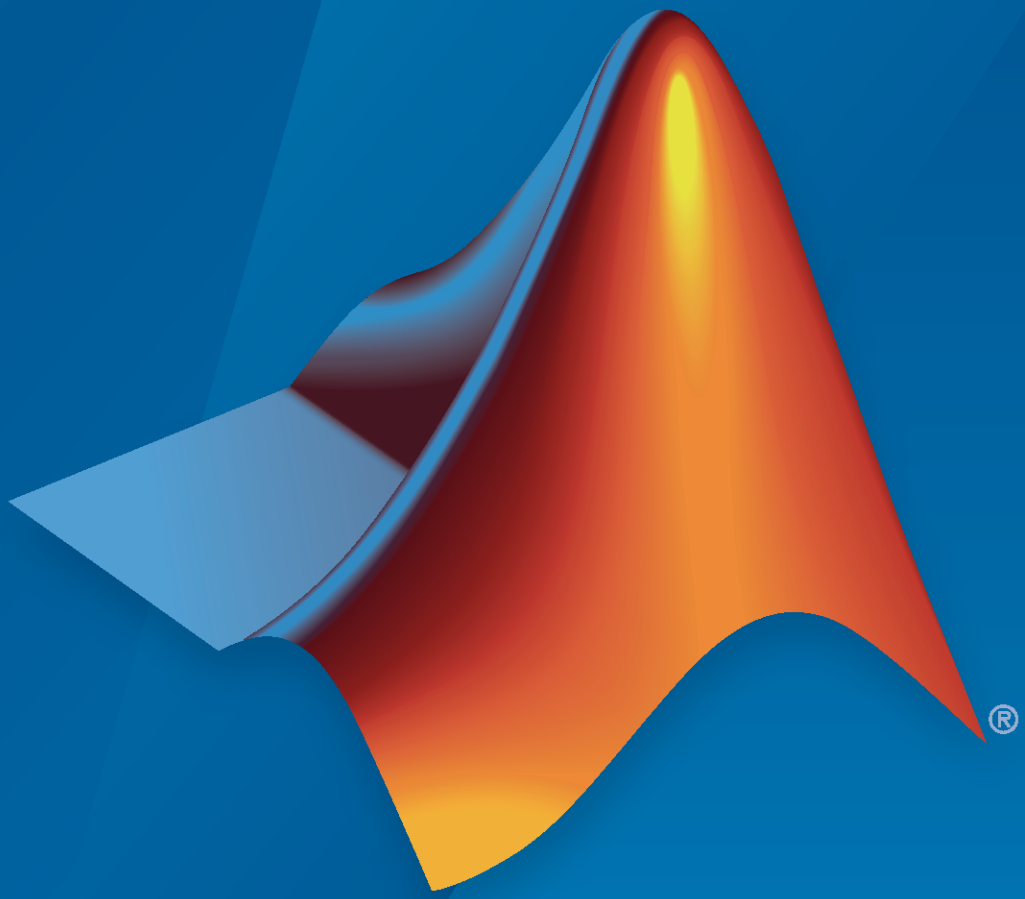
Simulink® Check™

Reference



MATLAB® & SIMULINK®

R2022b

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

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

Simulink® Check™ Reference

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)

Contents

1 Functions

2 Model Advisor Checks

3 Model Transformer Tasks

4 Clone Detection Tasks

5 Model Testing Metrics

6 Model Maintainability Metrics

7 Apps

Functions

Advisor.authoring.CustomCheck.actionCallback

Register action callback for model configuration check

Syntax

```
Advisor.authoring.CustomCheck.actionCallback(task)
```

Description

`Advisor.authoring.CustomCheck.actionCallback(task)` is used as the action callback function when registering custom checks that use an XML data file to specify check behavior.

Examples

This `sl_customization.m` file registers the action callback for configuration parameter checks with fix actions.

```
function defineModelAdvisorChecks

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

    % --- data file input parameters
    rec.setInputParametersLayoutGrid([1 1]);
    inputParam1 = ModelAdvisor.InputParameter;
    inputParam1.Name = 'Data File';
    inputParam1.Value = 'Check1.xml';
    inputParam1.Type = 'String';
    inputParam1.Description = 'Name or full path of XML data file.';
    inputParam1.setRowSpan([1 1]);
    inputParam1.setColSpan([1 1]);
    rec.setInputParameters({inputParam1});

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

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

See Also

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

Topics

“Create Model Advisor Check for Model Configuration Parameters”

addCheck

Class: `ModelAdvisor.FactoryGroup`
Package: `ModelAdvisor`

Add check to folder

Syntax

```
addCheck(fg_obj, check_ID)
```

Description

`addCheck(fg_obj, check_ID)` adds checks, identified by `check_ID`, to the folder specified by `fg_obj`, which is an instantiation of the `ModelAdvisor.FactoryGroup` class.

Examples

Add four checks to `rec`:

```
% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
.
.
.
addCheck(rec, 'com.mathworks.sample.Check0');
addCheck(rec, 'com.mathworks.sample.Check1');
addCheck(rec, 'com.mathworks.sample.Check2');
addCheck(rec, 'com.mathworks.sample.Check3');
```

addGroup

Class: `ModelAdvisor.Group`
Package: `ModelAdvisor`

Add subfolder to folder

Syntax

```
addGroup(group_obj, child_obj)
```

Description

`addGroup(group_obj, child_obj)` adds a new subfolder, identified by `child_obj`, to the folder specified by `group_obj`, which is an instantiation of the `ModelAdvisor.Group` class.

Examples

Add three checks to `rec`:

```
group_obj = ModelAdvisor.Group('com.mathworks.sample.group');
.
.
.
addGroup(group_obj, 'com.mathworks.sample.subgroup1');
addGroup(group_obj, 'com.mathworks.sample.subgroup2');
addGroup(group_obj, 'com.mathworks.sample.subgroup3');
```

To add `ModelAdvisor.Task` objects to a group using `addGroup`:

```
mdladvRoot = ModelAdvisor.Root();

% MAT1, MAT2, and MAT3 are registered ModelAdvisor.Task objects
% Create the group 'My Group'
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');
MAG.DisplayName='My Group';

% Add the first task to the 'My Group' folder
MAG.addTask(MAT1);

% Create a subfolder 'Folder1'
MAGSUB1 = ModelAdvisor.Group('com.mathworks.sample.Folder1');
MAGSUB1.DisplayName='Folder1';

% Add the second task to Folder1
MAGSUB1.addTask(MAT2);

% Create a subfolder 'Folder2'
MAGSUB2 = ModelAdvisor.Group('com.mathworks.sample.Folder2');
MAGSUB2.DisplayName='Folder2';

% Add the third task to Folder2
MAGSUB2.addTask(MAT3);
```

```
% Register the two subfolders. This must be done before calling addGroup
mdladvRoot.register(MAGSUB1);
mdladvRoot.register(MAGSUB2);

% Invoke addGroup to place the subfolders under 'My Group'
MAG.addGroup(MAGSUB1);
MAG.addGroup(MAGSUB2);

mdladvRoot.publish(MAG); % publish under Root
```

setHelp

Class: ModelAdvisor.Group
Package: ModelAdvisor

Set custom help for folders that have custom authored Model Advisor checks

Syntax

```
checkObj.setHelp('Format',format, 'Path', custom path)
```

Description

`checkObj.setHelp('Format',format, 'Path', custom path)` sets custom help for folders that have custom authored Model Advisor checks.

Input Arguments

`format` — Format of the help file

pdf | webpage

Format of the help file, specified as pdf or webpage.

Example: `checkObj.setHelp('Format','webpage','Path','c:/customhelp/help.html');`

Data Types: `char`

`custom path` — Path of the help file

char vector

Path of the help file for the custom check specified as a character vector.

Example: `checkObj.setHelp('Format','pdf','Path','c:/customhelp/help.pdf');`

Data Types: `char`

Examples

Set pdf as custom help to a custom check group

Create a group object.

```
group_obj = ModelAdvisor.Group('mathworks.example.ExampleGroup');
```

set pdf as custom help to your custom check.

```
group_obj.setHelp('Format','pdf','Path','c:/customhelp/help.pdf');
```

Version History

Introduced in R2022a

See Also

`setHelp` | “Create Help for Custom Model Advisor Checks”

addItem

Package: `ModelAdvisor`

Add list items to Model Advisor results

Syntax

```
addItem(obj, elements)
```

Description

`addItem(obj, elements)` adds element items to a `ModelAdvisor.List` object.

Examples

Format Model Advisor Results as List

1. Create a `ModelAdvisor.List` object.

```
subList = ModelAdvisor.List();
```

2. Specify the type as a numbered or bulleted list.

```
setType(subList, 'numbered');
```

3. Add items to the list.

```
addItem(subList, ModelAdvisor.Text('Sub entry 1', {'pass','bold'}));
addItem(subList, ModelAdvisor.Text('Sub entry 2', {'pass','bold'}));
```

Input Arguments

`obj` — Model Advisor list object

`ModelAdvisor.List`

Model Advisor list object for which you want to add elements.

`elements` — Model Advisor list elements

`character vector` | `ModelAdvisor.Text` | `ModelAdvisor.Image` | `ModelAdvisor.Table` | `ModelAdvisor.Paragraph` | `ModelAdvisor.List` | cell array

Elements that appear as part of a list in the Model Advisor results, specified as a character vector, a Model Advisor formatting object, or a cell array of character vectors and Model Advisor formatting objects. When you add a cell array to a list, they form different rows in the list.

Version History

Introduced in R2006b

See Also

`ModelAdvisor.Check`

Topics
“Define Custom Model Advisor Checks”
“Create and Deploy a Model Advisor Custom Configuration”
“Customize the Configuration of the Model Advisor Overview”

addItem

Class: `ModelAdvisor.Paragraph`
Package: `ModelAdvisor`

Add item to paragraph

Syntax

```
addItem(text, element)
```

Description

`addItem(text, element)` adds an element to `text`. `element` is one of the following:

- Character vector
- Element
- Cell array of elements

Examples

Add two lines of text:

```
result = ModelAdvisor.Paragraph;
addItem(result, [resultText1 ModelAdvisor.LineBreak resultText2]);
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

addProcedure

Add check procedure to Model Advisor folder

Syntax

```
addProcedure(GroupObj, ProcedureObjj)
```

Description

`addProcedure(GroupObj, ProcedureObjj)` adds a procedure, specified by `ProcedureObj`, to the folder `GroupObj`. `GroupObj` is an instantiation of the `ModelAdvisor.Group` class. A procedure organizes checks and other procedures by functionality and usage.

Input Arguments

`ProcedureObj` — Check procedure

`ModelAdvisor.Procedure` object

Model Advisor check procedure specified as a `ModelAdvisor.Procedure` object.

`GroupObj` — Check folder

`ModelAdvisor.Group` object

Model Advisor folder specified as a `ModelAdvisor.Procedure` object.

Examples

Add three procedures to `MAG`.

```
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');

MAP1=ModelAdvisor.Procedure('com.mathworks.sample.procedure1');
MAP2=ModelAdvisor.Procedure('com.mathworks.sample.procedure2');
MAP3=ModelAdvisor.Procedure('com.mathworks.sample.procedure3');

addProcedure(MAG, MAP1);
addProcedure(MAG, MAP2);
addProcedure(MAG, MAP3);
```

Version History

Introduced in R2008a

See Also

`ModelAdvisor.Procedure` | `ModelAdvisor.Check`

Topics

“Programmatically Create Procedural-Based Configurations”
“Customize the Configuration of the Model Advisor Overview”

addProcedure

Package: `ModelAdvisor`

Add subprocedure to procedure

Syntax

```
addProcedure(ProcedureObj1, ProcedureObj2)
```

Description

`addProcedure(ProcedureObj1, ProcedureObj2)` adds a procedure, specified by `ProcedureObj2`, to the procedure `ProcedureObj1`. `ProcedureObj2` and `ProcedureObj1` are `ModelAdvisor.Procedure` objects. Add additional procedures to organize checks by functionality or usage.

Examples

Add Procedures to Model Advisor Procedure

1. Create a `ModelAdvisor.Procedure` object named `MAP`.

```
MAP = ModelAdvisor.Procedure('com.mathworks.sample.ProcedureSample');
```

2. Create three subprocedures.

```
MAP1=ModelAdvisor.Procedure('com.mathworks.sample.procedure1');
MAP2=ModelAdvisor.Procedure('com.mathworks.sample.procedure2');
MAP3=ModelAdvisor.Procedure('com.mathworks.sample.procedure3');
```

3. Add three procedures to `MAP`.

```
addProcedure(MAP, MAP1);
addProcedure(MAP, MAP2);
addProcedure(MAP, MAP3);
```

Input Arguments

`ProcedureObj1` — Primary Procedure object

`ModelAdvisor.Procedure` object

Procedure that you want to add a procedure to specified as a `ModelAdvisor.Procedure` object.

`ProcedureObj2` — Secondary procedure object

`ModelAdvisor.Procedure` object

Sub-procedure that you want to add to a procedure to specified as a `ModelAdvisor.Procedure` object.

Version History

Introduced in R2006a

See Also

`ModelAdvisor.Check`

Topics

“Programmatically Create Procedural-Based Configurations”
“Define Custom Model Advisor Checks”
“Customize the Configuration of the Model Advisor Overview”

addRow

Package: `ModelAdvisor`

Add row to table in Model Advisor analysis results

Syntax

```
addRow(ftObj,rowEntries)
```

Description

`addRow(ftObj,rowEntries)` adds a row which contains the contents of `rowEntries` to the end of the table specified by the formatting template object `ftObj`. If you do not add data to a table, Model Advisor does not display the table in the results.

Note Before adding rows to a table, you must specify column titles using the `setColTitles` method.

The function `addRow` is for formatting tables in Model Advisor analysis results with Simulink Check. For more information, see “Simulink Check”.

For information on how to use tables in MATLAB®, see “Create Tables and Assign Data to Them”.

Examples

Add Rows to a Table Template Object

Create a Model Advisor formatting template object, `ft`, of `type 'TableTemplate'`, and add a row to the table.

Use `ModelAdvisor.FormatTemplate` to create a Model Advisor formatting template `ft` of `type 'TableTemplate'`.

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

Specify the table title.

```
setTableTitle(ft,{'Blocks in Model'});
```

Before adding rows to a table, you must specify column titles.

```
setColTitles(ft,{'Index','Block Name'});
```

Open the model `vdp`.

```
vdp
```

Find the blocks in the current system, `vdp`, and add them to the table.

```
allBlocks = find_system('vdp');
for inx = 2:length(allBlocks)
    addRow(ft,{inx-1,allBlocks(inx)});
end
```

Use `addRow` in a check callback function in your `sl_customization` file to format your Model Advisor analysis results.

```
function result = SampleStyleOneCallback(system)
ft = ModelAdvisor.FormatTemplate('TableTemplate');
setTableTitle(ft,{'Blocks in Model'});
setColTitles(ft,{'Index','Block Name'});
allBlocks = find_system('vdp');
for inx = 2:length(allBlocks)
    addRow(ft,{inx-1,allBlocks(inx)});
end
result = ft;
end
```

For more information on how to format check results, see “Define Custom Model Advisor Checks”.

Input Arguments

`ftObj` — Template object

template object handle

`ModelAdvisor.FormatTemplate` object, specified as a handle to the template object.

`rowEntries` — Table row contents

cell array of character vectors | cell array of objects

Table row entries, specified as a cell array of character vectors or cell array of objects. The order of the items in the cell array determines which column the item is in.

Example: {'Item 1', 'Item 2'}

See Also

`ModelAdvisor.FormatTemplate` | `setColTitles` | `setTableInfo`

Topics

“Define Custom Model Advisor Checks”

“Format Model Advisor Results” on page 1-198

addTask

Class: `ModelAdvisor.Group`
Package: `ModelAdvisor`

Add task to folder

Syntax

```
addTask(group_obj, task_obj)
```

Description

`addTask(group_obj, task_obj)` adds a task, specified by `task_obj`, to the folder `group_obj`. `group_obj` is an instantiation of the `ModelAdvisor.Group` class.

Examples

Add three tasks to a `ModelAdvisor.Group` `MAG`.

```
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');
addTask(MAG, MAT8);
addTask(MAG, MAT1);
addTask(MAG, MAT2);
addTask(MAG, MAT3);
```

setHelp

Set custom help for custom authored Model Advisor checks

Syntax

```
checkObj.setHelp('Format',format, 'Path',custom path)
```

Description

`checkObj.setHelp('Format',format, 'Path',custom path)` sets custom help for custom authored Model Advisor checks.

Input Arguments

`format` — Format of the help file

pdf | webpage

Format of the help file, specified as pdf or webpage.

Example: `checkobj.setHelp('Format','webpage','path','c:/customhelp/help.html');`

Data Types: `char`

`custom path` — Path of the help file

char vector

Path of the help file for the custom check specified as a character vector.

Example: `checkobj.setHelp('Format','pdf','path','c:/customhelp/help.pdf');`

Data Types: `char`

Examples

Set webpage as custom help to a custom check

Create a check object.

```
checkobj = ModelAdvisor.Group('SimplePassFailCheck');
```

set webpage as custom help to your custom check.

```
checkobj.setHelp('Format','webpage','Path','c:/customhelp/help.html');
```

Version History

Introduced in R2022a

See Also

`setHelp` | “Create Help for Custom Model Advisor Checks” | “Define Custom Model Advisor Checks” | “Create Model Advisor Check for Model Configuration Parameters”

addTask

Add task to procedure

Syntax

`addTask(procedure_obj, task_obj)`

Description

`addTask(procedure_obj, task_obj)` adds a task, specified by `task_obj`, to `procedure_obj`.`procedure_obj` is an instantiation of the `ModelAdvisor.Procedure` class.

Examples

Add three tasks to `MAP`.

```
MAP = ModelAdvisor.Procedure('com.mathworks.sample.ProcedureSample');

MAT1=ModelAdvisor.Task('com.mathworks.sample.task1');
MAT2=ModelAdvisor.Task('com.mathworks.sample.task2');
MAT3=ModelAdvisor.Task('com.mathworks.sample.task3');

addTask(MAP, MAT1);
addTask(MAP, MAT2);
addTask(MAP, MAT3);
```

Advisor.Application class

Package: `Advisor`

Run Model Advisor across model hierarchy

Description

Use instances of `Advisor.Application` to run Model Advisor checks across a model hierarchy. You can use `Advisor.Application` to:

- Run checks on referenced models.
- Select model components for Model Advisor analysis.
- Select checks to run during Model Advisor analysis.

Consider using `Advisor.Application` if you have a large model with subsystems and model references. `Advisor.Application` does not run checks on library models. If you want to run checks on multiple independent models that are not in a model reference hierarchy or you want to leverage parallel processing, use `ModelAdvisor.run` to run Model Advisor checks on your model.

Note If model references are used in hierarchy, and if the user has set **Simulation Mode** to `accelerator` and **UpdateModelReferenceTargets** to `IfOutOfDate`, then the child model shall compile only if it is out of date. For referenced models, compile time checks will not run.

The `Advisor.Application` methods use the following definitions:

- Model component — Model in the system hierarchy. Models that the root model references and that `setAnalysisroot` specifies are model components.
- Check instance — Instantiation of a `ModelAdvisor.Check` object in the Model Advisor configuration. Each check instance has an instance ID. When you change the Model Advisor configuration, the instance ID can change.

Construction

To create an `Advisor.Application` object, use `Advisor.Manager.createApplication`.

Properties

`AnalysisRoot` — Name of root model in the model hierarchy to analyze

character vector

Name of root model in the model hierarchy to analyze, as specified by the `Advisor.Application.setAnalysisRoot` method. This property is read only.

`ID` — Unique identifier

character vector

Unique identifier for the `Advisor.Application` object. This property is read only.

`UseTempDir` — Run analysis in a temporary working folder
false (default) | true

Run analysis in a temporary working folder. Specified by the `Advisor.Manager.createApplication` method. This property is read only.

Data Types: `logical`

`AnalyzeVariants` — Run analysis on active and inactive variants
false (default) | true

Run analysis on variant blocks in models with predefined configurations with variant choices created using Variant Manager (“Variant Manager for Simulink”). For each configuration, produce a Model Advisor report. This property is read/write.

Data Types: `logical`

Methods

delete	Delete `Advisor.Application` object
deselectCheckInstances	Clear check instances from Model Advisor analysis
deselectComponents	Clear model components from Model Advisor analysis
generateReport	Generate report for Model Advisor analysis
getCheckInstanceIDs	Obtain check instance IDs
getResults	Access Model Advisor analysis results
loadConfiguration	Load Model Advisor configuration
run	Run Model Advisor analysis on model components
selectCheckInstances	Select check instances to use in Model Advisor analysis
selectComponents	Select model components for Model Advisor analysis
setAnalysisRoot	Specify model hierarchy for Model Advisor analysis

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Examples

Run Model Advisor Checks on Referenced Model

This example shows how to run a check on model `sldemo_mdlref_counter` referenced from `sldemo_mdlref_basic`.

In the Command Window, open model `sldemo_mdlref_basic` and referenced model `sldemo_mdlref_counter`.

```
openExample('sldemo_mdlref_basic');
openExample('sldemo_mdlref_counter');
```

Save a copy of the models to a work folder, renaming them to `mdlref_basic` and `mdlref_counter`.

```
save_system('sldemo_mdlref_basic','mdlref_basic');
save_system('sldemo_mdlref_counter','mdlref_counter');
```

In `mdlref_basic`, change model reference from `sldemo_mdlref_counter` to `mdlref_counter`. Save `mdlref_basic`.

```
set_param('mdlref_basic/CounterA','ModelName','mdlref_counter');
set_param('mdlref_basic/CounterB','ModelName','mdlref_counter');
set_param('mdlref_basic/CounterC','ModelName','mdlref_counter');
save_system('mdlref_basic');
```

Set root model to `mdlref_basic`.

```
RootModel='mdlref_basic';
```

Create an `Application` object.

```
app = Advisor.Manager.createApplication();
```

Set root analysis.

```
setAnalysisRoot(app,'Root',RootModel);
```

Clear all check instances from Model Advisor analysis.

```
deselectCheckInstances(app);
```

Select check **Identify unconnected lines, input ports, and output ports** using check instance ID.

```
instanceID = getCheckInstanceIDs(app,'mathworks.design.UnconnectedLinesPorts');
checkinstanceID = instanceID(1);
selectCheckInstances(app,'IDs',checkinstanceID);
```

Run Model Advisor analysis.

```
run(app);
```

Get analysis results.

```
getResults(app);
```

Generate and view the Model Advisor report. The Model Advisor runs the check on both `mdlref_basic` and `mdlref_counter`.

```
report = generateReport(app);
web(report)
```

Close the models.

```
close_system('mdlref_basic');
close_system('mdlref_counter');
```

Run Model Advisor Checks on a Subsystem

This example shows how to run a check on subsystem `fuel_rate_control` referenced from `sldemo_fuelsys`.

In the Command Window, open model `sldemo_fuelsys`.

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

Set root model to `sldemo_fuelsys`.

```
RootModel='sldemo_fuelsys';
```

Create an `Application` object.

```
app = Advisor.Manager.createApplication();
```

Set root analysis to subsystem `sldemo_fuelsys/fuel_rate_control`.

```
setAnalysisRoot(app,'Root','sldemo_fuelsys/fuel_rate_control','RootType','Subsystem');
```

Clear all check instances from Model Advisor analysis.

```
deselectCheckInstances(app);
```

Select check **Identify unconnected lines, input ports, and output ports** using check instance ID.

```
instanceID = getCheckInstanceIDs(app,'mathworks.design.UnconnectedLinesPorts');
checkinstanceID = instanceID(1);
selectCheckInstances(app,'IDs',checkinstanceID);
```

Run Model Advisor analysis.

```
run(app);
```

Get analysis results.

```
getResults(app);
```

Generate and view the Model Advisor report. The Model Advisor runs the check on subsystem `sldemo_fuelsys/fuel_rate_control`.

```
report = generateReport(app);
web(report)
```

Close the model.

```
close_system('sldemo_fuelsys');
```

Version History

Introduced in R2015b

See Also

Topics

Class Attributes
Property Attributes

ModelAdvisor.CheckResult

Access Model Advisor check results

Description

View Model Advisor check results obtained by using the `run` method.

Creation

To access the properties of a `CheckResult` object, access the `CheckResultsObjs` property of a `System.Result` object.

Properties

`system` — Model or subsystem that Model Advisor analyzes
character vector

Model or subsystem that the Model Advisor analyzes.

Data Types: `char`

`status` — Model Advisor check status
`"Failed"` | `"Incomplete"` | `"Justified"` | `"NotRun"` | `"Passed"` | `"Warning"`

Status of Model Advisor check

Data Types: `string`

`checkID` — Model Advisor check ID
character vector

Model Advisor check ID

Data Types: `char`

`checkName` — Model Advisor check name
character vector

Name of Model Advisor check

Data Types: `char`

Examples

Run Model Advisor Checks on Referenced Model

This example shows how to run a check on model `sldemo_mdlref_counter` referenced from `sldemo_mdlref_basic`.

In the Command Window, open model `sldemo_mdlref_basic` and referenced model `sldemo_mdlref_counter`.

```
openExample('sldemo_mdlref_basic');
openExample('sldemo_mdlref_counter');
```

Save a copy of the models to a work folder, renaming them to `mdlref_basic` and `mdlref_counter`.

```
save_system('sldemo_mdlref_basic','mdlref_basic');
save_system('sldemo_mdlref_counter','mdlref_counter');
```

In `mdlref_basic`, change model reference from `sldemo_mdlref_counter` to `mdlref_counter`. Save `mdlref_basic`.

```
set_param('mdlref_basic/CounterA','ModelName','mdlref_counter');
set_param('mdlref_basic/CounterB','ModelName','mdlref_counter');
set_param('mdlref_basic/CounterC','ModelName','mdlref_counter');
save_system('mdlref_basic');
```

Set root model to `mdlref_basic`.

```
RootModel='mdlref_basic';
```

Create an `Application` object.

```
app = Advisor.Manager.createApplication();
```

Set root analysis.

```
setAnalysisRoot(app,'Root',RootModel);
```

Clear check instances from Model Advisor analysis.

```
deselectCheckInstances(app);
```

Select check **Identify unconnected lines, input ports, and output ports** using check instance ID.

```
instanceID = getCheckInstanceIDs(app,'mathworks.design.UnconnectedLinesPorts');
checkinstanceID = instanceID(1);
selectCheckInstances(app,'IDs',checkinstanceID);
```

Run Model Advisor analysis.

```
run(app);
```

Get analysis results and view the properties of the `ModelAdvisor.SystemResult` and `ModelAdvisor.CheckResult` objects.

```
getResults(app);
```

Close the models.

```
close_system('mdlref_basic');
close_system('mdlref_counter');
```

Version History

Introduced in R2015b

`status` property uses string values
Behavior changed in R2022b

The valid values for the `status` property were updated to the string values:

- `"Failed"`
- `"Incomplete"`
- `"Justified"`
- `"NotRun"`
- `"Passed"`
- `"Warning"`

`status` property has different values
Behavior changed in R2022a

The valid values for the `status` property were updated to the enumerated values:

- `Failed`
- `Incomplete`
- `Justified`
- `NotRun`
- `Passed`
- `Warning`

`status` property has different value for warnings
Behavior changed in R2021a

The `status` property uses a value of:

- `'Warning'` in the check result from `ModelAdvisor.run`
- `'Warn'` in the check result from `Advisor.Application.run`

See Also

`Advisor.Manager`

ModelAdvisor.SystemResult

Access system-level Model Advisor results

Description

View Model Advisor results obtained by using the `run` method.

Creation

To access the properties of a `System.Result` object, apply the `getResults` method to an `Advisor.Application` object.

Properties

`system` — Model or subsystem that Model Advisor analyzes
character vector

Model or subsystem that the Model Advisor analyzes.

Data Types: `char`

Type — Model component
'Library' | 'Model' | 'Subsystem'

Type of model component that Model Advisor analyzes.

Data Types: `char`

`numPass` — Number of Model Advisor checks that pass
double

Obtain the number of Model Advisor checks that pass after running the Model Advisor using the `run` method.

Data Types: `double`

`numFail` — Number of Model Advisor checks that fail
double

Obtain the number of Model Advisor checks that fail after running the Model Advisor using the `run` method.

Data Types: `double`

`numNotRun` — Number of Model Advisor checks that do not run
double

Obtain the number of Model Advisor checks that do not run after running the Model Advisor using the `run` method.

Data Types: `double`

numWarn — Number of Model Advisor checks that warn
double

Obtain the number of Model Advisor checks that warn after running the Model Advisor using the `run` method.

Number of Model Advisor checks that warn.

Data Types: `double`

CheckResultObjs — Model Advisor Check results
cell array of ModelAdvisor.CheckResult objects

Cell array containing a `ModelAdvisor.CheckResult` object for each Model Advisor check. Access these object properties to obtain Model Advisor check results.

Data Types: `double`

Examples

Run Model Advisor Checks on Referenced Model

This example shows how to run a check on model `sldemo_mdlref_counter` referenced from `sldemo_mdlref_basic`.

In the Command Window, open model `sldemo_mdlref_basic` and referenced model `sldemo_mdlref_counter`.

```
openExample('sldemo_mdlref_basic');
openExample('sldemo_mdlref_counter');
```

Save a copy of the models to a work folder, renaming them to `mdlref_basic` and `mdlref_counter`.

```
save_system('sldemo_mdlref_basic','mdlref_basic');
save_system('sldemo_mdlref_counter','mdlref_counter');
```

In `mdlref_basic`, change model reference from `sldemo_mdlref_counter` to `mdlref_counter`. Save `mdlref_basic`.

```
set_param('mdlref_basic/CounterA','ModelName','mdlref_counter');
set_param('mdlref_basic/CounterB','ModelName','mdlref_counter');
set_param('mdlref_basic/CounterC','ModelName','mdlref_counter');
save_system('mdlref_basic');
```

Set root model to `mdlref_basic`.

```
RootModel='mdlref_basic';
```

Create an `Application` object.

```
app = Advisor.Manager.createApplication();
```

Set root analysis.

```
setAnalysisRoot(app,'Root',RootModel);
```

Clear check instances from Model Advisor analysis.

```
deselectCheckInstances(app);
```

Select check **Identify unconnected lines, input ports, and output ports** using check instance ID.

```
instanceID = getCheckInstanceIDs(app,'mathworks.design.UnconnectedLinesPorts');
checkinstanceID = instanceID(1);
selectCheckInstances(app,'IDs',checkinstanceID);
```

Run Model Advisor analysis.

```
run(app);
```

Get analysis results and view the properties of the `ModelAdvisor.SystemResult` and `ModelAdvisor.CheckResult` objects.

```
Results=getResults(app);
```

Generate and view the Model Advisor report. The Model Advisor runs the check on both `mdlref_basic` and `mdlref_counter`.

```
report = generateReport(app);
web(report)
```

Close the models.

```
close_system('mdlref_basic');
close_system('mdlref_counter');
```

Version History

Introduced in R2015b

See Also

`Advisor.Manager`

Advisor.authoring.generateConfigurationParameterDataFile

Package: `Advisor.authoring`

Generate XML data file for custom configuration parameter check

Syntax

```
Advisor.authoring.generateConfigurationParameterDataFile(dataFile,source)
Advisor.authoring.generateConfigurationParameterDataFile(dataFile,source,
Name,Value)
```

Description

`Advisor.authoring.generateConfigurationParameterDataFile(dataFile,source)` generates an XML data file named `dataFile` specifying the configuration parameters for `source`. The data file uses tagging to specify the configuration parameter settings you want. When you create a check for configuration parameters, you use the data file. Each model configuration parameter specified in the data file is a subcheck.

`Advisor.authoring.generateConfigurationParameterDataFile(dataFile,source, Name,Value)` generates an XML data file named `dataFile` specifying the configuration parameters for `source`. It also specifies additional options by one or more optional `Name,Value` arguments. The data file uses tagging to specify the configuration parameter settings you want. When you create a check for configuration parameters, you use the data file. Each model configuration parameter specified in the data file is a subcheck.

Examples

Create data file for configuration parameter check

Create a data file with all the configuration parameters. You use the data file to create a configuration parameter.

```
model = 'vdp';
dataFile = 'myDataFile.xml';
Advisor.authoring.generateConfigurationParameterDataFile( ...
               dataFile, model);
```

Data file `myDataFile.xml` has tagging specifying subcheck information for each configuration parameter. `myDataFile.xml` specifies the configuration parameters settings you want. The following specifies XML tagging for configuration parameter `AbsTol`. If the configuration parameter is set to `1e-6`, the configuration parameter subcheck specified in `myDataFile.xml` passes.

```
<!-- Absolute tolerance: (AbsTol)-->
  <PositiveModelParameterConstraint>
     <parameter>AbsTol</parameter>
```

```
    <value>1e-6</value>
    </PositiveModelParameterConstraint>
```

Create data file for Solver pane configuration parameter check with fix action

Create a data file with configuration parameters for the **Solver** pane. You use the data file to create a **Solver** pane configuration parameter check with fix actions.

```
model = 'vdp';
dataFile = 'myDataFile.xml';
Advisor.authoring.generateConfigurationParameterDataFile( ...
               dataFile, model, 'Pane', 'Solver', 'FixValues', true);
```

Data file `myDataFile.xml` has tagging specifying subcheck information for each configuration parameter. `myDataFile.xml` specifies the configuration parameters settings that you want. The following specifies XML tagging for configuration parameter `AbsTol`. If the configuration parameter is set to `1e-6`, the configuration parameter subcheck specified in `myDataFile.xml` passes. If the subcheck does not pass, the check fix action modifies the configuration parameter to `1e-6`.

```
<!-- Absolute tolerance: (AbsTol)-->
  <PositiveModelParameterConstraint>
     <parameter>AbsTol</parameter>
     <value>1e-6</value>
     <fixvalue>1e-6</fixvalue>
     </PositiveModelParameterConstraint>
```

Input Arguments

`dataFile` — Name of data file to create

character vector

Name of XML data file to create, specified as a character vector.

Example: `'myDataFile.xml'`

`source` — Name of model or configuration set

character vector | `Simulink.ConfigSet`

Name of model or `Simulink.ConfigSet` object used to specify configuration parameters

Example: 'vdp'

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Pane', 'Solver', 'FixValues', true` specifies a `dataFile` with Solver pane configuration parameters and fix tagging.

Pane — Limit the configuration parameters in the dataFile
`Solver` | `Data Import/Export` | `Optimization` | `Diagnostics` | `Hardware Implementation` | `Model Referencing` | `Code Generation`

Option to limit the configuration parameters in the data file to the pane specified as the comma-separated pair of `'Pane'` and one of the following:

- `Solver`
- `Data Import/Export`
- `Optimization`
- `Diagnostics`
- `Hardware Implementation`
- `Model Referencing`
- `Code Generation`

Example: `'Pane','Solver'` limits the `dataFile` to configuration parameters on the Solver pane.

Data Types: `char`

FixValues — Create fix tagging in the dataFile
false | true

Setting `FixValues` to `true` provides the `dataFile` with fix tagging. When you generate a custom configuration parameter check using a `dataFile` with fix tagging, each configuration parameter subcheck has a fix action. Specified as the comma-separated pair of `'FixValues'` and either `true` or `false`.

Example: `'FixValues,true` specifies fix tagging in the `dataFile`.

Data Types: `logical`

Version History
Introduced in R2014a

See Also

Topics
“Create Model Advisor Check for Model Configuration Parameters”
“Data File for Configuration Parameter Check”

Advisor.authoring.generateBlockConstraintsDataFile

Package: `Advisor.authoring`

(Not recommended) Generate XML data file for custom check for block constraints

---

Note `Advisor.authoring.generateBlockConstraintsDataFile` is not recommended. Use `Advisor.authoring.createBlockConstraintCheck` instead.

For more information, see “Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters”.

Syntax

```
Advisor.authoring.generateBlockConstraintsDataFile(dataFile, 'constraints',
constraintslist)
```

Description

`Advisor.authoring.generateBlockConstraintsDataFile(dataFile, 'constraints', constraintslist)` generates an XML data file named `dataFile`. This data file specifies the constraints that a custom check contains. The data file uses tagging to specify the constraint information. When you create a custom check, you use the data file.

Define constraint objects in the base workspace and then pass these objects as inputs to this function. These constraints may be root constraints and prerequisites to root constraints. You can also define a composite constraint. If you specify multiple root constraints and no composite constraint, Simulink implements a composite constraint with a `CompositeOperator` of `and`.

Examples

Create Data File for Custom Check for Block Constraints

Create a custom check for this MAB check “Check settings for data ports in Multiport Switch blocks” on page 2-203. For Multiport Switch blocks, the check contains a constraint that checks that the **Data port order** parameter setting is `Specify indices`. If the parameter has this setting, there are constraints that check that the **Data port for default case** parameter setting is `Additional data port` and the **Diagnostic for default case** setting is `None`.

Create three `PositiveBlockParameter` constraint objects.

```
c1 = Advisor.authoring.PositiveBlockParameterConstraint();
c1.ID = 'ID_A2';
c1.BlockType = 'MultiPortSwitch';
c1.ParameterName = 'DataPortOrder';
c1.SupportedParameterValues = {'Specify indices'};
c1.ValueOperator = 'eq';

c2 = Advisor.authoring.PositiveBlockParameterConstraint();
c2.ID = 'ID_A3';
c2.BlockType = 'MultiPortSwitch';
```

```
c2.ParameterName = 'DataPortForDefault';
c2.SupportedParameterValues = {'Additional data port'};
c2.ValueOperator = 'eq';

c3 = Advisor.authoring.PositiveBlockParameterConstraint();
c3.ID = 'ID_A4';
c3.BlockType = 'MultiPortSwitch';
c3.ParameterName = 'DiagnosticForDefault';
c3.SupportedParameterValues = {'None'};
c3.ValueOperator = 'eq';
```

Use the `addPreRequisiteConstraintID` method to make `c1` a prerequisite to checking constraints `c2` and `c3`.

```
c2.addPreRequisiteConstraintID('ID_A2');
c3.addPreRequisiteConstraintID('ID_A2');
```

Create a composite constraint that specifies that if a Multiport Switch block does not meet constraints `c2` and `c3`, the block is in violation of this check.

```
cc = Advisor.authoring.CompositeConstraint();
cc.addConstraintID('ID_A3');
cc.addConstraintID ('ID_A4');
cc.CompositeOperator = 'and';
```

Create a data file that contains the constraints.

```
dataFile = 'myDataFile.xml';
Advisor.authoring.generateBlockConstraintsDataFile( ...
               dataFile,'constraints',{c1,c2,c3,cc});
```

Data file `myDataFile.xml` has tagging specifying the constraint information for the custom check.

```
<?xml version="1.0" encoding="utf-8"?>
<customcheck>
   <checkdata>
      <PositiveBlockParameterConstraint BlockType="MultiPortSwitch" id="ID_A2">
         <parameter type="string">DataPortOrder</parameter>
         <value>Specify indices</value>
         <operator>eq</operator>
      </PositiveBlockParameterConstraint>
      <PositiveBlockParameterConstraint BlockType="MultiPortSwitch" id="ID_A3">
         <parameter type="string">DataPortForDefault</parameter>
         <value>Additional data port</value>
         <operator>eq</operator>
         <dependson>ID_A2</dependson>
      </PositiveBlockParameterConstraint>
      <PositiveBlockParameterConstraint BlockType="MultiPortSwitch" id="ID_A4">
         <parameter type="string">DiagnosticForDefault</parameter>
         <value>None</value>
         <operator>eq</operator>
         <dependson>ID_A2</dependson>
      </PositiveBlockParameterConstraint>
      <CompositeConstraint>
         <ID>ID_A3</ID>
         <ID>ID_A4</ID>
         <operator>and</operator>
      </CompositeConstraint>
   </checkdata>
</customcheck>
```

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

Specify the data file `myDataFile.xml` as an input to the check definition function. To specify and register this check, use this `sl_customization.m` file.

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

% Copyright 2019 The MathWorks, Inc.

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

end

% ----------------------------
% defines Model Advisor Check
% ----------------------------
function defineBlockConstraintCheck

rec = Advisor.authoring.createBlockConstraintCheck('com.mathworks.sample.blockConstraint');
rec.Title = 'Example of block paramter constraints check';
rec.setCallbackFcn(@(system)(Advisor.authoring.CustomCheck.checkCallback...
    (system)), 'None', 'StyleOne');
rec.TitleTips = 'Example check for block paramter constraints';

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

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

end
```

Input Arguments

`dataFile` — Name of data file to create

character vector

Name of XML data file to create, specified as a character vector.

Example: `'myDataFile.xml'`

`constraintslist` — cell array of constraint objects

cell array of objects

Use these classes to create constraint objects:

- `Advisor.authoring.PositiveBlockParameterConstraint`
- `Advisor.authoring.NegativeBlockParameterConstraint`
- `Advisor.authoring.PositiveModelParameterConstraint`
- `Advisor.authoring.NegativeModelParameterConstraint`
- `Advisor.authoring.PositiveBlockTypeConstraint`
- `Advisor.authoring.NegativeBlockTypeConstraint`
- `Advisor.authoring.CompositeConstraint`

Example: {c1,c2,c3}

Version History

Introduced in R2018a

`Advisor.authoring.generateBlockConstraintsDataFile` is not recommended.
Not recommended starting in R2021b

In previous releases, when authoring a block constraint check, you had to create a separate XML file with the block constraints data and then specify the properties of this XML file as part of the check definition function. Starting in R2021b, the constraint creation is part of the block constraint check definition. Consequently, the `Advisor.authoring.generateBlockConstraintsDataFile` function is not required and the `Advisor.authoring.createBlockConstraintCheck` function has a `'Constraints'` name-value argument that accepts a callback to a constraints creation function.

For more information, see “Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters”.

There are no plans to remove `Advisor.authoring.generateBlockConstraintsDataFile`.

See Also

`Advisor.authoring.createBlockConstraintCheck` | `PositiveBlockParameterConstraint` | `NegativeBlockParameterConstraint` | `PositiveBlockTypeConstraint` | `NegativeBlockTypeConstraint` | `CompositeConstraint`

Topics
“Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters”

Advisor.authoring.createBlockConstraintCheck

Package: `Advisor.authoring`

Create Model Advisor check with block constraints

Syntax

```
check_obj = Advisor.authoring.createBlockConstraintCheck(
check_ID,'Constraints',@handle)
check_obj = Advisor.authoring.createBlockConstraintCheck(check_ID)
```

Description

`check_obj = Advisor.authoring.createBlockConstraintCheck( check_ID,'Constraints',@handle)` creates a `ModelAdvisor.check` object, `check_obj`, assigns it the identifier `check_ID`, and specifies block constraints in the constraints creation function `@handle`.

`check_obj = Advisor.authoring.createBlockConstraintCheck(check_ID)` creates a `ModelAdvisor.check` object, `check_obj`, and assigns it the identifier `check_ID`. Use the `Advisor.authoring.generateBlockConstraintsDataFile` function to create the block constraints data file. Specify the block constraints data file as an input parameter to the `ModelAdvisor.check` object using the `setInputParameters` function. For more information, see `Advisor.authoring.generateBlockConstraintsDataFile`.

Note The `ModelAdvisor.Check` object created by `Advisor.authoring.createBlockConstraintCheck` does not support setting exclusions.

Examples

Create Model Advisor Check from Constraints

This example shows how to specify and register a Model Advisor constraint check. In the function `newCheck`, the `Advisor.authoring.createBlockConstraintCheck` function creates the `ModelAdvisor.check` object `rec` using the constraints created by the function `createConstraints`.

```
function newCheck()

    rec = Advisor.authoring.createBlockConstraintCheck(...
    'mathworks.check_0001',...
    'Constraints',@createConstraints);

    rec.Title = 'Example1: Check block parameter constraints';
    rec.TitleTips = 'Example check block parameter constraints';

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


end
```

```
function constraints = createConstraints()

    c1=Advisor.authoring.PositiveBlockParameterConstraint;
    c1.ID='ID_1';
    c1.BlockType='Constant';
    c1.ParameterName='Value';
    c1.SupportedParameterValues={'1'};
    c1.ValueOperator='eq';

    c2=Advisor.authoring.PositiveBlockParameterConstraint;
    c2.ID='ID_2';
    c2.BlockType='Gain';
    c2.ParameterName='Gain';
    c2.SupportedParameterValues={'1'};
    c2.ValueOperator='gt';

    constraints = {c1,c2};

end
```

Input Arguments

`check_ID` — Name of Model Advisor check

character vector

Name of model advisor check, specified as a character vector

Example: `'com.mathworks.sample.Check1'`

`@handle` — Constraints creation function

function handle

Constraints creation function, specified as a function handle.

Example: `@createConstraints`

Output Arguments

`check_obj` — Model Advisor check

`ModelAdvisor.check` object

Model Advisor check, returned as a `ModelAdvisor.check` object

Version History

Introduced in R2018a

See Also

`Advisor.authoring.generateBlockConstraintsDataFile` | `PositiveBlockParameterConstraint` | `NegativeBlockParameterConstraint` | `PositiveBlockTypeConstraint` | `NegativeBlockTypeConstraint` | `CompositeConstraint`

Topics

“Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters”

Advisor.authoring.CustomCheck

Define custom configuration parameter check

Description

Instances of the `Advisor.authoring.CustomCheck` class provide a container for static methods used as callback functions when defining a configuration parameter check. The configuration parameter check is defined in an XML data file.

Object Functions

Advisor.authoring.CustomCheck.actionCallback	Register action callback for model configuration check
Advisor.authoring.CustomCheck.checkCallback	Register check callback for model configuration check

Version History

Introduced in R2014a

See Also

`Advisor.authoring.DataFile` |
`Advisor.authoring.generateConfigurationParameterDataFile`

Topics

“Create Model Advisor Check for Model Configuration Parameters”

Advisor.authoring.DataFile

Interact with data file for model configuration checks

Description

The `Advisor.authoring.DataFile` class provides a container for a static method used when interacting with the data file for configuration parameter checks.

Object Functions

Advisor.authoring.DataFile.validate Validate XML data file used for model configuration check

Version History

Introduced in R2014a

See Also

`Advisor.authoring.CustomCheck` |
`Advisor.authoring.generateConfigurationParameterDataFile`

Topics

“Create Model Advisor Check for Model Configuration Parameters”

Advisor.Manager class

Package: `Advisor`

Manage applications

Description

The `Advisor.Manager` class defines application objects.

Methods

createApplication	Create `Advisor.Application` object
getApplication	Return handle to `Advisor.Application` object
refresh_customizations	Refresh Model Advisor check information cache

Copy Semantics

Handle. To learn how handle classes affect copy operations, see Copying Objects.

Version History

Introduced in R2015b

See Also

Topics
Class Attributes
Property Attributes

Advisor.authoring.CustomCheck.checkCallback

Register check callback for model configuration check

Syntax

```
Advisor.authoring.CustomCheck.checkCallback(system, CheckObj)
```

Description

`Advisor.authoring.CustomCheck.checkCallback(system, CheckObj)` is used as the check callback function when registering custom checks that use an XML data file to specify check behavior.

Examples

In the following example, the `sl_customization.m` file registers a configuration parameter check using `Advisor.authoring.CustomCheck.checkCallback(system)`.

```
function defineModelAdvisorChecks

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

    % --- data file input parameters
    rec.setInputParametersLayoutGrid([1 1]);
    inputParam1 = ModelAdvisor.InputParameter;
    inputParam1.Name = 'Data File';
    inputParam1.Value = 'Check1.xml';
    inputParam1.Type = 'String';
    inputParam1.Description = 'Name or full path of XML data file.';
    inputParam1.setRowSpan([1 1]);
    inputParam1.setColSpan([1 1]);
    rec.setInputParameters({inputParam1});

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

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

See Also

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

Topics

“Create Model Advisor Check for Model Configuration Parameters”

Advisor.Manager.createApplication

Class: `Advisor.Manager`
Package: `Advisor`

Create `Advisor.Application` object

Syntax

```
app = Advisor.Manager.createApplication()
app = Advisor.Manager.createApplication(Name,Value)
```

Description

`app = Advisor.Manager.createApplication()` constructs an `Advisor.Application` object.

`app = Advisor.Manager.createApplication(Name,Value)` constructs an `Advisor.Application` object that operates in a temporary working folder.

Examples

Use Application Object to Run Model Advisor Analysis

Create an `Application` object and run Model Advisor analysis on the object.

Create an `Application` object.

```
app = Advisor.Manager.createApplication();
```

Open the model and set the root analysis to `RootModel`.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to sldemo_mdlref_basic model
RootModel = 'sldemo_mdlref_basic';

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);
```

Run Model Advisor analysis.

```
run(app);
```

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'UseTempDir',true` specifies that `Advisor.Application` object operates in a temporary working folder.

`UseTempDir` — Create `Advisor.Application` object that operates in a temporary working folder
false (default) | true

Data Types: `logical`

Output Arguments

app — Application
`Advisor.Application` object

Constructed `Advisor.Application` object.

Version History

Introduced in R2015b

See Also

`Advisor.Application` | `getApplication`

delete

Class: `Advisor.Application`
Package: `Advisor`

Delete `Advisor.Application` object

Syntax

```
delete(app)
```

Description

`delete(app)` deletes the `Application` object when you close the root model specified using `Advisor.Application.setAnalysisRoot`. `Application` objects are implicitly closed.

Examples

Create and delete an `Application` object

Use `delete` to delete an `Application` object.

Create an `Application` object.

```
app = Advisor.Manager.createApplication();
```

Delete the `Application` object.

```
delete(app);
```

Input Arguments

`app` — `Advisor.Application` object to destroy

handle

`Advisor.Application` object to destroy, as specified by `Advisor.Manager.createApplication`.

Version History

Introduced in R2015b

See Also

`Advisor.Manager.createApplication` | `setAnalysisRoot`

deselectCheckInstances

Class: `Advisor.Application`
Package: `Advisor`

Clear check instances from Model Advisor analysis

Syntax

```
deselectCheckInstances(app)
deselectCheckInstances(app,Name,Value)
```

Description

You can clear check instances from Model Advisor analysis. A check instance is an instantiation of a `ModelAdvisor.Check` object in the Model Advisor configuration. When you change the Model Advisor configuration, the check instance ID might change. To obtain the check instance ID, use the `getCheckInstanceIDs` method.

`deselectCheckInstances(app)` clears all check instances from Model Advisor analysis.

`deselectCheckInstances(app,Name,Value)` clears check instances specified by `Name,Value` pair arguments from Model Advisor analysis.

Input Arguments

app — Application

`Advisor.Application` object

`Advisor.Application` object, created by `Advisor.Manager.createApplication`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

IDs — Checks instance IDs

cell array

Check instances to clear from Model Advisor analysis, as specified by a cell array of IDs

Data Types: `cell`

Examples

Clear All Check Instances from Model Advisor Analysis

This example shows how to set the root model, create an `Application` object, set root analysis, and clear checks instances from Model Advisor analysis.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to sldemo_mdlref_basic model
RootModel='sldemo_mdlref_basic';

% Create an Application object
app = Advisor.Manager.createApplication();

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);

% Deselect all checks
deselectCheckInstances(app);
```

Clear Check Instance from Model Advisor Analysis Using Instance ID

This example shows how to set the root model, create an `Application` object, set root analysis, and deselect checks instances using instance IDs.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to sldemo_mdlref_basic model
RootModel='sldemo_mdlref_basic';

% Create an Application object
app = Advisor.Manager.createApplication();

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);

% Deselect "Identify unconnected lines, input ports, and output
% ports" check using instance ID
instanceID = getCheckInstanceIDs(app,'mathworks.design.UnconnectedLinesPorts');
checkinstanceID = instanceID(1);
deselectCheckInstances(app,'IDs',checkinstanceID);
```

Version History

Introduced in R2015b

See Also

`Advisor.Application` | `Advisor.Manager.createApplication` | `setAnalysisRoot` | `getCheckInstanceIDs` | `selectCheckInstances`

deselectComponents

Class: `Advisor.Application`
Package: `Advisor`

Clear model components from Model Advisor analysis

Syntax

```
deselectComponents(app)
deselectComponents(app,Name,Value)
```

Description

You can clear model components from Model Advisor analysis. A model component is a model in the system hierarchy. Models that the root model references and that `Advisor.Application.setAnalysisRoot` specifies are model components.

`deselectComponents(app)` clears all components from Model Advisor analysis.

`deselectComponents(app,Name,Value)` clears model components specified by `Name,Value` pair arguments from Model Advisor analysis.

Input Arguments

app — Application

`Advisor.Application` object

`Advisor.Application` object, created by `Advisor.Manager.createApplication`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

IDs — Component IDs

cell array

Components to clear from Model Advisor analysis, as specified by a cell array of IDs

Data Types: `cell`

HierarchicalSelection — Clear component and component children

false (default) | true

Clear components specified by IDs and component children from Model Advisor analysis

Data Types: `logical`

Examples

Clear All Components from Model Advisor Analysis

This example shows how to set the root model, create an `Application` object, set root analysis, and clear all components from Model Advisor analysis.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to  sldemo_mdlref_basic model
RootModel='sldemo_mdlref_basic';

% Create an Application object
app = Advisor.Manager.createApplication();

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);

% Deselect all components
deselectComponents(app);
```

Clear Components from Model Advisor Analysis Using IDs

This example shows how to set the root model, create an `Application` object, set root analysis, and clear model components using IDs.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to  sldemo_mdlref_basic model
RootModel='sldemo_mdlref_basic';

% Create an Application object
app = Advisor.Manager.createApplication();

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);

% Deselect component using IDs
deselectComponents(app,'IDs',RootModel);
```

Version History

Introduced in R2015b

See Also

`Advisor.Manager.createApplication` | `setAnalysisRoot` | `selectComponents`

generateReport

Class: `Advisor.Application`
Package: `Advisor`

Generate report for Model Advisor analysis

Syntax

```
generateReport(app)
generateReport(app,Name,Value)
```

Description

Generate a Model Advisor report for an `Application` object analysis.

`generateReport(app)` generates a Model Advisor report for each component specified by the `Application` object. By default, a report with the name of the analysis root is generated in the current folder.

`generateReport(app,Name,Value)` generates a Model Advisor report for each component specified by the `Application` object. Use the `Name,Value` pairs to specify the location and name of the report.

Input Arguments

app — Application

`Advisor.Application` object

`Advisor.Application` object, created by `Advisor.Manager.createApplication`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

`Location` — Path to report location

character vector

`Name` — Report name

character vector

Examples

Generate Report

This example shows how to generate a report with the analysis root name in the current folder.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to  sldemo_mdlref_basic model
RootModel='sldemo_mdlref_basic';

% Create an Application object
app = Advisor.Manager.createApplication();

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);

% Run Model Advisor analysis
run(app);

% Generate report
report = generateReport(app);

% Open the report in web browser
web(report);
```

Generate Report with Specified Name and Location

This example shows how to generate a report with a specified name and location.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to  sldemo_mdlref_basic model
RootModel='sldemo_mdlref_basic';

% Create an Application object
app = Advisor.Manager.createApplication();

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);

% Run Model Advisor analysis
run(app);

% Generate report in my_work directory
mkdir my_work
report = generateReport(app,'Location','my_work','Name','RootModelReport');

%Open the report in web browser
web(report);
```

Version History

Introduced in R2015b

See Also

`Advisor.Application` | `Advisor.Manager.createApplication` | `setAnalysisRoot` | `run`

getApplication

Class: `Advisor.Manager`
Package: `Advisor`

Return handle to `Advisor.Application` object

Syntax

```
app = getApplication(Name,Value)
```

Description

`app = getApplication(Name,Value)` returns the handle to an `Advisor.Application` object by using the object properties.

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Id',appID` returns handle to an `Advisor.Application` using the object ID.

`Id` — `Advisor.Application` object ID
`Advisor.Application` object

Data Types: `function_handle`

`Root` — Root model name
character vector

Data Types: `char`

`RootType` — Type of root analysis
'`Model`' (default) | '`Subsystem`'

Data Types: `char`

Output Arguments

`app` — Handle to `Advisor.Application` object
`Advisor.Application` object

Data Types: `function_handle`

Version History

Introduced in R2015b

See Also

`Advisor.Application` | `Advisor.Manager.createApplication`

getCheckInstanceIDs

Class: `Advisor.Application`
Package: `Advisor`

Obtain check instance IDs

Syntax

```
CheckInstanceIDs = getCheckInstanceIDs(app)
CheckInstanceIDs = getCheckInstanceIDs(app,CheckID)
```

Description

Obtain the check instance ID for a check using the check ID. A check instance is an instantiation of a `ModelAdvisor.Check` object in the Model Advisor configuration. When you change the Model Advisor configuration, the check instance ID might change. The check ID is a static identifier that does not change.

`CheckInstanceIDs = getCheckInstanceIDs(app)` returns a cell array of IDs.

`CheckInstanceIDs = getCheckInstanceIDs(app,CheckID)` returns a instance ID for a check.

Input Arguments

app — Application
`Advisor.Application` object

`Advisor.Application` object, created by `Advisor.Manager.createApplication`

`CheckID` — Check ID associated with Model Advisor check
character vector

Check ID associated with Model Advisor check.

Example: `'mathworks.design.UnconnectedLinesPorts'`

Output Arguments

`CheckInstanceIDs` — Cell array of check instance IDs
cell array

Check instance IDs, returned as a cell array of IDs

Examples

Obtain Check Instance IDs

This example shows how to set the root model, create an `Application` object, set root analysis, and obtain the check instance ID.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to sldemo_mdlref_basic model
RootModel='sldemo_mdlref_basic';

% Create an Application object
app = Advisor.Manager.createApplication();

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);

% Select all check instances
selectCheckInstances(app);

% Obtain check instance IDs
CheckInstanceIDs = getCheckInstanceIDs(app);
```

Obtain Check Instance ID for a Check

This example shows how to set the root model, create an `Application` object, set root analysis, and obtain the check instance ID for check **Identify unconnected lines, input ports**.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to sldemo_mdlref_basic model
RootModel='sldemo_mdlref_basic';

% Create an Application object
app = Advisor.Manager.createApplication();

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);

% Select all check instances
selectCheckInstances(app);

%  Obtain check instance ID for Model Advisor check "Identify unconnected lines,
%   input ports"
CheckInstanceIDs = getCheckInstanceIDs(app,'mathworks.design.UnconnectedLinesPorts');
```

Alternatives

In the left-hand pane of the Model Advisor window, right-click the check and select **Send Check Instance ID to Workspace**.

Version History

Introduced in R2015b

See Also

`Advisor.Application` | `Advisor.Manager.createApplication` | `setAnalysisRoot` | `selectCheckInstances`

getEntry

Package: `ModelAdvisor`

Get cell contents from table in Model Advisor analysis results

Syntax

```
cellEntry = getEntry(tableObj,row,column)
```

Description

`cellEntry = getEntry(tableObj,row,column)` returns the cell content, `cellEntry`, for the cell specified by `row` and `column` of the Model Advisor table object `tableObj`.

Note The function `getEntry` is for returning Model Advisor analysis results with Simulink Check. For more information, see “Simulink Check”.

For information on how to get cell contents in MATLAB, see “Access Data in Tables”.

Examples

Get the Content of a Cell in a Model Advisor Results Table

Create a Model Advisor table object and get the content of a cell in the third row and the fourth column.

Use `ModelAdvisor.Table` to create a Model Advisor table object with four rows and four columns.

```
T1 = ModelAdvisor.Table(4,4);
```

Get the cell content from the cell in the third row and fourth column of the Model Advisor table object.

```
content = getEntry(T1,3,4);
```

Use `getEntry` in a check callback function in your `sl_customization` file to get table cell content from your Model Advisor results.

For more information, see “Define Custom Model Advisor Checks”.

Input Arguments

`tableObj` — Model Advisor table object

`ModelAdvisor.Table` object

Table of Model Advisor results, specified as a `ModelAdvisor.Table` object.

`row` — Table row
integer

Row of the table, specified by an integer.

`column` — Table column
integer

Column of the table, specified by an integer.

Output Arguments

`cellEntry` — Table cell content
element object | object array

Table cell content, specified by an element object or object array.

See Also

`ModelAdvisor.Table`

Topics
“Define Custom Model Advisor Checks”

getID

Return check identifier

Syntax

```
id = getID(check_obj)
```

Description

`id = getID(check_obj)` returns the ID of the check `check_obj`. `id` is a unique identifier for the check.

You create this unique identifier when you create the check. This unique identifier is the equivalent of the `ModelAdvisor.Check` `ID` property.

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Define Custom Model Advisor Checks”
“Create Model Advisor Checks”

execute

Class: `slmetric.Engine`
Package: `slmetric`

(To be removed) Collect metric data

Note `slmetric.Engine.execute` will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and "Collect Model Maintainability Metrics Programmatically".

Syntax

```
execute(metric_engine)
execute(slmetric_obj,MetricIDs)
```

Description

Collect model metric data for the specified metric engine object. The model metric data is based on defined architectural components. The components are these Simulink objects:

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

`execute(metric_engine)` collects metric data for available model metrics, which can include MathWorks metrics and custom metrics.

`execute(slmetric_obj,MetricIDs)` collects metric data for only the specified metrics, which can be MathWorks metrics or custom metrics.

Input Arguments

`metric_engine` — Metric engine object

`slmetric.Engine` object

Create a `slmetric.Engine` object.

```
metric_engine = slmetric.Engine();
```

`MetricIDs` — Metric identifier

character vector | cell array of character vectors

Metric identifier for "Model Metrics" on page 2-299 or custom model metrics that you create. You can specify one or multiple metric identifiers. You can get metric identifiers by calling `slmetric.metric.getAvailableMetrics`.

Example: `'mathworks.metrics.DescriptiveBlockNames'`

Examples

Collect and Access Metric Data for a Model

Collect and access model metric data for the model `sldemo_mdlref_basic`.

Open the model.

```
openExample('sldemo_mdlref_basic');
```

Create an `slmetric.Engine` object and set the root in the model for analysis.

```
metric_engine = slmetric.Engine();

% Include referenced models and libraries in the analysis.
% These properties are on by default.
metric_engine.ModelReferencesSimulationMode = 'AllModes';
metric_engine.AnalyzeLibraries = 1;

setAnalysisRoot(metric_engine, 'Root',  'sldemo_mdlref_basic');
```

Collect model metric data

```
execute(metric_engine);
```

Get the model metric data that returns an array of `slmetric.metric.ResultCollection` objects, `res_col`.

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

Display the results for the `mathworks.metrics.SimulinkBlockCount` metric.

```
for n=1:length(res_col)
    if res_col(n).Status == 0
        result = res_col(n).Results;

        for m=1:length(result)
            disp(['MetricID: ',result(m).MetricID]);
            disp(['  ComponentPath: ', result(m).ComponentPath]);
            disp(['  Value: ', num2str(result(m).Value)]);
            disp(['  AggregatedValue: ', num2str(result(m).AggregatedValue)]);
        end
    else
        disp(['No results for:', result(n).MetricID]);
    end
    disp(' ');
end
```

Collect and Access Metric Data for One Metric

Collect and access model metric data for the model `sldemo_mdlref_basic`.

Open the model.

```
openExample('sldemo_mdlref_basic');
```

Create an `slmetric.Engine` object. Include referenced models and libraries in the analysis and set the root in the model for analysis.

```
metric_engine = slmetric.Engine();
metric_engine.ModelReferencesSimulationMode = 'AllModes';
metric_engine.AnalyzeLibraries = 1;
setAnalysisRoot(metric_engine, 'Root',  'sldemo_mdlref_basic');
```

Collect model metric data

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

Get the model metric data that returns an array of `slmetric.metric.ResultCollection` objects, `res_col`.

```
res_col = getMetrics(metric_engine, 'mathworks.metrics.ExplicitIOCount');
```

Display the results for the `mathworks.metrics.ExplicitIOCount` metric.

```
for n=1:length(res_col)
    if res_col(n).Status == 0
        result = res_col(n).Results;

        for m=1:length(result)
            disp(['MetricID: ',result(m).MetricID]);
            disp(['  ComponentPath: ', result(m).ComponentPath]);
            disp(['  Value: ', num2str(result(m).Value)]);
            disp(['  AggregatedValue: ', num2str(result(m).AggregatedValue)]);
            disp(['  Measures: ', num2str(result(m).Measures)]);
            disp(['  AggregatedMeasures: ', num2str(result(m).AggregatedMeasures)]);
        end
    else
        disp(['No results for:', result(n).MetricID]);
    end
    disp(' ');
end
```

Here are the results:

```
MetricID: mathworks.metrics.ExplicitIOCount
  ComponentPath: sldemo_mdlref_basic
  Value: 3
  AggregatedValue: 4
  Measures: 0  3
  AggregatedMeasures: 3  3
MetricID: mathworks.metrics.ExplicitIOCount
  ComponentPath: sldemo_mdlref_basic/More Info
  Value: 0
  AggregatedValue: 0
  Measures: 0  0
  AggregatedMeasures: 0  0
MetricID: mathworks.metrics.ExplicitIOCount
  ComponentPath: sldemo_mdlref_counter
  Value: 4
  AggregatedValue: 4
  Measures: 3  1
  AggregatedMeasures: 3  1
```

For the `ComponentPath: sldemo_mdlref_basic`, the value is `3` because there are `3` outputs. The three outputs are in the second element of the `Measures` array. The `slmetric.metric.AggregationMode` is `Max`, so the `AggregatedValue` is `4` which is the number

of inputs and outputs to `sldemo_mdlref_counter`. The `AggregratedMeasures` array contains the maximum number of inputs and outputs for a component or subcomponent.

Version History

Introduced in R2016a

Warns

Warns starting in R2022a

The `slmetric.Engine` API will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and “Collect Model Maintainability Metrics Programmatically”.

See Also

`slmetric.metric.ResultCollection` | `slmetric.metric.getAvailableMetrics`

Topics

“Collect Model Metrics Programmatically”
“Model Metrics” on page 2-299

getAnalysisRootMetric

Class: `slmetric.Engine`
Package: `slmetric`

Get metric data for one metric for analysis root only

Syntax

```
metricResult = getAnalysisRootMetric(metric_engine,MetricID)
```

Description

Get metric data from the metric engine where the root of analysis was set using `setAnalysisRoot`.

`metricResult = getAnalysisRootMetric(metric_engine,MetricID)` get the metric data from `metric_engine`, for a specified metric identifier, `MetricID`, only for the analysis root.

Input Arguments

`metric_engine` — Collects and accesses metric data

`slmetric.Engine` object

When you call `execute`, `metric_engine` collects metric data for all available metrics or for the specified `MetricID`. Calling `getMetrics` accesses the collected metric data in `metric_engine`.

`MetricID` — Metric identifier

character vector

Metric identifier for “Model Metrics” on page 2-299 or custom model metrics, that you create. You can get metric identifiers by calling `slmetric.metric.getAvailableMetrics`.

Example: `'mathworks.metrics.DescriptiveBlockNames'`

Output Arguments

`metricResult` — Result of metric analysis on the analysis root

`slmetric.metric.Result` object

Outputs the object of the `slmetric.metric.Result` object containing the result data for the requested analysis root and metric.

Examples

Collect and Access Metric Data for the Analysis Root

This example shows how to set the analysis root, collect, and access the metric data for a metric.

```
% Create an slmetric.Engine object
metric_engine = slmetric.Engine();
```

```
% Specify the model for metric analysis
setAnalysisRoot(metric_engine, 'Root', 'sldemo_fuelsys');

% Collect model metrics for only the analysis root
metricID = 'mathworks.metrics.SimulinkBlockCount';
execute(metric_engine, metricID);

metricResult = getAnalysisRootMetric(metric_engine, metricID);
```

Version History

Introduced in R2017a

See Also

`slmetric.metric.ResultCollection` | `slmetric.metric.getAvailableMetrics`

Topics

“Collect Model Metrics Programmatically”
“Model Metrics” on page 2-299

getErrorLog

Class: `slmetric.Engine`
Package: `slmetric`

(To be removed) Get error log

Note `slmetric.Engine.getErrorLog` will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and "Collect Model Maintainability Metrics Programmatically".

Syntax

```
metricLog = getErrorLog(metric_engine)
```

Description

Get a log of errors and warnings that occurred during metric data collection of a specified metric engine object. The log includes errors that occurred during the execution of metric algorithms, model compilation, and metric data validation.

`metricLog = getErrorLog(metric_engine)`.

Input Arguments

`metric_engine` — Metric engine object

`slmetric.Engine` object

Constructed `slmetric.Engine` object.

Output Arguments

`metricLog` — Log of metric errors and warnings

string array

The `metricLog` string contains the errors and warnings from metric analysis and is formatted in HTML.

Examples

Get Error Log

This example shows how to create a `slmetric.Engine` object, set the analysis root, generate metrics, and create and display the error log for the model `sldemo_fuelsys`.

```
% Create an slmetric.Engine object
metric_engine = slmetric.Engine();
```

```
% Specify model for metric analysis
setAnalysisRoot(metric_engine, 'Root', 'sldemo_fuelsys');

% Collect model metrics for only the analysis root
metricID = 'mathworks.metrics.SimulinkBlockCount';
execute(metric_engine, metricID);

metricLog = getErrorLog(metric_engine);
disp(metricLog);
```

Version History

Introduced in R2017a

Warns

Warns starting in R2022a

The `slmetric.Engine` API will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and “Collect Model Maintainability Metrics Programmatically”.

See Also

`slmetric.metric.ResultCollection` | `slmetric.metric.getAvailableMetrics`

Topics

“Collect Model Metrics Programmatically”
“Model Metrics” on page 2-299

getMetricDistribution

Class: `slmetric.Engine`
Package: `slmetric`

Get metric distribution

Syntax

```
getMetricDistribution(metric_engine,MetricID)
```

Description

`getMetricDistribution(metric_engine,MetricID)` generates distribution for a specific metric, `MetricID`, for the metric data in the `slmetric.Engine` object, `metric_engine`. The distribution is on the metric data from the Value property of a `slmetric.metric.Result` object.

Input Arguments

`metric_engine` — Collects and accesses metric data

`slmetric.Engine` object

When you call `execute`, `metric_engine` collects metric data for all available metrics or for the specified `MetricID`. Calling `getMetrics` accesses the collected metric data in `metric_engine`.

`MetricID` — Metric identifier

character vector

Metric identifier for a model metric, specified as a character vector.

Example: `'mathworks.metrics.DescriptiveBlockNames'`

Output Arguments

`dist` — Distribution of the metric data

`slmetric.metric.MetricDistribution` object

Distribution of the metric data contains the following properties:

- `MetricID` is a `char` array that returns the metric ID specified in the `getMetricDistribution` function call.
- `BinCounts` is an `uint64` array of the number of components corresponding to a bin.
- `BinEdges` is a `double` array of equally spaced edges of each bin.

Examples

Generate Metric Distribution

To generate the distribution for a specific metric, create a `slmetric.Engine` object, set the analysis root for the `sldemo_fuelsys` model, and create a histogram of the data. The histogram shows the number of components corresponding to a number of blocks.

```
% Create an slmetric.Engine object
metric_engine = slmetric.Engine();

% Specify model for metric analysis
setAnalysisRoot(metric_engine, 'Root', 'sldemo_fuelsys');

% Collect model metrics and get distribution
metricID = 'mathworks.metrics.SimulinkBlockCount';
execute(metric_engine, metricID);
dist = getMetricDistribution(metric_engine, metricID);

% View the distribution using a histogram.
histogram('BinEdges',dist.BinEdges,'BinCounts',dist.BinCounts);
```

Version History

Introduced in R2017a

See Also

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

Topics

“Collect Model Metrics Programmatically”
“Model Metrics” on page 2-299

getMetrics

Class: `slmetric.Engine`
Package: `slmetric`

(To be removed) Access model metric data

Note `slmetric.Engine.getMetrics` will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and "Collect Model Maintainability Metrics Programmatically".

Syntax

```
Results = getMetrics(metric_engine)
Results = getMetrics(metric_engine,MetricIDs)
Results = getMetrics(metric_engine,MetricIDs,'AggregationDepth',ad)
```

Description

Access model metric data from the specified model metric engine. When you call `execute`, the metric engine collects the metric data. The returned metric data is based on defined architectural components. The components are these Simulink objects:

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

`Results = getMetrics(metric_engine)` returns metric data for all metrics that the metric engine executed.

`Results = getMetrics(metric_engine,MetricIDs)` returns metric data for the specified metric identifiers.

`Results = getMetrics(metric_engine,MetricIDs,'AggregationDepth',ad)` returns metric data for the specified metric identifiers and specifying how to aggregate data.

Input Arguments

`metric_engine` — Collects and accesses metric data

`slmetric.Engine` object

When you call `execute`, `metric_engine` collects metric data for all available MathWorks metrics or for the specified `MetricIDs`. Calling `getMetrics` accesses the collected metric data in `metric_engine`.

`MetricIDs` — Metric identifier
character vector | cell array of character vectors

Metric identifier for “Model Metrics” on page 2-299 or custom model metrics that you create. You can specify one or multiple metric identifiers. You can get metric identifiers by calling `slmetric.metric.getAvailableMetrics`.

Example: `'mathworks.metrics.DescriptiveBlockNames'`

`AggregationDepth` — Depth or level in the component hierarchy to which `getMetrics` aggregates the metric data
`All` (default) | `None`

Depth or level in the component for which `getMetrics` aggregates the metric data, specified as a name-value pair argument. Values are one of the following:

- `All` — `getMetrics` aggregates the detailed results to the component level. Then, the component level results are used to calculate the aggregated values by traversing the component hierarchy. `getMetrics` returns only the component-level results.
- `None` — Do not aggregate measures and values. If you specify this option, `getMetrics` returns metric values as collected by the metric algorithm. For example, if the metric algorithm returns detailed results, the detailed results are returned without aggregation. `AggregatedValue` and `AggregatedMeasures` properties of the returned `slmetric.metric.Result` objects are empty.

Example: `'AggregationDepth','None'`

Data Types: `char`

Output Arguments

`Results` — Metric data from the metric engine
array of `slmetric.metric.ResultCollection` objects

Metric data from the metric engine.

Examples

Collect and Access Metric Data for a Model

Collect and access model metric data for the model `sldemo_mdlref_basic`.

Open the model

```
openExample('sldemo_mdlref_basic');
```

Create an `slmetric.Engine` object and set the root in the model for analysis.

```
metric_engine = slmetric.Engine();

% Include referenced models and libraries in the analysis.
% These properties are on by default.
metric_engine.ModelReferencesSimulationMode = 'AllModes';
metric_engine.AnalyzeLibraries = 1;

setAnalysisRoot(metric_engine, 'Root',  'sldemo_mdlref_basic');
```

Collect model metric data

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

Get the model metric data that returns an array of `slmetric.metric.ResultCollection` objects, `res_col`.

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

Display the results for the `mathworks.metrics.SimulinkBlockCount` metric.

```
for n=1:length(res_col)
    if res_col(n).Status == 0
        result = res_col(n).Results;

        for m=1:length(result)
            disp(['MetricID: ',result(m).MetricID]);
            disp(['  ComponentPath: ', result(m).ComponentPath]);
            disp(['  Value: ', num2str(result(m).Value)]);
            disp(['  AggregatedValue: ', num2str(result(m).AggregatedValue)]);
        end
    else
        disp(['No results for:', result(n).MetricID]);
    end
    disp(' ');
end
```

Version History

Introduced in R2016a

Warns

Warns starting in R2022a

The `slmetric.Engine` API will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and “Collect Model Maintainability Metrics Programmatically”.

See Also

`slmetric.metric.Result` | `slmetric.metric.ResultCollection` | `slmetric.metric.getAvailableMetrics`

Topics

“Collect Model Metrics Programmatically”
“Model Metrics” on page 2-299

getResults

Class: `Advisor.Application`
Package: `Advisor`

Access Model Advisor analysis results

Syntax

```
Results = getResults(app)
Results = getResults(app,Name,Value)
```

Description

Access `Application` object analysis results.

`Results = getResults(app)` provides access to Model Advisor analysis results.

`Results = getResults(app,Name,Value)`

Examples

Get Results from Model Advisor Analysis

Create an `Application` object, run Model Advisor analysis, and get the results.

Create an `Application` object.

```
app = Advisor.Manager.createApplication();
```

Open the model and set the root analysis to `RootModel`.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to sldemo_mdlref_basic model
RootModel = 'sldemo_mdlref_basic';

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);
```

Run Model Advisor analysis.

```
run(app);
```

Get the analysis results.

```
getResults(app);
```

Input Arguments

app — Application

`Advisor.Application` object

`Advisor.Application` object, created by `Advisor.Manager.createApplication`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

IDs — Component IDs

cell array

Component IDs, as specified as a cell array of IDs

Data Types: `cell`

Output Arguments

Result — Analysis results

cell array of `ModelAdvisor.SystemResult` objects

Analysis results, returned as a cell array of `ModelAdvisor.SystemResult` objects.

Version History

Introduced in R2015b

See Also

`Advisor.Application` | `Advisor.Manager.createApplication` | `setAnalysisRoot` | `run` | `selectCheckInstances` | `deselectCheckInstances` | `ModelAdvisor.run`

getStatistics

Class: `slmetric.Engine`
Package: `slmetric`

Get statistics on metric data

Syntax

```
stats = getStatistics(metric_engine,MetricID)
```

Description

Generate statistics on the `Value` properties of the `slmetric.metric.Result` objects for the specified metric engine object, `metric_engine`.

`stats = getStatistics(metric_engine,MetricID)` generate statistics for the specified metric identifier.

Input Arguments

`metric_engine` — Collects and accesses metric data

`slmetric.Engine` object

When you call `execute`, `metric_engine` collects metric data for all available metrics or for the specified `MetricID`. Calling `getMetrics` accesses the collected metric data in `metric_engine`.

`MetricID` — Metric identifier

character vector

Metric identifier for “Model Metrics” on page 2-299 or custom model metrics that you create. You can get metric identifiers by calling `slmetric.metric.getAvailableMetrics`.

Example: `'mathworks.metrics.DescriptiveBlockNames'`

Output Arguments

`stats` — Metric statistics

`slmetric.metric.Statistics` object

The `Statistics` object contains the following properties:

- `MinValue` is a double that returns the minimum of the `Value` of the `slmetric.metric.Result` object.
- `MaxValue` is a double that returns the maximum of the `Value` of the `slmetric.metric.Result` object.
- `MeanValue` is a double that returns the mean of the `Value` of the `slmetric.metric.Result` object.
- `StandardDeviation` is a double that returns the standard deviation of the `Value` of the `slmetric.metric.Result` object.

Examples

Collect Statistics

This example shows how to create a `slmetric.Engine` object, set the analysis root, collect the block count metric, and collect statistics for the model `sldemo_fuelsys`.

```
% Create an slmetric.Engine object
metric_engine = slmetric.Engine();

% Specify model for metric analysis
setAnalysisRoot(metric_engine, 'Root','sldemo_fuelsys');

% Generate and collect model metrics
metricID = 'mathworks.metrics.SimulinkBlockCount';
execute(metric_engine, metricID);
stats = getStatistics(metric_engine, metricID);
```

Version History

Introduced in R2017a

See Also

`slmetric.metric.ResultCollection` | `slmetric.metric.getAvailableMetrics`

Topics

“Collect Model Metrics Programmatically”

“Model Metrics” on page 2-299

loadConfiguration

Class: `Advisor.Application`
Package: `Advisor`

Load Model Advisor configuration

Syntax

```
loadConfiguration(app,filename)
```

Description

`loadConfiguration(app,filename)` loads the Model Advisor configuration from `filename`.

Input Arguments

app — Application

`Advisor.Application` object

`Advisor.Application` object, created by `Advisor.Manager.createApplication`

`filename` — Name of Model Advisor configuration file

character vector

Name of Model Advisor configuration file, specified as a character vector.

The Model Advisor configuration file can be in the format of a `.json` file or a `.mat` file.

Example: `'MyConfiguration.json'`

Example: `'MyConfiguration.mat'`

Data Types: `char`

Version History

Introduced in R2015b

See Also

`Advisor.Manager.createApplication` | `setAnalysisRoot`

mdltransformer

Open Model Transformer

Syntax

```
mdltransformer(model)
```

Description

`mdltransformer(model)` opens the Model Transformer for a model specified by `model`. If the specified model is not open, this command opens it.

Examples

Open Model Transformer for model

Open the Model Transformer for `fuel_rate_control` example model:

```
mdltransformer('fuel_rate_control')
```

Input Arguments

`model` — Model name

character vector

Model name or handle, specified as a character vector.

Data Types: `char`

Version History

Introduced in R2016b

See Also

Topics

“Transform Model to Variant System”
“Improve Model Readability by Eliminating Local Data Store Blocks”
“Improve Efficiency of Simulation by Optimizing Prelookup Operation of Lookup Table Blocks”
“Improve Code Efficiency by Merging Multiple Interpolation Using Prelookup Blocks”

metricsdashboard

(To be removed) Open Metrics Dashboard

Note The `metricsdashboard` function and the Metrics Dashboard will be removed in a future release. For the size, architecture, and complexity metrics, use the **Model Maintainability Dashboard** instead. For more information, see `modelDesignDashboard` and "Monitor the Complexity of Your Design Using the Model Maintainability Dashboard".

Syntax

```
metricsdashboard(system)
```

Description

`metricsdashboard(system)` opens the Metrics Dashboard for a system specified by `system`. The *`system`* can be either a model name or a block path to a subsystem. The system cannot be a Configurable Subsystem block.

Examples

Open Metrics Dashboard for system

Open the Metrics Dashboard for `vdp` example model:

```
metricsdashboard('vdp')
```

Input Arguments

`system` — System name

character vector

System name, specified as a character vector.

Data Types: `char`

Version History

Introduced in R2017b

Warns

Warns starting in R2022a

The `metricsdashboard` function issues a warning that it and the Metrics Dashboard will be removed in a future release. For the size, architecture, and complexity metrics, use the **Model Maintainability Dashboard** instead. For more information, see `modelDesignDashboard` and "Monitor the Complexity of Your Design Using the Model Maintainability Dashboard".

See Also

Topics
“Collect and Explore Metric Data by Using the Metrics Dashboard”

slmetric.metric.Metric class

Package: `slmetric.metric`

Abstract class for creating model metrics

Description

Abstract base class for creating model metrics. To create a model metric, create a MATLAB class that derives from the `slmetric.metric.Metric` class.

Properties

`CompileContext` — Compile mode

character vector

Compile mode for metric calculation. If your model metric requires model compilation, specify `PostCompile`. If your model metric does not require model compilation, specify `None`.

Example: `'PostCompile'`

Data Types: `char`

`ComponentScope` — Component scope

array of `Advisor.component.Types` enum values

Model components for which metric is calculated. The metric is calculated for all components that match the type.

`Description` — Metric description

character vector

Metric description.

Data Types: `char`

`ID` — Metric ID

character vector

Unique metric identifier.

Data Types: `char`

`Version` — Metric version number

integer

Use this property to communicate changes in your metric algorithm to the metric engine.

Data Types: `uint32`

`Name` — Name of the metric algorithm

character vector

Specify a name for the custom metric algorithm.

Data Types: `char`

`ValueName` — Name of the collected value
character vector

Specify a name for the value that the metric collects. When you view detailed results in the Metrics Dashboard, the `ValueName` is the column header for the collected value in the table and tree views.

Data Types: `char`

`AggregatedValueName` — Name of the aggregated value
character vector

Specify a name for the aggregated value that the metric collects. To specify this property, set `AggregationMode` to `None`. When you view detailed results in the Metrics Dashboard, the `AggregatedValueName` is the column header for the collected aggregated value in the table and tree views.

Data Types: `char`

`MeasuresNames` — Names of the collected measures
cell array of character vectors

Specify the names of the measures collected by the metric. Set this property only if the metric algorithm writes measures to the result object that the algorithm returns. When you view detailed results in the Metrics Dashboard, the `MeasuresNames` are column headers for the collected measures in the table and tree views.

Data Types: `char`

`AggregatedMeasuresNames` — Names of the aggregated measures
cell array of character vectors

Specify the names of the aggregated measures collected by the metric. Set this property only if the metric algorithm writes measures to the result object that the algorithm returns and `AggregationMode` is set to `None`. When you view detailed results in the Metrics Dashboard, the `AggregatedMeasuresNames` are column headers for the collected aggregated measures in the table and tree views.

