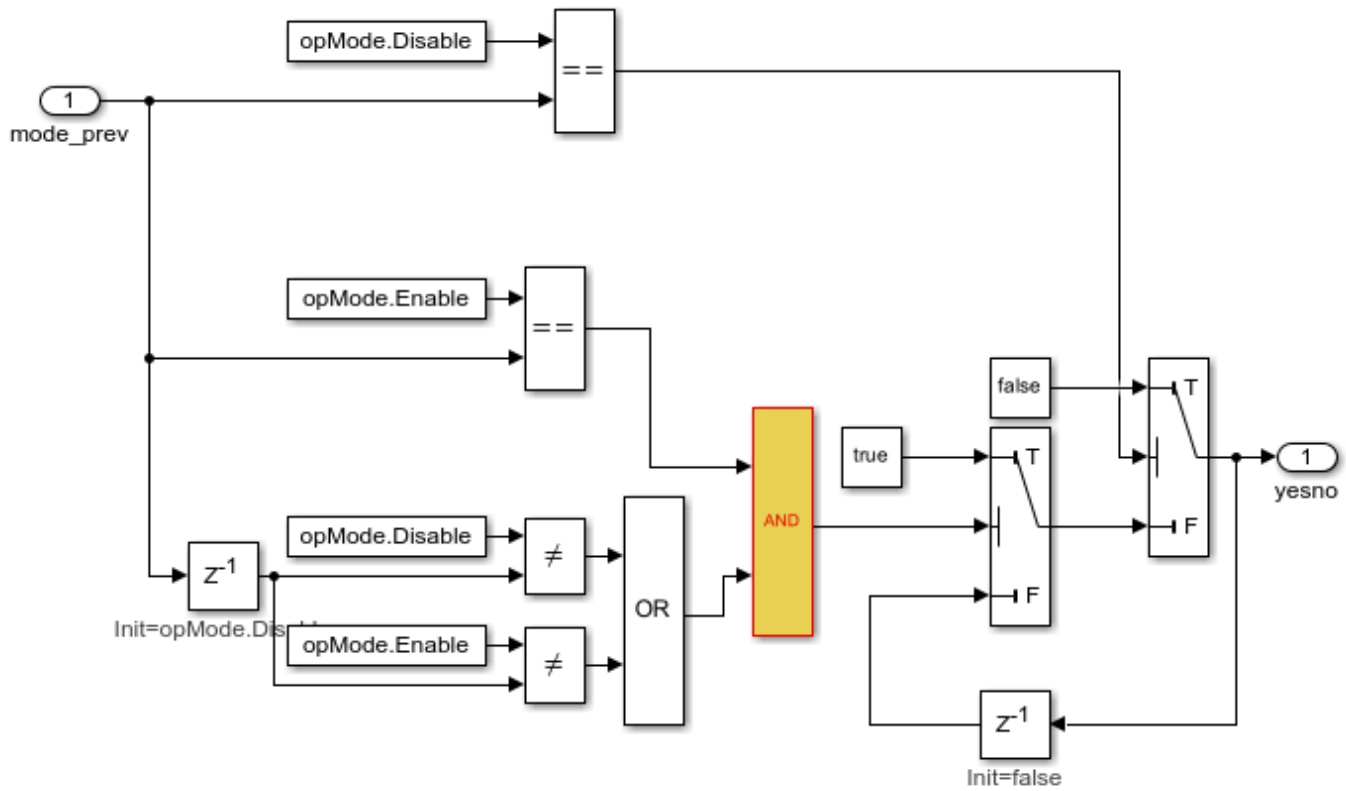
The *OR* Logical Operator block in this subsystem is always *true* in the current configuration. Changing the *OR* Logical Operator block to an *AND* Logical Operator block rectifies this error.

4. Before making edits, create new copies of the cruise control model and the test harness model.

```
save_system('sldvSliceCruiseControl', 'sldvSliceCruiseControl_fixed')
save_system('sldvSliceCruiseControlHarness', 'sldvSliceCruiseControlHarness_fixed')
```

Copyright 2015 The MathWorks, Inc.



5. Update the model reference in the test harness to refer to the newly saved model.

```
set_param('sldvSliceCruiseControlHarness_fixed/Model', ...
    'modelNameDialog', 'sldvSliceCruiseControl_fixed.slx')
```

6. Use the block path of the erroneous Logical Operator block to fix the error.

```
set_param('sldvSliceCruiseControl_fixed/CruiseControlMode/opMode/resumeCondition/hasCanceled/Logi
```

7. Simulate the test harness for 45 seconds with the fixed model to confirm the corrected behavior.

```
sim('sldvSliceCruiseControlHarness_fixed')
```

The scope now shows that the expected operation mode is the same as the actual operation mode.

Clean Up

To complete the demo, save and close all models and remove the Model Slicer configuration object.

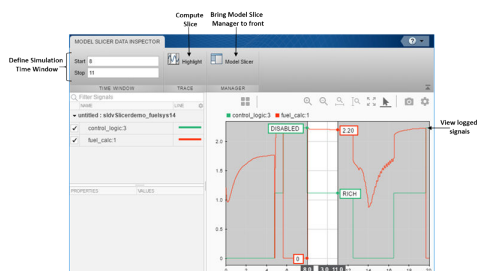
```
save_system('sldvSliceCruiseControl_fixed')
save_system('sldvSliceCruiseControlHarness_fixed')
close_system('sldvSliceCruiseControl_fixed')
close_system('sldvSliceCruiseControlHarness_fixed')
close_system('sldvSliceCruiseControlHarness_sliced')
clear obj
```

Refine Highlighted Model Slice by Using Model Slicer Data Inspector

Using the Model Slicer Data Inspector, you can inspect logged signals and refine the highlighted model slice. To refine the highlighted model slice, select the time window in the graphical plot by using data cursors.

In the Model Slicer Data Inspector, you can:

- View signals — Inspect logged signal data after model simulation. See “Inspect Simulation Data”.
- Select simulation time window — Define simulation time window by using data cursors in the graphical plot or by defining the **Start** and **Stop** time in the Inspector.
- Highlight — Compute a slice for the defined simulation time window. See “Highlight Functional Dependencies” on page 8-2.



Investigate Highlighted Model Slice by Using Model Slicer Data Inspector

This example shows how to investigate and refine the highlighted model slice by using the Model Slicer Data Inspector.

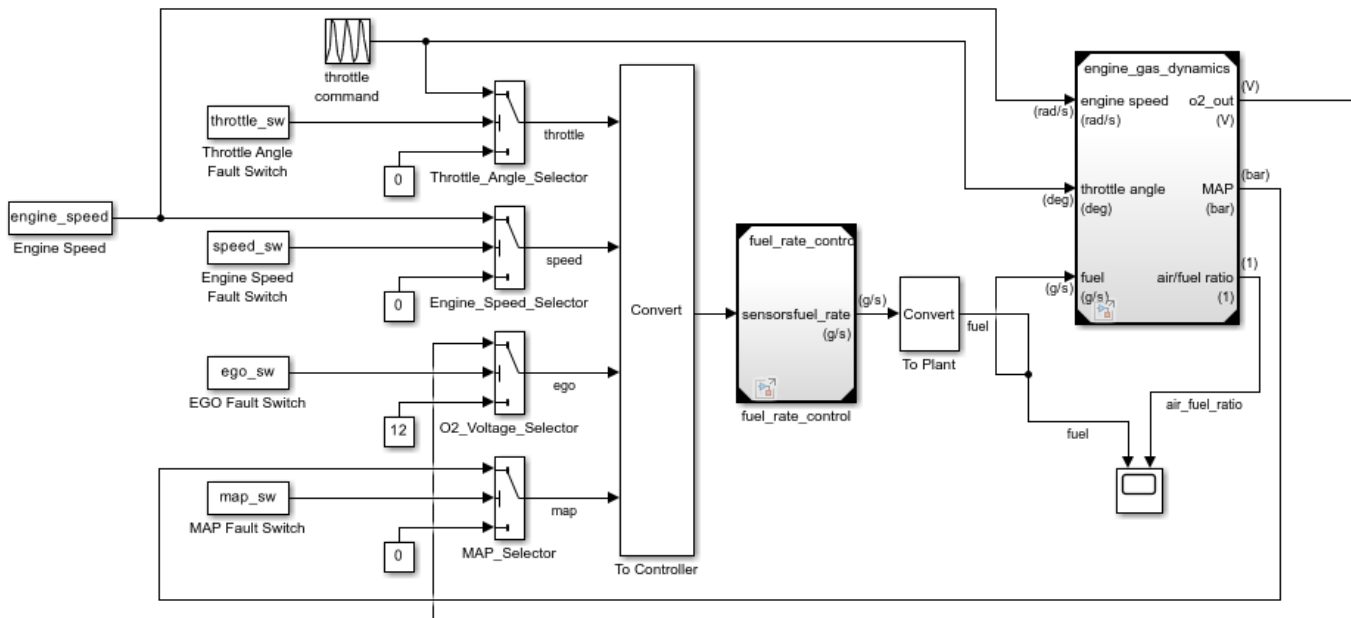
In the fault-tolerant fuel control system, the `control_logic` controls the fueling mode of the engine. In this example, you slice the `fuel_rate_control` referenced model. Then, investigate the effect of `fuel_rate_ratio` on the `Fueling_mode` of the engine. For more information, see “Model a Fault-Tolerant Fuel Control System”.

Step 1: Start the Model Slicer

To start the Model Slicer, open the `fuel_rate_control` model, and select **Apps > Model Verification, Validation, and Test > Model Slicer**.

```
open_system('sldvSlicerdemo_fuelsys');
```

Fault-Tolerant Fuel Control System

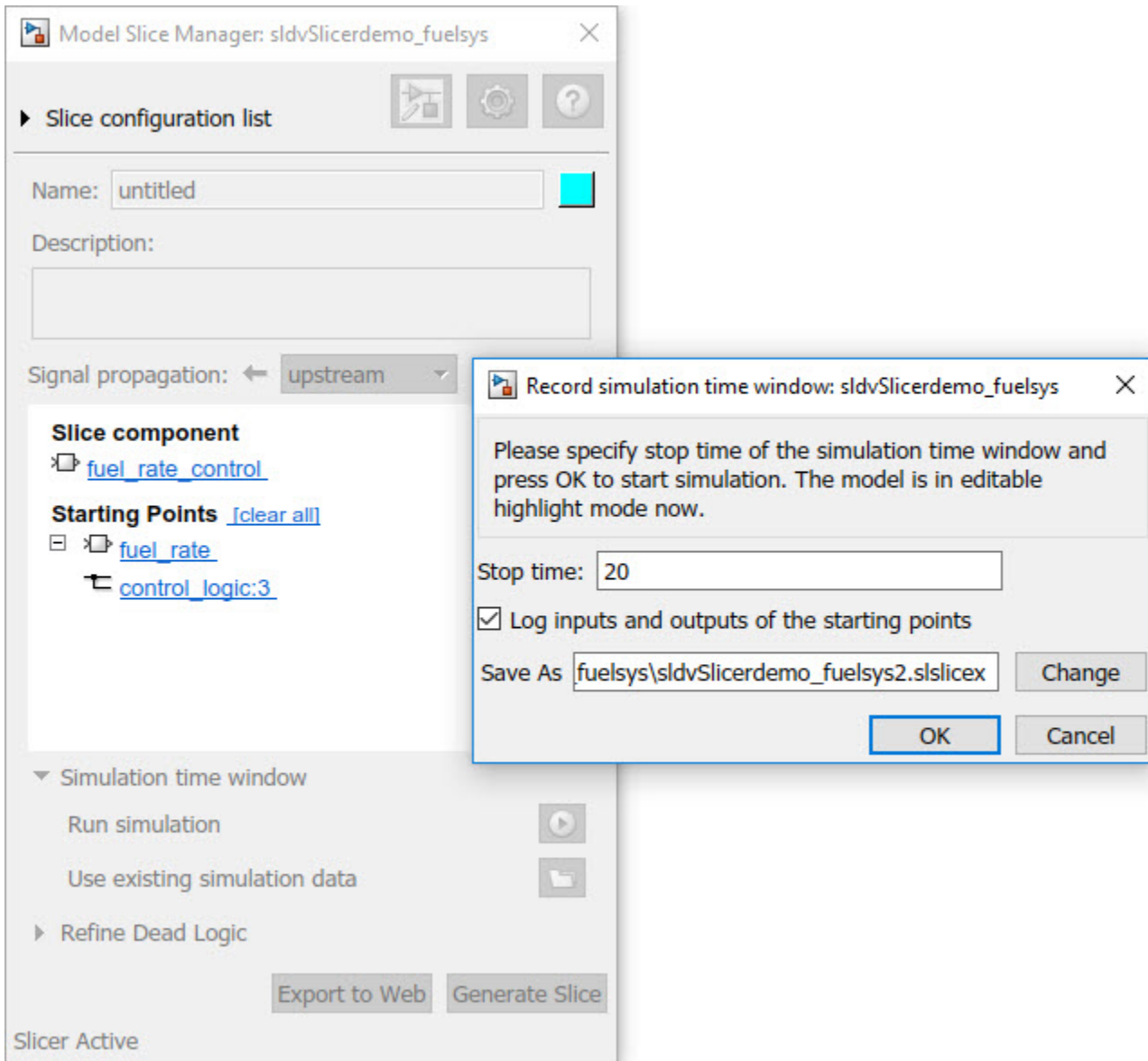


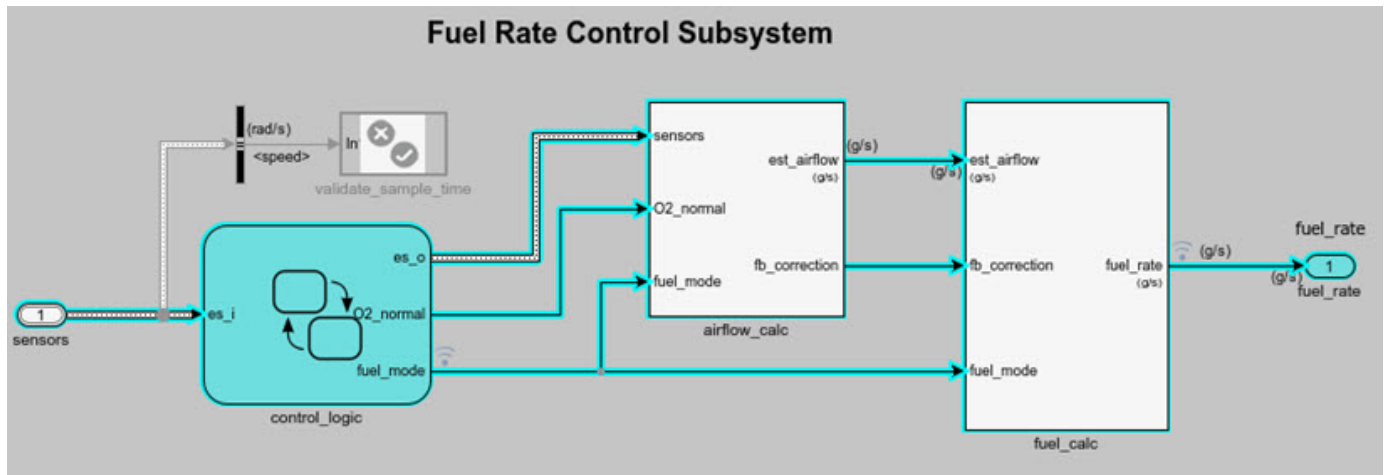
Copyright 1990-2017 The MathWorks, Inc.