Data Types: `char`

`ResultChecksumCoverage` — Reuse metric data
logical

If `true`, results produced by the metric algorithm change only if the model or library source files change. If the source file and the metric `Version` have not changed, metric data is not regenerated. If `false`, each call to `slmetric.Engine.execute` collects new data for this metric and stores it in the metric repository.

Data Types: `logical`

`AggregationMode` — How the metric algorithm aggregates the metric data
character array

Specify the operation to aggregate the `slmetric.metric.Result` object properties `Value` and `Measure` across the component hierarchy. The metric algorithm outputs the aggregated values in the

`slmetric.metric.Result` object properties `AggregatedValues` and `AggregatedMeasures`. Options are:

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

Data Types: `char`

`SupportsResultDetails` — Specify whether `Details` property contains data
logical

Specify whether the `slmetric.metric.Result` object property `Details` contains data. The default value is `false`. Metrics Dashboard filters are enabled when you set the value of this property to `false`.

Data Types: `logical`

Methods

algorithm	Specify logic for metric data analysis

Version History

Introduced in R2016a

See Also

`slmetric.Engine` | `slmetric.metric.Result` | `slmetric.metric.createNewMetricClass` | `slmetric.metric.getAvailableMetrics`

Topics

“Create a Custom Model Metric for Nonvirtual Block Count”
“Model Metrics” on page 2-299

algorithm

Class: `slmetric.metric.Metric`
Package: `slmetric.metric`

Specify logic for metric data analysis

Syntax

```
Result = algorithm(Metric,Component)
```

Description

Specify logic for metric algorithm analysis. Custom-authored metric algorithms are not called for library links and external MATLAB file components.

`Result = algorithm(Metric,Component)` specifies logic for metric algorithm analysis.

Input Arguments

`Metric` — New model metric class

`slmetric.metric.Metric` object

Model metric class you are defining for a new metric.

`Component` — Component for metric analysis

`Advisor.component.Component` object

Instance of `Advisor.component.Component` for metric analysis.

Output Arguments

`Result` — Algorithm result data

array of `slmetric.metric.Result` objects

Algorithm data, returned as an array of `slmetric.metric.Result` objects.

Examples

Create Metric Algorithm for Nonvirtual Block Count

This example shows how to use the `algorithm` method to create a nonvirtual block count metric.

Using the `createNewMetricClass` function, create a metric class with the name `nonvirtualblockcount`. The function creates the `nonvirtualblockcount.m` file in the current working folder.

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

Open and edit the metric algorithm file `nonvirtualblockcount.m`. The file contains an empty metric algorithm method.

```
edit(className);
```

Copy and paste the following code into the `nonvirtualblockcount.m` file. Save `nonvirtualblockcount.m`. The code provides a metric algorithm for counting the nonvirtual blocks.

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

    methods
    function this = nonvirtualblockcount()
        this.ID = 'nonvirtualblockcount';
        this.Version = 1;
        this.CompileContext = 'None';
        this.Description = 'Algorithm that counts nonvirtual blocks per level.';
        this.ComponentScope = [Advisor.component.Types.Model, ...
            Advisor.component.Types.SubSystem];
    end

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

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

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

        isNonVirtual = true(size(blocks));

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

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

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

                        if nod == 1 && any(strcmp(ios_settings, ios))
                            isNonVirtual(n) = false;
```

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

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

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

    blocks = blocks(isNonVirtual);

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

Version History

Introduced in R2016a

See Also

`slmetric.metric.createNewMetricClass` | `slmetric.metric.Result`

Topics

“Create a Custom Model Metric for Nonvirtual Block Count”

“Model Metrics” on page 2-299

slmetric.metric.ResultDetail class

Package: `slmetric.metric`

(To be removed) Details about instances of `slmetric.metric.Result` objects

Note `slmetric.metric.ResultDetail` will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and "Collect Model Maintainability Metrics Programmatically".

Description

Details about what the metric engine counts for the `slmetric.metric.Result` object property `Value`.

Construction

Calling the `slmetric.Engine.execute` method creates the `slmetric.metric.Result` objects, which optionally includes the `slmetric.metric.ResultDetail` objects. `Details1 = slmetric.metric.ResultDetail = (ID, Name)` creates an `slmetric.metric.ResultDetail` object. You must supply the ID and Name as inputs to the constructor.

Properties

`ID` — Unique identifier

character vector

Unique identifier for the entity that the result detail instance counts. This property is read/write.

Data Types: `char`

`Name` — Name of model entity

character vector

Name of model entity that result detail instance counts. This property is read/write.

Data Types: `char`

`Value` — Value of `ID` property

double

Scalar value generated by metric algorithm for `ID`. This property is read/write.

Data Types: `double`

Methods

setGroup	Set the name and identifier for a group of `slmetric.metric.ResultDetail` objects
getGroupIdentifier	Obtain the identifier for a group of `slmetric.metric.ResultDetail` objects
getGroupName	Obtain the name for a group of `slmetric.metric.ResultDetail` objects

Examples

Obtain Clone Group Names and Identifiers

Use the `getGroupName` and `getGroupIdentfier` methods to obtain the name and identifier for a group of clones.

Open the example model.

```
open_system([docroot '\toolbox\simulink\examples\ex_clone_detection.slx']);
```

Save the example model to your current working folder.

Call the `execute` method. Apply the `getMetrics` method for the`mathworks.metric.CloneDetection` metric.

```
metric_engine = slmetric.Engine();
setAnalysisRoot(metric_engine,'Root','ex_clone_detection','RootType','Model');
execute(metric_engine);
rc = getMetrics(metric_engine,'mathworks.metrics.CloneDetection');
```

For each `slmetric.metric.Result` object, display the `ComponentPath`. For each `slmetric.metric.ResultDetail` object, display the clone group name and identifier.

```
for n=1:length(rc.Results)
    if rc.Results(n).Value > 0
    for m=1:length(rc.Results(n).Details)
      disp(['ComponentPath: ',rc.Results(n).ComponentPath]);
          disp(['Group Name: ',rc.Results(n).Details(m).getGroupName]);
          disp(['Group Identifier: ',rc.Results(n).Details(m).getGroupIdentifier]);
        end
    else
        disp(['No results for ComponentPath: ',rc.Results(n).ComponentPath]);
    end
    disp(' ');
end
```

The results show that the model contains one clone group, `CloneGroup1`, which contains two clones.

Set Group Names and Group Identifiers for a Custom Model Metric

Use the `setGroup` method to group detailed results. When you create a custom model metric, you apply this method as part of the `algorithm` method.

Using the `createNewMetricClass` function, create a metric class named `DataStoreCount`. This metric counts the number of Data Store Read and Data Store Write blocks and groups them together by the corresponding Data Store Memory block. The `createNewMetricClass` function creates a file, `DataStoreCount.m` in the current working folder. The file contains a constructor and empty metric algorithm method. For this example, make sure that you are working in a writable folder.

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

To write the metric algorithm, open the `DataStoreCount.m` file and add the metric to the file. For this example, you can create the metric algorithm by copying this logic into the `DataStoreCount.m` file.

```
classdef DataStoreCount < slmetric.metric.Metric
    % Count the number of Data Store Read and Data Store Write
    % blocks and correlate them across components.

    methods
        function this = DataStoreCount()
            this.ID = 'DataStoreCount';
            this.ComponentScope = [Advisor.component.Types.Model, ...
                Advisor.component.Types.SubSystem];
            this.AggregationMode = slmetric.AggregationMode.Sum;
            this.CompileContext = 'None';
            this.Version = 1;
            this.SupportsResultDetails = true;

            %Textual information on the metric algorithm
            this.Name = 'Data store usage';
            this.Description = 'Metric that counts the number of Data Store Read and Write';
                  'blocks and groups them by the corresponding Data Store Memory block.';

        end

        function res = algorithm(this, component)
            % Use find_system to get all blocks inside this component.
            dswBlocks = find_system(getPath(component), ...
                'SearchDepth', 1, ...
                'BlockType', 'DataStoreWrite');
            dsrBlocks = find_system(getPath(component), ...
                'SearchDepth', 1, ...
                'BlockType', 'DataStoreRead');

            % Create a ResultDetail object for each data store read and write block.
            % Group ResultDetails by the data store name.
            details1 = slmetric.metric.ResultDetail.empty();
            for i=1:length(dswBlocks)
                details1(i) = slmetric.metric.ResultDetail(getfullname(dswBlocks{i}),...
                              get_param(dswBlocks{i}, 'Name'));
           groupID = get_param(dswBlocks{i},'DataStoreName');
           groupName = get_param(dswBlocks{i},'DataStoreName');
                details1(i).setGroup(groupID, groupName);
                details1(i).Value = 1;
            end

            details2 = slmetric.metric.ResultDetail.empty();
            for i=1:length(dsrBlocks)
                details2(i) = slmetric.metric.ResultDetail(getfullname(dsrBlocks{i}),...
                   get_param(dsrBlocks{i}, 'Name'));
                groupID = get_param(dsrBlocks{i},'DataStoreName');
                groupName = get_param(dsrBlocks{i},'DataStoreName');
                details2(i).setGroup(groupID, groupName);
                details2(i).Value = 1;
            end

            res = slmetric.metric.Result();
            res.ComponentID = component.ID;
            res.MetricID = this.ID;
            res.Value = length(dswBlocks)+ length(dsrBlocks);
            res.Details = [details1 details2];
        end
    end
end
```

In the `DataStoreCount` metric class, the `SupportsResultDetail` method is set to true. The metric algorithm contains the logic for the `setGroup` method.

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

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

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

Open the `sldemo_mdlref_dsm` model.

```
openExample('sldemo_mdlref_dsm');
```

Create a metric engine object and set the analysis root.

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

Collect metric data for the Data Store count metric.

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

For each `slmetric.metric.Result` object, display the `ComponentPath`. For each `slmetric.metric.ResultDetails` object, display the Data Store group name and identifier.

```
for n=1:length(rc.Results)
    if rc.Results(n).Value > 0
    for m=1:length(rc.Results(n).Details)
      disp(['ComponentPath: ',rc.Results(n).ComponentPath]);
          disp(['Group Name: ',rc.Results(n).Details(m).getGroupName]);
          disp(['Group Identifier: ',rc.Results(n).Details(m).getGroupIdentifier]);
        end
    else
        disp(['No results for ComponentPath: ',rc.Results(n).ComponentPath]);
    end
    disp(' ');
end
```

Here are the results.

```
ComponentPath: sldemo_mdlref_dsm
Group Name: ErrorCond
Group Identifier: ErrorCond

No results for ComponentPath: sldemo_mdlref_dsm/A

No results for ComponentPath: sldemo_mdlref_dsm/A1

No results for ComponentPath: sldemo_mdlref_dsm/More Info1

ComponentPath: sldemo_mdlref_dsm_bot
Group Name: RefSignalVal
Group Identifier: RefSignalVal

ComponentPath: sldemo_mdlref_dsm_bot2
Group Name: ErrorCond
Group Identifier: ErrorCond

ComponentPath: sldemo_mdlref_dsm_bot/PositiveSS
Group Name: RefSignalVal
Group Identifier: RefSignalVal

ComponentPath: sldemo_mdlref_dsm_bot/NegativeSS
Group Name: RefSignalVal
Group Identifier: RefSignalVal
```

For this example, unregister the data store count metric.

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

Close the model.

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

Version History

Introduced in R2017b

Warns

Warns starting in R2022a

The `slmetric.Engine` API will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and “Collect Model Maintainability Metrics Programmatically”.

See Also

`slmetric.metric.Result` | `slmetric.metric.ResultCollection` | `slmetric.metric.ResultDetail` | `slmetric.metric.getAvailableMetrics`

setGroup

Class: `slmetric.metric.ResultDetail`
Package: `slmetric.metric`

Set the name and identifier for a group of `slmetric.metric.ResultDetail` objects

Syntax

```
setGroup(groupIdentifier,groupName)
```

Description

For a custom-authored metric, set the identifier and name for a group of `slmetric.metric.ResultDetail` objects. Apply this method from within the part of the metric algorithm that specifies the details for `getMetrics` objects.

`setGroup(groupIdentifier,groupName)` sets the values of the group name and identifier for an `slmetric.metric.ResultDetail` object.

Input Arguments

`groupIdentifier` — Group identifier

character vector

Specify a value for the identifier for a group of `slmetric.metric.ResultDetail` objects.

`groupName` — Group name

character vector

Specify a value for the name of a group of `slmetric.metric.ResultDetail` objects.

Examples

Set Group Names and Group Identifiers for a Custom Model Metric

Use the `setGroup` method to group detailed results. When you create a custom model metric, you apply this method as part of the `algorithm` method.

Using the `createNewMetricClass` function, create a metric class named `DataStoreCount`. This metric counts the number of Data Store Read and Data Store Write blocks and groups them together by the corresponding Data Store Memory block. The `createNewMetricClass` function creates a file `DataStoreCount.m` in the current working folder. The file contains a constructor and empty metric algorithm method. For this example, make sure that you are working in a writable folder.

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

To write the metric algorithm, open the `DataStoreCount.m` file and add the metric to the file. For this example, you can create the metric algorithm by copying this logic into the `DataStoreCount.m` file.

```
classdef DataStoreCount < slmetric.metric.Metric
    % Count the number of Data Store Read and Data Store Write
    % blocks and correlate them across components.

    methods
        function this = DataStoreCount()
            this.ID = 'DataStoreCount';
            this.ComponentScope = [Advisor.component.Types.Model, ...
                Advisor.component.Types.SubSystem];
            this.AggregationMode = slmetric.AggregationMode.Sum;
            this.CompileContext = 'None';
            this.Version = 1;
            this.SupportsResultDetails = true;

            %Textual information on the metric algorithm
            this.Name = 'Data store usage';
            this.Description = 'Metric that counts the number of Data Store Read and Write';
                  'blocks and groups them by the corresponding Data Store Memory block.';

        end

        function res = algorithm(this, component)
            % Use find_system to get all blocks inside this component.
            dswBlocks = find_system(getPath(component), ...
                'SearchDepth', 1, ...
                'BlockType', 'DataStoreWrite');
            dsrBlocks = find_system(getPath(component), ...
                'SearchDepth', 1, ...
                'BlockType', 'DataStoreRead');

            % Create a ResultDetail object for each data store read and write block.
            % Group ResultDetails by the data store name.
            details1 = slmetric.metric.ResultDetail.empty();
            for i=1:length(dswBlocks)
                details1(i) = slmetric.metric.ResultDetail(getfullname(dswBlocks{i}),...
                            get_param(dswBlocks{i}, 'Name'));
           groupID = get_param(dswBlocks{i},'DataStoreName');
           groupName = get_param(dswBlocks{i},'DataStoreName');
                details1(i).setGroup(groupID, groupName);
                details1(i).Value = 1;
            end

            details2 = slmetric.metric.ResultDetail.empty();
            for i=1:length(dsrBlocks)
                details2(i) = slmetric.metric.ResultDetail(getfullname(dsrBlocks{i}),...
                   get_param(dsrBlocks{i}, 'Name'));
                groupID = get_param(dsrBlocks{i},'DataStoreName');
                groupName = get_param(dsrBlocks{i},'DataStoreName');
                details2(i).setGroup(groupID, groupName);
                details2(i).Value = 1;
            end

            res = slmetric.metric.Result();
            res.ComponentID = component.ID;
            res.MetricID = this.ID;
            res.Value = length(dswBlocks)+ length(dsrBlocks);
            res.Details = [details1 details2];
        end
    end
end
```

In the `DataStoreCount` metric class, the `SupportsResultDetail` method is set to true. The metric algorithm contains the logic for the `setGroup` method.

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

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

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

Open the `sldemo_mdlref_dsm` model.

```
openExample('sldemo_mdlref_dsm');
```

Create a metric engine object and set the analysis root.

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

Collect metric data for the Data Store count metric.

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

For each `slmetric.metric.Result` object, display the `ComponentPath`. For each `slmetric.metric.ResultDetails` object, display the Data Store group name and identifier.

```
for n=1:length(rc.Results)
    if rc.Results(n).Value > 0
    for m=1:length(rc.Results(n).Details)
      disp(['ComponentPath: ',rc.Results(n).ComponentPath]);
          disp(['Group Name: ',rc.Results(n).Details(m).getGroupName]);
          disp(['Group Identifier: ',rc.Results(n).Details(m).getGroupIdentifier]);
        end
    else
        disp(['No results for ComponentPath: ',rc.Results(n).ComponentPath]);
    end
    disp(' ');
end
```

Here are the results.

```
ComponentPath: sldemo_mdlref_dsm
Group Name: ErrorCond
Group Identifier: ErrorCond

No results for ComponentPath: sldemo_mdlref_dsm/More Info1

ComponentPath: sldemo_mdlref_dsm_bot
Group Name: RefSignalVal
Group Identifier: RefSignalVal

ComponentPath: sldemo_mdlref_dsm_bot2
Group Name: ErrorCond
Group Identifier: ErrorCond

ComponentPath: sldemo_mdlref_dsm_bot/PositiveSS
Group Name: RefSignalVal
Group Identifier: RefSignalVal

ComponentPath: sldemo_mdlref_dsm_bot/NegativeSS
Group Name: RefSignalVal
Group Identifier: RefSignalVal
```

For this example, unregister the data store count metric.

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

Close the model.

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

Version History

Introduced in R2017b

See Also

`slmetric.metric.Result` | `slmetric.metric.ResultCollection` | `slmetric.metric.ResultDetail` | `slmetric.metric.getAvailableMetrics`

getGroupIdentifier

Class: `slmetric.metric.ResultDetail`
Package: `slmetric.metric`

Obtain the identifier for a group of `slmetric.metric.ResultDetail` objects

Syntax

```
groupIdentifier = getGroupIdentifier(mrd)
```

Description

Obtain the identifier for a group of `slmetric.metric.ResultDetail` objects. Calling the `execute` method collects metric data. Calling `getMetrics` accesses the `slmetric.metric.Result` objects, which include the `slmetric.metric.ResultDetail` objects. Apply the `getGroupIdentifier` method to the `slmetric.metric.ResultDetail` object.

`groupIdentifier = getGroupIdentifier(mrd)` obtains the group identifier for the `slmetric.metric.ResultDetail` object `mrd`.

Input Arguments

`mrd` — `slmetric.metric.ResultDetail` object

character vector

Calling the `slmetric.Engine.execute` method creates the `slmetric.metric.Result` objects, which include the `slmetric.metric.ResultDetail` objects.

Output Arguments

`groupIdentifier` — Group identifier

character vector

Identifier for a group of `slmetric.metric.ResultDetail` objects.

Examples

Obtain Clone Group Names and Identifiers

Use the `getGroupName` and `getGroupIdentfier` methods to obtain the name and identifier for a group of clones.

Open the example model `ex_clone_detection.slx` and save the model to your current working folder.

```
openExample('slcheck/EnableSubsystemReuseWithCloneExample','supportingfile','ex_clone_detection');
```

Call the `execute` method. Apply the `getMetrics` method for the `mathworks.metric.CloneDetection` metric.

```
metric_engine = slmetric.Engine();
setAnalysisRoot(metric_engine,'Root','ex_clone_detection','RootType','Model');
execute(metric_engine);
rc = getMetrics(metric_engine,'mathworks.metrics.CloneDetection');
```

For each `slmetric.metric.Result` object, display the `ComponentPath`. For each `slmetric.metric.ResultDetail` object, display the clone group name and identifier.

```
for n=1:length(rc.Results)
    if rc.Results(n).Value > 0
    for m=1:length(rc.Results(n).Details)
      disp(['ComponentPath: ',rc.Results(n).ComponentPath]);
          disp(['Group Name: ',rc.Results(n).Details(m).getGroupName]);
          disp(['Group Identifier: ',rc.Results(n).Details(m).getGroupIdentifier]);
        end
    else
        disp(['No results for ComponentPath: ',rc.Results(n).ComponentPath]);
    end
    disp(' ');
end
```

The results show that the model contains one clone group, `CloneGroup1`, which contains two clones.

Set Group Names and Group Identifiers for a Custom Model Metric

Use the `setGroup` method to group detailed results. When you create a custom model metric, you apply this method as part of the `algorithm` method.

Using the `createNewMetricClass` function, create a new metric class named `DataStoreCount`. This metric counts the number of Data Store Read and Data Store Write blocks and groups them together by the corresponding Data Store Memory block. The `createNewMetricClass` function creates a file, `DataStoreCount.m` in the current working folder. The file contains a constructor and empty metric algorithm method. For this example, make sure that you are working in a writable folder.

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

To write the metric algorithm, open the `DataStoreCount.m` file and add the metric to the file. For this example, you can create the metric algorithm by copying this logic into the `DataStoreCount.m` file.

```
classdef DataStoreCount < slmetric.metric.Metric
    % Count the number of Data Store Read and Data Store Write
    % blocks and correlate them across components.

    methods
        function this = DataStoreCount()
            this.ID = 'DataStoreCount';
            this.ComponentScope = [Advisor.component.Types.Model, ...
                Advisor.component.Types.SubSystem];
            this.AggregationMode = slmetric.AggregationMode.Sum;
            this.CompileContext = 'None';
            this.Version = 1;
            this.SupportsResultDetails = true;

            %Textual information on the metric algorithm
            this.Name = 'Data store usage';
            this.Description = 'Metric that counts the number of Data Store Read and Write';
                  'blocks and groups them by the corresponding Data Store Memory block.';

        end

        function res = algorithm(this, component)
            % Use find_system to get all blocks inside this component.
            dswBlocks = find_system(getPath(component), ...
                'SearchDepth', 1, ...
                'BlockType', 'DataStoreWrite');
            dsrBlocks = find_system(getPath(component), ...
                'SearchDepth', 1, ...
                'BlockType', 'DataStoreRead');
```

```
            % Create a ResultDetail object for each data store read and write block.
            % Group ResultDetails by the data store name.
            details1 = slmetric.metric.ResultDetail.empty();
            for i=1:length(dswBlocks)
                details1(i) = slmetric.metric.ResultDetail(getfullname(dswBlocks{i}),...
                              get_param(dswBlocks{i}, 'Name'));
           groupID = get_param(dswBlocks{i},'DataStoreName');
           groupName = get_param(dswBlocks{i},'DataStoreName');
                details1(i).setGroup(groupID, groupName);
                details1(i).Value = 1;
            end

            details2 = slmetric.metric.ResultDetail.empty();
            for i=1:length(dsrBlocks)
                details2(i) = slmetric.metric.ResultDetail(getfullname(dsrBlocks{i}),...
                   get_param(dsrBlocks{i}, 'Name'));
                groupID = get_param(dsrBlocks{i},'DataStoreName');
                groupName = get_param(dsrBlocks{i},'DataStoreName');
                details2(i).setGroup(groupID, groupName);
                details2(i).Value = 1;
            end

            res = slmetric.metric.Result();
            res.ComponentID = component.ID;
            res.MetricID = this.ID;
            res.Value = length(dswBlocks)+ length(dsrBlocks);
            res.Details = [details1 details2];
        end
    end
end
```

In the `DataStoreCount` metric class, the `SupportsResultDetail` method is set to true. The metric algorithm contains the logic for the `setGroup` method.

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

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

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

Load the `sldemo_mdlref_dsm` model.

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

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

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

Collect metric data for the Data Store count metric.

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

For each `slmetric.metric.Result` object, display the `ComponentPath`. For each `slmetric.metric.ResultDetails` object, display the Data Store group name and identifier.

```
for n=1:length(rc.Results)
    if rc.Results(n).Value > 0
    for m=1:length(rc.Results(n).Details)
      disp(['ComponentPath: ',rc.Results(n).ComponentPath]);
          disp(['Group Name: ',rc.Results(n).Details(m).getGroupName]);
          disp(['Group Identifier: ',rc.Results(n).Details(m).getGroupIdentifier]);
        end
    else
        disp(['No results for ComponentPath: ',rc.Results(n).ComponentPath]);
    end
```

```
    disp(' ');
end
```

Here are the results.

```
ComponentPath: sldemo_mdlref_dsm
Group Name: ErrorCond
Group Identifier: ErrorCond

No results for ComponentPath: sldemo_mdlref_dsm/More Info1

ComponentPath: sldemo_mdlref_dsm_bot
Group Name: RefSignalVal
Group Identifier: RefSignalVal

ComponentPath: sldemo_mdlref_dsm_bot2
Group Name: ErrorCond
Group Identifier: ErrorCond

ComponentPath: sldemo_mdlref_dsm_bot/PositiveSS
Group Name: RefSignalVal
Group Identifier: RefSignalVal

ComponentPath: sldemo_mdlref_dsm_bot/NegativeSS
Group Name: RefSignalVal
Group Identifier: RefSignalVal
```

For this example, unregister the data store count metric.

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

Close the model.

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

Version History

Introduced in R2017b

See Also

`slmetric.metric.Result` | `slmetric.metric.ResultCollection` | `slmetric.metric.ResultDetail` | `slmetric.metric.getAvailableMetrics`

getGroupName

Class: `slmetric.metric.ResultDetail`
Package: `slmetric.metric`

Obtain the name for a group of `slmetric.metric.ResultDetail` objects

Syntax

```
groupName = getGroupName(mrd)
```

Description

Obtain the name of a group of `slmetric.metric.ResultDetail` objects. Calling the `execute` method collects metric data. Calling `getMetrics` accesses the `slmetric.metric.Result` objects which include the `slmetric.metric.ResultDetail` objects. Apply the `getGroupName` method to the `slmetric.metric.ResultDetail` object.

`groupName = getGroupName(mrd)` obtains the name for the `slmetric.metric.ResultDetail` object `mrd`.

Input Arguments

`mrd` — `slmetric.metric.ResultDetail` object

character vector

Calling the `slmetric.Engine.execute` method creates the `slmetric.metric.Result` objects, which include the `slmetric.metric.ResultDetail` objects.

Output Arguments

`groupName` — Group name

character vector

Name for a group of `slmetric.metric.ResultDetail` objects

Examples

Obtain Clone Group Names and Identifiers

Use the `getGroupName` and `getGroupIdentifier` methods to obtain the name and identifier for a group of clones.

Open the example model `ex_clone_detection.slx` and save the model to your current working folder.

```
openExample('slcheck/EnableSubsystemReuseWithCloneExample','supportingfile','ex_clone_detection');
```

Call the `execute` method. Apply the `getMetrics` method for the `mathworks.metric.CloneDetection` metrics.

```
metric_engine = slmetric.Engine();
setAnalysisRoot(metric_engine,'Root','ex_clone_detection','RootType','Model');
execute(metric_engine);
rc = getMetrics(metric_engine,'mathworks.metrics.CloneDetection');
```

For each `slmetric.metric.Result` object, display the `ComponentPath`. For each `slmetric.metric.ResultDetail` object, display the clone group name and identifier.

```
for n=1:length(rc.Results)
    if rc.Results(n).Value > 0
    for m=1:length(rc.Results(n).Details)
      disp(['ComponentPath: ',rc.Results(n).ComponentPath]);
          disp(['Group Name: ',rc.Results(n).Details(m).getGroupName]);
          disp(['Group Identifier: ',rc.Results(n).Details(m).getGroupIdentifier]);
        end
    else
        disp(['No results for ComponentPath: ',rc.Results(n).ComponentPath]);
    end
    disp(' ');
end
```

The results show that the model contains one clone group, `CloneGroup1`, which contains two clones.

Set Group Names and Group Identifiers for a Custom Model Metric

Use the `setGroup` method to group detailed results. When you create a custom model metric, you apply this method as part of the `algorithm` method.

Using the `createNewMetricClass` function, create a metric class named `DataStoreCount`. This metric counts the number of Data Store Read and Data Store Write blocks and groups them together by the corresponding Data Store Memory block. The `createNewMetricClass` function creates a file, `DataStoreCount.m`, in the current working folder. The file contains a constructor and empty metric algorithm method. For this example, make sure that you are working in a writable folder.

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

To write the metric algorithm, open the `DataStoreCount.m` file and add the metric to the file. For this example, you can create the metric algorithm by copying this logic into the `DataStoreCount.m` file.

```
classdef DataStoreCount < slmetric.metric.Metric
    % Count the number of Data Store Read and Data Store Write
    % blocks and correlate them across components.

    methods
        function this = DataStoreCount()
            this.ID = 'DataStoreCount';
            this.ComponentScope = [Advisor.component.Types.Model, ...
                Advisor.component.Types.SubSystem];
            this.AggregationMode = slmetric.AggregationMode.Sum;
            this.CompileContext = 'None';
            this.Version = 1;
            this.SupportsResultDetails = true;

            %Textual information on the metric algorithm
            this.Name = 'Data store usage';
            this.Description = 'Metric that counts the number of Data Store Read and Write';
                  'blocks and groups them by the corresponding Data Store Memory block.';

        end

        function res = algorithm(this, component)
            % Use find_system to get all blocks inside this component.
            dswBlocks = find_system(getPath(component), ...
                'SearchDepth', 1, ...
                'BlockType', 'DataStoreWrite');
            dsrBlocks = find_system(getPath(component), ...
                'SearchDepth', 1, ...
                'BlockType', 'DataStoreRead');

            % Create a ResultDetail object for each data store read and write block.
```

```
            % Group ResultDetails by the data store name.
            details1 = slmetric.metric.ResultDetail.empty();
            for i=1:length(dswBlocks)
                details1(i) = slmetric.metric.ResultDetail(getfullname(dswBlocks{i}),...
                              get_param(dswBlocks{i}, 'Name'));
               groupID = get_param(dswBlocks{i},'DataStoreName');
               groupName = get_param(dswBlocks{i},'DataStoreName');
                details1(i).setGroup(groupID, groupName);
                details1(i).Value = 1;
            end

            details2 = slmetric.metric.ResultDetail.empty();
            for i=1:length(dsrBlocks)
                details2(i) = slmetric.metric.ResultDetail(getfullname(dsrBlocks{i}),...
                   get_param(dsrBlocks{i}, 'Name'));
                groupID = get_param(dsrBlocks{i},'DataStoreName');
                groupName = get_param(dsrBlocks{i},'DataStoreName');
                details2(i).setGroup(groupID, groupName);
                details2(i).Value = 1;
            end

            res = slmetric.metric.Result();
            res.ComponentID = component.ID;
            res.MetricID = this.ID;
            res.Value = length(dswBlocks)+ length(dsrBlocks);
            res.Details = [details1 details2];
        end
    end
end
```

In the `DataStoreCount` metric class, the `SupportsResultDetail` method is set to true. The metric algorithm contains the logic for the `setGroup` method.

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

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

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

Load the `sldemo_mdlref_dsm` model.

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

Create a metric engine object and set the analysis root.

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

Collect metric data for the Data Store count metric.

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

For each `slmetric.metric.Result` object, display the `ComponentPath`. For each `slmetric.metric.ResultDetails` object, display the Data Store group name and identifier.

```
for n=1:length(rc.Results)
    if rc.Results(n).Value > 0
    for m=1:length(rc.Results(n).Details)
      disp(['ComponentPath: ',rc.Results(n).ComponentPath]);
          disp(['Group Name: ',rc.Results(n).Details(m).getGroupName]);
          disp(['Group Identifier: ',rc.Results(n).Details(m).getGroupIdentifier]);
       end
    else
        disp(['No results for ComponentPath: ',rc.Results(n).ComponentPath]);
    end
    disp(' ');
end
```

Here are the results.

```
ComponentPath: sldemo_mdlref_dsm
Group Name: ErrorCond
Group Identifier: ErrorCond

No results for ComponentPath: sldemo_mdlref_dsm/More Info1

ComponentPath: sldemo_mdlref_dsm_bot
Group Name: RefSignalVal
Group Identifier: RefSignalVal

ComponentPath: sldemo_mdlref_dsm_bot2
Group Name: ErrorCond
Group Identifier: ErrorCond

ComponentPath: sldemo_mdlref_dsm_bot/PositiveSS
Group Name: RefSignalVal
Group Identifier: RefSignalVal

ComponentPath: sldemo_mdlref_dsm_bot/NegativeSS
Group Name: RefSignalVal
Group Identifier: RefSignalVal
```

For this example, unregister the data store count metric.

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

Close the model.

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

Version History

Introduced in R2017b

See Also

`slmetric.metric.Result` | `slmetric.metric.ResultCollection` | `slmetric.metric.ResultDetail` | `slmetric.metric.getAvailableMetrics`

slmetric.config.Classification class

Package: `slmetric.config`

Specify categorical metric data ranges

Description

Use the `slmetric.config.Classification` class to classify metric data ranges as `Compliant`, `Warning`, and `NonCompliant`. The Metrics Dashboard indicates the range that your metric data falls under.

Construction

For an `slmetric.config.Threshold` object, there must be one `slmetric.config.Classification` object corresponding to the Compliant range. There can be only one compliant range. You can specify multiple `slmetric.config.Classification` objects corresponding to `Warning` and `Noncompliant` ranges.

By default, threshold objects contain an `slmetric.config.Classification` object with a `Compliant` range of `-inf` to `inf`. To add additional classification objects, use the `slmetric.config.Classification.addClassification` method.

Properties

`Category` — Categorize metric data

'Compliant' (default) | 'Warning' | 'NonCompliant'

You can classify metric data values into these three categories:

- 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.

This property is read/write.

Data Types: `char`

Range — Metric range object

`slmetric.config.MetricRange` object

For each `slmetric.metric.config.Classification` object, specify the properties of the `slmetric.config.MetricRange` object. This property is read/write.

Examples

Specify Metric Thresholds to Add to Metric Dashboard

Use the `slmetric.config` packaged classes to add threshold information to the Metrics Dashboard. You can add thresholds that define metric data ranges for these three categories:

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

Create an `slmetric.config.Configuration` object.

```
CONF = slmetric.config.Configuration.new('name', 'Config');
```

Get the default `slmetric.config.ThresholdConfiguration` object in `CONF`.

```
TC = getThresholdConfigurations(CONF);
```

Add an `slmetric.config.Threshold` object to the `slmetric.config.ThresholdConfiguration` object. This threshold is for the `mathworks.metrics.SimulinkBlockCount` metric and the `Value` property of the `slmetric.metric.Results` object.

```
T = addThreshold(TC, 'mathworks.metrics.SimulinkBlockCount', 'Value');
```

An `slmetric.config.Threshold` object contains a default `slmetric.config.Classification` object that corresponds to the `Compliant` category. Use the `slmetric.metric.MetricRange` class to specify metric values for the `Compliant` metric range.

```
C = getClassifications(T); % default classification is Compliant
C.Range.Start = 5;
C.Range.IncludeStart = 0;
C.Range.End = 100;
C.Range.IncludeEnd = 0;
```

These values specify that a compliant range is a block count from `5` to `100`. This range does not include the values `5` and `100`.

Specify values for the `Warning` metric range.

```
C = addClassification(T,'Warning');
C.Range.Start = -inf;
C.Range.IncludeStart = 0;
C.Range.End = 5;
C.Range.IncludeEnd = 1
```

These values specify that a warning is a block count between `-inf` and `5`. This range does not include `-inf`. It does include `5`.

Specify values for the `NonCompliant` metric range.

```
C = addClassification(T,'NonCompliant');
C.Range.Start = 100;
C.Range.IncludeStart = 1;
C.Range.End = inf;
C.Range.IncludeEnd = 0;
```

These values specify that a block count greater than 100 is noncompliant. This range includes `100`. It does not include `inf`.

Use the `validate` method to validate the metric ranges corresponding to the thresholds in the `slmetric.config.ThresholdConfiguration` object.

```
validate(T)
```

If the ranges are not valid, you get an error message. In this example, the ranges are valid.

Save the changes to the configuration file. Use the `slmetric.config.setActiveConfiguration` function to activate this configuration for the metric engine to use.

```
configName = 'Config.xml';
save(CONF,'FileName', configName);
slmetric.config.setActiveConfiguration(fullfile(pwd, configName));
```

You can now run the Metrics Dashboard with this custom configuration on a model.

Version History

Introduced in R2018b

See Also

`slmetric.config.Configuration` | `slmetric.config.ThresholdConfiguration` | `slmetric.config.Threshold` | `slmetric.config.MetricRange` | `slmetric.metric.ResultClassification` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.config.Configuration class

Package: `slmetric.config`

Specify metric data categories and custom metric families

Description

Instances of `slmetric.config.Configuration` contain customizations pertaining to thresholds and custom metric families. The metric engine uses these customizations when collecting data and displays them on the Metrics Dashboard.

Construction

Use the `slmetric.config.Configuration` class to add metric threshold values and custom metric families to the Metrics Dashboard. To create an `slmetric.config.Configuration` object, use the `new` method. Each `slmetric.config.Configuration` object contains one `slmetric.config.ThresholdConfiguration` object.

Properties

`Name` — Configuration object name

character vector | string scalar

Name of configuration object that you use to create Metrics Dashboard customizations. This property is read/write.

Data Types: `char`

`FileName` — Name of XML file that contains custom configurations

character vector | string scalar

Name of the XML file that contains Metrics Dashboard customizations. This property is read/write.

Data Types: `char`

`Location` — Location of XML file that contains custom configuration

character vector | string scalar

Location of the XML file that contains Metrics Dashboard customizations. This property is optional and is read/write.

Methods

getThresholdConfigurations	Specify metric threshold configurations
new	Create configuration object for customizing the Metrics Dashboard
openDefaultConfiguration	Return shipping Metrics Dashboard configuration object in base workspace
open	Create configuration object associated with XML configuration file in the base workspace
save	Save contents of `slmetric.config.Configuration` object to XML file
getMetricFamilyParameterValues	Obtain metric family Check Group IDs
isMetricFamilyParameterParameterized	Determine whether Metrics Dashboard configuration object has metric family parameter values
resetMetricFamilyParameterValues	Clear metric family parameter values
setMetricFamilyParameterValues	Obtain compliance and issues metric data on your Model Advisor configuration

Examples

Specify Metric Thresholds to Add to Metrics Dashboard

Use the `slmetric.config` packaged classes to add threshold information to the Metrics Dashboard. You can add thresholds that define metric data ranges for these categories:

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

Create an `slmetric.config.Configuration` object.

```
CONF = slmetric.config.Configuration.new('name', 'Config');
```

Get the default `slmetric.config.ThresholdConfiguration` object in `CONF`.

```
TC = getThresholdConfigurations(CONF);
```

Add an `slmetric.config.Threshold` object to the `slmetric.config.ThresholdConfiguration` object. This threshold is for the `mathworks.metrics.SimulinkBlockCount` metric and the `Value` property of the `slmetric.metric.Results` object.

```
T = addThreshold(TC, 'mathworks.metrics.SimulinkBlockCount', 'Value');
```

An `slmetric.config.Threshold` object contains a default `slmetric.config.Classification` object that corresponds to the `Compliant` category. Use the `slmetric.metric.MetricRange` class to specify metric values for the `Compliant` metric range.

```
C = getClassifications(T); % default classification is Compliant
C.Range.Start = 5;
C.Range.IncludeStart = 0;
C.Range.End = 100;
C.Range.IncludeEnd = 0;
```

These values specify that a compliant range is a block count from `5` to `100`. This range does not include the values `5` and `100`.

Specify values for the `Warning` metric range.

```
C = addClassification(T,'Warning');
C.Range.Start = -inf;
C.Range.IncludeStart = 0;
C.Range.End = 5;
C.Range.IncludeEnd = 1
```

These values specify that a warning is a block count between `-inf` and `5`. This range does not include `-inf`. It does include `5`.

Specify values for the `NonCompliant` metric range.

```
C = addClassification(T,'NonCompliant');
C.Range.Start = 100;
C.Range.IncludeStart = 1;
C.Range.End = inf;
C.Range.IncludeEnd = 0;
```

These values specify that a block count greater than 100 is noncompliant. This range includes `100`. It does not include `inf`.

Use the `validate` method to validate the metric ranges corresponding to the thresholds in the `slmetric.config.ThresholdConfiguration` object.

```
validate(T)
```

If the ranges are not valid, you get an error message. In this example, the ranges are valid.

Save the changes to the configuration file. Use the `slmetric.config.setActiveConfiguration` function to activate this configuration for the metric engine to use.

```
configName = 'Config.xml';
save(CONF,'FileName', configName);
slmetric.config.setActiveConfiguration(fullfile(pwd, configName));
```

You can now run the Metrics Dashboard with this custom configuration on a model.

Version History

Introduced in R2018b

See Also

`slmetric.config.ThresholdConfiguration` | `slmetric.config.Threshold` | `slmetric.config.MetricRange` | `slmetric.config.Classification` | `slmetric.metric.ResultClassification` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics
“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.config.MetricRange class

Package: `slmetric.config`

Specify metric data threshold values

Description

Specify metric data thresholds corresponding to the `Category` property of an `slmetric.config.Classification` object. These thresholds define metric data ranges for these three categories: complaint, noncompliant, and warning. The Metrics Dashboard alerts you to the category that your data falls under.

Construction

Use the `slmetric.config.Threshold.getClassifications` method to access the default `Compliant` `slmetric.config.Classification` object. Or, use the `slmetric.config.Threshold.addClassification` method to create `NonCompliant` and `Warning` `slmetric.config.Classication` objects. Then write directly to the `slmetric.config.MetricRange` properties.

Properties

`Start` — Beginning of a metric data range

-inf (default)

Specify the beginning of a metric range corresponding to the `Category` property of an `slmetric.config.Classification` object. This property is read/write.

Data Types: `double`

`End` — End of a metric data range

inf (default)

Specify the end of a metric range corresponding to the `Category` property of an `slmetric.config.Classification` object. This property is read/write.

Data Types: `double`

`IncludeStart` — Include the value of the `Start` property

`0` (default)

Specify whether to include the `Start` value in the metric data range corresponding to the `Category` property of an `slmetric.config.Classification` object. This property is read/write.

Data Types: `logical`

`IncludeEnd` — Include the value of the `End` property

`0` (default)

Specify whether to include the `End` value in the metric data range corresponding to the `Category` property of an `slmetric.config.Classification` object. This property is read/write.

Data Types: `logical`

Note For the **High Integrity Compliance**, **MAB Compliance**, **Actual Reuse**, and **Potential Reuse** widgets, you must specify the metric ranges as fractions.

Examples

Specify Metric Thresholds to Add to Metrics Dashboard

Use the `slmetric.config` packaged classes to add threshold information to the Metrics Dashboard. You can add thresholds that define metric data ranges for these three categories:

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

Create an `slmetric.config.Configuration` object.

```
CONF = slmetric.config.Configuration.new('name', 'Config');
```

Get the default `slmetric.config.ThresholdConfiguration` object in `CONF`.

```
TC = getThresholdConfigurations(CONF);
```

Add an `slmetric.config.Threshold` object to the `slmetric.config.ThresholdConfiguration` object. This threshold is for the `mathworks.metrics.SimulinkBlockCount` metric and the `Value` property of the `slmetric.metric.Results` object.

```
T = addThreshold(TC, 'mathworks.metrics.SimulinkBlockCount', 'Value');
```

An `slmetric.config.Threshold` object contains a default `slmetric.config.Classification` object that corresponds to the `Compliant` category. Use the `slmetric.metric.MetricRange` class to specify metric values for the `Compliant` metric range.

```
C = getClassifications(T); % default classification is Compliant
C.Range.Start = 5;
C.Range.IncludeStart = 0;
C.Range.End = 100;
C.Range.IncludeEnd = 0;
```

These values specify that a compliant range is a block count from `5` to `100`. This range does not include the values `5` and `100`.

Specify values for the `Warning` metric range.

```
C = addClassification(T,'Warning');
C.Range.Start = -inf;
C.Range.IncludeStart = 0;
C.Range.End = 5;
C.Range.IncludeEnd = 1
```

These values specify that a warning is a block count between `-inf` and `5`. This range does not include `-inf`. It does include `5`.

Specify values for the `NonCompliant` metric range.

```
C = addClassification(T,'NonCompliant');
C.Range.Start = 100;
C.Range.IncludeStart = 1;
C.Range.End = inf;
C.Range.IncludeEnd = 0;
```

These values specify that a block count greater than 100 is noncompliant. This range includes `100`. It does not include `inf`.

Use the `validate` method to validate the metric ranges corresponding to the thresholds in the `slmetric.config.ThresholdConfiguration` object.

```
validate(T)
```

If the ranges are not valid, you get an error message. In this example, the ranges are valid.

Save the changes to the configuration file. Use the `slmetric.config.setActiveConfiguration` function to activate this configuration for the metric engine to use.

```
configName = 'Config.xml';
save(CONF,'FileName', configName);
slmetric.config.setActiveConfiguration(fullfile(pwd, configName));
```

You can now run the Metrics Dashboard with this custom configuration on a model.

Version History

Introduced in R2018b

See Also

`slmetric.config.Configuration` | `slmetric.config.ThresholdConfiguration` | `slmetric.config.Threshold` | `slmetric.config.Classification` | `slmetric.metric.ResultClassification` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.config.Threshold

Object for holding metric result thresholds

Description

Specify threshold values for model metric results

Creation

Create an `slmetric.config.Threshold` object by using `addThreshold`.

Properties

`MetricID` — Metric identifier
character vector | string scalar

This property is read-only.

Metric identifier for the model metric or custom model metric, returned as a character vector.

Example: 'mathworks.metrics.SimulinkBlockCount'

Data Types: `char`

`AppliesTo` — Result object property
character vector | string scalar

This property is read-only.

`slmetric.metric.Result` property to which you apply thresholds, returned as a character vector. You can apply thresholds to the `Value` and `AggregatedValue` properties.

Data Types: `char`

Object Functions

addClassification	Add metric data classification to slmetric.config.Threshold object
getClassifications	Obtain metric data classifications
removeClassification	Remove metric threshold classification
validate	Validate metric range thresholds

Examples

Specify Metric Thresholds to Add to Metrics Dashboard

Add threshold information to the Metrics Dashboard by using `slmetric.config.Threshold` and `slmetric.config.Configuration` objects. You can add thresholds that define metric data ranges for these three categories:

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

Create an `slmetric.config.Configuration` object.

```
CONF = slmetric.config.Configuration.new('name', 'Config');
```

Get the default `slmetric.config.ThresholdConfiguration` object in `CONF`.

```
TC = getThresholdConfigurations(CONF);
```

Create an `slmetric.config.Threshold` object and add it to the `slmetric.config.ThresholdConfiguration` object. This threshold is for the `mathworks.metrics.SimulinkBlockCount` metric and the `Value` property of the `slmetric.metric.Results` object.

```
T = addThreshold(TC, 'mathworks.metrics.SimulinkBlockCount', 'Value');
```

By default, the `slmetric.config.Threshold` object contains a `slmetric.config.Classification` object that defines metric ranges in the compliant category. Get the classification object by using the function `getClassifications` on the threshold object `T`.

```
C = getClassifications(T);
```

The `Range` property of the classification object is a `slmetric.metric.MetricRange` object. Specify metric values for the compliant category by using the `slmetric.metric.MetricRange` functions on the range of the classification object.

```
C.Range.Start = 5;
C.Range.IncludeStart = 0;
C.Range.End = 100;
C.Range.IncludeEnd = 0;
```

These values specify that a compliant range is a block count from `5` to `100`. This range does not include the values `5` and `100`.

Specify values for the warning metric range.

```
C = addClassification(T,'Warning');
C.Range.Start = -inf;
C.Range.IncludeStart = 0;
C.Range.End = 5;
C.Range.IncludeEnd = 1;
```

These values specify that a warning is a block count between `-inf` and `5`. This range does not include `-inf`. It does include `5`.

Specify values for the noncompliant metric range.

```
C = addClassification(T,'NonCompliant');
C.Range.Start = 100;
C.Range.IncludeStart = 1;
C.Range.End = inf;
C.Range.IncludeEnd = 0;
```

These values specify that a block count greater than 100 is noncompliant. This range includes `100`. It does not include `inf`.

Use the `validate` method to validate the metric ranges corresponding to the thresholds in the `slmetric.config.ThresholdConfiguration` object.

```
validate(T)
```

If the ranges are not valid, you get an error message. In this example, the ranges are valid, so the function returns nothing.

Save the changes to the configuration file. Use the `slmetric.config.setActiveConfiguration` function to activate this configuration for the metric engine to use.

```
configName = 'Config.xml';
save(CONF,'FileName', configName);
slmetric.config.setActiveConfiguration(fullfile(pwd, configName));
```

You can now run the Metrics Dashboard with this custom configuration on a model.

Version History

Introduced in R2018b

See Also

`slmetric.config.Configuration` | `slmetric.config.ThresholdConfiguration` | `slmetric.config.MetricRange` | `slmetric.config.Classification` | `slmetric.metric.ResultClassification` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.config.ThresholdConfiguration class

Package: `slmetric.config`

Specify metric and `slmetric.metric.Result` property for thresholding

Description

Instances of `slmetric.config.ThresholdConfiguration` contain thresholds that you specify for a metric. Each threshold specification corresponds to an `slmetric.config.Threshold` object. An `slmetric.config.ThresholdConfiguration` object can hold multiple `slmetric.config.Threshold` objects.

Construction

For an `slmetric.config.Configuration` object, use the `getThresholdConfigurations` method to access an `slmetric.config.ThresholdConfiguration` object.

Methods

addThreshold	Create an `slmetric.config.Threshold` object
getThresholds	Obtain properties of threshold objects
removeThreshold	Remove threshold object from threshold configuration object

Examples

Specify Metric Thresholds to Add to Metrics Dashboard

Use the `slmetric.config` packaged classes to add threshold information to the Metrics Dashboard. You can add thresholds that define metric data ranges for these three categories:

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

Create an `slmetric.config.Configuration` object.

```
CONF = slmetric.config.Configuration.new('name', 'Config');
```

Get the default `slmetric.config.ThresholdConfiguration` object in `CONF`.

```
TC = getThresholdConfigurations(CONF);
```

Add an `slmetric.config.Threshold` object to the `slmetric.config.ThresholdConfiguration` object. This threshold is for the `mathworks.metrics.SimulinkBlockCount` metric and the `Value` property of the `slmetric.metric.Results` object.

```
T = addThreshold(TC, 'mathworks.metrics.SimulinkBlockCount', 'Value');
```

An `slmetric.config.Threshold` object contains a default `slmetric.config.Classification` object that corresponds to the `Compliant` category. Use the `slmetric.metric.MetricRange` class to specify metric values for the `Compliant` metric range.

```
C = getClassifications(T); % default classification is Compliant
C.Range.Start = 5;
C.Range.IncludeStart = 0;
C.Range.End = 100;
C.Range.IncludeEnd = 0;
```

These values specify that a compliant range is a block count from `5` to `100`. This range does not include the values `5` and `100`.

Specify values for the `Warning` metric range.

```
C = addClassification(T,'Warning');
C.Range.Start = -inf;
C.Range.IncludeStart = 0;
C.Range.End = 5;
C.Range.IncludeEnd = 1
```

These values specify that a warning is a block count between `-inf` and `5`. This range does not include `-inf`. It does include `5`.

Specify values for the `NonCompliant` metric range.

```
C = addClassification(T,'NonCompliant');
C.Range.Start = 100;
C.Range.IncludeStart = 1;
C.Range.End = inf;
C.Range.IncludeEnd = 0;
```

These values specify that a block count greater than 100 is noncompliant. This range includes `100`. It does not include `inf`.

Use the `slmetric.config.validate` function to validate the metric ranges corresponding to the thresholds in the `slmetric.config.ThresholdConfiguration` object.

```
validate(T)
```

If the ranges are not valid, you get an error message. In this example, the ranges are valid.

Save the changes to the configuration file. Use the `slmetric.config.setActiveConfiguration` function to activate this configuration for the metric engine to use.

```
configName = 'Config.xml';
save(CONF,'FileName', configName);
slmetric.config.setActiveConfiguration(fullfile(pwd, configName));
```

You can now run the Metrics Dashboard with this custom configuration on a model.

Version History

Introduced in R2018b

See Also

`slmetric.config.Configuration` | `slmetric.config.Threshold` | `slmetric.config.MetricRange` | `slmetric.config.Classification` |

`slmetric.metric.ResultClassification` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics
“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.metric.ResultClassification class

Package: `slmetric.metric`

Access metric data thresholds results

Description

For the `Value` and `AggregatedValue` properties of an `slmetric.metric.Result` object, access properties of the `slmetric.metric.ResultClassification` class to determine the metric data ranges that correspond to the `Compliant`, `NonCompliant`, and `Warning` categories. From an `slmetric.metric.ResultClassification` object, also determine which of the three categories your metric data falls under.

Construction

The value of the `Classifications` property of an `slmetric.metric.Result` object is the `slmetric.metric.ResultClassification` object.

Properties

`Threshold` — Model metric and `slmetric.metric.Result` property with thresholds
`slmetric.config.Threshold` object

Access this property to determine the model metric and the `slmetric.metric.Result` property that has thresholds.

`Classification` — Status of component data
'Compliant' | 'NonCompliant' | 'Warning' | 'Uncategorized'

Metric data values fall into one of these four categories:

- 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.
- Uncategorized — Metric data that has no threshold values set.

If at least one component is `NonCompliant`, this property returns `NonCompliant`. If at least one component is `Warning` and no components are `NonCompliant`, this property returns `Warning`. If all components are `Compliant`, this category returns `Compliant`.

This property is read-only.

Examples

Collect and Classify Metric Data

For the `mathworks.metric.SimulinkBlockCount` metric, define `slmetric.metric.Result` values corresponding to `Compliant`, `NonCompliant`, and `Warning` categories. For the `sldemo_mdl_ref` model, run the metrics engine and categorize results for this metric.

Open the model.

```
openExample('sldemo_mdlref_basic');
```

Create an `slmetric.config.Configuration` object.

```
CONF = slmetric.config.Configuration.new('name', 'Config');
```

Get the default `slmetric.config.ThresholdConfiguration` object in `CONF`.

```
TC = getThresholdConfigurations(CONF);
```

Add an `slmetric.config.Threshold` object to the `slmetric.config.ThresholdConfiguration` object. This threshold is for the `mathworks.metrics.SimulinkBlockCount` metric and the `Value` property of the `slmetric.metric.Results` object.

```
T = addThreshold(TC, 'mathworks.metrics.SimulinkBlockCount', 'Value');
```

An `slmetric.config.Threshold` object contains a default `slmetric.config.Classification` object that corresponds to the `Compliant` category. Use the `slmetric.metric.MetricRange` class to specify metric values for the `Compliant`, `NonCompliant`, and `Warning` metric ranges.

```
C = getClassifications(T); % default classification is Compliant
C.Range.Start = 5;
C.Range.IncludeStart = 0;
C.Range.End = 100;
C.Range.IncludeEnd = 0;

C = addClassification(T,'Warning');
C.Range.Start = -inf;
C.Range.IncludeStart = 0;
C.Range.End = 5;
C.Range.IncludeEnd = 1

C = addClassification(T,'NonCompliant');
C.Range.Start = 100;
C.Range.IncludeStart = 1;
C.Range.End = inf;
C.Range.IncludeEnd = 0;
```

Use the `validate` method to validate the metric ranges corresponding to the thresholds in the `slmetric.config.ThresholdConfiguration` object.

```
validate(T)
```

If the ranges are not valid, you get an error message. In this example, the ranges are valid.

Save the changes to the configuration file. Use the `slmetric.config.setActiveConfiguration` function to activate this configuration for the metric engine to use.

```
configName = 'Config.xml';
save(CONF,'FileName', configName);
slmetric.config.setActiveConfiguration(fullfile(pwd, configName));
```

Create an `slmetric.Engine` object, set the root in the model for analysis, and collect data for the `mathworks.metrics.SimulinkBlockCount` metric.

```
metric_engine = slmetric.Engine();
setAnalysisRoot(metric_engine, 'Root',  'sldemo_mdlref_basic');
execute(metric_engine, 'mathworks.metrics.SimulinkBlockCount');
```

Get the model metric data that returns an array of `slmetric.metric.ResultCollection` objects, `res_col`.

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

Display the results for the `mathworks.metrics.SimulinkBlockCount` metric.

```
for n=1:length(res_col)
    if res_col(n).Status == 0
        result = res_col(n).Results;

        for m=1:length(result)
            disp(['MetricID: ',result(m).MetricID]);
            disp(['  ComponentPath: ', result(m).ComponentPath]);
            disp(['  Value: ', num2str(result(m).Value)]);
            disp(['  Classifications: ', result(m).Classifications.Classification.Category]);
            disp(['  Measures: ', num2str(result(m).Measures)]);
            disp(['  AggregatedMeasures: ', num2str(result(m).AggregatedMeasures)]);
        end
    else
        disp(['No results for:', result(n).MetricID]);
    end
    disp(' ');
end
```

```
MetricID: mathworks.metrics.SimulinkBlockCount
  ComponentPath: sldemo_mdlref_basic
  Value: 12
  Classifications: Compliant
  Measures:
  AggregatedMeasures:
MetricID: mathworks.metrics.SimulinkBlockCount
  ComponentPath: sldemo_mdlref_basic/More Info
  Value: 0
  Classifications: Warning
  Measures:
  AggregatedMeasures:
MetricID: mathworks.metrics.SimulinkBlockCount
  ComponentPath: sldemo_mdlref_counter
  Value: 18
  Classifications: Compliant
  Measures:
  AggregatedMeasures:
```

For `ComponentPath: sldemo_mdlref_basic` and `ComponentPath: sldemo_mdlref_counter`, the results are `Compliant` because of the values `12` and `18`, respectively. For `ComponentPath: sldemo_mdlref_basic/More Info`, the results fall under the `Warning` category because of the `0` value.

Version History

Introduced in R2018b

See Also

`slmetric.config.Configuration` | `slmetric.config.ThresholdConfiguration` | `slmetric.config.Threshold` | `slmetric.config.MetricRange` |

slmetric.config.Classification | slmetric.config.getActiveConfiguration | slmetric.config.setActiveConfiguration

getThresholdConfigurations

Class: `slmetric.config.Configuration`
Package: `slmetric.config`

Specify metric threshold configurations

Syntax

```
TH = getThresholdConfigurations(CO)
```

Description

`TH = getThresholdConfigurations(CO)` returns the `slmetric.config.ThresholdConfiguration` object that an `slmetric.config.Configuration` object owns. Use this object to hold specific metric threshold configurations. Metric threshold configurations are compliant, warning, and noncompliant ranges for a specific metric.

Input Arguments

`CO` — Configuration object

`slmetric.config.Configuration` object

`slmetric.config.Configuration` object for which you create a metric threshold configuration. By default, an `slmetric.config.Configuration` object holds an empty `slmetric.config.ThresholdConfiguration` object.

Output Arguments

`TH` — Metric threshold configuration object

`slmetric.config.ThresholdConfiguration` object

`slmetric.config.ThresholdConfiguration` object for which you add thresholds corresponding to compliant, noncompliant, and warning ranges for a specific metric.

Examples

Add Thresholds to a Threshold Configuration Object

By default, an `slmetric.config.Configuration` object holds one `slmetric.config.ThresholdConfiguration` object. Use the `getThresholdConfigurations` method to add this object to the base workspace. You can then use the `slmetric.config.addThreshold` method to add `slmetric.config.Threshold` objects to this `slmetric.config.ThresholdConfiguration` object.

Create an `slmetric.config.Configuration` object.

```
CONF = slmetric.config.Configuration.new('name', 'Config');
```

Get the default `slmetric.config.ThresholdConfiguration` object in `CONF`.

```
TC = getThresholdConfigurations(CONF);
```

Add an `slmetric.config.Threshold` object to the `slmetric.config.ThresholdConfiguration` object `TC`. This threshold is for the `mathworks.metrics.SubSystemCount` metric and the `Value` property of the `slmetric.metric.Results` object.

```
E = addThreshold(TC,'mathworks.metrics.SubSystemCount','Value');
```

Use the `slmetric.config.Classification` and `slmetric.config.MetricRange` class properties to specify threshold values corresponding to the `mathworks.metrics.SubsystemCount` metric.

Version History

Introduced in R2018b

See Also

`slmetric.config.Configuration` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.config.Configuration.new

Class: `slmetric.config.Configuration`
Package: `slmetric.config`

Create configuration object for customizing the Metrics Dashboard

Syntax

```
Co = slmetric.config.Configuration.new('Name', 'Config')
```

Description

Create an `slmetric.config.Configuration` object for holding Metrics Dashboard customizations pertaining to metric thresholds and custom metric families. Use the `save` command to create and store the associated XML configuration file.

`Co = slmetric.config.Configuration.new('Name', 'Config')` creates a configuration object.

Input Arguments

`Name` — Name of configuration object that is tagged in XML file

character vector | string scalar

Name of configuration object in XML file that contains Metrics Dashboard customizations pertaining to metric thresholds and custom metric families.

Data Types: `char`

Output Arguments

`co` — Configuration object

character vector | string scalar

Name of `slmetric.config.Configuration` object that contains Metrics Dashboard customizations pertaining to metric thresholds and custom families.

Data Types: `char`

Examples

Create a Configuration Object

Use the `new` method to create an `slmetric.config.Configuration` object. The configuration object contains information on custom metric families and metric thresholds. As an input, specify a configuration object name. This name is then associated with a tag in the configuration object XML file. After adding information to the configuration object, use the `slmetric.config.Configuration.save` method to create and store the associated XML file.

```
CONF = slmetric.config.Configuration.new('Name','Config')
```

Version History

Introduced in R2018b

See Also

`slmetric.config.Configuration` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

openDefaultConfiguration

Class: `slmetric.config.Configuration`
Package: `slmetric.config`

Return shipping Metrics Dashboard configuration object in base workspace

Syntax

```
DT = slmetric.config.Configuration.openDefaultConfiguration
```

Description

`DT = slmetric.config.Configuration.openDefaultConfiguration` returns the `slmetric.config.Configuration` object corresponding to the shipping Metrics Dashboard configuration in the base workspace. Use this object to add or remove threshold values corresponding to Compliant, NonCompliant, or Warning categories. These `MetricID`s contain default shipping thresholds:

- `mathworks.metrics.CloneContent`
- `mathworks.metrics.CyclomaticComplexity`
- `mathworks.metrics.DiagnosticWarningsCount`
- `mathworks.metrics.MatlabCodeAnalyzerWarnings`
- `mathworks.metrics.ModelAdvisorCheckCompliance.hisl_do178`
- `mathworks.metrics.ModelAdvisorCheckCompliance.maab`
- `mathworks.metrics.ModelAdvisorCheckIssues.hisl_do178`
- `mathworks.metrics.ModelAdvisorCheckIssues.maab`

You can also use this object to obtain compliance and issues metric data on your Model Advisor configuration.

Output Arguments

`DT` — Default Metric Dashboard threshold configuration object

`slmetric.config.Configuration` object

`slmetric.config.ThresholdConfiguration` object for adding and removing thresholds corresponding to Compliant, Noncompliant, and Warning Categories for a specific metric.

Examples

Open the Shipping `slmetric.config.Configuration` Object

Use the `openDefaultConfiguration` method to add the shipping `slmetric.config.Configuration` object to the base workspace. If you modify the information that this configuration object contains, use the `slmetric.config.Configuration.save` method to save this information to an XML file.

```
Config = slmetric.config.Configuration.openDefaultConfiguration
```

Version History

Introduced in R2018b

See Also

`slmetric.config.ThresholdConfiguration` | `slmetric.config.Threshold` | `slmetric.config.MetricRange` | `slmetric.config.Classification` | `slmetric.metric.ResultClassification` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.config.Configuration.open

Class: `slmetric.config.Configuration`
Package: `slmetric.config`

Create configuration object associated with XML configuration file in the base workspace

Syntax

```
Co = slmetric.config.Configuration.open('FileName','myConfig.xml',...
'Location', pwd)
```

Description

Reads the contents of an XML file containing Metrics Dashboard customizations into memory and returns the corresponding configuration object. The XML file contains customizations pertaining to metric thresholds and custom metric families. If you modify the contents of the configuration object, invoke the `save` method to write to the associated XML file.

`Co = slmetric.config.Configuration.open('FileName','myConfig.xml',... 'Location', pwd)` reads a configuration file.

Note If you do not supply an input argument, the `slmetric.config.Configuration.open` command reads the contents of the default Metrics Dashboard configuration XML file into memory and returns the corresponding `slmetric.dashboard.Configuration` object.

Input Arguments

`FileName` — Name of XML file

character vector | string scalar

Name of XML file containing Metrics Dashboard customizations pertaining to metric thresholds and custom metric families.

Data Types: `char`

`Location` — Folder containing XML file

character vector | string scalar

Name of folder containing XML file that contains Metrics Dashboard customizations pertaining to metric thresholds and custom metric families. This input argument is optional.

Data Types: `char`

`locale` — Name of folder containing XML file

character vector | string scalar

Name of folder containing XML file that contains Metrics Dashboard customizations. This input argument is optional.

Data Types: `char`

Output Arguments

`Co` — Configuration object

character vector | string scalar

`slmetric.config.Configuration` object that you want to open.

Data Types: `char`

Examples

Access an Existing Configuration Object

Use the `open` method to add an existing `slmetric.config.Configuration` object to the base workspace. As an input, specify the name of the XML file that contains the information on the custom metric families and metric thresholds corresponding to the configuration object. If you modify the information that this configuration object contains, use the `slmetric.config.Configuration.save` method to save this information to the XML file.

```
CONF = slmetric.config.Configuration.open('FileName', 'myConfig.xml',...
 'Location', pwd());
```

Version History

Introduced in R2018b

See Also

`slmetric.config.Configuration` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

save

Class: `slmetric.config.Configuration`
Package: `slmetric.config`

Save contents of `slmetric.config.Configuration` object to XML file

Syntax

```
save(Config,'FileName','myConfig.xml',... 'Location',pwd, 'locale', 'ja_JP');
```

Description

Save the contents of a configuration object to an XML file. The configuration object contains Metrics Dashboard customizations pertaining to metric thresholds and custom metric families.

`save(Config,'FileName','myConfig.xml',... 'Location',pwd, 'locale', 'ja_JP');` saves the contents of a configuration object to an XML file.

Note Do not manually edit the XML file.

Input Arguments

`Config` — Configuration object

`slmetric.config.Configuration` object

`slmetric.config.Configuration` object to create Metrics Dashboard customizations. This property is read/write.

`FileName` — Name of XML file

character vector | string scalar

Name of XML file that contains Metrics Dashboard customizations pertaining to metric thresholds and custom metric families.

Data Types: `char`

`Location` — Name of folder containing XML file

character vector | string scalar

Name of folder containing XML file, which contains Metrics Dashboard customizations pertaining to metric thresholds and custom metric families.

Data Types: `char`

`locale` — Create folder that is to contain XML file

character vector | string scalar

Name of new folder that is to contain XML file that contains information on Metrics Dashboard customizations pertaining to metric thresholds and custom metric families. If you do not specify a

value for `locale`, Simulink creates the XML file in the folder that you specify with the `Location` property. This input argument is optional.

Data Types: `char`

Examples

Serialize a Configuration Object to XML File

Serialize configuration object to XML file.

Use the `save` method to add an existing `slmetric.config.Configuration` object to the base workspace. As an input, specify the name of the XML file that contains information on the custom metric families and metric thresholds corresponding to the configuration object. If you modify the information that this configuration object contains, use the `slmetric.config.Configuration.save` method to save the information to this file.

```
save(CONF,'config','FileName','Configfile.xml','Location',pwd)
```

Use the `slmetric.config.setActiveConfiguration` function to specify that the metric engine use this configuration.

```
slmetric.config.setActiveConfiguration('C:\temp\Configfile.xml');
```

Version History

Introduced in R2018b

See Also

`slmetric.config.Configuration` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

addClassification

Package: `slmetric.config`

Add metric data classification to `slmetric.config.Threshold` object

Syntax

```
TC = addClassification(threshold,category)
```

Description

`TC = addClassification(threshold,category)` adds a classification category to an `slmetric.config.Threshold` object.

After creating the object, use the `slmetric.config.MetricRange` object functions to specify ranges for `Compliant`, `NonCompliant`, and `Warning`. By default, a classification object has a `Compliant` range of `-inf` to `inf`. The Metrics Dashboard indicates the range that your metric data falls under.

Examples

Specify Metric Thresholds to Add to Metrics Dashboard

Add threshold information to the Metrics Dashboard by using `slmetric.config.Threshold` and `slmetric.config.Configuration` objects. You can add thresholds that define metric data ranges for these three categories:

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

Create an `slmetric.config.Configuration` object.

```
CONF = slmetric.config.Configuration.new('name', 'Config');
```

Get the default `slmetric.config.ThresholdConfiguration` object in `CONF`.

```
TC = getThresholdConfigurations(CONF);
```

Create an `slmetric.config.Threshold` object and add it to the `slmetric.config.ThresholdConfiguration` object. This threshold is for the `mathworks.metrics.SimulinkBlockCount` metric and the `Value` property of the `slmetric.metric.Results` object.

```
T = addThreshold(TC, 'mathworks.metrics.SimulinkBlockCount', 'Value');
```

By default, the `slmetric.config.Threshold` object contains a `slmetric.config.Classification` object that defines metric ranges in the compliant category. Get the classification object by using the function `getClassifications` on the threshold object `T`.

```
C = getClassifications(T);
```

The `Range` property of the classification object is a `slmetric.metric.MetricRange` object. Specify metric values for the compliant category by using the `slmetric.metric.MetricRange` functions on the range of the classification object.

```
C.Range.Start = 5;
C.Range.IncludeStart = 0;
C.Range.End = 100;
C.Range.IncludeEnd = 0;
```

These values specify that a compliant range is a block count from `5` to `100`. This range does not include the values `5` and `100`.

Specify values for the warning metric range.

```
C = addClassification(T,'Warning');
C.Range.Start = -inf;
C.Range.IncludeStart = 0;
C.Range.End = 5;
C.Range.IncludeEnd = 1;
```

These values specify that a warning is a block count between `-inf` and `5`. This range does not include `-inf`. It does include `5`.

Specify values for the noncompliant metric range.

```
C = addClassification(T,'NonCompliant');
C.Range.Start = 100;
C.Range.IncludeStart = 1;
C.Range.End = inf;
C.Range.IncludeEnd = 0;
```

These values specify that a block count greater than 100 is noncompliant. This range includes `100`. It does not include `inf`.

Use the `validate` method to validate the metric ranges corresponding to the thresholds in the `slmetric.config.ThresholdConfiguration` object.

```
validate(T)
```

If the ranges are not valid, you get an error message. In this example, the ranges are valid, so the function returns nothing.

Save the changes to the configuration file. Use the `slmetric.config.setActiveConfiguration` function to activate this configuration for the metric engine to use.

```
configName = 'Config.xml';
save(CONF,'FileName', configName);
slmetric.config.setActiveConfiguration(fullfile(pwd, configName));
```

You can now run the Metrics Dashboard with this custom configuration on a model.

Input Arguments

`threshold` — Metric threshold

`slmetric.config.threshold` object

Metric threshold, specified as an `slmetric.config.threshold` object.

`category` — Threshold category
`'Compliant'` (default) | `'Warning'` | `'NonCompliant'`

Threshold category, specified as one of these three categories:

- 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

Data Types: `char`

Output Arguments

`TC` — Classification category
`slmetric.config.Classification` object

Classification category, returned as a `slmetric.config.Classification` object.

Version History

Introduced in R2018b

See Also

`slmetric.config.Threshold` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics
“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getClassifications

Package: `slmetric.config`

Obtain metric data classifications

Syntax

```
classifications = getClassifications(threshold)
```

Description

`classifications = getClassifications(threshold)` returns the `slmetric.config.Classification` object or an array of `slmetric.config.Classification` objects that are in the threshold object.

Examples

Remove Classification Object from Threshold Object

Add classification information to an `slmetric.config.Threshold` object. Then, use the `getClassifications` function to identify the `slmetric.config.Classification` objects that belong to the threshold object. Use the `removeClassification` function to remove an classification object from the threshold object.

Create an `slmetric.config.Configuration` object and get the default `slmetric.config.ThresholdConfiguration` object.

```
CONF = slmetric.config.Configuration.new('name', 'Config');
TC = getThresholdConfigurations(CONF);
```

Create an `slmetric.config.Threshold` object and add it to the `slmetric.config.ThresholdConfiguration` object. This threshold is for the `mathworks.metrics.SimulinkBlockCount` metric and the `Value` property of the `slmetric.metric.Results` object.

```
T = addThreshold(TC, 'mathworks.metrics.SimulinkBlockCount', 'Value');
```

Get the default classification object, which defines the compliant category, by using the function `getClassifications` on the threshold object `T`. Specify metric values for the compliant category by using the `slmetric.metric.MetricRange` functions on the range of the classification object.

```
C = getClassifications(T);
C.Range.Start = 5;
C.Range.IncludeStart = 0;
C.Range.End = 100;
C.Range.IncludeEnd = 0;
```

Specify values for the `Warning` metric range.

```
C = addClassification(T,'Warning');
C.Range.Start = -inf;
C.Range.IncludeStart = 0;
```

```
C.Range.End = 5;
C.Range.IncludeEnd = 1;
```

Specify values for the `NonCompliant` metric range.

```
C = addClassification(T,'NonCompliant');
C.Range.Start = 100;
C.Range.IncludeStart = 1;
C.Range.End = inf;
C.Range.IncludeEnd = 0;
```

Now, the `slmetric.config.Threshold` object, `T`, contains three `slmetric.config.Classification` objects. Each one corresponds to one of the categories compliant, noncompliant, and warning.

```
P = getClassifications(T)
```

```
P =

  1×3 Classification array with properties:

    Category
    Range
```

Look at the contents of the `Category` property.

```
P.Category
```

```
P.Category

ans =

    'Warning'


ans =

    'Compliant'


ans =

    'NonCompliant'
```

Use the `removeClassification` function to remove the warning category from the `slmetric.config.Threshold` object.

```
removeClassification(T,P(1))
```

Input Arguments

`threshold` — Metric data thresholds

`slmetric.config.Threshold` object

Metric data thresholds, specified as an `slmetric.config.Threshold` object.

Output Arguments

`classifications` — Classification object
`slmetric.config.Classification` object | array of `slmetric.config.classification` objects

`slmetric.config.Classification` object or array of `slmetric.config.Classification` objects that contain metric data classifications.

Version History

Introduced in R2018b

See Also

`slmetric.config.Threshold` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics
“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

removeClassification

Package: `slmetric.config`

Remove metric threshold classification

Syntax

```
removeClassification(T,C1)
```

Description

`removeClassification(T,C1)` removes the `slmetric.config.Classification` object `C1` from the `slmetric.config.Threshold` object `T`.

Examples

Remove Classification Object from Threshold Object

Add classification information to an `slmetric.config.Threshold` object. Then, use the `getClassifications` function to identify the `slmetric.config.Classification` objects that belong to the threshold object. Use the `removeClassification` function to remove an classification object from the threshold object.

Create an `slmetric.config.Configuration` object and get the default `slmetric.config.ThresholdConfiguration` object.

```
CONF = slmetric.config.Configuration.new('name', 'Config');
TC = getThresholdConfigurations(CONF);
```

Create an `slmetric.config.Threshold` object and add it to the `slmetric.config.ThresholdConfiguration` object. This threshold is for the `mathworks.metrics.SimulinkBlockCount` metric and the `Value` property of the `slmetric.metric.Results` object.

```
T = addThreshold(TC, 'mathworks.metrics.SimulinkBlockCount', 'Value');
```

Get the default classification object, which defines the compliant category, by using the function `getClassifications` on the threshold object `T`. Specify metric values for the compliant category by using the `slmetric.metric.MetricRange` functions on the range of the classification object.

```
C = getClassifications(T);
C.Range.Start = 5;
C.Range.IncludeStart = 0;
C.Range.End = 100;
C.Range.IncludeEnd = 0;
```

Specify values for the `Warning` metric range.

```
C = addClassification(T,'Warning');
C.Range.Start = -inf;
C.Range.IncludeStart = 0;
```

```
C.Range.End = 5;
C.Range.IncludeEnd = 1;
```

Specify values for the `NonCompliant` metric range.

```
C = addClassification(T,'NonCompliant');
C.Range.Start = 100;
C.Range.IncludeStart = 1;
C.Range.End = inf;
C.Range.IncludeEnd = 0;
```

Now, the `slmetric.config.Threshold` object, `T`, contains three `slmetric.config.Classification` objects. Each one corresponds to one of the categories compliant, noncompliant, and warning.

```
P = getClassifications(T)
```

```
P =

  1×3 Classification array with properties:

    Category
    Range
```

Look at the contents of the `Category` property.

```
P.Category
```

```
P.Category

ans =

    'Warning'


ans =

    'Compliant'


ans =

    'NonCompliant'
```

Use the `removeClassification` function to remove the warning category from the `slmetric.config.Threshold` object.

```
removeClassification(T,P(1))
```

Input Arguments

T — Threshold object

`slmetric.config.Threshold` object

Threshold object, specified as an `slmetric.config.Threshold` object.

C1 — Classification object to remove

`slmetric.config.Classification` object

Classification object to remove, specified as an `slmetric.config.Classification` object.

Version History

Introduced in R2018b

See Also

`slmetric.config.Threshold` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

validate

Package: `slmetric.config`

Validate metric range thresholds

Syntax

```
validate(threshold)
```

Description

`validate(threshold)` checks each `slmetric.config.Classification` object in the threshold object to verify that the values specified for the `Category` and `Range` properties are valid. The ranges of each category must not overlap and the ranges together must cover from `-inf` to `inf`. If the values are not valid, the function returns an error informing you of what to fix.

Examples

Specify Metric Thresholds to Add to Metrics Dashboard

Add threshold information to the Metrics Dashboard by using `slmetric.config.Threshold` and `slmetric.config.Configuration` objects. You can add thresholds that define metric data ranges for these three categories:

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

Create an `slmetric.config.Configuration` object.

```
CONF = slmetric.config.Configuration.new('name', 'Config');
```

Get the default `slmetric.config.ThresholdConfiguration` object in `CONF`.

```
TC = getThresholdConfigurations(CONF);
```

Create an `slmetric.config.Threshold` object and add it to the `slmetric.config.ThresholdConfiguration` object. This threshold is for the `mathworks.metrics.SimulinkBlockCount` metric and the `Value` property of the `slmetric.metric.Results` object.

```
T = addThreshold(TC, 'mathworks.metrics.SimulinkBlockCount', 'Value');
```

By default, the `slmetric.config.Threshold` object contains a `slmetric.config.Classification` object that defines metric ranges in the compliant category. Get the classification object by using the function `getClassifications` on the threshold object `T`.

```
C = getClassifications(T);
```

The `Range` property of the classification object is a `slmetric.metric.MetricRange` object. Specify metric values for the compliant category by using the `slmetric.metric.MetricRange` functions on the range of the classification object.

```
C.Range.Start = 5;
C.Range.IncludeStart = 0;
C.Range.End = 100;
C.Range.IncludeEnd = 0;
```

These values specify that a compliant range is a block count from `5` to `100`. This range does not include the values `5` and `100`.

Specify values for the warning metric range.

```
C = addClassification(T,'Warning');
C.Range.Start = -inf;
C.Range.IncludeStart = 0;
C.Range.End = 5;
C.Range.IncludeEnd = 1;
```

These values specify that a warning is a block count between `-inf` and `5`. This range does not include `-inf`. It does include `5`.

Specify values for the noncompliant metric range.

```
C = addClassification(T,'NonCompliant');
C.Range.Start = 100;
C.Range.IncludeStart = 1;
C.Range.End = inf;
C.Range.IncludeEnd = 0;
```

These values specify that a block count greater than 100 is noncompliant. This range includes `100`. It does not include `inf`.

Use the `validate` method to validate the metric ranges corresponding to the thresholds in the `slmetric.config.ThresholdConfiguration` object.

```
validate(T)
```

If the ranges are not valid, you get an error message. In this example, the ranges are valid, so the function returns nothing.

Save the changes to the configuration file. Use the `slmetric.config.setActiveConfiguration` function to activate this configuration for the metric engine to use.

```
configName = 'Config.xml';
save(CONF,'FileName', configName);
slmetric.config.setActiveConfiguration(fullfile(pwd, configName));
```

You can now run the Metrics Dashboard with this custom configuration on a model.

Input Arguments

`threshold` — Threshold object to validate

`slmetric.config.Threshold` object

Threshold object to validate, specified as an `slmetric.config.Threshold` object.

Version History

Introduced in R2018b

See Also

`slmetric.config.Threshold` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

addThreshold

Class: `slmetric.config.ThresholdConfiguration`
Package: `slmetric.config`

Create an `slmetric.config.Threshold` object

Syntax

```
TH = addThreshold(TC,metricid,thresholdproperty)
```

Description

Create an `slmetric.config.Threshold` object to apply thresholds to the data for a specific metric.

`TH = addThreshold(TC,metricid,thresholdproperty)` creates an `slmetric.config.Threshold` object.

Input Arguments

`TC` — Threshold configuration object

`slmetric.config.ThresholdConfiguration` object

`slmetric.config.ThresholdConfiguration` object for which you add an `slmetric.config.Threshold` object.

Data Types: `char`

`metricid` — Metric identifier

character vector | string scalar

Metric identifier for model metric or custom model metric that you create.

Example: 'mathworks.metrics.SimulinkBlockCount'

Data Types: `char`

`thresholdproperty` — Result object property

character vector | string scalar

`slmetric.metric.Result` property to which you apply thresholds. You can apply thresholds to the `Value` and `AggregatedValue` properties.

Output Arguments

`TH` — Threshold object

`slmetric.config.Threshold` object

`slmetric.config.Threshold` object for applying thresholds to either the `Value` or `AggregatedValue` properties of an `slmetric.metric.Result` object.

Examples

Add Thresholds to a Threshold Configuration Object

By default, an `slmetric.config.Configuration` object holds one `slmetric.config.ThresholdConfiguration` object. Use the `getThresholdConfigurations` method to add this object to the base workspace. Use the `slmetric.config.addThreshold` method to add `slmetric.config.Threshold` objects to this `slmetric.config.ThresholdConfiguration` object.

Create an `slmetric.config.Configuration` object.

```
CONF = slmetric.config.Configuration.new('name', 'Config');
```

Get the default `slmetric.config.ThresholdConfiguration` object in `CONF`.

```
TC = getThresholdConfigurations(CONF);
```

Add an `slmetric.config.Threshold` object to the `slmetric.config.ThresholdConfiguration` object `TC`. This threshold is for the `mathworks.metrics.SubSystemCount` metric and the `Value` property of the `slmetric.metric.Results` object.

```
E = addThreshold(TC,'mathworks.metrics.SubSystemCount','Value');
```

Use the `slmetric.config.Classification` and `slmetric.config.MetricRange` classes to specify threshold values corresponding to the `mathworks.metrics.SubsystemCount` metric.

Version History

Introduced in R2018b

See Also

`slmetric.config.ThresholdConfiguration` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getThresholds

Class: `slmetric.config.ThresholdConfiguration`
Package: `slmetric.config`

Obtain properties of threshold objects

Syntax

```
T = getThresholds(TH,metricid)
```

Description

Determine the properties of the threshold objects that a threshold configuration object holds. You can also use this method with the `slmetric.config.ThresholdConfiguration.removeThresholds` method to identify and remove a threshold object from a `slmetric.config.ThresholdConfiguration` object.

`T = getThresholds(TH,metricid)` creates a threshold object or an array of threshold objects.

Input Arguments

`TH` — Threshold configuration object

`slmetric.config.ThresholdConfiguration` object

`slmetric.config.ThresholdConfiguration` object for which you want information on the threshold objects it holds.

`MetricID` — Metric identifier

character vector | string scalar

Metric identifier for model metric or custom model metric that you create. This argument is optional. If you do not specify a metricID, you get information on all thresholds that the Threshold configuration object holds.

Example: 'mathworks.metrics.SimulinkBlockCount'

Data Types: `char`

Output Arguments

`T` — Threshold object or array of threshold objects

character vector | string scalar | array of character vectors | array of string scalars

`slmetric.config.Threshold` object or array of `slmetric.config.Threshold` objects corresponding to the `slmetric.config.ThresholdConfiguration` object that you specify as an input.

Examples

Identify Threshold Objects in a Threshold Configuration Object

Use the `getThresholds` method to identify the `slmetric.config.Threshold` objects that belong to an `slmetric.config.ThresholdConfiguration` object.

For the `slmetric.config.ThresholdConfiguration` object `TC`, use the `getThresholds` method.

```
getThresholds(TC)
```

```
1×2 Threshold array with properties:

    MetricID
    AppliesTo
```

The `slmetric.config.ThresholdConfiguration` object `TC` contains two `slmetric.config.Threshold` objects.

Version History

Introduced in R2018b

See Also

`slmetric.config.ThresholdConfiguration` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

removeThreshold

Class: `slmetric.config.ThresholdConfiguration`
Package: `slmetric.config`

Remove threshold object from threshold configuration object

Syntax

```
removeThreshold(TC,T)
```

Description

Remove a threshold object from a threshold configuration object.

`removeThreshold(TC,T)` removes the `slmetric.config.Threshold` object `T` from the `slmetric.config.ThresholdConfiguration` object `TC`.

Input Arguments

TC — Threshold configuration object

`slmetric.config.ThresholdConfiguration` object

`slmetric.config.ThresholdConfiguration` object from which you want to remove an `slmetric.config.Threshold` object.

Data Types: `char`

T — Threshold object

`slmetric.config.Threshold` | object

`slmetric.config.Threshold` object that you want to remove from an `slmetric.config.ThresholdConfiguration` object.

Data Types: `char`

Examples

Remove Threshold Object from a Threshold Configuration Object

Use the `getThresholds` method to identify the `slmetric.config.Threshold` objects that belong to an `slmetric.config.ThresholdConfiguration` object. Then, use the `removeThreshold` method to remove an `slmetric.config.Threshold` object.

For the `slmetric.config.ThresholdConfiguration` object `TC`, use the `getThresholds` method.

```
A = getThresholds(TC)
```

```
A =
```

```
  1×2 Threshold array with properties:

    MetricID
    AppliesTo
```

The `slmetric.config.ThresholdConfiguration` object `TC` contains two `slmetric.config.Threshold` objects.

Identify the `slmetric.config.Threshold` object that you want to remove from the `slmetric.config.ThresholdConfiguration` object.

```
A.MetricID
```

```
ans =

    'mathworks.metrics.SimulinkBlockCount'


ans =

    'mathworks.metricchecks.SubSystemCount'
```

Remove the second element of the array that corresponds to the `mathworks.metricchecks.SubSystemCount` metric.

```
removeThreshold(TC,A(2))
```

The `slmetric.ThresholdConfiguration` object now contains one `slmetric.config.Threshold` object corresponding to the `mathworks.metricchecks.SubSystemCount` metric.

```
getThresholds(TC)
```

```
ans =

  Threshold with properties:

     MetricID: 'mathworks.metrics.SimulinkBlockCount'
    AppliesTo: 'Value'
```

Version History

Introduced in R2018b

See Also

`slmetric.config.ThresholdConfiguration` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.config.getActiveConfiguration

Package: `slmetric.config`

Obtain file path and name of XML file containing active Metrics Dashboard custom configuration

Syntax

```
Path = slmetric.config.getActiveConfiguration
```

Description

`Path = slmetric.config.getActiveConfiguration` returns the file path and name of the active Metrics Dashboard custom configuration file.

Examples

Get Default Metrics Dashboard Configuration

At the MATLAB command line, enter this command to get the active Metrics Dashboard configuration:

```
slmetric.config.getActiveConfiguration();
```

Output Arguments

Path — File path to XML file

character vector | string scalar

Full file path to folder containing XML file, which contains the active Metrics Dashboard custom configuration.

Note Passing an empty string to the `slmetric.config.setActionConfiguration` function (that is, `slmetric.config.setActiveConfiguration('')`), resets the configuration to the default, which is the shipping configuration. If you then enter the `slmetric.config.getActiveConfiguration` method, the method returns an empty array.

Data Types: `char`

Version History

Introduced in R2018b

See Also

`slmetric.config.Configuration` | `slmetric.config.setActiveConfiguration`

External Websites

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.config.setActiveConfiguration

Package: `slmetric.config`

Activate custom configuration for metric engine to use

Syntax

```
slmetric.config.setActiveConfiguration(fullfile)
```

Description

`slmetric.config.setActiveConfiguration(fullfile)` sets the custom configuration as the default configuration. When you collect metric data that uses the Metrics Dashboard or the `slmetric.Engine` `execute` method, the metric engine uses this custom configuration.

Note Passing an empty string to this function (that is, `slmetric.config.setActiveConfiguration('')`), resets the configuration to the default, which is shipping configuration.

Examples

Activate Custom Configuration

To set the active metric configuration, at the MATLAB command line:

```
slmetric.config.setActiveConfiguration('C:\temp\MyConfig.xml');
```

Input Arguments

`fullfile` — File path to XML file

character vector | string scalar

Full file path to folder containing the XML file, which contains Metrics Dashboard custom configurations.

Example: 'C:\temp\MyConfig.xml'

Data Types: `char`

Version History

Introduced in R2018b

See Also

`slmetric.config.Configuration` | `slmetric.config.getActiveConfiguration`

Topics
“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

metric.Engine

Collect metric data

Description

Use a `metric.Engine` object represents the metric engine that you can execute with the `execute` object function to collect metric data. Use `getMetrics` to access the metric data and return an array of `metric.Result` objects. Use the metric data to assess the status and quality of your design. Use the model testing metrics to analyze testing artifacts such as requirements, test results, and coverage results. Use the model maintainability metrics to analyze the maintainability and complexity of the design. For additional metrics, see “Design Cost Model Metrics” (Fixed-Point Designer).

Creation

Syntax

```
metric_engine = metric.Engine()
metric_engine = metric.Engine(projectPath)
```

Description

`metric_engine = metric.Engine()` creates a metric engine object that collects metric data on the current project.

`metric_engine = metric.Engine(projectPath)` opens the project `projectPath` and creates a metric engine object that collects metric data on the project.

Input Arguments

`projectPath` — Path of project
character vector | string scalar

Path of the project for which you want to collect metric data, specified as a character vector or string scalar.

Properties

`ProjectPath` — Project for metric collection
string scalar

This property is read-only.

Project for which the engine collects metric data, returned as a string.

Object Functions

execute	Collect metric data

generateReport	Generate report file containing metric results
getArtifactErrors	Return errors that occurred during artifact tracing
getAvailableMetricIds	Return metric identifiers for available metrics
getMetrics	Access metric data
openArtifact	Open traced artifact from metric result
updateArtifacts	Update trace information for pending artifact changes in the project

Examples

Collect Metric Data on Design Artifacts in a Project

Use a `metric.Engine` object to collect metric data on the design artifacts in a project.

Open the project. At the command line, type `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Collect results for the metric `"slcomp.OverallCyclomaticComplexity"` by executing the metric engine. For more information on the metric, see “Model Maintainability Metrics” on page 6-2.

```
execute(metric_engine,'slcomp.OverallCyclomaticComplexity');
```

Use the function `getMetrics` to access the results.

```
results = getMetrics(metric_engine,'slcomp.OverallCyclomaticComplexity');
for n = 1:length(results)
    disp(['Model: ',results(n).Scope.Name])
    disp(['  Overall Design Cyclomatic Complexity: ',num2str(results(n).Value)])
end
```

```
Model: db_Controller
  Overall Design Cyclomatic Complexity: 1
Model: db_LightControl
  Overall Design Cyclomatic Complexity: 4
Model: db_ThrottleController
  Overall Design Cyclomatic Complexity: 4
Model: db_ControlMode
  Overall Design Cyclomatic Complexity: 22
Model: db_DriverSwRequest
  Overall Design Cyclomatic Complexity: 9
```

For more information on how to collect metrics for design artifacts, see “Collect Model Maintainability Metrics Programmatically”.

Collect Metric Data on Testing Artifacts in a Project

Use a `metric.Engine` object to collect metric data on the requirements-based testing artifacts in a project.

Open the project. At the command line, type `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to ensure that the artifact information is up to date.

```
updateArtifacts(metric_engine)
```

Collect results for the metric `RequirementsPerTestCase` by executing the metric engine.

```
execute(metric_engine,{'RequirementsPerTestCase'});
```

Use the function `getMetrics` to access the results.

```
results = getMetrics(metric_engine,'RequirementsPerTestCase');
for n = 1:length(results)
    disp(['Test Case: ',results(n).Artifacts(1).Name])
    disp(['  Number of Requirements: ',num2str(results(n).Value)])
end
```

```
Test Case: Set button
  Number of Requirements: 0
Test Case: Decrement button hold
  Number of Requirements: 1
Test Case: Resume button
  Number of Requirements: 1
Test Case: Cancel button
  Number of Requirements: 1
Test Case: Decrement button short
  Number of Requirements: 2
Test Case: Increment button hold
  Number of Requirements: 1
Test Case: Increment button short
  Number of Requirements: 2
Test Case: Enable button
  Number of Requirements: 1
```

The results show that the test case `Set button` is missing links to requirements. To fix this, you would link the test case to the requirement that it verifies.

Version History

Introduced in R2020b

See Also

`metric.Result`

Topics

“Collect Model Maintainability Metrics Programmatically”
“Collect Metrics on Model Testing Artifacts Programmatically”

execute

Package: `metric`

Collect metric data

Syntax

```
execute(metricEngine,metricIDs)
execute(metricEngine,metricIDs,'ArtifactScope',scope)
```

Description

`execute(metricEngine,metricIDs)` collects results in the specified `metric.Engine` object for the metrics that you specify in `metricIDs`.

Note that there is also a function `execute` in the Fixed-Point Designer™ documentation.

`execute(metricEngine,metricIDs,'ArtifactScope',scope)` collects metric results for the artifacts in the `scope` that you specify. For example, you can specify the `scope` to be a unit in your project.

Examples

Collect Metrics for Design Artifacts in a Project

Collect metric data on the design artifacts in a project.

Open the project. At the command line, type `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Collect results for the metric `"slcomp.OverallCyclomaticComplexity"` by executing the metric engine. For more information on the metric, see “Model Maintainability Metrics” on page 6-2.

```
execute(metric_engine,'slcomp.OverallCyclomaticComplexity');
```

Use the function `getMetrics` to access the results.

```
results = getMetrics(metric_engine,'slcomp.OverallCyclomaticComplexity');
for n = 1:length(results)
    disp(['Model: ',results(n).Scope.Name])
    disp(['  Overall Design Cyclomatic Complexity: ',num2str(results(n).Value)])
end
```

```
Model: db_Controller
  Overall Design Cyclomatic Complexity: 1
Model: db_LightControl
```

```
  Overall Design Cyclomatic Complexity: 4
Model: db_ThrottleController
  Overall Design Cyclomatic Complexity: 4
Model: db_ControlMode
  Overall Design Cyclomatic Complexity: 22
Model: db_DriverSwRequest
  Overall Design Cyclomatic Complexity: 9
```

For more information on how to collect metrics for design artifacts, see “Collect Model Maintainability Metrics Programmatically”.

Collect Metrics for Testing Artifacts in a Project

Collect metric data on the requirements-based testing artifacts in a project.

Open the project. At the command line, type `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to ensure that the artifact information is up to date.

```
updateArtifacts(metric_engine)
```

Collect results for the metric `'RequirementsPerTestCase'` by executing the metric engine. For more information on the metric, see “Model Testing Metrics” on page 5-2.

```
execute(metric_engine,'RequirementsPerTestCase');
```

Use the function `getMetrics` to access the results.

```
results = getMetrics(metric_engine,'RequirementsPerTestCase');
for n = 1:length(results)
    disp(['Test Case: ',results(n).Artifacts(1).Name])
    disp(['  Number of Requirements: ',num2str(results(n).Value)])
end
```

Collect Metrics for a Unit

Collect metrics for one unit in the project. Specify a model and collect metrics for only the artifacts that trace to the model.

Open the project that contains the model. At the command line, type `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to ensure that the artifact information is up to date.

```
updateArtifacts(metric_engine)
```

Create a variable that represents the path to the model `db_DriverSwRequest`.

```
modelPath = fullfile(pwd, 'models', 'db_DriverSwRequest.slx');
```

Collect results for the metric `'RequirementsPerTestCase'` by using the `execute` function on the engine object and limiting the scope to the `db_DriverSwRequest` model.

```
execute(metric_engine,'RequirementsPerTestCase',...
'ArtifactScope',{modelPath,'db_DriverSwRequest'});
```

Use the function `getMetrics` to access the results.

```
results = getMetrics(metric_engine,'RequirementsPerTestCase');
for n = 1:length(results)
    disp(['Test Case: ',results(n).Artifacts(1).Name])
    disp(['  Number of Requirements: ',num2str(results(n).Value)])
end
```

Input Arguments

`metricEngine` — Metric engine object

`metric.Engine` object

Metric engine object for which you want to collect metric results, specified as a `metric.Engine` object.

`metricIDs` — Metric identifiers

character vector | cell array of character vectors | string | string array

Metric identifiers for metrics that you want to collect, specified as a character vector, cell array of character vectors, string, or string array.

You can use the function `getAvailableMetricIds` to return a list of available metric identifiers or see the following lists of metric identifiers:

- For the model maintainability metrics, see “Model Maintainability Metrics” on page 6-2. You can use the model maintainability metrics to analyze the size, architecture, and complexity of the MATLAB, Simulink, and Stateflow® artifacts in your project.
- For the model testing metrics, see “Model Testing Metrics” on page 5-2. You can use the model testing metrics to assess the traceability and completeness of the models, requirements, tests, and test results in your project. Collecting results for model testing metrics requires a Simulink Test™ license, a Requirements Toolbox™ license, or a Simulink Coverage™ license.

Example: `'slcomp.OverallCyclomaticComplexity'`

Example: `{'slcomp.OverallMATLABeLOC', 'slcomp.OverallSignalLines'}`

Example: `'TestCasesPerRequirementDistribution'`

Example: `{'TestCaseStatus', 'DecisionCoverageBreakdown'}`

`scope` — Path and identifier of project file

cell array of character vectors | string array

Path and identifier of the project file for which you want to collect metric results, specified as a cell array of character vectors or a string array. The first entry is the full path to a project file and the second entry is the identifier of the object inside the project file.

For a unit model, the first entry is the full path to the model file and the second entry is the name of the block diagram. When you use this argument, the metric engine collects the results for the artifacts that trace to specified project file.

Example: `{'C:\work\MyModel.slx', 'MyModel'}`

Alternative Functionality

App

You can also collect metric data by using the dashboard user interface.

- For the model maintainability metrics, use the **Model Maintainability Dashboard**. For more information, see “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard”.
- For the model testing metrics, use the **Model Testing Dashboard**. For more information, see “Explore Status and Quality of Testing Activities Using Model Testing Dashboard”.

Version History

Introduced in R2020b

See Also

`metric.Engine` | `generateReport` | `getArtifactErrors` | `getAvailableMetricIds` | `getMetrics` | `openArtifact` | `updateArtifacts`

Topics

“Collect Model Maintainability Metrics Programmatically”
“Collect Metrics on Model Testing Artifacts Programmatically”

generateReport

Package: `metric`

Generate report file containing metric results

Syntax

```
reportFile = generateReport(metricEngine)
reportFile = generateReport(metricEngine,'App','DashboardApp','Dashboard',
dashboardIdentifier)
reportFile = generateReport( ___ ,Name,Value)
```

Description

`reportFile = generateReport(metricEngine)` creates a PDF report in the root folder of the project for the metric results in the Model Testing Dashboard for the metric engine, `metricEngine`. Before you generate the report, collect metric results for the metric engine by using the `execute` function.

`reportFile = generateReport(metricEngine,'App','DashboardApp','Dashboard', dashboardIdentifier)` creates a PDF report in the root folder of the project of the metric results in the `dashboardIdentifier` dashboard for the metric engine, `metricEngine`. Before you generate the report, collect metric results for the engine by using the `execute` function.

Note that there is also a function `generateReport` in the Fixed-Point Designer documentation.

`reportFile = generateReport( ___ ,Name,Value)` specifies options using one or more name-value arguments. For example, `'Type','html-file'` generates an HTML report.

Examples

Generate a Model Testing Report

Analyze the testing artifacts in a project and generate a report file that contains the results.

Open the project that you want to analyze. For this example, open an example project by using the command `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to ensure that the artifact information is up to date.

```
updateArtifacts(metric_engine)
```

Create a list of the available metric identifiers for the Model Testing Dashboard by specifying the dashboard version as `'ModelUnitTesting'`.

```
metric_ids = getAvailableMetricIds(metric_engine,...
'App','DashboardApp',...
'Dashboard','ModelUnitTesting');
```

Collect the results by executing the metric engine on the list of metric identifiers.

```
execute(metric_engine, metric_ids);
```

Generate a PDF report of the model testing results in the root folder of the project.

```
generateReport(metric_engine,'App','DashboardApp',...
'Dashboard','ModelUnitTesting');
```

The report opens automatically. To prevent the report from opening automatically, specify `'LaunchReport'` as `false` when you call the `generateReport` function.

For each unit in the report, there is an artifact summary table that displays the number of artifacts in the requirements, design, and tests.

Generate a Model Maintainability Report

Analyze the maintainability of artifacts in a project and generate a report file that contains the results.

Open the project that you want to analyze. For this example, open an example project by using the command `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Create a list of the metric identifiers for the Model Maintainability Dashboard by specifying the dashboard version as `'ModelMaintainability'`.

```
metric_ids = getAvailableMetricIds(metric_engine,...
'App','DashboardApp',...
'Dashboard','ModelMaintainability');
```

Collect the results by executing the metric engine on the list of metric identifiers.

```
execute(metric_engine, metric_ids);
```

Generate an HTML report named `maintainabilityResults` in the current directory, which is the root folder of the project.

```
reportLocation = fullfile(pwd, 'maintainabilityResults.html');
generateReport(metric_engine,'App','DashboardApp',...
'Dashboard','ModelMaintainability',...
'Type','html-file','Location',reportLocation);
```

The report opens automatically. To prevent the report from opening automatically, specify `'LaunchReport'` as `false` when you call the `generateReport` function.

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.

Input Arguments

`metricEngine` — Metric engine object

`metric.Engine` object

Metric engine object for which you collected metric results, specified as a `metric.Engine` object.

`dashboardIdentifier` — Dashboard identifier

`'ModelUnitTesting'` (default) | `'ModelMaintainability'`

Identifier for the dashboard, specified as either:

- `'ModelUnitTesting'` for the Model Testing Dashboard
- `'ModelMaintainability'` for the Model Maintainability Dashboard

Example: `'ModelMaintainability'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Type','html-file'`

`LaunchReport` — Option to open generated report

`true` (default) | `false`

Option to automatically open the generated report, specified as `true` or `false`.

Example: `false`

Data Types: `logical`

`Location` — Full file name

character vector | string scalar

Full file name for the generated report, specified as a character vector or string scalar. Use the location to specify the name of the report.

By default, the report name is the `dashboardIdentifier`, followed by an underscore, followed by the project name, and the report is generated in the root folder of the project.

Example: `'C:\MyProject\Reports\RBTResults.html'`

`Type` — File type

`'pdf'` (default) | `'html-file'`

File type for the generated report, specified as `'pdf'` or `'html-file'`.

Example: `'html-file'`

Output Arguments

`reportFile` — Full file name of report

character vector

Full file name of the generated report, returned as a character vector.

Alternative Functionality

App

You can use the dashboard user interface to generate a report.

To open the dashboard user interface, use one of these approaches:

- In the Command Window, enter:

  ```
  modelDesignDashboard
  ```

  The dashboard opens the **Model Maintainability Dashboard**.
- In the Command Window, enter:

  ```
  modelTestingDashboard
  ```

  The dashboard opens the **Model Testing Dashboard**.

Click the **Report** button on the toolstrip. The Create Metric Result Report dialog opens.

Click **Create** to create a report.

For an example of how to use the dashboard user interface, see “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard”.

Version History

Introduced in R2021a

See Also

`metric.Engine` | `execute` | `getArtifactErrors` | `getAvailableMetricIds` | `getMetrics` | `openArtifact` | `updateArtifacts`

Topics

“Collect Model Maintainability Metrics Programmatically”
“Collect Metrics on Model Testing Artifacts Programmatically”
“Collect Requirements-Based Testing Metrics Using Continuous Integration”

getArtifactErrors

Package: `metric`

Return errors that occurred during artifact tracing

Syntax

```
errors = getArtifactErrors(metricEngine)
```

Description

`errors = getArtifactErrors(metricEngine)` returns the errors that occurred when the metric engine analyzed the artifacts. When you collect metric results by using the `execute` function, the engine object does not collect results for artifacts that return errors during analysis.

Note that there is also a function `getArtifactErrors` in the Fixed-Point Designer documentation.

Examples

Check for Artifact Errors After Collecting Metric Results

Collect metrics for the testing artifacts in a project. Then, check if artifacts returned errors and were not analyzed.

Open the project. At the command line, type `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to ensure that the artifact information is up to date.

```
updateArtifacts(metric_engine)
```

Collect results for the metric `'RequirementsPerTestCase'` by using the `execute` function on the `metric.Engine` object.

```
execute(metric_engine,'RequirementsPerTestCase');
```

Access the errors that occurred during analysis.

```
getArtifactErrors(metric_engine)
```

```
ans =

  0×0 empty struct array with fields:

    Address
```

```
    UUID
    ErrorId
    ErrorMessage
```

For this example, the artifacts did not return errors.

Input Arguments

`metricEngine` — Metric engine object

`metric.Engine` object

Metric engine object that you want to check for errors, specified as a `metric.Engine` object.

Output Arguments

`errors` — Artifact errors

`struct` array

Artifact errors that occurred when you executed the `metric.Engine` object, returned as an array of structures that correspond to the errors. The structure for an error contains these fields:

- `Address` — Address of the artifact that returned the error
- `UUID` — Unique identifier of the artifact
- `ErrorID` — Identifier of the error
- `ErrorMessage` — Description of the error

Alternative Functionality

App

You can view artifact errors in the **Errors** folder in the dashboard user interface. To see details about the warnings and errors that the dashboard finds during artifact analysis, at the bottom of the Dashboard window, click **Diagnostics**.

For more information, see “Artifact Errors”.

Version History

Introduced in R2020b

See Also

`metric.Engine` | `execute` | `generateReport` | `getAvailableMetricIds` | `getMetrics` | `openArtifact` | `updateArtifacts`

Topics

“Collect Model Maintainability Metrics Programmatically”
“Collect Metrics on Model Testing Artifacts Programmatically”

getAvailableMetricIds

Package: `metric`

Return metric identifiers for available metrics

Syntax

```
availableMetricIds = getAvailableMetricIds(metricEngine)
availableMetricIds = getAvailableMetricIds(
metricEngine,'App','DashboardApp','Dashboard',dashboardIdentifier)
availableMetricIds = getAvailableMetricIds( ___ ,'Installed',
installationStatus)
```

Description

`availableMetricIds = getAvailableMetricIds(metricEngine)` returns the metric identifiers for the metrics available for the specified `Metric.engine` object. By default, the list includes only the metrics available with the current installation.

`availableMetricIds = getAvailableMetricIds(
metricEngine,'App','DashboardApp','Dashboard',dashboardIdentifier)` returns the metric identifiers associated with the `dashboardIdentifier`.

For example, this code returns the metric identifiers for the **Model Testing Dashboard**:

```
availableMetricIds = getAvailableMetricIds(metricEngine,...
'App','DashboardApp',...
'Dashboard','ModelUnitTesting');
```

For an additional syntax to display metric identifiers for design cost estimation, see `getAvailableMetricIds`.

`availableMetricIds = getAvailableMetricIds(___ ,'Installed',
installationStatus)` returns the metric identifiers, filtered by the `installationStatus`.

For example, specifying `installationStatus` as `false` allows you to return the metric identifiers for each of the available metrics, even if the associated MathWorks® products are not currently installed on your machine.

Examples

Collect Metrics for Testing Artifacts in a Project

Collect metric data on the requirements-based testing artifacts in a project.

Open a project that includes the models and testing files. At the command prompt, type `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to ensure that the artifact information is up to date.

```
updateArtifacts(metric_engine)
```

Create a list of the available metric identifiers for the **Model Testing Dashboard** by specifying the dashboard identifier as `'ModelUnitTesting'`.

```
metric_ids = getAvailableMetricIds(metric_engine,...
'App','DashboardApp',...
'Dashboard','ModelUnitTesting');
```

Collect results by executing the metric engine on the list of metric identifiers.

```
execute(metric_engine,metric_ids);
```

Input Arguments

`metricEngine` — Metric engine object

`metric.Engine` object

Metric engine object for which you want to collect metric results, specified as a `metric.Engine` object.

`dashboardIdentifier` — Dashboard identifier

`'ModelUnitTesting'` | `'ModelMaintainability'`

Identifier for the dashboard, specified as one of these values:

- `'ModelUnitTesting'` — Return the model testing metric identifiers associated with your project.
- `'ModelMaintainability'` — Return each of the model maintainability metric identifiers.

Example: 'ModelUnitTesting'

`installationStatus` — Filter for metric installation status

`1` (`true`) (default) | `0` (`false`)

Filter for metric installation status, specified as one of these values:

- `1` (`true`) — Returns only metric identifiers associated with the MathWorks products currently installed on your machine.
- `0` (`false`) — Returns metric identifiers for each of the available metrics, even if the associated MathWorks products are not currently installed on your machine. You can use the list of each of the available metric identifiers to access the metric data collected on a different machine.

Example: `false`

Data Types: `logical`

Output Arguments

`availableMetricIds` — Metric identifiers
string | string array

Metric identifiers for the available metrics, returned as a string or string array. For a list of model testing metrics and their identifiers, see “Model Testing Metrics” on page 5-2.

Example: `"ConditionCoverageBreakdown"`

Example: `["ConditionCoverageBreakdown", "DecisionCoverageBreakdown", "ExecutionCoverageBreakdown", "MCDCCoverageBreakdown", "RequirementWithTestCaseDistribution", "RequirementWithTestCasePercentage", "RequirementsPerTestCaseDistribution", "TestCaseStatusDistribution", "TestCaseStatusPercentage", "TestCaseTagDistribution", "TestCaseTypeDistribution", "TestCaseVerificationStatusDistribution", "TestCaseWithRequirementDistribution", "TestCaseWithRequirementPercentage", "TestCasesPerRequirementDistribution"]`

Version History

Introduced in R2021b

Output includes metrics from the Model Testing Dashboard app and Design Cost Estimation app
Behavior changed in R2022a

The output includes metrics from both the Model Testing Dashboard app and the Design Cost Estimation app if you:

- Specify `'Installed'` as `false`.
- Have Fixed-Point Designer installed on your machine.

See Also

`metric.Engine` | `execute` | `getMetrics` | `generateReport` | `updateArtifacts`

Topics
“Collect Metrics on Model Testing Artifacts Programmatically”

getMetrics

Package: `metric`

Access metric data

Syntax

```
results = getMetrics(metricEngine,metricIDs)
results = getMetrics(metricEngine,metricIDs,'ArtifactScope',scope)
```

Description

`results = getMetrics(metricEngine,metricIDs)` returns metric results for the specified `metric.Engine` object for the metrics that you specify in `metricIDs`. To collect metric results for the `metric.Engine`, use the `execute` function. Then, access the results by using `getMetrics`.

Note that there is also a function `getMetrics` in the Fixed-Point Designer documentation.

`results = getMetrics(metricEngine,metricIDs,'ArtifactScope',scope)` returns metric results for the artifacts in the `scope` that you specify. For example, you can specify the `scope` to be a unit in your project.

Examples

Collect Metrics for Testing Artifacts in a Project

Collect metric data on the requirements-based testing artifacts in a project.

Open the project. At the command line, type `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to ensure that the artifact information is up to date.

```
updateArtifacts(metric_engine)
```

Collect results for the metric `'RequirementsPerTestCase'` by executing the metric engine.

```
execute(metric_engine,'RequirementsPerTestCase');
```

Access the metric results.

```
results = getMetrics(metric_engine,'RequirementsPerTestCase');
for n = 1:length(results)
    disp(['Test Case: ',results(n).Artifacts(1).Name])
```

```
    disp(['  Number of Requirements: ',num2str(results(n).Value)])
end
```

```
Test Case: Set button
  Number of Requirements: 0
Test Case: Decrement button hold
  Number of Requirements: 1
Test Case: Resume button
  Number of Requirements: 1
Test Case: Cancel button
  Number of Requirements: 1
Test Case: Decrement button short
  Number of Requirements: 2
Test Case: Increment button hold
  Number of Requirements: 1
Test Case: Increment button short
  Number of Requirements: 2
Test Case: Enable button
  Number of Requirements: 1
```

Collect Metrics for a Unit

Collect metrics for one unit in the project. Specify the unit and collect metrics for only the artifacts that trace to the model.

Open the project that contains the model. At the command line, type `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to ensure that the artifact information is up to date.

```
updateArtifacts(metric_engine)
```

Create a variable that represents the path to the unit model `db_DriverSwRequest`.

```
modelPath = fullfile(pwd, 'models', 'db_DriverSwRequest.slx');
```

Collect results for the metric `'RequirementsPerTestCase'` by using the `execute` function on the engine object and limiting the scope to the `db_DriverSwRequest` model.

```
execute(metric_engine,'RequirementsPerTestCase',...
'ArtifactScope',{modelPath, 'db_DriverSwRequest'});
```

Use the function `getMetrics` to access the results.

```
results = getMetrics(metric_engine,'RequirementsPerTestCase');
for n = 1:length(results)
    disp(['Test Case: ',results(n).Artifacts(1).Name])
    disp(['  Number of Requirements: ',num2str(results(n).Value)])
end
```

```
Test Case: Set button
  Number of Requirements: 0
```

```
Test Case: Resume button
  Number of Requirements: 1
Test Case: Decrement button short
  Number of Requirements: 2
Test Case: Enable button
  Number of Requirements: 1
Test Case: Increment button hold
  Number of Requirements: 1
Test Case: Increment button short
  Number of Requirements: 2
Test Case: Cancel button
  Number of Requirements: 1
Test Case: Decrement button hold
  Number of Requirements: 1
```

Input Arguments

`metricEngine` — Metric engine object

`metric.Engine` object

Metric engine object for which you want to access metric results, specified as a `metric.Engine` object.

`metricIDs` — Metric identifiers

character vector | cell array of character vectors | string | string array

Metric identifiers for metrics that you want to collect, specified as a character vector, cell array of character vectors, string, or string array.

- For the model maintainability metrics, see “Model Maintainability Metrics” on page 6-2.
- For the model testing metrics, see “Model Testing Metrics” on page 5-2.

Example: `'slcomp.OverallCyclomaticComplexity'`

Example: `{'slcomp.OverallMATLABeLOC', 'slcomp.OverallSignalLines'}`

Example: `'TestCasesPerRequirementDistribution'`

Example: `{'TestCaseStatus', 'DecisionCoverageBreakdown'}`

`scope` — Path and identifier of project file

cell array of character vectors | string array

Path and identifier of the project file for which you want to collect metric results, specified as a cell array of character vectors or a string array. The first entry is the full path to a project file and the second entry is the identifier of the object inside the project file.

For a unit model, the first entry is the full path to the model file and the second entry is the name of the block diagram. When you use this argument, the metric engine collects the results for the artifacts that trace to specified project file.

Example: `{'C:\work\MyModel.slx', 'MyModel'}`

Output Arguments

`results` — Metric results
array of `metric.Result` objects

Metric results, returned as an array of `metric.Result` objects.

Alternative Functionality

App

You can also view metric results by using the dashboard user interface.

- For the model maintainability metrics, use the **Model Maintainability Dashboard**. For more information, see “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard”.
- For the model testing metrics, use the **Model Testing Dashboard**. For more information, see “Explore Status and Quality of Testing Activities Using Model Testing Dashboard”.

Version History

Introduced in R2020b

See Also

`metric.Engine` | `execute` | `generateReport` | `getArtifactErrors` | `getAvailableMetricIds` | `openArtifact` | `updateArtifacts`

Topics
“Collect Model Maintainability Metrics Programmatically”
“Collect Metrics on Model Testing Artifacts Programmatically”

metric.Result

Metric data for specified metric algorithm

Description

A `metric.Result` object contains the metric data for a specified metric algorithm that traces to the specified unit or component.

Creation

Syntax

```
metric_result = metric.Result
```

Description

`metric_result = metric.Result` creates a handle to a metric result object.

Alternatively, if you collect results by executing a `metric.Engine` object, using the `getMetrics` function on the engine object returns the collected `metric.Result` objects in an array.

Properties

`MetricID` — Metric identifier
string

Metric identifier for the metric algorithm that calculated the results, returned as a string.

Example: `'TestCasesPerRequirementDistribution'`

`Artifacts` — Project artifacts
structure | array of structures

Project artifacts for which the metric is calculated, returned as a structure or an array of structures. For each artifact that the metric analyzed, the returned structure contains these fields:

- `UUID` — Unique identifier of the artifact.
- `Name` — Name of the artifact.
- `Type` — Type of artifact.
- `ParentUUID` — Unique identifier of the file that contains the artifact.
- `ParentName` — Name of the file that contains the artifact.
- `ParentType` — Type of file that contains the artifact.

`Value` — Result value
integer | string | double vector | structure

Value of the metric result for the specified algorithm and artifacts, returned as an integer, string, double vector, or structure. For a list of model testing metrics and their result values, see “Model Testing Metrics” on page 5-2.

`Scope` — Scope of metric results
structure

Scope of the metric results, returned as a structure. The scope is the unit or component for which the metric collected results. The structure contains these fields:

- `UUID` — Unique identifier of the unit or component.
- `Name` — Name of the unit or component.
- `Type` — Type of unit or component.
- `ParentUUID` — Unique identifier of the file that contains the unit or component.
- `ParentName` — Name of the file that contains the unit or component.
- `ParentType` — Type of file that contains the unit or component.

`UserData` — User data
string

User data provided by the metric algorithm, returned as a string.

Examples

Collect Metric Data on Design Artifacts in a Project

Use a `metric.Engine` object to collect metric data on the design artifacts in a project.

Open the project. At the command line, type `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Collect results for the metric `"slcomp.OverallCyclomaticComplexity"` by executing the metric engine. For more information on the metric, see “Model Maintainability Metrics” on page 6-2.

```
execute(metric_engine,'slcomp.OverallCyclomaticComplexity');
```

Use the function `getMetrics` to access the results. Assign the array of result objects to the `results` variable.

```
results = getMetrics(metric_engine,'slcomp.OverallCyclomaticComplexity');
```

Access the metric results data by using the properties of the `metric.Result` objects in the `results` array.

```
for n = 1:length(results)
    disp(['Model: ',results(n).Scope.Name])
    disp(['  Overall Design Cyclomatic Complexity: ',num2str(results(n).Value)])
end
```

```
Model: db_Controller
  Overall Design Cyclomatic Complexity: 1
Model: db_LightControl
  Overall Design Cyclomatic Complexity: 4
Model: db_ThrottleController
  Overall Design Cyclomatic Complexity: 4
Model: db_ControlMode
  Overall Design Cyclomatic Complexity: 22
Model: db_DriverSwRequest
  Overall Design Cyclomatic Complexity: 9
```

For more information on how to collect metrics for design artifacts, see “Collect Model Maintainability Metrics Programmatically”.

Collect Metric Data on Testing Artifacts in a Project

Collect metric data on the requirements-based testing artifacts in a project. Then, access the data by using the `metric.Result` objects.

Open the project. At the command line, type `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to ensure that the artifact information is up to date.

```
updateArtifacts(metric_engine)
```

Collect results for the metric `'RequirementsPerTestCase'` by using the `execute` function on the `metric.Engine` object.

```
execute(metric_engine,'RequirementsPerTestCase');
```

Use the function `getMetrics` to access the results. Assign the array of result objects to the `results` variable.

```
results = getMetrics(metric_engine,'RequirementsPerTestCase');
```

Access the metric results data by using the properties of the `metric.Result` objects in the array.

```
for n = 1:length(results)
    disp(['Test Case: ',results(n).Artifacts(1).Name])
    disp(['  Number of Requirements: ',num2str(results(n).Value)])
end
```

Version History

Introduced in R2020b

See Also

`metric.Engine` | `execute` | `generateReport` | `getMetrics`

Topics
“Collect Model Maintainability Metrics Programmatically”
“Collect Metrics on Model Testing Artifacts Programmatically”

modelDesignDashboard

Open Model Maintainability Dashboard

Syntax

```
modelDesignDashboard
modelDesignDashboard(projectPath)
```

Description

`modelDesignDashboard` opens the **Model Maintainability Dashboard** for the current project. The **Model Maintainability Dashboard** is a type of **Model Design Dashboard**.

`modelDesignDashboard(projectPath)` opens the **Model Maintainability Dashboard** for the project at the specified path.

Examples

Collect Metric Results in a Dashboard for the Current Project

Assess the maintainability and complexity of a project by using the **Model Maintainability Dashboard**.

For this example, you can open an example project. In the MATLAB Command Window, enter:

```
dashboardCCProjectStart
```

Open the **Model Maintainability Dashboard** for the project.

```
modelDesignDashboard
```

Open Model Maintainability Dashboard for a Project Path

Open the **Model Maintainability Dashboard** for a project at a path that you specify.

Assign the variable `projectPath` to the path to the root folder of the project. For this example, the path is `C:/projects/TestingProject/`.

```
projectPath = "C:/projects/TestingProject/";
```

Open the **Model Maintainability Dashboard** for the project at the path.

```
modelDesignDashboard(projectPath)
```

If you have previously opened the dashboard for the project, the dashboard populates with existing traceability and metric data. The dashboard collects metric data for the design artifacts in the project and displays the results in the widgets.

Input Arguments

`projectPath` — Path to project

character vector | string scalar

Path to the project file or project root folder for which you want to open the dashboard, specified as a character vector or string scalar.

Example: `"C:/projects/project1/myProject.prj"`

Example: `"C:/projects/project1/"`

Alternative Functionality

App

You can also open the dashboard for the current project by clicking **Model Design Dashboard** on the **Project** tab.

Version History

Introduced in R2022b

See Also

`metric.Engine`

Topics

“Monitor the Complexity of Your Design Using the Model Maintainability Dashboard”
“Collect Model Maintainability Metrics Programmatically”

modelTestingDashboard

Open Model Testing Dashboard

Syntax

```
modelTestingDashboard
modelTestingDashboard(projectPath)
```

Description

`modelTestingDashboard` opens the **Model Testing Dashboard** for the current project.

`modelTestingDashboard(projectPath)` opens the **Model Testing Dashboard** for the project at the specified path.

Examples

Collect Metric Results in a Dashboard for the Current Project

Assess the testing status of a project by using the **Model Testing Dashboard**.

For this example, you can open an example project. At the command line, type `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Open the **Model Testing Dashboard** for the project.

```
modelTestingDashboard
```

Open Model Testing Dashboard for a Project Path

Open the **Model Testing Dashboard** for a project at a path that you specify.

Assign the variable `projectPath` to the path to the root folder of the project. For this example, the path is `C:/projects/TestingProject/`.

```
projectPath = 'C:/projects/TestingProject/';
```

Open the **Model Testing Dashboard** for the project at the path.

```
modelTestingDashboard(projectPath)
```

If you have previously opened the dashboard for the project, the dashboard populates with existing traceability and metric data. The dashboard collects metric data for the testing artifacts in the project and displays the results in the widgets.

Input Arguments

`projectPath` — Path to project

character vector | string scalar

Path to the project file or project root folder for which you want to open the dashboard, specified as a character vector or string scalar.

Example: `'C:/projects/project1/myProject.prj'`

Example: `'C:/projects/project1/'`

Alternative Functionality

App

You can also open the dashboard for the current project by clicking **Model Testing Dashboard** on the **Project** tab.

Version History

Introduced in R2020b

See Also

`metric.Engine`

Topics

“Explore Status and Quality of Testing Activities Using Model Testing Dashboard”
“Collect Metrics on Model Testing Artifacts Programmatically”

openArtifact

Package: `metric`

Open traced artifact from metric result

Syntax

```
openArtifact(metricEngine,artifactID)
```

Description

`openArtifact(metricEngine,artifactID)` opens the artifact that has the specified identifier in the specified `metric.Engine` object. The editor that opens depends on the type of artifact, for example:

- Requirements open in the Requirements Editor.
- Test cases and test results open in the Test Manager.

Note that there is also a function `openArtifact` in the Fixed-Point Designer documentation.

Examples

Open Test Case Artifact from Metric Result

Collect metric data on the number of requirements linked to each test in a project. Then, open one of the test cases in the Test Manager.

Open the project. At the command line, type `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to ensure that the artifact information is up to date.

```
updateArtifacts(metric_engine)
```

Collect results for the metric `'RequirementsPerTestCase'` by executing the metric engine.

```
execute(metric_engine,'RequirementsPerTestCase');
```

Use the function `getMetrics` to access the results.

```
results = getMetrics(metric_engine,'RequirementsPerTestCase');
for n = 1:length(results)
    disp(['Test Case: ',results(n).Artifacts(1).Name])
    disp(['  Number of Requirements: ',num2str(results(n).Value)])
end
```

```
Test Case: Set button
  Number of Requirements: 0
Test Case: Decrement button hold
  Number of Requirements: 1
Test Case: Resume button
  Number of Requirements: 1
Test Case: Cancel button
  Number of Requirements: 1
Test Case: Decrement button short
  Number of Requirements: 2
Test Case: Increment button hold
  Number of Requirements: 1
Test Case: Increment button short
  Number of Requirements: 2
Test Case: Enable button
  Number of Requirements: 1
```

Open the first test case in the Test Manager by using the artifact identifier.

```
openArtifact(metric_engine,results(1).Artifacts(1).UUID)
```

Input Arguments

`metricEngine` — Metric engine object

`metric.Engine` object

Metric engine object for which you collected metric results, specified as a `metric.Engine` object.

`artifactID` — Artifact identifier

character vector | string scalar

Artifact identifier, specified as a character vector or string scalar. In a `metric.Result` object, the `Artifacts` field contains a structure for each artifact that the result traces to. To get the identifier for an artifact, use the `UUID` field of the structure for the artifact.

Version History

Introduced in R2020b

See Also

`metric.Engine` | `execute` | `generateReport` | `getArtifactErrors` | `getAvailableMetricIds` | `getMetrics` | `updateArtifacts`

Topics

“Collect Model Maintainability Metrics Programmatically”
“Collect Metrics on Model Testing Artifacts Programmatically”

updateArtifacts

Package: `metric`

Update trace information for pending artifact changes in the project

Syntax

```
updateArtifacts(metricEngine)
```

Description

`updateArtifacts(metricEngine)` updates the trace information for pending artifact changes in the metric data specified by `metricEngine` to ensure that artifacts are captured by the metrics. If an artifact was created, deleted, or modified since the last time you used `updateArtifacts`, running `updateArtifacts` performs traceability analysis and updates the trace information.

Note When you collect metrics programmatically, call `updateArtifacts` before running tests to ensure that the dashboard tracks the test results.

If the dashboard user interface is not used for a project, the dashboard does not track test results produced in Simulink Test Manager that have not been exported to a results file or been saved to a report.

Note that there is also a function `updateArtifacts` in the Fixed-Point Designer documentation.

Examples

Collect Metrics for Testing Artifacts in a Project

Collect metric data on the requirements-based testing artifacts in a project.

Open the project that includes the models and testing files. At the command prompt, type `dashboardCCProjectStart`.

```
dashboardCCProjectStart
```

Create a `metric.Engine` object for the project.

```
metric_engine = metric.Engine();
```

Update the trace information for `metric_engine` to ensure that the artifact information is up to date.

```
updateArtifacts(metric_engine)
```

Create a list of the available metric identifiers.

```
metric_ids = getAvailableMetricIds(metric_engine);
```

Collect results by executing the metric engine on the list of metric identifiers.

```
execute(metric_engine,metric_ids);
```

Input Arguments

`metricEngine` — Metric engine object
`metric.Engine` object

Metric engine object for which you want to collect metric results, specified as a `metric.Engine` object.

Version History

Introduced in R2021b

See Also

`metric.Engine` | `execute` | `getMetrics` | `generateReport` | `getAvailableMetricIds`

Topics
“Collect Metrics on Model Testing Artifacts Programmatically”

Advisor.component.Component class

Package: `Advisor.component`

Create component for metric analysis

Description

Model component used for metric analysis. When you define a custom model metric, the component object defines the component for metric analysis.

Construction

`component_obj = Advisor.component.Component` creates a model component object.

Properties

`ID` — Component ID
character vector

Component identifier. This property is read/write.

`Type` — Component type
enum

Component type, as specified by `Advisor.component.Types`. This property is read/write.

`Name` — Component name
character vector

Model component name. This property is read/write.

`IsLinked` — Specifies if the component is linked to a library
logical

`IsLinked` is `true` if the component is linked to a library. Components of type `Model`, `ModelBlock`, `ProtectedModel` cannot be linked. For these properties, the `IsLinked` is always `true`.

Methods

getPath	Retrieve component path

Version History

Introduced in R2016a

See Also

`Advisor.component.Types` | `slmetric.metric.Metric`

Topics
“Create a Custom Model Metric for Nonvirtual Block Count”
“Model Metrics” on page 2-299

getPath

Class: `Advisor.component.Component`
Package: `Advisor.component`

Retrieve component path

Syntax

```
path = getPath(component)
```

Description

`path = getPath(component)` retrieves the path to the component.

Input Arguments

`component` — Component
`Advisor.component.Component` model object

Constructed `Advisor.component.Component` model object.

Output Arguments

`path` — Model component path
character vector

Model component path, specified as a character vector.

Version History

Introduced in R2016a

See Also

`Advisor.component.Types`

Advisor.component.Types class

Package: `Advisor.component`

Create `enum` class specifying component type

Description

Create an enumeration `Advisor.component.Types` class to specify the model component type.

Construction

`enum_comp_type = Advisor.component.Types.Model` creates an enumeration of component type `Model`. The following table lists the component types.

Type	Description
`Model`	Simulink block diagram.
`LibraryBlock`	Library linked block.
`MFile`	MATLAB code file.
`ProtectedModel`	Protect Simulink block diagram.
`SubSystem`	Simulink subsystem block.
`Chart`	Stateflow chart or Stateflow block.
`MATLABFunction`	MATLAB function block.

Version History

Introduced in R2016a

See Also

`slmetric.metric.Metric` | `Advisor.component.Component`

Topics

“Create a Custom Model Metric for Nonvirtual Block Count”
“Model Metrics” on page 2-299

ModelAdvisor.Action class

Package: `ModelAdvisor`

Add actions to custom checks

Description

Instances of this class define actions you take when the Model Advisor checks do not pass. Users access actions by clicking the **Action** button that you define in the Model Advisor window.

Construction

ModelAdvisor.Action	Add actions to custom checks

Methods

setCallbackFcn	Specify action callback function

Properties

Description	Message in **Action** box
Name	Action button label

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

```
% define action (fix) operation
myAction = ModelAdvisor.Action;
myAction.Name='Fix block fonts';
myAction.Description=...
   'Click the button to update all blocks with specified font';
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

ModelAdvisor.Action

Class: `ModelAdvisor.Action`
Package: `ModelAdvisor`

Add actions to custom checks

Syntax

```
action_obj = ModelAdvisor.Action
```

Description

`action_obj = ModelAdvisor.Action` creates a handle to an action object.

Note

- Include an action definition in a check definition.
- Each check can contain only one action.

Examples

```
% define action (fix) operation
myAction = ModelAdvisor.Action;
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

ModelAdvisor.Check

Create custom Model Advisor checks

Description

The `ModelAdvisor.Check` class creates Model Advisor checks.

Creation

Syntax

```
check_obj = ModelAdvisor.Check(check_ID)
```

Description

`check_obj = ModelAdvisor.Check(check_ID)` creates a check object, `check_obj`, and assigns it a unique identifier, `check_ID`. To display checks in the Model Advisor tree, the checks must have an associated `ModelAdvisor.Task` or `ModelAdvisor.Root` object.

You can use one `ModelAdvisor.Check` object in multiple `ModelAdvisor.Task` objects, allowing you to place the same check in multiple locations in the Model Advisor tree. For example, **Check for implicit signal resolution** is displayed in the **By Product > Simulink** folder and in the **By Task > Model Referencing** folder in the Model Advisor tree.

When you use checks in task definitions, the following rules apply:

- If you define the properties of the check in the check definition and the task definition functions, the Model Advisor only displays the information contained in the task definition. For example, if you define the name of the check using the `ModelAdvisor.Task.DisplayName` property and using the `ModelAdvisor.Check.Title` property, the Model Advisor displays the information provided in `ModelAdvisor.Task.DisplayName`.
- If you define the properties of the check in the check definition but not the task definition, the task uses the properties from the check. For example, if you define the name of the check in the check definition function using the `ModelAdvisor.Check.Title` property, and you register the check using a task definition, the Model Advisor displays the information provided in `ModelAdvisor.Check.Title`.
- If you define the properties of the check in the task definition function but not the check definition function, the Model Advisor displays the information as long as you register the task with the Model Advisor instead of the check. For example, if you define the name of the check in the task definition using the `ModelAdvisor.Task.DisplayName` property instead of the `ModelAdvisor.Check.Title` property, and you register the check using a task definition, the Model Advisor displays the information provided in `ModelAdvisor.Task.DisplayName`.

Input Arguments

`check_ID` — Check ID for the custom Model Advisor check, specified as a character vector
character vector

Unique identifier for the custom Model Advisor check.

Properties

CallbackContext	Specify when to run check
CallbackHandle	Callback function handle for check
CallbackStyle	Callback function type
ErrorSeverity	Set severity of check failure
EmitInputParametersToReport	Display check input parameters in the Model Advisor report
Enable	Indicate whether user can enable or disable check
ID	Identifier for check
LicenseName	Product license names required to display and run check
Result	Results cell array
SupportExclusion	Set to support exclusions
SupportLibrary	Set to support library models
Title	Name of check
TitleTips	Description of check
Value	Status of check
Visible	Indicate to display or hide check
ResultDetails	Result details in a cell array

Object Functions

getID	Return check identifier
setAction	Specify action for check
setHelp	Set custom help for custom authored Model Advisor checks
setCallbackFcn	Specify callback function for check
setInputParameters	Specify input parameters for check
setInputParametersLayoutGrid	Specify layout grid for input parameters
setResultDetails	Associates result details with a check object

Version History

Introduced in R2008a

See Also

Topics

“Define Custom Model Advisor Checks”
“Customize the Configuration of the Model Advisor Overview”
“Create and Deploy a Model Advisor Custom Configuration”

ModelAdvisor.FactoryGroup class

Package: `ModelAdvisor`

Define subfolder in **By Task** folder

Description

The `ModelAdvisor.FactoryGroup` class defines a new subfolder to add to the **By Task** folder.

Construction

ModelAdvisor.FactoryGroup	Define subfolder in **By Task** folder

Methods

addCheck	Add check to folder

Properties

Description	Description of folder
DisplayName	Name of folder
ID	Identifier for folder
MAObj	Model Advisor object

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

```
% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

ModelAdvisor.FactoryGroup

Class: `ModelAdvisor.FactoryGroup`
Package: `ModelAdvisor`

Define subfolder in **By Task** folder

Syntax

```
fg_obj = ModelAdvisor.FactoryGroup(fg_ID)
```

Description

`fg_obj = ModelAdvisor.FactoryGroup(fg_ID)` creates a handle to a factory group object, `fg_obj`, and assigns it a unique identifier, `fg_ID`. `fg_ID` must remain constant.

Examples

```
% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
```

See Also

Topics
“Customize the Configuration of the Model Advisor Overview”
“Programmatically Customize Tasks and Folders for the Model Advisor”
“Create Model Advisor Checks”

ModelAdvisor.FormatTemplate

Template for formatting Model Advisor analysis results

Description

Use the `ModelAdvisor.FormatTemplate` class to format the result of a check in the analysis result pane of the Model Advisor for a uniform look and feel among the checks you create. You can format the analysis results as a table or a list.

Creation

Syntax

```
obj = ModelAdvisor.FormatTemplate(type)
```

Description

`obj = ModelAdvisor.FormatTemplate(type)` creates an object of the `ModelAdvisor.FormatTemplate` class. *`type`* is a character vector identifying the format type of the template, either list or table.

You must return the result object to the Model Advisor to display the formatted result in the analysis result pane.

Note Use the `ModelAdvisor.FormatTemplate` class in check callbacks.

Input Arguments

`type` — Template type
`ListTemplate` | `TableTemplate`

Type of `ModelAdvisor.FormatTemplate`.

Object Functions

addRow	Add row to table in Model Advisor analysis results
setCheckText	Add description of check to result
setColTitles	Add column titles to table in Model Advisor analysis results
setInformation	Add description of subcheck to result
setListObj	Add list of hyperlinks to model objects
setRecAction	Add Recommended Action section and text
setRefLink	Add See Also section and links
setSubBar	Add line between subcheck results
setSubResultStatus	Add status to the check or subcheck result
setSubResultStatusText	Add text below status in result
setSubTitle	Add title for subcheck in result

setTableInfo	Add data to table
setTableTitle	Add title to table in Model Advisor analysis results

Examples

Format Model Advisor Results

1 The following `sl_customization` file contains code that creates two template objects, `ft1` and `ft2`, and uses them to format the result of running a check in a table and a list. The result identifies the blocks in the model.

```
function sl_customization(cm)

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

% register custom factory group
cm.addModelAdvisorTaskFcn(@defineModelAdvisorTasks);

end


% ------------------------------
% defines Model Advisor Checks
% ------------------------------
function defineModelAdvisorChecks

% Define and register a sample check
rec = ModelAdvisor.Check('mathworks.example.SampleDetailStyle');
rec.Title = 'Sample check for Model Advisor using the ModelAdvisor.FormatTemplate';
setCallbackFcn(rec, @SampleDetailStyleCallback,'None','DetailStyle');
rec.setReportStyle('ModelAdvisor.Report.DefaultStyle');

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

end


% ------------------------------
% defines Model Advisor Tasks
% ------------------------------
function defineModelAdvisorTasks
mdladvRoot = ModelAdvisor.Root;

% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
rec.DisplayName='My Group 1';
rec.Description='Demo Factory Group';
rec.addCheck('mathworks.example.SampleDetailStyle');
mdladvRoot.publish(rec); % publish inside By Group list

end


% ------------------------------
% Sample Check With Subchecks Callback Function
% ------------------------------
function [ResultDescription] = SampleDetailStyleCallback(system, CheckObj)

% Initialize variables
ElementResults = ModelAdvisor.ResultDetail.empty();

% Perform the check actions
allBlocks = find_system(system);
[ResultDescription] = getFormattedTemplate(allBlocks);
```

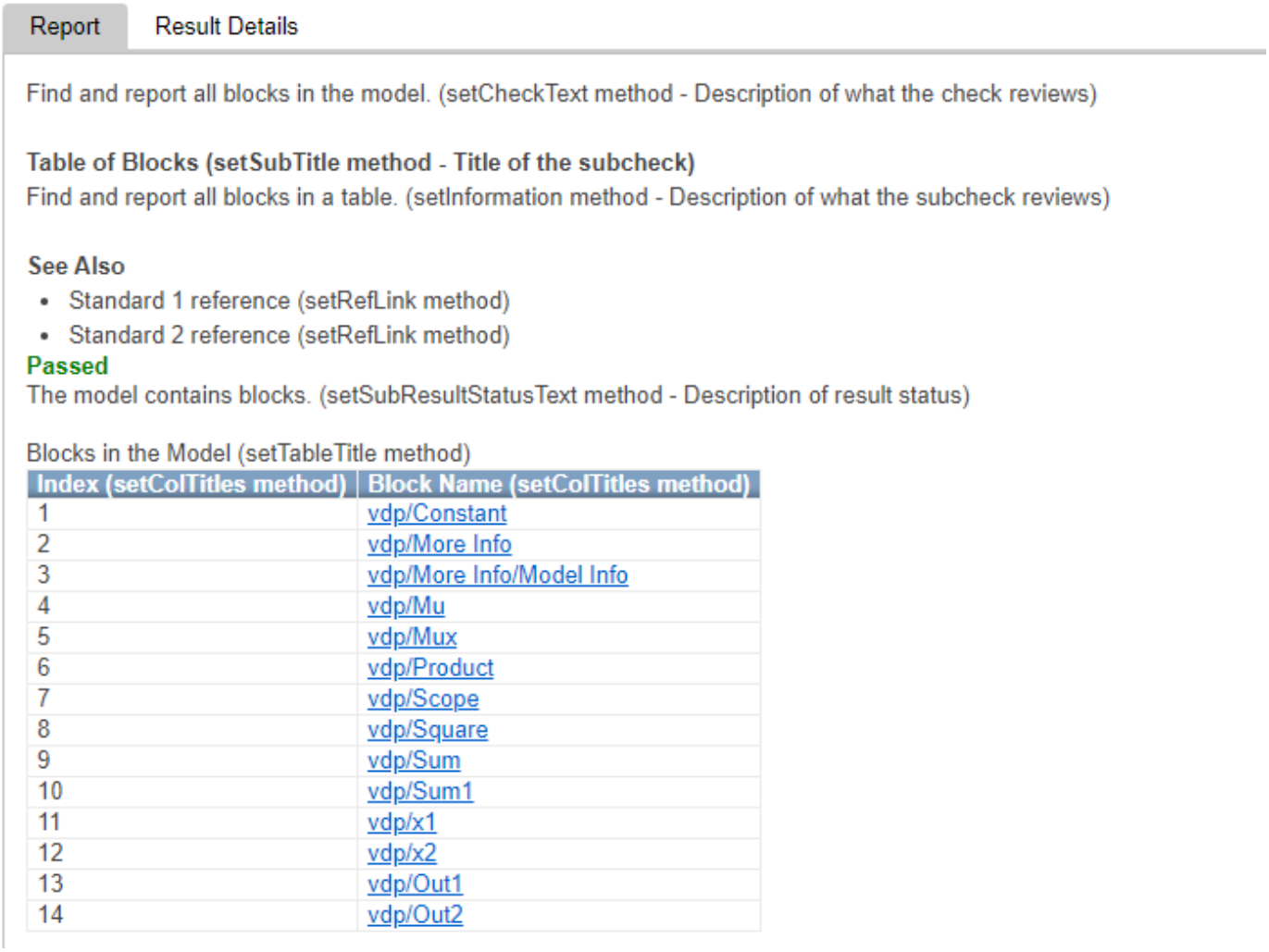
```
% Perform the subcheck actions - Result Details - Table
if length(allBlocks) == 1
    % Add result details for detailed style check
    ElementResults(end + 1) = ModelAdvisor.ResultDetail;
    ElementResults(end).IsViolation = true;
    ElementResults(end).Description =  ['Find and report all blocks in a table. '...
        '(setInformation method - Description of what the subcheck reviews)'];
    ElementResults(end).Status =  ['The model does not contain blocks. '...
        '(setSubResultStatusText method - Description of result status)'];
else
    for i=1:numel(allBlocks)
        ElementResults(end+1) = ModelAdvisor.ResultDetail;
        ElementResults(end).IsViolation = false;
        ElementResults(end).Format = 'Table';
        ModelAdvisor.ResultDetail.setData(ElementResults(end),'SID',allBlocks{i});
        ElementResults(end).Description =  ['Find and report all blocks in a table. '...
            '(setInformation method - Description of what the subcheck reviews)'];
        ElementResults(end).Status = ['The model contains blocks. '...
            '(setSubResultStatusText method - Description of result status)'];
    end
end

% Perform the subcheck actions  - Result Details - List
if length(allBlocks) == 1
     ElementResults(end+1) = ModelAdvisor.ResultDetail;
     ElementResults(end).IsViolation = true;
     ElementResults(end).Description =  ['Find and report all blocks in a table. '...
        '(setInformation method - Description of what the subcheck reviews)'];
     ElementResults(end).Status =  ['The model does not contain blocks. '...
        '(setSubResultStatusText method - Description of result status)'];
else
    for i= 1:numel(allBlocks)
        ElementResults(end+1) = ModelAdvisor.ResultDetail;
        ElementResults(end).IsViolation = false;
        ModelAdvisor.ResultDetail.setData(ElementResults(end),'SID',allBlocks{i});
        ElementResults(end).Description =  ['Find and report all blocks in a list. '...
            '(setInformation method - Description of what the subcheck reviews)'];
        ElementResults(end).Status = ['The model contains blocks. '...
            '(setSubResultStatusText method - Description of result status)'];
    end
end

%Set check result details
CheckObj.setResultDetails(ElementResults);

end

function [ResultDescription] = getFormattedTemplate(allBlocks)
ResultDescription={};

% Create FormatTemplate object for first subcheck, specify table format
ft1 = ModelAdvisor.FormatTemplate('TableTemplate');

% Add information describing the overall check
setCheckText(ft1, ['Find and report all blocks in the model. '...
    '(setCheckText method - Description of what the check reviews)']);

% Add information describing the subcheck
setSubTitle(ft1, 'Table of Blocks (setSubTitle method - Title of the subcheck)');
setInformation(ft1, ['Find and report all blocks in a table. '...
    '(setInformation method - Description of what the subcheck reviews)']);

% Add See Also section for references to standards
setRefLink(ft1, {{'Standard 1 reference (setRefLink method)'},
    {'Standard 2 reference (setRefLink method)'}});

% Add information to the table
setTableTitle(ft1, {'Blocks in the Model (setTableTitle method)'});
setColTitles(ft1, {'Index (setColTitles method)',
    'Block Name (setColTitles method)'});
```

```
if length(allBlocks) == 1
    % Add status for subcheck
    setSubResultStatus(ft1, 'Warn');
    setSubResultStatusText(ft1, ['The model does not contain blocks. '...
        '(setSubResultStatusText method - Description of result status)']);
    setRecAction(ft1, {'Add blocks to the model. '...
        '(setRecAction method - Description of how to fix the problem)'});
else
    % Add status for subcheck
    setSubResultStatus(ft1, 'Pass');
    setSubResultStatusText(ft1, ['The model contains blocks. '...
        '(setSubResultStatusText method - Description of result status)']);
    for inx = 2 : length(allBlocks)
        % Add information to the table
        addRow(ft1, {inx-1,allBlocks(inx)});
    end
end

% Pass table template object for subcheck to Model Advisor
ResultDescription{end+1} = ft1;

% Create FormatTemplate object for second subcheck, specify list format
ft2 = ModelAdvisor.FormatTemplate('ListTemplate');

% Add information describing the subcheck
setSubTitle(ft2, 'List of Blocks (setSubTitle method - Title of the subcheck)');
setInformation(ft2, ['Find and report all blocks in a list. '...
    '(setInformation method - Description of what the subcheck reviews)']);

% Add See Also section for references to standards
setRefLink(ft2, {{'Standard 1 reference (setRefLink method)'},
    {'Standard 2 reference (setRefLink method)'}});

% Last subcheck, suppress line
setSubBar(ft2, false);

% Perform the subcheck actions
if length(allBlocks) == 1
    % Add status for subcheck
    setSubResultStatus(ft2, 'Warn');
    setSubResultStatusText(ft2, ['The model does not contain blocks. '...
        '(setSubResultStatusText method - Description of result status)']);
    setRecAction(ft2, {'Add blocks to the model. '...
        '(setRecAction method - Description of how to fix the problem)'});

else
    % Add status for subcheck
    setSubResultStatus(ft2, 'Pass');
    setSubResultStatusText(ft2, ['The model contains blocks. '...
        '(setSubResultStatusText method - Description of result status)']);
    % Add information to the list
    setListObj(ft2, allBlocks);
end

% Pass list template object for the subcheck to Model Advisor
ResultDescription{end+1} = ft2;

end
```

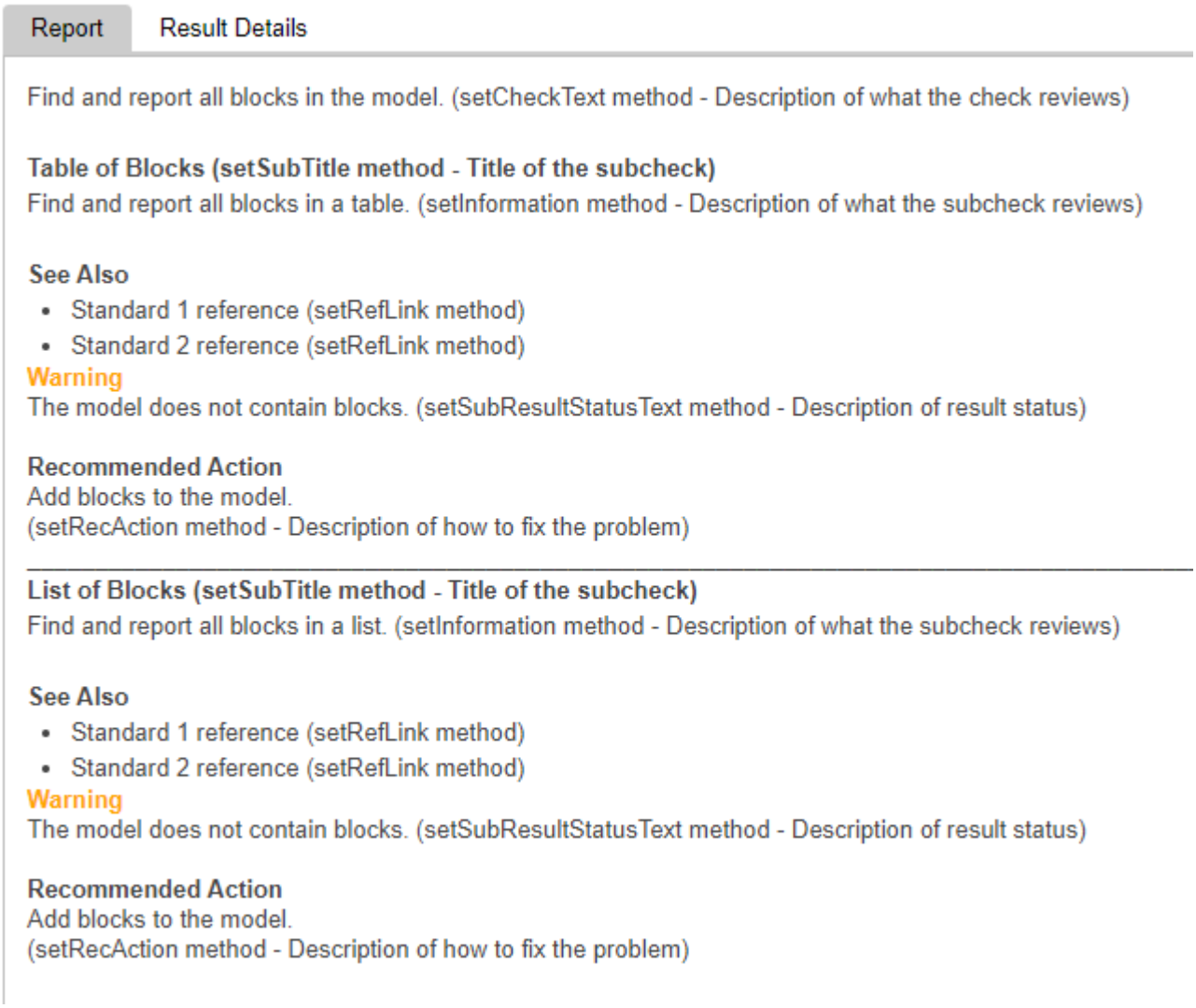
2. Save the `sl_customization` file to your working directory.
3. In the MATLAB command window, enter:

   ```
   Advisor.Manager.refresh_customizations
   ```
4. Open a model.
5. In the **Modeling** tab, select **Model Advisor**.
6. In the **By Task > My Group 1** folder, select **Sample check for Model Advisor using ModelAdvisor.FormatTemplate**.
7. Click **Run This Check**.

The following graphic displays the output as it appears in the Model Advisor when the check passes.



The following graphic displays the output as it appears in the Model Advisor when the check warns.



Alternatives

When you define a `ModelAdvisor.Check` object, for the `CallbackStyle` property, if you specify `DetailStyle`, you do not have to use the `ModelAdvisor.FormatTemplate` API or the other formatting APIs to the format results that appear in the Model Advisor report. `DetailStyle` also allows you to view results by block, subsystem, or recommended action.

If the default formatting does not meet your needs, use the `ModelAdvisor.FormatTemplate` API or the other formatting APIs. The `ModelAdvisor.FormatTemplate` class provides a uniform look and feel among the checks you create.

Version History

Introduced in R2009a

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

ModelAdvisor.Group class

Package: `ModelAdvisor`

Define custom folder

Description

The `ModelAdvisor.Group` class defines a folder that is displayed in the Model Advisor tree. Use folders to consolidate checks by functionality or usage.

Construction

ModelAdvisor.Group	Define custom folder

Methods

addGroup	Add subfolder to folder
addTask	Add task to folder

Properties

Description	Description of folder
DisplayName	Name of folder
ID	Identifier for folder
MAObj	Model Advisor object

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

ModelAdvisor.Group

Class: `ModelAdvisor.Group`
Package: `ModelAdvisor`

Define custom folder

Syntax

```
group_obj = ModelAdvisor.Group(group_ID)
```

Description

`group_obj = ModelAdvisor.Group(group_ID)` creates a handle to a group object, `group_obj`, and assigns it a unique identifier, `group_ID`. `group_ID` must remain constant.

Examples

```
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');
```

See Also

Topics
“Customize the Configuration of the Model Advisor Overview”
“Programmatically Customize Tasks and Folders for the Model Advisor”
“Create Model Advisor Checks”

ModelAdvisor.Image class

Package: `ModelAdvisor`

Include image in Model Advisor output

Description

The `ModelAdvisor.Image` class adds an image to the Model Advisor output.

Construction

ModelAdvisor.Image	Include image in Model Advisor output

Methods

setHyperlink	Specify hyperlink location
setImageSource	Specify image location

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

ModelAdvisor.Image

Class: `ModelAdvisor.Image`
Package: `ModelAdvisor`

Include image in Model Advisor output

Syntax

```
object = ModelAdvisor.Image
```

Description

`object = ModelAdvisor.Image` creates a handle to an image object, `object`, that the Model Advisor displays in the output. The Model Advisor supports many image formats, including, but not limited to, JPEG, BMP, and GIF.

Examples

```
image_obj = ModelAdvisor.Image;
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

ModelAdvisor.InputParameter class

Package: `ModelAdvisor`

Add input parameters to custom checks

Description

Instances of the `ModelAdvisor.InputParameter` class specify the input parameters a custom check uses in analyzing the model. Access input parameters in the Model Advisor window.

Construction

ModelAdvisor.InputParameter	Add input parameters to custom checks

Methods

setColSpan	Specify number of columns for input parameter
setRowSpan	Specify rows for input parameter

Properties

Description	Description of input parameter
Entries	Drop-down list entries
Name	Input parameter name
Type	Input parameter type
Value	Value of input parameter

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

ModelAdvisor.InputParameter

Class: `ModelAdvisor.InputParameter`
Package: `ModelAdvisor`

Add input parameters to custom checks

Syntax

```
input_param = ModelAdvisor.InputParameter
```

Description

`input_param = ModelAdvisor.InputParameter` creates a handle to an input parameter object, `input_param`.

Note You must include input parameter definitions in a check definition.

Examples

The following example is a fragment of code from a check definition function. The example does not execute as shown without the full check definition function.

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

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

ModelAdvisor.LineBreak

Insert line break

Description

Use instances of the `ModelAdvisor.LineBreak` class to insert line breaks in the Model Advisor outputs.

Examples

Insert Line Break

Add a line break between two lines of text:

```
result = ModelAdvisor.Paragraph;
addItem(result, [resultText1 ModelAdvisor.LineBreak resultText2]);
```

Version History

Introduced in R2006b

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

ModelAdvisor.List

Create list class

Description

Use instances of the `ModelAdvisor.List` class to create list-formatted outputs.

Creation

Syntax

`list = ModelAdvisor.List`

Description

`list = ModelAdvisor.List` creates a list object, `list`.

Object Functions

addItem Add list items to Model Advisor results
setType Specify Model Advisor list type

Example

Create Numbered and Bulleted Lists

You can create two types of lists: numbered and bulleted. The default list formatting is bulleted. Use the `ModelAdvisor.List` class to create and format lists. You can create lists with indented subsections, formatted as either numbered or bulleted.

```
subList = ModelAdvisor.List();
subList.setType('numbered')
subList.addItem(ModelAdvisor.Text('Sub entry 1', {'pass','bold'}));
subList.addItem(ModelAdvisor.Text('Sub entry 2', {'pass','bold'}));

topList = ModelAdvisor.List();
topList.addItem([ModelAdvisor.Text('Entry level 1',{'keyword','bold'}), subList]);
topList.addItem([ModelAdvisor.Text('Entry level 2',{'keyword','bold'}), subList]);
```

Version History

Introduced in R2006b

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

ModelAdvisor.ListViewParameter class

Package: `ModelAdvisor`

(To be removed) Add list view parameters to custom checks

Note `ModelAdvisor.ListViewParameter` will be removed.

Use hyperlinks in the Model Advisor results to view and modify model elements that are being flagged by the Model Advisor check instead. For more information on the recommended workflow for authoring custom checks, see “Fix a Model to Comply with Conditions that You Specify with the Model Advisor” and “Create and Deploy a Model Advisor Custom Configuration”.

Description

The Model Advisor uses list view parameters to populate the Model Advisor Result Explorer. Access the information in list views by clicking **Explore Result** in the Model Advisor window.

Construction

ModelAdvisor.ListViewParameter	Add list view parameters to custom checks

Properties

Attributes	Attributes to display in Model Advisor Report Explorer
Data	Objects in Model Advisor Result Explorer
Name	Drop-down list entry

Examples

The following example is a fragment of code from a check definition function. The example does not execute as shown without the full check definition function.

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

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

Version History

`ModelAdvisor.ListViewParameter` will be removed.
Not recommended starting in R2021b

Use hyperlinks in the Model Advisor results to view and modify model elements that are being flagged by the Model Advisor check. For more information on the recommended workflow for authoring custom checks, see “Fix a Model to Comply with Conditions that You Specify with the Model Advisor” and “Create and Deploy a Model Advisor Custom Configuration”.

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

ModelAdvisor.ListViewParameter

Class: `ModelAdvisor.ListViewParameter`
Package: `ModelAdvisor`

Add list view parameters to custom checks

Syntax

```
lv_param = ModelAdvisor.ListViewParameter
```

Description

`lv_param = ModelAdvisor.ListViewParameter` defines a list view, `lv_param`.

Note Include list view parameter definitions in a check definition.

See Also

"Customize the Configuration of the Model Advisor Overview"

Topics

"Create Model Advisor Checks"
"Customize the Configuration of the Model Advisor Overview"
`getListViewParameters`
`setListViewParameters`

ModelAdvisor.lookupCheckID

Package: `ModelAdvisor`

Look up current Model Advisor check ID for legacy Model Advisor check ID

Syntax

```
NewID = ModelAdvisor.lookupCheckID(OldCheckID)
```

Description

`NewID = ModelAdvisor.lookupCheckID(OldCheckID)` returns the current Model Advisor check ID, `NewID`, for the legacy Model Advisor check ID specified by `OldCheckID`.

Examples

Find the Current Model Advisor Check ID by Using a Legacy Check ID

Use the legacy check ID `"mathworks.iec61508.OutportRange"` to find the current Model Advisor check ID.

In release R2018a and earlier, the check ID for the Model Advisor check Check for root Outports with missing range definitions was `"mathworks.iec61508.OutportRange"`.

Use the `ModelAdvisor.lookupCheckID` function to return the current Model Advisor check ID for the check.

```
currentID = ModelAdvisor.lookupCheckID("mathworks.iec61508.OutportRange")
```

```
currentID =

    'mathworks.hism.hisl_0026'
```

Input Arguments

`OldCheckID` — Legacy identifier of Model Advisor check

character vector | string scalar

Legacy identifier of a Model Advisor check, specified as a character vector or string scalar.

Example: "mathworks.do178.himl_0002"

Data Types: `char` | `string`

Output Arguments

`NewID` — Current identifier of Model Advisor check

character vector

Current identifier of a Model Advisor check, returned as a character vector.

See Also

`ModelAdvisor.run`

Topics

“Archive and View Results”

ModelAdvisor.Paragraph class

Package: `ModelAdvisor`

Create and format paragraph

Description

The `ModelAdvisor.Paragraph` class creates and formats a paragraph object.

Construction

ModelAdvisor.Paragraph	Create and format paragraph

Methods

addItem	Add item to paragraph
setAlign	Specify paragraph alignment

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

```
% Check Simulation optimization setting
ResultDescription = ModelAdvisor.Paragraph(['Check Simulation '...
'optimization settings:']);
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

ModelAdvisor.Paragraph

Class: `ModelAdvisor.Paragraph`
Package: `ModelAdvisor`

Create and format paragraph

Syntax

```
para_obj = ModelAdvisor.Paragraph
```

Description

`para_obj = ModelAdvisor.Paragraph` defines a paragraph object `para_obj`.

Example

To change default formatting, use the `ModelAdvisor.Paragraph` class. The default paragraph formatting is:

- Empty
- Default color (black)
- Unformatted, (not bold, italicized, underlined, linked, subscripted, or superscripted)
- Aligned left

```
% Check Simulation optimization setting
ResultDescription = ModelAdvisor.Paragraph(['Check Simulation '...
'optimization settings:']);
```

You must handle paragraphs explicitly because most markup languages do not support line breaks.

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

ModelAdvisor.Procedure

Define custom procedures

Description

The `ModelAdvisor.Procedure` class defines a procedure that is displayed in the Model Advisor tree. Use procedures to organize additional procedures or checks by functionality or usage.

Creation

Syntax

```
ProcedureObject = ModelAdvisor.Procedure(ProcedureID)
```

Description

`ProcedureObject = ModelAdvisor.Procedure(ProcedureID)` creates a Model Advisor procedure object, `ProcedureObject` and assigns it a unique identifier, `ProcedureID` which must remain constant.

Input Arguments

`ProcedureID` — Unique identifier
`string`

Unique identifier for the Model Advisor procedure.

Data Types: `char`

Properties

`Description` — Procedure details
`character vector`

Provides information about the procedure. The right pane of the Model Advisor displays the procedure details.

Data Types: `char`

`DisplayName` — Procedure name
`string`

Specifies the name of the procedure that the Model Advisor displays.

Data Types: `char`

`ID` — Unique identifier
`string`

Specifies a permanent, unique identifier for the procedure.

Note

- You must specify this field.
- The value of `ID` must remain constant.
- The Model Advisor generates an error if `ID` is not unique.
- Procedure definitions must refer to other procedures by `ID`.

Data Types: `char`

`MAObj` — Model Advisor object
`Simulink.ModelAdvisor` object

Specifies a handle to the current Model Advisor object.

Object Functions

addProcedure	Add subprocedure to procedure
addTask	Add task to procedure

Examples

Add Procedures to Model Advisor Procedure

1. Create a `ModelAdvisor.Procedure` object named `MAP`.

```
MAP = ModelAdvisor.Procedure('com.mathworks.sample.ProcedureSample');
```

2. Create three subprocedures.

```
MAP1=ModelAdvisor.Procedure('com.mathworks.sample.procedure1');
MAP2=ModelAdvisor.Procedure('com.mathworks.sample.procedure2');
MAP3=ModelAdvisor.Procedure('com.mathworks.sample.procedure3');
```

3. Add three procedures to `MAP`.

```
addProcedure(MAP, MAP1);
addProcedure(MAP, MAP2);
addProcedure(MAP, MAP3);
```

Version History

Introduced in R2006a

See Also

Topics

“Programmatically Create Procedural-Based Configurations”
“Customize the Configuration of the Model Advisor Overview”

ModelAdvisor.Root

Identify root node

Description

The `ModelAdvisor.Root` class returns the root object.

Creation

Syntax

```
root_obj = ModelAdvisor.Root
```

Description

`root_obj = ModelAdvisor.Root` creates a handle to the root object, `root_obj`.

Object Functions

publish	Publish object in Model Advisor root
register	Register object in Model Advisor root

Version History

Introduced in R2008a

See Also

Topics

“Customize the Configuration of the Model Advisor Overview”
“Create Model Advisor Checks”

ModelAdvisor.run

Use Model Advisor to run checks on systems

Syntax

```
ModelAdvisor.run(Systems,CheckIDList)
ModelAdvisor.run(Systems,'Configuration',FileName)

ModelAdvisor.run(Systems,{{CheckID,'InputParam',{paramName,paramValue}}})
ModelAdvisor.run(Systems,{{CheckID1,'InputParam',
{paramName1,paramValue1,...,paramNameN,paramValueN}},...,
{CheckIDN,'InputParam',{paramName1,paramValue1,...,paramNameN,paramValueN}}})

Results = ModelAdvisor.run( ___ ,Name,Value)
```

Description

`ModelAdvisor.run(Systems,CheckIDList)` runs the Model Advisor on the models or subsystems specified by `Systems` for the list of check IDs specified by `CheckIDList`.

`ModelAdvisor.run(Systems,'Configuration',FileName)` runs the Model Advisor for the list of checks specified by the Model Advisor configuration file `FileName`.

`ModelAdvisor.run(Systems,{{CheckID,'InputParam',{paramName,paramValue}}})` runs the Model Advisor for the check ID specified by `CheckID` with the input parameter setting specified by the input parameter name `paramName` and the input parameter value `paramValue`.

`ModelAdvisor.run(Systems,{{CheckID1,'InputParam',`
`{paramName1,paramValue1,...,paramNameN,paramValueN}},...,`
`{CheckIDN,'InputParam',{paramName1,paramValue1,...,paramNameN,paramValueN}}})`
runs the Model Advisor for one or more check IDs with the input parameter settings specified using `'InputParam'`.

The check IDs are specified by `CheckID1` through `CheckIDN`. For each check ID, you can specify the input parameters for the check by using `'InputParam'` with the parameter names, `paramName1` through `paramNameN`, and parameter values, `paramValue1` through `paramValueN`.

For example, the following code runs the Model Advisor checks:

- `'mathworks.jmaab.jc_0281'` with the input parameter `'Follow links'` set to `'off'` and the `'jc_0281_a'` sub-checks disabled
- `'mathworks.jmaab.db_0032'` with the input parameter `'Follow links'` set to `'off'` and the input parameter `'Look under masks'` set to `'all'`

```
Results = ModelAdvisor.run('vdp',...
{{'mathworks.jmaab.jc_0281','InputParam',{'Follow links','off','jc_0281_a',0}},...
{'mathworks.jmaab.db_0032','InputParam',{'Follow links','off','Look under masks','all'}}})
```

`Results = ModelAdvisor.run( ___ ,Name,Value)` specifies the properties of the Model Advisor analysis using one or more `Name,Value` pair arguments and returns the results in `Results`. Use

`Results` to view the properties of the Model Advisor run. Use this option with one of the previous syntaxes.

Examples

Use the Model Advisor to Run a List of Checks

Create a list of check IDs and use `ModelAdvisor.run` to run the checks on the specified subsystems.

Create a list of the check IDs for the checks “Check model diagnostic parameters” on page 2-135 and “Check for unconnected objects” on page 2-107.

```
checkIDs = {'mathworks.maab.jc_0021',...
'mathworks.iec61508.UnconnectedObjects'};
```

Open the example model `sldemo_auto_climatecontrol`.

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

Create a list of the subsystems `sldemo_auto_climatecontrol/Heater Control` and `sldemo_auto_climatecontrol/AC Control`.

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

Use Model Advisor to run the checks on the subsystems.

```
results = ModelAdvisor.run(systems,checkIDs);
```

For more information on Model Advisor checks, see “Model Advisor Checks Documentation”. For details on how to find check IDs, see “Find Model Advisor Check IDs”.

Use the Model Advisor to Run Sub-Checks

Create a list of checks that specify the input parameter values, then use the Model Advisor to run the checks.

Open the model `vdp`.

```
open_system('vdp')
```

Create a list of the input parameters used by the checks “Check signal line connections” on page 2-148 and “Check trigger signal names” on page 2-155.