To select the starting point, open the `fuel_rate_control` model, and add the `fuel_rate` port and the `fuel_mode` output signal as the starting point. To add a port or a signal as a starting point, right-click the port or signal, and select **Model Slicer > Add as Starting Point**.

Step 2: Log input and output signals

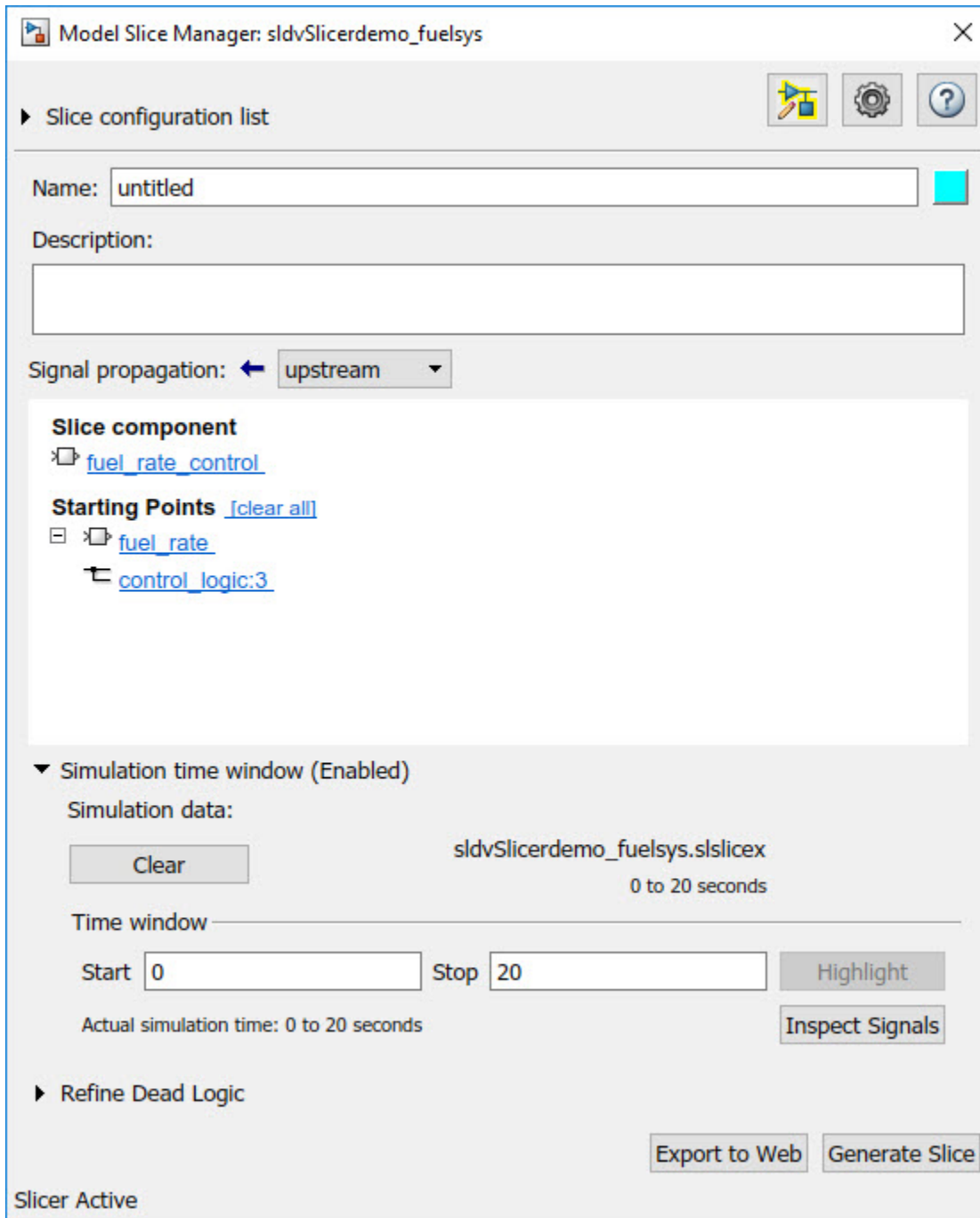
- In the Model Slicer dialog box, select the **Simulation time window** and **Run simulation**.
- In the Record simulation time window, for the **Stop time**, type 20.
- Select the **Log inputs and outputs of the starting points**.
- Click **OK**.





Step 3: Inspect signals

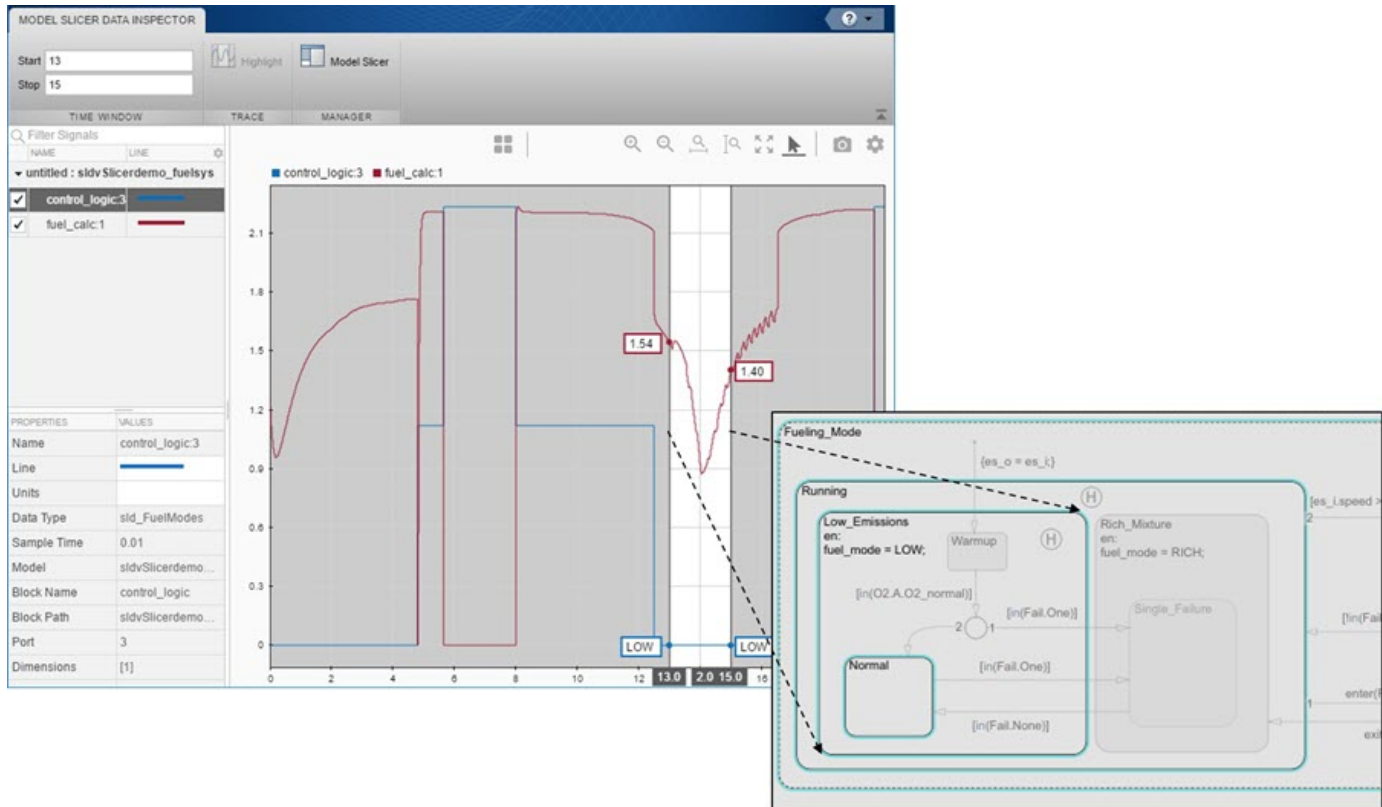
To open the Model Slicer Data Inspector, click **Inspect Signals**.



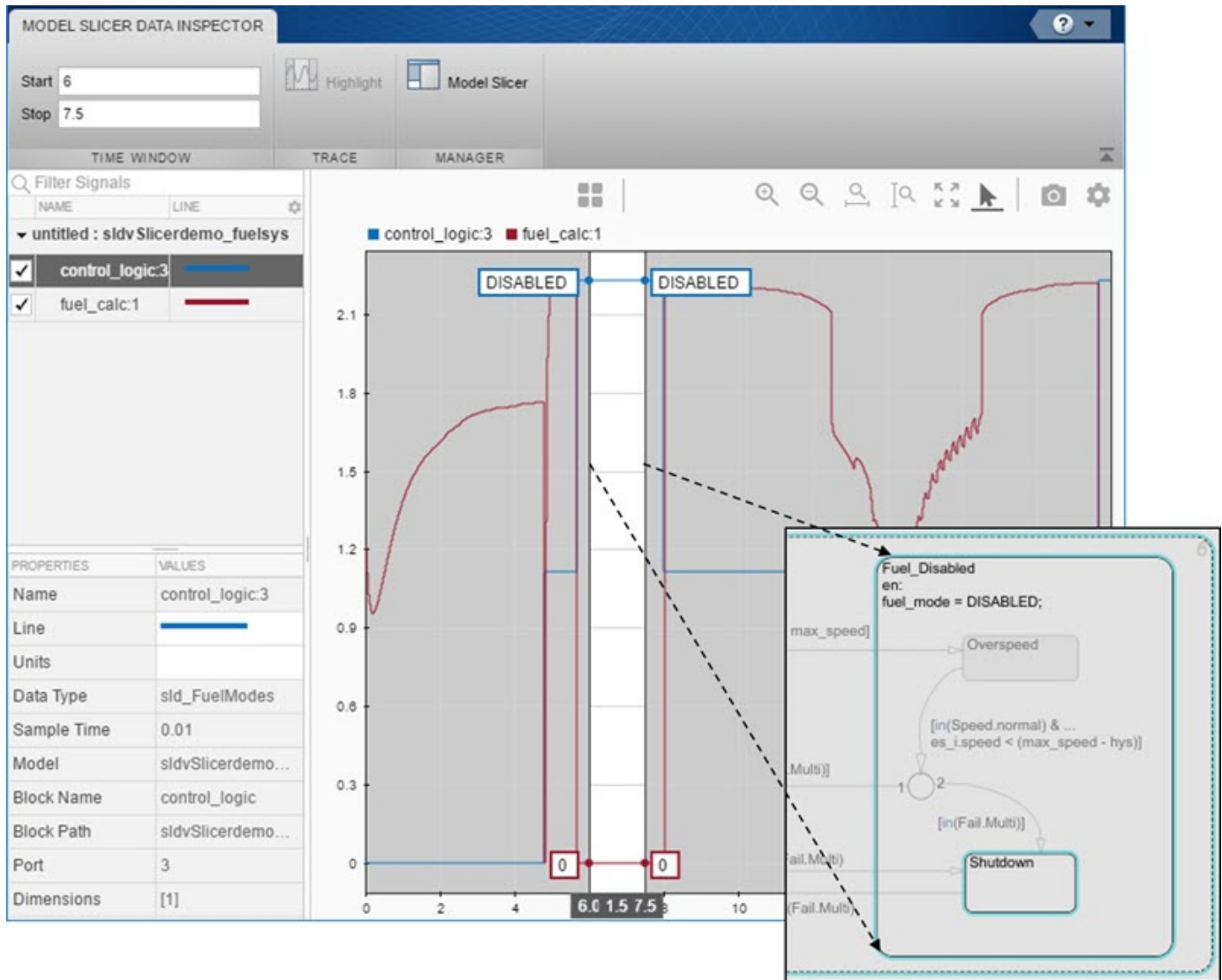
The logged input and output signals appear in the Model Slicer Data Inspector. When you open the Model Slicer Data Inspector, Model Slicer saves the existing Simulation Data Inspector session as MLDATX-file in the current working directory.

You can select the time window by dragging the data cursors to a specific location or by specifying the **Start** and **Stop** time in the navigation pane. To highlight the model for the defined simulation time window, Click **Highlight**.

To investigate the Fueling_mode, open the control_logic Stateflow™ chart, available in the fuel_rate_control referenced model. Select the time window for 13-15 seconds and click **Highlight**. For the defined simulation time window, the Low_Emissions fueling mode is active and highlighted.