To find the input parameters for the checks, create a Model Advisor object for the model and use `getInputParameters`. The check ID for the check “Check signal line connections” on page 2-148 is `'mathworks.jmaab.db_0032'` and the check ID for the check “Check trigger signal names” on page 2-155 is `'mathworks.jmaab.jc_0281'`.

```
ma = Simulink.ModelAdvisor.getModelAdvisor('vdp');

db_0032_parameters = getInputParameters(ma, 'mathworks.jmaab.db_0032');
jc_0281_parameters = getInputParameters(ma, 'mathworks.jmaab.jc_0281');
```

For the check `'mathworks.jmaab.db_0032'`, `db_0032_parameters{6}` contains the `InputParameter` properties for the input parameter **Follow links**. Save the input parameter name to the variable `followLinks`.

```
followLinks = db_0032_parameters{6}.Name;
```

For `'mathworks.jmaab.db_0032'`, set the input parameter **Follow links** to `'off'`. When **Follow links** is `'off'`, the Model Advisor does not analyze the content of library-linked blocks.

```
check1 = {'mathworks.jmaab.db_0032',...
'InputParam',{followLinks,'off'}};
```

For the check `'mathworks.jmaab.jc_0281'`, `jc_0281_parameters{1}` contains the `InputParameter` properties for the input parameter **jc_0281_a** and `jc_0281_parameters{2}` contains the `InputParameter` properties for the input parameter **jc_0281_b**. **jc_0281_a** and **jc_0281_b** contain sub-checks for `'mathworks.jmaab.jc_0281'`. Save the input parameter names to the variables `subCheckA` and `subCheckB`.

```
subCheckA = jc_0281_parameters{1}.Name;
subCheckB = jc_0281_parameters{2}.Name;
```

Select the sub-check `'jc_0281_a2'` and disable the `'jc_0281_b'` sub-checks. The input parameter value `2` selects the second sub-check of `'jc_0281_a'`. The input parameter value `0` disables `'jc_0281_b'` sub-checks. For more information on `'mathworks.jmaab.jc_0281'` and its sub-checks, see “Check trigger signal names” on page 2-155.

```
check2 = {'mathworks.jmaab.jc_0281',...
'InputParam',{subCheckA, 2, subCheckB, 0}};
```

Create the list of checks.

```
listOfChecks = {check1, check2};
```

Use Model Advisor to run the specified checks on the system `vdp`.

```
results = ModelAdvisor.run('vdp', listOfChecks)
```

For more information on Model Advisor checks and sub-checks, see “Model Advisor Checks Documentation”. For details on how to find check IDs, see “Find Model Advisor Check IDs”.

Use the Model Advisor to Run a List of Checks Specified by a Configuration File

Run the Model Advisor on a list of checks specified by a Model Advisor configuration file.

Open the example model `sldemo_auto_climatecontrol`.

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

Copy the example script `prepare_cust_chk_code.m` to the current folder and run the script. The script copies files for this example to the current folder.

```
copyfile(fullfile(matlabroot,'examples','slcheck','main',...
    'prepare_cust_chk_code.m'),'prepare_cust_chk_code.m','f');

run('prepare_cust_chk_code.m');
```

Refresh the Model Advisor check information cache to include the files for this example.

```
Advisor.Manager.refresh_customizations()
```

Use the example configuration file `demoConfiguration.json` to specify which checks to include in the Model Advisor analysis. Save the file name `demoConfiguration.json` to the variable `fileName`.

```
fileName = 'demoConfiguration.json';
```

Create a list of the subsystems `sldemo_auto_climatecontrol/Heater Control` and `sldemo_auto_climatecontrol/AC Control`.

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

Use the Model Advisor to run the checks specified by the configuration file `demoConfiguration.json` on the subsystems specified by `systems`.

```
results = ModelAdvisor.run(systems,'Configuration',fileName);
```

For information on Model Advisor checks and sub-checks, use the links to the product-specific check documentation in “Model Advisor Checks Documentation”. For details on how to find check IDs, see “Find Model Advisor Check IDs”.

Use the Model Advisor to Generate a Report

Use the Model Advisor to run the checks and generate a report.

Open the model `vdp`.

```
open_system('vdp')
```

Create a list of check IDs. The check ID for the check “Check signal line connections” on page 2-148 is `'mathworks.jmaab.db_0032'` and the check ID for the check “Check trigger signal names” on page 2-155 is `'mathworks.jmaab.jc_0281'`.

```
checkIDs = {'mathworks.jmaab.db_0032','mathworks.jmaab.jc_0281'}
```

Use Model Advisor to run the checks on the model. Use the name-value arguments `'ReportFormat'`, `'ReportPath'`, and `'ReportName'` to generate a Model Advisor Report in the current folder, `pwd`, and in the format of a Microsoft® Word document.

```
ModelAdvisor.run('vdp', checkIDs,...
'ReportFormat', 'docx', 'ReportPath', pwd, 'ReportName', 'myReport')
```

For more information on Model Advisor checks and sub-checks, see “Model Advisor Checks Documentation”. For details on how to find check IDs, see “Find Model Advisor Check IDs”.

Input Arguments

`Systems` — List of models or subsystems

cell array of model names | cell array of subsystem names

List of models or subsystems, specified as a cell array of model names or subsystem names.

Example: {'vdp', 'sldemo_2counters'}

Example: {'sldemo_auto_climatecontrol/Heater Control', 'sldemo_auto_climatecontrol/AC Control'}

`CheckIDList` — List of check IDs

character vector | cell array of character vectors

Unique identifiers for the Model Advisor checks, specified as a character vector, or cell array of character vectors.

For information on how to find check IDs, see “Find Model Advisor Check IDs”.

Do not include duplicate check IDs in `CheckIDList`. If you need to run the same check multiple times, but with different input parameters, use either of the following approaches:

- Call `ModelAdvisor.run` separately for each different input parameter.

  For example, to run the check `'mathworks.jmaab.jc_0281'` two times, one time with the sub-check `'jc_0281_a'` disabled and one time with the sub-check `'jc_0281_a2'` selected:

  ```
  myCheck = 'mathworks.jmaab.jc_0281';
  inputParam1 = {'jc_0281_a',0}; % disable 'jc_0281_a' sub-checks
  inputParam2 = {'jc_0281_a',2}; % select sub-check 'jc_0281_a2'

  ModelAdvisor.run(modelName,{{myCheck,'InputParam',inputParam1}})
  ModelAdvisor.run(modelName,{{myCheck,'InputParam',inputParam2}})
  ```

- Create and run a Model Advisor configuration that uses separate check instances for each different input parameter. In the Model Advisor Configuration Editor, create a new folder for each different input parameter that you want to run, copy the check into each folder, and modify the checks in each folder to specify different input parameters. Use the `ModelAdvisor.run(Systems,'Configuration',FileName)` syntax to run the configuration.

Example: 'mathworks.jmaab.jc_0281'

Example: {'mathworks.maab.jc_0021', 'mathworks.misra.BlockNames'}

`CheckID` — Check ID

character vector

Unique identifier for the Model Advisor check, specified as a character vector.

For information on how to find check IDs, see “Find Model Advisor Check IDs”.

Example: 'mathworks.jmaab.jc_0281'

`paramName` — Name of Input Parameter for Model Advisor Check

character vector

The name of an input parameter, specified as a character vector.

You can view the input parameter names for a Model Advisor check by using `getInputParameters` on a `Simulink.ModelAdvisor` object. The input parameter name, `Name`, is a property of the `ModelAdvisor.InputParameter` object. For more information, see “Use the Model Advisor to Run Sub-Checks” on page 1-223.

The input parameters for a check can also be viewed in the Model Advisor Configuration Editor. Input parameter values can be saved to your custom Model Advisor configuration file. For more information, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”.

For more information on Model Advisor checks, sub-checks and input parameters, use the links to product-specific check documentation in “Model Advisor Checks Documentation”.

Example: 'Follow links'

Example: 'jc_0281_a'

`paramValue` — Value of Input Parameter for Model Advisor Check
character vector | integer

The value of an input parameter, specified as a character vector or integer.

You can view the input parameter values for a Model Advisor check by using `getInputParameters` on a `Simulink.ModelAdvisor` object. The current input parameter value, `Value`, is a property of the `ModelAdvisor.InputParameter` object. For more information, see “Use the Model Advisor to Run Sub-Checks” on page 1-223.

The input parameters for a check can also be viewed in the Model Advisor Configuration Editor. Input parameter values can be saved to your custom Model Advisor configuration file. For more information, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”.

For more information on Model Advisor checks, sub-checks and input parameters, use the links to product-specific check documentation in “Model Advisor Checks Documentation”.

Example: 'off'

Example: 4

`FileName` — Name of Model Advisor configuration file
character vector

Name of the Model Advisor configuration file, specified as a character vector. For details on creating a configuration file, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”.

Example: 'demoConfiguration.json'

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: 'ExtensiveAnalysis','Off'

`DisplayResults` — Result display setting
'Summary' (default) | 'Details' | 'None'

Report display setting for the Model Advisor, specified as `'Summary'`, `'Details'`, or `'None'`. Setting `DisplayResults` to `'Summary'` displays a summary of the system results in the Command Window. Setting `DisplayResults` to `'Details'` displays:

- Which system the Model Advisor is checking while the run is in progress
- The pass and fail results of each check for each system
- A summary of the system results

Setting `DisplayResults` to `'None'` displays no information in the Command Window.

Example: 'Details'

`ExtensiveAnalysis` — Extensive analysis setting
'On' (default) | 'Off'

Extensive analysis setting for the Model Advisor, specified as `'On'` or `'Off'`. Setting `'ExtensiveAnalysis'` to `'On'` runs each of the checks in your Model Advisor configuration, including checks that trigger extensive analysis. Setting `'ExtensiveAnalysis'` to `'Off'` runs only checks that do not trigger extensive analysis.

Checks that trigger extensive analysis of the model use additional analysis techniques, such as analysis with Simulink Design Verifier™. The checks that trigger extensive analysis have the icon  in front of the check name in the Model Advisor.

To use the `'ExtensiveAnalysis'` name-value argument, you must specify a check configuration file name with the `'Configuration'` argument.

Example: 'Off'

`Force` — Force delete setting
'Off' (default) | 'On'

Force delete setting for the Model Advisor, specified as `'Off'` or `'On'`. Setting `Force` to `'On'` removes existing `modeladvisor/`*`system`* folders. Setting `Force` to `'Off'` prompts you before removing existing `modeladvisor/`*`system`* folders.

Example: 'On'

`ParallelMode` — Parallel execution setting
'Off' (default) | 'On'

Parallel execution setting for the Model Advisor, specified as `'Off'` or `'On'`. Setting `ParallelMode` to `'On'` runs the Model Advisor in parallel mode if you have a Parallel Computing Toolbox license and a multicore machine. Parallel Computing Toolbox does not support 32-bit Windows® machines. Each parallel process runs checks on one model at a time. In parallel mode, load the model data from the model workspace or data dictionary. In parallel mode, the Model Advisor does not support model data in the base workspace. For an example, see “Create a Function to Check Multiple Systems”.

Example: 'On'

`TempDir` — Temporary working folder setting
'Off' (default) | 'On'

Temporary working folder setting for the Model Advisor, specified as `'Off'` or `'On'`. Setting `TempDir` to `'On'` runs the Model Advisor from a temporary working folder to avoid concurrency issues when running using a parallel pool. For more information, see “Resolving Data Concurrency Issues”. Setting `TempDir` to `'Off'` runs the Model Advisor in the current working folder.

Example: 'On'

`ShowExclusions` — Report exclusion display setting
'On' (default) | 'Off'

Exclusion display setting for the report, specified as `'On'` or `'Off'`. Setting `ShowExclusions` to `'On'` lists Model Advisor check exclusions in the report. Setting `ShowExclusions` to `'Off'` does not list Model Advisor check exclusion in the report.

Example: 'Off'

`ReportFormat` — Format of generated report
'html' (default) | 'pdf' | 'docx'

Format of the generated report, specified as HTML format, PDF format, or Microsoft Word document format.

Note Model Advisor displays an error if unsupported formats are selected. For the Windows operating system, the supported formats are `'html'`, `'pdf'`, and `'docx'`. For UNIX® like operating systems, the only supported format is `'html'`.

Example: 'docx'

`ReportPath` — Path to report file folder
'slprj/modeladvisor/*`modelName`*' (default) | character vector

Folder for the generated report, specified as a character vector. By default, `'ReportPath'` is the `'slprj/modeladvisor/`*`modelName`*`'` folder in the current working directory.

Example: 'C:\MyProject\MyReports'

`ReportName` — Prefix to Model Advisor report name
'report' (default) | character vector

Prefix for the Model Advisor report file name, specified as a character vector. An underscore and the model name, `'_`*`modelName`*`'`, are appended to the report file name. For example, if you run the Model Advisor on the model `vdp` with the `ReportName` `'MyReport'`, the Model Advisor report has the file name `'MyReport_vdp'`.

Example: 'MyReport'

Output Arguments

`Results` — Model Advisor check results
cell array of `ModelAdvisor.SystemResult` objects

Model Advisor check results, specified as a cell array of `ModelAdvisor.SystemResult` objects. The function returns one object for each model or subsystem specified by the input argument `Systems`.

Each `ModelAdvisor.SystemResult` object contains a `CheckResultObjs` property that contains an array of `ModelAdvisor.CheckResult` objects, one for each check run by `ModelAdvisor.run`.

To review the results without having to rerun the Model Advisor, save the results. For more information, see “Save and Load Process for Objects”.

Tips

- If you have a Parallel Computing Toolbox™ license and a multicore machine, Model Advisor can run on multiple systems in parallel. You can run the Model Advisor in parallel mode by using `ModelAdvisor.run` with `'ParallelMode'` set to `'On'`. By default, `'ParallelMode'` is set to `'Off'`. When you use `ModelAdvisor.run` with `'ParallelMode'` set to `'On'`, MATLAB automatically creates a parallel pool.

Alternatives

- Use the Model Advisor user interface to run each system. In the user interface, you can run only one system at a time.
- Create a script or function by using a `Simulink.ModelAdvisor` object to run each system, one at a time.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'ParallelMode'` to `'On'`.

See Also

`ModelAdvisor.lookupCheckID` | `getReportFileName` | `ModelAdvisor.summaryReport` | `view` | `viewReport`

Topics

“Create a Function to Check Multiple Systems”
“Create and Deploy a Model Advisor Custom Configuration”
“Find Model Advisor Check IDs”
“Model Advisor Checks Documentation”
“Use the Model Advisor Configuration Editor to Customize the Model Advisor”

ModelAdvisor.summaryReport

Package: `ModelAdvisor`

Open Model Advisor Command-Line Summary report

Syntax

```
ModelAdvisor.summaryReport(Results)
```

Description

`ModelAdvisor.summaryReport(Results)` opens the Model Advisor Command-Line Summary report in a web browser. `Results` is a cell array of `ModelAdvisor.SystemResult` objects returned by `ModelAdvisor.run`.

Examples

Open a Model Advisor Command-Line Summary Report

Run a check on a model and use the Command-Line Summary report to view the results.

Use Model Advisor to run the check **Check optimization settings** on the model `vdp`.

```
results = ModelAdvisor.run(["vdp"],...
    ["mathworks.design.OptimizationSettings"]);
```

Open the Model Advisor Command-Line Summary report to review the results.

```
ModelAdvisor.summaryReport(results)
```

For more information on Model Advisor checks, see “Model Advisor Checks Documentation”. For details on how to find check IDs, see “Find Model Advisor Check IDs”.

Input Arguments

`Results` — Model Advisor check results

cell array of `ModelAdvisor.SystemResult` objects

Model Advisor check results, specified as a cell array of `ModelAdvisor.SystemResult` objects returned by the function `ModelAdvisor.run`.

Each `ModelAdvisor.SystemResult` object contains a `CheckResultObjs` property that contains an array of `ModelAdvisor.CheckResult` objects, one for each check run by `ModelAdvisor.run`.

Alternative Functionality

“View Results in Model Advisor Command-Line Summary Report”

See Also

`getReportFileName` | `ModelAdvisor.lookupCheckID` | `ModelAdvisor.run` | `view` | `viewReport`

Topics

“Checking Systems Programmatically”
“Create a Function to Check Multiple Systems”
“Automate Model Advisor Check Execution”
“Archive and View Model Advisor Run Results”
“Model Advisor Checks Documentation”
“Find Model Advisor Check IDs”

ModelAdvisor.Table

Create table for Model Advisor results

Description

`ModelAdvisor.Table` objects create and format tables in the Model Advisor results. Specify the number of rows and columns in a table, excluding the table title and table heading row.

Creation

Syntax

```
table = ModelAdvisor.Table(row,column)
```

Description

`table = ModelAdvisor.Table(row,column)` creates a table object containing the number of rows and columns that you specify.

Input Arguments

`row` — Number of rows
positive integer

Number of rows to create in the Model Advisor results table.

`column` — Number of columns
positive integer

Number of columns to create in the Model Advisor results table.

Object Functions

getEntry	Get cell contents from table in Model Advisor analysis results
setColHeading	Specify column title for table in Model Advisor analysis results
setColHeadingAlign	Specify column title alignment
setColHeadingValign	Specify column title vertical alignment
setColWidth	Specify column widths
setEntries	Specify contents of table in Model Advisor analysis results
setEntry	Specify content cell in table in Model Advisor analysis results
setEntryAlign	Specify cell alignment for table in Model Advisor analysis results
setEntryValign	Specify table cell vertical alignment
setHeading	Specify title for table in Model Advisor analysis results
setHeadingAlign	Specify table title alignment
setRowHeading	Specify table row title
setRowHeadingAlign	Specify table row title alignment
setRowHeadingValign	Specify table row title vertical alignment

Examples

Create Table with Five Rows and Five Columns

Create a table that will appear in the Model Advisor results. This table has five rows and five columns that contain randomly generated numbers.

Use the following MATLAB code in a callback function. The Model Advisor displays `table1` in the results.

```
matrixData = rand(5,5) * 10^5;

% Initialize a table with 5 rows and 5 columns (heading rows not counting).
table1 = ModelAdvisor.Table(5,5);

% Set column headings
for n=1:5
    table1.setColHeading(n, ['Column ', num2str(n)]);
end

% Center the second column heading
table1.setColHeadingAlign(2, 'center');

% Set column width of the second column
table1.setColWidth(2, 3);

% Set the row headings
for n=1:5
    table1.setRowHeading(n, ['Row ', num2str(n)]);
end

% Enter table content
for rowIndex=1:5
    for colIndex=1:5
        table1.setEntry(rowIndex, colIndex, ...
            num2str(matrixData(rowIndex, colIndex)));

        % set alignment of entries in second row
        if colIndex == 2
            table1.setEntryAlign(rowIndex, colIndex, 'center');
        end
    end
end

% Overwrite the content of cell 3,3 with a ModelAdvisor.Text  object
text = ModelAdvisor.Text('Example Text');
table1.setEntry(3,3, text)
```

	Column 1	Column 2	Column 3	Column 4	Column 5
Row 1	81472.3686	9754.0405	15761.3082	14188.6339	65574.0699
Row 2	90579.1937	27849.8219	97059.2782	42176.1283	3571.1679
Row 3	12698.6816	54688.1519	Example Text	91573.5525	84912.9306
Row 4	91337.5856	95750.6835	48537.5649	79220.733	93399.3248
Row 5	63235.9246	96488.8535	80028.0469	95949.2426	67873.5155

Version History

Introduced in R2006b

See Also

`ModelAdvisor.Check`

Topics

“Create Model Advisor Checks”
“Customize the Configuration of the Model Advisor Overview”

ModelAdvisor.Task class

Package: `ModelAdvisor`

Define custom tasks

Description

The `ModelAdvisor.Task` class is a wrapper for a check so that you can access the check with the Model Advisor.

You can use one `ModelAdvisor.Check` object in multiple `ModelAdvisor.Task` objects, allowing you to place the same check in multiple locations in the Model Advisor tree. For example, **Check for implicit signal resolution** is displayed in the **By Product > Simulink** folder and in the **By Task > Model Referencing** folder in the Model Advisor tree.

When adding checks as tasks, the Model Advisor uses the task properties instead of the check properties, except for `Visible` and `LicenseName`.

Construction

ModelAdvisor.Task	Define custom tasks

Methods

setCheck	Specify check used in task

Properties

Description	Description of task
DisplayName	Name of task
Enable	Indicate if user can enable and disable task
ID	Identifier for task
LicenseName	Product license names required to display and run task
MAObj	Model Advisor object
Value	Status of task
Visible	Indicate to display or hide task

Copy Semantics

Handle. To learn how this affects your use of the class, see Copying Objects in the MATLAB Programming Fundamentals documentation.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');
MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
```

See Also

Topics

“Create Model Advisor Checks”
“Customize the Configuration of the Model Advisor Overview”

ModelAdvisor.Task

Class: `ModelAdvisor.Task`
Package: `ModelAdvisor`

Define custom tasks

Syntax

```
task_obj = ModelAdvisor.Task(task_ID)
```

Description

`task_obj = ModelAdvisor.Task(task_ID)` creates a task object, `task_obj`, with a unique identifier, `task_ID`. `task_ID` must remain constant. If you do not specify `task_ID`, the Model Advisor assigns a random `task_ID` to the task object.

You can use one `ModelAdvisor.Check` object in multiple `ModelAdvisor.Task` objects, allowing you to place the same check in multiple locations in the Model Advisor tree. For example, **Check for implicit signal resolution appears** in the **By Product > Simulink folder** and in the **By Task > Model Referencing** folder in the Model Advisor tree.

When adding checks as tasks, the Model Advisor uses the task properties instead of the check properties, except for `Visible` and `LicenseName`.

Examples

In the following example, you create three task objects, `MAT1`, `MAT2`, and `MAT3`.

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');
MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
```

See Also

Topics

“Customize the Configuration of the Model Advisor Overview”
“Programmatically Customize Tasks and Folders for the Model Advisor”
“Create Model Advisor Checks”

ModelAdvisor.Text

Create Model Advisor text output

Description

`ModelAdvisor.Text` objects create formatted text for the Model Advisor output.

Creation

Syntax

```
text = ModelAdvisor.Text(content,attribute)
```

Description

`text = ModelAdvisor.Text(content,attribute)` creates a text object for the Model Advisor output.

Input Arguments

`content` — Text object content
character vector

Optional character vector specifying the content of the text object. If `content` is empty, empty text is output.

`attribute` — Optional cell array
normal (default) | character vector | cell array of character vectors

Optional cell array of character vectors specifying the formatting of the content. If attribute is empty, the output text has the default coloring. Possible formatting options include:

- `normal` (default) — Text is black and unformatted.
- `bold` — Text is bold.
- `italic` — Text is italicized.
- `underline` — Text is underlined.
- `pass` — Text is green.
- `warn` — Text is yellow.
- `fail` — Text is red.
- `keyword` — Text is blue.
- `subscript` — Text is subscripted.
- `superscript` — Text is superscripted.

Add ASCII and Extended ASCII characters using the MATLAB `char` command.

Object Functions

setBold	Specify bold text in Model Advisor analysis results
setColor	Specify text color in Model Advisor analysis results
setHyperlink	Specify hyperlinked text
setItalic	Specify italic text in Model Advisor analysis results
setRetainSpaceReturn	Retain spacing and returns in text
setSubscript	Specify subscripted text
setSuperscript	Specify superscripted text
setUnderlined	Underline text

Examples

Specify Text in the Model Advisor Output

Text is the simplest form of output. You can format text in many different ways.

When you want one type of formatting for all text, use this syntax:

`ModelAdvisor.Text(content, {`*`attributes`*`})`

To apply multiple types of formatting, you must create several text objects and combine them.

```
t1 = ModelAdvisor.Text('It is ');
t2 = ModelAdvisor.Text('recommended', {'italic'});
t3 = ModelAdvisor.Text(' to use same font for ');
t4 = ModelAdvisor.Text('blocks', {'bold'});
t5 = ModelAdvisor.Text(' for a uniform appearance in the model.');

result = ([t1, t2, t3, t4, t5]);
```

Here is an example of a simple check callback function using the Model Advisor Formatting APIs:

```
function result = SampleStyleOneCallback(system)
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
if strcmp(get_param(bdroot(system), 'ScreenColor'),'white')
   result = ModelAdvisor.Text('Passed',{'pass'});
   mdladvObj.setCheckResultStatus(true);
else
   msg1 = ModelAdvisor.Text(...
       ['It is recommended to select a Simulink window screen color'...
       ' of white for a readable and printable model. Click ']);
   msg2 = ModelAdvisor.Text('here');
   msg2.setHyperlink('matlab: set_param(bdroot,''ScreenColor'',''white'')');
   msg3 = ModelAdvisor.Text(' to change screen color to white.');
   result = [msg1, msg2, msg3];
   mdladvObj.setCheckResultStatus(false);
end
```

Version History

Introduced in R2006b

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

publish

Publish object in Model Advisor root

Syntax

```
publish(root_obj, check_obj, location)
publish(root_obj, group_obj)
publish(root_obj, fg_obj)
```

Description

`publish(root_obj, check_obj, location)` specifies where the Model Advisor places the check in the Model Advisor tree. `location` is either one of the subfolders in the **By Product** folder, or the name of a new subfolder to put in the **By Product** folder. Use a pipe-delimited character vector to indicate multiple subfolders. For example, to add a check to the **Simulink Check > Modeling Standards** folder, use the following: `'Simulink Check|Modeling Standards'`.

`publish(root_obj, group_obj)` specifies the `ModelAdvisor.Group` object to publish as a folder in the **Model Advisor Task Manager** folder.

`publish(root_obj, fg_obj)` specifies the `ModelAdvisor.FactoryGroup` object to publish as a subfolder in the **By Task** folder.

Examples

```
% publish check into By Product > Demo group.
mdladvRoot.publish(rec, 'Demo');
```

See Also

Topics
“Define Custom Model Advisor Checks”
“Create and Deploy a Model Advisor Custom Configuration”

refresh_customizations

Class: `Advisor.Manager`
Package: `Advisor`

Refresh Model Advisor check information cache

Syntax

```
Advisor.Manager.refresh_customizations()
```

Description

`Advisor.Manager.refresh_customizations()` refreshes the Model Advisor check information cache.

Version History

Introduced in R2016b

See Also

Topics

“Fix a Model to Comply with Conditions that You Specify with the Model Advisor”
“Review a Model Against Conditions that You Specify with the Model Advisor”
“Create Model Advisor Check for Model Configuration Parameters”

getReportFileName

Get report file name for `ModelAdvisor.run` system result object

Syntax

```
getReportFileName(resultObj)
```

Description

`getReportFileName(resultObj)` gets the report file name of `ModelAdvisor.run` system result object. For more information on the system result object, see `ModelAdvisor.run`.

Example:

```
res = ModelAdvisor.run('vdp','mathworks.jmaab.jc_0627');
getReportFileName(res{1})
```

```
ans =

    'C:\Users\user21\AppData\Local\Temp\tp5ba7c1f9\slprj\modeladvisor\vdp\report.html'
```

Version History

Introduced in R2021a

See Also

`Advisor.Manager` | `ModelAdvisor.run`

Topics

"Automate Model Advisor Check Execution"

register

Register object in Model Advisor root

Syntax

```
register(MAobj, obj)
```

Description

`register(MAobj, obj)` registers the object, *`obj`*, in the root object `MAobj`.

In the Model Advisor memory, the `register` method registers the following types of objects:

- `ModelAdvisor.Check`
- `ModelAdvisor.FactoryGroup`
- `ModelAdvisor.Group`
- `ModelAdvisor.Task`

The `register` method places objects in the Model Advisor memory that you use in other functions. The `register` method does not place objects in the Model Advisor tree.

Examples

```
mdladvRoot = ModelAdvisor.Root;

MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.DisplayName='Example task with input parameter and auto-fix ability';
MAT1.setCheck('com.mathworks.sample.Check1');
mdladvRoot.register(MAT1);

MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');
MAT2.DisplayName='Example task 2';
MAT2.setCheck('com.mathworks.sample.Check2');
mdladvRoot.register(MAT2);

MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
MAT3.DisplayName='Example task 3';
MAT3.setCheck('com.mathworks.sample.Check3');
mdladvRoot.register(MAT3)
```

run

Class: `Advisor.Application`
Package: `Advisor`

Run Model Advisor analysis on model components

Syntax

```
run(app)
```

Description

`run(app)` runs a Model Advisor analysis, as specified by the `Application` object.

Examples

Run Model Advisor analysis

Create an `Application` object and run Model Advisor analysis.

Create an `Application` object.

```
app = Advisor.Manager.createApplication();
```

Open the model and set the root analysis to `RootModel`.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to sldemo_mdlref_basic model
RootModel = 'sldemo_mdlref_basic';

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);
```

Run a Model Advisor analysis.

```
run(app);
```

Input Arguments

app — Application

`Advisor.Application` object

`Advisor.Application` object, created by `Advisor.Manager.createApplication`

Version History

Introduced in R2015b

See Also

`Advisor.Application` | `Advisor.Manager.createApplication` | `setAnalysisRoot`

selectCheckInstances

Class: `Advisor.Application`
Package: `Advisor`

Select check instances to use in Model Advisor analysis

Syntax

```
selectCheckInstances(app)
selectCheckInstances(app,Name,Value)
```

Description

You can select check instances to use in a Model Advisor analysis. A check instance is an instantiation of a `ModelAdvisor.Check` object in the Model Advisor configuration. When you change the Model Advisor configuration, the check instance ID might change. To obtain the check instance ID, use the `getCheckInstanceIDs` method.

`selectCheckInstances(app)` selects all check instances to use for Model Advisor analysis.

`selectCheckInstances(app,Name,Value)` selects check instances specified by `Name,Value` pair arguments to use for Model Advisor analysis.

Input Arguments

app — Application

`Advisor.Application` object

`Advisor.Application` object, created by `Advisor.Manager.createApplication`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

IDs — Check instance IDs

cell array

Select check instances to use in Model Advisor analysis, as specified as a cell array of IDs

Data Types: `cell`

Examples

Select All Check Instances to Use in Model Advisor Analysis

This example shows how to set the root model, create an `Application` object, set root analysis, and select all check instances for Model Advisor analysis.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to sldemo_mdlref_basic model
RootModel='sldemo_mdlref_basic';

% Create an Application object
app = Advisor.Manager.createApplication();

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);

% Select all checks
selectCheckInstances(app);
```

Select Check Instance for Model Advisor Analysis Using Instance ID

This example shows how to set the root model, create an `Application` object, set root analysis, and select a check using instance ID.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to sldemo_mdlref_basic model
RootModel='sldemo_mdlref_basic';

% Create an Application object
app = Advisor.Manager.createApplication();

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);

% Select "Identify unconnected lines, input ports, and output
% ports" check using check instance ID
instanceID = getCheckInstanceIDs(app,'mathworks.design.UnconnectedLinesPorts');
checkinstanceID = instanceID(1);
selectCheckInstances(app,'IDs',checkinstanceID);
```

Version History

Introduced in R2015b

See Also

`Advisor.Manager.createApplication` | `setAnalysisRoot` | `getCheckInstanceIDs` | `deselectCheckInstances`

selectComponents

Class: `Advisor.Application`
Package: `Advisor`

Select model components for Model Advisor analysis

Syntax

```
selectComponents(app)
selectComponents(app,Name,Value)
```

Description

You can select model components for Model Advisor analysis. A model component is a model in the system hierarchy. Models that the root model references and that `Advisor.Application.setAnalysisRoot` specifies are model components. By default, all components are selected.

`selectComponents(app)` includes all components for Model Advisor analysis.

`selectComponents(app,Name,Value)` includes model components specified by `Name,Value` pair arguments for Model Advisor analysis.

Input Arguments

app — Application

`Advisor.Application` object

`Advisor.Application` object, created by `Advisor.Manager.createApplication`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

`IDs` — Component IDs

cell array

Components to select for Model Advisor analysis, as specified by a cell array of IDs

Data Types: `cell`

`HierarchicalSelection` — Select component and component children

false (default) | true

Select components specified by IDs and component children from Model Advisor analysis.

Data Types: `logical`

Examples

Include All Components in Model Advisor Analysis

This example shows how to set the root model, create an `Application` object, set root analysis, and include model components in Model Advisor analysis.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to  sldemo_mdlref_basic model
RootModel='sldemo_mdlref_basic';

% Create an Application object
app = Advisor.Manager.createApplication();

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);

% Select all components
selectComponents(app);
```

Select Components for Model Advisor Analysis Using IDs

This example shows how to set the root model, create an `Application` object, set root analysis, and include model components using IDs.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to  sldemo_mdlref_basic model
RootModel='sldemo_mdlref_basic';

% Create an Application object
app = Advisor.Manager.createApplication();

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);

% Select component using IDs
selectComponents(app,'IDs',RootModel);
```

Version History

Introduced in R2015b

See Also

`Advisor.Manager.createApplication` | `setAnalysisRoot` | `deselectComponents`

setAction

Specify action for check

Syntax

```
setAction(check_obj, action_obj)
```

Description

`setAction(check_obj, action_obj)` returns the action object `action.obj` to use in the check `check_obj`. The `setAction` method identifies the action you want to use in a check.

See Also

`ModelAdvisor.Action`

Topics

“Customize the Configuration of the Model Advisor Overview”
“Create Model Advisor Checks”

setAlign

Class: `ModelAdvisor.Paragraph`
Package: `ModelAdvisor`

Specify paragraph alignment

Syntax

```
setAlign(paragraph, alignment)
```

Description

`setAlign(paragraph, alignment)` specifies the alignment of text. Possible values are:

- `'left'` (default)
- `'right'`
- `'center'`

Examples

```
report_paragraph = ModelAdvisor.Paragraph;
setAlign(report_paragraph, 'center');
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setAnalysisRoot

Class: `Advisor.Application`
Package: `Advisor`

Specify model hierarchy for Model Advisor analysis

Syntax

```
setAnalysisRoot(app,'Root',root)
setAnalysisRoot(app,'Root',root,Name,Value)
```

Description

Specify the model hierarchy for an `Application` object analysis.

`setAnalysisRoot(app,'Root',root)` specifies the analysis root.

`setAnalysisRoot(app,'Root',root,Name,Value)` specifies the analysis root using `Name,Value` options.

Input Arguments

app — Application

`Advisor.Application` object

`Advisor.Application` object, created by `Advisor.Manager.createApplication`

'Root',root — Name,Value argument specifying model or subsystem path

character vector

Comma-separated `Name,Value` argument specifying model or subsystem path

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

RootType — Analysis root

`Model` (default) | `Subsystem`

Examples

Specify Root Model as Analysis Root

This example shows how to set the root model, create an `Application` object, and set the root analysis.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to sldemo_mdlref_basic model
RootModel='sldemo_mdlref_basic';

% Create an Application object
app = Advisor.Manager.createApplication();

% Set the Application object root analysis
setAnalysisRoot(app,'Root',RootModel);
```

Specify Subsystem as Analysis Root

This example shows how to set the root model, create an `Application` object, and specify a subsystem as the analysis root.

```
% Open the model
openExample('sldemo_mdlref_basic');

% Set root model to sldemo_mdlref_basic model
RootModel='sldemo_mdlref_basic';

% Create an Application object
app = Advisor.Manager.createApplication();

% Set the Application object root analysis
setAnalysisRoot(app,'Root','sldemo_mdlref_basic/CounterA','RootType','Subsystem');
```

Version History

Introduced in R2015b

See Also

`Advisor.Application` | `Advisor.Manager.createApplication`

setAnalysisRoot

Class: `slmetric.Engine`
Package: `slmetric`

Specify model or subsystem for metric analysis

Syntax

```
setAnalysisRoot(metric_engine,'Root',root)
setAnalysisRoot(metric_engine,'Root',root,Name,Value)
```

Description

Specify the model or subsystem for `slmetric.Engine` metric object analysis.

`setAnalysisRoot(metric_engine,'Root',root)` specifies the metric analysis root.

For an `slmetric.Engine` object, before you set the `ModelReferencesSimulationMode` and `AnalyzeLibraries` properties, specify the `setAnalysisRoot` method. The `setAnalysisRoot` method persists between model runs. If you change these property values and then apply the `setAnalysisRoot` method, the original values override the new values. For example, for one run, suppose you set these values:

```
metric_engine = slmetric.Engine();
setAnalysisRoot(metric_engine,'Root','vdp');
metric_engine.ModelReferencesSimulationMode = 'AllModes';
metric_engine.AnalyzeLibraries = true;
```

For the next run, if you want to change the `ModelReferencesSimulationMode` or the `AnalyzeLibraries` properties, be sure to first specify the `setAnalysisroot` method:

```
metric_engine = slmetric.Engine();
setAnalysisRoot(metric_engine,'Root','vdp');
metric_engine.AnalyzeModelReferences = false;
metric_engine.AnalyzeLibraries = false;
```

`setAnalysisRoot(metric_engine,'Root',root,Name,Value)` specifies the metric analysis root by using `Name,Value` pairs.

Input Arguments

`metric_engine` — Collects and accesses metric data

`slmetric.Engine` object

When you call `execute`, `metric_engine` collects metric data for all MathWorks metrics or for the specified `MetricIDs`. Calling `getMetrics` accesses the collected metric data in `metric_engine`.

`'Root'` — `Name,Value` argument specifying model or subsystem path

character vector

Comma-separated `Name,Value` argument specifying model or subsystem path. `Name` is the argument name and `Value` is the corresponding value. Name must appear inside quotes.

`RootType` — Type of model component for metric analysis
`Model` (default) | `Subsystem`

Comma-separated `Name,Value` argument specifying the `RootType`

Examples

Specify Model for Metric Analysis

This example shows how to specify a model for metric analysis.

Create an `slmetric.Engine` object.

```
metric_engine = slmetric.Engine();
```

Specify the model `vdp` as the root model for metric analysis.

```
setAnalysisRoot(metric_engine,'Root','vdp');
```

Specify Subsystem for Metric Analysis

This example shows how specify a subsystem for metric analysis.

Create an `slmetric.Engine` object.

```
metric_engine = slmetric.Engine();
```

Set the subsystem `enab1` as the root for metric analysis.

```
Subsys = 'sl_subsys_enab1/enab1';
setAnalysisRoot(metric_engine,'Root',Subsys,'RootType','Subsystem');
```

Version History

Introduced in R2016a

See Also

`slmetric.metric.ResultCollection` | `slmetric.metric.Metric` | `slmetric.metric.getAvailableMetrics` | `execute`

Topics
“Collect Model Metrics Programmatically”
“Model Metrics” on page 2-299

setBold

Package: `ModelAdvisor`

Specify bold text in Model Advisor analysis results

Syntax

```
setBold(textObj,boldText)
```

Description

`setBold(textObj,boldText)` makes `textObj` bold if `boldText` is `true` and makes `textObj` not bold if `boldText` is `false`.

Note The function `setBold` is for formatting Model Advisor analysis results with Simulink Check. For more information, see “Simulink Check”.

For information on how to format text in MATLAB, see “Formatting Text”.

Examples

Make Text Bold in Model Advisor Analysis Results

Create a Model Advisor text object and make the text bold.

Use `ModelAdvisor.Text` to create Model Advisor text object `t1`.

```
result = ModelAdvisor.Text('This is a message in bold.');
```

Make the text bold.

```
setBold(result, true);
```

Use `setBold` in a check callback function in your `sl_customization` file to format your results.

```
function result = SampleStyleOneCallback(system)
result = ModelAdvisor.Text('This is a message in bold.');
setBold(result, true);
end
```

For more information on using check callback functions to format text, see “Specify Text in the Model Advisor Output” on page 1-240. For more information on how to format check results, see “Display and Enable Check”.

Input Arguments

`textObj` — Model Advisor output text

`ModelAdvisor.Text` object

Model Advisor result text, specified as a `ModelAdvisor.Text` object.

`boldText` — Bold text setting
`true` or `1` | `false` or `0`

Bold text setting, specified as a numeric or logical `1` (`true`) or `0` (`false`). If `boldText` is `1`, the `textObj` is formatted bold. If `boldText` is `0`, the `textObj` is not formatted bold.

Data Types: `logical`

See Also

`ModelAdvisor.Text` | `setColor`

Topics
“Define Custom Model Advisor Checks”
“Display and Enable Check”

setCallbackFcn

Class: `ModelAdvisor.Action`
Package: `ModelAdvisor`

Specify action callback function

Syntax

```
setCallbackFcn(action_obj, @handle)
```

Description

`setCallbackFcn(action_obj, @handle)` specifies the handle to the callback function, `handle`, to use with the action object, `action_obj`.

Examples

The following example is a fragment of code is from “Create and Deploy a Model Advisor Custom Configuration”. The example does not execute as shown without the additional content found in the `sl_customization.m` and `defineDetailStyleCheck.m` files.

```
% 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
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”
`setActionEnable`

setCallbackFcn

Specify callback function for check

Syntax

`setCallbackFcn(check_obj, @handle, context, style)`

Description

`setCallbackFcn(check_obj, @handle, context, style)` specifies the callback function to use with the check, `check_obj`.

For the style input argument, to use the default format of the Model Advisor report, specify `DetailStyle`. With the default format, 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. `DetailStyle` also allows you to view results by block, subsystem, or recommended action. If the default formatting does not meet your needs, use the `ModelAdvisor.FormatTemplate` API or the other formatting APIs.

Input Arguments

`check_obj`	Instantiation of the `ModelAdvisor.Check` class
`handle`	Handle to a check callback function
`context`	Context for checking the model or subsystem: • `'None'` — No special requirements. • `'PostCompile'` — The model must be compiled. • `'PostCompileForCodegen'`— The model is compiled for code generation but is not simulated. Use this option to check the code generation readiness of a model and to analyze both active and inactive variant paths. To analyze both active and inactive variants, you must set the **System target file** configuration parameter to `ert.tlc`.
`style`	Type of callback function: • `'StyleOne'` — Simple check callback function, for formatting results using template • `'StyleTwo'` — Detailed check callback function • `'StyleThree'` — Check callback functions with hyperlinked results • `'DetailStyle'` — Check callback function for result collections. This style is recommended for authoring Model Advisor checks.

Examples

This example illustrates the definition for a check using a callback function whose style is defined as `DetailStyle`.

```
% This is the recommended style to author checks.
function defineModelAdvisorChecks
mdladvRoot = ModelAdvisor.Root;
rec = ModelAdvisor.Check('com.mathworks.sample.Check0');
rec.Title = 'Check whether block names appear below blocks (recommended check style)';
rec.TitleTips = 'Example new style callback (recommended check style)';
rec.setCallbackFcn(@SampleNewCheckStyleCallback,'None','DetailStyle');
% set fix operation
myAction0 = ModelAdvisor.Action;
myAction0.setCallbackFcn(@sampleActionCB0);
myAction0.Name='Make block names appear below blocks';
myAction0.Description='Click the button to place block names below blocks';
rec.setAction(myAction0);
mdladvRoot.publish(rec, 'Demo'); % publish check into Demo group.
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Define Custom Model Advisor Checks”
“Create Model Advisor Checks”

setCheck

Class: `ModelAdvisor.Task`
Package: `ModelAdvisor`

Specify check used in task

Syntax

```
setCheck(task, check_ID)
```

Description

`setCheck(task, check_ID)` specifies the check to use in the task.

You can use one `ModelAdvisor.Check` object in multiple `ModelAdvisor.Task` objects, allowing you to place the same check in multiple locations in the Model Advisor tree. For example, **Check for implicit signal resolution** appears in the **By Product > Simulink folder** and in the **By Task > Model Referencing** folder in the Model Advisor tree.

When adding checks as tasks, the Model Advisor uses the task properties instead of the check properties, except for `Visible` and `LicenseName`.

Input Arguments

`task`	Instantiation of the `ModelAdvisor.Task` class
`check_ID`	A unique identifier for the check to use in the task

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
setCheck(MAT1, 'com.mathworks.sample.Check1');
```

setCheckText

Add description of check to result

Syntax

`setCheckText(`*`ft_obj`*`, `*`text`*`)`

Description

`setCheckText(`*`ft_obj`*`, `*`text`*`)` is an optional method that adds text or a model advisor template object as the first item in the report. Use this method to add information describing the overall check.

Input Arguments

`ft_obj`

A handle to a template object.

`text`

A character vector or a handle to a formatting object.

Valid formatting objects are: `ModelAdvisor.Image`, `ModelAdvisor.LineBreak`, `ModelAdvisor.List`, `ModelAdvisor.Paragraph`, `ModelAdvisor.Table`, and `ModelAdvisor.Text`.

`text` appears as the first line in the analysis result.

Examples

Create a list object, `ft`, and add a line of text to the result:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setCheckText(ft, ['Identify unconnected lines, input ports,'...
    'and output ports in the model']);
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setColHeading

Package: `ModelAdvisor`

Specify column title for table in Model Advisor analysis results

Syntax

```
setColHeading(tableObj,column,heading)
```

Description

`setColHeading(tableObj,column,heading)` specifies the `heading` of the `column` in the Model Advisor table object `tableObj`.

Note The function `setColHeading` is for formatting tables in Model Advisor analysis results with Simulink Check. For more information, see “Simulink Check”.

For information on how to use tables in MATLAB, see “Create Tables and Assign Data to Them”.

Examples

Set the Column Heading for a Table in Model Advisor Results

Create a Model Advisor table object and add column headings.

Use `ModelAdvisor.Table` to create a Model Advisor table object with two rows and three columns.

```
T1 = ModelAdvisor.Table(2,3);
```

Specify column headings for the three columns in the Model Advisor table object.

```
setColHeading(T1,1,'Header 1');
setColHeading(T1,2,'Header 2');
setColHeading(T1,3,'Header 3');
```

Use `setEntry` in a check callback function in your `sl_customization` file to format your Model Advisor analysis results.

```
function result = SampleStyleOneCallback(system)
T1 = ModelAdvisor.Table(1,1);
T2 = ModelAdvisor.Table(2,3);
setEntry(T1,1,1,T2);
result = T1;
end
```

For more information on how to format check results, see “Define Custom Model Advisor Checks”.

Input Arguments

`tableObj` — Model Advisor table object

`ModelAdvisor.Table` object

Table of Model Advisor results, specified as a `ModelAdvisor.Table` object.

`column` — Table column

integer

Column of the table, specified by an integer.

`heading` — Heading title text

character vector | `Advisor.Element` object | `Advisor.Element` object array

Column heading title, specified by a character vector, `Advisor.Element` object, or `Advisor.Element` object array.

Example: 'Heading 1'

See Also

`ModelAdvisor.Table`

Topics

“Define Custom Model Advisor Checks”

setColHeadingAlign

Specify column title alignment

Syntax

`setColHeadingAlign(table, column, alignment)`

Description

`setColHeadingAlign(table, column, alignment)` specifies the alignment of the column heading.

Input Arguments

`table`	Instantiation of the `ModelAdvisor.Table` class
`column`	An integer specifying the column number
`alignment`	Alignment of the column heading. *`alignment`* can have one of the following values: • `left` (default) • `right` • `center`

Examples

```
table1 = ModelAdvisor.Table(2, 3);
setColHeading(table1, 1, 'Header 1');
setColHeadingAlign(table1, 1, 'center');
setColHeading(table1, 2, 'Header 2');
setColHeadingAlign(table1, 2, 'center');
setColHeading(table1, 3, 'Header 3');
setColHeadingAlign(table1, 3, 'center');
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setColHeadingValign

Specify column title vertical alignment

Syntax

`setColHeadingValign(table, column, alignment)`

Description

`setColHeadingValign(table, column, alignment)` specifies the vertical alignment of the column heading.

Input Arguments

`table`	Instantiation of the `ModelAdvisor.Table` class
`column`	An integer specifying the column number
`alignment`	Vertical alignment of the column heading. *`alignment`* can have one of the following values: • `top` (default) • `middle` • `bottom`

Examples

```
table1 = ModelAdvisor.Table(2, 3);
setColHeading(table1, 1, 'Header 1');
setColHeadingValign(table1, 1, 'middle');
setColHeading(table1, 2, 'Header 2');
setColHeadingValign(table1, 2, 'middle');
setColHeading(table1, 3, 'Header 3');
setColHeadingValign(table1, 3, 'middle');
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setColor

Package: `ModelAdvisor`

Specify text color in Model Advisor analysis results

Syntax

```
setColor(textObj,colorValue)
```

Description

`setColor(textObj,colorValue)` sets the `textObj` color to the color specified by `colorValue`.

Note The function `setColor` is for formatting Model Advisor analysis results with Simulink Check. For more information, see “Simulink Check”.

For information on how to format text in MATLAB, see “Formatting Text”.

Examples

Set Text Color in Model Advisor Analysis Results

Create a Model Advisor text object and make the text yellow.

Use `ModelAdvisor.Text` to create Model Advisor text object `t1`.

```
result = ModelAdvisor.Text('This is a warning in yellow.');
```

Set the text color to yellow. The text color yellow is associated with the value `'warn'`.

```
setColor(result, 'warn');
```

Use `setColor` in a check callback function in your `sl_customization` file to format your results.

```
function result = SampleStyleOneCallback(system)
result = ModelAdvisor.Text('This is a warning in yellow.');
setColor(result, 'warn');
end
```

For more information on using check callback functions to format text, see “Specify Text in the Model Advisor Output” on page 1-240. For more information on how to format check results, see “Display and Enable Check”.

Input Arguments

`textObj` — Model Advisor output text

`ModelAdvisor.Text` object

Model Advisor result text, specified as a `ModelAdvisor.Text` object.

`colorValue` — Text color setting
'normal' | 'pass' | 'warn' | 'fail' | 'keyword'

Text color setting, specified by one of the values listed in this table.

Value	Text Color
'normal'	Black
'pass'	Green
'warn'	Yellow
'fail'	Red
'keyword'	Blue

See Also

`ModelAdvisor.Text` | `setBold`

Topics
“Define Custom Model Advisor Checks”
“Display and Enable Check”

setColSpan

Class: `ModelAdvisor.InputParameter`
Package: `ModelAdvisor`

Specify number of columns for input parameter

Syntax

```
setColSpan(input_param, [start_col end_col])
```

Description

`setColSpan(input_param, [start_col end_col])` specifies the number of columns that the parameter occupies. Use the `setColSpan` method to specify where you want an input parameter located in the layout grid when there are multiple input parameters.

Input Arguments

`input_param`	Instantiation of the `ModelAdvisor.InputParameter` class
`start_col`	A positive integer representing the first column that the input parameter occupies in the layout grid
`end_col`	A positive integer representing the last column that the input parameter occupies in the layout grid

Examples

```
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
inputParam2.setRowSpan([2 2]);
inputParam2.setColSpan([1 1]);
```

setColTitles

Package: `ModelAdvisor`

Add column titles to table in Model Advisor analysis results

Syntax

```
setColTitles(ftObj,colTitles)
```

Description

`setColTitles(ftObj,colTitles)` sets the column titles in the table specified by Model Advisor formatting object `ftObj` to the title text specified by `colTitles`. If you do not add data to a table, Model Advisor does not display the table in the results.

Note Before adding data to a table, you must specify column titles using the `setColTitles` method.

The function `setColTitles` is for formatting tables in Model Advisor analysis results with Simulink Check. For more information, see “Simulink Check”.

For information on how to use tables in MATLAB, see “Create Tables and Assign Data to Them”.

Examples

Add Column Titles to a Table Template Object

Create a Model Advisor format template object, `ft`, of `type 'TableTemplate'`, and specify two column titles.

Use `ModelAdvisor.FormatTemplate` to create a Model Advisor formatting template `ft` of `type 'TableTemplate'`.

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

Specify the column titles for the table template object. Specify `'Index'` as the title for the first column and `'Block Name'` for the second column.

```
setColTitles(ft,{'Index','Block Name'});
```

Use `setColTitles` in a check callback function in your `sl_customization` file.

```
function result = SampleStyleOneCallback(system)
ft = ModelAdvisor.FormatTemplate('TableTemplate');
setTableTitle(ft,{'Blocks in Model'});
setColTitles(ft,{'Index','Block Name'});
setTableInfo(ft,{'1', 'Gain'})
result = ft;
end
```

For more information on how to format check results, see “Define Custom Model Advisor Checks”.

Input Arguments

`ftObj` — Template object

template object handle

`ModelAdvisor.FormatTemplate` object, specified as a handle to the template object.

`colTitles` — Table column title text

cell array of character vectors | handles to formatting objects

Table column title text, specified as a cell array of character vectors or a handle to a valid formatting object.

Valid formatting objects:

- `ModelAdvisor.Image`
- `ModelAdvisor.LineBreak`
- `ModelAdvisor.List`
- `ModelAdvisor.Paragraph`
- `ModelAdvisor.Table`
- `ModelAdvisor.Text`

The order of the items in the cell array determines which column the item is in.

Example: `{'Index','Block Name'}`

See Also

`ModelAdvisor.FormatTemplate` | `setTableInfo` | `setTableTitle`

Topics

“Define Custom Model Advisor Checks”

“Format Model Advisor Results” on page 1-198

setColWidth

Specify column widths

Syntax

```
setColWidth(table, column, width)
```

Description

`setColWidth(table, column, width)` specifies the `column`.

The `setColWidth` method specifies the table column widths relative to the entire table width. If column widths are [1 2 3], the second column is twice the width of the first column, and the third column is three times the width of the first column. Unspecified columns have a default width of 1. For example:

```
setColWidth(1, 1);
setColWidth(3, 2);
```

specifies [1 1 2] column widths.

Input Arguments

`table`	Instantiation of the `ModelAdvisor.Table` class
`column`	An integer specifying column number
`width`	An integer or array of integers specifying the column widths, relative to the entire table width

Examples

```
table1 = ModelAdvisor.Table(2, 3)
setColWidth(table1, 1, 1);
setColWidth(table1, 3, 2);
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setEntries

Package: `ModelAdvisor`

Specify contents of table in Model Advisor analysis results

Syntax

```
setEntries(tableObj,tableContent)
```

Description

`setEntries(tableObj,tableContent)` specifies the content, `tableContent`, of a table, `tableObj`, in Model Advisor analysis results.

Note The function `setEntries` is for formatting Model Advisor analysis results with Simulink Check. For more information, see “Simulink Check”.

For information on how to set cell contents in MATLAB, see “Create Tables and Assign Data to Them”.

Examples

Add Content to a Table in a Model Advisor Table Object

Create a Model Advisor table object, `T1`, and insert the cell array, `contents`, into the cells of the table.

Use `ModelAdvisor.Table` to create a table object.

```
T1 = ModelAdvisor.Table(2,3);
```

Create a two-dimensional cell array. For this example, create a 2-by-3 cell array where each cell contains a character vector.

```
contents = {'a','b','c';'d','e','f'};
```

Use `setEntries` in a check callback function in your `sl_customization` file to format your results.

```
function result = SampleStyleOneCallback(system)
T1 = ModelAdvisor.Table(2,3);
contents = {'a','b','c';'d','e','f'};
setEntries(T1,contents);
result = T1;
end
```

For more information on how to format check results, see “Define Custom Model Advisor Checks”.

Input Arguments

`tableObj` — Model Advisor table object

`ModelAdvisor.Table` object

Table of Model Advisor results, specified as a `ModelAdvisor.Table` object.

`tableContent` — Content in table of Model Advisor results

two-dimensional cell array

Content in table of Model Advisor results, specified as a two-dimensional cell array containing the contents of the table. Each cell in the cell array must be either a character vector or an instance of `ModelAdvisor.Element`. The size of the cell array must be equal to the size of the table specified in the `ModelAdvisor.Table` constructor.

Example: `{'Cell 1','Cell 3','Cell 5';'Cell 2','Cell 4','Cell 6'}`

Data Types: `char`

See Also

`ModelAdvisor.Table`

Topics

“Define Custom Model Advisor Checks”

setEntry

Package: `ModelAdvisor`

Specify content cell in table in Model Advisor analysis results

Syntax

```
setEntry(tableObj,row,column,cellEntry)
```

Description

`setEntry(tableObj,row,column,cellEntry)` adds the content specified by `cellEntry` to the cell in row, `row`, and column, `column`, in the Model Advisor table object `tableObj`.

Note The function `setEntry` is for formatting tables in Model Advisor analysis results with Simulink Check. For more information, see “Simulink Check”.

For information on how to set cell contents in MATLAB, see “Create Tables and Assign Data to Them”.

Examples

Add Content to a Cell in a Model Advisor Table Object

Create two Model Advisor table objects, `T1` and `T2`, and insert `T2` into the first cell of `T1`.

Use `ModelAdvisor.Table` to create two Model Advisor table objects.

```
T1 = ModelAdvisor.Table(1,1);
T2 = ModelAdvisor.Table(2,3);
```

Add `T2` to the cell in the first row and first column of the Model Advisor table object `T1`.

```
setEntry(T1,1,1,T2);
```

Use `setEntry` in a check callback function in your `sl_customization` file to set table cell content in your Model Advisor results.

```
function result = SampleStyleOneCallback(system)
T1 = ModelAdvisor.Table(1,1);
T2 = ModelAdvisor.Table(2,3);
setEntry(T1,1,1,T2);
result = T1;
end
```

For more information on how to format check results, see “Define Custom Model Advisor Checks”.

Input Arguments

`tableObj` — Model Advisor table object

`ModelAdvisor.Table` object

Table of Model Advisor results, specified as a `ModelAdvisor.Table` object.

`row` — Table row

integer

Row of the table, specified by an integer.

`column` — Table column

integer

Column of the table, specified by an integer.

`cellEntry` — Table cell content

character vector | `ModelAdvisor.Element` object | `ModelAdvisor.Element` object array

Table cell content, specified by a character vector, `ModelAdvisor.Element` object, or `ModelAdvisor.Element` object array.

See Also

`ModelAdvisor.Table`

Topics

“Define Custom Model Advisor Checks”

setEntryAlign

Package: `ModelAdvisor`

Specify cell alignment for table in Model Advisor analysis results

Syntax

```
setEntryAlign(tableObj,row,column,alignment)
```

Description

`setEntryAlign(tableObj,row,column,alignment)` sets the horizontal alignment of the cell in the row, `row`, and column, `column`, of Model Advisor table object `tableObj` to the alignment specified by `alignment`.

Note The function `setEntryAlign` is for formatting tables in Model Advisor analysis results with Simulink Check. For more information, see “Simulink Check”.

For information on how to use tables in MATLAB, see “Create Tables and Assign Data to Them”.

Examples

Set Cell Content Alignment in a Model Advisor Table Object

Create a Model Advisor table object, add content to a cell, and set the cell text alignment.

Use `ModelAdvisor.Table` to create a Model Advisor table object with two rows and three columns.

```
T1 = ModelAdvisor.Table(2,3);
```

Add a title to the table.

```
setHeading(T1,'Title of New Table');
```

Add the text `'First Entry'` to the cell in the first row and first column of the Model Advisor table object `T1`.

```
setEntry(T1,1,1,'First Entry');
```

Center the content in the cell in the first row and first column of the Model Advisor table object.

```
setEntryAlign(T1,1,1,'center');
```

Use `setEntryAlign` in a check callback function in your `sl_customization` file to format your Model Advisor analysis results.

```
function result = SampleStyleOneCallback(system)
T1 = ModelAdvisor.Table(2,3);
setHeading(T1,'Title of New Table');
```

```
setEntry(T1,1,1,'First Entry');
setEntryAlign(T1,1,1,'center');
result = T1;
end
```

For more information on how to format check results, see “Define Custom Model Advisor Checks”.

Input Arguments

`tableObj` — Model Advisor table object

`ModelAdvisor.Table` object

Table of Model Advisor results, specified as a `ModelAdvisor.Table` object.

`row` — Table row

integer

Row of the table, specified by an integer.

`column` — Table column

integer

Column of the table, specified by an integer.

`alignment` — Horizontal cell alignment

'left' | 'center' | 'right'

Horizontal cell alignment, specified as 'left', 'center', or 'right'.

alignment	Definition
'left'	Horizontally align contents to the left edge of the cell.
'center'	Horizontally center cell contents.
'right'	Horizontally align contents to the right edge of the cell.

See Also

`ModelAdvisor.Table`

Topics

“Define Custom Model Advisor Checks”

setEntryValign

Specify table cell vertical alignment

Syntax

`setEntryValign(table, row, column, alignment)`

Description

`setEntryValign(table, row, column, alignment)` specifies the cell alignment of the designated cell.

Input Arguments

`table`	Instantiation of the `ModelAdvisor.Table` class
`row`	An integer specifying row number
`column`	An integer specifying column number
`alignment`	Cell vertical alignment, specified as one of the following: • `'top'` (default) • `'middle'` • `'bottom'`

Examples

```
table1 = ModelAdvisor.Table(2,3);
setHeading(table1, 'New Table');
.
.
.
setEntry(table1, 1, 1, 'First Entry');
setEntryValign(table1, 1, 1, 'middle');
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setHeading

Package: `ModelAdvisor`

Specify title for table in Model Advisor analysis results

Syntax

```
setHeading(tableObj,titleText)
```

Description

`setHeading(tableObj,titleText)` adds the title specified by `titleText` to the Model Advisor table object `tableObj`.

Note The function `setHeading` is for formatting tables in Model Advisor analysis results with Simulink Check. For more information, see “Simulink Check”.

For information on how to use tables in MATLAB, see “Create Tables and Assign Data to Them”.

Examples

Add a Title to a Model Advisor Table Object

Create a Model Advisor table object and add a title to the table.

Use `ModelAdvisor.Table` to create a Model Advisor table object with two rows and three columns.

```
T1 = ModelAdvisor.Table(2,3);
```

Add a title to the table.

```
setHeading(T1,'Title of New Table');
```

Use `setHeading` in a check callback function in your `sl_customization` file to format your Model Advisor analysis results.

```
function result = SampleStyleOneCallback(system)
T1 = ModelAdvisor.Table(2,3);
setHeading(T1,'Title of New Table');
result = T1;
end
```

For more information on how to format check results, see “Define Custom Model Advisor Checks”.

Input Arguments

`tableObj` — Model Advisor table object

`ModelAdvisor.Table` object

Table of Model Advisor results, specified as a `ModelAdvisor.Table` object.

`titleText` — Title text
character vector | `ModelAdvisor.Element` object | `ModelAdvisor.Element` object array

Title text, specified as a character vector, `ModelAdvisor.Element` object, or `ModelAdvisor.Element` object array.

Example: 'Table of fonts and styles used in model'

See Also

`ModelAdvisor.Table`

Topics
“Define Custom Model Advisor Checks”

setHeadingAlign

Specify table title alignment

Syntax

`setHeadingAlign(table, alignment)`

Description

`setHeadingAlign(table, alignment)` specifies the alignment for the table title.

Input Arguments

`table`	Instantiation of the `ModelAdvisor.Table` class
`alignment`	Table title alignment, specified as one of the following: • `'left'` (default) • `'right'` • `'center'`

Examples

```
table1 = ModelAdvisor.Table(2, 3);
setHeading(table1, 'New Table');
setHeadingAlign(table1, 'center');
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setHyperlink

Class: `ModelAdvisor.Image`
Package: `ModelAdvisor`

Specify hyperlink location

Syntax

```
setHyperlink(image, url)
```

Description

`setHyperlink(image, url)` specifies the target location of the hyperlink associated with `image`.

Input Arguments

`image`	Instantiation of the `ModelAdvisor.Image` class
`url`	The target URL

Examples

```
matlab_logo=ModelAdvisor.Image;
setHyperlink(matlab_logo, 'https://www.mathworks.com');
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setHyperlink

Specify hyperlinked text

Syntax

```
setHyperlink(text, url)
```

Description

`setHyperlink(text, url)` creates a hyperlink from the text to the specified URL.

Input Arguments

`text`	Instantiation of the `ModelAdvisor.Text` class
`url`	The target location of the URL

Examples

```
t1 = ModelAdvisor.Text('MathWorks home page');
setHyperlink(t1, 'https://www.mathworks.com');
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setImageSource

Class: `ModelAdvisor.Image`
Package: `ModelAdvisor`

Specify image location

Syntax

```
setImageSource(image_obj, source)
```

Description

`setImageSource(image_obj, source)` specifies the location of the image.

Input Arguments

`image_obj`	Instantiation of the `ModelAdvisor.Image` class
source	The location of the image file

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setInformation

Add description of subcheck to result

Syntax

`setInformation(`*`ft_obj`*`, `*`text`*`)`

Description

`setInformation(`*`ft_obj`*`, `*`text`*`)` is an optional method that adds text as the first item after the subcheck title. Use this method to add information describing the subcheck.

Input Arguments

`ft_obj`

A handle to a template object.

`text`

A character vector or a handle to a formatting object, that describes the subcheck.

Valid formatting objects are: `ModelAdvisor.Image`, `ModelAdvisor.LineBreak`, `ModelAdvisor.List`, `ModelAdvisor.Paragraph`, `ModelAdvisor.Table`, and `ModelAdvisor.Text`.

The Model Advisor displays *`text`* after the title of the subcheck.

Examples

Create a list object, `ft`, and specify a subcheck title and description:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setSubTitle(ft, ['Check for constructs in the model '...
    'that are not supported when generating code']);
setInformation(ft, ['Identify blocks that should not '...
    'be used for code generation.']);
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setInputParameters

Specify input parameters for check

Syntax

```
setInputParameters(check_obj, params)
```

Description

`setInputParameters(check_obj, params)` specifies `ModelAdvisor.InputParameter` objects (`params`) to be used as input parameters to a check (`check_obj`).

Input Arguments

`check_obj`	Instantiation of the `ModelAdvisor.Check` class
`params`	A cell array of `ModelAdvisor.InputParameters` objects

Examples

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
inputParam1 = ModelAdvisor.InputParameter;
inputParam2 = ModelAdvisor.InputParameter;
inputParam3 = ModelAdvisor.InputParameter;
setInputParameters(rec, {inputParam1,inputParam2,inputParam3});
```

See Also

`ModelAdvisor.InputParameter`

Topics

“Customize the Configuration of the Model Advisor Overview”
“Create Model Advisor Checks”

setInputParametersLayoutGrid

Specify layout grid for input parameters

Syntax

```
setInputParametersLayoutGrid(check_obj, [row col])
```

Description

`setInputParametersLayoutGrid(check_obj, [row col])` specifies the layout grid for input parameters in the Model Advisor. Use the `setInputParametersLayoutGrid` method when there are multiple input parameters.

Input Arguments

`check_obj`	Instantiation of the `ModelAdvisor.Check` class
`row`	Number of rows in the layout grid
`col`	Number of columns in the layout grid

Examples

The following example is a fragment of code from a check definition function. The example does not execute as shown without the full check definition function.

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

See Also

`ModelAdvisor.InputParameter`

Topics
“Customize the Configuration of the Model Advisor Overview”
“Create Model Advisor Checks”

setItalic

Package: `ModelAdvisor`

Specify italic text in Model Advisor analysis results

Syntax

```
setItalic(textObj,italicizeText)
```

Description

`setItalic(textObj,italicizeText)` makes `textObj` italicized if `italicizeText` is `true` and makes `textObj` not italicized if `italicizeText` is `false`.

Note The function `setItalic` is for formatting text in Model Advisor analysis results with Simulink Check. For more information, see “Simulink Check”.

For information on how to format text in MATLAB, see “Formatting Text”.

Examples

Make Text Italicized in Model Advisor Analysis Results

Create a Model Advisor text object and make the text italicized.

Use `ModelAdvisor.Text` to create Model Advisor text object `t1`.

```
result = ModelAdvisor.Text('This is a message in italics.');
```

Make the text italicized.

```
setItalic(result, true);
```

Use `setItalic` in a check callback function in your `sl_customization` file to format your Model Advisor analysis results.

```
function result = SampleStyleOneCallback(system)
result = ModelAdvisor.Text('This is a message in italics.');
setItalic(result, true);
end
```

For more information on using check callback functions to format text, see “Specify Text in the Model Advisor Output” on page 1-240. For more information on how to format check results, see “Define Custom Model Advisor Checks”.

Input Arguments

`textObj` — Model Advisor output text

`ModelAdvisor.Text` object

Model Advisor result text, specified as a `ModelAdvisor.Text` object.

`italicizeText` — Italic text setting
`true` or `1` | `false` or `0`

Italic text setting, specified as a numeric or logical `1` (`true`) or `0` (`false`). If `italicizeText` is `1`, the `textObj` is italicized. If `italicizeText` is `0`, the `textObj` is not italicized.

Data Types: `logical`

See Also

`ModelAdvisor.Text` | `setBold` | `setColor`

Topics
“Define Custom Model Advisor Checks”

setListObj

Add list of hyperlinks to model objects

Syntax

`setListObj(`*`ft_obj`*`, {`*`model_obj`*`})`

Description

`setListObj(`*`ft_obj`*`, {`*`model_obj`*`})` is an optional method that generates a bulleted list of hyperlinks to model objects. *`ft_obj`* is a handle to a list template object. *`model_obj`* is a cell array of handles or full paths to blocks, or model objects that the Model Advisor displays as a bulleted list of hyperlinks in the report.

Examples

Create a list object, `ft`, and add a list of the blocks found in the model:

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

% Find all the blocks in the system
allBlocks = find_system(system);

% Add the blocks to a list
setListObj(ft, allBlocks);
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setRecAction

Add Recommended Action section and text

Syntax

`setRecAction(ft_obj,recActionText)`

Description

`setRecAction(ft_obj,recActionText)` is an optional method that adds a Recommended Action section to the report. Use this method to describe how to fix the check.

Input Arguments

`ft_obj`

A handle to a template object.

`recActionText`

A cell array of character vectors or handles to formatting objects, that describes the recommended action to fix the issues reported by the check.

Valid formatting objects are: `ModelAdvisor.Image`, `ModelAdvisor.LineBreak`, `ModelAdvisor.List`, `ModelAdvisor.Paragraph`, `ModelAdvisor.Table`, and `ModelAdvisor.Text`.

The Model Advisor displays the recommended action as a separate section below the list or table in the report.

Examples

Create a list object, `ft`, find Gain blocks in the model, and recommend changing them:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
% Find all Gain blocks
gainBlocks = find_system(gcs, 'BlockType','Gain');

% Find Gain blocks
for idx = 1:length(gainBlocks)
    gainObj = get_param(gainBlocks(idx), 'Object');

    setRecAction(ft, {'If you are using these blocks '...
    'as buffers, you should replace them with '...
    'Signal Conversion blocks'});
end
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics
“Create Model Advisor Checks”

setRefLink

Add See Also section and links

Syntax

```
setRefLink(ft_obj, {{'standard'}})
setRefLink(ft_obj, {{'url', 'standard'}})
```

Description

`setRefLink(ft_obj, {{'standard'}})` is an optional method that adds a See Also section above the table or list in the result. Use this method to add references to standards. *`ft_obj`* is a handle to a template object. *`standard`* is a cell array of character vectors that you want to display in the result. If you include more than one cell, the Model Advisor displays the character vectors in a bulleted list.

`setRefLink(ft_obj, {{'url', 'standard'}})` generates a list of links in the See Also section. *`url`* indicates the location to link to. You must provide the full link including the protocol. For example, `https:\\www.mathworks.com` is a valid link, while `www.mathworks.com` is not a valid link. You can create a link to a protocol that is valid URL, such as a web site address, a full path to a file, or a relative path to a file.

Note `setRefLink` expects a cell array of cell arrays for the second input.

Examples

Create a list object, `ft`, and add a related standard:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setRefLink(ft, {{'IEC 61508-3, Table A.3 (3) ''Language subset'''}});
```

Create a list object, `ft`, and add a list of related standards:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setRefLink(ft, {
    {'IEC 61508-3, Table A.3 (2) ''Strongly typed programming language'''},...
    {'IEC 61508-3, Table A.3 (3) ''Language subset'''}});
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setRetainSpaceReturn

Retain spacing and returns in text

Syntax

`setRetainSpaceReturn(text, mode)`

Description

`setRetainSpaceReturn(text, mode)` specifies whether the text must retain the spaces and carriage returns.

Input Arguments

`text`	Instantiation of the `ModelAdvisor.Text` class
`mode`	A Boolean value indicating whether to preserve spaces and carriage returns in the text: • `true` (default) — Preserve spaces and carriage returns. • `false` — Do not preserve spaces and carriage returns.

Examples

```
t1 = ModelAdvisor.Text('MathWorks home page');
setRetainSpaceReturn(t1, 'true');“Customize the Configuration of the Model Advisor Overview”
```

setRowHeading

Specify table row title

Syntax

```
setRowHeading(table, row, heading)
```

Description

`setRowHeading(table, row, heading)` specifies a title for the designated table row.

Input Arguments

`table`	Instantiation of the `ModelAdvisor.Table` class
`row`	An integer specifying row number
`heading`	A character vector, element object, or object array specifying the table row title

Examples

```
table1 = ModelAdvisor.Table(2,3);
setRowHeading(table1, 1, 'Row 1 Title');
setRowHeading(table1, 2, 'Row 2 Title');
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setRowHeadingAlign

Specify table row title alignment

Syntax

`setRowHeadingAlign(table, row, alignment)`

Description

`setRowHeadingAlign(table, row, alignment)` specifies the alignment for the designated table row.

Input Arguments

`table`	Instantiation of the `ModelAdvisor.Table` class
`row`	An integer specifying row number.
`alignment`	Cell alignment, specified as one of the following: • `'left'` (default) • `'right'` • `'center'`

Examples

```
table1 = ModelAdvisor.Table(2, 3);
setRowHeading(table1, 1, 'Row 1 Title');
setRowHeadingAlign(table1, 1, 'center');
setRowHeading(table1, 2, 'Row 2 Title');
setRowHeadingAlign(table1, 2, 'center');
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setRowHeadingValign

Specify table row title vertical alignment

Syntax

`setRowHeadingValign(table, row, alignment)`

Description

`setRowHeadingValign(table, row, alignment)` specifies the vertical alignment for the designated table row.

Input Arguments

`table`	Instantiation of the `ModelAdvisor.Table` class
`row`	An integer specifying row number.
`alignment`	Cell vertical alignment, specified as one of the following: • `'top'` (default) • `'middle'` • `'bottom'`

Examples

```
table1 = ModelAdvisor.Table(2, 3);
setRowHeading(table1, 1, 'Row 1 Title');
setRowHeadingValign(table1, 1, 'middle');
setRowHeading(table1, 2, 'Row 2 Title');
setRowHeadingValign(table1, 2, 'middle');
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setRowSpan

Class: `ModelAdvisor.InputParameter`
Package: `ModelAdvisor`

Specify rows for input parameter

Syntax

```
setRowSpan(input_param, [start_row end_row])
```

Description

`setRowSpan(input_param, [start_row end_row])` specifies the number of rows that the parameter occupies. Specify where you want an input parameter located in the layout grid when there are multiple input parameters.

Input Arguments

`input_param`	The input parameter object
`start_row`	A positive integer representing the first row that the input parameter occupies in the layout grid
`end_row`	A positive integer representing the last row that the input parameter occupies in the layout grid

Examples

```
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
inputParam2.setRowSpan([2 2]);
inputParam2.setColSpan([1 1]);
```

setSubBar

Add line between subcheck results

Syntax

`setSubBar(ft_obj, value)`

Description

`setSubBar(ft_obj, value)` is an optional method that adds lines between results for subchecks. *`ft_obj`* is a handle to a template object. *`value`* is a boolean value that specifies when the Model Advisor includes a line between subchecks in the check results. By default, the value is `true`, and the Model Advisor displays the bar. The Model Advisor does not display the bar when you set the value to `false`.

Examples

Create a list object, `ft`, turn off the subbar:

```
ft = ModelAdvisor.FormatTemplate('ListTemplate');
setSubBar(ft, false);
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setSubResultStatus

Add status to the check or subcheck result

Syntax

`setSubResultStatus(`*`ft_obj`*`, '`*`status`*`')`

Description

`setSubResultStatus(`*`ft_obj`*`, '`*`status`*`')` is an optional method that displays the status in the result. Use this method to display the status of the check or subcheck in the result. *`ft_obj`* is a handle to a template object. *`status`* is a character vector identifying the status of the check:
`Pass`: Check did not identify issues.
`D-Pass`: Dependent on configuration parameter. Check did not identify issues.
`Warn`: Check has identified issues.
`Fail`: Check fails to execute.

Examples

This example shows how to create a callback function for a custom check that finds and reports optimization settings. The check consists of two subchecks. The first reviews the **Block reduction** optimization setting and the second reviews the **Conditional input branch execution** optimization setting.

A check with subchecks includes the following items in the results:

- A description of what the overall check is reviewing.
- A title for the subcheck.
- A description of what the subcheck is reviewing.
- References to standards, if applicable.
- The status of the subcheck.
- A description of the status.
- Results for the subcheck.
- Recommended actions to take when the subcheck does not pass.
- A line between the subcheck results.

```
% Sample Check 3 Callback Function: Check with Subchecks and Actions
% Find and report optimization settings
function ResultDescription = OptmizationSettingCallback(system)
% Initialize variables
system =getfullname(system);
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(false); % Default check status is 'Warning'
ResultDescription = {};

% Format results in a list using Model Advisor Result Template API
% Create a list template object for first subcheck
ft1 = ModelAdvisor.FormatTemplate('ListTemplate');

% Description of check in results
setCheckText(ft1,'Check optimization settings');
```

```
% Title and description of first subcheck
setSubTitle(ft1,'Verify Block reduction optimization setting');
setInformation(ft1,['Check to confirm that the Block reduction ' ...
                                'check box is cleared.']);
% Add See Also section with references to applicable standards
docLinks{1}     = {['Reference DO331 Section MB.6.3.4.e - Source code ' ...
                           'is traceable to low-level requirements']};
% Review 'Block reduction' optimization
setRefLink(ft1,docLinks);
if strcmp(get_param(system,'BlockReduction'),'off')
    % 'Block reduction' is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft1,'Pass');
    setSubResultStatusText(ft1,['The ''Block reduction'' ' ...
                                    'check box is cleared']);
    ResultStatus = true;
else
    % 'Block reduction' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft1,'Warn');
    setSubResultStatusText(ft1,['The Block reduction ' ...
                                   'check box is selected.']);
    setRecAction(ft1,['Clear the ''Optimization > Block reduction''' ...
                ' check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft1;

% Title and description of second subcheck
ft2 = ModelAdvisor.FormatTemplate('ListTemplate');
setSubTitle(ft2,'Verify Conditional input branch execution setting');
setInformation(ft2,['Check to confirm that the ''Conditional input branch ' ...
                                   'execution'' check box is cleared.'])
% Add See Also section and references to applicable standards
docLinks{1} = {['Reference DO331 Section MB.6.4.4.2 - Test coverage ' ...
                           'of software structure is achieved']};
setRefLink(ft2,docLinks);

% Last subcheck, supress line
setSubBar(ft2,0);

% Check status of the 'Conditional input branch execution' check box
if strcmp(get_param(system,'ConditionallyExecuteInputs'),'off')
    % The 'Conditional input branch execution' check box is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft2,'Pass');
    setSubResultStatusText(ft2,['The ''Conditional input branch ' ...
                                  'execution'' check box is cleared.']);
else
    % 'Conditional input branch execution' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft2,'Warn');
    setSubResultStatusText(ft2,['The ''Conditional input branch ' ...
                         'execution'' check box is selected.']);
    setRecAction(ft2,['Clear the ''Optimization > Conditional input branch ' ...
            'execution'' check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft2; % Pass list template object to Model Advisor
mdladvObj.setCheckResultStatus(ResultStatus); % Set overall check status
% Enable Modify Settings button when check fails
mdladvObj.setActionEnable(~ResultStatus);
```

See Also

Topics

“Define Custom Model Advisor Checks”
“Create Model Advisor Checks”
“Customize the Configuration of the Model Advisor Overview”

setSubResultStatusText

Add text below status in result

Syntax

`setSubResultStatusText(`*`ft_obj`*`, `*`message`*`)`

Description

`setSubResultStatusText(`*`ft_obj`*`, `*`message`*`)` is an optional method that displays text below the status in the result. Use this method to describe the status.

Input Arguments

`ft_obj`

A handle to a template object.

`message`

A character vector or a handle to a formatting object that the Model Advisor displays below the status in the report.

Valid formatting objects are: `ModelAdvisor.Image`, `ModelAdvisor.LineBreak`, `ModelAdvisor.List`, `ModelAdvisor.Paragraph`, `ModelAdvisor.Table`, and `ModelAdvisor.Text`.

Examples

This example shows how to create a callback function for a custom check that finds and reports optimization settings. The check consists of two subchecks. The first reviews the **Block reduction** optimization setting and the second reviews the **Conditional input branch execution** optimization setting.

A check with subchecks includes the following items in the results:

- A description of what the overall check is reviewing.
- A title for the subcheck.
- A description of what the subcheck is reviewing.
- References to standards, if applicable.
- The status of the subcheck.
- A description of the status.
- Results for the subcheck.
- Recommended actions to take when the subcheck does not pass.
- A line between the subcheck results.

```
% Sample Check 3 Callback Function: Check with Subchecks and Actions
% Find and report optimization settings
```

```
function ResultDescription = OptmizationSettingCallback(system)
% Initialize variables
system =getfullname(system);
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(false); % Default check status is 'Warning'
ResultDescription = {};

% Format results in a list using Model Advisor Result Template API
% Create a list template object for first subcheck
ft1 = ModelAdvisor.FormatTemplate('ListTemplate');

% Description of check in results
setCheckText(ft1,'Check optimization settings');

% Title and description of first subcheck
setSubTitle(ft1,'Verify Block reduction optimization setting');
setInformation(ft1,['Check to confirm that the Block reduction ' ...
                                'check box is cleared.']);
% Add See Also section with references to applicable standards
docLinks{1}     = {['Reference DO331 Section MB.6.3.4.e - Source code ' ...
                           'is traceable to low-level requirements']};
% Review 'Block reduction' optimization
setRefLink(ft1,docLinks);
if strcmp(get_param(system,'BlockReduction'),'off')
    % 'Block reduction' is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft1,'Pass');
    setSubResultStatusText(ft1,['The ''Block reduction'' ' ...
                                    'check box is cleared']);
    ResultStatus = true;
else
    % 'Block reduction' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft1,'Warn');
    setSubResultStatusText(ft1,['The Block reduction ' ...
                                    'check box is selected.']);
    setRecAction(ft1,['Clear the ''Optimization > Block reduction''' ...
                ' check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft1;

% Title and description of second subcheck
ft2 = ModelAdvisor.FormatTemplate('ListTemplate');
setSubTitle(ft2,'Verify Conditional input branch execution setting');
setInformation(ft2,['Check to confirm that the ''Conditional input branch ' ...
                                    'execution'' check box is cleared.'])
% Add See Also section and references to applicable standards
docLinks{1} = {['Reference DO331 Section MB.6.4.4.2 - Test coverage ' ...
                           'of software structure is achieved']};
setRefLink(ft2,docLinks);

% Last subcheck, supress line
setSubBar(ft2,0);

% Check status of the 'Conditional input branch execution' check box
if strcmp(get_param(system,'ConditionallyExecuteInputs'),'off')
    % The 'Conditional input branch execution' check box is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft2,'Pass');
    setSubResultStatusText(ft2,['The ''Conditional input branch ' ...
                                'execution'' check box is cleared.']);
else
    % 'Conditional input branch execution' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft2,'Warn');
    setSubResultStatusText(ft2,['The ''Conditional input branch ' ...
                        'execution'' check box is selected.']);
    setRecAction(ft2,['Clear the ''Optimization > Conditional input branch ' ...
            'execution'' check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft2; % Pass list template object to Model Advisor
mdladvObj.setCheckResultStatus(ResultStatus); % Set overall check status
% Enable Modify Settings button when check fails
mdladvObj.setActionEnable(~ResultStatus);
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Display and Enable Check”

setSubscript

Specify subscripted text

Syntax

`setSubscript(text, mode)`

Description

`setSubscript(text, mode)` indicates whether to make `text` subscript.

Input Arguments

`text`	Instantiation of the `ModelAdvisor.Text` class
`mode`	A Boolean value indicating subscripted formatting of text: • `true` — Make the text subscript. • `false` — Do not make the text subscript.

Examples

```
t1 = ModelAdvisor.Text('This is some text');
setSubscript(t1, 'true');
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setSuperscript

Specify superscripted text

Syntax

`setSuperscript(text, mode)`

Description

`setSuperscript(text, mode)` indicates whether to make `text` superscript.

Input Arguments

`text`	Instantiation of the `ModelAdvisor.Text` class
`mode`	A Boolean value indicating superscripted formatting of text: • `true` — Make the text superscript. • `false` — Do not make the text superscript.

Examples

```
t1 = ModelAdvisor.Text('This is some text');
setSuperscript(t1, 'true');
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setSubTitle

Add title for subcheck in result

Syntax

`setSubTitle(ft_obj, title)`

Description

`setSubTitle(ft_obj, title)` is an optional method that adds a subcheck result title. Use this method when you create subchecks to distinguish between them in the result.

Input Arguments

ft_obj

A handle to a template object.

title

A character vector or a handle to a formatting object specifying the title of the subcheck.

Valid formatting objects are: `ModelAdvisor.Image`, `ModelAdvisor.LineBreak`, `ModelAdvisor.List`, `ModelAdvisor.Paragraph`, `ModelAdvisor.Table`, and `ModelAdvisor.Text`.

Examples

This example shows how to create a callback function for a custom check that finds and reports optimization settings. The check consists of two subchecks. The first reviews the **Block reduction** optimization setting and the second reviews the **Conditional input branch execution** optimization setting.

A check with subchecks includes the following items in the results:

- A description of what the overall check is reviewing.
- A title for the subcheck.
- A description of what the subcheck is reviewing.
- References to standards, if applicable.
- The status of the subcheck.
- A description of the status.
- Results for the subcheck.
- Recommended actions to take when the subcheck does not pass.
- A line between the subcheck results.

```
% Sample Check 3 Callback Function: Check with Subchecks and Actions
% Find and report optimization settings
```

```
function ResultDescription = OptmizationSettingCallback(system)
% Initialize variables
system =getfullname(system);
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);
mdladvObj.setCheckResultStatus(false); % Default check status is 'Warning'
ResultDescription = {};

% Format results in a list using Model Advisor Result Template API
% Create a list template object for first subcheck
ft1 = ModelAdvisor.FormatTemplate('ListTemplate');

% Description of check in results
setCheckText(ft1,'Check optimization settings');

% Title and description of first subcheck
setSubTitle(ft1,'Verify Block reduction optimization setting');
setInformation(ft1,['Check to confirm that the Block reduction ' ...
                                'check box is cleared.']);
% Add See Also section with references to applicable standards
docLinks{1}     = {['Reference DO331 Section MB.6.3.4.e - Source code ' ...
                           'is traceable to low-level requirements']};
% Review 'Block reduction' optimization
setRefLink(ft1,docLinks);
if strcmp(get_param(system,'BlockReduction'),'off')
    % 'Block reduction' is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft1,'Pass');
    setSubResultStatusText(ft1,['The ''Block reduction'' ' ...
                                    'check box is cleared']);
    ResultStatus = true;
else
    % 'Block reduction' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft1,'Warn');
    setSubResultStatusText(ft1,['The Block reduction ' ...
                                    'check box is selected.']);
    setRecAction(ft1,['Clear the ''Optimization > Block reduction''' ...
                ' check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft1;

% Title and description of second subcheck
ft2 = ModelAdvisor.FormatTemplate('ListTemplate');
setSubTitle(ft2,'Verify Conditional input branch execution setting');
setInformation(ft2,['Check to confirm that the ''Conditional input branch ' ...
                                    'execution'' check box is cleared.'])
% Add See Also section and references to applicable standards
docLinks{1} = {['Reference DO331 Section MB.6.4.4.2 - Test coverage ' ...
                           'of software structure is achieved']};
setRefLink(ft2,docLinks);

% Last subcheck, supress line
setSubBar(ft2,0);

% Check status of the 'Conditional input branch execution' check box
if strcmp(get_param(system,'ConditionallyExecuteInputs'),'off')
    % The 'Conditional input branch execution' check box is cleared
    % Set subresult status to 'Pass' and display text describing the status
    setSubResultStatus(ft2,'Pass');
    setSubResultStatusText(ft2,['The ''Conditional input branch ' ...
                                'execution'' check box is cleared.']);
else
    % 'Conditional input branch execution' is selected
    % Set subresult status to 'Warning' and display text describing the status
    setSubResultStatus(ft2,'Warn');
    setSubResultStatusText(ft2,['The ''Conditional input branch ' ...
                        'execution'' check box is selected.']);
    setRecAction(ft2,['Clear the ''Optimization > Conditional input branch ' ...
            'execution'' check box in the Configuration Parameters dialog box.']);
    ResultStatus = false;
end

ResultDescription{end+1} = ft2; % Pass list template object to Model Advisor
mdladvObj.setCheckResultStatus(ResultStatus); % Set overall check status
% Enable Modify Settings button when check fails
mdladvObj.setActionEnable(~ResultStatus);
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setTableInfo

Add data to table

Syntax

`setTableInfo(`*`ft_obj`*`, {`*`data`*`})`

Description

`setTableInfo(`*`ft_obj`*`, {`*`data`*`})` is an optional method that creates a table. *`ft_obj`* is a handle to a table template object. *`data`* is a cell array of character vectors or objects specifying the information in the body of the table. The Model Advisor creates hyperlinks to objects. If you do not add data to the table, the Model Advisor does not display the table in the result.

Note Before creating a table, you must specify column titles using the `setColTitle` method.

Examples

Create a table object, `ft`, add column titles, and add data to the table:

```
ft = ModelAdvisor.FormatTemplate('TableTemplate');
setColTitle(ft, {'Index', 'Block Name'});
setTableInfo(ft, {'1', 'Gain'});
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

setTableTitle

Package: `ModelAdvisor`

Add title to table in Model Advisor analysis results

Syntax

```
setTableTitle(ftObj,titleText)
```

Description

`setTableTitle(ftObj,titleText)` adds the title specified by `titleText` to the template object `ftObj`. The template object formats the Model Advisor analysis results.

Note The function `setTableTitle` is for formatting Model Advisor analysis results with Simulink Check. For more information, see “Simulink Check”.

For information on how to use tables in MATLAB, see “Create Tables and Assign Data to Them”.

Examples

Add a Title to a Table Template Object

Create a Model Advisor formatting template and add a table title.

Use `ModelAdvisor.FormatTemplate` to create a Model Advisor formatting template `ft` of `type 'TableTemplate'`.

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

Add a table title to the Model Advisor formatting template `ft`.

```
setTableTitle(ft, 'Title of the table');
```

Use `setTableTitle` in a check callback function in your `sl_customization` file to format your results. For an example of how to use `setTableTitle`, see “Format Model Advisor Results” on page 1-198.

For more information on how to format check results, see “Display and Enable Check”.

Input Arguments

`ftObj` — Template object

template object handle

Template object, specified as a handle to the template object.

`titleText` — Title text
character vector | formatting object handle

Title text specifying the title of the table, specified as a character vector or a handle to a valid formatting object.

Valid formatting objects:

- `ModelAdvisor.Image`
- `ModelAdvisor.LineBreak`
- `ModelAdvisor.List`
- `ModelAdvisor.Paragraph`
- `ModelAdvisor.Table`
- `ModelAdvisor.Text`

The title appears above the table. If you do not add data to the table, Model Advisor does not display the table or title in the result.

Example: 'Table of fonts and styles used in model'

See Also

`ModelAdvisor.FormatTemplate` | `setColTitles` | `setTableInfo`

Topics
“Define Custom Model Advisor Checks”
“Display and Enable Check”
“Format Model Advisor Results” on page 1-198

setType

Package: `ModelAdvisor`

Specify Model Advisor list type

Syntax

```
setType (obj,listType)
```

Description

`setType (obj,listType)` specifies the type of the `ModelAdvisor.List` object.

Examples

Format Model Advisor Results as List

1. Create a `ModelAdvisor.List` object.

```
subList = ModelAdvisor.List();
```

2. Specify the type as a numbered or bulleted list.

```
setType(subList, 'numbered')
```

3. Add items to the list.

```
addItem(subList, ModelAdvisor.Text('Sub entry 1', {'pass','bold'}));
addItem(subList, ModelAdvisor.Text('Sub entry 2', {'pass','bold'}));
```

Input Arguments

`obj` — Model Advisor list object

`ModelAdvisor.List`

Model Advisor list object for which you want to specify the type.

`listType` — Model Advisor list type

`numbered` | `bulleted`

Type of list that appears in the Model Advisor results.

Data Types: `char`

Version History

Introduced in R2006b

See Also

`ModelAdvisor.Check`

Topics
“Define Custom Model Advisor Checks”
“Create and Deploy a Model Advisor Custom Configuration”
“Customize the Configuration of the Model Advisor Overview”

setUnderlined

Underline text

Syntax

`setUnderlined(text, mode)`

Description

`setUnderlined(text, mode)` indicates whether to underline `text`.

Input Arguments

`text`	Instantiation of the `ModelAdvisor.Text` class
`mode`	A Boolean value indicating underlined formatting of text: • `true` — Underline the text. • `false` — Do not underline the text.

Examples

```
t1 = ModelAdvisor.Text('This is some text');
setUnderlined(t1, 'true');
```

See Also

“Customize the Configuration of the Model Advisor Overview”

Topics

“Create Model Advisor Checks”

slmetric.Engine class

Package: `slmetric`

(To be removed) Collect metric data on models or model components

Note `slmetric.Engine` will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and “Collect Model Maintainability Metrics Programmatically”.

Description

Use a `slmetric.Engine` object to collect metric data on models by calling `execute`. Use `getMetrics` to access the metric data and return an array of `slmetric.metric.ResultCollection` objects. This metric data is persistent in the simulation cache folder. Future instantiations of the `slmetric.Engine` object for the same model can access the cached metric data without regenerating the metric data.

Construction

`metric_engine = slmetric.Engine()` creates a metric engine object.

Properties

`AnalysisRoot` — Name of root model or subsystem on which to collect metric data

character vector

Name of root model or subsystem on which to collect metric data, as specified by the `slmetric.Engine.setAnalysisRoot` method. This property is read-only.

`AnalyzeLibraries` — Collect metric data on library linked subsystems in the model

1 (default)

Specify if the metric engine analyzes library-linked subsystems in the root model, including libraries inside referenced models under the root. Metric analysis does not include linked blocks to Simulink built-in libraries. Set this parameter to `false` or `0` to not include libraries in the metric analysis.

Data Types: `logical`

`ModelReferencesSimulationMode` — Collect metric data on referenced models under the root model

'AllModes' (default) | 'NormalModeOnly' | 'None'

Specify if the metric engine analyzes referenced models in your root model. Choose from these values:

Value	Description
`'None'`	Metric engine does not collect metric data for referenced models.

Value	Description
`'NormalModeOnly'`	Metric engine collects metric data only for referenced models running in normal simulation mode.
`'AllModes'`	Metric engine collects metric data for referenced models running in normal and accelerated simulation modes.

Data Types: `char`

Methods

execute	(To be removed) Collect metric data
getAnalysisRootMetric	Get metric data for one metric for analysis root only
getErrorLog	(To be removed) Get error log
getMetricDistribution	Get metric distribution
getMetrics	(To be removed) Access model metric data
getStatistics	Get statistics on metric data
setAnalysisRoot	Specify model or subsystem for metric analysis
exportMetrics	Export model metrics
getMetricMetaInformation	Obtain metric metadata

Examples

Collect and Access Metric Data for One Metric

Collect and access model metric data for the model `sldemo_mdlref_basic`.

Open the model.

```
openExample('sldemo_mdlref_basic');
```

Create an `slmetric.Engine` object and set the root in the model for analysis.

```
metric_engine = slmetric.Engine();

% Include referenced models and libraries in the analysis,
%     these properties are on by default
metric_engine.ModelReferencesSimulationMode = 'AllModes';
metric_engine.AnalyzeLibraries = 1;

setAnalysisRoot(metric_engine, 'Root',  'sldemo_mdlref_basic');
```

Collect model metric data

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

Get the model metric data that returns an array of `slmetric.metric.ResultCollection` objects, `res_col`.

```
res_col = getMetrics(metric_engine, 'mathworks.metrics.ExplicitIOCount');
```

Display the results for the `mathworks.metrics.ExplicitIOCount` metric.

```
for n=1:length(res_col)
    if res_col(n).Status == 0
        result = res_col(n).Results;

        for m=1:length(result)
            disp(['MetricID: ',result(m).MetricID]);
            disp(['  ComponentPath: ', result(m).ComponentPath]);
            disp(['  Value: ', num2str(result(m).Value)]);
            disp(['  AggregatedValue: ', num2str(result(m).AggregatedValue)]);
            disp(['  Measures: ', num2str(result(m).Measures)]);
            disp(['  AggregatedMeasures: ', num2str(result(m).AggregatedMeasures)]);
        end
    else
        disp(['No results for:', result(n).MetricID]);
    end
    disp(' ');
end
```

Here are the results:

```
MetricID: mathworks.metrics.ExplicitIOCount
  ComponentPath: sldemo_mdlref_basic
  Value: 3
  AggregatedValue: 4
  Measures: 0  3
  AggregatedMeasures: 3  3
MetricID: mathworks.metrics.ExplicitIOCount
  ComponentPath: sldemo_mdlref_basic/More Info
  Value: 0
  AggregatedValue: 0
  Measures: 0  0
  AggregatedMeasures: 0  0
MetricID: mathworks.metrics.ExplicitIOCount
  ComponentPath: sldemo_mdlref_counter
  Value: 4
  AggregatedValue: 4
  Measures: 3  1
  AggregatedMeasures: 3  1
```

For the `ComponentPath: sldemo_mdlref_basic`, the value is `3` because there are `3` outputs. The three outputs are in the second element of the `Measures` array. The `slmetric.metric.AggregationMode` is `Max`, so the `AggregatedValue` is `4` which is the number of inputs and outputs to `sldemo_mdlref_counter`. The `AggregratedMeasures` array contains the maximum number of inputs and outputs for a component or subcomponent.

Version History

Introduced in R2016a

Warns

Warns starting in R2022a

The `slmetric.Engine` class issues a warning that it will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and “Collect Model Maintainability Metrics Programmatically”.

`ModelReferencesSimulationMode` property replaces `AnalyzeModelReferences` property

Warns starting in R2019a

The `ModelReferencesSimulationMode` property replaces the `AnalyzeModelReferences` property.

Use the `ModelReferencesSimulationMode` property to specify whether referenced models are included in the metric analysis.

`AnalyzeModelReferences` Value	**`ModelReferencesSimulationMode` Value**	**Description**
`0`	`'None'`	Metric engine does not collect metric data for referenced models.
`1`	`'NormalModeOnly'`	Metric engine collects metric data only for referenced models running in normal simulation mode.
Not applicable	`'AllModes'`	Metric engine collects metric data for referenced models running in normal and accelerated simulation modes.

See Also

`slmetric.metric.Result` | `slmetric.metric.ResultCollection` | `slmetric.metric.getAvailableMetrics`

Topics

“Collect Model Metrics Programmatically”

“Model Metrics” on page 2-299

slmetric.metric.getAvailableMetrics

Package: `slmetric.metric`

(To be removed) Obtain available metrics

Note `slmetric.metric.getAvailableMetrics` will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and "Collect Model Maintainability Metrics Programmatically".

Syntax

```
IDs = slmetric.metric.getAvailableMetrics()
[IDs,props] = slmetric.metric.getAvailableMetrics()
```

Description

`IDs = slmetric.metric.getAvailableMetrics()` get metric identifiers for available metrics in the metric engine.

`[IDs,props] = slmetric.metric.getAvailableMetrics()` get metric identifiers and properties.

Examples

Obtain Available Metric IDs for Model

This example shows how to obtain the available model metric IDs.

```
ID = slmetric.metric.getAvailableMetrics()
```

```
ID =

  26×1 cell array

    {'mathworks.metrics.CloneContent'                       }
    {'mathworks.metrics.CloneDetection'                     }
    {'mathworks.metrics.CyclomaticComplexity'               }
    {'mathworks.metrics.DescriptiveBlockNames'              }
    {'mathworks.metrics.DiagnosticWarningsCount'            }
    {'mathworks.metrics.ExplicitIOCount'                    }
    {'mathworks.metrics.FileCount'                          }
    {'mathworks.metrics.IOCount'                            }
    {'mathworks.metrics.LayerSeparation'                    }
    {'mathworks.metrics.LibraryContent'                     }
    {'mathworks.metrics.LibraryLinkCount'                   }
    {'mathworks.metrics.MatlabCodeAnalyzerWarnings'         }
    {'mathworks.metrics.MatlabFunctionCount'                }
    {'mathworks.metrics.MatlabLOCCount'                     }
```

```
    {'mathworks.metrics.ModelAdvisorCheckCompliance.hisl_do178'}
    {'mathworks.metrics.ModelAdvisorCheckCompliance.maab'      }
    {'mathworks.metrics.ModelAdvisorCheckIssues.hisl_do178'    }
    {'mathworks.metrics.ModelAdvisorCheckIssues.maab'          }
    {'mathworks.metrics.ModelFileCount'                        }
    {'mathworks.metrics.ParameterCount'                        }
    {'mathworks.metrics.SimulinkBlockCount'                    }
    {'mathworks.metrics.StateflowChartCount'                   }
    {'mathworks.metrics.StateflowChartObjectCount'             }
    {'mathworks.metrics.StateflowLOCCount'                     }
    {'mathworks.metrics.SubSystemCount'                        }
    {'mathworks.metrics.SubSystemDepth'                        }
```

Obtain Available Metric IDs and Metric Properties

This example shows how to obtain the available model metric properties.

```
[ID,PROPS]=slmetric.metric.getAvailableMetrics()
```

```
ID =

  26×1 cell array

    {'mathworks.metrics.CloneContent'                          }
    {'mathworks.metrics.CloneDetection'                        }
    {'mathworks.metrics.CyclomaticComplexity'                  }
    {'mathworks.metrics.DescriptiveBlockNames'                 }
    {'mathworks.metrics.DiagnosticWarningsCount'               }
    {'mathworks.metrics.ExplicitIOCount'                       }
    {'mathworks.metrics.FileCount'                             }
    {'mathworks.metrics.IOCount'                               }
    {'mathworks.metrics.LayerSeparation'                       }
    {'mathworks.metrics.LibraryContent'                        }
    {'mathworks.metrics.LibraryLinkCount'                      }
    {'mathworks.metrics.MatlabCodeAnalyzerWarnings'            }
    {'mathworks.metrics.MatlabFunctionCount'                   }
    {'mathworks.metrics.MatlabLOCCount'                        }
    {'mathworks.metrics.ModelAdvisorCheckCompliance.hisl_do178'}
    {'mathworks.metrics.ModelAdvisorCheckCompliance.maab'      }
    {'mathworks.metrics.ModelAdvisorCheckIssues.hisl_do178'    }
    {'mathworks.metrics.ModelAdvisorCheckIssues.maab'          }
    {'mathworks.metrics.ModelFileCount'                        }
    {'mathworks.metrics.ParameterCount'                        }
    {'mathworks.metrics.SimulinkBlockCount'                    }
    {'mathworks.metrics.StateflowChartCount'                   }
    {'mathworks.metrics.StateflowChartObjectCount'             }
    {'mathworks.metrics.StateflowLOCCount'                     }
    {'mathworks.metrics.SubSystemCount'                        }
    {'mathworks.metrics.SubSystemDepth'                        }


PROPS =

  1×26 struct array with fields:

    Name
```

```
Description
IsBuiltIn
Version
```

Output Arguments

`IDs` — Metric identifiers
cell array of character vectors

Metric identifiers in the metric engine.

`props` — Metric properties
structure array

Metric properties, returned as a structure array with the following fields:

Name	Name of the metric algorithm.
Description	Description of the metric algorithm.
IsBuiltIn	Boolean indicating if the metric is included with Simulink Check.
Version	Metric algorithm version.

Data Types: `struct`

Version History

Introduced in R2016a

Warns
Warns starting in R2022a

The `slmetric.Engine` API will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and “Collect Model Maintainability Metrics Programmatically”.

See Also

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

slmetric.metric.Result

(To be removed) Metric data for specified model component and metric algorithm

Note `slmetric.metric.Result` will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and “Collect Model Maintainability Metrics Programmatically”.

Description

An `slmetric.metric.Result` object contains the metric data for a specified model component and metric algorithm.

Creation

Syntax

```
metric_result = slmetric.metric.Result
```

Description

`metric_result = slmetric.metric.Result` creates a handle to a metric results object.

Alternatively, if you collect results in an `slmetric.metric.ResultCollection` object, the `Results` property of the collection object returns the collected `slmetric.metric.Result` objects in an array.

Properties

`ID` — Numeric identifier

integer

This property is read-only.

Unique numeric identifier for the metric result object, returned as an integer.

Data Types: `uint64`

`ComponentID` — Component identifier

character vector

Unique identifier of the component object for which the metric is calculated, specified as a character vector. Use `ComponentID` to trace the generated result object to the analyzed component. Set the `ComponentID` or `ComponentPath` properties by using the `algorithm` method.

Example: `'sldemo_mdlref_basic'`

Data Types: `char`

`ComponentPath` — Component path

character vector

Component path for which metric is calculated, specified as a character vector. Use `ComponentPath` as an alternative to setting the `ComponentID` property. The metric engine converts the `ComponentPath` to a `ComponentID`. Set the `ComponentID` or `ComponentPath` properties by using the `algorithm` method.

Example: `'vdp/More Info/Model Info/EmptySubsystem'`

Data Types: `char`

`MetricID` — Metric identifier

character vector

Metric identifier for “Model Metrics” on page 2-299 or custom model metrics that you create, specified as a character vector. You can get metric identifiers by calling `slmetric.metric.getAvailableMetrics`.

Example: `'mathworks.metrics.SimulinkBlockCount'`

Data Types: `char`

`Value` — Metric value

double

Metric scalar value generated by the algorithm for the metric specified by `MetricID` and the component specified by `ComponentID`, specified as a double.

If the algorithm does not specify a metric scalar value, the value of `Value` is `NaN`. For example, suppose you collect metric data for a model that contains a Stateflow Chart. For the `StateflowChartObjectCount` metric, the `Value` property of the model `slmetric.metric.Result` object is `NaN` because the model itself cannot have Stateflow objects. The `AggregatedValue` property of the model `slmetric.metric.Result` object contains the total number of Stateflow objects in the chart.

Data Types: `double`

`AggregatedValue` — Aggregated metric value

double

This property is read-only.

Metric value aggregated across the model hierarchy, returned as a double. The metric engine implicitly aggregates the metric values based on the `AggregationMode`. If the `Value` property is `NaN` for all components, the `AggregatedValue` is zero.

Data Types: `double`

`Measures` — Metric measures

double array

Metric measures, specified by the metric algorithm, specified as a double array. Metric measures contain detailed information about the metric value. For example, for a metric that counts the number of blocks per subsystem, you can specify measures that contain the number of virtual and nonvirtual blocks. The metric value is the sum of the virtual and nonvirtual block count.

Set this property by using the `slmetric.metric.Metric.algorithm` method.

Data Types: `double`

`AggregatedMeasures` — Aggregated metric measures
double array

This property is read-only.

Metric measures value aggregated across the model hierarchy, returned as a double array. The metric engine implicitly aggregates the metric measure values based on the `AggregationMode`.

Data Types: `double`

`Details` — Metric result details
array of `slmetric.metric.ResultDetail` objects

Details about what the metric engine counts for the `Value` property, specified as an array of `slmetric.metric.ResultDetail` objects.

`Category` — Metric data category based on thresholding criteria
'Compliant' | 'NonCompliant' | 'Warning' | 'Uncategorized'

This property is read-only.

Metric data category, returned as one of these four categories:

- 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.
- Uncategorized — Metric data that does not have threshold values set.

`Classifications` — Metric data category and thresholding criteria
`slmetric.config.ResultClassification` object

Metric data category and the ranges that correspond to each category, specified as an `slmetric.config.ResultClassification` object. This property is empty if no threshold values are set.

`UserData` — User data
character vector

User data optionally provided by the metric algorithm, specified as a character vector.

Data Types: `char`

Examples

Collect and Access Metric Data for One Metric

This example shows how to collect and access metric data for the model `sldemo_mdlref_basic`.

Open the `sldemo_mdlref_basic` model.

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

Create an `slmetric.Engine` object and set the root in the model for analysis.

```
metric_engine = slmetric.Engine();

% Include referenced models and libraries in the analysis,
%   these properties are on by default
metric_engine.ModelReferencesSimulationMode = 'AllModes';
metric_engine.AnalyzeLibraries = 1;

setAnalysisRoot(metric_engine, 'Root', 'sldemo_mdlref_basic')
```

Collect model metric data.

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

Return the model metric data as an array of `slmetric.metric.ResultCollection` objects and assign it to `res_col`.

```
res_col = getMetrics(metric_engine, 'mathworks.metrics.ExplicitIOCount');
```

Display the results for the `mathworks.metrics.ExplicitIOCount` metric.

```
for n=1:length(res_col)
    if res_col(n).Status == 0
        result = res_col(n).Results;

        for m=1:length(result)
            disp(['MetricID: ',result(m).MetricID]);
            disp(['  ComponentPath: ',result(m).ComponentPath]);
            disp(['  Value: ', num2str(result(m).Value)]);
            disp(['  AggregatedValue: ', num2str(result(m).AggregatedValue)]);
            disp(['  Measures: ', num2str(result(m).Measures)]);
            disp(['  AggregatedMeasures: ', num2str(result(m).AggregatedMeasures)]);
        end
    else
        disp(['No results for:', result(n).MetricID]);
    end
    disp(' ');
end
```

For `ComponentPath: sldemo_mdlref_basic`, the value is `3` because there are three outputs. The three outputs are in the second element of the `Measures` array. The `slmetric.metric.AggregationMode` is `Max`, so the `AggregatedValue` is `4`, which is the number of inputs and outputs to `sldemo_mdlref_counter`. The `AggregatedMeasures` array contains the maximum number of inputs and outputs for a component or subcomponent.

Version History

Introduced in R2016a

Warns

Warns starting in R2022a

The `slmetric.Engine` API will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For

more information, see `metric.Engine` and “Collect Model Maintainability Metrics Programmatically”.

See Also

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

Topics

“Collect Model Metrics Programmatically”

“Model Metrics” on page 2-299

slmetric.metric.ResultCollection

(To be removed) Metric data for specified model metric

Note `slmetric.metric.ResultCollection` will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and “Collect Model Maintainability Metrics Programmatically”.

Description

An `slmetric.metric.ResultCollection` object contains the metric data for a specific model metric.

Creation

To create an `slmetric.metric.ResultCollection` object, use `getMetrics` on an `slmetric.Engine` object. `getMetrics` returns an array of result collection objects for all metrics that the metric engine executed.

Properties

`MetricID` — Metric identifier

character vector

Metric identifier for a MathWorks metric or a custom metric, specified as a character vector. You can get metric identifiers by calling `slmetric.metric.getAvailableMetrics`.

Example: `'mathworks.metrics.SimulinkBlockCount'`

`Status` — Metric execution status

integer

This property is read-only.

Status code of the metric execution, returned as an integer.

Integer	Status
1	No result. The metric algorithm is not applicable to the analyzed system. The components analyzed by the metric were not found, or the metric has a compile requirement cannot be executed on the library model.
0	Result collected.
-1	No result. Error executing metric.
-2	No result available from previous run.
-3	No result. Compilation error.

Integer	Status
-4	Empty result. Missing prerequisite.

`Category` — Metric data category based on thresholding criteria
`'Compliant'` | `'NonCompliant'` | `'Warning'` | `'Uncategorized'`

This property is read-only.

Metric data category, returned as one of these four categories:

- 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.
- Uncategorized — Metric data that has no threshold values.

If at least one component is `NonCompliant`, this property returns `NonCompliant`. If at least one component is `Warning` and no components are `NonCompliant`, this property returns `Warning`. If all components are `Compliant`, this property returns `Compliant`.

`Outdated` — Determine if metric data is current
logical

This property is read-only.

Whether metric data is current, returned as `true` or `false`. If `true`, the metric data is out-of-date because the model or source files have changed.

`Results` — Metric data collected for executing one or more metrics
array of `slmetric.metric.Result` objects

This property is read-only.

Metric data collected when you call the `execute` method for one or more metrics, returned as an array of `slmetric.metric.Result` objects.

Examples

Collect and Access Metric Data for One Metric

This example shows how to collect and access metric data for the model `sldemo_mdlref_basic`.

Open the `sldemo_mdlref_basic` model.

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

Create an `slmetric.Engine` object and set the root in the model for analysis.

```
metric_engine = slmetric.Engine();
```

```
% Include referenced models and libraries in the analysis,
%   these properties are on by default
metric_engine.ModelReferencesSimulationMode = 'AllModes';
metric_engine.AnalyzeLibraries = 1;
```

```
setAnalysisRoot(metric_engine, 'Root', 'sldemo_mdlref_basic')
```

Collect model metric data.

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

Return the model metric data as an array of `slmetric.metric.ResultCollection` objects and assign it to `res_col`.

```
res_col = getMetrics(metric_engine, 'mathworks.metrics.ExplicitIOCount');
```

Display the results for the `mathworks.metrics.ExplicitIOCount` metric.

```
for n=1:length(res_col)
    if res_col(n).Status == 0
        result = res_col(n).Results;

        for m=1:length(result)
            disp(['MetricID: ',result(m).MetricID]);
            disp(['  ComponentPath: ',result(m).ComponentPath]);
            disp(['  Value: ', num2str(result(m).Value)]);
            disp(['  AggregatedValue: ', num2str(result(m).AggregatedValue)]);
            disp(['  Measures: ', num2str(result(m).Measures)]);
            disp(['  AggregatedMeasures: ', num2str(result(m).AggregatedMeasures)]);
        end
    else
        disp(['No results for:', result(n).MetricID]);
    end
    disp(' ');
end
```

For `ComponentPath: sldemo_mdlref_basic`, the value is `3` because there are three outputs. The three outputs are in the second element of the `Measures` array. The `slmetric.metric.AggregationMode` is `Max`, so the `AggregatedValue` is `4`, which is the number of inputs and outputs to `sldemo_mdlref_counter`. The `AggregatedMeasures` array contains the maximum number of inputs and outputs for a component or subcomponent.

Version History

Introduced in R2016a

Warns

Warns starting in R2022a

The `slmetric.Engine` API will be removed in a future release. For size, architecture, and complexity metrics, use the `metric.Engine` API and the model maintainability metrics instead. For more information, see `metric.Engine` and “Collect Model Maintainability Metrics Programmatically”.

See Also

`slmetric.Engine` | `slmetric.metric.Result` | `slmetric.metric.getAvailableMetrics`

Attributes property

Class: `ModelAdvisor.ListViewParameter`
Package: `ModelAdvisor`

Attributes to display in Model Advisor Report Explorer

Values

Cell array

Default: `{}` (empty cell array)

Description

The `Attributes` property specifies the attributes to display in the center pane of the Model Advisor Results Explorer.

Examples

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

CallbackContext property

Specify when to run check

Values

`'PostCompileForCodegen'`
`'PostCompile'`
`'None'` (default)

Description

The `CallbackContext` property specifies the context for checking the model or subsystem.

`'None'`	No special requirements for the model before checking.
`'Postcompile'`	The model is compiled and simulated. Use this option for checks that analyze simulated models.
`'PostCompileForCodegen'`	The model is compiled for code generation, but is not simulated. Use this option to check the code generation readiness of a model and to analyze both active and inactive variant paths. To analyze both active and inactive variants, you must set the **System target file** configuration parameter to `ert.tlc`. Note, this property value is not supported for custom edit-time checks.

CallbackHandle property

Callback function handle for check

Values

Function handle.

An empty handle `[ ]` is the default.

Description

The `CallbackHandle` property specifies the handle to the check callback function.

CallbackStyle property

Callback function type

Values

`'DetailStyle'` (recommended)
`'StyleOne'` (default)
`'StyleTwo'`
`'StyleThree'`

Description

The `CallbackStyle` property specifies the type of the callback function.

The `'DetailStyle'` check callback function is for detailed results collection. This type of callback function is the recommended one and enables you to use the default format of the Model Advisor report. The keyword for this callback function is `DetailStyle`. The check definition requires this keyword. With the default format, 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. This style also allows you to view results by block, subsystem, or recommended action.

The detailed results collection callback function takes the following arguments.

Argument	I/O Type	Description
`system`	Input	Path to the model or subsystem analyzed by the Model Advisor.
`CheckObj`	Input	`ModelAdvisor.Check` object

To indicate whether a model has passed or failed the check, or to recommend fixing an issue using the Model Advisor Formatting APIs to format results, use the *simple check callback function*. The keyword for this callback function is `StyleOne`. The check definition requires this keyword.

The simple check callback function takes the following arguments.

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

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

The detailed callback function takes the following arguments.

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

To automatically display hyperlinks for every object returned by the check, use the *callback function with hyperlinked results*. The keyword for this type of callback function is `StyleThree`. The check definition requires this keyword.

This callback function takes the following arguments.

Argument	I/O Type	Description
`system`	Input	Path to the model or system analyzed by the Model Advisor.
`ResultDescription`	Output	Cell array of MATLAB character vectors that supports the Model Advisor Formatting API calls or embedded HTML tags for text formatting.
`ResultDetails`	Output	Cell array of cell arrays, each of which contains one or more Simulink objects such as blocks, ports, lines, and Stateflow charts. The objects must be in the form of a handle or Simulink path. The `ResultDetails` cell array must be the same length as the `ResultDescription` cell array.

See Also

`ModelAdvisor.Check`

Topics

“Fix a Model to Comply with Conditions that You Specify with the Model Advisor”
“Create the Check Definition Function for a Pass/Fail Check with No Fix Action”

ErrorSeverity property

Set severity of check failure

Values

`0`
`1`

Description

The `ErrorSeverity` property is an integer value that specifies whether the check is marked as a warning or failure when the check flags an issue in your model. This property is the programmatic equivalent of using the **Check result when issues are flagged** options in the Model Advisor Configuration Editor.

`0`	Marks the check as a warning.
`1`	Marks the check as a failure.

Example

In this sample code for defining the properties of a custom check, the value for `ErrorSeverity` is `"1"`. Therefore, if a violation of this check is flagged in a model, the check is marked as Fail in the results.

```
% 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.ErrorSeverity = 1;
```

EmitInputParametersToReport property

Display check input parameters in the Model Advisor report

Values

`true` (default)
`false`

Description

The `EmitInputParametersToReport` property specifies the display of check input parameters in the Model Advisor report.

`true`	Display check input parameters in the Model Advisor report
`false`	Do not display check input parameters in the Model Advisor report

Data property

Class: `ModelAdvisor.ListViewParameter`
Package: `ModelAdvisor`

Objects in Model Advisor Result Explorer

Values

Array of Simulink objects

Default: `[]` (empty array)

Description

The `Data` property specifies the objects displayed in the Model Advisor Result Explorer.

Examples

```
% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
myLVParam.Data = get_param(searchResult,'object')';
```

Description property

Class: `ModelAdvisor.Action`
Package: `ModelAdvisor`

Message in **Action** box

Values

Character vector

Default: `''` (empty character vector)

Description

The `Description` property specifies the message displayed in the Action box.

Examples

```
% define action (fix) operation
myAction = ModelAdvisor.Action;
%Specify a callback function for the action
myAction.setCallbackFcn(@sampleActionCB);
myAction.Name='Fix block fonts';
myAction.Description=...
   'Click the button to update all blocks with specified font';
```

Description property

Class: `ModelAdvisor.FactoryGroup`
Package: `ModelAdvisor`

Description of folder

Values

Character vector

Default: `''` (empty character vector)

Description

The `Description` property provides information about the folder. Details about the folder are displayed in the right pane of the Model Advisor.

Examples

```
% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
rec.Description='Sample Factory Group';
```

Description property

Class: `ModelAdvisor.Group`
Package: `ModelAdvisor`

Description of folder

Values

Character vector

Default: `''` (empty character vector)

Description

The `Description` property provides information about the folder. Details about the folder are displayed in the right pane of the Model Advisor.

Examples

```
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');
MAG.Description='This is my group';
```

Description property

Class: `ModelAdvisor.InputParameter`
Package: `ModelAdvisor`

Description of input parameter

Values

Character vector.

Default: `''` (empty character vector)

Description

The `Description` property specifies a description of the input parameter. Details about the check are displayed in the right pane of the Model Advisor.

Examples

```
% define input parameters
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
```

Description property

Class: `ModelAdvisor.Task`
Package: `ModelAdvisor`

Description of task

Values

Character vector

Default: `''` (empty character vector)

Description

The `Description` property is a description of the task that the Model Advisor displays in the **Analysis** box.

When adding checks as tasks, the Model Advisor uses the task `Description` property instead of the check `TitleTips` property.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.DisplayName='Example task 1';
MAT1.Description='This is the first example task.'

MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');
MAT2.DisplayName='Example task 2';
MAT2.Description='This is the second example task.'

MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
MAT3.DisplayName='Example task 3';
MAT3.Description='This is the third example task.'
```

DisplayName property

Class: `ModelAdvisor.FactoryGroup`
Package: `ModelAdvisor`

Name of folder

Values

Character vector

Default: `''` (empty character vector)

Description

The `DisplayName` specifies the name of the folder that is displayed in the Model Advisor.

Examples

```
% --- sample factory group
rec = ModelAdvisor.FactoryGroup('com.mathworks.sample.factorygroup');
rec.DisplayName='Sample Factory Group';
```

DisplayName property

Class: `ModelAdvisor.Group`
Package: `ModelAdvisor`

Name of folder

Values

Character vector

Default: `''` (empty character vector)

Description

The `DisplayName` specifies the name of the folder that is displayed in the Model Advisor.

Examples

```
MAG = ModelAdvisor.Group('com.mathworks.sample.GroupSample');
MAG.DisplayName='My Group';
```

DisplayName property

Class: `ModelAdvisor.Task`
Package: `ModelAdvisor`

Name of task

Values

Character vector

Default: `''` (empty character vector)

Description

The `DisplayName` property specifies the name of the task. The Model Advisor displays each custom task in the tree using the name of the task. Therefore, you should specify a unique name for each task. When you specify the same name for multiple tasks, the Model Advisor generates a warning.

When adding checks as tasks, the Model Advisor uses the task `DisplayName` property instead of the check `Title` property.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.DisplayName='Example task with input parameter and auto-fix ability';

MAT2 = ModelAdvisor.Task('com.mathworks.sample.TaskSample2');
MAT2.DisplayName='Example task 2';

MAT3 = ModelAdvisor.Task('com.mathworks.sample.TaskSample3');
MAT3.DisplayName='Example task 3';
```

Enable property

Indicate whether user can enable or disable check

Values

`true` (default)
`false`

Description

The `Enable` property specifies whether the user can enable or disable the check.

`true`	Display the check box control
`false`	Hide the check box control

Enable property

Class: `ModelAdvisor.Task`
Package: `ModelAdvisor`

Indicate if user can enable and disable task

Values

`true` (default)
`false`

Description

The `Enable` property specifies whether the user can enable or disable a task.

`true` (default)	Display the check box control for task
`false`	Hide the check box control for task

When adding checks as tasks, the Model Advisor uses the task `Enable` property instead of the check `Enable` property.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.Enable = false;
```

Entries property

Class: ModelAdvisor.InputParameter
Package: ModelAdvisor

Drop-down list entries

Values

Depends on the value of the Type property.

Description

The Entries property is valid only when the Type property is one of the following:

- Enum
- ComboBox
- PushButton

Examples

```
inputParam3 = ModelAdvisor.InputParameter;
inputParam3.Name='Valid font';
inputParam3.Type='Combobox';
inputParam3.Description='sample tooltip';
inputParam3.Entries={'Arial', 'Arial Black'};
```

ID property

Identifier for check

Values

Character vector

Default: `''` (empty character vector)

Description

The `ID` property specifies a permanent, unique identifier for the check. Note the following about the `ID` property:

- You must specify this property.
- The value of `ID` must remain constant.
- The Model Advisor generates an error if `ID` is not unique.
- Tasks and factory group definitions must refer to checks by `ID`.

ID property

Class: `ModelAdvisor.FactoryGroup`
Package: `ModelAdvisor`

Identifier for folder

Values

Character vector

Description

The `ID` property specifies a permanent, unique identifier for the folder.

Note

- You must specify this field.
- The value of `ID` must remain constant.
- The Model Advisor generates an error if `ID` is not unique.
- Group definitions must refer to other groups by `ID`.

ID property

Class: `ModelAdvisor.Group`
Package: `ModelAdvisor`

Identifier for folder

Values

Character vector

Description

The `ID` property specifies a permanent, unique identifier for the folder.

Note

- You must specify this field.
- The value of `ID` must remain constant.
- The Model Advisor generates an error if `ID` is not unique.
- Group definitions must refer to other groups by `ID`.

ID property

Class: `ModelAdvisor.Task`
Package: `ModelAdvisor`

Identifier for task

Values

Character vector

Default: `''` (empty character vector)

Description

The `ID` property specifies a permanent, unique identifier for the task.

Note

- The Model Advisor automatically assigns a unique identifier to `ID` if you do not specify it.
- The value of `ID` must remain constant.
- The Model Advisor generates an error if `ID` is not unique.
- Group definitions must refer to tasks using `ID`.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.ID='Task_ID_1234';
```

LicenseName property

Product license names required to display and run check

Values

Cell array of product license names
`{}`(empty cell array) (default)

Description

The `LicenseName` property specifies a cell array of names for product licenses required to display and run the check.

When the Model Advisor starts, it tests whether the product license exists. If you do not meet the license requirements, the Model Advisor does not display the check.

The Model Advisor performs a checkout of the product licenses when you run the custom check. If you do not have the product licenses available, you see an error message that the required license is not available.

Tip To find the text for license strings, type `help license` at the MATLAB command line.

LicenseName property

Class: `ModelAdvisor.Task`
Package: `ModelAdvisor`

Product license names required to display and run task

Values

Cell array of product license names

Default: `{}` (empty cell array)

Description

The `LicenseName` property specifies a cell array of names for product licenses required to display and run the check.

When the Model Advisor starts, it tests whether the product license exists. If you do not meet the license requirements, the Model Advisor does not display the check.

The Model Advisor performs a checkout of the product licenses when you run the custom check. If you do not have the product licenses available, you see an error message that the required license is not available.

If you specify `ModelAdvisor.Check.LicenseName`, the Model Advisor displays the check when the union of both properties is true.

Tip To find the text for license strings, type `help license` at the MATLAB command line.

MAObj property

Class: `ModelAdvisor.FactoryGroup`
Package: `ModelAdvisor`

Model Advisor object

Values

Handle to a `Simulink.ModelAdvisor` object

Description

The `MAObj` property specifies a handle to the current Model Advisor object.

MAObj property

Class: `ModelAdvisor.Group`
Package: `ModelAdvisor`

Model Advisor object

Values

Handle to `Simulink.ModelAdvisor` object

Description

The `MAObj` property specifies a handle to the current Model Advisor object.

MAObj property

Class: `ModelAdvisor.Task`
Package: `ModelAdvisor`

Model Advisor object

Values

Handle to a `Simulink.ModelAdvisor` object

Description

The `MAObj` property specifies the current Model Advisor object.

When adding checks as tasks, the Model Advisor uses the task `MAObj` property instead of the check `MAObj` property.

Name property

Class: `ModelAdvisor.Action`
Package: `ModelAdvisor`

Action button label

Values

Character vector

Default: `''` (empty character vector)

Description

The `Name` property specifies the label for the action button. This property is required.

Examples

```
% define action (fix) operation
myAction = ModelAdvisor.Action;
%Specify a callback function for the action
myAction.setCallbackFcn(@sampleActionCB);
myAction.Name='Fix block fonts';
```

Name property

Class: `ModelAdvisor.InputParameter`
Package: `ModelAdvisor`

Input parameter name

Values

Character vector.

Default: `''` (empty character vector)

Description

The `Name` property specifies the name of the input parameter in the custom check.

Examples

```
inputParam2 = ModelAdvisor.InputParameter;
inputParam2.Name = 'Standard font size';
inputParam2.Value='12';
inputParam2.Type='String';
inputParam2.Description='sample tooltip';
```

Name property

Class: `ModelAdvisor.ListViewParameter`
Package: `ModelAdvisor`

Drop-down list entry

Values

Character vector

Default: `''` (empty character vector)

Description

The `Name` property specifies an entry in the **Show** drop-down list in the Model Advisor Result Explorer.

Examples

```
% define list view parameters
myLVParam = ModelAdvisor.ListViewParameter;
myLVParam.Name = 'Invalid font blocks'; % the name appeared at pull down filter
```

Result property

Results cell array

Values

Cell array

Default: `{}` (empty cell array)

Description

The `Result` property specifies the cell array for storing the results that are returned by the callback function specified in `CallbackHandle`.

Tip To set the icon associated with the check, use the `Simulink.ModelAdvisor` `setCheckResultStatus` and `setCheckErrorSeverity` methods.

SupportExclusion property

Set to support exclusions

Values

Boolean value specifying that the check supports exclusions.
`true`. The check supports exclusions.
`false` (default). The check does not support exclusions.

Description

The `SupportExclusion` property specifies whether the check supports exclusions.

`true`	Check supports exclusions.
`false`	Check does not support exclusions.

Examples

```
% specify that a check supports exclusions
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.SupportExclusion = true;
```

SupportLibrary property

Set to support library models

Values

Boolean value specifying that the check supports library models.
`true`. The check supports library models.
`false` (default). The check does not support library models.

Description

The `SupportLibrary` property specifies whether the check supports library models.

`true`	Check supports library models.
`false`	Check does not support library models.

Examples

```
% specify that a check supports library models
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.SupportLibrary = true;
```

Title property

Name of check

Values

Character vector

Default: `''` (empty character vector)

Description

The `Title` property specifies the name of the check in the Model Advisor. The Model Advisor displays each custom check in the tree using the title of the check. Therefore, you should specify a unique title for each check. When you specify the same title for multiple checks, the Model Advisor generates a warning.

Examples

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.Title = 'Check Simulink block font';
```

TitleTips property

Description of check

Values

Character vector

Default: `''` (empty character vector)

Description

The `TitleTips` property specifies a description of the check. Details about the check are displayed in the right pane of the Model Advisor.

Examples

```
rec = ModelAdvisor.Check('com.mathworks.sample.Check1');
rec.Title = 'Check Simulink block font';
rec.TitleTips = 'Example style three callback';
```

Type property

Class: `ModelAdvisor.InputParameter`
Package: `ModelAdvisor`

Input parameter type

Values

character vector

Default: `''`

Description

The `Type` property specifies the type of input parameter.

Use the `Type` property with the `Value` and `Entries` properties to define input parameters.

Valid values are listed in the following table.

Type	Data Type	Default Value	Description
`Bool`	Boolean	`false`	A check box
`ComboBox`	Cell array	First entry in the list	A drop-down menu • Use `Entries` to define the entries in the list. • Use `Value` to indicate a specific entry in the menu or to enter a value not in the list.
`Enum`	Cell array	First entry in the list	A drop-down menu • Use `Entries` to define the entries in the list. • Use `Value` to indicate a specific entry in the list.
`PushButton`	N/A	N/A	A button When you click the button, the callback function specified by `Entries` is called.
`String`	Character vector	`''`	A text box

Examples

```
% define input parameters
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Skip font checks.';
```

```
inputParam1.Type = 'Bool';
inputParam1.Value = false;
```

Advisor.authoring.DataFile.validate

Validate XML data file used for model configuration check

Syntax

```
msg = Advisor.authoring.DataFile.validate(dataFile)
```

Description

`msg = Advisor.authoring.DataFile.validate(dataFile)` validates the syntax of the XML data file used for model configuration checks.

Input Arguments

`dataFile`	XML data file name (character vector)

Examples

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

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

See Also

`Advisor.authoring.CustomCheck` |
`Advisor.authoring.generateConfigurationParameterDataFile`

Topics

“Create Model Advisor Check for Model Configuration Parameters”

Advisor.authoring.CompositeConstraint class

Package: `Advisor.authoring`

Create a Model Advisor constraint that checks for multiple constraints

Description

Instances of `Advisor.authoring.CompositeConstraint` class contain multiple constraints. Depending on the instance definition, the Model Advisor reports a violation if a model does not meet one or all of the constraints.

Construction

`cc = Advisor.authoring.CompositeConstraint()` creates an instance of this class

Properties

`ConstraintID` — IDs of constraints

character vector | cell array of character vectors

IDs of constraints that compose an `Advisor.authoring.CompositeConstraint` object. This property is read-only. Use the `addConstraintID` method to create a `CompositeConstraint`.

`CompositeOperator` — Operator for specifying whether the Model Advisor reports a violation

character vector

Use `and` operator to specify that the Model Advisor reports a violation if a model does not meet all of the check constraints. Use `or` operator to specify that the Model Advisor reports a violation if a model does not meet at least one of the check constraints. This property is read/write.

Methods

addConstraintID	Add constraint to composite constraint

Examples

Specify a Composite Constraint

These commands specify a composite constraint for Multi-Port Switch blocks.

Create three `PositiveBlockParameter` constraint objects.

```
c1 = Advisor.authoring.PositiveBlockParameterConstraint();
c1.ID = 'ID_A2';
c1.BlockType = 'MultiPortSwitch';
c1.ParameterName = 'DataPortOrder';
c1.SupportedParameterValues = {'Specify indices'};
c1.ValueOperator = 'eq';
```

```
c2 = Advisor.authoring.PositiveBlockParameterConstraint();
c2.ID = 'ID_A3';
c2.BlockType = 'MultiPortSwitch';
c2.ParameterName = 'DataPortForDefault';
c2.SupportedParameterValues = {'Additional data port'};
c2.ValueOperator = 'eq';

c3 = Advisor.authoring.PositiveBlockParameterConstraint();
c3.ID = 'ID_A4';
c3.BlockType = 'MultiPortSwitch';
c3.ParameterName = 'DiagnosticForDefault';
c3.SupportedParameterValues = {'None'};
c3.ValueOperator = 'eq';
```

Use the `addPreRequisiteConstraintID` method to make `c1` a prerequisite to checking constraints `c2` and `c3`.

```
c2.addPreRequisiteConstraintID('ID_A2');
c3.addPreRequisiteConstraintID('ID_A2');
```

Create a composite constraint that specifies that if a Rate Transition block does not meet both constraints `c2` and `c3`, the block is in violation of this check.

```
CC = Advisor.authoring.CompositeConstraint();
CC.addConstraintID('ID_A3');
CC.addConstraintID('ID_A4');
CC.CompositeOperator = 'and';
```

Version History

Introduced in R2018a

See Also

`PositiveBlockParameterConstraint` | `NegativeBlockParameterConstraint` | `PositiveModelParameterConstraint` | `NegativeModelParameterConstraint` | `PositiveBlockTypeConstraint` | `NegativeBlockTypeConstraint`

Topics

“Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters”

Advisor.authoring.PositiveBlockTypeConstraint class

Package: `Advisor.authoring`

Create a Model Advisor constraint to check for supported block types

Description

Instances of `Advisor.authoring.PositiveBlockTypeConstraint` class define the only blocks that a model can contain.

Construction

`constraint = Advisor.authoring.PositiveBlockTypeConstraint()` creates an instance of this class.

Properties

`ID` — Unique identifier

character vector

Unique identifier for the positive block type constraint. This property is read/write.

`SupportedBlockTypes` — Supported block types

structure of character vectors

Structure consisting of these fields:

- `''BlockType'`
- `'MaskType'`

List of supported block types. For more information on the **MaskType** field, see “Mask Editor Overview” and “Mask Parameters”.This property is read/write.

`PreRequisiteConstraintIDs` — IDs of prerequisite constraints

cell array of character vectors

IDs of constraints that you specify as prerequisites by using the `addPreRequisiteConstraintID` method. If a prerequisite is not satisfied, the Model Advisor does not check the constraint that has the prerequisite. This property is read-only.

Examples

Specify Supported Block Types

These commands specify that a model contain only Inport, Outport, and Gain blocks and Constant blocks that have a specified mask:

```
c1=Advisor.authoring.PositiveBlockTypeConstraint;
c1.ID='ID_1';
```

```
s1=struct('BlockType','Inport','MaskType','');
s2=struct('BlockType','Outport','MaskType','');
s3=struct('BlockType','Gain','MaskType','');
s4=struct('BlockType','Constant','MaskType','Stateflow');
c1.SupportedBlockTypes={s1;s2;s3;s4;};
```

Version History

Introduced in R2018a

See Also

`PositiveModelParameterConstraint` | `NegativeModelParameterConstraint` | `PositiveBlockParameterConstraint` | `NegativeBlockParameterConstraint` | `NegativeBlockTypeConstraint` | `CompositeConstraint`

Topics

“Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters”

Advisor.authoring.NegativeModelParameterConstraint class

Package: `Advisor.authoring`

Create a Model Advisor constraint to check for unsupported model parameter values

Description

Instances of `Advisor.authoring.NegativeModelParameterConstraint` class define unsupported values for specified model parameters.

Construction

`constraint = Advisor.authoring.NegativeModelParameterConstraint` creates an instance of this class.

Properties

`ID` — Unique identifier

character vector

Unique identifier for the negative model parameter constraint. This property is read/write.

`ParameterName` — Name of model parameter

character vector

Model parameter for which you are specifying a constraint. This property is read/write.

`UnsupportedParameterValues` — Unsupported model parameter values

cell array of character vectors | cell array of structs | cell array of array of character vectors

List of unsupported values for the model parameter specified by the `ParameterName` field. This property is read/write.

`PreRequisiteConstraintIDs` — IDs of prerequisite constraints

cell array of character vectors

IDs of constraints that you specify as prerequisites by using the `addPreRequisiteConstraintID` method. If a prerequisite is not satisfied, the Model Advisor does not check the constraint that has the prerequisite. This property is read-only.

Examples

Specify Unsupported Model Parameter Value

These commands specify that the **MaxType** parameter does not support a value of zero:

```
c1=Advisor.authoring.NegativeModelParameterConstraint;
c1.ID='ID_1';
c1.ParameterName='MaxStep';
c1.UnsupportedParameterValues={'0'};
```

Version History

Introduced in R2018a

See Also

`PositiveModelParameterConstraint` | `NegativeBlockParameterConstraint` | `PositiveBlockParameterConstraint` | `PositiveBlockTypeConstraint` | `NegativeBlockTypeConstraint` | `CompositeConstraint`

Topics

“Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters”

Advisor.authoring.PositiveModelParameterConstraint class

Package: `Advisor.authoring`

Create a Model Advisor constraint to check for supported model parameter values

Description

Instances of `Advisor.authoring.PositiveModelParameterConstraint` class define supported values for specified model parameters.

Construction

`constraint = Advisor.authoring.PositiveModelParameterConstraint()` creates an instance of this class.

Properties

`ID` — Unique identifier

character vector

Unique identifier for the positive model parameter constraint. This property is read/write.

`ParameterName` — Name of model parameter

character vector

Model parameter for which you are specifying a constraint. This property is read/write.

`SupportedParameterValues` — Supported model parameter values

cell array of character vectors | cell array of structs | cell array of array of character vectors

List of supported values for the model parameter specified by the `ParameterName` field. This property is read/write.

`PreRequisiteConstraintIDs` — IDs of prerequisite constraints

cell array of character vectors

IDs of constraints that you specify as prerequisites by using the `addPreRequisiteConstraintID` method. If a prerequisite is not satisfied, the Model Advisor does not check the constraint that has the prerequisite. This property is read-only.

Examples

Specify Supported Model Parameter Values

These commands specify that the Solver **Type** model parameter must have a value of `Variable-step`:

```
c1=Advisor.authoring.PositiveModelParameterConstraint;
c1.ID='ID_1';
```

```
c1.ParameterName='SolverType';
c1.SupportedParameterValues={'Variable-step'};
```

These commands specify that the **Stop time** model parameter must have a value of `10` or `15`:

```
c1=Advisor.authoring.PositiveModelParameterConstraint;
c1.ID='ID_1';
c1.ParameterName='StopTime';
c1.SupportedParameterValues={'10','15'};
```

For the **ReplacementTypes** model parameter (Embedded Coder Users), these commands specify two sets of supported values for the `double` and `single` data types:

```
c1 = Advisor.authoring.PositiveModelParameterConstraint();
c1.ID='ID_2';
c1.ParameterName = 'ReplacementTypes';
s1 = struct('double', 'a', 'single', 'b');
s2 = struct('double', 'c', 'single', 'b');
c1.SupportedParameterValues = {s1, s2};
```

Version History

Introduced in R2018a

See Also

`NegativeModelParameterConstraint` | `PositiveBlockParameterConstraint` | `NegativeBlockParameterConstraint` | `PositiveBlockTypeConstraint` | `NegativeBlockTypeConstraint` | `CompositeConstraint`

Topics

“Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters”

Advisor.authoring.NegativeBlockParameterConstraint class

Package: `Advisor.authoring`

Create a Model Advisor constraint to check for unsupported block parameter values

Description

Instances of `Advisor.authoring.NegativeBlockParameterConstraint` class define unsupported values for specified block parameters.

Construction

`constraint = Advisor.authoring.NegativeBlockParameterConstraint()` creates an instance of this class.

Properties

`ID` — Unique identifier

character vector

Unique identifier for the negative block constraint. This property is read/write.

`BlockType` — Block type

character vector

Block that contains the parameter for which you are specifying a constraint. For a list of block types, see “Block-Specific Parameters”. This property is read/write.

`ParameterName` — Name of block parameter

character vector

Block parameter for which you are specifying a constraint. For a list of block parameters, see “Block-Specific Parameters”. This property is read/write.

`UnsupportedParameterValues` — Unsupported block parameter values

cell array of character vectors | cell array of structs | cell array of array of character vectors

List of unsupported values for the block parameter specified by the `BlockType` and `ParameterName` fields. This property is read/write.

`ValueOperator` — Operator for specifying unsupported parameter values

character vector

To specify one or more unsupported values, use these operators:

- `'eq'`
- `'or'`

- 'lt'
- 'gt'
- 'ge'
- 'le'
- 'range'
- 'regex'

This property is read/write. For more information on the regex operator, see regexp.

PreRequisiteConstraintIDs — IDs of prerequisite constraints
cell array of character vectors

IDs of constraints that you specify as prerequisites by using the addPreRequisiteConstraintID method. If a prerequisite is not satisfied, the Model Advisor does not check the constraint that has the prerequisite. This property is read/write.

Examples

Specify Unsupported Block Parameter Values

For a Constant block, these commands specify that one or four values are unsupported for the **Value** parameter:

```
c1=Advisor.authoring.NegativeBlockParameterConstraint;
c1.ID='ID_1';
c1.BlockType='Constant';
c1.ParameterName='Value';
c1.UnsupportedParameterValues={'1','4'};
c1.ValueOperator='or';
```

Version History

Introduced in R2018a

See Also

PositiveBlockParameterConstraint | PositiveModelParameterConstraint | NegativeModelParameterConstraint | NegativeModelParameterConstraint | PositiveBlockTypeConstraint | NegativeBlockTypeConstraint | CompositeConstraint

Topics
“Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters”

Advisor.authoring.PositiveBlockParameterConstraint class

Package: `Advisor.authoring`

Create a Model Advisor constraint to check for supported block parameter values

Description

Instances of `Advisor.authoring.PositiveBlockParameterConstraint` class define supported values for a specified block parameter.

Construction

`constraint = Advisor.authoring.PositiveBlockParameterConstraint` creates an instance of this class.

Properties

`ID` — Unique identifier

character vector

Unique identifier for the positive block parameter constraint. This property is read/write.

`BlockType` — Block type

character vector

Block that contains the parameter for which you are specifying a constraint. For a list of block types, see “Block-Specific Parameters”. This property is read/write.

`ParameterName` — Name of block parameter

character vector

Block parameter for which you are specifying a constraint. For a list of block parameters, see “Block-Specific Parameters”. This property is read/write.

`SupportedParameterValues` — Supported block parameter values

cell array of character vectors | cell array of structs | cell array of array of character vectors

List of supported values for the block parameter specified by the `BlockType` and `ParameterName` fields. This property is read/write.

`ValueOperator` — Operator for specifying supported parameter values

character vector

Use these operators to specify one or more supported values:

- `'eq'`
- `'or'`

- 'lt'
- 'gt'
- 'ge'
- 'le'
- 'range'
- 'regex'

This property is read/write. For more information on the regex operator, see regexp.

PreRequisiteConstraintIDs — IDs of prerequisite constraints
cell array of character vectors

IDs of constraints that you specify as prerequisites by using the addPreRequisiteConstraintID method. If a prerequisite is not satisfied, the Model Advisor does not check the constraint that has the prerequisite. This property is read-only.

Example

Specify Supported Block Parameter Values

For a Constant block, these commands specify that the **Value** parameter must have values of 2 and 5.

```
c1=Advisor.authoring.PositiveBlockParameterConstraint;
c1.ID='ID_1';
c1.BlockType='Constant';
c1.ParameterName='Value';
c1.SupportedParameterValues={'[2,5]'};
c1.ValueOperator='eq';
```

For a Constant block, these commands specify that the **Value** parameter must have a value between 1 and 4.

```
c1=Advisor.authoring.PositiveBlockParameterConstraint;
c1.ID='ID_1';
c1.BlockType='Constant';
c1.ParameterName='Value';
c1.SupportedParameterValues={'1','4'};
c1.ValueOperator='range';
```

Version History

Introduced in R2018a

See Also

NegativeBlockParameterConstraint | PositiveModelParameterConstraint | NegativeModelParameterConstraint | PositiveBlockTypeConstraint | NegativeBlockTypeConstraint | CompositeConstraint

Topics
“Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters”

addPreRequisiteConstraintID

Class: `Advisor.authoring.PositiveBlockParameterConstraint`, `Advisor.authoring.NegativeBlockParameterConstraint`, `Advisor.authoring.PositiveModelParameterConstraint`, `Advisor.authoring.NegativeModelParameterConstraint`, `Advisor.authoring.PositiveBlockTypeConstraint`, `Advisor.authoring.NegativeBlockTypeConstraint`
Package: `Advisor.authoring`

Check a prerequisite `constraint` object before the actual `constraint` object

Syntax

```
addPreRequisiteConstraintID(ID_1)
```

Description

Specify a constraint as a prerequisite to a `constraint` object. The Model Advisor checks the prerequisite constraint before checking the actual `constraint` object.

`addPreRequisiteConstraintID(ID_1)` specifies a prerequisite constraint ID `ID_1` that the Model Advisor checks before checking the actual `constraint` object.

Input Arguments

`ID_1` — `ID` of constraint object

character vector

To create constraint objects that you can specify as prerequisite constraints, use these classes:

- `Advisor.authoring.PositiveBlockParameterConstraint`
- `Advisor.authoring.NegativeBlockParameterConstraint`
- `Advisor.authoring.PositiveModelParameterConstraint`
- `Advisor.authoring.NegativeModelParameterConstraint`
- `Advisor.authoring.PositiveBlockTypeConstraint`
- `Advisor.authoring.NegativeBlockTypeConstraint`

Examples

Specify a Prerequisite Constraint

Specify a constraint on a Gain block. Specify this constraint as a prerequisite for a constraint on a Constant block.

Use the `PositiveBlockParameterConstraint` class to create a constraint on the **Gain** parameter of a Gain block.

```
c1=Advisor.authoring.PositiveBlockParameterConstraint;
c1.ID='ID_1';
c1.BlockType='Gain';
c1.ParameterName='Gain';
c1.SupportedParameterValues={'0','5'};
c1.ValueOperator='range';
```

Use the `NegativeBlockParameterConstraint` class to create a negative constraint on the **Value** parameter of a Constant block.

```
c2=Advisor.authoring.NegativeBlockParameterConstraint;
c2.ID='ID_2';
c2.BlockType='Constant';
c2.ParameterName='Value';
c2.UnsupportedParameterValues={'5'};
c2.ValueOperator='lt';
```

Use the `AddPreRequisiteConstraintID` method to specify the Gain block constraint as a prerequisite to the Constant block constraint.

```
c2.addPreRequisiteConstraintID('ID_1');
```

The Model Advisor does not check the Constant block constraint unless the **Gain** parameter has a value between `0` and `5`.

Version History

Introduced in R2018a

See Also

`PositiveBlockParameterConstraint` | `NegativeBlockParameterConstraint` | `PositiveModelParameterConstraint` | `NegativeModelParameterConstraint` | `PositiveBlockTypeConstraint` | `NegativeBlockTypeConstraint`

Topics

“Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters”

addConstraintID

Class: `Advisor.authoring.CompositeConstraint`
Package: `Advisor.authoring`

Add constraint to composite constraint

Syntax

```
addConstraintID(ID_1)
```

Description

Specify a constraint ID to add to a composite constraint.

`addConstraintID(ID_1)` specifies a constraint ID `ID_1` that the Model Advisor checks as part of a `CompositeConstraint` object.

Input Arguments

`ID_1` — ID of constraint object

character vector

To create root constraint objects that you can specify as part of a composite constraint, use these classes:

- `Advisor.authoring.PositiveBlockParameterConstraint`
- `Advisor.authoring.NegativeBlockParameterConstraint`
- `Advisor.authoring.PositiveBlockTypeConstraint`
- `Advisor.authoring.NegativeBlockTypeConstraint`

Examples

Specify a Composite Constraint

These commands specify a composite constraint for Multi-Port Switch blocks:

Create three `PositiveBlockParameter` constraint objects.

```
c1 = Advisor.authoring.PositiveBlockParameterConstraint();
c1.ID = 'ID_A1';
c1.BlockType = 'MultiPortSwitch';
c1.ParameterName = 'DataPortOrder';
c1.SupportedParameterValues = {'Specify indices'};
c1.ValueOperator = 'eq';

c2 = Advisor.authoring.PositiveBlockParameterConstraint();
c2.ID = 'ID_A2';
c2.BlockType = 'MultiPortSwitch';
c2.ParameterName = 'DataPortForDefault';
c2.SupportedParameterValues = {'Additional data port'};
c2.ValueOperator = 'eq';

c3 = Advisor.authoring.PositiveBlockParameterConstraint();
```

```
c3.ID = 'ID_A3';
c3.BlockType = 'MultiPortSwitch';
c3.ParameterName = 'DiagnosticForDefault';
c3.SupportedParameterValues = {'None'};
c3.ValueOperator = 'eq';
```

Use the `addPreRequisiteConstraintID` method to make `c1` a prerequisite to checking constraints `c2` and `c3`.

```
c2.addPreRequisiteConstraintID('ID_1');
c3.addPreRequisiteConstraintID('ID_2');
```

Create a composite constraint that specifies that if a Rate Transition block does not meet both constraints `c2` and `c3`, the block is in violation of this check.

```
CC = Advisor.authoring.CompositeConstraint();
CC.addConstraintID('ID_A2');
CC.addConstraintID('ID_A3');
CC.CompositeOperator = 'and';
```

Version History

Introduced in R2018a

See Also

`PositiveModelParameterConstraint` | `NegativeModelParameterConstraint` | `PositiveBlockParameterConstraint` | `NegativeBlockParameterConstraint` | `PositiveBlockTypeConstraint` | `NegativeBlockTypeConstraint` | `CompositeConstraint`

Topics

“Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters”

Advisor.authoring.NegativeBlockTypeConstraint class

Package: `Advisor.authoring`

Create a Model Advisor constraint to check for unsupported blocks

Description

Instances of `Advisor.authoring.NegativeBlockTypeConstraint` class define blocks that a model must not contain.

Construction

`constraint = Advisor.authoring.NegativeBlockTypeConstraint` creates an instance of this class.

Properties

`ID` — Unique identifier

character vector

Unique identifier for the block type constraint. This property is read/write.

`UnsupportedBlockTypes` — Unsupported block types

structure of character vectors

Structure consisting of these fields:

- `'BlockType'`
- `'MaskType'`

List of unsupported block types. This property is read/write. For more information on the **MaskType** field, see “Mask Editor Overview” and “Mask Parameters”.

`PreRequisiteConstraintIDs` — IDs of prerequisite constraints

cell array of character vectors

IDs of constraints that you specify as prerequisites by using the `addPreRequisiteConstraintID` method. If a prerequisite is not satisfied, the Model Advisor does not check the constraint that has the prerequisite. This property is read-only.

Examples

Specify Unsupported Block Types

These commands specify that a model cannot contain Rate Transition and Integrator blocks and Constant blocks with a specified mask:

```
c1=Advisor.authoring.NegativeBlockTypeConstraint;
c1.ID='ID_1';
```

```
s1=struct('BlockType','Integrator','MaskType','');
s2=struct('BlockType','RateTransition','MaskType','');
s3=struct('BlockType','Constant','MaskType','Stateflow');
c1.UnsupportedBlockTypes={s1;s2;};
```

Version History

Introduced in R2018a

See Also

`PositiveBlockParameterConstraint` | `NegativeBlockParameterConstraint` | `PositiveModelParameterConstraint` | `NegativeModelParameterConstraint` | `PositiveBlockTypeConstraint` | `CompositeConstraint`

Topics

“Define Model Advisor Checks for Supported and Unsupported Blocks and Parameters”

Value property

Status of check

Values

`true`
`false`
`''` (default)

Description

The `Value` property specifies the initial status of the check. When you use the `Value` property to specify the initial status of the check, you enable or disable **Run This Check** in the Model Advisor window.

If you want to specify the initial status of a check in the **By Product** folder, before starting Model Advisor, make sure `ModelAdvisor.Preferences.DeselectByProduct` is `false`.

`true`	Check is enabled
`false`	Check is disabled

Examples

```
% hide all checks that do not belong to Demo group
if ~(strcmp(checkCellArray{i}.Group, 'Demo'))
    checkCellArray{i}.Visible = false;
    checkCellArray{i}.Value = false;
end
```

See Also

`ModelAdvisor.Preferences`

Value property

Class: `ModelAdvisor.InputParameter`
Package: `ModelAdvisor`

Value of input parameter

Values

Depends on the `Type` property.

Description

The `Value` property specifies the initial value of the input parameter. This property is valid only when the `Type` property is one of the following:

- `'Bool'`
- `'String'`
- `'Enum'`
- `'ComboBox'`

Examples

```
% define input parameters
inputParam1 = ModelAdvisor.InputParameter;
inputParam1.Name = 'Skip font checks.';
inputParam1.Type = 'Bool';
inputParam1.Value = false;
```

Value property

Class: `ModelAdvisor.Task`
Package: `ModelAdvisor`

Status of task

Values

`true` — Initial status of task is enabled
`false` — Initial status of task is disabled

Description

The `Value` property indicates the initial status of a task—whether it is enabled or disabled.

When adding checks as tasks, the Model Advisor uses the task `Value` property instead of the check `Value` property.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.Value = false;
```

view

View Model Advisor run results for checks

Syntax

```
view(CheckResultObj)
```

Description

`view(CheckResultObj)` opens a web browser and displays the results of the check specified by `CheckResultObj`. `CheckResultObj` is a `ModelAdvisor.CheckResult` object returned by `ModelAdvisor.run`.

Examples

View Model Advisor Results

Run a check on a model and view the results in a web browser.

Use Model Advisor to run the check **Check optimization settings** on the model `vdp`.

```
results = ModelAdvisor.run({'vdp'},...
    {'mathworks.design.OptimizationSettings'});
```

View the results of the Model Advisor run.

```
view(results{1}.CheckResultObjs(1))
```

The results for the model `vdp` correspond to the first `ModelAdvisor.SystemResult` object in `results`. The results for the check **Check optimization settings** correspond to the first `ModelAdvisor.CheckResult` object in the array of `CheckResultObj` objects.

For more information on Model Advisor checks, see “Model Advisor Checks Documentation”. For details on how to find check IDs, see “Find Model Advisor Check IDs”.

Input Arguments

`CheckResultObj` — Model Advisor check results

`ModelAdvisor.CheckResult`

`ModelAdvisor.CheckResult` object which is a part of a `ModelAdvisor.SystemResult` object returned by `ModelAdvisor.run`.

Alternatives

“View Model Advisor Report”

Version History

Introduced in R2010b

See Also

`ModelAdvisor.lookupCheckID` | `ModelAdvisor.run` | `getReportFileName` | `ModelAdvisor.summaryReport` | `viewReport`

Topics

“Checking Systems Programmatically”
“Create a Function to Check Multiple Systems”
“Archive and View Model Advisor Run Results”
“Model Advisor Checks Documentation”
“Find Model Advisor Check IDs”

viewReport

View Model Advisor run results for systems

Syntax

```
viewReport(Result)
viewReport(Result,'MA')
viewReport(Result,'Cmd')
```

Description

`viewReport(Result)` opens the Model Advisor Report for the system result specified by `Result`. `Result` is a `ModelAdvisor.SystemResult` object returned by `ModelAdvisor.run`.

`viewReport(Result,'MA')` displays the results in the Model Advisor.

`viewReport(Result,'Cmd')` displays the Model Advisor run summary in the Command Window.

Examples

Open a Model Advisor Report

Open the Model Advisor Report for the specified system.

Use Model Advisor to run the check **Check optimization settings** on the model `vdp`.

```
results = ModelAdvisor.run({'vdp'},...
    {'mathworks.design.OptimizationSettings'});
```

Open the Model Advisor report for `vdp`. The results for the model `vdp` correspond to the first `ModelAdvisor.SystemResult` object in `results`.

```
viewReport(results{1})
```

For more information on Model Advisor checks, see “Model Advisor Checks Documentation”. For details on how to find check IDs, see “Find Model Advisor Check IDs”.

Display Results in Model Advisor

Open Model Advisor and display the Model Advisor run results.

Use Model Advisor to run the check **Check optimization settings** on the model `vdp`.

```
results = ModelAdvisor.run({'vdp'},...
    {'mathworks.design.OptimizationSettings'});
```

Open Model Advisor and display the results of the Model Advisor run for `vdp`.

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

For more information on Model Advisor checks, see “Model Advisor Checks Documentation”. For details on how to find check IDs, see “Find Model Advisor Check IDs”.

Display Summary in the Command Window

Display the Model Advisor run summary in the Command Window.

Use Model Advisor to run the check **Check optimization settings** on the model `vdp`.

```
results = ModelAdvisor.run({'vdp'},...
    {'mathworks.design.OptimizationSettings'});
```

Display the results for `vdp` in the Command Window.

```
viewReport(results{1}, 'Cmd')
```

For more information on Model Advisor checks, see “Model Advisor Checks Documentation”. For details on how to find check IDs, see “Find Model Advisor Check IDs”.

Input Arguments

`Result` — Model Advisor results

`ModelAdvisor.SystemResult` object

`ModelAdvisor.SystemResult` object returned by `ModelAdvisor.run`.

Alternatives

- “View Model Advisor Report”
- “View Results in Command Window”

Version History

Introduced in R2010b

See Also

`ModelAdvisor.lookupCheckID` | `ModelAdvisor.run` | `getReportFileName` | `ModelAdvisor.summaryReport` | `view`

Topics

“Checking Systems Programmatically”
“Create a Function to Check Multiple Systems”
“Automate Model Advisor Check Execution”
“Archive and View Model Advisor Run Results”
“Model Advisor Checks Documentation”
“Find Model Advisor Check IDs”

Visible property

Indicate to display or hide check

Values

`true` (default)
`false`

Description

The `Visible` property specifies whether the Model Advisor displays the check.

`true`	Display the check
`false`	Hide the check

Examples

```
% hide all checks that do not belong to Demo group
if ~(strcmp(checkCellArray{i}.Group, 'Demo'))
    checkCellArray{i}.Visible = false;
    checkCellArray{i}.Value = false;
end
```

Visible property

Class: `ModelAdvisor.Task`
Package: `ModelAdvisor`

Indicate to display or hide task

Values

`true` (default) — Display task in the Model Advisor
`false` — Hide task

Description

The `Visible` property specifies whether the Model Advisor displays the task.

Caution When adding checks as tasks, you cannot specify both the task and check `Visible` properties, you must specify one or the other. If you specify both properties, the Model Advisor generates an error when the check `Visible` property is `false`.

Examples

```
MAT1 = ModelAdvisor.Task('com.mathworks.sample.TaskSample1');
MAT1.Visible = false;
```

slmetric.metric.registerMetric

Package: `slmetric.metric`

Register a custom model metric with the model metric repository

Syntax

```
[MetricID,err_msg] = slmetric.metric.registerMetric(classname)
```

Description

`[MetricID,err_msg] = slmetric.metric.registerMetric(classname)` register a custom model metric with the model metric repository. The new metric class must be on the MATLAB search path and derived from `slmetric.metric.Metric`.

Examples

Register a Custom Model Metric with the Model Metric Repository

This example shows how to register a custom model metric.

Create a new metric class, derived from `slmetric.metric.Metric`, called `my_metric`.

```
slmetric.metric.createNewMetricClass('my_metric')
```

Finish the custom model metric implementation and testing.

Register the new custom metric in the model metric repository.

```
[MetricID, err_msg] = slmetric.metric.registerMetric('my_metric');
```

Input Arguments

`classname` — Metric class name

character vector

New metric class name.

Data Types: `char`

Output Arguments

`MetricID` — Metric ID

character vector

Unique metric identifier.

Data Types: `char`

`err_msg` — Error message
character vector

If you cannot register a new class, the function returns an error message.

Data Types: `char`

Version History

Introduced in R2016a

See Also

`slmetric.metric.Metric` | `slmetric.metric.unregisterMetric` | `slmetric.metric.refresh` | `slmetric.metric.createNewMetricClass`

slmetric.metric.unregisterMetric

Package: `slmetric.metric`

Unregister a custom model metric from the model metric repository

Syntax

```
slmetric.metric.unregisterMetric(MetricID)
```

Description

`slmetric.metric.unregisterMetric(MetricID)` unregister a custom model metric from the model metric repository.

Input Arguments

`MetricID` — Unique metric identifier
character vector

Metric identifier for a custom model metric that you created.

Version History

Introduced in R2016a

See Also

`slmetric.metric.Metric` | `slmetric.metric.registerMetric` | `slmetric.metric.refresh` | `slmetric.metric.createNewMetricClass`

slmetric.metric.refresh

Package: `slmetric.metric`

Update available model metrics

Syntax

```
slmetric.metric.refresh()
```

Description

`slmetric.metric.refresh()` updates available metrics after manual updates to the metric registration file.

Version History

Introduced in R2016a

See Also

`slmetric.metric.Metric` | `slmetric.metric.registerMetric` | `slmetric.metric.unregisterMetric` | `slmetric.metric.createNewMetricClass`

slmetric.metric.createNewMetricClass

Package: `slmetric.metric`

Create new metric class for a custom model metric

Syntax

```
slmetric.metric.createNewMetricClass(class_name)
```

Description

`slmetric.metric.createNewMetricClass(class_name)` creates a `slmetric.metric.Metric` class in the current working folder. The new metric class is used to define a custom model metric and supports the following `Advisor.component.Types`:

- `Model`
- `SubSystem`
- `ModelBlock`
- `Chart`
- `MATLABFunction`

Examples

Create a Custom Model Metric Class

This example shows how to create a new metric class `my_metric`.

Call the function and provide a name for the new metric class:

```
slmetric.metric.createNewMetricClass('my_metric')
```

The function creates a `my_metric.m` file in the current working folder.

The file contains the class definition for `my_metric`, which includes the constructor and an empty metric `algorithm` method.

```
classdef my_metric < slmetric.metric.Metric
    %my_metric Summary of this metric class goes here
    %   Detailed explanation goes here
    properties
    end

    methods
        function this = my_metric()
            this.ID = 'my_metric';
            this.ComponentScope = [Advisor.component.Types.Model, ...
                Advisor.component.Types.SubSystem];
            this.AggregationMode = slmetric.AggregationMode.Sum;
            this.CompileContext = 'None';
```

```
            this.Version = 1;
            this.SupportsResultDetails = false;

            %Textual information on the metric algorithm
            this.Name = '';
            this.Description = '';
            this.ValueName = '';
            this.AggregatedValueName = '';
            this.MeasuresNames = {};
            this.AggregatedMeasuresNames = {};
        end

        function res = algorithm(this, component)
            res = slmetric.metric.Result();
            res.ComponentID = component.ID;
            res.MetricID = this.ID;
            res.Value = 0;
        end
    end
end
```

Write your custom metric algorithm in `algorithm`.

When your custom metric class is working and tested, register your metric using `slmetric.metric.registerMetric`.

Input Arguments

`class_name` — Name of the new metric class

character vector

Name of the new metric class you are creating for a custom metric.

Data Types: `char`

Version History

Introduced in R2016a

See Also

`Advisor.component.Types` | `slmetric.metric.Metric` | `slmetric.metric.registerMetric` | `slmetric.metric.unregisterMetric`

exportMetrics

Class: `slmetric.Engine`
Package: `slmetric`

Export model metrics

Syntax

```
exportMetrics(metric_engine,filename)
exportMetrics(metric_engine,filename,filelocation)
```

Description

Export model metric data to an XML file.

`exportMetrics(metric_engine,filename)` exports an XML `filename` containing metric data to your current folder.

`exportMetrics(metric_engine,filename,filelocation)` exports an XML `filename` containing metric data to `filelocation`.

Input Arguments

`metric_engine` — Collects and accesses metric data

`slmetric.Engine` object

When you call `execute`, `metric_engine` collects metric data for available metrics or for the specified `MetricIDs`. Calling `getMetrics` accesses the collected metric data in `metric_engine`.

`filename` — XML file name

character vector

Name of XML file.

Example: `'MyMetrics.xml'`

`filelocation` — File path

character vector

Path to XML file

Example: `'C:/mywork'`

Examples

Export Metrics to Current Folder

This example shows how to export metrics for model `vdp` to XML file `MyMetrics.xml` in your current folder.

```
% Create an slmetric.Engine object
metric_engine = slmetric.Engine();

% Specify model for metric analysis
setAnalysisRoot(metric_engine, 'Root', 'vdp', 'RootType', 'Model');

% Generate and collect model metrics
execute(metric_engine);
rc = getMetrics(metric_engine);

% Export metrics to XML file myMetrics.xml
exportMetrics(metric_engine, 'MyMetrics.xml');
```

Export Metrics to Specified Location

This example shows how to export metrics for model `vdp` to XML file `MyMetrics.xml` in a specified folder, `C:/work`.

```
% Create an slmetric.Engine object
metric_engine = slmetric.Engine();

% Specify model for metric analysis
setAnalysisRoot(metric_engine, 'Root', 'vdp', 'RootType', 'Model');

% Collect model metrics
execute(metric_engine);
rc = getMetrics(metric_engine);

% Export metrics to XML file myMetrics.xml
exportMetrics(metric_engine, 'MyMetrics.xml', 'C:/work');
```

Version History

Introduced in R2016a

See Also

`slmetric.metric.ResultCollection` | `slmetric.metric.getAvailableMetrics`

Topics

“Collect Model Metrics Programmatically”
“Model Metrics” on page 2-299

clonedetection

Open Clone Detector app

Syntax

```
clonedetection(model)
```

Description

`clonedetection(model)` opens the Clone Detector App for a model specified by `model`. If the specified model is not open, this command opens it.

Examples

Open Identify Modeling Clones tool for a model

Open the Clone Detector App for `rtwdemo_preprocessor_subsys` example model:

```
clonedetection('rtwdemo_preprocessor_subsys')
```

Input Arguments

`model` — Model name

character vector

Model name or handle, specified as a character vector.

Data Types: `char`

Version History

Introduced in R2017a

See Also

“Enable Component Reuse by Using Clone Detection”

slmetric.dashboard.Configuration class

Package: `slmetric.dashboard`

Object containing information on Metrics Dashboard layout and widgets

Description

Instances of `slmetric.dashboard.Configuration` contain information on the layout and types of widgets in the Metric Dashboard.

Construction

Use the `slmetric.dashboard.Configuration` class to specify the layout and types of widgets in the Metrics Dashboard. To create an `slmetric.dashboard.Configuration` object, use the `new` method. Each `slmetric.dashboard.Configuration` object contains one `slmetric.dashboard.Layout` object. Use the methods and properties of the `slmetric.dashboard.Layout` class to customize the widgets and layout of the Metrics Dashboard.

You can modify an existing Metrics Dashboard layout, such as the shipped Metrics Dashboard layout, by using the `getDashboardLayout` method.

Properties

`Name` — Configuration object name

character vector | string scalar

Name of configuration object that you use to specify the Metrics Dashboard layout. This property is read/write.

Data Types: `char`

`FileName` — XML file name

character vector | string scalar

XML file name that contains information on the current Metrics Dashboard layout. This property is read/write.

Data Types: `char`

`Location` — XML file location

character vector | string scalar

Location of XML file that contains Metrics Dashboard layout. This property is optional and read/write.