Select the data cursor for the time window 6-7.5 seconds, with 0 fuel_cal:1. Click **Highlight**. In the control_logic model, the Fuel_Disabled state is highlighted. The engine is in Shutdown mode.



See Also

“Highlight Functional Dependencies” on page 8-2 | “Refine Highlighted Model” on page 8-12

Debug Slice Simulation by Using Fast Restart Mode

Perform multiple slicer simulations and streamline model debugging workflows by using Model Slicer in fast restart mode. For more information, see “Get Started with Fast Restart”.

If you enable fast restart mode, you can:

- Perform multiple slicer simulations efficiently with different inputs, without recompiling the model.
- Debug a simulation by stepping through the major time steps of a simulation and inspecting how a slice changes. For more information, see “Step Through Simulation”.

Simulate and Debug a Test Case in a Model Slice

This example shows how the fast restart mode performs slicer simulations with different test case inputs, without recompiling the model. You can simulate a sliced harness model with a test case input and highlight the dependency analysis in the model.

Analyze the highlighted slice by stepping through the time steps. You use the simulation stepper to analyze how the slice changes at each time step.

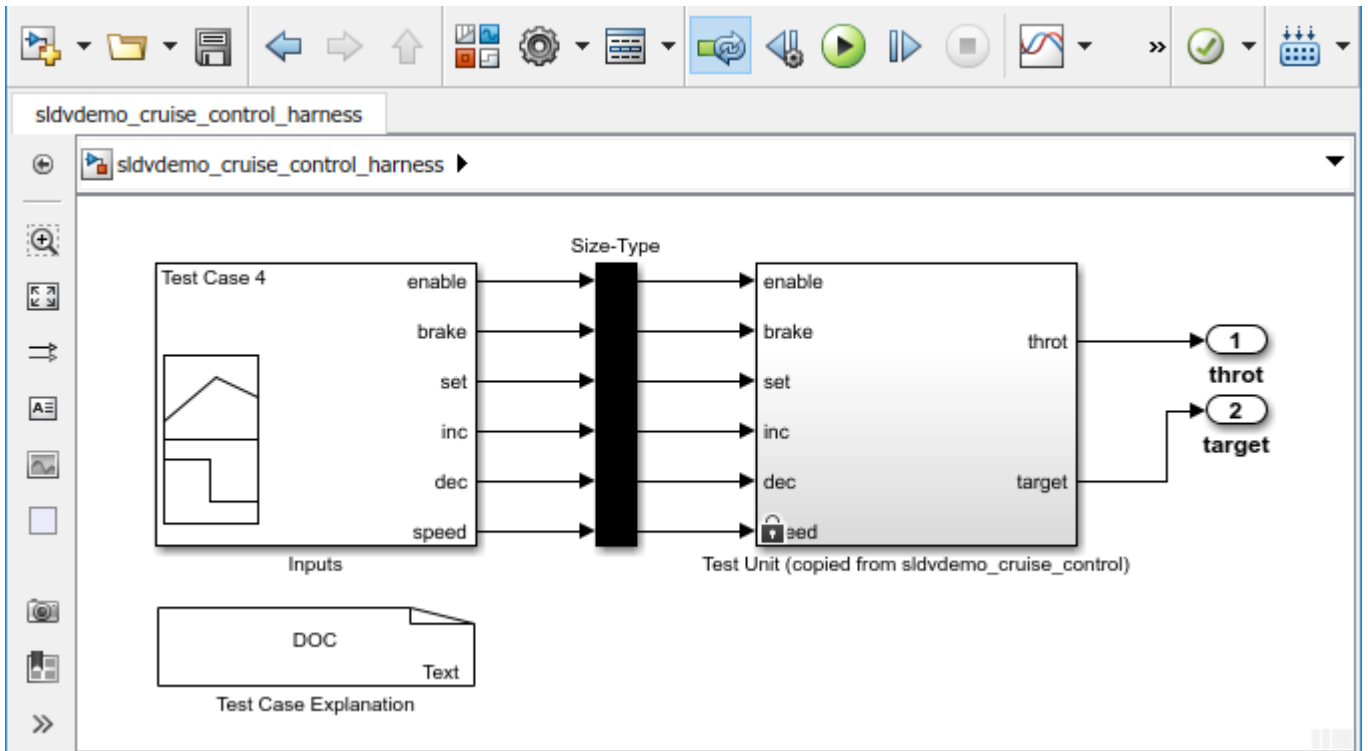
- 1 Open the `sldvdemo_cruise_control` model.

```
open_system('sldvdemo_cruise_control');
```


- 2 Set `sldvoptions` parameters and analyze the model by using the specified options.

```
opts = sldvoptions;  
opts.Mode = 'TestGeneration';           % Perform test-generation analysis  
opts.ModelCoverageObjectives = 'MCDC';  % Specify type of model coverage  
opts.SaveHarnessModel = 'on';           % Save harness as model file  
[ status, files ] = sldvrun('sldvdemo_cruise_control', opts);
```


After the analysis, the software opens a harness model `sldvdemo_cruise_control_harness` and saves it in the default location `<current_folder>\sldv_output\sldvdemo_cruise_control\sldvdemo_cruise_control_harness.slx`. For more information, see “Manage Simulink Design Verifier Harness Models” (Simulink Design Verifier).

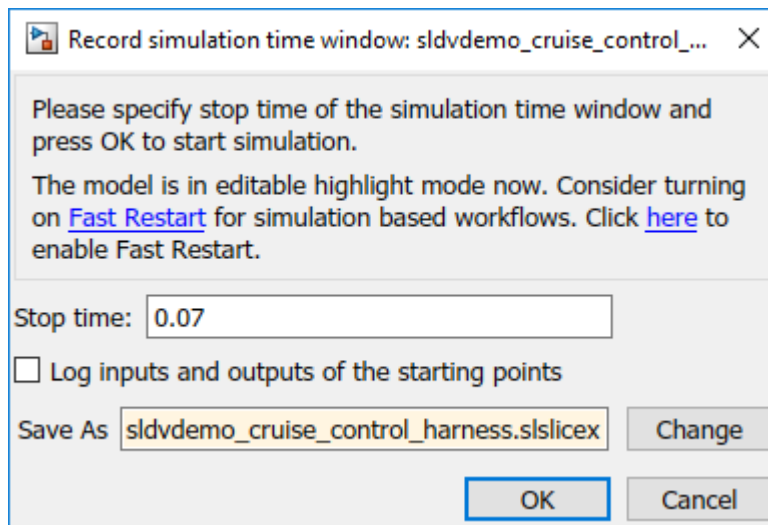


3

To enable the fast restart mode, click **Enable Fast Restart** button .

4 On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**. Model Slicer compiles the model.

Optionally, you can enable fast restart after opening the Model Slicer. Select **Simulation time window** and click the run simulation button . To enable fast restart, in the Record simulation time window, click the **here** link.




5 To add **Starting Points**, in the Model Slicer, click **Add all outputs**..

The `throt` and `target` outports are added as the **Starting Points**.

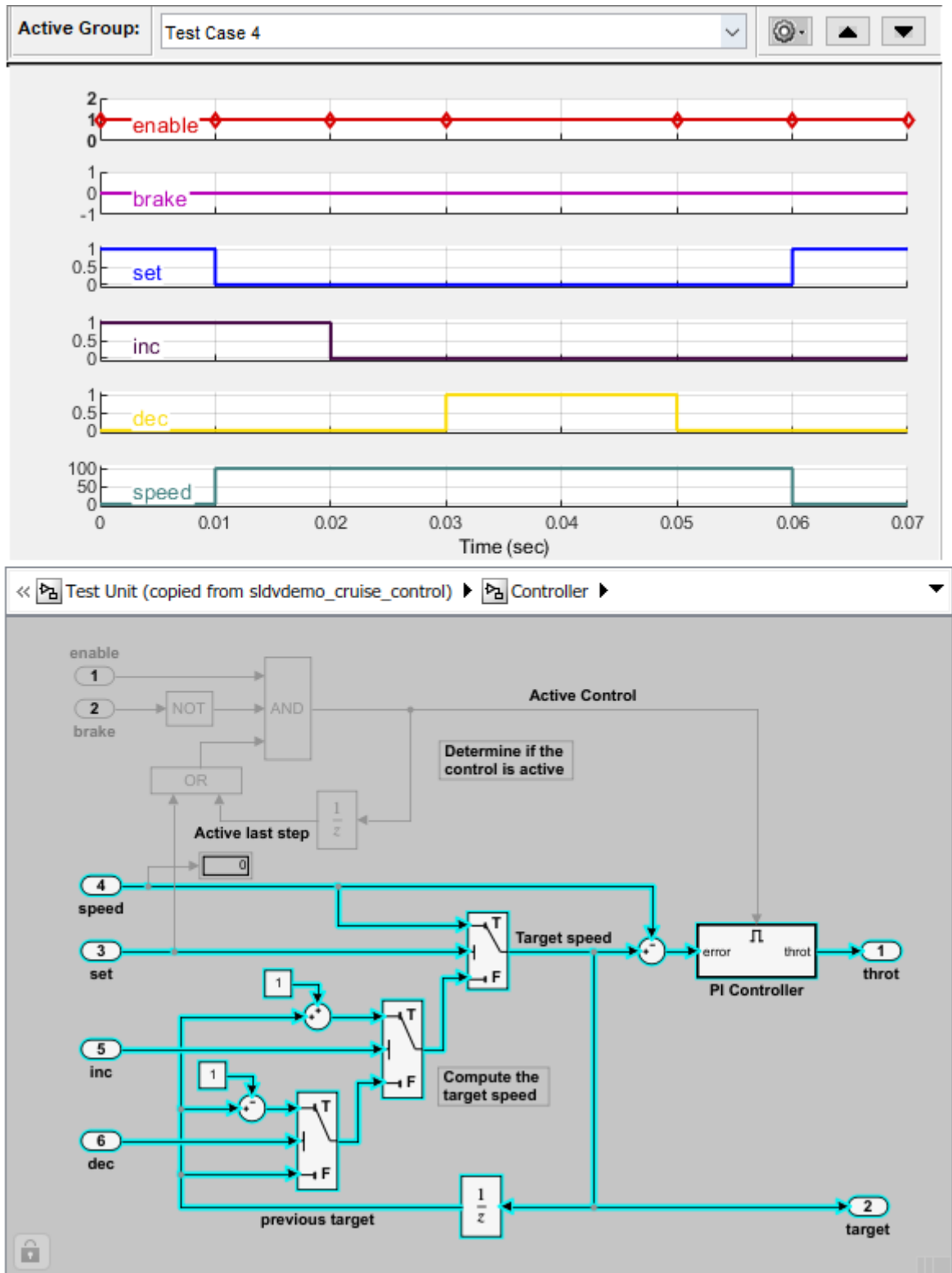
6 You can simulate a test case and analyze the highlighted dependencies in the slice.

a In the Signal Builder block, select Test Case 4.

b To simulate the test case, click **Start simulation** button, .

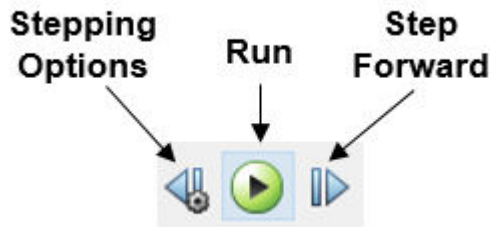
Optionally, you can simulate the model by using the **Run** button  in the Simulink editor. You can also simulate by using the **Simulation time window** in the Model Slicer.

The slice shows the highlighted dependencies for the Test Case 4 inputs.

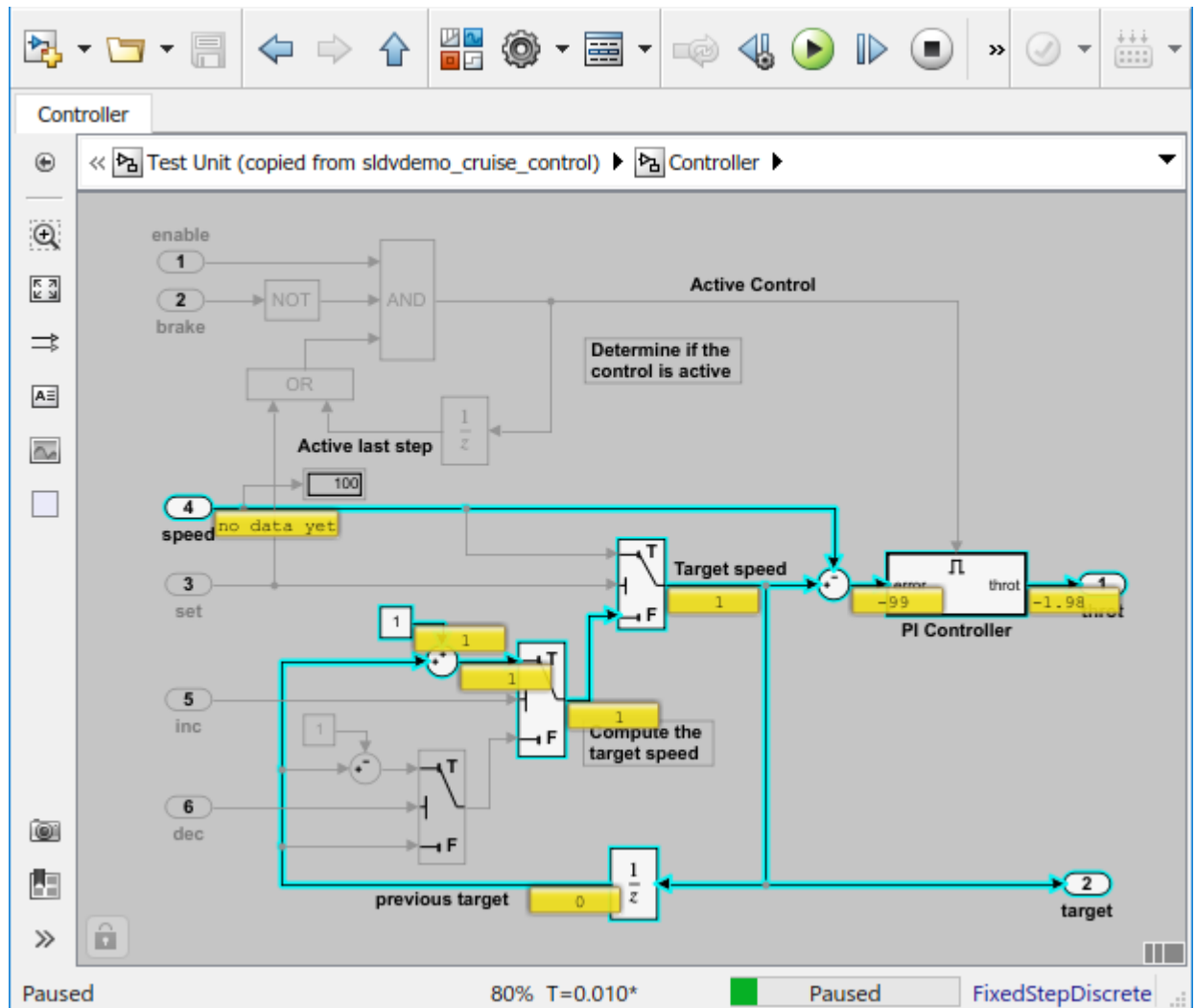


You can simulate a slice for different test case inputs and analyze the dependency analysis.