Data Types: `char`

Methods

getDashboardLayout	Create Metrics Dashboard layout object in base workspace
new	Create configuration object for customizing Metrics Dashboard layout
open	Create `slmetric.dashboard.Configuration` object associated with XML configuration file in the base workspace
openDefaultConfiguration	Return shipping Metrics Dashboard configuration object in base workspace
save	Save contents of `slmetric.dashboard.Configuration` object to XML file

Examples

Create a Configuration Object

Use the `new` method to create an `slmetric.dashboard.Configuration` object. As an input, specify the name of the XML file that is to contain information on a custom metrics dashboard layout. After you add this information to the configuration object, use the `save` method to save the file.

```
CONF = slmetric.dashboard.Configuration.new('Name','default')
```

```
CONF =

  Configuration with properties:

       Name: 'default'
    FileName: ''
    Location: ''
```

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.dashboard.Container class

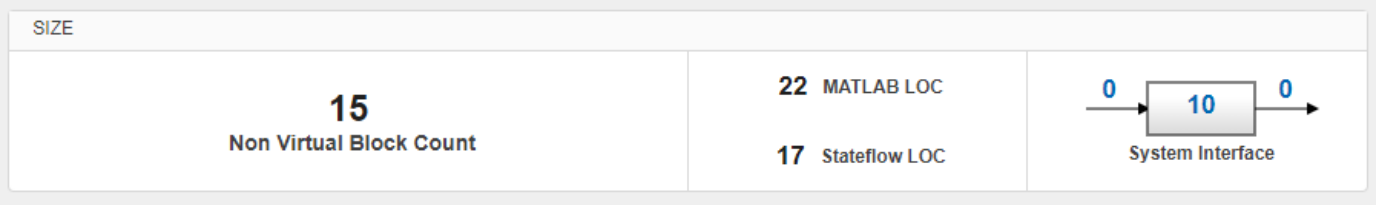
Package: `slmetric.dashboard`

Widget for holding `slmetric.dashboard.Widget` and `slmetric.dashboard.CustomWidget` objects in Metrics Dashboard

Description

An `slmetric.dashboard.Container` object holds `slmetric.dashboard.Widget` and `slmetric.dashboard.CustomWidget` objects. You can use the `slmetric.dashboard.Container` methods to specify the container size and border.

For example, the following image is from a Metrics Dashboard layout. This portion of the Metrics Dashboard contains an `slmetric.dashboard.Group` widget with the title `Size`. This group contains three `slmetric.dashboard.Container` widgets. The containers on the left and right each contain one `slmetric.dashboard.Widget` object. The middle container contains two `slmetric.dashboard.Widget` objects.



Construction

`container = slmetric.dashboard.Container` creates a handle to an `slmetric.dashboard.Container` object.

Properties

`ShowBorder` — Display a border around the `slmetric.dashboard.Container` object

`0` (default) | `logical`

If `true`, the `slmetric.dashboard.Container` object has a border around it in the Metrics Dashboard. This property is read/write.

`Type` — Widget type

`Container` (default)

This widget type is a container. This property is read-only.

Methods

addWidget	Add widget to `slmetric.dashboard.Container` object
getSeparators	Determine whether there are lines on sides of Metrics Dashboard container
getWidgets	Obtain a list of widgets in an `slmetric.dashboard.Container` object
getWidths	Obtain widths of Metrics Dashboard container
removeWidget	Remove widget from `slmetric.dashboard.Container` object
setSeparators	Specify lines on Metrics Dashboard container sides
setWidths	Specify multiple widths for Metrics Dashboard container
setMargin	Specify distance from container edge to its contents
getMargin	Obtain distance from container edge to its contents
getPosition	Obtain container position within Metrics Dashboard
setPosition	Set container position within Metrics Dashboard

Examples

Configure Compliance Metrics

You can use the Metrics Dashboard and metric APIs to obtain compliance and issues metric data on your Model Advisor configuration. To set up your Model Advisor configuration, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”. You can also use an existing check group such as the MISRA checks. 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:

Open the default configuration:

```
config=slmetric.config.Configuration.open()
```

Specify a metric family ID that you associate with those check groups:

```
famParamID = 'ModelAdvisorStandard';
```

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

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

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

To set up the configuration, pass the `values` cell array into the `setMetricFamilyParameterValues` method .

```
setMetricFamilyParameterValues(config, famParamID, values);
```

Point the **High Integrity Compliance** and **High Integrity Check Issues** widgets to the MISRA check group. To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain the widget objects that are in the layout object.

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

Obtain the compliance group from the layout. This group contains two containers. The first container contains the High Integrity and MAB Compliance and Check Issues widgets. Remove the **High Integrity Compliance** widget.

```
complianceGroup = layoutWidget(3);
complianceContainers = getWidgets(complianceGroup);
complianceContainerWidgets = getWidgets(complianceContainers(1));
complianceContainers(1).removeWidget(complianceContainerWidgets(1));
setMetricIDs(complianceContainerWidgets(1),...
({'mathworks.metrics.ModelAdvisorCompliance._SYSTEM_By Task_misra_c'}));
complianceContainerWidgets(1).Labels = {'MISRA'};
```

Add a custom widget for visualizing MISRA check issues metrics to the `complianceContainers` `slmetric.dashboard.Container` object.

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

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

```
save(config,'FileName','MetricConfig.xml');
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configurations.

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

For a model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”

“Customize Metrics Dashboard Layout and Functionality”

slmetric.dashboard.CustomWidget class

Package: `slmetric.dashboard`

Object for holding custom Metrics Dashboard widgets

Description

For custom or shipped metrics, use the `slmetric.dashboard.CustomWidget` object to visualize metric data in the Metrics Dashboard. Choose a single value, radial gauge, bar chart, or distribution heat map approach.

Construction

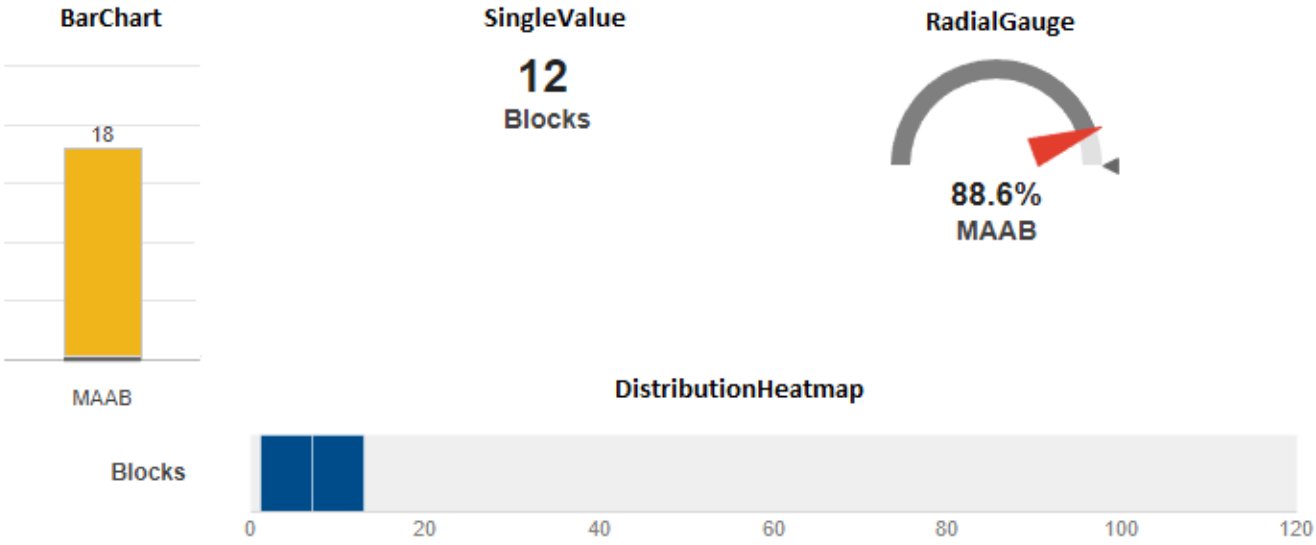
For `slmetric.dashboard.Layout`, `slmetric.dashboard.Container`, or `slmetric.dashboard.Group` objects, use the `addWidget` or `removeWidget` methods to add or remove `slmetric.dashboard.CustomWidget` objects from the Metrics Dashboard. Use `slmetric.dashboard.CustomWidget` methods to specify the widget size.

Properties

`VisualizationType` — Type of `slmetric.dashboard.CustomWidget` object
`SingleValue` (default) | `RadialGauge` | `BarChart` | `DistributionHeatmap`

Type of `slmetric.dashboard.CustomWidget` object that you want to add, remove, or modify in the Metrics Dashboard. This property is read/write. Choose from these widget types:

- bar chart
- single value
- radial gauge
- distribution heatmap



Data Types: `char`

`Labels` — Add labels to custom widget
`character vector | string scalar`

Add labels to custom widget. This property is only for the `BarChart` `VisualizationType` property, so you can add apply labels to each individual bar. This property is read/write.

Data Types: `char`

`Title` — Title of `slmetric.dashboard.CustomWidget` object
`character vector | string scalar`

Specify a title for the custom widget. For a radial gauge, there is a 16 character limit. This property is read/write.

Data Types: `char`

`Type` — Type of `slmetric.dashboard.CustomWidget` object
`'Custom'`

Type of `slmetric.dashboard.CustomWidget` object. This property is read-only.

Data Types: `char`

Methods

getSeparators	Determine whether there are lines on sides of Metrics Dashboard custom widget
getWidths	Obtain widths of Metrics Dashboard custom widget
setSeparators	Specify lines on Metrics Dashboard custom widget sides
setWidths	Specify multiples widths for Metrics Dashboard custom widget
setMetricIDs	Set metric identifier for custom Metrics Dashboard widget
getHeight	Obtain height of Metrics Dashboard custom widget
getMetricIDs	Obtain metric identifier for custom Metrics Dashboard widget
getPosition	Obtain custom widget position within Metrics Dashboard
setPosition	Set custom widget position within Metrics Dashboard
setHeight	Specify height of Metrics Dashboard custom widget

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

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

```
end

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

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

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



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

    isNonVirtual = true(size(blocks));

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

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

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

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

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

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

        blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For a model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.dashboard.Group class

Package: `slmetric.dashboard`

Widget for holding `slmetric.dashboard.Container`, `slmetric.dashboard.Widget` and `slmetric.dashboard.CustomWidget` objects on Metrics Dashboard

Description

An `slmetric.dashboard.Group` object can hold `slmetric.dashboard.Container`, `slmetric.dashboard.Widget` and `slmetric.dashboard.CustomWidget` objects. You can use the `slmetric.dashboard.Group` methods and properties to specify the group size, width, and title.

For example, the image is of the default Metrics Dashboard layout. This portion of the Metrics Dashboard contains an `slmetric.dashboard.Group` widget with the title `Size`. This group contains three `slmetric.dashboard.Container` widgets. The containers on the left and right each contain one `slmetric.dashboard.Widget` object. The middle container contains two `slmetric.dashboard.Widget` objects.

Construction

`group = slmetric.dashboard.Group` creates a handle to an `slmetric.dashboard.Group` object.

Properties

`Title` — Specify title of metrics group

`Character vector` | `String scalar`

Specify title for a group of `slmetric.dashboard.Widget` and `slmetric.dashboard.CustomWidget` objects. The title must summarize the types of widgets in the group. For example, a group with the title `Size` contains widgets pertaining to the size of the model. This property is read/write.

`Type` — Widget type

`Group` (default)

This widget type is a group. This property is read-only.

`ShowBorder` — Display a border around the `slmetric.dashboard.Group` object

`0` (default) | `logical`

If `true`, the `slmetric.dashboard.Group` object has a border around it in the Metrics Dashboard. This property is read/write.

Methods

addWidget	Add widget to `slmetric.dashboard.Group` object
getSeparators	Determine whether there are lines on sides of Metrics Dashboard group
getWidgets	Obtain a list of widgets in an `slmetric.dashboard.Group` object
getWidths	Obtain widths of Metrics Dashboard group
removeWidget	Remove widget from `slmetric.dashboard.Group` object
setSeparators	Specify lines on Metrics Dashboard group sides
setWidths	Specify multiple widths for Metrics Dashboard group
getMargin	Obtain distance from group edge to contents
getPosition	Obtain group position within Metrics Dashboard
setMargin	Specify distance from group edge to its contents
setPosition	Set group position within Metrics Dashboard

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

    methods
    function this = nonvirtualblockcount()
        this.ID = 'nonvirtualblockcount';
        this.Name = 'Nonvirtual Block Count';
        this.Version = 1;
        this.CompileContext = 'None';
        this.Description = 'Algorithm that counts nonvirtual blocks per level.';
        this.AggregatedValueName = 'Nonvirtual Blocks (incl. Descendants)'
        this.ValueName = 'Nonvirtual Blocks'
        this.ComponentScope = [Advisor.component.Types.Model, ...
```

```
        Advisor.component.Types.SubSystem];
    this.AggregationMode = slmetric.AggregationMode.Sum;
    this.ResultChecksumCoverage = true;
    this.SupportsResultDetails = true;

end

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

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

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



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

    isNonVirtual = true(size(blocks));

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

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

```
                        ios = get_param(blocks{n}, 'IndexOptionArray');

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

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

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

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

        blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.dashboard.Layout class

Package: `slmetric.dashboard`

Create object for holding Metrics Dashboard customizations

Description

Object that holds an array of widget objects. The size, types, and locations of widgets in an `slmetric.dashboard.Layout` object determine the Metrics Dashboard appearance. These are the widget objects:

- `slmetric.dashboard.Group`
- `slmetric.dashboard.Container`
- `slmetric.dashboard.Widget`
- `slmetric.dashboard.CustomWidget`

Construction

For an `slmetric.dashboard.Configuration` object, use the `getDashboardLayout` method to create an `slmetric.dashboard.Layout` object. You can add or remove widgets from this object. You can specify the size and location of these widgets in the Metrics Dashboard. Once you complete your specification, apply the `slmetric.dashboard.Configuration.save` method to save your configuration. Use the `slmetric.dashboard.setActiveConfiguration` function to set the active configuration.

Methods

addWidget	Add widget to `slmetric.dashboard.Layout` object
getWidgets	Obtain a list of widgets in an `slmetric.dashboard.Layout` object
removeWidget	Remove widget from `slmetric.dashboard.Layout` object

Examples

Configure Compliance Metrics

You can use the Metrics Dashboard and metric APIs to obtain compliance and issues metric data on your Model Advisor configuration. To set up your Model Advisor configuration, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”. You can also use an existing check group such as the MISRA checks. 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:

Open the default configuration:

```
config = slmetric.config.Configuration.open()
```

Specify a metric family ID that you associate with those check groups:

```
famParamID = 'ModelAdvisorStandard';
```

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

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

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

To set up the configuration, pass the `values` cell array into the `setMetricFamilyParameterValues` method.

```
setMetricFamilyParameterValues(config, famParamID, values);
```

Point the **High Integrity Compliance** and **High Integrity Check Issues** widgets to the MISRA check group. To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain the widget objects that are in the layout object.

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

Obtain the compliance group from the layout. This group contains two containers. The first container contains the High Integrity and MAB Compliance and Check Issues widgets. Remove the **High Integrity Compliance** widget.

```
complianceGroup = layoutWidget(3);
complianceContainers = getWidgets(complianceGroup);
complianceContainerWidgets = getWidgets(complianceContainers(1));
complianceContainers(1).removeWidget(complianceContainerWidgets(1));
setMetricIDs(complianceContainerWidgets(1),...
({'mathworks.metrics.ModelAdvisorCompliance._SYSTEM_By Task_misra_c'}));
complianceContainerWidgets(1).Labels = {'MISRA'};
```

Add a custom widget for visualizing MISRA check issues metrics to the `complianceContainers` `slmetric.dashboard.Container` object.

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

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

```
save(config,'FileName','MetricConfig.xml');
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configurations.

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

For a model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.dashboard.Widget class

Package: `slmetric.dashboard`

Object for holding Actual/Potential Reuse, System Interface, or System Info widgets

Description

You can add, remove, or modify `slmetric.dashboard.Widget` objects in the Metrics Dashboard. The types of `slmetric.dashboard.Widget` objects are **Actual Reuse/Potential Reuse**, **System Interface**, or **System Info** widgets.

Construction

For `slmetric.dashboard.Layout`, `slmetric.dashboard.Container`, or `slmetric.dashboard.Group` objects, use the `addWidget` or `removeWidget` methods to add or remove `slmetric.dashboard.Widget` objects from the Metrics Dashboard. Use the `slmetric.dashboard.Widget` methods to specify widget size.

Properties

`Title` — Title of `slmetric.dashboard.Widget` object

character vector | string scalar

By default, the `LibraryReuse` widget title is `Library Reuse`, the `SystemInfo` widget title is blank, and the `GlocalInterface` widget title is `System Interface`. This property is read/write.

Data Types: `char`

`Type` — Type of `slmetric.dashboard.Widget` object

`LibraryReuse` | `SystemInfo` | `GlocalInterface`

Type of `slmetric.dashboard.Widget` object that you want to add, remove, or modify in the Metrics Dashboard. This property is read-only.

Data Types: `char`

Methods

getSeparators	Determine whether there are lines on sides of Metrics Dashboard widget
getWidths	Obtain widths of Metrics Dashboard widget
setSeparators	Specify lines on Metrics Dashboard widget sides
setWidths	Specify multiple widths for Metrics Dashboard widget
getMetricIDs	Obtain metric identifier for Metrics Dashboard widget
getHeight	Obtain height of Metrics Dashboard widget
getPosition	Obtain widget position within Metrics Dashboard
setHeight	Specify height of Metrics Dashboard widget
setPosition	Set widget position within Metrics Dashboard

Examples

Create Metrics Dashboard with Three Widget Objects

Create a Metrics Dashboard with the three types of `slmetric.dashboard.Widget` objects.

To begin, create a new `slmetric.dashboard.Configuration` object.

```
config = slmetric.dashboard.Configuration.new('Name','default');
```

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

```
layout = getDashboardLayout(config);
```

Add the three `slmetric.dashboard.Widget` objects to the `slmetric.dashboard.Layout` object.

```
addWidget(layout,'LibraryReuse');
addWidget(layout,'SystemInfo');
addWidget(layout,'GlocalInterface');
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(config,'FileName','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

The Metrics Dashboard contains the three `slmetric.dashboard.Widget` objects.

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getDashboardLayout

Class: `slmetric.dashboard.Configuration`
Package: `slmetric.dashboard`

Create Metrics Dashboard layout object in base workspace

Syntax

```
Layout = getDashboardLayout(conf)
```

Description

`Layout = getDashboardLayout(conf)` creates an `slmetric.dashboard.Layout` object in the base workspace. Use this object to specify the location, size, and types of widgets that are in the Metrics Dashboard.

Input Arguments

`conf` — Metrics Dashboard configuration object

`slmetric.dashboard.Configuration` object

`slmetric.dashboard.Configuration` object for which to create a custom Metrics Dashboard configuration. By default, an `slmetric.dashboard.Configuration` object holds an empty `slmetric.dashboard.Layout` object.

Output Arguments

`Layout` — Metrics Dashboard layout object

`slmetric.dashboard.Layout` object

`slmetric.dashboard.Layout` object for which to specify the location, size, and types of widgets in the Metrics Dashboard.

Examples

Obtain an `slmetric.dashboard.Layout` Object

Use the `new` method to create an `slmetric.dashboard.Configuration` object. As an input, specify the name of the XML file that is to contain information on a custom metrics dashboard layout. After you add this information to the configuration object, use the `slmetric.dashboard.Configuration.save` method to save the file.

```
CONF = slmetric.dashboard.Configuration.new('Name','default')
```

```
CONF =

  Configuration with properties:
```

```
        Name: 'default'
    FileName: ''
    Location: ''
```

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

```
layout = getDashboardLayout(CONF);
```

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration` | `slmetric.dashboard.Configuration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.dashboard.Configuration.new

Class: `slmetric.dashboard.Configuration`
Package: `slmetric.dashboard`

Create configuration object for customizing Metrics Dashboard layout

Syntax

```
Co = slmetric.dashboard.Configuration.new('Name', 'Default')
```

Description

Create an `slmetric.dashboard.Configuration` object for holding customizations pertaining to the Metrics Dashboard layout. Use the `save` command to create and store an associated XML configuration file.

`Co = slmetric.dashboard.Configuration.new('Name', 'Default')` creates a configuration object.

Input Arguments

`Name` — Name of configuration object that is tagged in XML file

character vector | string scalar

Name of configuration object in XML file that contains customizations pertaining to the layout and types of widgets on the Metrics Dashboard.

Data Types: `char`

Output Arguments

`Co` — Configuration object

character vector | string scalar

Name of `slmetric.dashboard.Configuration` object that contains customizations pertaining to the layout and types of widgets on the Metrics Dashboard.

Data Types: `char`

Examples

Create a Configuration Object

Use the `new` method to create an `slmetric.dashboard.Configuration` object. As an input, specify a configuration object name. This name is then associated with a tag in the configuration object XML file. After adding information to the configuration object, use the `save` method to create and store an associated XML file.

```
CONF = slmetric.dashboard.Configuration.new('Name','default')
```

```
CONF =

  Configuration with properties:

       Name: 'default'
    FileName: ''
    Location: ''
```

Version History

Introduced in R2018b

See Also

`slmetric.config.Configuration` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

open

Class: `slmetric.dashboard.Configuration`
Package: `slmetric.dashboard`

Create `slmetric.dashboard.Configuration` object associated with XML configuration file in the base workspace

Syntax

```
Co = slmetric.dashboard.Configuration.open('FileName','myConfig.xml',...
'Location', pwd,'locale', 'ja_JP')
```

Description

Reads the contents of the XML file containing the Metrics Dashboard layout into memory and returns the corresponding configuration object. If you modify the contents of the configuration object, invoke the `save` method to write to the XML file.

`Co = slmetric.dashboard.Configuration.open('FileName','myConfig.xml',... 'Location', pwd,'locale', 'ja_JP')` reads a configuration file.

Note If you do not supply an input argument, the `slmetric.dashboard.Configuration.open` command reads the contents of the default Metrics Dashboard layout XML file into memory and returns the corresponding `slmetric.dashboard.Configuration` object.

Input Arguments

`FileName` — Name of XML file

character vector | string scalar

Name of XML file containing custom Metrics Dashboard layout and widgets.

Data Types: `char`

`Location` — Folder containing XML file

character vector | string scalar

Name of folder containing XML file that contains Metrics Dashboard layout. This input argument is optional.

Data Types: `char`

`locale` — Name of folder containing XML file

character vector | string scalar

Name of folder containing XML file that contains Metrics Dashboard layout. This input argument is optional.

Data Types: `char`

Output Arguments

`Co` — Configuration object
character vector | string scalar

Name of `slmetric.dashboard.Configuration` object that you want to open.

Data Types: `char`

Examples

Access an Existing Configuration Object

Use the `open` method to add an existing `slmetric.dashboard.Configuration` object to the base workspace. As an input, specify the name of the XML file that contains the information in the configuration object. If you modify the information that this configuration object contains, use the `save` method to save this information to the XML file.

```
CONF = slmetric.dashboard.Configuration.open('FileName',...
 'myConfig.xml', 'Location', pwd(), 'locale', 'ja_JP');
```

Version History

Introduced in R2018b

See Also

openDefaultConfiguration

Class: `slmetric.dashboard.Configuration`
Package: `slmetric.dashboard`

Return shipping Metrics Dashboard configuration object in base workspace

Syntax

```
DefaultLayout = slmetric.dashboard.Configuration.openDefaultConfiguration
```

Description

`DefaultLayout = slmetric.dashboard.Configuration.openDefaultConfiguration` returns the `slmetric.dashboard.Configuration` object corresponding to the shipping Metrics Dashboard layout in the base workspace. This object contains information on the size, type, and location of all widgets that ship with the Metrics Dashboard. Use this object to add or remove widgets from the shipping Metrics Dashboard configuration.

Output Arguments

`DefaultLayout` — Default Metrics Dashboard configuration object

`slmetric.dashboard.Configuration` object

`slmetric.dashboard.Configuration` object corresponding to the shipping `slmetric.dashboard.Configuration` object.

Examples

Open shipping Metrics Dashboard Configuration Object

Use the `openDefaultConfiguration` method to return the shipping `slmetric.dashboard.Configuration` object. If you modify the information that this configuration object contains, use the `slmetric.dashboard.Configuration.save` method to save this information to an XML file.

```
CONF = slmetric.dashboard.Configuration.openDefaultConfiguration
```

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”

“Customize Metrics Dashboard Layout and Functionality”

save

Class: `slmetric.dashboard.Configuration`
Package: `slmetric.dashboard`

Save contents of `slmetric.dashboard.Configuration` object to XML file

Syntax

```
save(Co,'FileName','myConfig.xml', ... ,'Location',pwd, 'locale', 'ja_JP');
```

Description

Save the contents of a configuration object to an XML file. The configuration object contains information on a custom Metrics Dashboard layout.

`save(Co,'FileName','myConfig.xml', ... ,'Location',pwd, 'locale', 'ja_JP');` saves the contents of a configuration object to an XML file. The XML file applies your customizations to the Metrics Dashboard.

Note Do not manually edit the XML file.

Input Arguments

`Co` — Metrics Dashboard Configuration object

`slmetric.dashboard.Configuration` object

`slmetric.dashboard.Configuration` object to save to an XML file.

`Filename` — Name of XML file that contains custom Metrics Dashboard layout

character vector | string scalar

Name of XML file that contains information on the location and types of widgets that are on the Metrics Dashboard.

Data Types: `char`

`Location` — Name of folder containing XML file that contains custom Metrics Dashboard layout

character vector | string scalar

Name of XML file that contains information on the location and types of widgets in the Metrics Dashboard. This input argument is optional.

Data Types: `char`

`locale` — Create folder that is to contain XML file

character vector | string scalar

Name of new folder that is to contain the XML file that contains information on the location and types of widgets in the Metrics Dashboard. If you do not specify a value for `locale`, Simulink creates the XML file in the folder that you specify with the `Location` property. This input argument is optional.

Data Types: `char`

Serialize a Configuration Object to XML File

Serialize a configuration object to an XML file.

Use the `save` method to serialize an `slmetric.dashboard.Configuration` object to an XML file. If you modify the information that this configuration object contains, use the `slmetric.dashboard.Configuration.save` method to save information to this file.

```
save(CONF,'config','FileName','Configfile.xml','Location',pwd)
```

Use the `slmetric.config.setActiveConfiguration` function to specify that the metric engine use this configuration.

```
slmetric.config.setActiveConfiguration('C:\temp\Configfile.xml');
```

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

addWidget

Class: `slmetric.dashboard.Container`
Package: `slmetric.dashboard`

Add widget to `slmetric.dashboard.Container` object

Syntax

```
newWidget = addWidget(container,widgetType,num)
```

Description

`newWidget = addWidget(container,widgetType,num)` adds a widget to an `slmetric.dashboard.Container` object.

Input Arguments

`container` — Add widget to Metrics Dashboard

`slmetric.dashboard.Container` object

`slmetric.dashboard.Container` object for which you want to add widgets to customize the Metrics Dashboard layout. This property is read-write.

`widgetType` — Metrics Dashboard widget

`Group` | `Container` | `SystemInfo` | `GlocalInterface` | `LibraryReuse` | `Custom`

Specify the `Type` property of an `slmetric.dashboard.Container`, `slmetric.dashboard.Widget`, `slmetric.dashboard.Group`, or `slmetric.dashboard.CustomWidget` object.

Data Types: `char`

`num` — Widget placement

`int`

Placement of widget in container on Metrics Dashboard. Order of widgets in the container proceeds from left to right, and then down in the container.

Output Arguments

`newWidget` — New Metrics Dashboard widget

widget object

New widget that you are adding to an `slmetric.dashboard.Container` object on the Metrics Dashboard. You can add these widgets to a container:

- `slmetric.dashboard.Group`
- `slmetric.dashboard.Container`
- `slmetric.dashboard.CustomWidget`

- `slmetric.dashboard.Widget`

Examples

Configure Compliance Metrics

You can use the Metrics Dashboard and metric APIs to obtain compliance and issues metric data on your Model Advisor configuration. To set up your Model Advisor configuration, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”. You can also use an existing check group such as the MISRA checks. 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:

Open the default configuration:

```
config=slmetric.config.Configuration.open()
```

Specify a metric family ID that you associate with those check groups:

```
famParamID = 'ModelAdvisorStandard';
```

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

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

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

To set up the configuration, pass the `values` cell array into the `setMetricFamilyParameterValues` method.

```
setMetricFamilyParameterValues(config, famParamID, values);
```

Point the **High Integrity Compliance** and **High Integrity Check Issues** widgets to MISRA check group. To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain the widget objects that are in the layout object.

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

Obtain the compliance group from the layout. This group contains two containers. The first container contains the High Integrity and MAB Compliance and Check Issues widgets. Remove the **High Integrity Compliance** widget.

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

```
complianceContainerWidgets = getWidgets(complianceContainers(1));
complianceContainers(1).removeWidget(complianceContainerWidgets(1));
setMetricIDs(complianceContainerWidgets(1),...
({'mathworks.metrics.ModelAdvisorCompliance._SYSTEM_By Task_misra_c'}));
complianceContainerWidgets(1).Labels={'MISRA'};
```

Add a custom widget for visualizing MISRA check issues metrics to the `complianceContainers` `slmetric.dashboard.Container` object.

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

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

```
save(config,'FileName','MetricConfig.xml');
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configurations.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getSeparators

Class: `slmetric.dashboard.Container`
Package: `slmetric.dashboard`

Determine whether there are lines on sides of Metrics Dashboard container

Syntax

```
S = getSeparators(container)
```

Description

`S = getSeparators(container)` returns a structure or an array of structures indicating whether there are lines on the sides of an `slmetric.dashboard.Container` object.

Input Arguments

`container` — Container for which you want to know whether there are separators

`slmetric.dashboard.Container` object

Determine whether there are separators on the sides of an `slmetric.dashboard.Container` object.

Output Arguments

S — Structure of four fields

`Structure` | `Array of Structures`

The output is a structure or an array of structures consisting of these fields:

- `S.top`
- `S.bottom`
- `S.left`
- `S.right`

Each field is empty or has a value of `1` or `0`. An empty field indicates that you did not set a value. A value of `1` indicates that there is a line on that container side. A value of `0` indicates that there is not a line on that container side.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` |
`slmetric.dashboard.getActiveConfiguration`

Topics
“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getWidgets

Class: `slmetric.dashboard.Container`
Package: `slmetric.dashboard`

Obtain a list of widgets in an `slmetric.dashboard.Container` object

Syntax

```
containerList = getWidgets(container)
```

Description

`containerList = getWidgets(container)` creates an array of objects that are in the `slmetric.dashboard.Container` object. These objects are widgets of the following types:

- `slmetric.dashboard.Group`
- `slmetric.dashboard.Container`
- `slmetric.dashboard.CustomWidget`
- `slmetric.dashboard.Widget`

Use the `getWidgets` method to identify widgets that you want to modify or remove from the `slmetric.dashboard.Container` object.

Input Arguments

`container` — Object that holds metric dashboard layout customizations
`slmetric.dashboard.Container` object

`slmetric.dashboard.Container` object for which you want to obtain a list of widgets.

Output Arguments

`containerList` — Array of objects in `slmetric.dashboard.Container` object
array of objects

Array of objects in `slmetric.dashboard.Container` object.

Examples

Configure Compliance Metrics

You can use the Metrics Dashboard and metric APIs to obtain compliance and issues metric data on your Model Advisor configuration. To set up your Model Advisor configuration, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”. You can also use an existing check group such as the MISRA checks. 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:

Open the default configuration:

```
config=slmetric.config.Configuration.open()
```

Specify a metric family ID that you associate with those check groups:

```
famParamID = 'ModelAdvisorStandard';
```

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

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

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

To set up the configuration, pass the `values` cell array into the `setMetricFamilyParameterValues` method.

```
setMetricFamilyParameterValues(config, famParamID, values);
```

Point the **High Integrity Compliance** and **High Integrity Check Issues** widgets to the MISRA check group. To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain the widget objects that are in the layout object.

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

Obtain the compliance group from the layout. This group contains two containers. The first container contains the High Integrity and MAB Compliance and Check Issues widgets. Remove the **High Integrity Compliance** widget.

```
complianceGroup = layoutWidget(3);
complianceContainers = getWidgets(complianceGroup);
complianceContainerWidgets = getWidgets(complianceContainers(1));
complianceContainers(1).removeWidget(complianceContainerWidgets(1));
setMetricIDs(complianceContainerWidgets(1),...
({'mathworks.metrics.ModelAdvisorCompliance._SYSTEM_By Task_misra_c'}));
complianceContainerWidgets(1).Labels={'MISRA'};
```

Add a custom widget for visualizing MISRA check issues metrics to the `complianceContainers` `slmetric.dashboard.Container` object.

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

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

```
save(config,'FileName','MetricConfig.xml');
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configurations.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getWidths

Class: `slmetric.dashboard.Container`
Package: `slmetric.dashboard`

Obtain widths of Metrics Dashboard container

Syntax

```
Widths = getWidths(containerName)
```

Description

`Widths = getWidths(containerName)` returns an `slmetric.dashboard.Width` object array consisting of four enumerations. Use the `slmetric.dashboard.Container.setWidths` method to set the width sizes. You can set between one and four sizes. If you set just one size, the array contains four of the same enumerations. These are the possible enumeration values:

- `slmetric.dashboard.Width.ExtraSmall`
- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`
- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

These values correspond to the sizes that a container can have as the screen size changes. If the container has one value, the container always has the same size regardless of the screen size. If the container has four different values, the container size can change four times as you maximize or minimize the screen.

Input Arguments

`containerName` — Metrics Dashboard container

`slmetric.dashboard.Container` object

Container for which you want to obtain widths

Data Types: `char`

Output Arguments

`Widths` — Container widths

`slmetric.dashboard.Width` enumeration array

`slmetric.dashboard.Width` enumeration array consisting of between one and four of these values:

- `slmetric.dashboard.Width.ExtraSmall`

- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`
- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

Examples

Configure Compliance Metrics

You can use the Metrics Dashboard and metric APIs to obtain compliance and issues metric data on your Model Advisor configuration. To set up your Model Advisor configuration, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”. You can also use an existing check group such as the MISRA checks. 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:

Open the default configuration:

```
config=slmetric.config.Configuration.open()
```

Specify a metric family ID that you associate with those check groups:

```
famParamID = 'ModelAdvisorStandard';
```

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

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

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

To set up the configuration, pass the `values` cell array into the `setMetricFamilyParameterValues` method.

```
setMetricFamilyParameterValues(config, famParamID, values);
```

Point the **High Integrity Compliance** and **High Integrity Check Issues** widgets to MISRA check group. To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain the widget objects that are in the layout object.

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

Obtain the compliance group from the layout. This group contains two containers. The first container contains the High Integrity and MAB Compliance and Check Issues widgets. Remove the **High Integrity Compliance** widget.

```
complianceGroup = layoutWidget(3);
complianceContainers = getWidgets(complianceGroup);
complianceContainerWidgets = getWidgets(complianceContainers(1));
complianceContainers(1).removeWidget(complianceContainerWidgets(1));
setMetricIDs(complianceContainerWidgets(1),...
({'mathworks.metrics.ModelAdvisorCompliance._SYSTEM_By Task_misra_c'}));
complianceContainerWidgets(1).Labels={'MISRA'};
```

Add a custom widget for visualizing MISRA check issues metrics to the `complianceContainers` `slmetric.dashboard.Container` object.

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

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

```
save(config,'FileName','MetricConfig.xml');
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configurations.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

removeWidget

Class: `slmetric.dashboard.Container`
Package: `slmetric.dashboard`

Remove widget from `slmetric.dashboard.Container` object

Syntax

```
removeWidget(container,widget)
```

Description

`removeWidget(container,widget)` removes a widget from an `slmetric.dashboard.Container` object. You can remove these widgets from the Metrics Dashboard:

- `slmetric.dashboard.Group`
- `slmetric.dashboard.Container`
- `slmetric.dashboard.CustomWidget`
- `slmetric.dashboard.Widget`

Use the `getWidgets` method to identify widgets that you want to remove from an `slmetric.dashboard.Container` object.

Input Arguments

`group` — Remove widget from group in Metrics Dashboard

`slmetric.dashboard.Group` object

Remove widget object from an `slmetric.dashboad.Group` object.

`widget` — Widget that you want to remove from a `slmetric.dashboard.Group` object

`index of widget in array`

Widget object that you want to remove from an `slmetric.dashboard.Group` object. Apply the `removeWidget` method to the array index containing the widget that you want to remove from the group in the `slmetric.dashboard.Layout` object.

Examples

Configure Compliance Metrics

You can use the Metrics Dashboard and metric APIs to obtain compliance and issues metric data on your Model Advisor configuration. To set up your Model Advisor configuration, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”. You can also use an existing check group such as the MISRA checks. 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:

Open the default configuration:

```
config=slmetric.config.Configuration.open()
```

Specify a metric family ID that you associate with those check groups:

```
famParamID = 'ModelAdvisorStandard';
```

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

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

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

To set up the configuration, pass the `values` cell array into the `setMetricFamilyParameterValues` method.

```
setMetricFamilyParameterValues(config, famParamID, values);
```

Point the **High Integrity Compliance** and **High Integrity Check Issues** widgets to the MISRA check group. To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain the widget objects that are in the layout object.

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

Obtain the compliance group from the layout. This group contains two containers. The first container contains the High Integrity and MAB Compliance and Check Issues widgets. Remove the **High Integrity Compliance** widget.

```
complianceGroup = layoutWidget(3);
complianceContainers = getWidgets(complianceGroup);
complianceContainerWidgets = getWidgets(complianceContainers(1));
complianceContainers(1).removeWidget(complianceContainerWidgets(1));
setMetricIDs(complianceContainerWidgets(1),...
({'mathworks.metrics.ModelAdvisorCompliance._SYSTEM_By Task_misra_c'}));
complianceContainerWidgets(1).Labels = {'MISRA'};
```

Add a custom widget for visualizing MISRA check issues metrics to the `complianceContainers` `slmetric.dashboard.Container` object.

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

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

```
save(config,'FileName','MetricConfig.xml');
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configurations.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setSeparators

Class: `slmetric.dashboard.Container`
Package: `slmetric.dashboard`

Specify lines on Metrics Dashboard container sides

Syntax

```
setSeparators(S)
```

Description

`setSeparators(S)` specifies whether there are lines on the sides of an `slmetric.dashboard.Container` object.

Input Arguments

S — Structure of four Boolean values

`Structure | Array of Structures`

The input is a structure or an array of four structures consisting of these fields:

- `S.top`
- `S.bottom`
- `S.left`
- `S.right`

Each field must be set to `1` or `0`. A value of `1` indicates that there is a line on that container side. A value of `0` indicates that there is no line on that container side. To indicate that the container sides are always the same even if the screen size changes, you can pass one structure. Passing four structures indicates that the container sides can have different separators as the screen width size changes. Use the `setWidths` method to specify up to four different widths.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setWidths

Class: `slmetric.dashboard.Container`
Package: `slmetric.dashboard`

Specify multiple widths for Metrics Dashboard container

Syntax

```
setWidths(containerName, widths)
```

Description

`setWidths(containerName, widths)` specifies possible widths that an `slmetric.dashboard.Container` object can have. You can specify up to four different widths. For the input argument `widths`, pass either one value or an array of four values. You can choose from these possible values:

- `slmetric.dashboard.Width.ExtraSmall`
- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`
- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

These values correspond to the different sizes that a container can have as the screen size changes. If you specify one value, the container always has that value regardless of the screen size. If you specify four different values, the container size can change four times as you maximize and minimize the screen.

Input Arguments

`containerName` — Container that is to have between one and four widths

`slmetric.dashboard.Container` object

`slmetric.dashboard.Container` object that is to have between one and four widths

`widths` — Width array

`character vector` | `array of character vectors` | `string scalar` | `array of string scalars`

Specify one or as many as four of these values:

- `slmetric.dashboard.Width.ExtraSmall`
- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`

- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getSeparators

Class: `slmetric.dashboard.CustomWidget`
Package: `slmetric.dashboard`

Determine whether there are lines on sides of Metrics Dashboard custom widget

Syntax

```
S=getSeparators(customWid)
```

Description

`S=getSeparators(customWid)` returns a structure or an array of structures indicating whether there are lines on the sides of an `slmetric.dashboard.CustomWidget` object.

Input Arguments

`customWid` — Custom widget for which you want to know whether there are separators

`slmetric.dashboard.CustomWidget` object

Determine whether there are separators on the sides of an `slmetric.dashboard.CustomWidget` object.

Output Arguments

`S` — Structure of four fields

`Structure` | `Array of Structures`

The output is a structure or an array of structures consisting of these fields:

- `S.top`
- `S.bottom`
- `S.left`
- `S.right`

Each field is empty or has a value of `1` or `0`. An empty field indicates that you did not set a value. A value of `1` indicates that there is a line on that custom widget side. A value of `0` indicates that there is no line on that custom widget side.

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

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

    end

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

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

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



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

        isNonVirtual = true(size(blocks));

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

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

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

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

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

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

        blocks = blocks(isNonVirtual);

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

```
    end
end
```

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getWidths

Class: `slmetric.dashboard.CustomWidget`
Package: `slmetric.dashboard`

Obtain widths of Metrics Dashboard custom widget

Syntax

```
Widths=getWidths(customName)
```

Description

`Widths=getWidths(customName)` returns an `slmetric.dashboard.Width` object array consisting of four enumerations. Use the `slmetric.dashboard.CustomWidget.setWidths` method to set the width sizes. You can set between one and four sizes. If you set just one size, the array contains four of the same enumerations. These are the possible enumeration values:

- `slmetric.dashboard.Width.ExtraSmall`
- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`
- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

These values correspond to the sizes that a custom widget can have as the screen size changes. If the custom widget has one value, the custom widget always has the same size regardless of the screen size. If the custom widget has four different values, the custom widget size can change four times as you maximize and minimize the screen.

Input Arguments

`customName` — Metrics Dashboard custom widget

`slmetric.dashboard.CustomWidget` object

Custom widget for which you want to obtain widths

Data Types: `char`

Output Arguments

`Widths` — Custom widget widths

`slmetric.dashboard.Width` enumeration array

`slmetric.dashboard.Width` enumeration array consisting of between one and four of these values:

- `slmetric.dashboard.Width.ExtraSmall`

- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`
- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

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

    end

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

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

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



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

isNonVirtual = true(size(blocks));

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

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

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

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

```
                        isNonVirtual(n) = false;

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

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

            blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setSeparators

Class: `slmetric.dashboard.CustomWidget`
Package: `slmetric.dashboard`

Specify lines on Metrics Dashboard custom widget sides

Syntax

```
setSeparators(S)
```

Description

`setSeparators(S)` specifies whether there are lines on the sides of an `slmetric.dashboard.CustomWidget` object.

Input Arguments

S — Structure of four Boolean values

`Structure | Array of Structures`

The input is a structure or an array of structures consisting of these fields:

- `S.top`
- `S.bottom`
- `S.left`
- `S.right`

Each field must be set to `1` or `0`. A value of `1` indicates that there is a line on that custom widget side. A value of `0` indicates that there is no line on that custom widget side. To indicate that the custom widget sides are always the same even if the screen size changes, you can pass one structure. Passing four structures indicates that the custom widget sides can have different separators as the screen width size changes. Use the `setWidths` method to specify up to four different widths.

Data Types: `char`

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

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

    end

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

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

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



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

        isNonVirtual = true(size(blocks));

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

            if any(strcmp(this.VirtualBlockTypes, blockType))
                isNonVirtual(n) = false;
            else
                switch blockType
```

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

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

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

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

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

        blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setWidths

Class: `slmetric.dashboard.CustomWidget`
Package: `slmetric.dashboard`

Specify multiples widths for Metrics Dashboard custom widget

Syntax

```
setWidths(customName, widths)
```

Description

`setWidths(customName, widths)` specifies possible widths that an `slmetric.dashboard.CustomWidget` object can have. You can specify up to four different widths. For the input argument `widths`, pass either one value or an array of four values. You can choose from these possible values:

- `slmetric.dashboard.Width.ExtraSmall`
- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`
- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

These values correspond to the different sizes that a custom widget can have as the screen size changes. If you specify one value, the widget always has that value regardless of the screen size. If you specify four different values, the widget size can change four times as you maximize and minimize the screen.

Input Arguments

`customName` — Custom widget that is to have between one and four widths

`slmetric.dashboard.CustomWidget` object

`slmetric.dashboard.CustomWidget` object that is to have between one and four widths

`widths` — Width array

`character vector|array of character vectors|string scalar|array of string scalars`

Specify one or as many as four of these values:

- `slmetric.dashboard.Width.ExtraSmall`
- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`

- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

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

    end

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

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

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

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



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

isNonVirtual = true(size(blocks));

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

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

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

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

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

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

            blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

addWidget

Class: `slmetric.dashboard.Group`
Package: `slmetric.dashboard`

Add widget to `slmetric.dashboard.Group` object

Syntax

```
newWidget = addWidget(group,widgetType,num)
```

Description

`newWidget = addWidget(group,widgetType,num)` adds a widget to an `slmetric.dashboard.Container` object.

Input Arguments

`group` — Add widget to Metrics Dashboard

`slmetric.dashboard.Group` object

`slmetric.dashboard.Group` object for which you want to add widgets to customize Metrics Dashboard layout.

`widgetType` — Metrics Dashboard widget

`Container` | `SystemInfo` | `GlocalInterface` | `LibraryReuse` | `Custom`

Specify the `Type` of an `slmetric.dashboard.Container`, `slmetric.dashboard.Widget`, `slmetric.dashboard.Group`, or `slmetric.dashboard.CustomWidget` object. This property is read/write.

Data Types: `char`

`num` — Widget placement

`int`

Placement of widget in group on Metrics Dashboard. Order of widgets in the group proceeds from left to right, and then down in the group.

Output Arguments

`newWidget` — New Metrics Dashboard widget

`slmetric.dashboard.Widget` object

New widget that you are adding to an `slmetric.dashboard.Group` object on the Metrics Dashboard. You can add these widgets to a group:

- `slmetric.dashboard.Container`
- `slmetric.dashboard.CustomWidget`
- `slmetric.dashboard.Widget`

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

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

    end

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

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

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

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

isNonVirtual = true(size(blocks));

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

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

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

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

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

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

```
                        end
                    end
                end
            end
        end

        blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getSeparators

Class: `slmetric.dashboard.Group`
Package: `slmetric.dashboard`

Determine whether there are lines on sides of Metrics Dashboard group

Syntax

```
S = getSeparators(group)
```

Description

`S = getSeparators(group)` returns a structure or an array of structures indicating whether there are lines on the sides of an `slmetric.dashboard.Group` object.

Input Arguments

`group` — Group for which you want to know whether there are separators

`slmetric.dashboard.Group` object

Determine whether there are separators on the sides of an `slmetric.dashboard.Group` object.

Output Arguments

`S` — Structure of four fields

`Structure | Array of Structures`

The output is a structure or an array of structures consisting of these fields:

- `S.top`
- `S.bottom`
- `S.left`
- `S.right`

Each field is empty or has a value of `1` or `0`. An empty field indicates that you did not set a value. A value of `1` indicates that there is a line on that group side. A value of `0` indicates that there is no line on that group side.

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

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

    end

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

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

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



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

        isNonVirtual = true(size(blocks));

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

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

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

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

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

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

        blocks = blocks(isNonVirtual);

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

```
    end
end
```

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getWidgets

Class: `slmetric.dashboard.Group`
Package: `slmetric.dashboard`

Obtain a list of widgets in an `slmetric.dashboard.Group` object

Syntax

```
groupList = getWidgets(group)
```

Description

`groupList = getWidgets(group)` creates an array of objects that are in the `slmetric.dashboard.Group` object. These objects are widgets of the following types:

- `slmetric.dashboard.Container`
- `slmetric.dashboard.CustomWidget`
- `slmetric.dashboard.Widget`

Use the `getWidgets` method to identify widgets that you want to modify or remove from the `slmetric.dashboard.Group` object.

Input Arguments

`group` — Object that holds metric dashboard layout customizations

`slmetric.dashboard.Group` object

`slmetric.dashboard.Group` object for which you want to obtain a list of widgets.

Output Arguments

`group` — Array of objects in an `slmetric.dashboard.Group` object

array of objects

Array of widget objects in an `slmetric.dashboard.Group` object.

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

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

    end

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

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

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



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

        isNonVirtual = true(size(blocks));

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

            if any(strcmp(this.VirtualBlockTypes, blockType))
                isNonVirtual(n) = false;
```

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

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

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

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

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

        blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getWidths

Class: `slmetric.dashboard.Group`
Package: `slmetric.dashboard`

Obtain widths of Metrics Dashboard group

Syntax

```
Widths=getWidths(groupName)
```

Description

`Widths=getWidths(groupName)` returns an `slmetric.dashboard.Width` object array consisting of four enumerations. Use the `slmetric.dashboard.Group.setWidths` method to set the width sizes. You can set between one and four sizes. If you set just one size, the array contains four of the same enumerations. These are the possible enumeration values:

- `slmetric.dashboard.Width.ExtraSmall`
- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`
- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

These values correspond to the sizes that a group can have as the screen size changes. If the group has one value, the group always has the same size regardless of the screen size. If the group has four different values, the group size can change four times as you maximize and minimize the screen.

Input Arguments

`groupName` — Metrics Dashboard group

`slmetric.dashboard.Group` object

Group for which you want to obtain widths

Data Types: `char`

Output Arguments

`Widths` — Group widths

`slmetric.dashboard.Width` enumeration array

`slmetric.dashboard.Width` enumeration array consisting of between one and four of these values:

- `slmetric.dashboard.Width.ExtraSmall`
- `slmetric.dashboard.Width.Small`

- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`
- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

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

    end

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

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

        % Practice
        D1=slmetric.metric.ResultDetail('identifier 1','Name 1');
```

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



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

isNonVirtual = true(size(blocks));

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

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

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

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

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

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

            blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

removeWidget

Class: `slmetric.dashboard.Group`
Package: `slmetric.dashboard`

Remove widget from `slmetric.dashboard.Group` object

Syntax

```
removeWidget(group,widget)
```

Description

`removeWidget(group,widget)` removes a widget from an `slmetric.dashboard.Group` object. You can remove these widgets from the Metrics Dashboard:

- `slmetric.dashboard.Group`
- `slmetric.dashboard.Container`
- `slmetric.dashboard.Custom`
- `slmetric.dashboard.Widget`

Use the `getWidgets` method to identify widgets that you want to remove from an `slmetric.dashboard.Group` object.

Input Arguments

`group` — Remove widget from group in Metrics Dashboard

`slmetric.dashboard.Group` object

Remove widget object from an `slmetric.dashboad.Group` object.

`widget` — Widget that you want to remove from a `slmetric.dashboard.Group` object

`index of widget in array`

Widget object that you want to remove from an `slmetric.dashboard.Group` object. Apply the `removeWidget` method to the array index containing the widget that you want to remove from the group in the `slmetric.dashboard.Layout` object.

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

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

    end

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

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

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



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

        isNonVirtual = true(size(blocks));

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

            if any(strcmp(this.VirtualBlockTypes, blockType))
                isNonVirtual(n) = false;
```

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

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

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

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

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

        blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setSeparators

Class: `slmetric.dashboard.Group`
Package: `slmetric.dashboard`

Specify lines on Metrics Dashboard group sides

Syntax

```
setSeparators(S)
```

Description

`setSeparators(S)` specifies whether there are lines on the sides of an `slmetric.dashboard.Group` object.

Input Arguments

S — Structure of four Boolean values

`Structure | Array of Structures`

The input is a structure array consisting of these fields:

- `S.top`
- `S.bottom`
- `S.left`
- `S.right`

Each field must be set to `1` or `0`. A value of `1` indicates that there is a line on that group side. A value of `0` indicates that there is no line on that group side. To indicate that the group sides are always the same even if the screen size changes, you can pass one structure. Passing four structures indicates that the group sides can have different separators as the screen width size changes. Use the `setWidths` method to specify up to four different widths.

Data Types: `char`

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

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

    end

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

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

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



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

        isNonVirtual = true(size(blocks));

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

            if any(strcmp(this.VirtualBlockTypes, blockType))
                isNonVirtual(n) = false;
            else
                switch blockType
                    case 'SubSystem'
                        % Virtual unless the block is conditionally executed
```

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

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

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

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

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

        blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setWidths

Class: `slmetric.dashboard.Group`
Package: `slmetric.dashboard`

Specify multiple widths for Metrics Dashboard group

Syntax

```
setWidths(groupName, widths)
```

Description

`setWidths(groupName, widths)` specifies possible widths for an `slmetric.dashboard.Group` object. You can specify up to four different widths. For the input argument `widths`, pass either one value or an array of four values. You can choose from these possible values:

- `slmetric.dashboard.Width.ExtraSmall`
- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`
- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

These values correspond to the different sizes that a group can have as the screen size changes. If you specify one value, the group always has that value regardless of the screen size. If you specify four different values, the container size can change four times as you maximize and minimize the screen.

Input Arguments

`groupName` — Group that is to have between one and four widths
`slmetric.dashboard.Group` object

`slmetric.dashboard.Container` object that is to have between one and four widths

`widths` — Width array
`character vector|array of character vectors|string scalar|array of string scalars`

Specify one or as many as four of these values:

- `slmetric.dashboard.Width.ExtraSmall`
- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`
- `slmetric.dashboard.Width.XLarge`

- `slmetric.dashboard.Width.XXLarge`

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

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

    end

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

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

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

```
D2.setGroup('Group1','Group1Name');



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

isNonVirtual = true(size(blocks));

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

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

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

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

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

                    if ~isempty(pc.DstBlock) && ...
```

```
                                    strcmp(get_param(pc.DstBlock, 'BlockType'), ...
                                    'Outport')
                                isNonVirtual(n) = true;
                            end
                        end
                    end
                end
            end

            blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

addWidget

Class: `slmetric.dashboard.Layout`
Package: `slmetric.dashboard`

Add widget to `slmetric.dashboard.Layout` object

Syntax

```
newWidget = addWidget(dashboardLayout,widgetType,num)
```

Description

`newWidget = addWidget(dashboardLayout,widgetType,num)` adds a widget to an `slmetric.dashboard.Layout` object.

Input Arguments

`dashboardLayout` — Add widget to Metrics Dashboard

`slmetric.dashboard.Layout` object

`slmetric.dashboard.Layout` object for which you want to add widgets to customize Metrics Dashboard layout.

`widgetType` — Metrics Dashboard widget

`Group` | `Container` | `SystemInfo` | `GlocalInterface` | `LibraryReuse` | `Custom`

Specify the `Type` property of an `slmetric.dashboard.Container`, `slmetric.dashboard.Widget`, `slmetric.dashboard.Group`, or `slmetric.dashboard.CustomWidget` object.

Data Types: `char`

`num` — Widget placement

`int`

Placement of widget on Metrics Dashboard. Order of widgets proceeds from left to right, and then down.

Output Arguments

`newWidget` — New Metrics Dashboard widget

`slmetric.dashboard.Widget` object

New widget that you are adding to Metrics Dashboard. Choose from one of these widgets:

- `slmetric.dashboard.Group`
- `slmetric.dashboard.Container`
- `slmetric.dashboard.Widget`

- `slmetric.dashboard.Custom`

Examples

Configure Compliance Metrics

You can use the Metrics Dashboard and metric APIs to obtain compliance and issues metric data on your Model Advisor configuration. To set up your Model Advisor configuration, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”. You can also use an existing check group such as the MISRA checks. 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:

Open the default configuration:

```
config=slmetric.config.Configuration.open()
```

Specify a metric family ID that you associate with those check groups:

```
famParamID = 'ModelAdvisorStandard';
```

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

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

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

To set up the configuration, pass the `values` cell array into the `setMetricFamilyParameterValues` method .

```
setMetricFamilyParameterValues(config, famParamID, values);
```

Point the **High Integrity Compliance** and **High Integrity Check Issues** widgets to the MISRA check group. To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain the widget objects that are in the layout object.

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

Obtain the compliance group from the layout. This group contains two containers. The first container contains the High Integrity and MAB Compliance and Check Issues widgets. Remove the **High Integrity Compliance** widget.

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

```
complianceContainerWidgets = getWidgets(complianceContainers(1));
complianceContainers(1).removeWidget(complianceContainerWidgets(1));
setMetricIDs(complianceContainerWidgets(1),...
({'mathworks.metrics.ModelAdvisorCompliance._SYSTEM_By Task_misra_c'}));
complianceContainerWidgets(1).Labels = {'MISRA'};
```

Add a custom widget for visualizing MISRA check issues metrics to the `complianceContainers` `slmetric.dashboard.Container` object.

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

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

```
save(config,'FileName','MetricConfig.xml');
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configurations.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getWidgets

Class: `slmetric.dashboard.Layout`
Package: `slmetric.dashboard`

Obtain a list of widgets in an `slmetric.dashboard.Layout` object

Syntax

```
Layout = getWidgets(dashboardLayout)
```

Description

`Layout = getWidgets(dashboardLayout)` creates an array of objects that are in the `slmetric.dashboard.Layout` object. These objects are widgets of the following types:

- `slmetric.dashboard.Group`
- `slmetric.dashboard.Container`
- `slmetric.dashboard.Widget`
- `slmetric.dashboard.CustomWidget`

Use the `getWidgets` method to identify widgets that you want to modify or remove from the `slmetric.dashboard.Layout` object.

Input Arguments

`dashboardLayout` — Object that holds Metrics Dashboard layout customizations

`slmetric.dashboard.Layout` object

`slmetric.dashboard.Layout` object for which you want to obtain a list of widgets.

Data Types: `char`

Output Arguments

`Layout` — Array of objects in `slmetric.dashboard.Layout` object

array of objects

Array of objects in `slmetric.dashboard.Layout` object.

Examples

Configure Compliance Metrics

You can use the Metrics Dashboard and metric APIs to obtain compliance and issues metric data on your Model Advisor configuration. To set up your Model Advisor configuration, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”. You can also use an existing check group such as the MISRA checks. 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:

Open the default configuration:

```
config=slmetric.config.Configuration.open()
```

Specify a metric family ID that you associate with those check groups:

```
famParamID = 'ModelAdvisorStandard';
```

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

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

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

To set up the configuration, pass the `values` cell array into the `setMetricFamilyParameterValues` method.

```
setMetricFamilyParameterValues(config, famParamID, values);
```

Point the **High Integrity Compliance** and **High Integrity Check Issues** widgets to the MISRA check group. To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain the widget objects that are in the layout object.

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

Obtain the compliance group from the layout. This group contains two containers. The first container contains the High Integrity and MAB Compliance and Check Issues widgets. Remove the **High Integrity Compliance** widget.

```
complianceGroup = layoutWidget(3);
complianceContainers = getWidgets(complianceGroup);
complianceContainerWidgets = getWidgets(complianceContainers(1));
complianceContainers(1).removeWidget(complianceContainerWidgets(1));
setMetricIDs(complianceContainerWidgets(1),...
({'mathworks.metrics.ModelAdvisorCompliance._SYSTEM_By Task_misra_c'}));
complianceContainerWidgets(1).Labels={'MISRA'};
```

Add a custom widget for visualizing MISRA check issues metrics to the `complianceContainers` `slmetric.dashboard.Container` object.

```
misraWidget = complianceContainers(1).addWidget('Custom', 1);
misraWidget.Title=('MISRA');
misraWidget.VisualizationType = 'RadialGauge';
```

```
misraWidget.setMetricIDs('mathworks.metrics.ModelAdvisorCheckCompliance._SYSTEM_By Task_misra_c');
misraWidget.setWidths(slmetric.dashboard.Width.Medium);
```

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

```
save(config,'FileName','MetricConfig.xml');
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configurations.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

removeWidget

Class: `slmetric.dashboard.Layout`
Package: `slmetric.dashboard`

Remove widget from `slmetric.dashboard.Layout` object

Syntax

```
removeWidget(dashboardLayout,widget in array)
```

Description

`removeWidget(dashboardLayout,widget in array)` removes a widget from an `slmetric.dashboard.Layout` object. You can remove these widgets from the Metrics Dashboard:

- `slmetric.dashboard.Group`
- `slmetric.dashboard.Container`
- `slmetric.dashboard.CustomWidget`
- `slmetric.dashboard.Widget`

Use the `getWidgets` method to identify widgets that you want to remove from a `slmetric.dashboard.Layout` object.

Input Arguments

`dashboardlayout` — Remove widget from Metrics Dashboard

`slmetric.dashboard.Layout` object

Remove widget object from an `slmetric.dashboad.Layout` object.

`widget` — Widget to remove from an `slmetrics.dashboard.Layout` object

`index of widget in array`

Widget object that you want to remove from an `slmetric.dashboard.layout` object. Use the `getWidgets` method to return an array of widgets in the `slmetrics.dashboard.layout` object. Apply the `removeWidget` method to the array index containing the widget that you want to remove from the Metrics Dashboard.

Examples

Configure Compliance Metrics

You can use the Metrics Dashboard and metric APIs to obtain compliance and issues metric data on your Model Advisor configuration. To set up your Model Advisor configuration, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”. You can also use an existing check group such as the MISRA checks. 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:

Open the default configuration:

```
config = slmetric.config.Configuration.open()
```

Specify a metric family ID that you associate with those check groups:

```
famParamID = 'ModelAdvisorStandard';
```

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

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

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

To set up the configuration, pass the `values` cell array into the `setMetricFamilyParameterValues` method.

```
setMetricFamilyParameterValues(config, famParamID, values);
```

Point the **High Integrity Compliance** and **High Integrity Check Issues** widgets to the MISRA check group. To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain the widget objects that are in the layout object.

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

Obtain the compliance group from the layout. This group contains two containers. The first container contains the High Integrity and MAB Compliance and Check Issues widgets. Remove the **High Integrity Compliance** widget.

```
complianceGroup = layoutWidget(3);
complianceContainers = getWidgets(complianceGroup);
complianceContainerWidgets = getWidgets(complianceContainers(1));
complianceContainers(1).removeWidget(complianceContainerWidgets(1));
setMetricIDs(complianceContainerWidgets(1),...
({'mathworks.metrics.ModelAdvisorCompliance._SYSTEM_By Task_misra_c'}));
complianceContainerWidgets(1).Labels={'MISRA'};
```

Add a custom widget for visualizing MISRA check issues metrics to the `complianceContainers` `slmetric.dashboard.Container` object.

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

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

```
save(config,'FileName','MetricConfig.xml');
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configurations.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getSeparators

Class: `slmetric.dashboard.Widget`
Package: `slmetric.dashboard`

Determine whether there are lines on sides of Metrics Dashboard widget

Syntax

```
S=getSeparators(widget)
```

Description

`S=getSeparators(widget)` returns a structure or an array of structures indicating whether there are lines on the sides of an `slmetric.dashboard.Widget` object.

Input Arguments

`widget` — Widget for which you want to know whether there are separators

`slmetric.dashboard.Widget` object

Determine whether there are separators on the sides of an `slmetric.dashboard.Widget` object.

Output Arguments

`S` — Structure of four fields

`Structure` | `Array of Structures`

The structure array contains these fields:

- `S.top`
- `S.bottom`
- `S.left`
- `S.right`

Each field is empty or has a value of `1` or `0`. An empty field indicates that you did not set a value. A value of `1` indicates that there is a line on that widget side. A value of `0` indicates that there is no line on that widget side.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` |
`slmetric.dashboard.getActiveConfiguration`

Topics
“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getWidths

Class: `slmetric.dashboard.Widget`
Package: `slmetric.dashboard`

Obtain widths of Metrics Dashboard widget

Syntax

```
Widths=getWidths(widgetName)
```

Description

`Widths=getWidths(widgetName)` returns an `slmetric.dashboard.Width` object array consisting of four enumerations. Use the `slmetric.dashboard.Widgets.setWidths` method to set the width sizes. You can set between one and four sizes. If you set just one size, the array contains four of the same enumerations. These are the possible enumeration values:

- `slmetric.dashboard.Width.ExtraSmall`
- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`
- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

These values correspond to the sizes that a widget can have as the screen size changes. If the widget has one value, the widget always has the same size regardless of the screen size. If the widget has four different values, the widget size can change four times as you maximize and minimize the screen.

Input Arguments

`widgetName` — Metrics Dashboard widget

`slmetric.dashboard.Widget` object

Widget for which you want to obtain widths.

Data Types: `char`

Output Arguments

`Widths` — Widget widths

`slmetric.dashboard.Width` enumeration array

`slmetric.dashboard.Width` enumeration array consisting of between one and four of these values:

- `slmetric.dashboard.Width.ExtraSmall`

- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`
- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setSeparators

Class: `slmetric.dashboard.Widget`
Package: `slmetric.dashboard`

Specify lines on Metrics Dashboard widget sides

Syntax

```
setSeparators(S)
```

Description

`setSeparators(S)` specifies whether there are lines on the sides of an `slmetric.dashboard.Widget` object.

Input Arguments

S — Structure of four Boolean values

`Structure array`

The input is a structure array consisting of these fields:

- `S.top`
- `S.bottom`
- `S.left`
- `S.right`

Each field must be set to `1` or `0`. A value of `1` indicates that there is a line on that widget side. A value of `0` indicates that there is no line on that widget side. To indicate that the widget sides are always the same even if the screen size changes, you can pass one structure. Passing four structures indicates that the widget sides can have different separators as the screen width size changes. Use the `setWidths` method to specify up to four different widths.

Data Types: `char`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setWidths

Class: `slmetric.dashboard.Widget`
Package: `slmetric.dashboard`

Specify multiple widths for Metrics Dashboard widget

Syntax

```
setWidths(widgetName, widths)
```

Description

`setWidths(widgetName, widths)` specifies possible widths that an `slmetric.dashboard.Widget` object can have. You can specify up to four different widths. For the input argument `widths`, pass either one value or an array of four values. You can choose from these possible values:

- `slmetric.dashboard.Width.ExtraSmall`
- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`
- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

These values correspond to the different sizes that a widget can have as the screen size changes. If you specify one value, the widget always has that value regardless of the screen size. If you specify four different values, the widget size can change four times as you maximize and minimize the screen.

Input Arguments

`widgetName` — Widget that is to have between one and four widths

`slmetric.dashboard.Widget` object

`slmetric.dashboard.Widget` object that is to have between one and four widths

`widths` — Width array

`character vector|array of character vectors|string scalar|array of string scalars`

Specify one or as many as four of these values:

- `slmetric.dashboard.Width.ExtraSmall`
- `slmetric.dashboard.Width.Small`
- `slmetric.dashboard.Width.Medium`
- `slmetric.dashboard.Width.Large`

- `slmetric.dashboard.Width.XLarge`
- `slmetric.dashboard.Width.XXLarge`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.dashboard.getActiveConfiguration

Package: `slmetric.dashboard`

Obtain file path and name of XML file containing active Metrics Dashboard layout

Syntax

```
Path = slmetric.dashboard.getActiveConfiguration
```

Description

`Path = slmetric.dashboard.getActiveConfiguration` returns the file path and name of the active Metrics Dashboard layout XML file. This file contains information on the location, size, and types of widgets in the Metrics Dashboard.

Examples

Get Default Metrics Dashboard Layout

At the MATLAB command line, enter this command to get the active metric dashboard layout:

```
slmetric.dashboard.getActiveConfiguration();
```

Output Arguments

Path — File path to XML file

character vector | string scalar

Full file path to folder containing XML file that contains the active Metrics Dashboard layout.

Note Passing an empty string to the `slmetric.dashboard.setActionConfiguration` function (that is, `slmetric.dasbhoard.setActiveConfiguration('')`), resets the configuration to the default, which is the shipping configuration. If you then enter the `slmetric.dashboard.getActiveConfiguration` method, the method returns an empty array.

Data Types: `char`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.Configuration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”

“Customize Metrics Dashboard Layout and Functionality”

slmetric.dashboard.setActiveConfiguration

Package: `slmetric.dashboard`

Activate custom metric dashboard layout

Syntax

```
slmetric.dashboard.setActiveConfiguration(fullfile)
```

Description

`slmetric.dashboard.setActiveConfiguration(fullfile)` sets a custom Metrics Dashboard layout as the default configuration. When you collect metric data by using the Metrics Dashboard, the metric engine uses this custom layout.

Note Passing an empty string to this function (that is, `slmetric.dashboard.setActiveConfiguration('')`), resets the configuration to the default, shipping configuration.

Examples

Activate Custom Configuration

At the MATLAB command line, enter this command to set the active metric configuration:

```
slmetric.config.setActiveConfiguration('C:\temp\MyConfig.xml');
```

Input Arguments

`fullfile` — File path to XML file

character vector | string scalar

Full file path to folder containing XML file that contains Metrics Dashboard custom configurations.

Example: 'C:\temp\MyConfig.xml'

Data Types: `char`

Version History

Introduced in R2018b

See Also

`slmetric.config.Configuration` | `slmetric.config.getActiveConfiguration`

External Websites

“Collect and Explore Metric Data by Using the Metrics Dashboard”

“Customize Metrics Dashboard Layout and Functionality”

setMargin

Class: `slmetric.dashboard.Container`
Package: `slmetric.dashboard`

Specify distance from container edge to its contents

Syntax

```
pixels = setMargin(Container,px)
```

Description

`pixels = setMargin(Container,px)` specifies how far in pixels the edges of an `slmetric.dashboard.Container` object is from the widgets that it contains.

Input Arguments

`Container` — Metrics Dashboard container

`slmetric.dashboard.Container` object

The `slmetric.dashboard.Container` object for which you are specifying margin size in pixels.

Data Types: `char`

px — Container margins

`character vector` | `string scalar`

Margin distance from container contents in pixels.

Example: `'40 px'`

Data Types: `char`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setMetricIDs

Class: `slmetric.dashboard.CustomWidget`
Package: `slmetric.dashboard`

Set metric identifier for custom Metrics Dashboard widget

Syntax

```
setMetricIDs(CustomWidget, metricID)
```

Description

`setMetricIDs(CustomWidget, metricID)` assigns a metric identifier to an `slmetric.dashboard.CustomWidget` object.

Input Arguments

`CustomWidget` — Custom widget object

`slmetric.dashboard.CustomWidget` object

`slmetric.dashboard.CustomWidget` object for which you want to assign a metric identifier. The `slmetric.dashboard.CustomWidget` object is the means of visualizing metric data for the metric identifier.

Data Types: `char`

`metricID` — Metric identifier

`character vector` `string scalar`

Metric identifier associated with an `slmetric.dashboard.CustomWidget` object.

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

```
        'GotoTagVisiblity','Mux','SignalSpecification', ...
        'Terminator','Inport'};
end

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

end

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

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

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



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

    isNonVirtual = true(size(blocks));

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

        if any(strcmp(this.VirtualBlockTypes, blockType))
            isNonVirtual(n) = false;
        else
            switch blockType
                case 'SubSystem'
                    % Virtual unless the block is conditionally executed
                    % or the Treat as atomic unit check box is selected.
                    if strcmp(get_param(blocks{n}, 'IsSubSystemVirtual'), ...
                            'on')
```

```
                            isNonVirtual(n) = false;
                        end
                    case 'Outport'
                        % Outport: Virtual when the block resides within
                        % SubSystem block (conditional or not), and
                        % does not reside in the root (top-level) Simulink window.
                        if component.Type ~= Advisor.component.Types.Model
                            isNonVirtual(n) = false;
                        end
                    case 'Selector'
                        % Virtual only when Number of input dimensions
                        % specifies 1 and Index Option specifies Select
                        % all, Index vector (dialog), or Starting index (dialog).
                        nod = get_param(blocks{n}, 'NumberOfDimensions');
                        ios = get_param(blocks{n}, 'IndexOptionArray');

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

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

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

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

        blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For a model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics
“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getMetricIDs

Class: `slmetric.dashboard.Widget`
Package: `slmetric.dashboard`

Obtain metric identifier for Metrics Dashboard widget

Syntax

```
metricID = getMetricIDs(widget)
```

Description

`metricID = getMetricIDs(widget)` returns the metric identifer for an `slmetric.dashboard.Widget` object.

Input Arguments

`widget` — Widget object

`slmetric.dashboard.Widget` object

`slmetric.dashboard.Widget` object for which you want to obtain the associated metric identifer. The `slmetric.dashboard.Widget` object is the means of visualizing metric data for the metric identifier.

Data Types: `char`

Output Arguments

`metricID` — Metric identifier

`character vector` | `string scalar`

Metric identifier associated with an `slmetric.dashboard.Widget` object.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getMetricFamilyParameterValues

Class: `slmetric.config.Configuration`
Package: `slmetric.config`

Obtain metric family Check Group IDs

Syntax

```
ArraysValue = getMetricFamilyParameterValues(config,...
'ModelAdvisorStandard')
```

Description

For an `slmetric.config.Configuration` object, use the `ArraysValue = getMetricFamilyParameterValues(config,... 'ModelAdvisorStandard')` method to obtain the metric family parameter values. These values are the Check Group IDs corresponding to the check groups for which you obtain compliance and issues metric data. Compliance metric data is the percentage of passed checks. Issues metric data is the number of check issues.

Input Arguments

`config` — Configuration object

`slmetric.config.Configuration` object

`slmetric.config.Configuration` object for which to obtain checks groups that have compliance and issues metric data.

`'ModelAdvisorStandard'` — Required string

character vector | string scalar

String that you must supply as an input.

Output Arguments

`ValuesArray` — Metric family parameter values

cell array of character vectors | cell array of string scalars

Cell array of metric family parameter values. For an `slmetric.config.Configuration` object, these values are the check groups for obtaining compliance and issues metric data.

Examples

Obtain Compliance and Issues Data for Groups of Model Advisor Checks

Obtain compliance and issues metric data on the Modeling Standards for MISRA C:2012, MAB, and High-Integrity Systems check groups.

Open the default configuration.

```
config = slmetric.config.Configuration.open();
```

Specify the metric family parameter ID, `famParamID`, and the metric family parameter values, `values`. The values `maab` and `hisl_do178` correspond to subsets of MAAB checks and High-Integrity System checks. The MISRA value `_SYSTEM_By Task_misra_c` is the **Check Group ID** for the MISRA check group `Modeling Standards for MISRA C:2012`.

```
famParamID = 'ModelAdvisorStandard';
values = {'maab', 'hisl_do178', '_SYSTEM_By Task_misra_c'};
setMetricFamilyParameterValues(config, famParamID, values);
```

To obtain the Model Advisor **Check Group ID** for a group of checks, open the Model Advisor Configuration Editor and select the desired folder 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”.

Check the metric family parameter values associated with the `slmetric.config.Configuration` object.

```
ValuesArray = getMetricFamilyParameterValues(config, famParamID);
```

This code is for the `ValuesArray` cell array:

```
ValuesArray =

  3×1 cell array

    {'_SYSTEM_By Task_misra_c'}
    {'hisl_do178'             }
    {'maab'                   }
```

Save the new configuration.

```
config.save('FileName', 'MetricConfig.xml');
```

Set the active Metrics Dashboard configuration.

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

For more information, see “Customize Metrics Dashboard Layout and Functionality”.

Version History

Introduced in R2018b

See Also

`slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration` | `slmetric.config.Configuration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

isMetricFamilyParameterParameterized

Class: `slmetric.config.Configuration`
Package: `slmetric.config`

Determine whether Metrics Dashboard configuration object has metric family parameter values

Syntax

```
ParameterizedConfig = isMetricFamilyParameterParameterized(config,...
'ModelAdvisorStandard')
```

Description

For an `slmetric.config.Configuration` object, use the `ParameterizedConfig = isMetricFamilyParameterParameterized(config,... 'ModelAdvisorStandard')` method to determine whether an `slmetric.config.Configuration` object contains metric family parameter values. These values are the Check Group IDs corresponding to the check groups for which you obtain compliance and issues metric data. Compliance metric data is the percentage of passed checks. Issues metric data is the number of check issues.

Input Arguments

`config` — Configuration object

`slmetric.config.Configuration` object

`slmetric.config.Configuration` object for which to obtain checks groups that have compliance and issues metric data.

`'ModelAdvisorStandard'` — Required string

character vector | string scalar

Standard string that you must supply as an input.

Output Arguments

`ParameterizedConfig` — Determine whether Metrics Dashboard configuration object has metric family parameter values

boolean

Determine whether a Metrics Dashboard configuration object has metric family parameter values.

Data Types: `Logical`

Examples

Obtain Compliance and Issues Data for Groups of Model Advisor Checks

Obtain compliance and issues metric data on the Modeling Standards for MISRA C:2012, MAB, and High-Integrity Systems check groups.

Open the default configuration.

```
config = slmetric.config.Configuration.open();
```

Specify the metric family parameter ID, `famParamID`, and the metric family parameter values, `values`. The values `maab` and `hisl_do178` correspond to subsets of MAAB checks and High-Integrity System checks. The MISRA value `_SYSTEM_By Task_misra_c` is the **Check Group ID** for the MISRA check group `Modeling Standards for MISRA C:2012`.

```
famParamID = 'ModelAdvisorStandard';
values = {'maab', 'hisl_do178', '_SYSTEM_By Task_misra_c'};
setMetricFamilyParameterValues(config, famParamID, values);
```

To obtain the Model Advisor **Check Group ID** for a group of checks, open the Model Advisor Configuration Editor and select the desired folder 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”.

Check that the `slmetric.config.Configuration` object has metric family parameter values.

```
PC = isMetricFamilyParameterParameterized(config, famParamID);
```

```
PC =

  logical

   1
```

Save the new configuration.

```
config.save('FileName', 'MetricConfig.xml');
```

Set the active Metrics Dashboard configuration.

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

For more information, see “Customize Metrics Dashboard Layout and Functionality”.

Version History

Introduced in R2018b

See Also

`slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration` | `slmetric.config.Configuration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

resetMetricFamilyParameterValues

Class: `slmetric.config.Configuration`
Package: `slmetric.config`

Clear metric family parameter values

Syntax

```
resetMetricFamilyParameterValues(config,... 'ModelAdvisorStandard')
```

Description

For an `slmetric.config.Configuration` object, use the `resetMetricFamilyParameterValues(config,... 'ModelAdvisorStandard')` method to clear the metric family parameter values. These values are the Check Group IDs corresponding to the check groups for which you obtain compliance and issues metric data. Compliance metric data is the percentage of passed checks. Issues metric data is the number of check issues.

Input Arguments

`config` — Configuration object
`slmetric.config.Configuration` object

`slmetric.config.Configuration` object for which to clear the metric family parameter values.

`'ModelAdvisorStandard'` — Required string
character vector | string scalar

Standard string that you must supply as an input.

Examples

Reset Metric Family Parameter Values

Obtain compliance and issues metric data on the Modeling Standards for MISRA C:2012, MAB, and High-Integrity Systems check groups.

Open the default configuration.

```
config = slmetric.config.Configuration.open();
```

Specify the metric family parameter ID, `famParamID`, and the metric family parameter values, `values`. The values `maab` and `hisl_do178` correspond to subsets of MAAB checks and High-Integrity System checks. The MISRA value `_SYSTEM_By Task_misra_c` is the **Check Group ID** for the MISRA check group `Modeling Standards for MISRA C:2012`.

```
famParamID = 'ModelAdvisorStandard';
values = {'maab', 'hisl_do178', '_SYSTEM_By Task_misra_c'};
setMetricFamilyParameterValues(config, famParamID, values);
```

To obtain the Model Advisor **Check Group ID** for a group of checks, open the Model Advisor Configuration Editor and select the desired folder 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”.

Check the metric family parameter values associated with the `slmetric.config.Configuration` object.

```
ValuesArray = getMetricFamilyParameterValues(config, famParamID);
```

This code is for the `ValuesArray` cell array:

```
ValuesArray =

  3×1 cell array

    {'_SYSTEM_By Task_misra_c'}
    {'hisl_do178'             }
    {'maab'                   }
```

Reset the values.

```
resetMetricFamilyParameterValues(config, famParamID)
```

Check that the `slmetric.config.Configuration` object does have associated metric family parameter values.

```
ValuesArray = getMetricFamilyParameterValues(config, famParamID);
```

For more information, see “Customize Metrics Dashboard Layout and Functionality”.

Version History

Introduced in R2018b

See Also

`slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration` | `slmetric.config.Configuration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setMetricFamilyParameterValues

Class: `slmetric.config.Configuration`
Package: `slmetric.config`

Obtain compliance and issues metric data on your Model Advisor configuration

Syntax

```
setMetricFamilyParameterValues(config,... 'ModelAdvisorStandard', values)
```

Description

Use the Model Advisor Configuration Editor to create groups of Model Advisor checks or use a shipped check group. Then, use the `setMetricFamilyParameterValues(config,... 'ModelAdvisorStandard', values)` method to obtain compliance and issues data for this group and any other groups that you specify as part of the `values` input. Compliance data is the percentage of passed checks. Issues data is the number of check issues. The `values` input sets the groups that are members of the family that you associate with a particular `slmetric.config.Configuration` object.

Input Arguments

`config` — Configuration object

`slmetric.config.Configuration` object

`slmetric.config.Configuration` object to add check groups for which to obtain compliance and issues data.

`'ModelAdvisorStandard'` — Required string

character vector | string scalar

Standard string that you must supply as an input.

`values` — Cell array of Check Group IDs

cell array of character vectors | cell array of string scalars

Specify Check Group IDs for each group of Model Advisor checks for which to obtain compliance and issues metric data. Obtain the Check Group IDs by opening up the Model Advisor Configuration Editor and selecting the folder that contains the group of checks. The **Check Group ID** parameter is in the folder.

Examples

Obtain Compliance and Issues Data for Groups of Model Advisor Checks

Obtain compliance and issues data on the Modeling Standards for MISRA C:2012, MAB, and High-Integrity Systems check groups.

Open the default configuration.

```
config = slmetric.config.Configuration.open();
```

Specify the metric family parameter ID, `famParamID`, and the metric family parameter values, `values`. The values `maab` and `hisl_do178` correspond to subsets of MAAB checks and High-Integrity System checks. The MISRA value `_SYSTEM_By Task_misra_c` is the **Check Group ID** for the MISRA check group `Modeling Standards for MISRA C:2012`.

```
famParamID = 'ModelAdvisorStandard';
values = {'maab', 'hisl_do178', '_SYSTEM_By Task_misra_c'};
setMetricFamilyParameterValues(config, famParamID, values);
```

To obtain the Model Advisor **Check Group ID** for a group of checks, open the Model Advisor Configuration Editor and select the desired folder 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”.

Save the new configuration.

```
config.save('FileName', 'MetricConfig.xml');
```

Set the active Metrics Dashboard configuration.

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

For more information, see “Customize Metrics Dashboard Layout and Functionality”.

Version History

Introduced in R2018b

See Also

`slmetric.config.Configuration` | `slmetric.config.getActiveConfiguration` | `slmetric.config.setActiveConfiguration`

Topics

“Collect and Explore Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getMargin

Class: `slmetric.dashboard.Container`
Package: `slmetric.dashboard`

Obtain distance from container edge to its contents

Syntax

```
pixels = getMargin(Container)
```

Description

`pixels = getMargin(Container)` returns how far in pixels the edges of an `slmetric.dashboard.Container` object is from the widgets that it contains.

Input Arguments

`Container` — Metrics Dashboard container

`slmetric.dashboard.Container` object

The `slmetric.dashboard.Container` object for which you are obtaining the margin distance.

Output Arguments

`pixels` — Container margins

`character vector` | `string scalar`

Margin distance from container contents in pixels.

Example: `'40 px'`

Data Types: `char`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getPosition

Class: `slmetric.dashboard.Container`
Package: `slmetric.dashboard`

Obtain container position within Metrics Dashboard

Syntax

```
Num = getPosition(Container)
```

Description

`Num = getPosition(Container)` returns the position of an `slmetric.dashboard.Container` object in an array that holds Metrics Dashboard objects. These objects are in an `slmetric.dashboard.Layout`, an `slmetric.dashboard.Container`, or an `slmetric.dashboard.Group` object. The order of containers in the array corresponds to proceeding from left to right, and then down in the Metrics Dashboard.

Input Arguments

`Container` — Metrics Dashboard container

`slmetric.dashboard.Container` object

Specify the `slmetric.dashboard.Container` object for which you get its position in the array.

Output Arguments

`Num` — Position of container object

`double`

Position of `slmetric.dashboard.Container` object within an array that holds the Metrics Dashboard objects in an `slmetric.dashboard.Layout`, an `slmetric.dashboard.Container`, or an `slmetric.dashboard.Group` object.

Data Types: `double`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setPosition

Class: `slmetric.dashboard.Container`
Package: `slmetric.dashboard`

Set container position within Metrics Dashboard

Syntax

```
setPosition(Container, num)
```

Description

`setPosition(Container, num)` sets the position of an `slmetric.dashboard.Container` object in an array that holds Metrics Dashboard objects. This array contains the Metrics Dashboard objects in an `slmetric.dashboard.Layout`, an `slmetric.dashboard.Container`, or an `slmetric.dashboard.Group` object. The order of containers in the array corresponds to proceeding from left to right, and then down in the Metrics Dashboard.

Input Arguments

`Container` — Metrics Dashboard container

`slmetric.dashboard.Container` object

Specify the `slmetric.dashboard.Container` object for which you set its position in the array.

Output Arguments

`Num` — Position of container object

`double`

Position of `slmetric.dashboard.Container` object within an array that holds the Metrics Dashboard objects in either an `slmetric.dashboard.Layout`, an `slmetric.dashboard.Container`, or an `slmetric.dashboard.Group` object.

Data Types: `double`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getHeight

Class: `slmetric.dashboard.CustomWidget`
Package: `slmetric.dashboard`

Obtain height of Metrics Dashboard custom widget

Syntax

```
Height = getHeight(CustomWidget)
```

Description

`Height = getHeight(CustomWidget)` returns the height of a custom widget in pixels.

Input Arguments

`CustomWidget` — Metrics Dashboard custom widget

`slmetric.dashboard.CustomWidget` object

`slmetric.dashboard.CustomWidget` for which you want to obtain its height.

Output Arguments

`num` — Height in pixels

integer

Height of `slmetric.dashboard.CustomWidget` object in pixels.

Example: `Height = getHeight(CustomWidget)`

Data Types: `uint32`

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

```
classdef nonvirtualblockcount < slmetric.metric.Metric
    %nonvirtualblockcount calculates number of nonvirtual blocks per level.
    % BusCreator, BusSelector and BusAssign are treated as nonvirtual.
```

```
properties
    VirtualBlockTypes = {'Demux','From','Goto','Ground', ...
        'GotoTagVisiblity','Mux','SignalSpecification', ...
        'Terminator','Inport'};
end

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

end

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

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

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



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

    isNonVirtual = true(size(blocks));

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

        if any(strcmp(this.VirtualBlockTypes, blockType))
            isNonVirtual(n) = false;
        else
            switch blockType
                case 'SubSystem'
                    % Virtual unless the block is conditionally executed
                    % or the Treat as atomic unit check box is selected.
```

```
                        if strcmp(get_param(blocks{n}, 'IsSubSystemVirtual'), ...
                                'on')
                            isNonVirtual(n) = false;
                        end
                    case 'Outport'
                        % Outport: Virtual when the block resides within
                        % SubSystem block (conditional or not), and
                        % does not reside in the root (top-level) Simulink window.
                        if component.Type ~= Advisor.component.Types.Model
                            isNonVirtual(n) = false;
                        end
                    case 'Selector'
                        % Virtual only when Number of input dimensions
                        % specifies 1 and Index Option specifies Select
                        % all, Index vector (dialog), or Starting index (dialog).
                        nod = get_param(blocks{n}, 'NumberOfDimensions');
                        ios = get_param(blocks{n}, 'IndexOptionArray');

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

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

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

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

        blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics
“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getMetricIDs

Class: `slmetric.dashboard.CustomWidget`
Package: `slmetric.dashboard`

Obtain metric identifier for custom Metrics Dashboard widget

Syntax

```
metricID = getMetricIDs(CustomWidget)
```

Description

`metricID = getMetricIDs(CustomWidget)` returns the metric identifier for an `slmetric.dashboard.CustomWidget` object.

Input Arguments

`CustomWidget` — Custom widget object

`slmetric.dashboard.CustomWidget` object

`slmetric.dashboard.CustomWidget` object for which you want to obtain the associated metric identifier. The `slmetric.dashboard.CustomWidget` object is the means of visualizing metric data for the metric identifier.

Data Types: `char`

Output Arguments

`metricID` — Metric identifier

`character vectorstring scalar`

Metric identifier associated with an `slmetric.dashboard.CustomWidget` object.

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

```
classdef nonvirtualblockcount < slmetric.metric.Metric
    %nonvirtualblockcount calculates number of nonvirtual blocks per level.
```

```
% BusCreator, BusSelector and BusAssign are treated as nonvirtual.
properties
    VirtualBlockTypes = {'Demux','From','Goto','Ground', ...
        'GotoTagVisiblity','Mux','SignalSpecification', ...
        'Terminator','Inport'};
end

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

end

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

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

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



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

    isNonVirtual = true(size(blocks));

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

        if any(strcmp(this.VirtualBlockTypes, blockType))
            isNonVirtual(n) = false;
        else
            switch blockType
                case 'SubSystem'
                    % Virtual unless the block is conditionally executed
```

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

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

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

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

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

        blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getPosition

Class: `slmetric.dashboard.CustomWidget`
Package: `slmetric.dashboard`

Obtain custom widget position within Metrics Dashboard

Syntax

```
Num = getPosition(CustomWidget)
```

Description

`Num = getPosition(CustomWidget)` returns the position of an `slmetric.dashboard.CustomWidget` object within an array that holds Metrics Dashboard objects. These objects are in an `slmetric.dashboard.Layout`, an `slmetric.dashboard.Group`, or an `slmetric.dashboard.Container` object. The order of objects in the array corresponds to proceeding from left to right, and then down in the Metrics Dashboard.

Input Arguments

`CustomWidget` — Metrics Dashboard custom widget

`slmetric.dashboard.CustomWidget` object

Specify the `slmetric.dashboard.CustomWidget` object for which you get its position in the array.

Output Arguments

`Num` — Position of custom widget object

`double`

Position of `slmetric.dashboard.CustomWidget` object within an array that holds the Metrics Dashboard objects in an `slmetric.dashboard.Layout`, an `slmetric.dashboard.Group`, or an `slmetric.dashboard.Container` object.

Data Types: `double`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setPosition

Class: `slmetric.dashboard.CustomWidget`
Package: `slmetric.dashboard`

Set custom widget position within Metrics Dashboard

Syntax

```
setPosition(CustomWidget, num)
```

Description

`setPosition(CustomWidget, num)` sets the position of an `slmetric.dashboard.CustomWidget` object in an array that holds Metrics Dashboard objects. This array contains the Metrics Dashboard objects in an `slmetric.dashboard.Layout`, an `slmetric.dashboard.Container`, or an `slmetric.dashboard.Group` object. The order of objects in the array corresponds to proceeding from left to right, and then down in the Metrics Dashboard.

Input Arguments

`CustomWidget` — Metrics Dashboard custom widget

`slmetric.dashboard.CustomWidget` object

Specify the `slmetric.dashboard.CustomWidget` object for which you set its position in the array.

Output Arguments

`Num` — Position of widget object

`double`

Position of `slmetric.dashboard.CustomWidget` object within an array that holds the Metrics Dashboard objects in either an `slmetric.dashboard.Layout`, an `slmetric.dashboard.Container`, or an `slmetric.dashboard.Group` object.

Data Types: `double`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”

“Customize Metrics Dashboard Layout and Functionality”

setHeight

Class: `slmetric.dashboard.CustomWidget`
Package: `slmetric.dashboard`

Specify height of Metrics Dashboard custom widget

Syntax

```
setHeight(CustomWidget,num)
```

Description

`setHeight(CustomWidget,num)` specifies the height of a custom widget in pixels.

Input Arguments

`CustomWidget` — Metrics Dashboard customwidget

`slmetric.dashboard.CustomWidget`

`slmetric.dashboard.CustomWidget` for which you want to specify its height.

`num` — Height in pixels

integer

Height of `slmetric.dashboard.CustomWidget` object in pixels. These are the minimum heights that you can set.

- For the `SingleValue` custom widget, the minimum height is 25 pixels.
- For the `BarChart` custom widget, the minimum height is 150 pixels.
- For the `RadialGauge` custom widget, the minimum height is 120 pixels.
- For the `DistributionHeatMap` custom widget, the minimum height is 90 pixels.

Example: `setHeight(widget, 50)`

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

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

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

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

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

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

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



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

        isNonVirtual = true(size(blocks));

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

            if any(strcmp(this.VirtualBlockTypes, blockType))
                isNonVirtual(n) = false;
            else
                switch blockType
                    case 'SubSystem'
```

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

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

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

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

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

        blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For your model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getMargin

Class: `slmetric.dashboard.Group`
Package: `slmetric.dashboard`

Obtain distance from group edge to contents

Syntax

```
pixels = getMargin(Group)
```

Description

`pixels = getMargin(Group)` returns how far in pixels the edges of an `slmetric.dashboard.Group` object is from the widgets that it contains.

Input Arguments

`Group` — Metrics Dashboard group

`slmetric.dashboard.Group` object

The `slmetric.dashboard.Group` object for which you are obtaining the margin distance.

Output Arguments

`pixels` — Group margins

`character vector` | `string scalar`

Margin distance from group contents in pixels.

Example: `'40 px'`

Data Types: `char`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getPosition

Class: `slmetric.dashboard.Group`
Package: `slmetric.dashboard`

Obtain group position within Metrics Dashboard

Syntax

```
Num = getPosition(Group)
```

Description

`Num = getPosition(Group)` returns the position of an `slmetric.dashboard.Group` object within an array that holds Metrics Dashboard objects. These objects are in an `slmetric.dashboard.Layout` or an `slmetric.dashboard.Container` object. The order of objects in the array corresponds to proceeding from left to right, and then down in the Metrics Dashboard.

Input Arguments

`Group` — Metrics Dashboard group

`slmetric.dashboard.Group` object

Specify the `slmetric.dashboard.Group` object for which you get its position in the array.

Output Arguments

`Num` — Position of group object

`double`

Position of `slmetric.dashboard.Group` object within an array that holds the Metrics Dashboard objects in an `slmetric.dashboard.Layout` or an `slmetric.dashboard.Container` object.

Data Types: `double`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setMargin

Class: `slmetric.dashboard.Group`
Package: `slmetric.dashboard`

Specify distance from group edge to its contents

Syntax

```
pixels = setMargin(Group,px)
```

Description

`pixels = setMargin(Group,px)` specifies how far in pixels the edges of an `slmetric.dashboard.Group` object is from the widgets that it contains.

Input Arguments

`Group` — Metrics Dashboard group

`slmetric.dashboard.Group` object

The `slmetric.dashboard.Group` object for which you are specifying margin size in pixels.

Data Types: `char`

`px` — Group margins

`character vector` | `string scalar`

Margin distance from group contents in pixels.

Example: `'40 px'`

Data Types: `char`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setPosition

Class: `slmetric.dashboard.Group`
Package: `slmetric.dashboard`

Set group position within Metrics Dashboard

Syntax

```
setPosition(Group, num)
```

Description

`setPosition(Group, num)` sets the position of an `slmetric.dashboard.Group` object in an array that holds Metrics Dashboard objects. This array contains the Metrics Dashboard objects in an `slmetric.dashboard.Layout` or an `slmetric.dashboard.Container` object. The order of objects in the array corresponds to proceeding from left to right, and then down in the Metrics Dashboard.

Input Arguments

`Group` — Metrics Dashboard group

`slmetric.dashboard.Group` object

Specify the `slmetric.dashboard.Group` object for which you set its position in the array.

Output Arguments

`Num` — Position of group object

`double`

Position of `slmetric.dashboard.Group` object within an array that holds the Metrics Dashboard objects in either an `slmetric.dashboard.Layout` or an `slmetric.dashboard.Container` object.

Data Types: `double`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getHeight

Class: `slmetric.dashboard.Widget`
Package: `slmetric.dashboard`

Obtain height of Metrics Dashboard widget

Syntax

```
Height = getHeight(widget)
```

Description

`Height = getHeight(widget)` returns the height of a widget in pixels.

Input Arguments

`widget` — Metrics Dashboard widget

`slmetric.dashboard.Widget` object

`slmetric.dashboard.Widget` for which you want to specify its height.

Output Arguments

`Height` — Height in pixels

integer

Height of `slmetric.dashboard.Widget` object in pixels.

Example: `Height = getHeight(widget)`

Data Types: `uint32`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

getPosition

Class: `slmetric.dashboard.Widget`
Package: `slmetric.dashboard`

Obtain widget position within Metrics Dashboard

Syntax

```
Num = getPosition(Widget)
```

Description

`Num = getPosition(Widget)` returns the position of an `slmetric.dashboard.Widget` object within an array that holds Metrics Dashboard objects. These objects are in an `slmetric.dashboard.Layout`, an `slmetric.dashboard.Group`, or an `slmetric.dashboard.Container` object. The order of objects in the array corresponds to proceeding from left to right, and then down in the Metrics Dashboard.

Input Arguments

`Widget` — Metrics Dashboard widget

`slmetric.dashboard.Widget` object

Specify the `slmetric.dashboard.Widget` object for which you get its position in the array.

Output Arguments

`Num` — Position of widget object

`double`

Position of `slmetric.dashboard.Widget` object within an array that holds the Metrics Dashboard objects in an `slmetric.dashboard.Layout`, an `slmetric.dashboard.Group`, or an `slmetric.dashboard.Container` object.

Data Types: `double`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setHeight

Class: `slmetric.dashboard.Widget`
Package: `slmetric.dashboard`

Specify height of Metrics Dashboard widget

Syntax

```
setHeight(widget,num)
```

Description

`setHeight(widget,num)` specifies the height of a widget in pixels.

Input Arguments

`widget` — Metrics Dashboard widget

`slmetric.dashboard.Widget`

`slmetric.dashboard.Widget` for which you want to specify its height.

`num` — Height in pixels

integer

Height of `slmetric.dashboard.Widget` object in pixels. These are the minimum heights that you can set.

- For the `SystemInfo` widget, the minimum height is 90 pixels.
- For the `LibraryReuse` widget, the minimum height is 110 pixels.
- For the `GlocalInterface` widget, the minimum height is 60 pixels.

Example: `setHeight(widget, 70)`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

setPosition

Class: `slmetric.dashboard.Widget`
Package: `slmetric.dashboard`

Set widget position within Metrics Dashboard

Syntax

```
setPosition(Widget, num)
```

Description

`setPosition(Widget, num)` sets the position of an `slmetric.dashboard.Widget` object in an array that holds Metrics Dashboard objects. This array contains the Metrics Dashboard objects in an `slmetric.dashboard.Layout`, an `slmetric.dashboard.Container`, or an `slmetric.dashboard.Group` object. The order of objects in the array corresponds to proceeding from left to right, and then down in the Metrics Dashboard.

Input Arguments

`Widget` — Metrics Dashboard widget

`slmetric.dashboard.Widget` object

Specify the `slmetric.dashboard.Widget` object for which you set its position in the array.

Output Arguments

`Num` — Position of widget object

`double`

Position of `slmetric.dashboard.Widget` object within an array that holds the Metrics Dashboard objects in either an `slmetric.dashboard.Layout`, an `slmetric.dashboard.Container`, or an `slmetric.dashboard.Group` object.

Data Types: `double`

Version History

Introduced in R2018b

See Also

`slmetric.dashboard.setActiveConfiguration` | `slmetric.dashboard.getActiveConfiguration`

Topics

“Collect Model Metric Data by Using the Metrics Dashboard”
“Customize Metrics Dashboard Layout and Functionality”

slmetric.metric.MetaInformation class

Package: `slmetric.metric`

Set metadata for custom metrics

Description

The `slmetric.metric.MetaInformation` class properties contain metric metadata. On the Metrics Dashboard, when you click the widget for an individual metric, this metadata is in the table. For custom metrics, when you create a custom metric class, you specify the `slmetric.metric.MetaInformation` the applicable properties.

Construction

Create an `slmetric.Engine` object. Use the `getMetricMetaInformation` property to return an `slmetric.metric.MetaInformation` object.

Properties

`Name` — Metric name

character value | string scalar

For custom metrics, when you define the custom metric class, specify this property. For shipped metrics, this property is already set.

Example: 'Model Advisor standards check compliance for High Integrity'

Data Types: `char`

`Description` — Metric description

character value | string scalar

For custom metrics, when you define the custom metric class, specify this property. For shipped metrics, this property is already set.

Example: 'Metric that counts the percentage of checks that passed for the High Integrity Model Advisor standards check grouping.'

Data Types: `char`

`MeasuresNames` — Names of metric measures

cell array of character vectors | cell array of string scalars

For custom metrics, when you define the custom metric class, if applicable, specify this property. For shipped metrics, this property is already set.

Example: {'Passed Checks'} {'Total Checks'}

Data Types: `char`

`AggregatedMeasuresNames` — Names of aggregated metric measures

cell array of character vectors | cell array of string scalars

For custom metrics, when you define the custom metric class, if applicable, specify this property. For shipped metrics, this property is already set.

Example: {'Passed Checks (incl. Descendants'} {'Total Checks'}

Data Types: `char`

`ValueName` — Value name
character vector | string scalar

For custom metrics, when you define the custom metric class, specify this property. For shipped metrics, this property is already set.

Example: 'Passed Checks'

Data Types: `char`

`AggregatedValueName` — Name of aggregated metric value
cell array of character vectors | cell array of string scalars

For custom metrics, when you define the custom metric class, specify this property. For shipped metrics, this property is already set.

Example: {'Passed Checks (incl. Descendants'} {'Total Checks'}

Data Types: `char`

Examples

Add a Custom Widget to a Group

Create a custom metric that counts nonvirtual blocks. Specify a widget to display this metric on the Metrics Dashboard. Add it to the Size Group.

Create a custom metric class.

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

Create the nonvirtual block count metric by adding this code to the `nonvirtualblockcount.m` file. The `this = nonvirtualblockcount` function sets the `slmetric.metric.MetaInformation` properties.

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

    methods
    function this = nonvirtualblockcount()
        this.ID = 'nonvirtualblockcount';
        this.Name = 'Nonvirtual Block Count';
        this.Version = 1;
```

```
        this.CompileContext = 'None';
        this.Description = 'Algorithm that counts nonvirtual blocks per level.';
        this.AggregatedValueName = 'Nonvirtual Blocks (incl. Descendants)'
        this.ValueName = 'Nonvirtual Blocks'
        this.ComponentScope = [Advisor.component.Types.Model, ...
            Advisor.component.Types.SubSystem];
        this.AggregationMode = slmetric.AggregationMode.Sum;
        this.ResultChecksumCoverage = true;
        this.SupportsResultDetails = true;

    end

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

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

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



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

        isNonVirtual = true(size(blocks));

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

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

```
                    case 'Selector'
                        % Virtual only when Number of input dimensions
                        % specifies 1 and Index Option specifies Select
                        % all, Index vector (dialog), or Starting index (dialog).
                        nod = get_param(blocks{n}, 'NumberOfDimensions');
                        ios = get_param(blocks{n}, 'IndexOptionArray');

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

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

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

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

        blocks = blocks(isNonVirtual);

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

Register the new metric in the metric repository.

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

To begin, open the default configuration for the Metrics Dashboard layout.

```
conf = slmetric.dashboard.Configuration.open();
```

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

```
layout = getDashboardLayout(conf);
```

Obtain widget objects that are in the layout object.

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

Remove the widget that represents the Simulink block count metric.

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

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 technique, 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);
```

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]);
```

Save the configuration object. This command serializes the API information to an XML file.

```
save(conf,'Filename','DashboardConfig.xml');
```

Set the active configuration.

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

For a model, open the Metrics Dashboard.

```
metricsdashboard vdp
```

Click the **All Metrics** button to run all metrics.

Version History

Introduced in R2018b

See Also

getMetricMetaInformation

Class: `slmetric.Engine`
Package: `slmetric`

Obtain metric metadata

Syntax

```
metaInfo = getMetricMetaInformation(metric_engine,metricID)
```

Description

`metaInfo = getMetricMetaInformation(metric_engine,metricID)` returns the `slmetric.metric.MetaInformation` object corresponding to the `metricID`.

Input Arguments

`metric_engine` — Metric engine object

`slmetric.Engine` object

Create an `slmetric.Engine` object.

```
metric_engine = slmetric.Engine();
```

Data Types: `char`

`MetricID` — Metric identifier

character vector | string scalar

Metric identifier for shipped or custom metrics. You can get metric identifiers by calling `slmetric.metric.getAvailableMetrics`.

Data Types: `char`

Output Arguments

`metaInfo` — Meta information object

`slmetric.metric.MetaInformation` object

For a `metricID`, the `slmetric.metric.MetaInformation` object contains its metadata. On the Metrics Dashboard, when you click a widget, this metadata appears on the table.

Examples

Obtain Metric Metadata

Obtain metadata for the high-integrity check compliance metric. This metric has a **metric ID** of `mathworks.metrics.ModelAdvisorCheckCompliance.hisl_do178`.

Create an `slmetric.Engine` object.

```
metric_engine = slmetric.Engine();
```

To obtain metadata, use the `getMetricMetaInformation` method.

```
metaInfo = getMetricMetaInformation(metric_engine,...
'mathworks.metrics.ModelAdvisorCheckCompliance.hisl_do178')
```

The high-integrity check compliance metric contains this metadata:

```
metaInfo =

  MetaInformation with properties:

                      Name: 'Model Advisor standards check compliance for High Integrity'
               Description: 'Metric that counts the percentage of checks that passed for the High Integrity Model Adviso
             MeasuresNames: {2×1 cell}
   AggregatedMeasuresNames: {2×1 cell}
                 ValueName: 'Checks Passed'
       AggregatedValueName: 'Checks Passed (incl. Descendants)'
```

Version History

Introduced in R2018b

See Also

ModelAdvisor.ResultDetail class

Package: `ModelAdvisor`

Define check result details

Description

Use objects of the `ModelAdvisor.ResultDetail` class in your custom check authoring algorithm to provide details about check results. For checks that run only in the Model Advisor, use the `setResultDetails` method to associate these results with the `ModelAdvisor.Check` object and specify '`DetailStyle`' as the callback style in the `ModelAdvisor.Check.setCallbackFcn` function.

Properties

`Data` — Block identifier or signal line handle

string

Simulink identifier (SID) for each block or signal handle for each signal that violates your custom check, specified as a string.

Data Types: `char`

Type — Data type

`SID` (default) | `Signal`

Data type, specified as the `ModelAdvisor.ResultDetailType` enumeration type that indicates the location of the check violation:

- `SID` – Check violation is on a block
- `Signal` – Check violation is on a signal

Data Types: `enum`

`IsInformer` — Whether check results do not have violations

`false` (default) | `true`

Whether check results do not have violations, specified as `true` or `false`. If `IsInformer = true`, then `IsViolation = false`.

Data Types: `logical`

`IsViolation` — Identifies check results with violations

`true` (default) | `false`

Whether check results have violations, specified as `true` or `false`. If `IsViolation = true`, then `IsInformer = false`.

Data Types: `logical`

`Description` — Description of check results

`string`

Description of check results, specified as a string.

Data Types: `char`

`Title` — Title
`string`

Title of check results, specified as a string.

Data Types: `char`

`Information` — Additional information
`string`

Additional information about the check results, specified as a string.

Data Types: `char`

`Status` — Status message
`string`

Status message that appears in the Model Advisor, specified as a string.

Data Types: `char`

`RecAction` — Recommended action
`string`

Recommended action for fixing the check, specified as a string.

Data Types: `char`

`ViolationType` — Severity of check results
`"warn"` (default) | `"pass"` | `"fail"`

Severity of check results specified as `"pass"`, `"fail"`, or `"warn"`.

Data Types: `char`

Methods

setData	Associate block identifier or signal handle with `ModelAdvisor.ResultDetail` object

Examples

Define Custom Model Advisor Check for Block Violations

Create a custom Model Advisor check that checks whether block names appear below blocks.

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, `defineDetailStyleCheck` is the check definition function. Create the `sl_customization` function and save it to your working folder.

```
function sl_customization(cm)
cm.addModelAdvisorCheckFcn(@defineDetailStyleCheck);
```

Create the check definition function. For this example, create the `defineDetailStyleCheck` function and copy the code below into it. Save the function to your working folder. The check definition function contains `ModelAdvisor.Check` and `ModelAdvisor.Action` objects that define the check actions and a fix. For more details on these aspects of the code, see “Fix a Model to Comply with Conditions that You Specify with the Model Advisor”.

The `defineDetailStyleCheck` function contains a `DetailStyleCallback` callback function. To return blocks whose name does not appear below the block, `violationBlks`, the `DetailStyleCallback` function uses the `find_system` function.

When `violationBlks` is empty, the code creates one `ModelAdvisor.ResultDetail` object, `ElementResults`. `ElementResults` specifies information about the passing check that appears in the Model Advisor.

When the `find_system` function returns a list of blocks that violate the check, `ElementResults` is an array of `ModelAdvisor.Results` objects. The array contains one object for each block that violates the check. Each object contains information about the violation block that appears in the Model Advisor.

```
function defineDetailStyleCheck

% 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);

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

end

% -----------------------------
% This callback function uses the DetailStyle CallbackStyle type.
% -----------------------------
function DetailStyleCallback(system, CheckObj)
% get Model Advisor object
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);

% 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.';
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.';
```

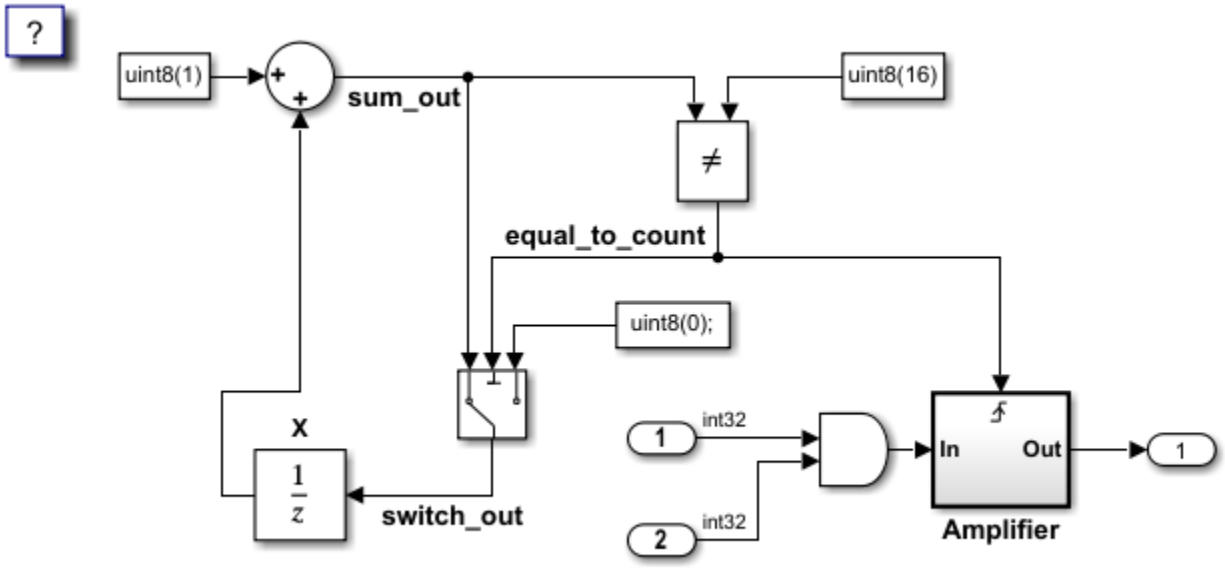
```
        ElementResults(i).Status = 'The following blocks have names that do not display below the blocks:';
        ElementResults(i).RecAction =  'Change the location such that the block name is below the block.';
        ElementResults(i).ViolationType = 'warn';
    end
    mdladvObj.setActionEnable(true);
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
```

Refresh the Model Advisor to update the cache with the new check on the path.

```
Advisor.Manager.refresh_customizations
```

To use the check, copy the `AdvisorCustomizationExample.slx` model to your current working folder. Open the model.

```
copyfile(fullfile(matlabroot,'examples','slcheck','main',...
'AdvisorCustomizationExample.slx'),'AdvisorCustomizationExample.slx','f');
```



Open the Model Advisor by clicking the **Modeling** tab and selecting **Model Advisor**.

In the left pane, select **By Product > Demo > Check whether block names appear below** and click **Run Checks**.

To address the warning, click **Fix**.

Define Custom Edit-Time Check for Signal Violations

Create a custom edit-time check that checks whether signals that connect to Outport blocks have labels.

To register the custom edit-time check, create an `sl_customization` function and save it to your working folder.

```
function sl_customization(cm)
cm.addModelAdvisorCheckFcn(@defineCheck);
```

Create the check definition function. Inside the function, create a `ModelAdvisor.Check` object and specify the check ID as the 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 `SignalLabel` is the class names. Then, publish the check to a new folder in the Model Advisor. For this example, the folder name is **DEMO: Edit Time Checks**. For this example, create a `defineCheck` function and include the code below in it. Save the `defineCheck` function to your working folder.

```
function defineCheck
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 = ModelAdvisor.Root;
mdladvRoot.publish(rec,'DEMO: Edit Time Checks');
```

Create a class that derives from the `ModelAdvisor.EdittimeCheck` abstract base class. For this example, create a class file named `SignalLabel.m`. Copy the code below into the `SignalLabel.m` file and save it to the `+MyEditTimeChecks` folder. For more information on the code in the `ModelAdvisor.EdittimeCheck` class, see “Define Edit-Time Checks to Comply with Conditions That You Specify with the Model Advisor”.

The `blockDiscovered` method contains an algorithm that defines check violations. Notice that, unlike custom checks that appear only in the Model Advisor, this algorithm does not contain the `find_system` function. The `blockDiscovered` method takes block handles as inputs and traverses the model, so you do not need the `find_system` function for custom edit-time checks.

For each model element that violates the check, the code creates a `ModelAdvisor.ResultDetail` object. For this example, because the violations are on signals, the check must use parameters on the line handles of blocks, `LineHandles` to find signals with violations. Specifically, for signals that connect to Outport blocks, this algorithm checks whether the `Name` signal parameter has a value. Then, because the violation is on a signal, the algorithm highlights the signal by creating a violation object with the `Type` property value set to `Signal`.

```
classdef SignalLabels < ModelAdvisor.EdittimeCheck
    methods
        function obj=SignalLabels(checkId)
            obj=obj@ModelAdvisor.EdittimeCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.BLKITER;
        end

        function violation = blockDiscovered(obj, blk)
            violation = [];
            ports     = get_param(blk,'Ports');
```

```
            lh = get_param(blk, 'LineHandles');
            if strcmp(get_param(blk,'BlockType'),'Outport')
                for j = 1 : ports(1)
                    if lh.Inport(j) ~= -1 % failure case: no connection
                        allsources = get_param(lh.Inport(j),'SrcPortHandle');
                        hiliteHandle = get_param(lh.Inport(j), 'DstPortHandle');
                        if (isempty(allsources) ~= 0) || (isempty(find(allsources==-1,1)) ~= 0)
                            lh_obj = get_param(lh.Inport(j),'Object');
                            if isempty(lh_obj.Name)
                                if strcmp(lh_obj.signalPropagation,'off') == 1
                                   allsources_parent = get_param(allsources,'Parent');
                                   if strcmp(get_param(allsources_parent,'BlockType'),'Inport')
                                        buscreator_outputs = get_param(allsources_parent,'IsBusElementPort');
                                    else
                                        buscreator_outputs = 'off';
                                    end
                                    if ~strcmp(buscreator_outputs,'on')
                                        violation = ModelAdvisor.ResultDetail;
                                        ModelAdvisor.ResultDetail.setData(violation, 'Signal',hiliteHandle);
                                        violation.Description ='This signal should have a label.';
                                        violation.CheckID = obj.checkId;
                                        violation.Title = 'Signal Label Missing';
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end
```

To use the checks, copy the `AdvisorCustomizationExample.slx` model to your current working folder and open the model.

```
copyfile(fullfile(matlabroot,'examples','slcheck','main',...
'AdvisorCustomizationExample.slx'),'AdvisorCustomizationExample.slx','f');
```

Refresh the Model Advisor to update the cache with the new checks on the path.

```
Advisor.Manager.refresh_customizations
```

Open the Model Advisor Configuration Editor by clicking the **Modeling** tab and selecting **Model Advisor > Configuration Editor**.

Create a custom configuration that consists of the custom edit-time check by deleting every folder except the **DEMO: Edit Time Checks** folder.

Save the configuration as `my_config.json`. When prompted to set this configuration as the default, click **No**.

Close the Model Advisor Configuration Editor.

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.

Turn on edit-time checking by selecting the **Edit-Time Checks** parameter. Close the Model Configuration Parameters dialog box.

To view the edit-time warning, click the signal highlighted in yellow. The signal connecting to the Outport block produces a warning because it does not have a label.

Version History

Introduced in R2018b

See Also

`ModelAdvisor.Check` | `ModelAdvisor.EdittimeCheck`

Topics

“Define Custom Model Advisor Checks”
“Fix a Model to Comply with Conditions that You Specify with the Model Advisor”
“Define Edit-Time Checks to Comply with Conditions That You Specify with the Model Advisor”

ModelAdvisor.ResultDetail.setData

Class: `ModelAdvisor.ResultDetail`
Package: `ModelAdvisor`

Associate block identifier or signal handle with `ModelAdvisor.ResultDetail` object

Syntax

```
ModelAdvisor.ResultDetail.setData(ElementResults,violation)
```

Description

`ModelAdvisor.ResultDetail.setData(ElementResults,violation)` associates the Simulink Identifiers (SID) of blocks or the signal handles of signals that violate a check, `violations` with the `ModelAdvisor.ResultDetail` objects, `ElementResults`. Use this method as part of your custom check to return information about blocks or signals that violate the check.

Input Arguments

`ElementResults` — Result details

array of `ModelAdvisor.ResultDetail` objects

Result details, specified as an array of `ModelAdvisor.ResultDetail` objects that correspond to each block or signal that violates a custom Model Advisor check.

`violation` — Simulink identifier or signal line handle

string | array of strings

Simulink identifier (SID) or signal line handle for blocks or signals that violate your custom check, specified as a string or an array of strings.

Examples

Define Custom Model Advisor Check for Block Violations

Create a custom Model Advisor check that checks whether block names appear below blocks.

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, `defineDetailStyleCheck` is the check definition function. Create the `sl_customization` function and save it to your working folder.

```
function sl_customization(cm)
cm.addModelAdvisorCheckFcn(@defineDetailStyleCheck);
```

Create the check definition function. For this example, create the `defineDetailStyleCheck` function and copy the code below into it. Save the function to your working folder. The check

definition function contains `ModelAdvisor.Check` and `ModelAdvisor.Action` objects that define the check actions and a fix. For more details on these aspects of the code, see “Fix a Model to Comply with Conditions that You Specify with the Model Advisor”.

The `defineDetailStyleCheck` function contains a `DetailStyleCallback` callback function. To return blocks whose name does not appear below the block, `violationBlks`, the `DetailStyleCallback` function uses the `find_system` function.

When `violationBlks` is empty, the code creates one `ModelAdvisor.ResultDetail` object, `ElementResults`. `ElementResults` specifies information about the passing check that appears in the Model Advisor.

When the `find_system` function returns a list of blocks that violate the check, `ElementResults` is an array of `ModelAdvisor.Results` objects. The array contains one object for each block that violates the check. Each object contains information about the violation block that appears in the Model Advisor.

```
function defineDetailStyleCheck

% 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);

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

end

% -----------------------------
% This callback function uses the DetailStyle CallbackStyle type.
% -----------------------------
function DetailStyleCallback(system, CheckObj)
% get Model Advisor object
mdladvObj = Simulink.ModelAdvisor.getModelAdvisor(system);

% 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.';
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.';
        ElementResults(i).ViolationType = 'warn';
    end
    mdladvObj.setActionEnable(true);
end
```

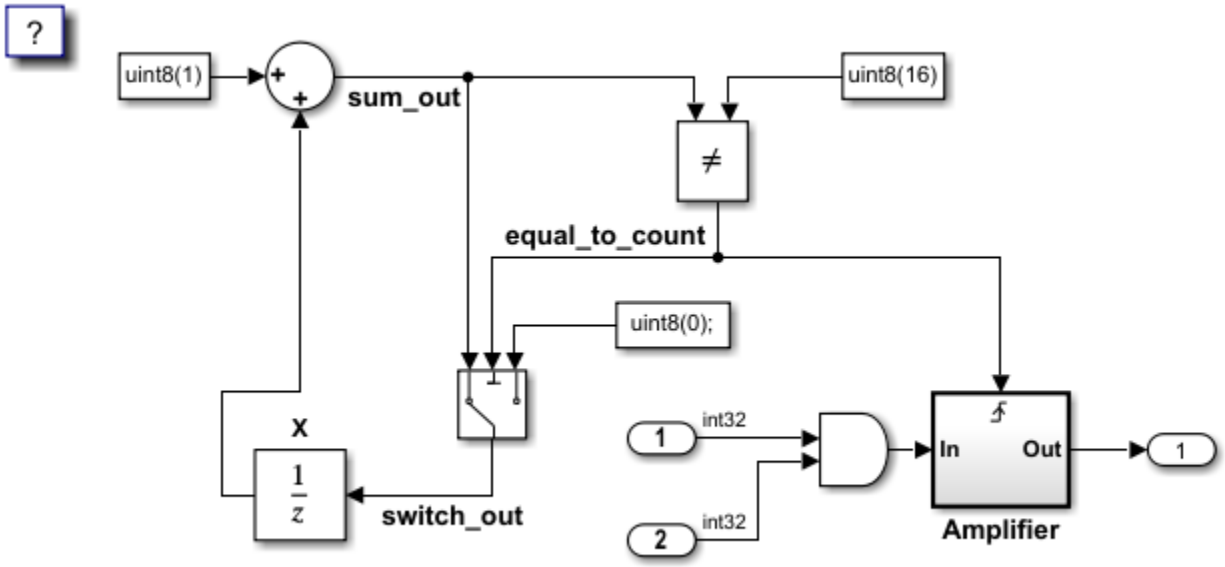
```
CheckObj.setResultDetails(ElementResults);
end

% -----------------------------
% 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
```

Refresh the Model Advisor to update the cache with the new check on the path.

```
Advisor.Manager.refresh_customizations
```

To use the check, copy the `AdvisorCustomizationExample.slx` model to your current working folder. Open the model.

```
copyfile(fullfile(matlabroot,'examples','slcheck','main',...
'AdvisorCustomizationExample.slx'),'AdvisorCustomizationExample.slx','f');
```



Open the Model Advisor by clicking the **Modeling** tab and selecting **Model Advisor**.

In the left pane, select **By Product > Demo > Check whether block names appear below** and click **Run Checks**.

To address the warning, click **Fix**.

Define Custom Edit-Time Check for Signal Violations

Create a custom edit-time check that checks whether signals that connect to Outport blocks have labels.

To register the custom edit-time check, create an `sl_customization` function and save it to your working folder.

```
function sl_customization(cm)
cm.addModelAdvisorCheckFcn(@defineCheck);
```

Create the check definition function. Inside the function, create a `ModelAdvisor.Check` object and specify the check ID as the 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 `SignalLabel` is the class names. Then, publish the check to a new folder in the Model Advisor. For this example, the folder name is **DEMO: Edit Time Checks**. For this example, create a `defineCheck` function and include the code below in it. Save the `defineCheck` function to your working folder.

```
function defineCheck
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 = ModelAdvisor.Root;
mdladvRoot.publish(rec,'DEMO: Edit Time Checks');
```

Create a class that derives from the `ModelAdvisor.EdittimeCheck` abstract base class. For this example, create a class file named `SignalLabel.m`. Copy the code below into the `SignalLabel.m` file and save it to the `+MyEditTimeChecks` folder. For more information on the code in the `ModelAdvisor.EdittimeCheck` class, see “Define Edit-Time Checks to Comply with Conditions That You Specify with the Model Advisor”.

The `blockDiscovered` method contains an algorithm that defines check violations. Notice that, unlike custom checks that appear only in the Model Advisor, this algorithm does not contain the `find_system` function. The `blockDiscovered` method takes block handles as inputs and traverses the model, so you do not need the `find_system` function for custom edit-time checks.

For each model element that violates the check, the code creates a `ModelAdvisor.ResultDetail` object. For this example, because the violations are on signals, the check must use parameters on the line handles of blocks, `LineHandles` to find signals with violations. Specifically, for signals that connect to Outport blocks, this algorithm checks whether the `Name` signal parameter has a value. Then, because the violation is on a signal, the algorithm highlights the signal by creating a violation object with the `Type` property value set to `Signal`.

```
classdef SignalLabels < ModelAdvisor.EdittimeCheck
    methods
        function obj=SignalLabels(checkId)
            obj=obj@ModelAdvisor.EdittimeCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.BLKITER;
        end

        function violation = blockDiscovered(obj, blk)
            violation = [];
            ports    = get_param(blk,'Ports');
            lh = get_param(blk, 'LineHandles');
            if strcmp(get_param(blk,'BlockType'),'Outport')
                for j = 1 : ports(1)
                    if lh.Inport(j) ~= -1 % failure case: no connection
                        allsources = get_param(lh.Inport(j),'SrcPortHandle');
```

```
                        hiliteHandle = get_param(lh.Inport(j), 'DstPortHandle');
                        if (isempty(allsources) ~= 0) || (isempty(find(allsources==-1,1)) ~= 0)
                            lh_obj = get_param(lh.Inport(j),'Object');
                            if isempty(lh_obj.Name)
                                if strcmp(lh_obj.signalPropagation,'off') == 1
                                   allsources_parent = get_param(allsources,'Parent');
                                   if strcmp(get_param(allsources_parent,'BlockType'),'Inport')
                                        buscreator_outputs = get_param(allsources_parent,'IsBusElementPort');
                                    else
                                        buscreator_outputs = 'off';
                                    end
                                    if ~strcmp(buscreator_outputs,'on')
                                        violation = ModelAdvisor.ResultDetail;
                                        ModelAdvisor.ResultDetail.setData(violation, 'Signal',hiliteHandle);
                                        violation.Description ='This signal should have a label.';
                                        violation.CheckID = obj.checkId;
                                        violation.Title = 'Signal Label Missing';
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end
```

To use the checks, copy the `AdvisorCustomizationExample.slx` model to your current working folder and open the model.

```
copyfile(fullfile(matlabroot,'examples','slcheck','main',...
'AdvisorCustomizationExample.slx'),'AdvisorCustomizationExample.slx','f');
```

Refresh the Model Advisor to update the cache with the new checks on the path.

```
Advisor.Manager.refresh_customizations
```

Open the Model Advisor Configuration Editor by clicking the **Modeling** tab and selecting **Model Advisor > Configuration Editor**.

Create a custom configuration that consists of the custom edit-time check by deleting every folder except the **DEMO: Edit Time Checks** folder.

Save the configuration as `my_config.json`. When prompted to set this configuration as the default, click **No**.

Close the Model Advisor Configuration Editor.

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.

Turn on edit-time checking by selecting the **Edit-Time Checks** parameter. Close the Model Configuration Parameters dialog box.

To view the edit-time warning, click the signal highlighted in yellow. The signal connecting to the Outport block produces a warning because it does not have a label.

Version History

Introduced in R2018b

See Also

`ModelAdvisor.ResultDetail` | `ModelAdvisor.Check`

Topics

“Create and Deploy a Model Advisor Custom Configuration”

setResultDetails

Associates result details with a check object

Syntax

`setResultDetails(ElementResults)`

Description

In the check callback function, use `setResultDetails(ElementResults)` to associate `ElementResults` with the check (`CheckObj`).

`ElementResults` is a collection of instances of the `ModelAdvisor.ResultDetail` class.

Input Arguments

`ElementResults`	Collection of `ResultDetailObjs` objects

Examples

This example shows the result details that correspond to the execution of check **Check whether block names appear below blocks** in the `AdvisorCustomizationExample` model. At the end of the code, `CheckObj.setResultDetails(ElementResults);` associates the results with the check object. For more information, see “Create and Deploy a Model Advisor Custom Configuration”.

```
% ----------------------------
% 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
```

Version History

Introduced in R2018b

See Also

`ModelAdvisor.ResultDetail` | ModelAdvisor.Check.ResultDetails

Topics

“Create Model Advisor Checks”
“Fix a Model to Comply with Conditions that You Specify with the Model Advisor”

ResultDetails property

Result details in a cell array

Values

Cell array

Default: `{}` (empty cell array)

Description

The `ResultDetails` property stores the `ResultDetailObjs` objects associated with the check. This property can contain multiple objects.

Version History

Introduced in R2018b

slslicer

Create API object for invoking Model Slicer

Syntax

```
slslicer(model)
slslicer(model,opts)
obj = slslicer(model)
```

Description

`slslicer(model)` creates a Model Slicer configuration object for the model `model` by exposing the methods for invoking Model Slicer.

`slslicer(model,opts)` creates a Model Slicer configuration object for the model `model` by using the options object `opts`, as defined by `slsliceroptions`.

`obj = slslicer(model)` creates a Model Slicer configuration object. You can apply the methods on the Model Slicer object `obj`.

Examples

Add Starting Point and Highlight the Model Slice

Add a new starting point to the active Model Slicer configuration, and then highlight the model.

Open the `sldvSliceClimateControlExample` example model.

```
addpath(fullfile(docroot,'toolbox','simulink','examples'));
open_system('sldvSliceClimateControlExample');
```

Create a Model Slicer configuration object for the model by using `slslicer`.

```
obj = slslicer('sldvSliceClimateControlExample');
```

Activate the slice highlighting mode of Model Slicer to compile the model and prepare the model slice for dependency analysis.

```
activate(obj);
```

Add the `Out1` outport block as the starting point and highlight the model slice.

```
addStartingPoint(obj,'sldvSliceClimateControlExample/Out1');
highlight(obj);
```

The area of the model upstream of the starting point and which is active during simulation is highlighted.

Terminate the model highlighting mode and discard the analysis data.

```
terminate(obj);
```

Input Arguments

`model` — Name or handle of model

character vector | string scalar

Name of the model whose Model Slicer options object you configure. `slslicer` uses the Model Slicer configurations associated with `model`, as defined by `slsliceroptions`.

`opts` — Options you attach to a model or save to a file

slsliceroptions object

Structure containing the options for the Model Slicer configuration. `slsliceroptions` defines the options object `opts`.

Output Arguments

`obj` — Model Slicer object

slslicer object

Model Slicer object handle.

Version History

Introduced in R2015b

See Also

`SLSlicerAPI.SLSlicer` | `slslicertrace` | `slsliceroptions`

Topics

“Workflow for Dependency Analysis”
“Configure Model Highlight and Sliced Models”
“Model Slicer Considerations and Limitations”

slsliceroptions

Create options object for configuring Model Slicer

Syntax

```
slsliceroptions
slsliceroptions(model)
slsliceroptions(file)
slsliceroptions(model,opts)
slsliceroptions(file,opts)
```

Description

`slsliceroptions` creates an options object for configuring the Model Slicer.

`slsliceroptions(model)` creates a copy of the Model Slicer options object associated with `model`.

`slsliceroptions(file)` creates a copy of the Model Slicer options object contained in the SLMS file `file`.

`slsliceroptions(model,opts)` attaches the slicer options `opts` to the model `model` by overwriting the existing options.

`slsliceroptions(file,opts)` attaches the slicer options `opts` to the SLMS file `file` by overwriting the existing options.

Examples

Add Starting Points and Exclusion Points to Active Configuration

Add a new starting point and a new exclusion point to the active Model Slicer configuration.

Open the `f14` example model.

```
open_system('f14')
```

Define the options file `opts` for the model.

```
opts = slsliceroptions('f14')
```

Add a new starting point on the `Gain` block.

```
addStartingPoint(opts,'f14/Gain')
```

Add a new exclusion point on the `alpha (rad)` block.

```
addExclusionPoint(opts,'f14/alpha (rad)')
```

Add Starting Points and Exclusion Points to New Configuration

Add a starting point and an exclusion point to the active Model Slicer configuration without overwriting the original configuration.

Open the `f14` example model.

```
open_system('f14')
```

Define the options file `opts` for the model.

```
opts = slsliceroptions('f14')
```

Create a second Model Slicer options configuration for the model.

```
addConfiguration(opts)
```

Add a new starting point on the `Gain` block for the second Model Slicer options configuration.

```
addStartingPoint(opts.Configuration(2),'f14/Gain')
```

Add a new exclusion point on the `alpha (rad)` block for the second Model Slicer options configuration.

```
addExclusionPoint(opts.Configuration(2),'f14/alpha (rad)')
```

Input Arguments

`model` — Name or handle of model

character vector | string scalar

Name of the model whose Model Slicer options object you configure.

`file` — Name of file

character vector | string scalar

Name of the SLMS file containing the Model Slicer options object that you configure.

Example: `slsliceroptions('f14.slms')`

`opts` — Options you attach to a model or save to a file

structure

Structure containing the options for the Model Slicer configuration.

Version History

Introduced in R2015b

See Also

`slslicertrace` | `slsliceroptions`

Topics

“Workflow for Dependency Analysis”
“Configure Model Highlight and Sliced Models”

“Model Slicer Considerations and Limitations”

slslicertrace

Return block handles in sliced model or source model after using Model Slicer

Syntax

```
slslicertrace('slice',object)
slslicertrace('source',object)
```

Description

`slslicertrace('slice',object)` returns the block handles in the sliced model that correspond to blocks specified by `object` in the source model.

`slslicertrace('source',object)` returns the block handles in the source model that correspond to blocks specified by `object` in the sliced model.

Examples

Highlight a Block in the Source Model

Highlight the Switch block in the `sldvSliceClimateControlExample` source model.

Open the `sldvSliceClimateControlExample` example model.

```
addpath(fullfile(docroot,'toolbox','simulink','examples'));
open_system('sldvSliceClimateControlExample');
```

Create a slicer object `obj` and add `Out1` as the starting point.

```
obj = slslicer('sldvSliceClimateControlExample');
activate(obj);
addStartingPoint(obj,'sldvSliceClimateControlExample/Out1');
highlight(obj)
```

Create a sliced model by using `slice`.

```
slice(obj,'sldvSliceClimateControlExample_sliced')
```

Highlight the `On` Switch block in the source model by using `slslicertrace`

```
h=slslicertrace('SOURCE', 'sldvSliceClimateControlExample_sliced/Refrigeration/On');
hilite_system(h);
terminate(obj);
```

Input Arguments

`object` — Object in source model or sliced model

character vector | cell array of character vectors | string array

An `object` can be specified as an array of block handles, cell arrays of block paths, or cell arrays of Simulink Identifiers (SID).

Version History

Introduced in R2015b

See Also

`slslicer` | `slsliceroptions`

Topics

“Workflow for Dependency Analysis”
“Configure Model Highlight and Sliced Models”
“Model Slicer Considerations and Limitations”

edittime.enableCheck

Enable disabled custom edit-time check

Syntax

```
edittime.enableCheck(checkID)
```

Description

`edittime.enableCheck(checkID)` enables the disabled custom edit-time check `checkID`. The Model Advisor disables custom edit-time checks that take longer than `500` milliseconds to run in at least three different models.

Examples

Re-enable a Disabled Custom Edit-Time Check

Re-enable a check that took too long to execute.

Suppose you have a custom edit-time check with the check identifier `advisor.edittimecheck.PortColor` that takes longer than `500` milliseconds to execute and the Model Advisor disables the check. When the Model Advisor disables the check, it displays a warning on the Simulink canvas.

To re-enable the check, provide the check identifier as the input to the `edittime.enableCheck` function.

```
edittime.enableCheck("advisor.edittimecheck.PortColor")
```

Input Arguments

`checkID` — Check identifier

string

Check identifier for a custom edit-time check, specified as a string.

Example: `"advisor.edittimecheck.PortColor"`

Data Types: `string`

Version History

Introduced in R2022a

See Also

`edittime.getAdvisorChecking` | `edittime.setAdvisorChecking`

Topics
“Define Edit-Time Checks to Comply with Conditions That You Specify with the Model Advisor”

edittime.getAdvisorChecking

Determine whether edit-time checking is on

Syntax

```
val = edittime.getAdvisorChecking(model)
```

Description

Use the `val = edittime.getAdvisorChecking(model)` returns whether edit-time checking is on or off.

Examples

Determine if edit-time checking is on

Determine your model interactively.

Open the `vdp` example model.

```
vdp
```

Check whether edit-time checking is on.

```
edittime.getAdvisorChecking('vdp')
```

Input Arguments

`model` — model name

character vector

Model for which you want to determine whether edit-time checking is on.

Data Types: `char`

Output Arguments

`val` — Returns on or off

char

If edit-time checking is on, this function returns `on`. If edit-time checking is off, this function returns `off`.

Version History

Introduced in R2019a

See Also

`edittime.setAdvisorChecking`

Topics

“Check Model Compliance by Using the Model Advisor”
“Check Model Compliance Using Edit-Time Checking”

edittime.setAdvisorChecking

Check your model interactively against modeling standards

Syntax

```
edittime.setAdvisorChecking(model,value)
```

Description

`edittime.setAdvisorChecking(model,value)` specifies whether to use the Model Advisor to check your model interactively against modeling standards. This function is the equivalent to selecting **Modeling > Model Advisor > Edit-Time Checks** and selecting the check box for **Edit-Time Checks**.

Examples

Enable edit-time checking through the command line

Check your model interactively.

Open the `vdp` example model.

```
vdp
```

Turn on edit-time checking.

```
edittime.setAdvisorChecking('vdp','on')
```

Confirm that edit-time checking is on.

```
edittime.getAdvisorChecking('vdp')
```

Input Arguments

`model` — model name

character vector

Model that you want to apply edit-time checking to

Data Types: `char`

`value` — Turn edit-time checking on or off

character vector

To turn edit-time checking on, set `value` to `on`. To turn edit-time checking off, set the `value` to `off`.

Data Types: `char`

Version History

Introduced in R2019a

See Also

`edittime.getAdvisorChecking`

Topics

“Check Model Compliance by Using the Model Advisor”
“Check Model Compliance Using Edit-Time Checking”

addExclusionPoint

Class: `SLSlicerAPI.SLSlicer`
Package: `SLSlicerAPI`

Adds block handles, block paths, or Simulink Identifiers (SID) as slice exclusion point

Syntax

```
addExclusionPoint(obj,ExclusionPoint)
```

Description

`addExclusionPoint(obj,ExclusionPoint)` adds the `ExclusionPoint` as the exclusion point in the model slice for dependency analysis.

Input Arguments

`obj` — Model Slicer configuration

slslicer object

Class containing the options of Model Slicer configuration.

`ExclusionPoint` — Slice exclusion point

character vector | cell array of character vectors | string array

Example

Exclusions at Block handles

Specifies the block handle. To get the block handle, use the `getSimulinkBlockHandle` command.

```
obj = slslicer('sldvSliceClimateControlExample');
blkH = get_param('sldvSliceClimateControlExample/Out1','Handle');
addExclusionPoint(obj,blkH);
```

To add multiple block handles as exclusion point, use cell array, for example:

```
blkH = {get_param('sldvSliceClimateControlExample/Out1','Handle'),...
        get_param('sldvSliceClimateControlExample/Out2','Handle')};
addExclusionPoint(obj,blkH);
```

Exclusions at Block Paths

Block path name, specified as a character vector or a cell array of character vectors.

```
bPath = {'sldvSliceClimateControlExample/Out1'};
addExclusionPoint(obj, bPath);
```

To add multiple block handles as exclusion point, use cell array, for example:

```
bPath = {'sldvSliceClimateControlExample/Out1',...
'sldvSliceClimateControlExample/Heater/HeaterAct'};
addExclusionPoint(obj, bPath);
```

Exclusions at SID

Simulink Identifier, a unique designation assigned to a Simulink block or model annotation. To get the SID, use the `Simulink.ID.getSID` command.

```
addExclusionPoint(obj, 'sldvSliceClimateControlExample:39')
```

Exclusions at LineHs

Handles of line that connects from the Inport block to the Outport block. To get the Line Handle, use `get_param` command.

```
 lh1 = get_param('sldvSliceClimateControlExample/Heater/Heat','LineHandles');
 lh2 = get_param('sldvSliceClimateControlExample/Heater','LineHandles');
 LineHs = [lh1.Inport(1), lh2.Outport(2)];
addExclusionPoint(obj,LineHs);
```

Alternatives

To open the Model Slicer manager, in the Simulink Editor, select the **APPS** tab, and click **Model Slicer**. To add the block as the exclusion point, in the model, right-click the block and select **Model Slicer > Add as Exclusion Point**.

Version History

Introduced in R2015b

See Also

“Highlight Functional Dependencies” | `removeExclusionPoint`

removeExclusionPoint

Class: `SLSlicerAPI.SLSlicer`
Package: `SLSlicerAPI`

Removes the exclusion point from the model slice

Syntax

```
removeExclusionPoint(obj,ExclusionPoint)
```

Description

`removeExclusionPoint(obj,ExclusionPoint)` removes the `ExclusionPoint` from the model slice for dependency analysis.

Input Arguments

`obj` — Model Slicer configuration

slslicer object

Class containing the options of Model Slicer configuration.

`ExclusionPoint` — Slice exclusion point

character vector | cell array of character vectors | string array

Specify the slice exclusion point to remove from the model slice. Table summarizes the options for slice exclusion point.

Starting Point	Description

Block handles	Specifies the block handle. To get the block handle, use the `getSimulinkBlockHandle` command.
	
	Example:
	
	`blkH = get_param('sldvSliceClimateControlExample/Out1','Handle');`
	`removeExclusionPoint(obj,blkH);`
	
	To remove multiple block handles exclusion point, use cell array, for example:
	
	`blkH = {get_param('sldvSliceClimateControlExample/Out1','Handle'),`
	`get_param('sldvSliceClimateControlExample/Out2','Handle')}`
	`removeExclusionPoint(obj,blkH);`

Starting Point	Description

Block paths	Block path name, specified as a character vector or a cell array of character vectors.
	
	Example:
	
	`bPath = {'sldvSliceClimateControlExample/Out1'};`
	`removeExclusionPoint(obj, bPath);`
	
	To remove multiple block handles exclusion point, use cell array, for example:
	
	`bPath = {'sldvSliceClimateControlExample/Out1',...`
	`'sldvSliceClimateControlExample/Heater/HeaterAct'};`
	`removeExclusionPoint(obj, bPath);`

SID	Simulink Identifier, a unique designation assigned to a Simulink block or model annotation. To get the SID, use the `Simulink.ID.getSID` command. Example: `removeExclusionPoint(obj, 'sldvSliceClimateControlExa`

LineHs	Handles of line that connects from the Inport block to the Outport block. To get the Line Handle, use `get_param` command.
	
	Example:
	
	`lh1 = get_param('sldvSliceClimateControlExample/Heater/Heat','Lin`
	`lh2 = get_param('sldvSliceClimateControlExample/Heater','LineHand`
	`LineHs = [lh1.Inport(1), lh2.Outport(2)];`
	`removeExclusionPoint(obj,LineHs);`

Alternatives

To open the Model Slicer manager, in the Simulink Editor, select the **APPS** tab, and click **Model Slicer**. To add the block as the starting point, in the model, right-click the block and select **Model Slicer > Remove Exclusion Point**.

Version History

Introduced in R2015b

See Also

“Highlight Functional Dependencies” | `addExclusionPoint`

removeStartingPoint

Class: SLSlicerAPI.SLSlicer
Package: SLSlicerAPI

Remove starting point for model slice

Syntax

```
removeStartingPoint(obj,StartingPoint)
removeStartingPoint(obj,PortHandle,busElementPath)
```

Description

`removeStartingPoint(obj,StartingPoint)` removes the starting point in `StartingPoint` from the model slice in `obj` for dependency analysis.

`removeStartingPoint(obj,PortHandle,busElementPath)` removes the starting point defined using the port handle in `PortHandle` and the bus element path in `busElementPath`. Removing a bus element also removes all its children added as starting points.

Examples

Remove Starting using Blocks Handles

Specifies the block handle. To get the block handle, use the `getSimulinkBlockHandle` command.

Add examples to the path:

```
addpath(fullfile(docroot,'toolbox','simulink','examples'));
open_system('sldvSliceClimateControlExample');
```

```
obj = slslicer('sldvSliceClimateControlExample');
blkH = get_param('sldvSliceClimateControlExample/Out1','Handle');
removeStartingPoint(obj,blkH);
```

To remove multiple block handles as the starting point, use a cell array.

```
blkH = {get_param('sldvSliceClimateControlExample/Out1','Handle'),...
        get_param('sldvSliceClimateControlExample/Out2','Handle')};
removeStartingPoint(obj,blkH);
```

Remove Bus Elements from Starting Points

Remove the bus element `upper_saturation_limit` from the bus `limits` as starting point. This example shows how to first add `upper_saturation_limit`.

```
openExample('sldemo_mdlref_bus');
obj = slslicer('sldemo_mdlref_bus');
ph = get_param('sldemo_mdlref_bus/COUNTERBUSCreator1', 'PortHandles');
obj.addStartingPoint(ph.Outport, "limits.upper_saturation_limit");
obj.highlight;
```

Once you have added bus element `upper_saturation_limit` as a starting point, you can remove this bus element from starting point by using:

```
obj.removeStartingPoint(ph.Outport, "limits.upper_saturation_limit");
```

Removing the bus `limits` from starting point will also remove its child `upper_saturation_limit` from starting point.

Remove Starting using Blocks Paths

Block path name, specified as a character vector or a cell array of character vectors.

```
bPath = {'sldvSliceClimateControlExample/Out1'};
removeStartingPoint(obj, bPath);
```

To remove multiple block paths starting point, use cell array, for example:

```
bPath = {'sldvSliceClimateControlExample/Out1',...
'sldvSliceClimateControlExample/Heater/HeaterAct'};
removeStartingPoint(obj, bPath);
```

Remove Starting using SID

Simulink Identifier, a unique designation assigned to a Simulink block or model annotation. To get the SID, use the `Simulink.ID.getSID` command.

```
removeStartingPoint(obj, 'sldvSliceClimateControlExample:39')
```

Remove Starting using LineHs

Handles of line that connects from the Inport block to the Outport block. To get the Line Handle, use `get_param` command.

```
 lh1 = get_param('sldvSliceClimateControlExample/Heater/Heat','LineHandles');
 lh2 = get_param('sldvSliceClimateControlExample/Heater','LineHandles');
 LineHs = [lh1.Inport(1), lh2.Outport(2)];
removeStartingPoint(obj,LineHs);
```

Input Arguments

`obj` — Model Slicer configuration

slslicer object

Class containing of Model Slicer configuration options.

`StartingPoint` — Slice starting point

character vector | cell array of character vectors | string array | bus

Block or signal line from which the Model Slicer analysis is started.

`PortHandle` — outport handle

cell array of character vectors | string array | bus

The outport handle of a port emitting the bus.

`busElementPath` — Path for bus element

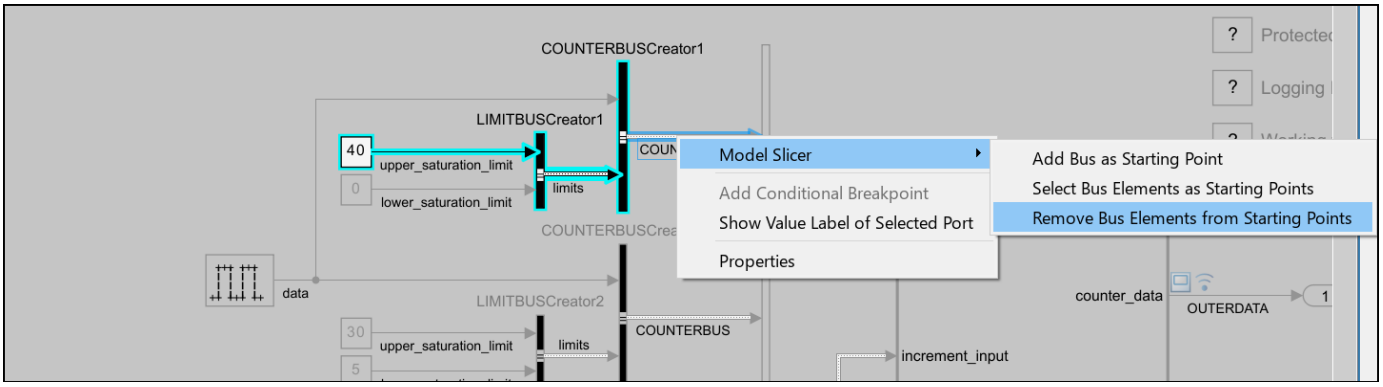
cell array of character vectors | string array

The path for the bus element from which the Model Slicer analysis is started.

Alternatives

To open the Model Slicer manager, in the Simulink Editor, on the **Apps** tab, click **Model Slicer**. To remove a block or signal as the starting point, in the model, right-click the bus signal and select **Model Slicer > Remove as Starting Point**.

To remove bus element(s) as a starting point, in the model, right-click the bus signal and select **Model Slicer > Remove Bus Elements from Starting Points**. This removes all the bus element starting points corresponding to this bus signal.



Remove Bus Elements from Starting Points

Version History

Introduced in R2015b

See Also

“Highlight Functional Dependencies” | `addStartingPoint`

addStartingPoint

Class: `SLSlicerAPI.SLSlicer`
Package: `SLSlicerAPI`

Add block handles, block paths, bus elements, or Simulink Identifiers (SID) as starting point

Syntax

```
addStartingPoint(obj,StartingPoint)
addStartingPoint(obj,PortHandle,busElementPath)
```

Description

`addStartingPoint(obj,StartingPoint)` adds block(s), block path(s), or Simulink Identifier (SID) as the `StartingPoint` to the model slice in `obj` for dependency analysis.

`addStartingPoint(obj,PortHandle,busElementPath)` adds a bus element as the starting point. Specify the port emitting the bus signal in `PortHandle` and the bus element path corresponding to the bus signal in `busElementPath`. You can add the starting point bus signal in the form of `subsignal1.subsignal2` and so on. You can specify the `subsignal` up to any level. You should not specify the top level signal name corresponding to the bus.

Examples

Starting Point at Block Handles

Specifies the block handle. To get the block handle, use the `getSimulinkBlockHandle` command.

Add examples to the path:

```
addpath(fullfile(docroot,'toolbox','simulink','examples'));
open_system('sldvSliceClimateControlExample');

blkH = get_param('sldvSliceClimateControlExample/Out1','Handle');
addStartingPoint(obj,blkH);
```

To add multiple block handles as the starting point, use a cell array.

```
blkH = {get_param('sldvSliceClimateControlExample/Out1','Handle'),...
        get_param('sldvSliceClimateControlExample/Out2','Handle')};
addStartingPoint(obj,blkH);
```

Starting Point as Bus Element

Specify bus element `limits` as the starting point.

```
openExample('sldemo_mdlref_bus');
obj = slslicer('sldemo_mdlref_bus');
ph = get_param('sldemo_mdlref_bus/COUNTERBUSCreator1', 'PortHandles');
obj.addStartingPoint(ph.Outport, "limits");
obj.highlight;
```

Starting Point at Block Path

Specify the block path as a character vector.

```
bPath = {'sldvSliceClimateControlExample/Out1'};
addStartingPoint(obj, bPath);
```

To add multiple block paths as the starting point, use a cell array, for example.

```
bPath = {'sldvSliceClimateControlExample/Out1',...
'sldvSliceClimateControlExample/Heater/HeaterAct'};
addStartingPoint(obj, bPath);
```

Starting Point at SID

Specify Simulink Identifier, a unique designation assigned to a Simulink block or model annotation, as the starting point. To get the SID, use the `Simulink.ID.getSID` command.

```
addStartingPoint(obj, 'sldvSliceClimateControlExample:39')
```

Starting Point at LineHs

Specify handles of the lines connecting the Inport block to the Outport block. To get the Line Handle, use the `get_param` command.

```
 lh1 = get_param('sldvSliceClimateControlExample/Heater/Heat','LineHandles');
 lh2 = get_param('sldvSliceClimateControlExample/Heater','LineHandles');
 LineHs = [lh1.Inport(1), lh2.Outport(2)];
addStartingPoint(obj,LineHs);
```

Input Arguments

`obj` — Model Slicer configuration

slslicer object

Model Slicer configuration, specified as an slslicer object containing Model Slicer configuration options.

`StartingPoint` — Starting point for model slice

character vector | cell array of character vectors | string array | bus

Starting point for the model slice, specified as character vector, cell array, or bus. You can specify block name(s), block path(s), or Simulink Identifier (SID) as the starting point.

`PortHandle` — Name of port

cell array of character vectors | string array | bus

Name of the port emitting the bus signal, specified as cell array of character vectors or string array.

`busElementPath` — Path for bus element

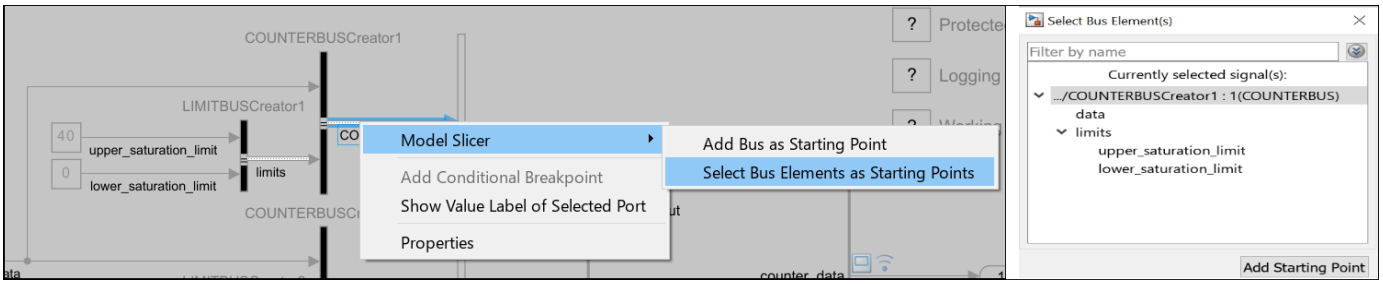
cell array of character vectors | string array

Path for the bus element from which you want to start the Model Slicer analysis is started, specified as cell array of character vectors or string array.

Alternatives

To open the Model Slicer manager, in the Simulink Editor, on the **Apps** tab, click **Model Slicer**. To add a block or signal as the starting point, in the model, right-click the bus signal and select **Model Slicer > Add as Starting Point**.

To add bus element(s) as the starting point in the model, right-click the bus signal and select **Model Slicer > Add Bus Elements as Starting Points**. The **Select Bus Element(s)** window opens, where you can select the bus element(s) that you want to add as starting point(s) and click **Add Starting Point**.



Add Bus Elements as Starting Points

Version History

Introduced in R2015b

See Also

“Highlight Functional Dependencies” | `removeStartingPoint`

activate

Class: `SLSlicerAPI.SLSlicer`
Package: `SLSlicerAPI`

Activates the model for analysis

Syntax

```
activate(obj)
```

Description

`activate(obj)` activates the model for dependency analysis.

Input Arguments

`obj` — Model Slicer configuration

slslicer object

Object containing Model Slicer configuration options.

Example

Activate Slicer Object

```
open_system('sldvSliceClimateControlExample');
obj=slslicer('sldvSliceClimateControlExample');
activate(obj);
```

Terminate Slicer Object

```
terminate(obj)
```

Alternatives

To open the Model Slicer manager, in the Simulink Editor, select the **APPS** tab, and click **Model Slicer**. Invoking Model Slicer on the model automatically activates the model for analysis.

Version History

Introduced in R2015b

See Also

`simulate` | `terminate`

ActiveBlocks

Class: `SLSlicerAPI.SLSlicer`
Package: `SLSlicerAPI`

Returns the active non-virtual block handles

Syntax

```
ActiveBlocks(obj)
```

Description

`ActiveBlocks(obj)` returns the active non-virtual block handles.

Input Arguments

`obj` — Model Slicer configuration

slslicer object

Object containing Model Slicer configuration options.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes.

`IncludeVirtual` — Include virtual blocks in the highlight

false (default) | boolean

This is an optional argument for including the virtual blocks in the list of blocks highlighted by Model Slicer.

Example:

```
open_system('sldvdemo_cruise_control');
slicerObj = slslicer('sldvdemo_cruise_control');
slicerObj.addStartingPoint('sldvdemo_cruise_control/Controller/throt');
slicerObj.highlight;
allActiveBlocks = slicerObj.ActiveBlocks('IncludeVirtual', true);
```

Example

```
open_system('sldvSliceClimateControlExample');
obj=slslicer('sldvSliceClimateControlExample');
ActiveBlocks(obj);
```

Alternatives

To open the Model Slicer manager, in the Simulink Editor, select the **APPS** tab, and click **Model Slicer**. To view the active blocks, simulate the model while the model is in slicer mode.

Version History

Introduced in R2015b

See Also

`activate` | `terminate`

highlight

Class: `SLSlicerAPI.SLSlicer`
Package: `SLSlicerAPI`

Updates the model highlighting

Syntax

```
highlight(obj)
```

Description

`highlight(obj)` updates the model highlighting.

Input Arguments

`obj` — Model Slicer configuration

slslicer object

Object containing Model Slicer configuration options.

Example

```
open_system('sldvSliceClimateControlExample');
obj=slslicer('sldvSliceClimateControlExample');
highlight(obj);
```

Alternatives

To open the Model Slicer manager, in the Simulink Editor, select the **APPS** tab, and click **Model Slicer**. To highlight the active blocks, simulate the model while the model is in slicer mode.

Version History

Introduced in R2015b

See Also

`slsliceroptions` | `terminate`

simulate

Class: SLSlicerAPI.SLSlicer
Package: SLSlicerAPI

Simulates a test case for dynamic slicing from time "t1" to time "t2"

Syntax

```
simulate(obj,t1,t2)
```

Description

`simulate(obj,t1,t2)` simulates a test case for dynamic slicing from time "t1" to time "t2". If t1 is a non-zero value, the simulation first happens from zero to t2 and then the results reported are trimmed automatically for the specified duration t1 to t2.

Input Arguments

`obj` — Model Slicer configuration

slslicer object

Object containing Model Slicer configuration options.

`(t1, t2)` — Upper(t2) and lower(t1) time period boundaries for the simulation results of the sliced model

integer, Floating-Point Number

Desired time period to view simulation results for the sliced model. Post simulation and analysis the simulation results for the selected time period (t1 to t2) is displayed.

Example

```
open_system('sldvSliceClimateControlExample');
obj=slslicer('sldvSliceClimateControlExample');
simulate(obj);
```

Alternatives

To open the Model Slice Manager, in the Simulink Editor, select the **APPS** tab, and click **Model Slicer**. The initial simulation of the model in slicer mode gives you the option to select the upper boundary of the simulation time (t2), with 0 being the lower boundary (t1). Post simulation and analysis you can view the simulation results for the selected time period (t1 to t2). To select the desired time period to view simulation results, from the **Model Slice Manager**, enter the t1 and t2 values in the **Time window** field and click **Highlight**.

Version History

Introduced in R2015b

See Also

`terminate`

terminate

Class: SLSlicerAPI.SLSlicer
Package: SLSlicerAPI

Discards the analysis data and reverts the model highlighting (invoked when the object goes out of scope)

Syntax

```
terminate(obj)
```

Description

`terminate(obj)` discards the analysis data and reverts the model highlighting (invoked when the object goes out of scope).

Input Arguments

`obj` — Model Slicer configuration
slslicer object

Object containing Model Slicer configuration options.

Example

```
open_system('sldvSliceClimateControlExample');
obj=slslicer('sldvSliceClimateControlExample');
activate(obj);
terminate(obj);
```

Alternatives

To open the Model Slicer manager, in the Simulink Editor, select the **APPS** tab, and click **Model Slicer**. To terminate slicer mode on the model, simply close the Model Slice Manager.

Version History

Introduced in R2015b

See Also

`activate` | `simulate`

unlock

Class: `SLSlicerAPI.SLSlicer`
Package: `SLSlicerAPI`

Discards the analysis data while retaining model highlights

Syntax

```
unlock(obj)
```

Description

`unlock(obj)` discards the analysis data while retaining model highlights.

Input Arguments

`obj` — Model Slicer configuration

slslicer object

Class containing Model Slicer configuration options.

Example

```
open_system('sldvSliceClimateControlExample');
obj=slslicer('sldvSliceClimateControlExample');
unlock(obj);
```

Alternatives

To open the Model Slicer manager, in the Simulink Editor, select the **APPS** tab, and click **Model Slicer**. To unlock the model, click <icon> on the **Model Slice Manager** window.

Version History

Introduced in R2015b

See Also

`simulate` | `terminate`

SLSlicerAPI.SLSlicer class

Package: `SLSlicerAPI`

Class representing a set of Model Slicer functionality

Description

The `SLSlicerAPI.SLSlicer` class provides a set of methods to access Model Slicer functionality from the MATLAB command line. Use an instance of `SLSlicerAPI.SLSlicer` class to:

- Activate or terminate slice simulation.
- Add or remove starting point, exclusion point, constraints, slice component and configuration.
- Highlight the functional dependencies.
- Set time window and simulate slice.
- Refine slice for dead logic analysis.

Creation

Description

`slslicer(model)` creates a Model Slicer configuration object for the model `model` by exposing the methods for invoking the Model Slicer.

`slslicer(model, opts)` creates a Model Slicer configuration object for the model `model` by using the options object `opts`, as defined by `slsliceroptions`.

Input Arguments

`model` — Name or handle of model
string

Name of the model whose Model Slicer options object you configure. `slslicer` uses the Model Slicer configurations associated with the `model`, as defined by `slsliceroptions`.

Data Types: `string`

`opts` — Options you attach to a model or save to a file
structure

Structure containing the options for the Model Slicer configuration. `slsliceroptions` defines the options object `opts`.

Data Types: `struct`

Properties

`Name` — Configuration name
preconfigured values (default) | `character vector`

The name of the Model Slicer configuration.

`Description` — Configuration description
empty (default) | `character vector`

A description of the Model Slicer configuration.

`SignalPropagation` — Direction of dependency analysis
empty (default) | `upstream` | `downstreambidirectional`

The direction of the dependency analysis.

`Color` — Highlight color
preconfigured (default) | `R` | `BG`

The color of a functional dependency highlight.

`DeadLogicFile` — `slslicex` file location
empty (default) | `character vector`

The location of the `slslicex` file containing the dead logic data.

`UseDeadLogic` — Specify dead logic analysis
`false` (default) | `boolean`

A flag for specifying whether the analysis should exclude elements of dead logic.

`CoverageFile` — `slslicex` file location
empty (default) | `character vector`

The location of the `slslicex` file containing simulation data.

`UseTimeWindow` — Specify simulation time window analysis
`false` (default) | `Boolean`

A flag for specifying whether the analysis should use simulation time window information.

`SliceComponent` — Specify slice components
`empty` (default) | `Struct`

Model blocks or subsystems that you add as slice components.

`Constraint` — Constraint model elements
`empty` (default) | `Struct`

Model elements that you add as constraints.

`ExclusionPoint` — Exclusion point model elements
`empty` (default) | `Struct`

Model elements that you specify as exclusion points.

`StartingPoint` — Starting point model elements
`empty` (default) | `Struct`

Model elements that you add as starting points.

`InlineOptions.Libraries` — Inline model elements inside sliced libraries
`True` (default) | `Boolean`

Specify whether to inline model items inside sliced libraries within the sliced model and maintain the library links.

`InlineOptions.ModelBlocks` — Inline model elements from referenced model within sliced model
`True` (default) | `Boolean`

Specify whether to inline model items from the model referenced by the Model block within the sliced model and remove the Model block.

`InlineOptions.Masks` — Inline model elements inside sliced masked subsystems within sliced model
`True` (default) | `Boolean`

Specify whether to inline model elements inside sliced masked subsystems within the sliced model and retain the mask in the sliced model.

`InlineOptions.Variants` — Inline model elements to the sliced model from active variants
`True` (default) | `Boolean`

Specify whether to inline model items to the sliced model from the active variant and remove the variants.

`InlineOptions.SubsystemReferences` — Inline subsystem reference contents
`True` (default) | `Boolean`

Specify whether to inline contents of subsystem references in sliced models.

`SliceOptions.ExtendSubsystems` — Expand trivial subsystem in sliced model
`True` (default) | `Boolean`

Expand trivial subsystem in sliced model and set subsystem boundary.

`SliceOptions.RootLevelInterfaces` — Retain root-level ports in sliced model
`True` (default) | `Boolean`

Retain root-level input and output ports in sliced model.

`SliceOptions.SignalObservers` — Retain signal observers in sliced model
`False` (default) | `Boolean`

Retain signal observer, such as scopes, displays, and test condition blocks in the sliced model.

Methods

Public Methods

ActiveBlocks	Returns the active non-virtual block handles
activate	Activates the model for analysis
addConstraint	Adds the constraint on Switch or Multiport switch blocks, and Stateflow state or transition

addExclusionPoint	Adds block handles, block paths, or Simulink Identifiers (SID) as slice exclusion point
addSliceComponent	Adds a model or a subsystem as a slice component
addStartingPoint	Add block handles, block paths, bus elements, or Simulink Identifiers (SID) as starting point
highlight	Updates the model highlighting
refineDeadLogic	Updates the model highlighting with dead logic refinement
removeConstraint	Removes the constraint from the model slice
removeDeadLogic	Removes the dead logic refinement
removeExclusionPoint	Removes the exclusion point from the model slice
removeSliceComponent	Removes the slice component from the model slice
removeStartingPoint	Remove starting point for model slice
report	Generate web report for the model
simulate	Simulates a test case for dynamic slicing from time "t1" to time "t2"
slice	Creates a sliced model from the model highlight
terminate	Discards the analysis data and reverts the model highlighting (invoked when the object goes out of scope)
unlock	Discards the analysis data while retaining model highlights

Examples

Add Starting Point and Highlight the Model Slice

Add a new starting point to the active Model Slicer configuration, and then highlight the model.

Open the `sldvSliceClimateControlExample` example model.

```
addpath(fullfile(docroot,'toolbox','simulink','examples'));
open_system('sldvSliceClimateControlExample');
```

Create a Model Slicer configuration object for the model by using `slslicer`.

```
obj = slslicer('sldvSliceClimateControlExample');
```

To compile the model and prepare the model slice for dependency analysis, activate the slice highlighting mode of Model Slicer.

```
activate(obj);
```

Add the `Out1` outport block as the starting point and highlight the model slice.

```
addStartingPoint(obj,'sldvSliceClimateControlExample/Out1');
highlight(obj);
```

The area of the model upstream of the starting point and which is active during simulation is highlighted.

Terminate the model highlighting mode and discard the analysis data.

```
terminate(obj);
```

Version History

Introduced in R2015b

See Also

`slsliceroptions` | `slslicertrace`

Topics

“Workflow for Dependency Analysis”
“Model Slicer Considerations and Limitations”

slice

Class: `SLSlicerAPI.SLSlicer`
Package: `SLSlicerAPI`

Creates a sliced model from the model highlight

Syntax

```
slice(obj,'sldvSliceClimateControlExample_sliced')
```

Description

`slice(obj,'sldvSliceClimateControlExample_sliced')` creates a sliced model from the model highlight.

Input Arguments

`obj` — Model Slicer configuration
slslicer object

Object containing Model Slicer configuration options.

`sldvSliceClimateControlExample_sliced` — Sliced model
character vector

Name of generated slice model.

Alternatives

To open the Model Slicer manager, in the Simulink Editor, select the **APPS** tab, and click **Model Slicer**. Invoking Model Slicer on the model automatically activates the model for analysis.

Version History

Introduced in R2015b

See Also

`simulate` | `terminate`

addConstraint

Class: `SLSlicerAPI.SLSlicer`
Package: `SLSlicerAPI`

Adds the constraint on Switch or Multiport switch blocks, and Stateflow state or transition

Syntax

```
addConstraint(obj,bpath,{1, 1})
```

Description

`addConstraint(obj,bpath,{1, 1})` adds the constraint from the model slice.

Input Arguments

`obj` — Model Slicer configuration

slslicer object

Object containing Model Slicer configuration options.

`bpath` — Block path handle

character vector

Path or handle to Switch or Multiport Switch or Stateflow State or Transition.

Example: bpath={'sldvSliceClimateControlExample/Refrigeration/On'};

Alternatives

To open the Model Slicer manager, in the Simulink Editor, select the **APPS** tab, and click **Model Slicer**. Invoking Model Slicer on the model automatically activates the model for analysis.

Version History

Introduced in R2015b

See Also

`simulate` | `terminate`

removeConstraint

Class: `SLSlicerAPI.SLSlicer`
Package: `SLSlicerAPI`

Removes the constraint from the model slice

Syntax

```
removeConstraint(obj,bpath)
```

Description

`removeConstraint(obj,bpath)` removes the constraint from the model slice.