- 7 Debug a slicer simulation by using a simulation stepper. For more information see, “Step Through Simulation”.



- a To debug the simulation for the test case, in the Simulink Editor for the `sldvdemo_cruise_control_harness` model, click **Step Forward** button. You can view the signal values and the highlighted slice at each time step. For more information, see Simulation Stepping Options. The signal values and the dependencies at $T=0.010$ appears.



Isolate Referenced Model for Functional Testing

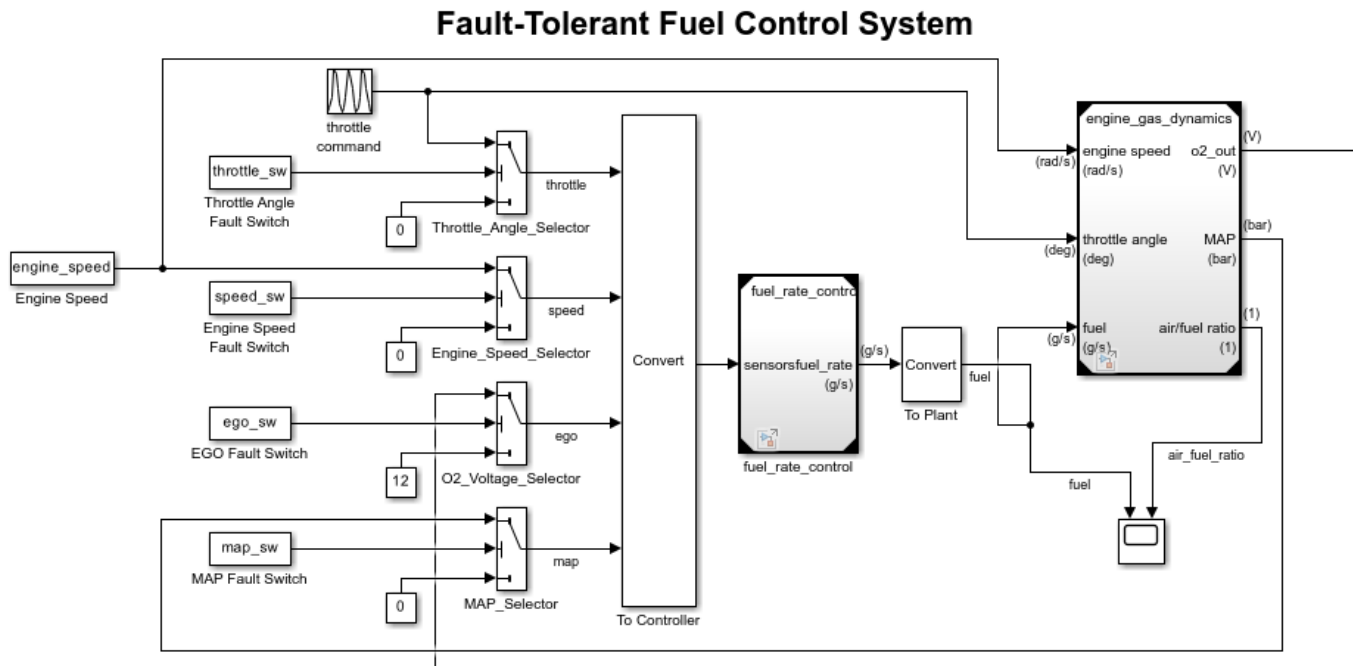
To functionally test a referenced model, you can create a slice of a referenced model treating it as an open-loop model. You can isolate the simplified open-loop referenced model with the inputs generated by simulating the close-loop system.

This example shows how to slice the referenced model controller of a fault-tolerant fuel control system for functional testing. To create a simplified open-loop referenced model for debugging and refinement, you generate a slice of the referenced controller.

Step 1: Open the Model

The fault-tolerant fuel control system model contains a referenced model controller `fuel_rate_control`.

```
open_system('sldvSlicerdemo_fuelsys');
```



Copyright 1990-2017 The MathWorks, Inc.

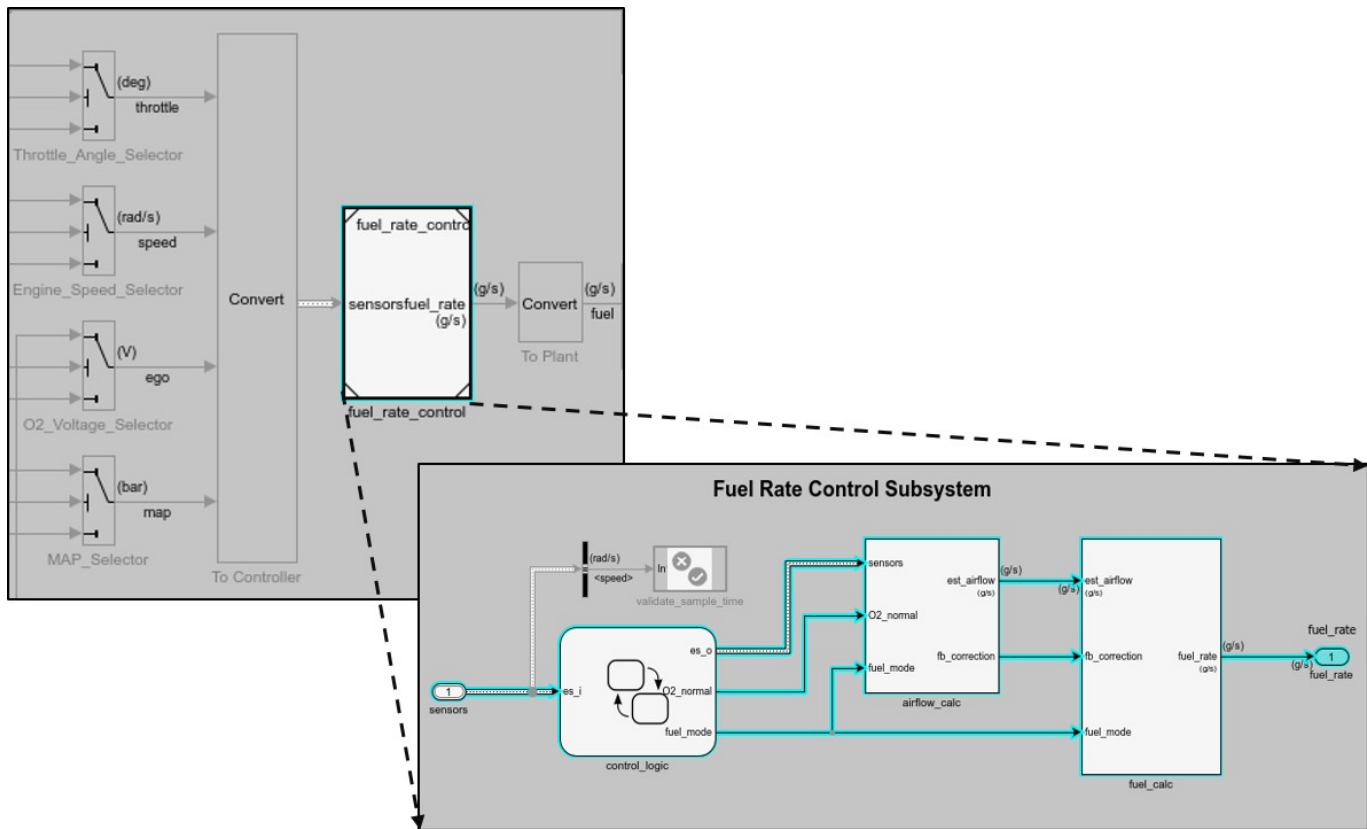
Step 2: Slice the Referenced Model

To analyze the `fuel_rate_control` referenced model, you slice it to create a standalone open-loop model. To open the Model Slice Manager, select **Apps > Model Verification, Validation, and Test > Model Slicer**, or right-click the `fuel_rate_control` model and select **Model Slicer > Slice component**. When you open the Model Slice Manager, the Model Slicer compiles the model. You then configure the model slice properties.

Note: The simulation mode of the `sldvSlicerdemo_fuelsys` model is Accelerator mode. When you slice the referenced model, the software configures the simulation mode to Normal mode and sets it back to its original simulation mode while exiting the Model Slicer.

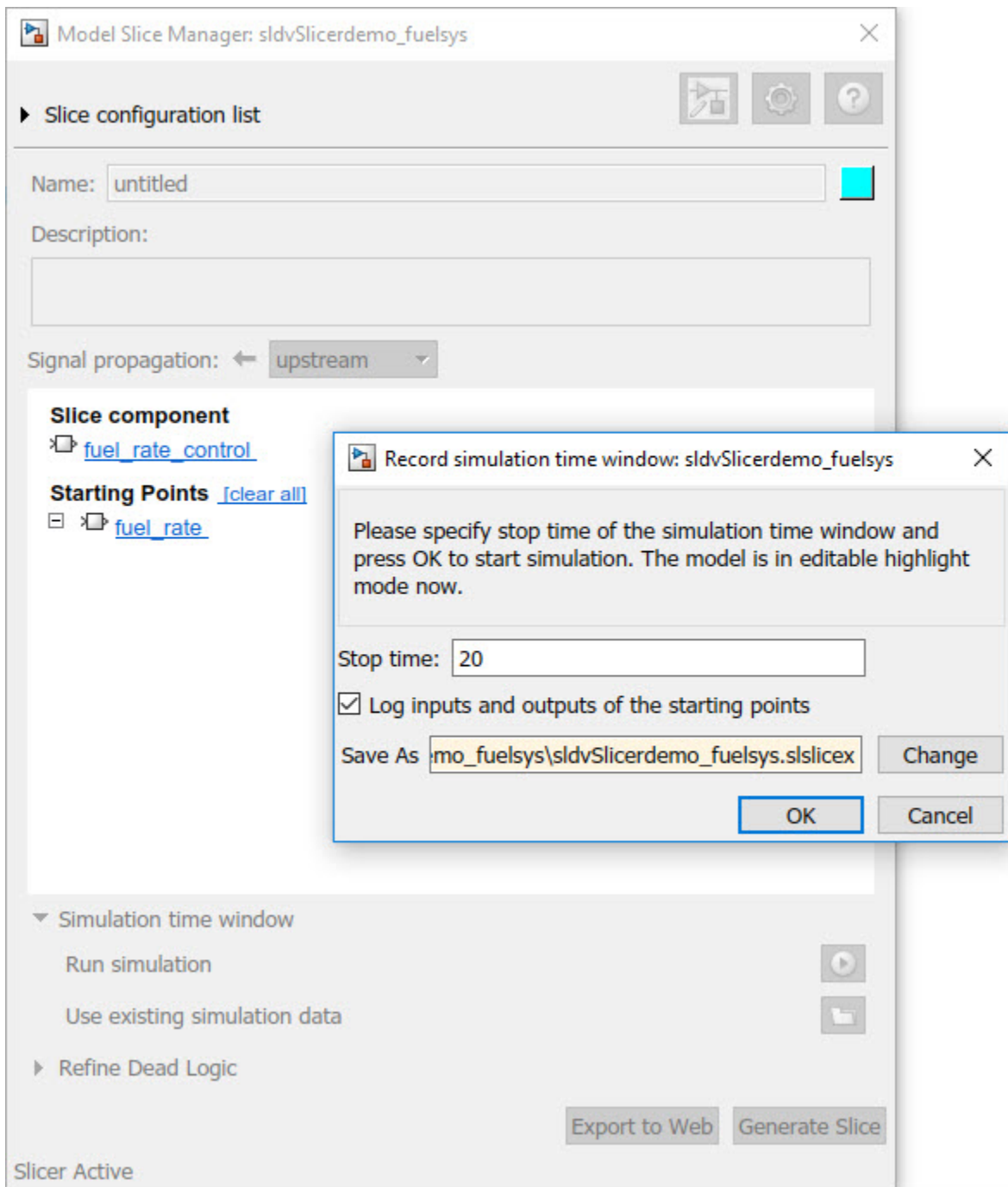
Step 3: Select Starting Point

Open the `fuel_rate_control` model, right-click the `fuel_rate` port, and select **Model Slicer > Add as starting point**. The Model Slicer highlights the upstream constructs that affect the `fuel_rate`.

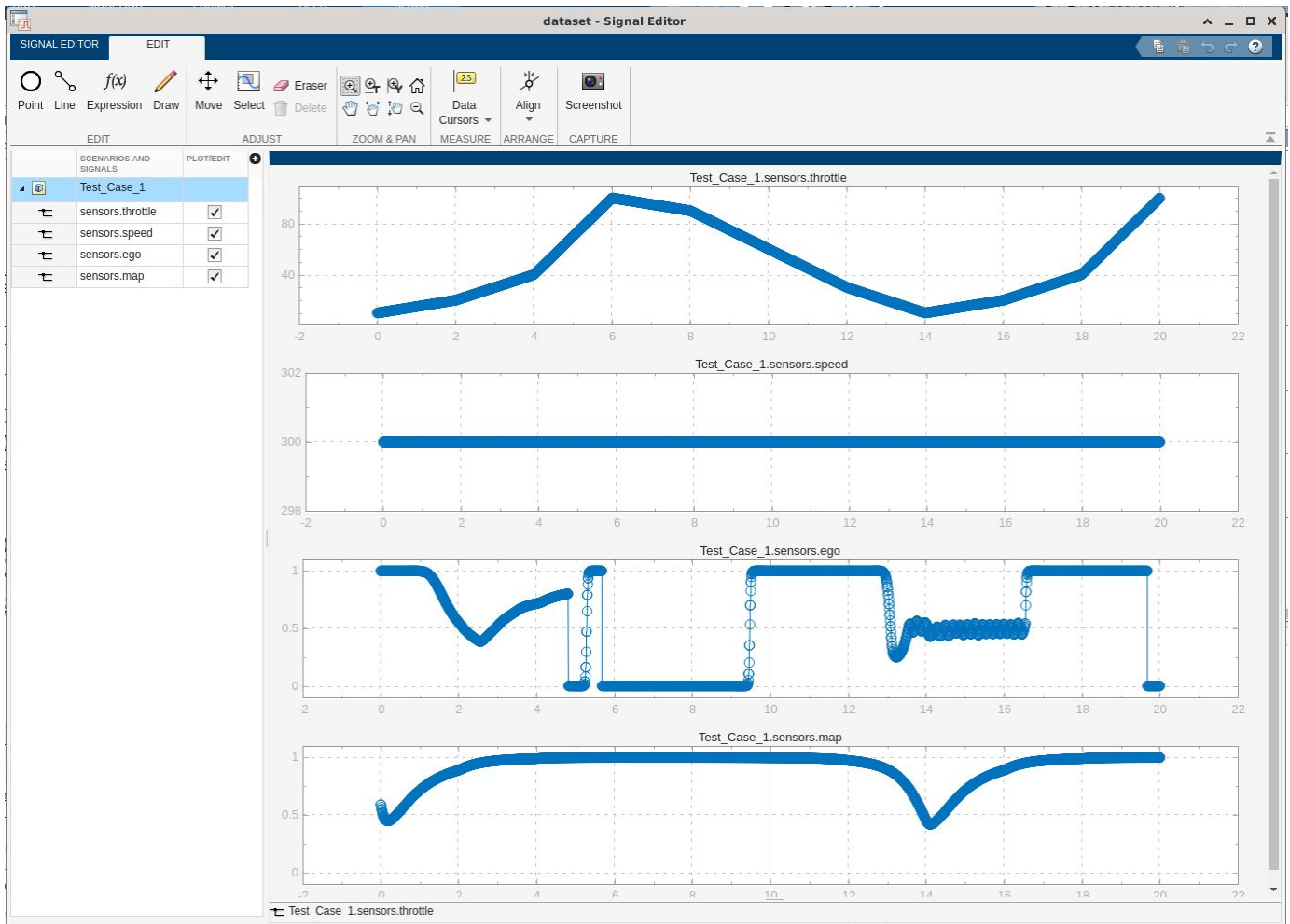
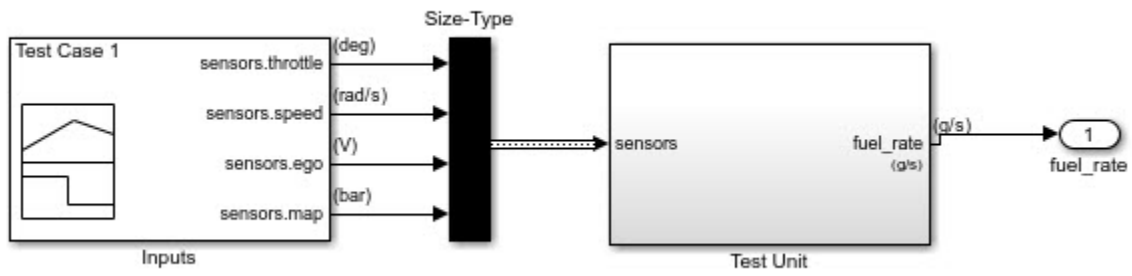


Step 4: Generate Slice

- In the Model Slice Manager dialog box, select the **Simulation time window**.
- Click **Run simulation**.
- For the **Stop time**, enter 20. Click **OK**.
- Click **Generate Slice**. The software simulates the sliced referenced model by using the inputs of the close-loop `sldvSlicerdemo_fuelsys` model.



For the sliced model, in the Signal Editor window, one test case is displayed that represents the signals input to the referenced model for simulation time 0-20 seconds.

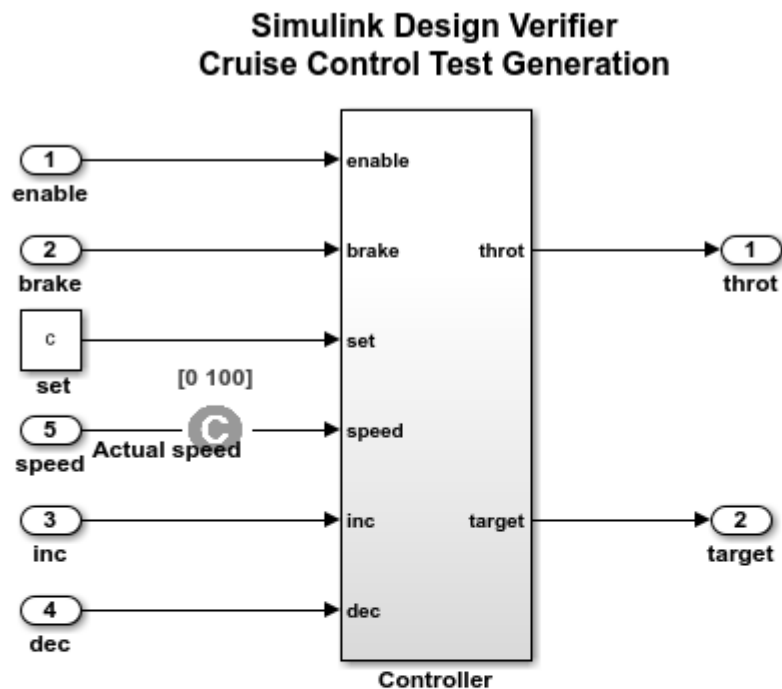


Analyze the Dead Logic

This example shows how to refine the model for dead logic. The `sldvSlicerdemo_dead_logic` model consists of dead logic paths that you refine for dependency analysis.

1. Open the `sldvSlicerdemo_dead_logic` model.
2. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

```
open_system('sldvSlicerdemo_dead_logic');
```



This example shows how to refine the model for dead logic. The model consists of a Controller subsystem that has a set value equal to 1. Dead logic refinement analysis identifies the dead logic in the model. The inactive elements are removed from the slice.

Run
(double-click)

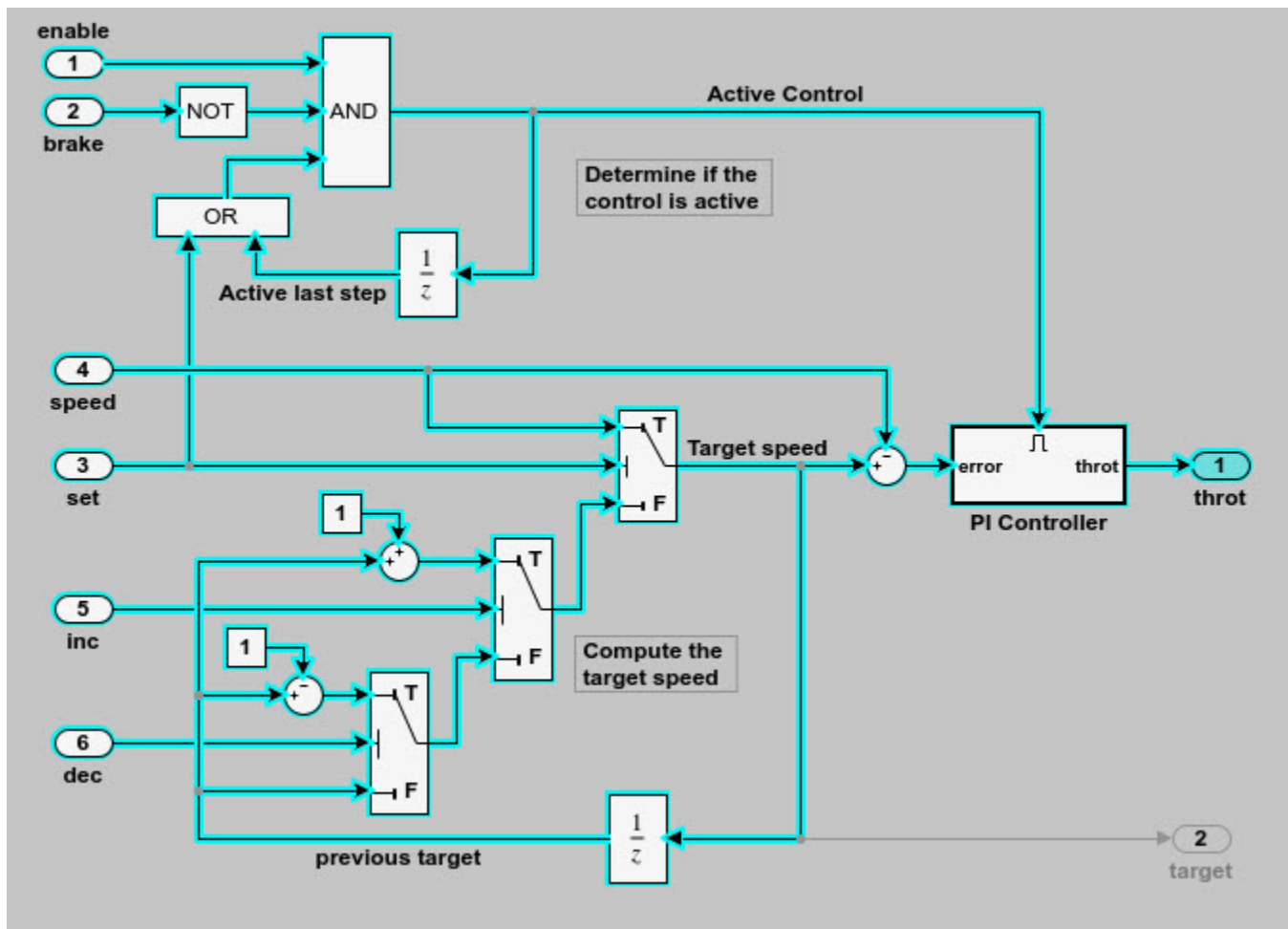
**Toggle Speed
Constraint**
(double-click)

View Options
(double-click)

Toggle Constraint

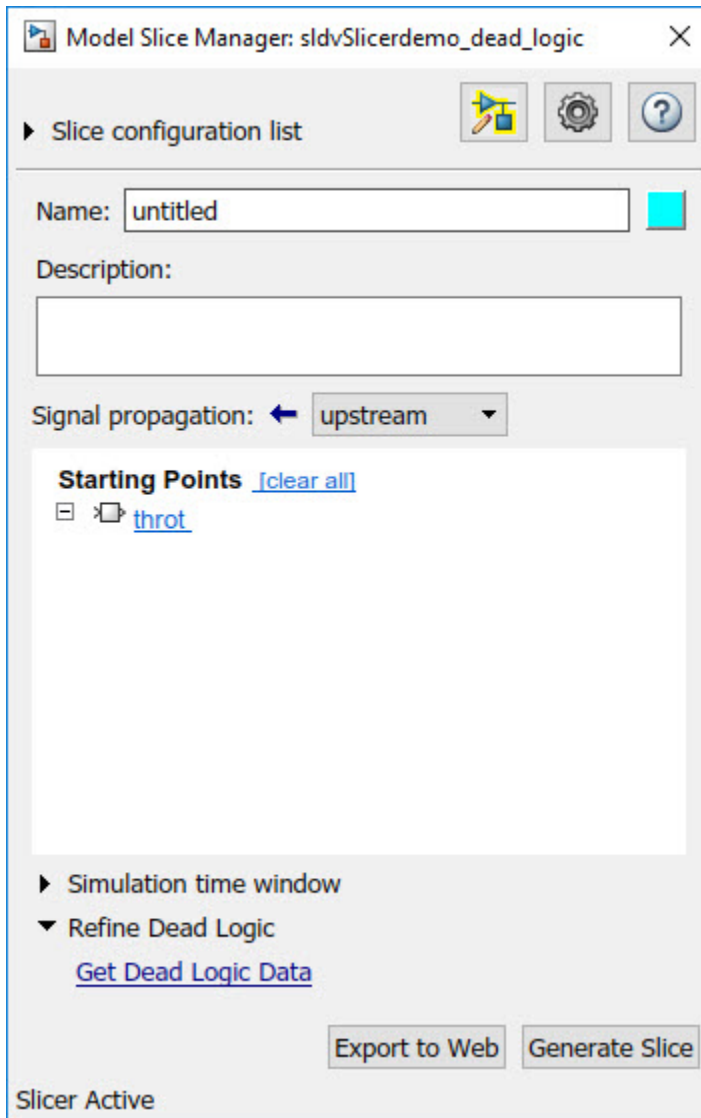
Copyright 2006-2018 The MathWorks, Inc.

Open the Controller subsystem and add the output `throt` as the starting point.

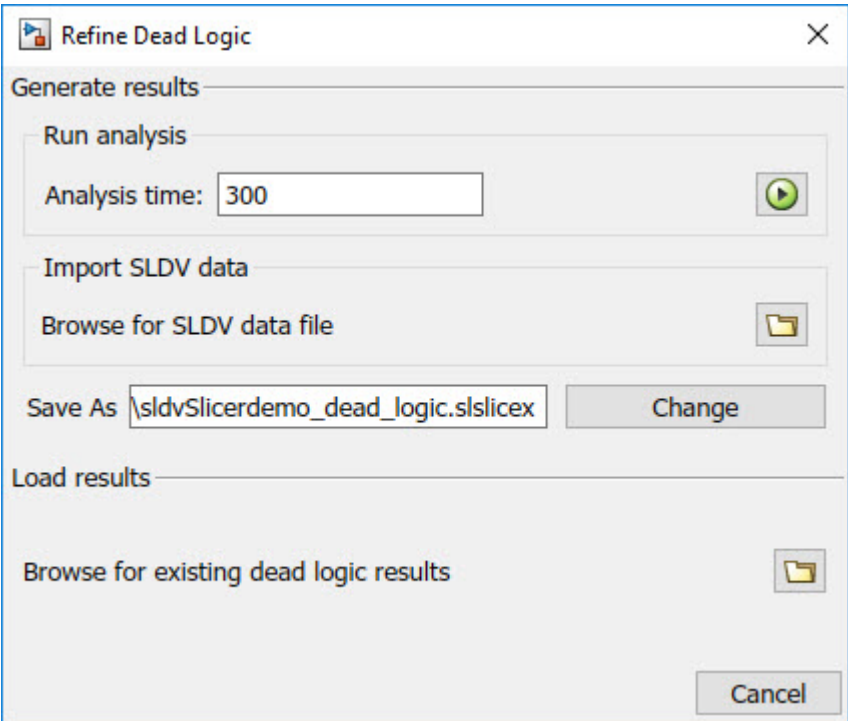


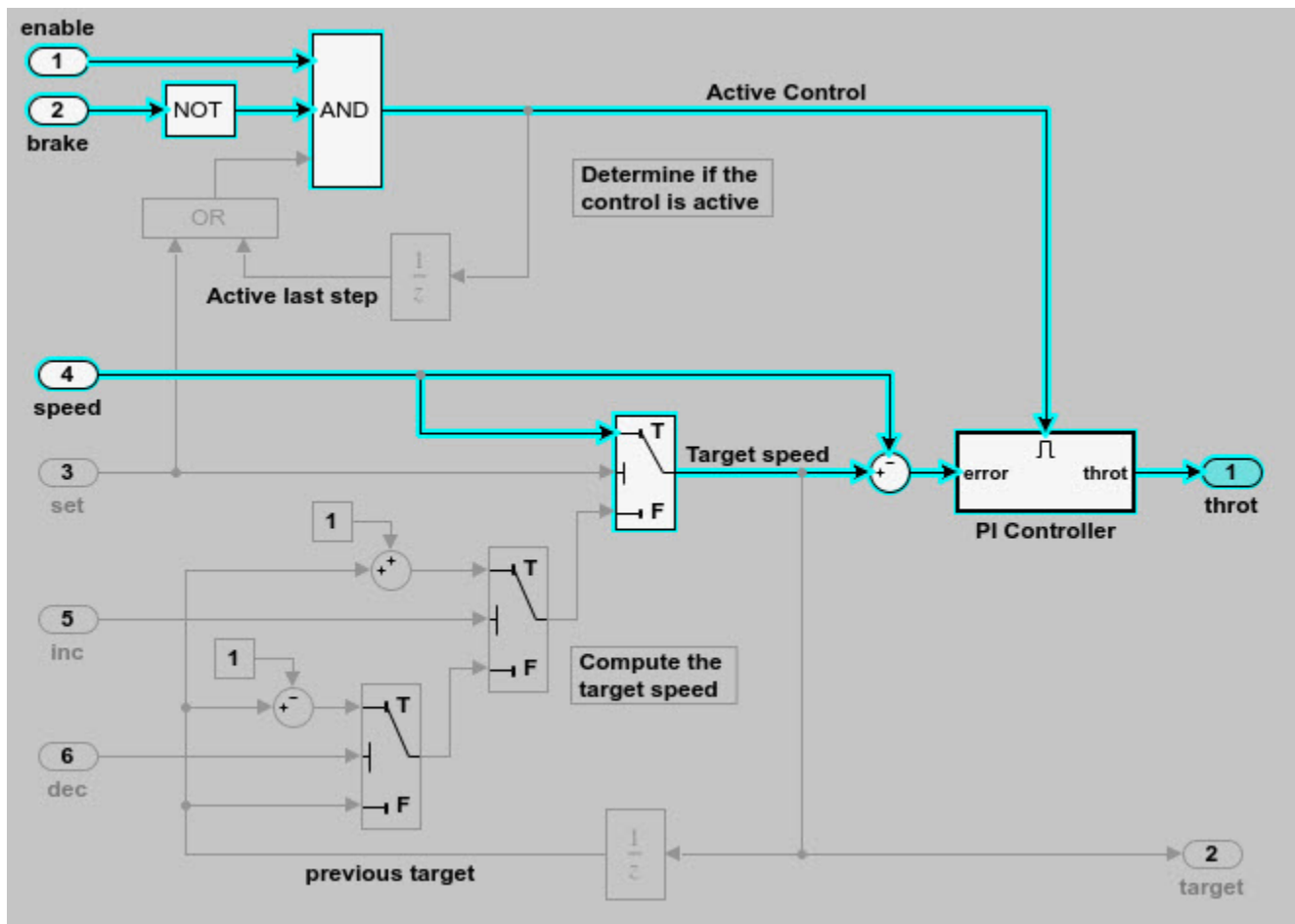
The Model Slicer highlights the upstream dependency of the throt output.

2. In the Model Slice Manager, select **Refine Dead Logic**.
3. Click **Get Dead Logic Data**.



4. Specify the **Analysis time** and run the analysis. You can import existing dead logic results from the `sldvData` file or load existing `.slicex` data for analysis. For more information, see “Refine Highlighted Model by Using Existing `.slicex` or Dead Logic Results” on page 8-63.





As the `set` input is equal to `true`, the `False` input to switch is removed for dependency analysis. Similarly, the output of block `OR` is always `true` and removed from the model slice.

Investigate Highlighted Model Slice by Using Model Slicer Data Inspector

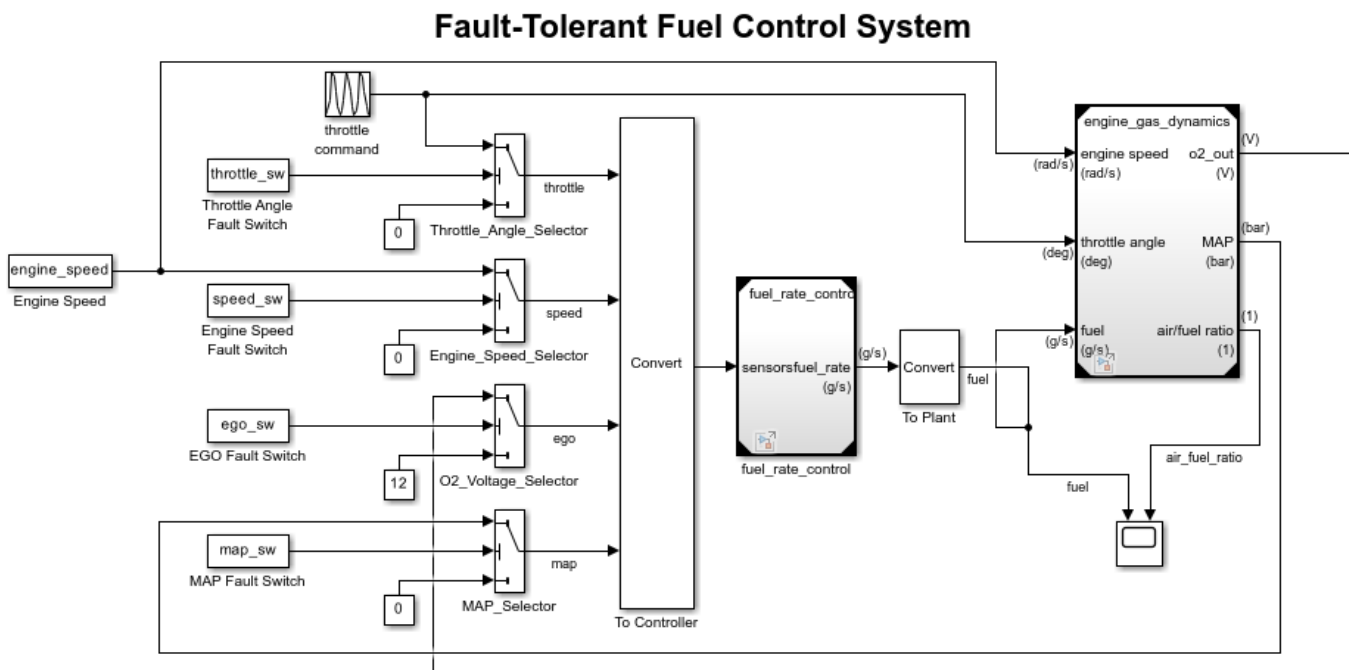
This example shows how to investigate and refine the highlighted model slice by using the Model Slicer Data Inspector.

In the fault-tolerant fuel control system, the `control_logic` controls the fueling mode of the engine. In this example, you slice the `fuel_rate_control` referenced model. Then, investigate the effect of `fuel_rate_ratio` on the `Fueling_mode` of the engine. For more information, see “Model a Fault-Tolerant Fuel Control System”.

Step 1: Start the Model Slicer

To start the Model Slicer, open the `fuel_rate_control` model, and select **Apps > Model Verification, Validation, and Test > Model Slicer**.

```
open_system('sldvSlicerdemo_fuelsys');
```



Copyright 1990-2017 The MathWorks, Inc.

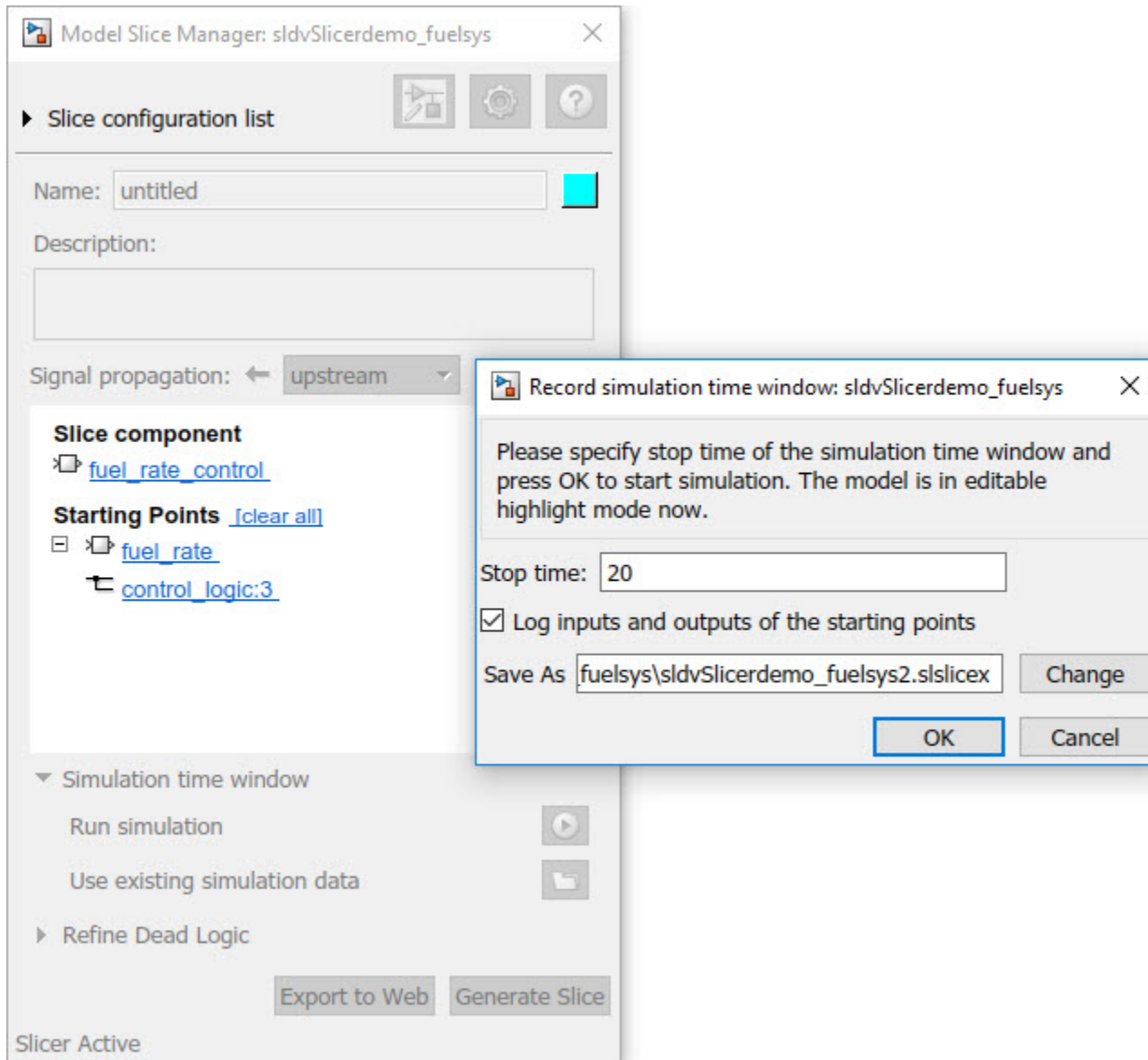
To select the starting point, open the `fuel_rate_control` model, and add the `fuel_rate` port and the `fuel_mode` output signal as the starting point. To add a port or a signal as a starting point, right-click the port or signal, and select **Model Slicer > Add as Starting Point**.

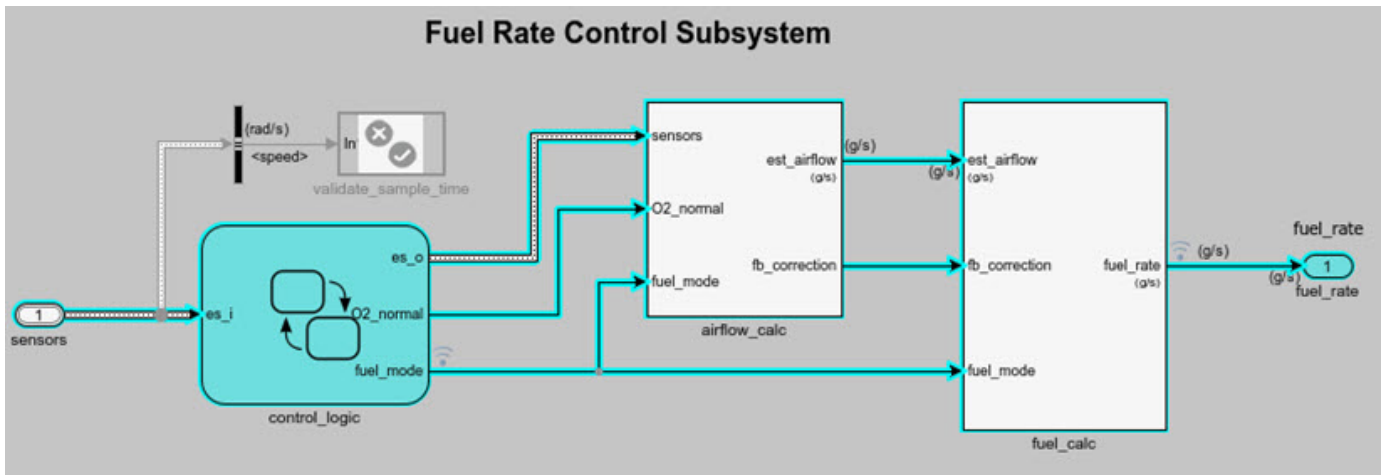
Step 2: Log input and output signals

- a. In the Model Slicer dialog box, select the **Simulation time window** and **Run simulation**.
- b. In the Record simulation time window, for the **Stop time**, type 20.

c. Select the **Log inputs and outputs of the starting points**.

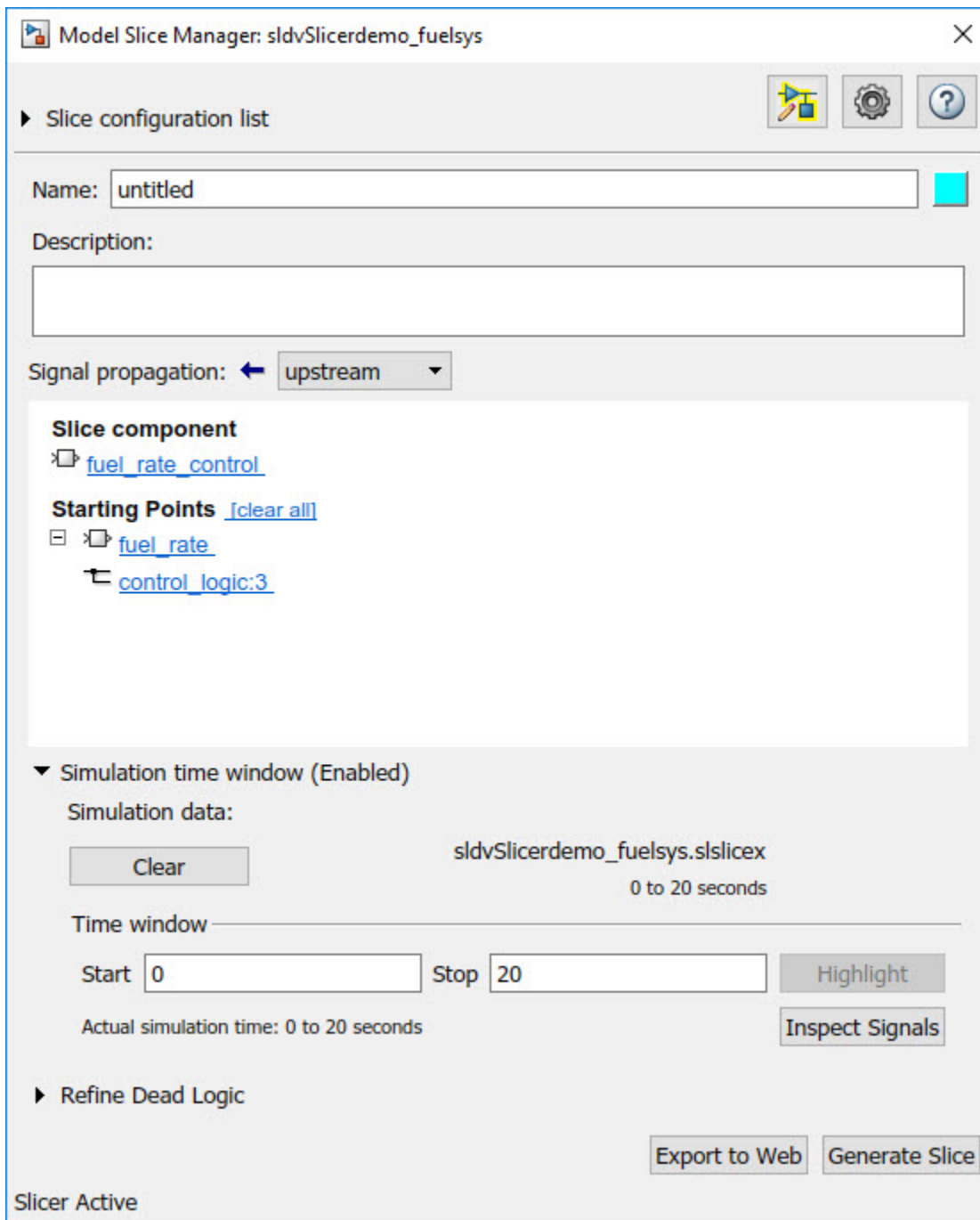
d. Click **OK**.





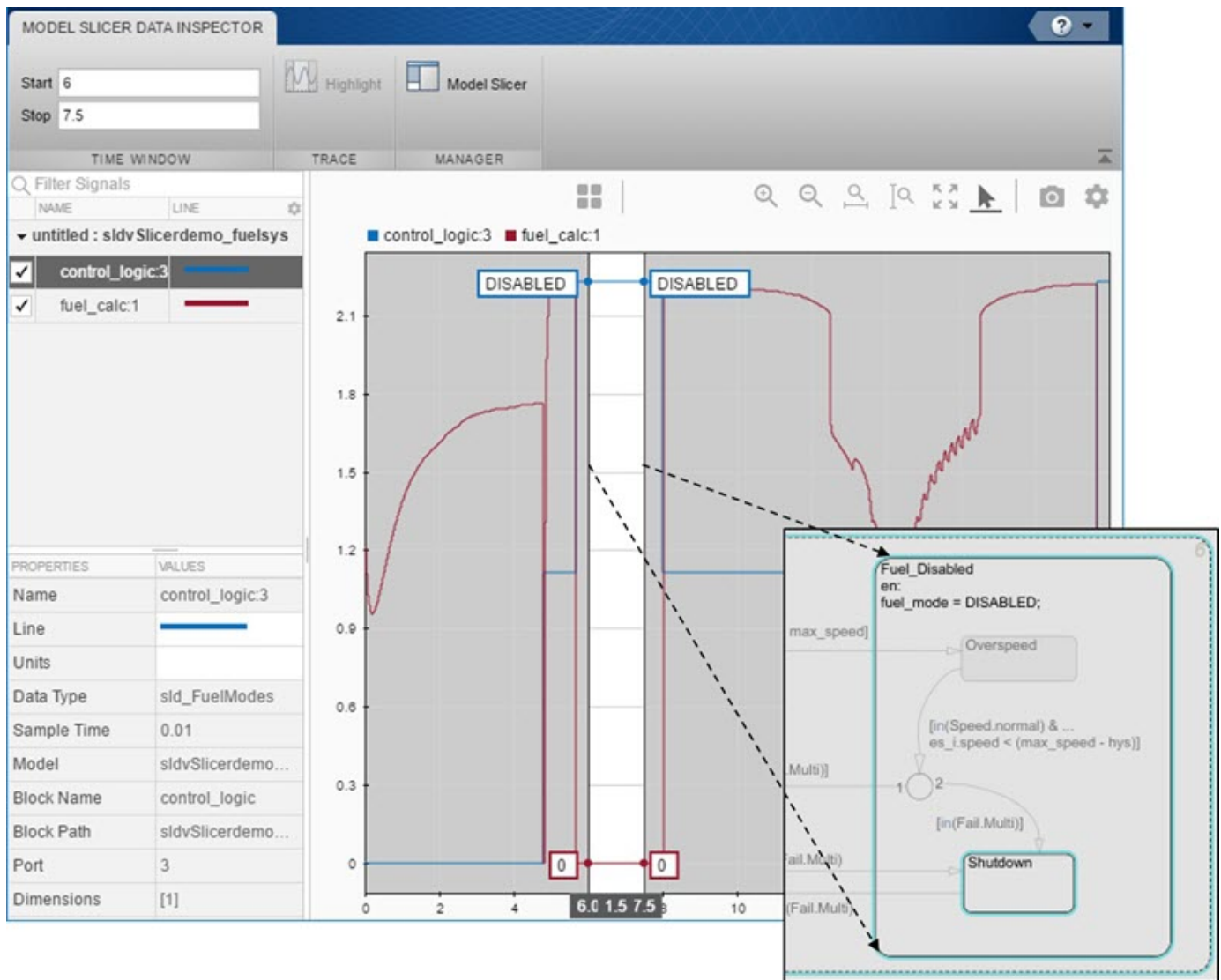
Step 3: Inspect signals

To open the Model Slicer Data Inspector, click **Inspect Signals**.



The logged input and output signals appear in the Model Slicer Data Inspector. When you open the Model Slicer Data Inspector, Model Slicer saves the existing Simulation Data Inspector session as MLDATX-file in the current working directory.

You can select the time window by dragging the data cursors to a specific location or by specifying the **Start** and **Stop** time in the navigation pane. To highlight the model for the defined simulation time window, Click **Highlight**.



Programmatically Generate I/O Dependency Matrix

This example shows how to programmatically generate a dependency matrix that shows the relationship between root level inports and outports.

To create the dependency matrix:

1. Open model `slcheckSliceCruiseControl`.

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

2. Create a `SysDependencyTabulator` object.

```
obj=SysDependencyTabulator(model);
```

3. Initialize a model handle.

```
sysH = get_param(model, 'handle');
```

4. Use **tabulateDependencies** method to create a dependency matrix for the model handle.

```
T1 = obj.tabulateDependencies(sysH)
```

T1=5x11 table

	enbl	cncl	set	resume	inc	dec	brakeP	key	gear	th
reqDrv	1	1	1	1	1	1	0	0	0	
status	1	1	1	1	1	1	1	1	1	
operation_mode	1	1	1	1	1	1	1	1	1	
targetSp	1	1	1	1	1	1	1	1	1	
throtCC	1	1	1	1	1	1	1	1	1	

5. Initialise a subsystem handle.

```
subsystemPath = [model '/CruiseControlMode'];
sysH = get_param(subsystemPath, 'handle');
```

6. Use **tabulateDependencies** method to create a dependency matrix for the subsystem handle.

```
T2 = obj.tabulateDependencies(sysH)
```

T2=2x5 table

	reqDrv	brakeP	vehSp	key	gear
status	1	1	1	1	1
mode	1	1	1	1	1

7. Delete the `SysDependencyTabulator` object.

```
delete(obj);
```

8. Close the model.

```
close_system(model);
```


Observe Impact of Simulink Parameters Using Model Slicer

Use Model Slicer to observe the impact a parameter has on a model.

This example demonstrates the ability of Model Slicer to display the parameters that affect a block (Option 1), and blocks that are affected by a parameter (Option 2) using the methods of the `SLSlicerAPI.ParameterDependence` class `parametersAffectingBlock`, and `blocksAffectedByParameter` respectively.

Open Model and Initialize ParameterDependence Class

1. Open the model `sldvSliceCruiseControl`.

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

2. Create an object of the `ParameterDependence` Class.

```
slicerObj = slslicer(model);
pd = slicerObj.parameterDependence;
```

Option 1: Find Parameters Affecting a Block

1. View the parameters that affect the `Switch3` block in the `DriverSwRequest` subsystem by entering:

```
params = parametersAffectingBlock(pd, 'sldvSliceCruiseControl/DriverSwRequest/Switch3')
```

```
params=1x49 object
    1x49 VariableUsage array with properties:
```

```
    Name
    Source
    SourceType
    Users
```

You can see that there are 49 parameters that affect the `Switch3` block. To view the details of individual parameters, explore each element of the array:

```
params(1)
```

```
ans =
    VariableUsage with properties:
        Name: 'CountValue'
        Source: 'sldvSliceCruiseControl/DriverSwRequest/decrement/counter'
        SourceType: 'mask workspace'
        Users: {'sldvSliceCruiseControl/DriverSwRequest/decrement/counter/Constant'}
```

Option 2: Get Blocks Affected by a Parameter

1. To observe the impact of a parameter, create a `Simulink.VariableUsage` object for that parameter.

```
param = Simulink.VariableUsage('CountValue', 'sldvSliceCruiseControl/DriverSwRequest/decrement/co
```

2. To view all the blocks affected by param:

```
affectedBlocks = blocksAffectedByParameter(pd, param)
```

```
affectedBlocks = 1×153  
103 ×
```

```
0.0130 0.0280 0.0290 0.0720 0.0730 0.0740 0.0840 0.1390 0.1720 0.1
```

You can further refine the blocks affected using the same options supported by `find_system`.

```
affectedOutputs = blocksAffectedByParameter(pd, param, 'blockType', 'Output')
```

```
affectedOutputs = 1×5  
103 ×
```

```
1.8780 1.8770 1.8790 1.8800 1.8810
```

Optional Step: Highlight Result on Model by Using Model Slicer

You can view the active section of the analyzed model by using the Model Slicer highlighting.

```
 slicerObj.highlight(slicerObj.ActiveConfig);
```

Clean Up

Model Slicer maintains the model in compiled state after analysis. To close the model, terminate the `slicerObj` object.

```
slicerObj.terminate;
```

Analyze Models Containing Simulink Functions Using Model Slicer

This example shows you how to analyze models that contain Simulink® functions using Model Slicer. You can visualize the interdependency of a Simulink function and Function Caller by adding them as starting points in Model Slicer. For more information on Simulink functions, see “Simulink Functions Overview”.

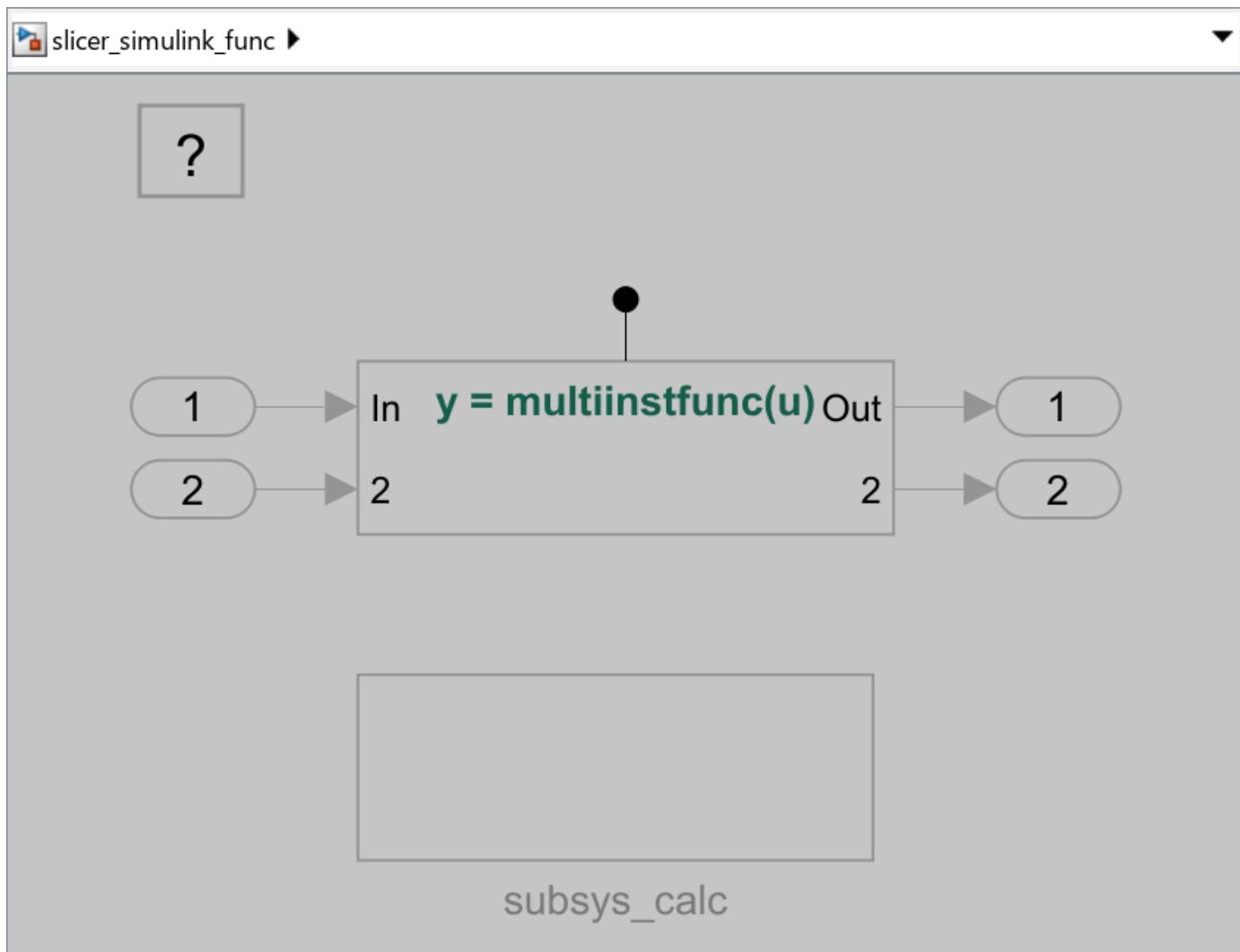
In this example, you visualize interdependencies between a Simulink Function and a Function Caller block in the upstream and downstream directions. The Simulink model `slicer_simulink_func.slx` contains the Simulink Function Caller `subsys_calc.func_calc()`, and its corresponding Simulink Function is defined in the `subsys_calc` subsystem.

Open the Model

1. Open the model `slicer_simulink_func`.

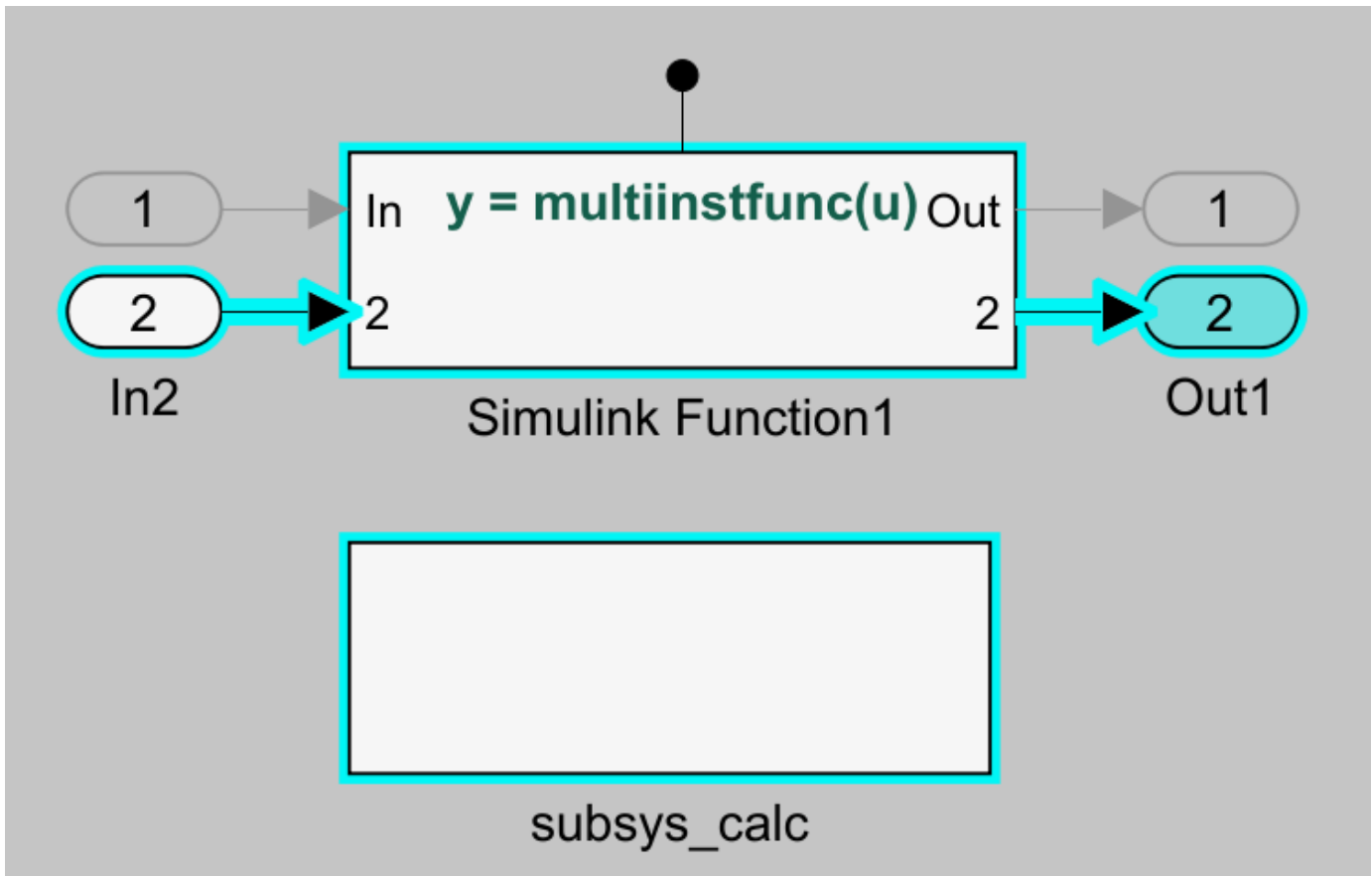
```
open_system('slicer_simulink_func')
```

2. On the **Apps** tab, under **Model Verification, Validation, and Test** gallery, click **Model Slicer**.

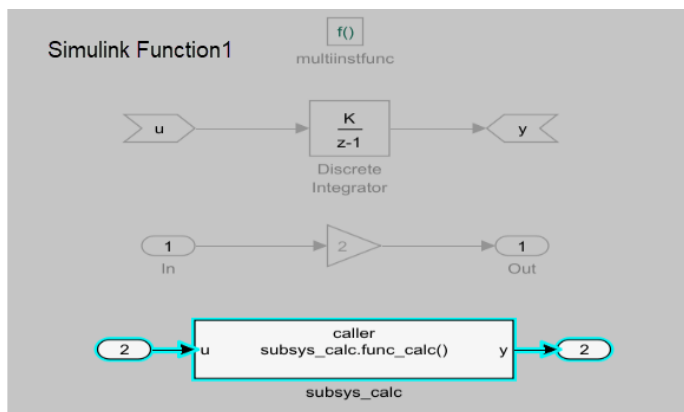


Upstream Highlighting

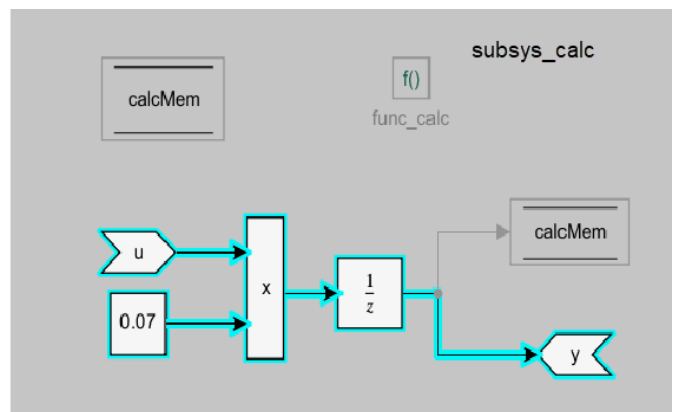
1. In the Model Slicer window, set **Signal propagation** to upstream.
2. In the `slicer_simulink_func` model, select output `Out1`. Right-click `Out1` and under **Model Slicer**, select **Add as Starting Point**. Model Slicer highlights the dependency of output `Out1` in the upstream direction.



3. Double-click the Simulink Function1 and subsys_calc subsystem and you can observe the dependencies. Output Out1 traces back to the Function Caller block subsys_calc.func_calc, the corresponding Simulink function for the block is defined in subsys_calc.



Function Caller

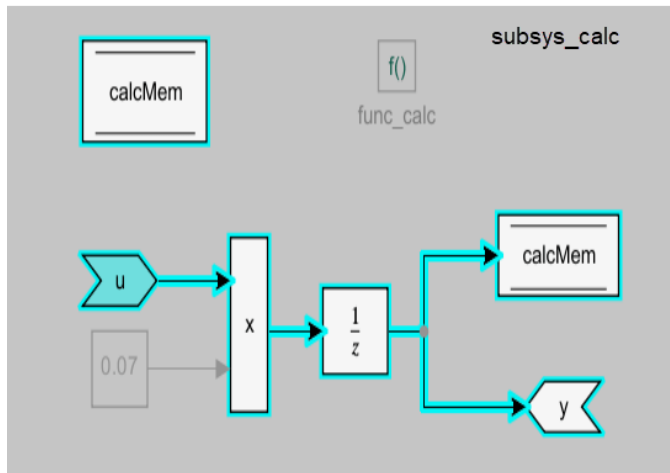


Function Definition

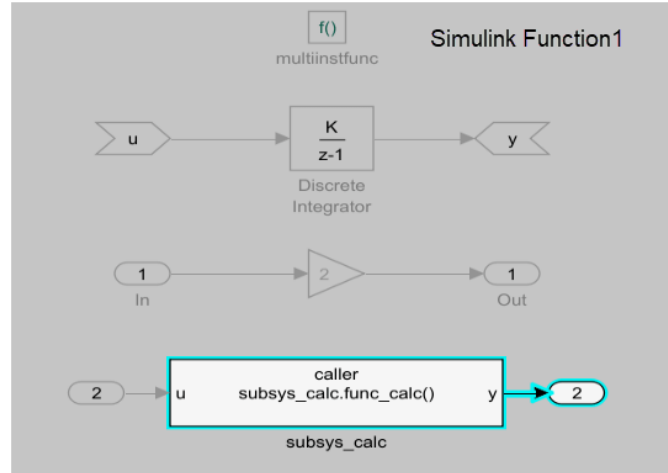
Downstream Highlighting

1. In the Model Slicer window, click **clear all** for **Starting points** and set the **Signal propagation** to downstream.

2. In the `subsys_calc` subsystem, select the `ArgIn` input parameter `u` as the starting point. Right-click `u`, and under **Model Slicer**, select **Add as Starting Point**. Model Slicer highlights the dependency of input `u` in the downstream direction from Function Caller `subsys_calc.func_calc` to the Simulink Function defined under `subsys_calc`.



Function Definition



Function Caller

See Also

- “Model Slicer Support Limitations for Simulink Blocks” on page 8-47
- “Client-Server Communication Interfaces” (Embedded Coder)