Input Arguments

`obj` — Model Slicer configuration

slslicer object

Object containing Model Slicer configuration options.

`bpath` — Block path or handle

character vector

Path of the constraint that needs to be added as slice component.

Example: bpath={'sldvSliceClimateControlExample/Refrigeration/On'};

Alternatives

To open the Model Slicer, in the Simulink Editor, select the **APPS** tab, and click **Model Slicer**. Invoking Model Slicer on the model automatically activates the model for analysis.

Version History

Introduced in R2015b

See Also

`simulate` | `terminate`

addSliceComponent

Class: `SLSlicerAPI.SLSlicer`
Package: `SLSlicerAPI`

Adds a model or a subsystem as a slice component

Syntax

```
addSliceComponent(obj, bpath)
```

Description

`addSliceComponent(obj, bpath)` adds a model or a subsystem as a slice component.

Input Arguments

`obj` — Model Slicer configuration

slslicer object

Object containing Model Slicer configuration options.

`bpath` — Block path or handle

character vector

Path of the model or subsystem that needs to be added as slice component.

Alternatives

To open the Model Slicer manager, in the Simulink Editor, select the **APPS** tab, and click **Model Slicer**. Invoking Model Slicer on the model automatically activates the model for analysis.

Version History

Introduced in R2015b

See Also

`simulate` | `terminate`

removeSliceComponent

Class: `SLSlicerAPI.SLSlicer`
Package: `SLSlicerAPI`

Removes the slice component from the model slice

Syntax

```
removeSliceComponent(obj)
```

Description

`removeSliceComponent(obj)` removes the slice component from the model slice.

Input Arguments

`obj` — Model Slicer configuration

slslicer object

Object containing Model Slicer configuration options.

Alternatives

To open the Model Slicer manager, in the Simulink Editor, select the **APPS** tab, and click **Model Slicer**. Invoking Model Slicer on the model automatically activates the model for analysis.

Version History

Introduced in R2015b

See Also

`simulate` | `terminate`

report

Class: SLSlicerAPI.SLSlicer
Package: SLSlicerAPI

Generate web report for the model

Syntax

```
report(obj)
```

Description

`report(obj)` creates a web view report from the model highlight using the Simulink Report Generator™.

Input Arguments

`obj` — Model Slicer configuration

slslicer object

Object containing Model Slicer configuration options.

Example

Activate Slicer Object

```
open_system('sldvdemo_cruise_control');
slicerObj = slslicer('sldvdemo_cruise_control');
slicerObj.addStartingPoint('sldvdemo_cruise_control/throt');
slicerObj.highlight;
slicerObj.report;
```

Terminate Slicer Object

```
terminate(obj)
```

Alternatives

Click on the Report button from the model slicer app after analysis.

Version History

Introduced in R2015b

See Also

`simulate` | `addStartingPoint`

refineDeadLogic

Class: `SLSlicerAPI.SLSlicer`
Package: `SLSlicerAPI`

Updates the model highlighting with dead logic refinement

Syntax

```
refineDeadLogic(obj,'sldvSlicerdemo_dead_logic',analysis_time)
```

Description

`refineDeadLogic(obj,'sldvSlicerdemo_dead_logic',analysis_time)` updates the model highlighting with dead logic refinement

Input Arguments

`obj` — Model Slicer configuration
slslicer object

Object containing Model Slicer configuration options.

`sldvSlicerdemo_dead_logic` — Simulink model for Model Slicer analysis
character vector

Model or subsystem being analyzed for dead logic.

`analysis_time` — Duration for analysis
duration

Amount of time used for dead logic analysis.

Example

1. Open example `AnalyzeTheDeadLogicExample`. The model to demonstrate refining dead logic is located in this example's folder.

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

2. Create a Slicer API object, and add the **target** outport as a starting point.

   ```
   slicerObj = slslicer(model);
   slicerObj.addStartingPoint([model '/Controller/target']);
   ```

3. Observe highlight on model.

   ```
   slicerObj.highlight;
   ```

4. Refine Dead Logic for a time period.

   ```
   slicerObj.refineDeadLogic(model, 300);
   ```

5 Observe refined highlight on model

```
slicerObj.highlight;
```

Alternatives

To perform dead logic refinement using the Model Slicer UI, open the Model Slicer manager, from the Simulink Editor, select the **APPS > Model Slicer**. In the Model Slicer pane, you can perform the refinement using **Refine Dead Logic** section. For more information, see “Refine Dead Logic for Dependency Analysis”.

Version History

Introduced in R2015b

See Also

`simulate` | `terminate`

removeDeadLogic

Class: `SLSlicerAPI.SLSlicer`
Package: `SLSlicerAPI`

Removes the dead logic refinement

Syntax

```
removeDeadLogic(obj,'sldvSlicerdemo_dead_logic')
```

Description

`removeDeadLogic(obj,'sldvSlicerdemo_dead_logic')` removes the dead logic refinement.

Input Arguments

`obj` — Model Slicer configuration
slslicer object

Object containing Model Slicer configuration options.

`sldvSlicerdemo_dead_logic` — Simulink Model for Model Slicer analysis
character vector

Model or subsystem that has dead logic refinement applied.

Alternatives

To open the Model Slicer manager, in the Simulink Editor, select the **APPS** tab, and click **Model Slicer**. Invoking Model Slicer on the model automatically activates the model for analysis.

Version History

Introduced in R2015b

See Also

`simulate` | `terminate`

ModelAdvisor.setDefaultConfiguration

Package: `ModelAdvisor`

Set the Model Advisor configuration

Syntax

```
ModelAdvisor.setDefaultConfiguration('ConfigFilePath')
```

Description

`ModelAdvisor.setDefaultConfiguration('ConfigFilePath')` specifies the configuration for the Model Advisor. The configuration defines the organization of the folders and checks in the Model Advisor.

To reset the configuration to the default configuration, pass an empty string to this function (that is, `ModelAdvisor.setDefaultConfiguration('')`). If you do not specify a custom configuration as the default, the default is the shipping configuration.

Note You create a custom configuration by using the Model Advisor Configuration Editor. You can specify your custom configuration as the default configuration.

Examples

Set the Model Advisor Configuration

To set the configuration that is applied to the Model Advisor, at the MATLAB command line:

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

Input Arguments

`ConfigFilePath` — File path to configuration file

character vector | string scalar

Full file path to folder containing the configuration file that contains Model Advisor custom configuration.

Example: 'C:\temp\demoConfiguration.json'

Data Types: `char`

Version History

Introduced in R2020a

See Also

`ModelAdvisor.Check`

Topics

“Use the Model Advisor Configuration Editor to Customize the Model Advisor”

ModelAdvisor.getModelConfiguration

Package: `ModelAdvisor`

Get Model Advisor configuration associated with model

Syntax

```
ConfigFileName = ModelAdvisor.getModelConfiguration(ModelName)
```

Description

`ConfigFileName = ModelAdvisor.getModelConfiguration(ModelName)` returns the name of the Model Advisor configuration file associated with the specified model, `ModelName`.

Examples

Get the Model Advisor Configuration Associated with a Model

Get the configuration file associated with the model `vdp`.

Load the model `vdp`. In the MATLAB Command Window, enter:

```
load_system("vdp")
```

Suppose you have a Model Advisor configuration file named `config.json` in your current working directory. Associate the configuration `config.json` with the model `vdp` by entering:

```
ModelAdvisor.setModelConfiguration("vdp","config.json")
```

View the Model Advisor configuration file associated with the model.

```
ModelAdvisor.getModelConfiguration("vdp")
```

```
ans =

    'config.json'
```

Input Arguments

`ModelName` — Name of model

character vector | string scalar

Name of a model, specified as a character vector or string scalar.

Example: "vdp"

Data Types: `char` | `string`

Output Arguments

`ConfigFileName` — Name of Model Advisor configuration file

character vector

Name of the Model Advisor configuration file, returned as a character vector. If a model does not have an associated Model Advisor configuration file, `ModelAdvisor.getModelConfiguration` returns an empty character array.

Version History

Introduced in R2022a

See Also

`ModelAdvisor.setModelConfiguration`

Topics

“Load and Associate a Custom Configuration with a Model”

ModelAdvisor.setModelConfiguration

Package: `ModelAdvisor`

Set Model Advisor configuration associated with model

Syntax

```
ModelAdvisor.setModelConfiguration(ModelName,ConfigFile)
```

Description

`ModelAdvisor.setModelConfiguration(ModelName,ConfigFile)` sets the Model Advisor configuration file `ConfigFile` as the configuration associated with the model `ModelName`.

For Model Advisor configuration files created in R2021b or earlier, use R2022a or later to open and re-save the configuration file in the Model Advisor Configuration Editor before associating the file with a model.

Examples

Set the Model Advisor Configuration Associated with a Model

Set the configuration file associated with the model `vdp`.

Load the model `vdp`. In the MATLAB Command Window, enter:

```
load_system("vdp")
```

Suppose you have a Model Advisor configuration file named `config.json` in your current working directory. Associate the configuration `config.json` with the model `vdp` by entering:

```
ModelAdvisor.setModelConfiguration("vdp","config.json")
```

Input Arguments

`ModelName` — Name of model

character vector | string scalar

Name of a model, specified as a character vector or string scalar.

Example: "vdp"

Data Types: `char` | `string`

`ConfigFile` — Name or path of Model Advisor configuration file

character vector | string scalar

Name or path of a Model Advisor configuration file, specified as a character vector or string scalar.

Example: "config.json"

Example: "C:\temp\demoConfiguration.json"

Data Types: `char` | `string`

Alternative Functionality

App

You can also use the Model Advisor to set the Model Advisor configuration associated with a model. In the Model Advisor, load your configuration file and click **Open > Associate Configuration to Model**. For more information, see “Load and Associate a Custom Configuration with a Model”.

Version History

Introduced in R2022a

See Also

`ModelAdvisor.getModelConfiguration`

Topics

“Load and Associate a Custom Configuration with a Model”

Simulink.CloneDetection.findClones

Find clones in a model

Syntax

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

Description

`cloneResults = Simulink.CloneDetection.findClones(model)` finds and returns subsystem clones for a specified `model`.

`cloneResults = Simulink.CloneDetection.findClones(model, cloneDetectionSettings)` uses the conditions specified in a `cloneDetectionSettings` object for a specified `model`.

`cloneResults = Simulink.CloneDetection.findClones(cloneDetectionSettings)` uses the conditions specified in a `cloneDetectionSettings` object.

Examples

Find Clones with Default Conditions

This example shows how to execute `findClones` function and store the results to `cloneResults` object. For an example model, see `ex_clone_detection`.

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

```
cloneResults =

  Results with properties:

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

```
cloneResults.Clones
```

```
  Results with properties:

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

Find Clones with Customised Settings Object in a single model

This example shows how to execute `findClones` function using the customised settings specified in `cloneDetectionSettings` object and change the value of property `ParamDifferenceThreshold` to zero.

```
cloneDetectionSettings = Simulink.CloneDetection.Settings();
cloneDetectionSettings.ParamDifferenceThreshold = 0;

cloneResults = Simulink.CloneDetection.findClones('ex_clone_detection', cloneDetectionSettings)

cloneResults =

  Results with properties:

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

Find Clones with Customised Settings Object in the folders containing models

This example shows how to execute `findClones` function using the customised settings specified in `cloneDetectionSettings` object and mention the folders name in `Folders` property.

```
cloneDetectionSettings = Simulink.CloneDetection.Settings();
cloneDetectionSettings.Folders = {'Folder 1', 'Folder 2', 'Folder 3'};

cloneResults = Simulink.CloneDetection.findClones(cloneDetectionSettings)


cloneResults =

  Results with properties:

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

Input Arguments

`model` — Model name

character vector

Model name, specified as a character vector.

`cloneDetectionSettings` — Configuration to find clones

`Simulink.CloneDetection.Settings` object

Clone Detection Settings, specified as a `Simulink.CloneDetection.Settings` object.

Output Arguments

`cloneResults` — Identified clones

`Simulink.CloneDetection.Results` object

Identified clones, returned as a `Simulink.CloneDetection.Results` object.

Version History

Introduced in R2021a

See Also

Detect and Replace Clones Using API | Simulink.CloneDetection.replaceClones on page 1-654

Simulink.CloneDetection.replaceClones

Replace clones in a model

Syntax

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

Description

`cloneReplacementResults = Simulink.CloneDetection.replaceClones(cloneResults)` replaces clones and returns the `cloneReplacementResults` object for the specified `cloneResults`.

`cloneReplacementResults = Simulink.CloneDetection.replaceClones(cloneResults, cloneReplacementConfig)` uses the replacement configurations specified in `cloneReplacementConfig`.

Examples

Replace Clones with Default Conditions

This example shows how to execute `replaceClones` function using `cloneResults` object and store the results to `cloneReplacementResults` object.

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

```
cloneReplacementResults =

  ReplacementResults with properties:

    ReplacedClones: [1×5 struct]
    ExcludedClones: {}
```

Replace clones with a Customised Configuration Object

This example shows how to execute `replaceClones` function using the replacement configurations specified in `cloneReplacementConfig` object.

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

```
cloneReplacementResults =

  ReplacementResults with properties:
```

```
    ReplacedClones: [1×4 struct]
    ExcludedClones: [1×1 struct]
```

Input Arguments

`cloneResults` — Identified clones

`Simulink.CloneDetection.Results` object

Clones identified in a model, specified as a `Simulink.CloneDetection.Results` object.

`cloneReplacementConfig` — Settings to replace clones

`Simulink.CloneDetection.ReplacementConfig` object

Settings to use to replace clones, specified as a `Simulink.CloneDetection.ReplacementConfig` object.

Output Arguments

`cloneReplacementResults` — Replaced clones

`Simulink.CloneDetection.ReplacementResults` object

Clones replaced in the model, returned as a `Simulink.CloneDetection.ReplacementResults` object.

Version History

Introduced in R2021a

See Also

Detect and Replace Clones Using API | Simulink.CloneDetection.findClones on page 1-652

Simulink.CloneDetection.checkEquivalency

Check equivalency of clone-replaced model and original model

Syntax

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

Description

`Simulink.CloneDetection.checkEquivalency(cloneReplacementResults)` checks the equivalency of a model that was updated to replace clones and the original model using a `cloneReplacementResults` object.

Examples

Check Equivalency of the Model

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

```
equivalencyCheckResults =
  equivalencyCheckResults with properties:
    List: [1×1 struct]

equivalencyCheckResults.List =
  struct with fields:
    IsEquivalencyCheckPassed: 1
               OriginalModel: 'm2m_ex_clone_detection_B/snapshot_2020_12_02_17_27_54_ex_clone_detection_B.slx'
                UpdatedModel: 'ex_clone_detection_B.slx'
```

Input Arguments

`cloneReplacementResults` — Results of replace clones operation

`Simulink.CloneDetection.ReplacementResults` object

Results of a replace clones operation, specified as a `Simulink.CloneDetection.ReplacementResults` object.

Version History

Introduced in R2021a

See Also

Detect and Replace Clones Programmatically | Simulink.CloneDetection.findClones on page 1-652 | Simulink.CloneDetection.replaceClones on page 1-654

Simulink.CloneDetection.ReplacementConfig class

Package: `Simulink.CloneDetection`

Conditions for clone replacement function

Description

Use object of the `Simulink.CloneDetection.ReplacementConfig` class to specify certain conditions when calling the `Simulink.CloneDetection.replaceClones` function.

Creation

Description

`Simulink.CloneDetection.ReplacementConfig` creates an object of the `Simulink.CloneDetection.ReplacementConfig` class.

Properties

`LibraryNameToAddSubsystemsTo` — Name for creating library file
character vector

Name of library to create, specified as a character vector.

`IgnoredClones` — Subsystems to ignore during clone replacement
character vector | array of character vector

Subsystems to ignore during clone replacement, specified as a character vector or an array of character vectors.

Methods

Public Methods

addCloneToIgnoreList	Add subsystem to clone replacement ignore list
removeCloneFromIgnoreList	Remove subsystem from clone replacement ignore list

Examples

Store Clone Replacement Configuration Information

This example shows to create and store clone replacement configuration information in a `cloneReplacementConfig` object and change the property of `LibraryNameToAddSubsystemsTo` to `UserDefinedLibName`.

-
```
cloneReplacementConfig = Simulink.CloneDetection.ReplacementConfig('UserDefinedLibName');
```

```
cloneReplacementConfig =
```

```
ReplacementConfig with properties:

  LibraryNameToAddSubsystemsTo: 'UserDefinedLibName'
                 IgnoredClones: {}
```

Version History

Introduced in R2021a

See Also

`Simulink.CloneDetection.Settings` | `Simulink.CloneDetection.ReplacementResults` | `Simulink.CloneDetection.Results`

Simulink.CloneDetection.ReplacementResults class

Package: `Simulink.CloneDetection`

Results of replace clones

Description

Use objects of the `Simulink.CloneDetection.ReplacementResults` class to view the results of a clone replacement operation.

Creation

Description

The `Simulink.CloneDetection.replaceClones` function creates an object of the `Simulink.CloneDetection.ReplacementResults` class when executed. You can use this object as the input argument of the `Simulink.CloneDetection.checkEquivalency` function.

Properties

`ReplacedClones` — List of clones replaced

structure

Name list of the replaced clones and reference subsystem, specified as array of character vector.

- `Name` — Name of the replaced subsystem clone
- `ReferenceSubsystem` — Subsystem used to replace clone

`ExcludedClones` — List of clones excluded

structure

Name list of the excluded clones and reason for exclusion, specified as array of character vector.

- `Name` — Name of the excluded subsystem clone
- `ReasonForExclusion` — Reason for excluding from clone replacement

Examples

View Results of a Replace Clones Operation

This example shows how to analyze the results of replace clones operation.

The replace clones function stores the information in `cloneReplacementResults` object.

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

```
cloneReplacementResults =

  ReplacementResults with properties:

    ReplacedClones: [1×5 struct]
    ExcludedClones: {}
```

```
cloneReplacementResults.ReplacedClones
```

2 To view the `ReplacedClones` field.

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

Version History

Introduced in R2021a

See Also

`Simulink.CloneDetection.Settings` | `Simulink.CloneDetection.ReplacementConfig` | `Simulink.CloneDetection.Results` | `Simulink.CloneDetection.EquivalencyCheckResults`

Simulink.CloneDetection.Results class

Package: `Simulink.CloneDetection`

Results of find clones

Description

Use an object of the `Simulink.CloneDetection.Results` class to analyze the results of the find clone operation.

Creation

Description

The `Simulink.CloneDetection.findClones` function creates an object of the `Simulink.CloneDetection.Results` class when executed. You can use this object as the input argument of the `Simulink.CloneDetection.replaceClones` function.

Note You should not write the `Simulink.CloneDetection.Results` object to avoid wrong execution of replace clones function.

Properties

`Clones` — Detected clone information
structure

Detected clone information, specified as a structure.

`Summary` — Clone detection summary
structure

Number of `CloneGroups`, `SimilarClones`, `ExactClones` and `PotentialReusePercentage`, returned as a structure.

`CloneGroups` — Number of clone groups
uint32

Number of identical clone patterns in the model, specified as a uint32 integer.

`SimilarClones` — Number of similar clones
uint32

Number of similar clones in the model, specified as a uint32 integer

`ExactClones` — Number of exact clones
uint32

Number of exact clones in the model, specified as a uint32 integer.

`Clones` — Total number of clones
uint32

Total number of clones in the model, specified as a uint32 integer.

`PotentialReusePercentage` — Amount of reuse opportunity in percentage
uint32

Amount of reuse opportunity in percentage in the model, specified as a uint32 integer.

`CloneGroups` — Group of clones considered as identical match
structure

Subsystems that are considered as clones, specified as a structure.

`Name` — Clone group name
character vector

Name of the clone group, specified as a character vector.

`Summary` — Clone group summary
structure

Summary of the clone group, specified as a structure with the fields:

- `ParameterDifferences` — List of differences in parameters
- `Clones` — Number of subsystem clones in a particular clone group
- `BlocksPerClone` — Number of block elements in the clone
- `CloneType` — Whether the clone is a `Similar` or `Exact` clone
- `BlockDifference` — Difference in block value

`CloneList` — List of subsystem clones
array of character vectors

List of subsystem clone names, specified as an array of character vectors.

`ExceptionLog` — Log of exceptions and warnings
array of character vectors

The `ExceptionLog` contains the exceptions and warnings from find clones operation, specified as an array of character vectors.

Examples

Analyze Results After Finding Clones in a Model

1 This example shows how to analyze the results of find clones operation. For an example model, see `ex_clone_detection`.

The find clones function stores the information in `cloneResults` object.

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

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

```
cloneResults.Clones
```

```
 struct with fields:

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

2 To view the summary of `cloneResults`.

```
cloneResults.Clones.Summary
```

```
 struct with fields:

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

```
cloneResults.Clones.CloneGroups
```

```
 1×2 struct array with fields:

    Name
    Summary
    CloneList
```

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

```
 struct with fields:

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

3 To view the summary of first clone group.

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

```
 struct with fields:

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

Version History

Introduced in R2021a

See Also

`Simulink.CloneDetection.Settings` | `Simulink.CloneDetection.ReplacementResults` | `Simulink.CloneDetection.EquivalencyCheckResults`

Simulink.CloneDetection.Settings class

Package: `Simulink.CloneDetection`

Conditions for findClones function

Description

Use objects of the `Simulink.CloneDetection.Settings` class to specify certain conditions when calling the `Simulink.CloneDetection.findClones` function.

Creation

Description

`Simulink.CloneDetection.Settings` creates an object of the `Simulink.CloneDetection.Settings` class.

Properties

`IgnoreSignalName` — Ignore differences in signal names
false or 0 | true or 1

Option to ignore signal name differences while detecting clones, specified as `false` or `true`.

`IgnoreBlockProperty` — Ignore differences in block property
false or 0 | true or 1

Option to ignore block property differences while detecting clones, specified as `false` or `true`.

`ExcludeModelReferences` — Model references exclusion
false or 0 | true or 1

Option to exclude Model Reference blocks from search clone patterns, specified as `false` or `true`.

`ExcludeLibraryLinks` — Exclude library links
false or 0 | true or 1

Option to exclude library links from search clone patterns, specified as `false` or `true`.

`FindClonesRecursivelyInFolders` — Find clones recursively in folders
false or 0 | true or 1

Option to find clones in models from the folders present inside the specified folder, specified as `false` or `true`.

`ParamDifferenceThreshold` — Maximum number of parameter differences allowed to consider pattern as clone
uint32

Number of differences allowed to consider a pattern as a clone, specified as a `uint32` integer.

`ReplaceExactClonesWithSubsystemReference` — Replace exact clones with Subsystem Reference blocks
false or 0 | true or 1

Option to replace exact clones with Subsystem Reference blocks, specified as `false` or `true`.

`Libraries` — Path to libraries
character vector | array of character vectors | Strings

Path to libraries used to replace patterns in the model, specified as a character vector or array of character vectors.

`Folders` — Path to folders
character vector | array of character vectors | Strings

Path to folders that contain the models, specified as a character vector or array of character vectors.

`DetectClonesAcrossModel` — Option to detect clones across model
false or 0 | true or 1

Option to detect clones anywhere across the model, specified as `false` or `true`.

`MinimumRegionSize` — Minimum number of blocks
uint32

Minimum number of blocks in a region required to consider a pattern as a clone, specified as a `uint32` integer.

This property applies only when `DetectClonesAcrossModel` is `true`.

`MinimumCloneGroupSize` — Minimum number of clone pattern occurrences
uint32

Minimum number of clone pattern occurrences required to consider a pattern as a clone group, specified as a `uint32` integer.

`ExcludeInactiveRegions` — Exclude inactive or commented-out regions
false or 0 | true or 1

Option to exclude inactive regions or commented-out regions from search clone patterns, specified as `false` or `true`.

Methods

Public Methods

addLibraries	Add library with subsystems
removeLibraries	Remove library
addFolders	Add folders with models
removeFolders	Remove folder

Examples

Create Settings Object for Clone Detection Operations

This example shows how to create `cloneDetectionSettings` object from `Simulink.CloneDetection.Settings` class for clone detection operation and change the property of `IgnoreSignalName` to `true` for ignoring the signal name differences while detecting clones.

- ```
  cloneDetectionSettings = Simulink.CloneDetection.Settings();
  cloneDetectionSettings.IgnoreSignalName = true
  ```

  ```
  cloneDetectionSettings =

    Settings with properties:
                              IgnoreSignalName: 1
                         IgnoreBlockProperty: 0
                       ExcludeModelReferences: 0
                          ExcludeLibraryLinks: 0
                       SelectedSystemBoundary: ''
                FindClonesRecursivelyInFolders: 1
                     ParamDifferenceThreshold: 50
      ReplaceExactClonesWithSubsystemReference: 0
                                     Libraries: {}
                                       Folders: {}
                      DetectClonesAcrossModel: 0
                       ExcludeInactiveRegions: 0
  ```

Version History

Introduced in R2021a

See Also

`Simulink.CloneDetection.Results` | `Simulink.CloneDetection.ReplacementConfig` | `Simulink.CloneDetection.ReplacementResults`
  ```

addLibraries

Class: `Simulink.CloneDetection.Settings`
Package: `Simulink.CloneDetection`

Add library with subsystems

Syntax

```
cloneDetectionSettings = cloneDetectionSettings.addLibraries(LibraryName)
```

Description

`cloneDetectionSettings = cloneDetectionSettings.addLibraries(LibraryName)` adds library file to the `cloneDetectionSettings` object.

Input Arguments

`LibraryName` — Library file name

character vector

Name of the library file, specified as a character vector.

Output Arguments

`cloneDetectionSettings` — Configuration to find clones

`Simulink.CloneDetection.Settings` object

Clone Detection Settings, specified as a `Simulink.CloneDetection.Settings` object.

Examples

- This example shows how to add a library file to `cloneDetectionSettings` object for matching the library subsystem clones. The library file is `TestLib_1`.

```
libName = 'TestLib_1';

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

```
cloneDetectionSettings =
  Settings with properties:
                          IgnoreSignalName: 0
                       IgnoreBlockProperty: 0
                    ExcludeModelReferences: 0
                       ExcludeLibraryLinks: 0
                    ExcludeInactiveRegions: 0
                  ParamDifferenceThreshold: 50
    ReplaceExactClonesWithSubsystemReference: 0
                                 Libraries: {'C:\Users\Desktop\ex_clone_detection\TestLib_1.slx'}
```

See Also

`Simulink.CloneDetection.Settings` |
`Simulink.CloneDetection.Settings.removeLibraries`

removeLibraries

Class: `Simulink.CloneDetection.Settings`
Package: `Simulink.CloneDetection`

Remove library

Syntax

```
cloneDetectionSettings = cloneDetectionSettings.removeLibraries(LibraryName)
```

Description

`cloneDetectionSettings = cloneDetectionSettings.removeLibraries(LibraryName)` removes library from `cloneDetectionSettings` object.

Input Arguments

`LibraryName` — Library file name

character vector

Name of the library file, specified as a character vector.

Output Arguments

`cloneDetectionSettings` — Configuration to find clones

`Simulink.CloneDetection.Settings` object

Clone Detection Settings, specified as a `Simulink.CloneDetection.Settings` object.

Examples

- This example shows how to remove a library file from `cloneDetectionSettings` object which is already added. The library file is `TestLib_1`.

```
cloneDetectionSettings = cloneDetectionSettings.removeLibraries(libName)
```

```
cloneDetectionSettings =
  Settings with properties:
                            IgnoreSignalName: 0
                         IgnoreBlockProperty: 0
                      ExcludeModelReferences: 0
                         ExcludeLibraryLinks: 0
                      ExcludeInactiveRegions: 0
                    ParamDifferenceThreshold: 50
    ReplaceExactClonesWithSubsystemReference: 0
                                   Libraries: {}
```

See Also

`Simulink.CloneDetection.Settings` |
`Simulink.CloneDetection.Settings.addLibraries`

addFolders

Class: `Simulink.CloneDetection.Settings`
Package: `Simulink.CloneDetection`

Add folders with models

Syntax

```
cloneDetectionSettings = cloneDetectionSettings.addFolders(FolderName)
```

Description

`cloneDetectionSettings = cloneDetectionSettings.addFolders(FolderName)` adds folder path to the `cloneDetectionSettings` object.

Input Arguments

`FolderName` — Folder name

character vector

Name or path of the folder, specified as a character vector.

Output Arguments

`cloneDetectionSettings` — Configuration to find clones

`Simulink.CloneDetection.Settings` object

Clone Detection Settings, specified as a `Simulink.CloneDetection.Settings` object.

Examples

- This example shows how to add a Folder to `cloneDetectionSettings` object for finding the clones from the models inside the folder. The folder name is `Folders1`.

```
FolderName = 'Folders1';

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

```
cloneDetectionSettings =
  Settings with properties:
                            IgnoreSignalName: 0
                       IgnoreBlockProperty: 0
                    ExcludeModelReferences: 0
                       ExcludeLibraryLinks: 0
                    SelectedSystemBoundary: []
              FindClonesRecursivelyInFolders: 1
                    ParamDifferenceThreshold: 50
   ReplaceExactClonesWithSubsystemReference: 0
                                   Libraries: {}
                                     Folders: {'C:\Users\Folders1}
                   DetectClonesAcrossModel: 0
                    ExcludeInactiveRegions: 0
```

See Also

`Simulink.CloneDetection.Settings` |
`Simulink.CloneDetection.Settings.removeFolders`

removeFolders

Class: `Simulink.CloneDetection.Settings`
Package: `Simulink.CloneDetection`

Remove folder

Syntax

```
cloneDetectionSettings = cloneDetectionSettings.removeFolders(FolderName)
```

Description

`cloneDetectionSettings = cloneDetectionSettings.removeFolders(FolderName)` removes folder from `cloneDetectionSettings` object.

Input Arguments

`FolderName` — Folder name

character vector

Name of the folder, specified as a character vector.

Output Arguments

`cloneDetectionSettings` — Configuration to find clones

`Simulink.CloneDetection.Settings` object

Clone Detection Settings, specified as a `Simulink.CloneDetection.Settings` object.

Examples

- This example shows how to remove a folder from `cloneDetectionSettings` object which is already added. The folder name is `Folder1`.

```
FolderName = 'Folder1';
cloneDetectionSettings = cloneDetectionSettings.removeFolders(FolderName)
```

```
cloneDetectionSettings =
  Settings with properties:
                            IgnoreSignalName: 0
                       IgnoreBlockProperty: 0
                    ExcludeModelReferences: 0
                       ExcludeLibraryLinks: 0
                    SelectedSystemBoundary: []
              FindClonesRecursivelyInFolders: 1
                    ParamDifferenceThreshold: 50
   ReplaceExactClonesWithSubsystemReference: 0
                                  Libraries: {}
                                    Folders: {}
                  DetectClonesAcrossModel: 0
                   ExcludeInactiveRegions: 0
```

See Also

`Simulink.CloneDetection.Settings` | `Simulink.CloneDetection.Settings.addFolders`

addCloneToIgnoreList

Class: `Simulink.CloneDetection.ReplacementConfig`
Package: `Simulink.CloneDetection`

Add subsystem to clone replacement ignore list

Syntax

```
cloneReplacementSettings.addCloneToIgnoreList(subsystemName)
```

Description

`cloneReplacementSettings.addCloneToIgnoreList(subsystemName)` adds the specified subsystem name to the clone replacement ignore list associated with the `CloneReplacementSettings` object.

Input Arguments

`subsystemName` — Subsystem name

character vector

Name of the subsystem, specified as a character vector.

Examples

Add Subsystem to Clone Replacement Ignore List

-
```
cloneReplacementConfig = Simulink.CloneDetection.ReplacementConfig();
cloneReplacementConfig.addCloneToIgnoreList('ex_clone_detection_E/Subsystem1');
```

```
cloneReplacementConfig =

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

See Also

`Simulink.CloneDetection.ReplacementConfig` | `Simulink.CloneDetection.ReplacementConfig.removeCloneFromIgnoreList`

removeCloneFromIgnoreList

Class: `Simulink.CloneDetection.ReplacementConfig`
Package: `Simulink.CloneDetection`

Remove subsystem from clone replacement ignore list

Syntax

```
cloneReplacementSettings.removeCloneFromIgnoreList(subsystemName)
```

Description

`cloneReplacementSettings.removeCloneFromIgnoreList(subsystemName)` removes the specified subsystem from the clone replacement ignore list associated with the `CloneReplacementSettings` object.

Input Arguments

`subsystemName` — Subsystem name
character vector

Name of the subsystem, specified as a character vector.

Examples

Remove Subsystem from Clone Replacement Ignore List

- `cloneReplacementConfig.removeCloneFromIgnoreList('ex_clone_detection_E/Subsystem1');`

See Also

`Simulink.CloneDetection.ReplacementConfig` |
`Simulink.CloneDetection.ReplacementConfig.addCloneToIgnoreList`

Simulink.CloneDetection.EquivalencyCheckResults class

Package: `Simulink.CloneDetection`

Results of equivalency check

Description

Use objects of the `Simulink.CloneDetection.EquivalencyCheckResults` class to view the results of an equivalency check.

Creation

Description

The `Simulink.CloneDetection.checkEquivalency` function creates an object of the `Simulink.CloneDetection.EquivalencyCheckResults` class when executed.

Properties

`List` — Equivalency check results
structure

Equivalency check results, specified as a structure.

`IsEquivalencyPassed` — Equivalency results
true | false

Pass result of equivalency check operation, specified as `true` or `false`.

`OriginalModel` — Model used for clone replacement
character array

Model used for clone replacement, specified as a character array.

`UpdatedModel` — Updated model
character array

Updated model with clones replaced, specified as a character array.

Examples

View Results of a Check Equivalency Operation

- ```
  equivalencyCheckResults = Simulink.CloneDetection.checkEquivalency(cloneReplacementResults)

  equivalencyCheckResults.List
  ```
  ```

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

Version History

Introduced in R2021a

See Also

Simulink.CloneDetection.Settings | Simulink.CloneDetection.ReplacementConfig | Simulink.CloneDetection.Results

Advisor.getExclusion

Package: `Advisor`

Get exclusion information for a model or a file

Syntax

```
Advisor.getExclusion('modelname')
Advisor.getExclusion('modelname','Name,Value')
```

Description

`Advisor.getExclusion('modelname')` displays the exclusion data of the model.

`Advisor.getExclusion('modelname','Name,Value')` displays the exclusion data of an entity in the model. This can be a block, a Subsystem or a Stateflow entity.

Note If the `checks` property in the displayed result is **.***, then the model object is excluded from all checks.

Input Arguments

`modelname` — Model name

string | character vector

Model name to display all the exclusions.

Example: Advisor.getExclusion('vdp');

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can only specify a single Name-Value pair argument.

Example: Advisor.getExclusion('vdp', 'BlockType', 'Integrator');

`Block` — Name of the block

string | character vector

This is an optional argument of type **Filter**. You can limit your exclusion results to a particular block in the model using this argument.

Example: Advisor.getExclusion('vdp', 'Block', 'vdp:2');

`BlockType` — Type of the block

string | character vector

This is an optional argument of type **Filter**. You can limit your exclusion results to a particular block type in the model using this argument.

Example: Advisor.getExclusion('vdp', 'BlockType', 'Integrator');

`Subsystem` — Name or SID of Subsystem
string | character vector

This is an optional argument of type **Filter**. You can fetch the exclusions applied to a subsystem in the model using this argument.

Example: Advisor.getExclusion('vdp', 'Subsystem', 'vdp:6');

`Library` — Name or SID of Library
string | character vector

This is an optional argument of type **Filter**. You can fetch the exclusions applied to a Library in the model using this argument.

Example: Advisor.getExclusion('vdp', 'Library', 'mCustomlib:6');

`MaskType` — Mask name applied to a Subsystem
string | character vector

This is an optional argument of type **Filter**. You can fetch the exclusions applied to a Subsystem with a specific mask type in the model using this argument.

Example: Advisor.getExclusion('vdp', 'MaskType', 'gearSystem');

`Stateflow` — Stateflow blocks imported to Simulink
string | character vector

This is an optional argument of type **Filter**. You can fetch the exclusions applied to Stateflow blocks and Subsystems in the model using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:4');

`Chart` — SID of Stateflow Chart
string | character vector

This is an optional argument of type **Filter**. You can fetch the exclusions applied to a Stateflow chart in the model using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`State` — SID of Stateflow State
string | character vector

This is an optional argument of type **Filter**. You can fetch the exclusions applied to a Stateflow State in the model using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`Transition` — SID of Stateflow Transition
string | character vector

This is an optional argument of type **Filter**. You can fetch the exclusions applied to a Stateflow Transition in the model using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`Junction` — SID of Stateflow Junction
string | character vector

This is an optional argument of type **Filter**. You can fetch the exclusions applied to a Stateflow Junction in the model using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`GraphicalFunction` — Graphical function inside Stateflow
string | character vector

This is an optional argument of type **Filter**. You can fetch the exclusions applied to a graphical function in Stateflow using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`MatlabFunction` — MATLAB function inside Stateflow
string | character vector

This is an optional argument of type **Filter**. You can fetch the exclusions applied to a MATLAB function in Stateflow using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`SimulinkFunction` — Simulink function inside Stateflow
string | character vector

This is an optional argument of type **Filter**. You can fetch the exclusions applied to a Simulink function inside Stateflow using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`TruthTable` — Stateflow Truth Table
string | character vector

This is an optional argument of type **Filter**. You can fetch the exclusions applied to a Stateflow Truth Table in the model using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`SimulinkBasedState` — Simulink based state in Stateflow
string | character vector

This is an optional argument of type **Filter**. You can fetch the exclusions applied to a Simulink based state in Stateflow using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

Version History

Introduced in R2021a

See Also

`Advisor.addExclusion` | `Advisor.removeExclusion` | `Advisor.clearExclusion` | `Advisor.saveExclusion` | `Advisor.loadExclusion`

Advisor.addExclusion

Package: `Advisor`

Add exclusions to a model or a file

Syntax

```
Advisor.addExclusion('modelname','Name,Value')
```

Description

`Advisor.addExclusion('modelname','Name,Value')` (the first name-value pair in the syntax should be of **Filter** type argument) adds an exclusion to an entity in the model. This can be a block, a Subsystem or a Stateflow entity.

Input Arguments

`modelname` — Model name

string | character vector

Model name to display all the exclusions.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can add any number of name-value pairs in the arguments provided that **Filter** type name-value pair is used only once.

Example: Advisor.addExclusion('vdp', 'BlockType', 'Integrator', 'checks', {'mathworks.jmaab.jc_0231', 'mathworks.jmaab.jc_0222'}, 'validateChecks', true);

`Block` — Name of the block

string | character vector

This is an optional argument of type **Filter**. You can add exclusions to a particular block in the model using this argument.

Example: Advisor.addExclusion('vdp', 'Block', 'vdp:2');

`BlockType` — Type of the block

string | character vector

This is an optional argument of type **Filter**. You can add exclusions to a particular block type in the model using this argument.

Example: Advisor.addExclusion('vdp', 'BlockType', 'Integrator');

`Subsystem` — Name or SID of Subsystem

string | character vector

This is an optional argument of type **Filter**. You can add exclusions to a subsystem in the model using this argument.

Example: Advisor.addExclusion('vdp', 'Subsystem', 'vdp:6');

`Library` — Name or SID of Library
string | character vector

This is an optional argument of type **Filter**. You can add exclusions to a Library in the model using this argument.

Example: Advisor.addExclusion('vdp', 'Library', 'mCustomlib:6');

`MaskType` — Mask name to a Subsystem
string | character vector

This is an optional argument of type **Filter**. You can add exclusions to a Subsystem with a specific mask type in the model using this argument.

Example: Advisor.addExclusion('vdp', 'MaskType', 'gearSystem');

`Stateflow` — Stateflow blocks imported to Simulink
string | character vector

This is an optional argument of type **Filter**. You can add exclusions to State Transition Table, MessageViewer or any other Stateflow blocks apart from existing **Name-Value** Stateflow filters mentioned on this function page.

Example: Advisor.addExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:4');

`Chart` — SID of Stateflow Chart
string | character vector

This is an optional argument of type **Filter**. You can add exclusions to a Stateflow chart in the model using this argument.

Example: Advisor.addExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`State` — SID of Stateflow State
string | character vector

This is an optional argument of type **Filter**. You can add exclusions to a Stateflow State in the model using this argument.

Example: Advisor.addExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`Transition` — SID of Stateflow Transition
string | character vector

This is an optional argument of type **Filter**. You can add exclusions to a Stateflow Transition in the model using this argument.

Example: Advisor.addExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`Junction` — SID of Stateflow Junction
string | character vector

This is an optional argument of type **Filter**. You can add exclusions to a Stateflow Junction in the model using this argument.

Example: Advisor.addExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`GraphicalFunction` — Graphical function inside Stateflow
string | character vector

This is an optional argument of type **Filter**. You can add exclusions to a graphical function in Stateflow using this argument.

Example: Advisor.addExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`MatlabFunction` — MATLAB function inside Stateflow
string | character vector

This is an optional argument of type **Filter**. You can add exclusions to a MATLAB function in Stateflow using this argument.

Example: Advisor.addExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`SimulinkFunction` — Simulink function inside Stateflow
string | character vector

This is an optional argument of type **Filter**. You can add exclusions to a Simulink function inside Stateflow using this argument.

Example: Advisor.addExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`TruthTable` — Stateflow Truth Table
string | character vector

This is an optional argument of type **Filter**. You can add exclusions to a Stateflow Truth Table in the model using this argument.

Example: Advisor.addExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`SimulinkBasedState` — Simulink based state in Stateflow
string | character vector

This is an optional argument of type **Filter**. You can add exclusions to a Simulink based state in Stateflow using this argument.

Example: Advisor.addExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`checks` — Check IDs to add exclusions
string | character vector

This is an optional argument. Using this argument you can add exclusions only to selected checks.

If this name-value pair is not used in the input arguments, then the model object will be excluded from all the checks. By default, this option is set to all checks.

Example: Advisor.addExclusion('vdp', 'Block', 'vdp:2', 'rationale', 'Block to be removed later', 'checks', {'mathworks.jmaab.jc_0231'});

`validateChecks` — Validates exclusion support
false (default) | true

This is an optional argument. Using this argument you can validate if the selected checks support exclusions.

Example: Advisor.addExclusion('vdp', 'BlockType', 'Integrator', 'checks', {'mathworks.jmaab.jc_0231', 'mathworks.jmaab.jc_0222'}, 'validateChecks', true);

`rationale` — Reason for exclusion
string | character vector

This is an optional argument. Using this argument you can add a rationale (comment) when adding exclusions.

Example: Advisor.addExclusion('vdp', 'Block', 'vdp:2', 'rationale', 'Block to be removed later', 'checks', {'mathworks.jmaab.jc_0231'});

Version History

Introduced in R2021a

See Also

`Advisor.removeExclusion` | `Advisor.clearExclusion` | `Advisor.getExclusion` | `Advisor.saveExclusion` | `Advisor.loadExclusion`

Advisor.removeExclusion

Package: `Advisor`

Removes exclusions for a model or a file

Syntax

```
Advisor.removeExclusion('modelname','Name,Value')
```

Description

`Advisor.removeExclusion('modelname','Name,Value')` removes the exclusions of Filter type in the model.

Input Arguments

`modelname` — Model name

string | character vector

Model name to remove the exclusions.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can only specify a single Name-Value pair argument.

Example: Advisor.removeExclusion('vdp', 'BlockType', 'Integrator');

`Block` — Name of the block

string | character vector

This is an optional argument of type **Filter**. You can remove exclusions for a particular block in the model using this argument.

Example: Advisor.getExclusion('vdp', 'Block', 'vdp:2');

`BlockType` — Type of the block

string | character vector

This is an optional argument of type **Filter**. You can remove exclusions for a particular block type in the model using this argument.

Example: Advisor.getExclusion('vdp', 'BlockType', 'Integrator');

`Subsystem` — Name or SID of Subsystem

string | character vector

This is an optional argument of type **Filter**. You can remove the exclusions applied to a subsystem in the model using this argument.

Example: Advisor.getExclusion('vdp', 'Subsystem', 'vdp:6');

`Library` — Name or SID of Library
string | character vector

This is an optional argument of type **Filter**. You can remove the exclusions applied to a Library in the model using this argument.

Example: Advisor.getExclusion('vdp', 'Library', 'mCustomlib:6');

`MaskType` — Mask name applied to a Subsystem
string | character vector

This is an optional argument of type **Filter**. You can remove the exclusions applied to a Subsystem with a specific mask type in the model using this argument.

Example: Advisor.getExclusion('vdp', 'MaskType', 'gearSystem');

`Stateflow` — Stateflow blocks imported to Simulink
string | character vector

This is an optional argument of type **Filter**. You can remove the exclusions applied to Stateflow blocks and Subsystems in the model using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:4');

`Chart` — SID of Stateflow Chart
string | character vector

This is an optional argument of type **Filter**. You can remove the exclusions applied to a Stateflow chart in the model using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`State` — SID of Stateflow State
string | character vector

This is an optional argument of type **Filter**. You can remove the exclusions applied to a Stateflow State in the model using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`Transition` — SID of Stateflow Transition
string | character vector

This is an optional argument of type **Filter**. You can remove the exclusions applied to a Stateflow Transition in the model using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`Junction` — SID of Stateflow Junction
string | character vector

This is an optional argument of type **Filter**. You can remove the exclusions applied to a Stateflow Junction in the model using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`GraphicalFunction` — Graphical function inside Stateflow
string | character vector

This is an optional argument of type **Filter**. You can remove the exclusions applied to a graphical function in Stateflow using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`MatlabFunction` — MATLAB function inside Stateflow
string | character vector

This is an optional argument of type **Filter**. You can remove the exclusions applied to a MATLAB function in Stateflow using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`SimulinkFunction` — Simulink function inside Stateflow
string | character vector

This is an optional argument of type **Filter**. You can remove the exclusions applied to a Simulink function inside Stateflow using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`TruthTable` — Stateflow Truth Table
string | character vector

This is an optional argument of type **Filter**. You can remove the exclusions applied to a Stateflow Truth Table in the model using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`SimulinkBasedState` — Simulink based state in Stateflow
string | character vector

This is an optional argument of type **Filter**. You can remove the exclusions applied to a Simulink based state in Stateflow using this argument.

Example: Advisor.getExclusion('sldvdemo_cruise_control', 'Stateflow, 'sldvdemo_cruise_control:8');

`Object` — Exclusion object
Object

This is an optional argument. You can remove an exclusion in the model by providing the exclusion object. It can deduce the type and identifier by itself.

```
allExclusions = Advisor.getExclusion('vdp');
Advisor.removeExclusion('vdp', 'object', allExclusions(1));
```

Version History

Introduced in R2021a

See Also

`Advisor.addExclusion` | `Advisor.clearExclusion` | `Advisor.getExclusion` | `Advisor.saveExclusion` | `Advisor.loadExclusion`

Advisor.clearExclusion

Package: `Advisor`

Clear all exclusions from a model or a file

Syntax

```
Advisor.clearExclusion('modelname')
```

Description

`Advisor.clearExclusion('modelname')` clears all the exclusions in the model.

Input Arguments

`modelname` — Model name
string | character vector

Model name to clear all the exclusions.

Version History

Introduced in R2021a

See Also

`Advisor.addExclusion` | `Advisor.removeExclusion` | `Advisor.getExclusion` | `Advisor.saveExclusion` | `Advisor.loadExclusion`

Advisor.saveExclusion

Package: `Advisor`

Save exclusions to the model or an external file

Syntax

```
Advisor.saveExclusion('modelname')
Advisor.saveExclusion('modelname', ' ')
Advisor.saveExclusion('modelname', 'path')
```

Description

`Advisor.saveExclusion('modelname')` saves the exclusions to the default option as mentioned in model parameter 'MAModelFilterFile'.

`Advisor.saveExclusion('modelname', ' ')` (second argument empty) saves the exclusions inside the model.

`Advisor.saveExclusion('modelname', 'path')` saves the exclusions to the selected path.

Input Arguments

`modelname` — Model name
string | character vector

Model name to display all the exclusions.

Advisor.saveExclusion('vdp')

`path` — Path to save the new exclusions.xml file
string | character vector

This is an optional parameter. You can use this parameter to save the exclusions to an external file.

Advisor.saveExclusion('vdp', 'C:\Documents\vdp_exclusion.xml')

Version History

Introduced in R2021a

See Also

`Advisor.addExclusion` | `Advisor.removeExclusion` | `Advisor.clearExclusion` | `Advisor.getExclusion` | `Advisor.loadExclusion`

Advisor.loadExclusion

Package: `Advisor`

Load exclusions to a model or a file

Syntax

```
Advisor.loadExclusion('modelname', 'path')
```

Description

`Advisor.loadExclusion('modelname', 'path')` loads the exclusions from the selected path.

Input Arguments

`modelname` — Model name
string | character vector

Model name to load the exclusions.

`path` — Path to load the exclusions.xml file
string | character vector

You can use this parameter to load the exclusions from an external file.

Advisor.loadExclusion('vdp', 'C:\Documents\vdp_exclusion.xml')

Version History

Introduced in R2021a

See Also

`Advisor.addExclusion` | `Advisor.removeExclusion` | `Advisor.clearExclusion` | `Advisor.getExclusion` | `Advisor.saveExclusion`

Simulink.ModelTransform.CommonSourceInterpolation.Results class

Package: `Simulink.ModelTransform.CommonSourceInterpolation`

Results of search for eligible Interpolation Using Prelookup blocks

Description

Use an object of the `Simulink.ModelTransform.CommonSourceInterpolation.Results` class to analyze the identified results of a search for eligible Interpolation Using Prelookup blocks.

Creation

Description

The `Simulink.ModelTransform.CommonSourceInterpolation.identifyCandidates` function creates an object of the `Simulink.ModelTransform.CommonSourceInterpolation.Results` class when executed. You can use this object as the input argument of the `Simulink.ModelTransform.CommonSourceInterpolation.refactorModel` function.

Note Do not change the properties of a `Simulink.ModelTransform.CommonSourceInterpolation.Results` object in order to avoid errors with model refactoring.

Properties

`Candidates` — Identified blocks information
structure

Identified Interpolation Using Prelookup blocks and Multiport Switch block, specified as a structure.

`InterpolationPorts` — Identified Interpolation Using Prelookup blocks
structure

`Block` and `Port` number, returned as a structure.

`Block` — Block names
array of characters

List of identified block names, specified as array of characters.

`Port` — Port number
uint32

List of port numbers of the identified blocks, specified as a uint32 integer

`SwitchPorts` — Identified Multiport Switch block
structure

`Block` and `Port` number, returned as a structure.

`Block` — Block names
array of characters

List of identified block names, specified as array of characters.

`Port` — Port number
uint32

List of port numbers of the identified blocks, specified as a uint32 integer

Examples

Analyze Results of Search for Eligible Interpolation Using Prelookup Blocks

- This example shows how to analyze the results of a search for eligible Interpolation Using Prelookup blocks. The function stores the information in a variable called `transformResults`.

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

```
transformResults =

  Results with properties:

    Candidates: [1×1 struct]
```

```
 transformResults.Candidates
```

```
  struct with fields:

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

Version History

Introduced in R2021b

See Also

Simulink.ModelTransform.CommonSourceInterpolation.identifyCandidates on page 1-696 | Simulink.ModelTransform.CommonSourceInterpolation.refactorModel on page 1-698 | `Simulink.ModelTransform.CommonSourceInterpolation.RefactorResults`

Simulink.ModelTransform.CommonSourceInterpolation.RefactorResults class

Package: `Simulink.ModelTransform.CommonSourceInterpolation`

Refactored model results

Description

Use an object of the `Simulink.ModelTransform.CommonSourceInterpolation.RefactorResults` class to view the results of a refactored model after the Model Transformer removes redundant Interpolation Using Prelookup blocks.

Creation

Description

The `Simulink.ModelTransform.CommonSourceInterpolation.refactorModel` function creates an object of the `Simulink.ModelTransform.CommonSourceInterpolation.RefactorResults` class.

Properties

`ModelName` — Refactored model name
character array

Name of the refactored model, specified as character array.

`ModelDirectory` — Refactored model directory
character array

Name of the directory that the refactored model is saved, specified as character array.

`TraceabilityInfo` — Map containing the refactored blocks info
container map

List of blocks in the refactored model, specified as container map.

Examples

View Results of a Refactored Model Operation

This example shows how to analyze the results of refactored model operation and store the information in a variable called `refactorResults`.

- `refactorResults = Simulink.ModelTransform.CommonSourceInterpolation.RefactorResults(transformResults)`

```
refactorResults =

  RefactorResults with properties:

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

Version History

Introduced in R2021b

See Also

Simulink.ModelTransform.CommonSourceInterpolation.identifyCandidates on page 1-696 | Simulink.ModelTransform.CommonSourceInterpolation.refactorModel on page 1-698 | `Simulink.ModelTransform.CommonSourceInterpolation.Results`

Simulink.ModelTransform.CommonSourceInterpolation.identifyCandidates

Identify eligible Interpolation Using Prelookup blocks to transform

Syntax

```
transformResults =
Simulink.ModelTransform.CommonSourceInterpolation.identifyCandidates(model)
transformResults =
Simulink.ModelTransform.CommonSourceInterpolation.identifyCandidates(model,
skipLibraryBlocks)
```

Description

`transformResults = Simulink.ModelTransform.CommonSourceInterpolation.identifyCandidates(model)` returns the Interpolation Using Prelookup blocks that the Model Transformer can transform in the specified `model`.

`transformResults = Simulink.ModelTransform.CommonSourceInterpolation.identifyCandidates(model, skipLibraryBlocks)` additionally specifies whether to ignore library blocks.

Examples

Identify Common Source Interpolation Blocks

This example shows how to search a model for eligible Interpolation Using Prelookup blocks and store the results.

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

```
transformResults =

  Results with properties:

    Candidates: [1×1 struct]

transformResults.Candidates =

  struct with fields:

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

Input Arguments

`model` — Model name

character vector

Model name, specified as a character vector.

`skipLibraryBlocks` — Skip library blocks
false or 0 | true or 1

Option to skip library blocks for transformation, specified as a numeric or logical 1 (`true`) or 0 (`false`).

Output Arguments

`transformResults` — Identified Interpolation Using Prelookup blocks
`Simulink.ModelTransform.CommonSourceInterpolation.Results` object

Identified redundant Interpolation Using Prelookup blocks, returned as a `Simulink.ModelTransform.CommonSourceInterpolation.Results` object.

Version History

Introduced in R2021b

See Also

“Improve Code Efficiency by Merging Multiple Interpolation Using Prelookup Blocks” | Simulink.ModelTransform.CommonSourceInterpolation.refactorModel on page 1-698

Simulink.ModelTransform.CommonSourceInterpolation.refactorModel

Replace Interpolation Using Prelookup blocks

Syntax

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

Description

`refactorResults = Simulink.ModelTransform.CommonSourceInterpolation.refactorModel(transformResults)` replaces the specified group of Interpolation Using Prelookup blocks and returns the refactored model.

`refactorResults = Simulink.ModelTransform.CommonSourceInterpolation.refactorModel(transformResults, preferredModelName)` uses the `preferredModelName` to save the refactored model.

Examples

Refactor redundant Interpolation Using Prelookup Blocks

This example shows how to replace the identified Interpolation Using Prelookup blocks in a model and store the refactored model in the variable, `refactorResults`.

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

```
refactorResults =

  RefactorResults with properties:

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

Input Arguments

`transformResults` — Identified Interpolation Using Prelookup blocks

`Simulink.ModelTransform.CommonSourceInterpolation.Results` object

Identified group of redundant Interpolation Using Prelookup blocks, specified as a `Simulink.ModelTransform.CommonSourceInterpolation.Results` object.

`preferredModelName` — Model name to use for refactored model
character vector

Model name to use for the refactored model, specified as a character vector.

Output Arguments

`refactorResults` — Refactored model
`Simulink.ModelTransform.CommonSourceInterpolation.RefactorResults` object

Refactored model, returned as a `Simulink.ModelTransform.CommonSourceInterpolation.RefactorResults` object.

Version History

Introduced in R2021b

See Also

“Improve Code Efficiency by Merging Multiple Interpolation Using Prelookup Blocks” | Simulink.ModelTransform.CommonSourceInterpolation.identifyCandidates on page 1-696

Simulink.CloneDetection.highlightClone

Highlight clone in a Simulink model

Syntax

```
Simulink.CloneDetection.highlightClone(cloneResults, subsystem)
```

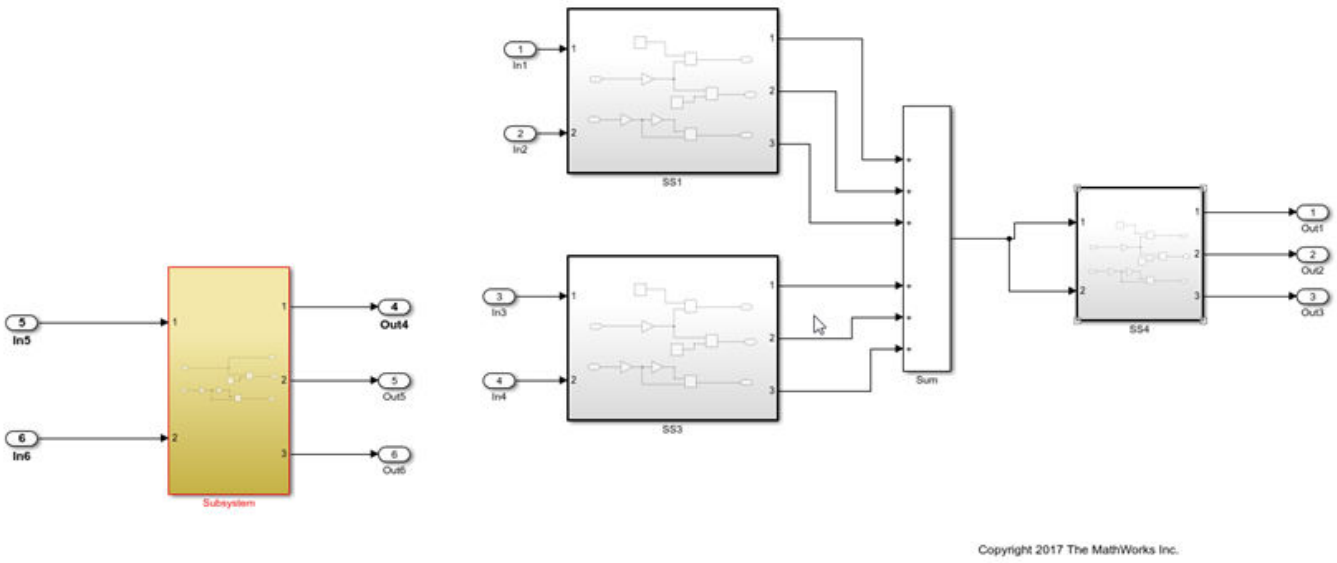
Description

`Simulink.CloneDetection.highlightClone(cloneResults, subsystem)` opens the Simulink model used in `findClones` function and highlights the `subsystem` specified. If the specified model is not open, this command opens it.

Examples

Highlight Clone in a Model

```
Simulink.CloneDetection.highlightClone(cloneResults, 'ex_clone_detection/Subsystem')
```



Input Arguments

`cloneResults` — Results of clone detection operation

`Simulink.CloneDetection.Results` object

Results of a find clones operation, specified as a `Simulink.CloneDetection.Results` object.

`subsystem` — Model and Subsystem name

character vector

Subsystem path in a model, specified as a character vector.

Version History

Introduced in R2021b

See Also

Detect and Replace Clones Programmatically | Simulink.CloneDetection.findClones on page 1-652 | Simulink.CloneDetection.replaceClones on page 1-654

SLSlicerAPI.ParameterDependence class

Package: `SLSlicerAPI`

Class to determine the impact of parameters on a Simulink Model

Description

The SLSlicerAPI.ParameterDependence class displays the dependency between parameters and the model from the MATLAB command line. You can use SLSlicerAPI.ParameterDependence class to:

- Find parameters which impact a particular block.
- Find blocks which are affected by a particular parameter.

Creation

`pd=slicerObj.parameterDependence` creates an object of `SLSlicerAPI.ParameterDependence` for the model used to construct `slicerObj`.

Properties

Public Properties

`slicerObj` — Object of `SLSicerAPI.SLSLicer`

Contains an instance of slicer object used to create pd object, also keeps track of analysis.

Example:

```
slicerObj = slslicer('f14');
```

Data Types: `SLSicerAPI.SLSLicer object`

Methods

Public Methods

parametersAffectingblocks	Finds parameters that impact a Simulink block
blocksAffectedByParameters	Finds the Simulink blocks affected by a parameter

Version History

Introduced in R2021b

See Also

`parametersAffectingBlock` | `blocksAffectedByParameter` | `slslicer` | `SLSlicerAPI.SLSlicer`

blocksAffectedByParameter

Class: `SLSlicerAPI.ParameterDependence`
Package: `SLSlicerAPI`

Finds the Simulink blocks affected by a parameter

Syntax

```
[affectedBlocks,slicerObj] = blocksAffectedByParameter(pd,varUsage)
```

Description

`[affectedBlocks,slicerObj] = blocksAffectedByParameter(pd,varUsage)` lists the blocks that are impacted by the parameter.

Input Arguments

`pd` — Object of `SLSlicerAPI.ParameterDependence` class

Encapsulates the functions which calculate the dependencies between parameters and model.

Example:

```
slicerObj = slslicer(model);
pd = slicerObj.parameterDependence;
```

Data Types: `SLSlicerAPI.ParameterDependence Object`

`varUsage` — Object of `Simulink.VariableUsage`

varUsage represents the parameters to query that affect the Simulink blocks in the model.

Example:

```
varUsage = Simulink.VariableUsage('firstParam','base workspace');
```

Data Types: `Simulink.VariableUsage Object`

Examples

```
[affectedBlocks, slicerObj] = blocksAffectedByParameter(pd, variableUsage);
```

Version History

Introduced in R2021b

See Also

SLSlicerAPI.parameterDependence | parametersAffectingBlock

parametersAffectingBlock

Class: `SLSlicerAPI.ParameterDependence`
Package: `SLSlicerAPI`

Finds parameters that impact a Simulink block

Syntax

```
[params,slicerObj] = parametersAffectingBlock(pd,block)
```

Description

`[params,slicerObj] = parametersAffectingBlock(pd,block)` lists the parameters that are impact a selected Simulink block.

Input Arguments

`pd` — Object of `SLSlicerAPI.ParameterDependence`

Encapsulates the functions which calculate the dependencies between parameters and model.

Example:

```
slicerObj = slslicer(model);
pd = slicerObj.parameterDependence;
```

Data Types: `SLSlicerAPI.ParameterDependence Object`

`block` — Simulink block name/handle/SID

Simulink block for analyzing the parameter dependencies.

Data Types: `handle` | `string` | `character vector`

Examples

```
[params, slicerObj] = parametersAffectingBlock(pd, 'blockA');
```

Version History

Introduced in R2021b

See Also

`SLSlicerAPI.parameterDependence` | `blocksAffectedByParameter`

ModelAdvisor.EdittimeCheck class

Package: `ModelAdvisor`

Abstract base class for creating edit-time checks

Description

Derive classes from the `ModelAdvisor.EdittimeCheck` abstract base class to create custom edit-time checks. To create a custom edit-time check, create a MATLAB class that derives from the abstract base class. Then, create a check definition function that includes a handle to your class and register the custom edit-time check using an `sl_customization` function.

The `ModelAdvisor.EdittimeCheck` class is a `handle` class.

Class Attributes

`Abstract`	`true`
`HandleCompatible`	`true`

For information on class attributes, see “Class Attributes”.

Public Properties

`checkID` — Identifier for check

string

Identifier for the check, specified as a string. Set this property by using the `ModelAdvisor.Check` class that you create as part of your check definition function. The `checkID` property specifies the permanent, unique identifier for the check.

Example: `"advisor.edittime.inoutnamelength"`

Attributes:

`GetAccess`	`protected`
`SetAccess`	`protected`

`TraversalType` — How Model Advisor runs edit-time check

`edittimecheck.TraversalTypes.BLKITER` | `edittimecheck.TraversalTypes.ACTIVEGRAPH`

How, the Model Advisor runs the edit-time check, specified as one of these values:

- `edittimecheck.TraversalTypes.BLKITER` — The check runs on newly added blocks and blocks that you have edited. These edits include changes to block names, types, and parameter values.
- `edittimecheck.TraversalTypes.ACTIVEGRAPH` — The check runs on newly added blocks, blocks that you have edited, *and* other blocks at the same level as an edited or newly added block. Specify this traversal type property if your check must look at the whole subsystem or level of a model that a user is editing. For example, you specify this traversal type to check that a Trigger block is the top-most block in a subsystem because the check must look at other blocks within the subsystem to make this determination.

Example: `eddittimecheck.TraversalTypes.BLKITER`

Attributes:

`GetAccess`	`protected`
`SetAccess`	`protected`

Methods

Protected Methods

blockDiscovered	Specify algorithm for identifying blocks that violate edit-time check
finishedTraversal	Specify edit-time check algorithm to run after blockDiscovered method
fix	Specify algorithm for fixing edit-time check block violations

Examples

Create Custom Edit-time Check for Inport and Outport Blocks

Create a custom edit-time check that checks that Inport and Outport blocks have certain colors depending on their output data types.

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);
```

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 `PortColor` is the class name. Then, publish the check to a new folder in the Model Advisor. For this example, the folder name is **DEMO: Edit-time Checks**. For this example, create a `defineCheck` function and include the code below in it. Save the `defineCheck` function to your working folder.

```
function defineCheck
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');
```

Create a class that derives from the `ModelAdvisor.EdittimeCheck` abstract base class. For this example, create a class file named `PortColor.m`. Copy the code below into the `PortColor.m` file. Then, create a folder named `+MyEditTimeChecks` and save the `PortColor.m` file in that folder. The class must be in a folder that has the same name as the package name.

The `PortColor` class defines three methods: `PortColor`, `blockDiscovered`, and `fix`. The `PortColor` method sets the `CheckId` and `TraversalType` properties. This check has a traversal type of `edittimecheck.TraversalTypes.BLKITER` because the check must check newly added and edited blocks, but it does not have to check for affected blocks in the same subsystem or model

as the edited or newly added blocks. The `blockDiscovered` method contains an algorithm that checks the color of Inport and Outport blocks. The `fix` method updates blocks that do not have correct colors.

```
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
        % Set the Check ID and traversal type.
        function obj=PortColor(checkId)
            obj=obj@ModelAdvisor.EdittimeCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.BLKITER;
        end
        % Specify the edit-time check algorithm using the blockDiscovered method.
        function violation = blockDiscovered(obj, blk)
            violation = [];
            % To check when this check gets called, insert a breakpoint here.
            if strcmp(get_param(blk,'BlockType'),'Inport') || strcmp(get_param(blk,'BlockType'),'Outport')

                dataType = get_param(blk,'OutDataTypeStr');
                currentBgColor = get_param(blk,'BackgroundColor');

                if strcmp(dataType,'boolean')
                        if ~strcmp(currentBgColor, 'orange')
                            % Create a violation object using the ModelAdvisor.ResultDetail class.
                            violation = ModelAdvisor.ResultDetail;
                            ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                            violation.CheckID = obj.checkId;
                            violation.Description = 'Inport/Outport blocks with Boolean outputs should be orange.';
                            violation.title = 'Port Block Color';
                            violation.ViolationType = 'Warning';
                        end
                elseif any(strcmp({'single','double'},dataType))
                        if ~strcmp(currentBgColor, 'green')
                            violation = ModelAdvisor.ResultDetail;
                            ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                            violation.CheckID = obj.checkId;
                            violation.Description = 'Inport/Outport blocks with floating-point outputs should be green.';
                            violation.title = 'Port Block Color';
                            violation.ViolationType = 'Warning';
                        end
                elseif any(strcmp({'uint8','uint16','uint32','int8','int16','int32'}, dataType))
                        if ~strcmp(currentBgColor, 'cyan')
                            violation = ModelAdvisor.ResultDetail;
                            ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                            violation.CheckID = obj.checkId;
                            violation.Description = 'Inport/Outport blocks with integer outputs should be cyan.';
                            violation.title = 'Port Block Color';
                            violation.ViolationType = 'Warning';
                        end
                elseif contains(dataType,'Bus:')
                    if ~strcmp(currentBgColor, 'lightBlue')
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with bus outputs should be light blue.';
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
                 elseif contains(dataType,'Enum:')
                    if ~strcmp(currentBgColor, 'lightBlue')
                        violation = ModelAdvisor.ResultDetail;
```

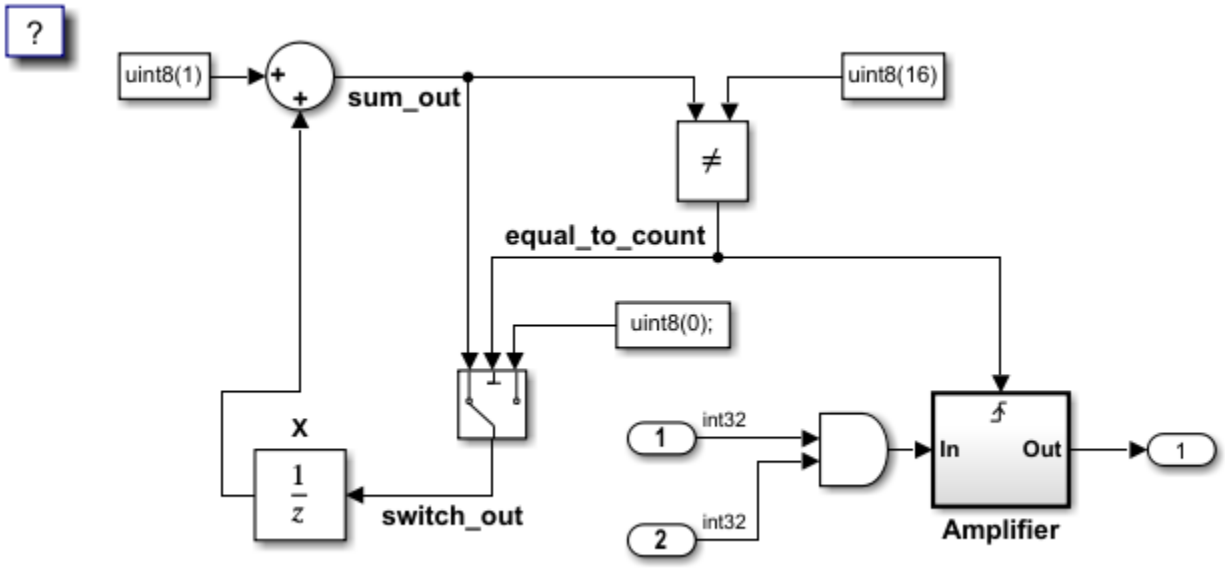
```
                        ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with enumeration outputs should be light blue.';
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
                elseif contains(dataType, 'auto')
                    if ~strcmp(currentBgColor, 'white')
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with auto outputs should be white.';
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
                end
            end
        end


        % Optionally, provide a fix  for the violation object.
        function success = fix(obj, violation)
            success = false;
            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
            success = true;
        end
    end
end
```

Refresh the Model Advisor to update the cache with the new check on the path.

```
Advisor.Manager.refresh_customizations
```

To use the check, copy the `AdvisorCustomizationExample.slx` model to your current working folder.

```
copyfile(fullfile(matlabroot,'examples','slcheck','main',...
'AdvisorCustomizationExample.slx'),'AdvisorCustomizationExample.slx','f');
```



Open the Model Advisor Configuration Editor by clicking the **Modeling** tab and selecting **Model Advisor > Configuration Editor** or by entering this command at the command prompt:

```
Simulink.ModelAdvisor.openConfigUI;
```

Create a custom configuration that consists of the custom edit-time check. Save the configuration as `my_config.json`. Close the Model Advisor Configuration Editor. Set the custom configuration to the `my_config.json` file.

```
ModelAdvisor.setModelConfiguration('AdvisorCustomizationExample', 'my_config.json');
```

Turn on edit-time checking by clicking the **Modeling** tab and selecting **Model Advisor > Edit-Time Checks**. The Configuration Parameters dialog box opens and you select the **Edit-Time Checks** parameter. Alternatively, you can enter this command at the command prompt:

```
edittime.setAdvisorChecking('AdvisorCustomizationExample','on');
```

To view the edit-time warnings, click the blocks 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.

To fix the edit-time warnings, in an edit-time check warning window, click **Fix**.

Version History

Introduced in R2022a

See Also

`ModelAdvisor.Check`

Topics
“Define Custom Model Advisor Checks”
“Define Edit-Time Checks to Comply with Conditions That You Specify with the Model Advisor”
“Define Custom Edit-Time Checks that Fix Issues in Architecture Models”

blockDiscovered

Class: ModelAdvisor.EdittimeCheck
Package: ModelAdvisor

Specify algorithm for identifying blocks that violate edit-time check

Syntax

```
violation = blockDiscovered(obj,blk)
```

Description

`violation = blockDiscovered(obj,blk)` is for specifying an algorithm that checks for blocks, `blk` that violate the edit-time check, `obj` and then returns the blocks that violate the check, `violation`. The scope of blocks that the algorithm checks depends on the scope of changes that you make to your model and the value that you specify for the `TraversalType` property of the edit-time check object.

If the value of the `TraversalType` property is `edittimecheck.TraversalTypes.BLKITER`, the `blockDiscovered` method is called for new and edited blocks.

If the value of the `TraversalType` property is `edittimecheck.TraversalTypes.ACTIVEGRAPH`, the `blockDiscovered` method is called for every block at the same level of the model or subsystem that the user is viewing. The `finishedTraversal` method is also available to be called when the `blockDiscovered` method completes its traversal of blocks within one level of a model or subsystem.

Input Arguments

`obj` — Edit-time check object

object of a class that derives from the `ModelAdvisor.EdittimeCheck` class

Edit-time check, specified as an object of a class that derives from the `ModelAdvisor.Edittime` class

`blk` — Numeric handle of block

`double`

Numeric handle of the block to check for edit-time check violations.

Output Arguments

`violation` — Blocks that violate edit-time check

`ModelAdvisor.ResultDetail` | array of `ModelAdvisor.ResultDetail` objects

Blocks that violate the edit-time check, returned as a `ModelAdvisor.ResultDetail` object or an array of `ModelAdvisor.ResultDetail` objects.

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes.

Examples

Create Custom Edit-time Check for Inport and Outport Blocks

Create a custom edit-time check that checks that Inport and Outport blocks have certain colors depending on their output data types.

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);
```

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 `PortColor` is the class name. Then, publish the check to a new folder in the Model Advisor. For this example, the folder name is **DEMO: Edit-time Checks**. For this example, create a `defineCheck` function and include the code below in it. Save the `defineCheck` function to your working folder.

```
function defineCheck
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');
```

Create a class that derives from the `ModelAdvisor.EdittimeCheck` abstract base class. For this example, create a class file named `PortColor.m`. Copy the code below into the `PortColor.m` file. Then, create a folder named `+MyEditTimeChecks` and save the `PortColor.m` file in that folder. The class must be in a folder that has the same name as the package name.

The `PortColor` class defines three methods: `PortColor`, `blockDiscovered`, and `fix`. The `PortColor` method sets the `CheckId` and `TraversalType` properties. This check has a traversal type of `edittimecheck.TraversalTypes.BLKITER` because the check must check newly added and edited blocks, but it does not have to check for affected blocks in the same subsystem or model as the edited or newly added blocks. The `blockDiscovered` method contains an algorithm that checks the color of Inport and Outport blocks. The `fix` method updates blocks that do not have correct colors.

```
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
    % Set the Check ID and traversal type.
    function obj=PortColor(checkId)
        obj=obj@ModelAdvisor.EdittimeCheck(checkId);
        obj.traversalType = edittimecheck.TraversalTypes.BLKITER;
    end
    % Specify the edit-time check algorithm using the blockDiscovered method.
    function violation = blockDiscovered(obj, blk)
        violation = [];
        % To check when this check gets called, insert a breakpoint here.
        if strcmp(get_param(blk,'BlockType'),'Inport') || strcmp(get_param(blk,'BlockType'),'Outport')

            dataType = get_param(blk,'OutDataTypeStr');
            currentBgColor = get_param(blk,'BackgroundColor');

            if strcmp(dataType,'boolean')
                    if ~strcmp(currentBgColor, 'orange')
                        % Create a violation object using the ModelAdvisor.ResultDetail class.
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with Boolean outputs should be orange.';
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
            elseif any(strcmp({'single','double'},dataType))
                    if ~strcmp(currentBgColor, 'green')
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with floating-point outputs should be green.';
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
            elseif any(strcmp({'uint8','uint16','uint32','int8','int16','int32'}, dataType))
                    if ~strcmp(currentBgColor, 'cyan')
                        violation = ModelAdvisor.ResultDetail;
                        ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                        violation.CheckID = obj.checkId;
                        violation.Description = 'Inport/Outport blocks with integer outputs should be cyan.';
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
            elseif contains(dataType,'Bus:')
                if ~strcmp(currentBgColor, 'lightBlue')
                    violation = ModelAdvisor.ResultDetail;
                    ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                    violation.CheckID = obj.checkId;
                    violation.Description = 'Inport/Outport blocks with bus outputs should be light blue.';
                    violation.title = 'Port Block Color';
                    violation.ViolationType = 'Warning';
                end
             elseif contains(dataType,'Enum:')
                if ~strcmp(currentBgColor, 'lightBlue')
                    violation = ModelAdvisor.ResultDetail;
                    ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                    violation.CheckID = obj.checkId;
                    violation.Description = 'Inport/Outport blocks with enumeration outputs should be light blue.';
                    violation.title = 'Port Block Color';
                    violation.ViolationType = 'Warning';
                end
            elseif contains(dataType, 'auto')
                if ~strcmp(currentBgColor, 'white')
                    violation = ModelAdvisor.ResultDetail;
                    ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                    violation.CheckID = obj.checkId;
```

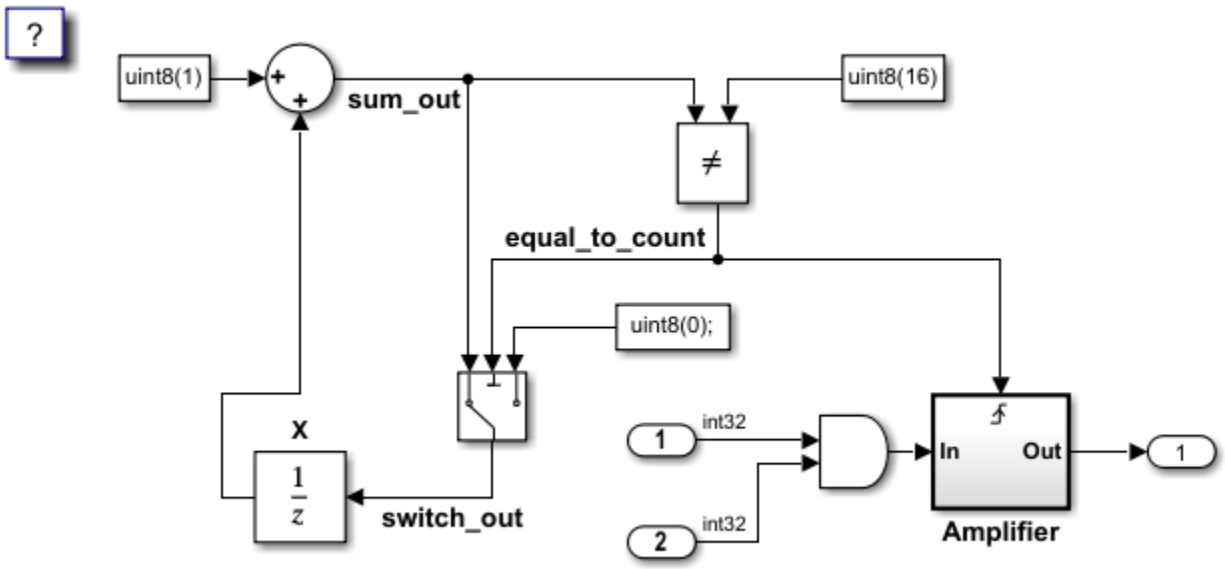
```
                        violation.Description = 'Inport/Outport blocks with auto outputs should be white.';
                        violation.title = 'Port Block Color';
                        violation.ViolationType = 'Warning';
                    end
                end
            end
        end


        % Optionally, provide a fix  for the violation object.
        function success = fix(obj, violation)
            success = false;
            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
            success = true;
        end
    end
end
```

Refresh the Model Advisor to update the cache with the new check on the path.

`Advisor.Manager.refresh_customizations`

To use the check, copy the `AdvisorCustomizationExample.slx` model to your current working folder.

```
copyfile(fullfile(matlabroot,'examples','slcheck','main',...
'AdvisorCustomizationExample.slx'),'AdvisorCustomizationExample.slx','f');
```



Open the Model Advisor Configuration Editor by clicking the **Modeling** tab and selecting **Model Advisor > Configuration Editor** or by entering this command at the command prompt:

```
Simulink.ModelAdvisor.openConfigUI;
```

Create a custom configuration that consists of the custom edit-time check. Save the configuration as `my_config.json`. Close the Model Advisor Configuration Editor. Set the custom configuration to the `my_config.json` file.

```
ModelAdvisor.setModelConfiguration('AdvisorCustomizationExample', 'my_config.json');
```

Turn on edit-time checking by clicking the **Modeling** tab and selecting **Model Advisor > Edit-Time Checks**. The Configuration Parameters dialog box opens and you select the **Edit-Time Checks** parameter. Alternatively, you can enter this command at the command prompt:

```
edittime.setAdvisorChecking('AdvisorCustomizationExample','on');
```

To view the edit-time warnings, click the blocks 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.

To fix the edit-time warnings, in an edit-time check warning window, click **Fix**.

Version History

Introduced in R2022a

See Also

`ModelAdvisor.Check`

Topics

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

finishedTraversal

Class: `ModelAdvisor.EdittimeCheck`
Package: `ModelAdvisor`

Specify edit-time check algorithm to run after `blockDiscovered` method

Syntax

```
violation = finishedTraversal(obj)
```

Description

`violation = finishedTraversal(obj)` is for specifying an algorithm that gets called after the `blockDiscovered` method traverses the blocks at the same level of the model or subsystem that the user is editing. Use this method when you specify a value of `edittimecheck.TraversalTypes.ACTIVEGRAPH` for the `TraversalType` property for the edit-time check object, `obj`. The method returns the blocks that violate your edit-time check algorithm, `violation`.

Use the `finishedTraversal` method for specifying what the edit-time check should do with the data it collects as part of the `blockDiscovered` method. For example, suppose you define an edit-time check that flags subsystems with more than 20 blocks. In this scenario, the `blockDiscovered` method traverses each block in the subsystem and counts the number of blocks. The `finishedTraversal` method is then called and if the subsystem contains more than 20 blocks, it creates an edit-time check violation.

Input Arguments

`obj` — Edit-time check object

object of a class that derives from the `ModelAdvisor.EdittimeCheck` class

Edit-time check object, specified as an object of a class that derives from the `ModelAdvisor.EdittimeCheck` class.

Output Arguments

`violation` — Model Advisor result detail object

`ModelAdvisor.ResultDetail` | array of `ModelAdvisor.ResultDetail` objects

Blocks that violate the edit-time check, returned as a `ModelAdvisor.ResultDetail` object or an array of `ModelAdvisor.ResultDetail` objects.

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes.

Examples

Create Custom Edit-time Check to Check the Position of a Trigger Block

Create a custom edit-time check that checks the position of a Trigger block within a subsystem. This check requires the `finishedTraversal` method because it must specify violation information after it checks the position of blocks in a subsystem.

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);
```

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 `TriggerBlockPosition` is the class name. Then, publish the check to a new folder in the Model Advisor. For this example, the folder name is **DEMO: Edit-time Checks**. For this example, create a `defineCheck` function and include the code below in it. Save the `defineCheck` function to your working folder.

```
function defineCheck
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');
```

Create a class that derives from the `ModelAdvisor.EdittimeCheck` abstract base class. For this example, create a class file named `TriggerBlockPosition.m`. Copy the code below into the `TriggerBlockPosition.m` file. Then, create a folder named `+MyEditTimeChecks` and save the `TriggerBlockPosition.m` file in that folder. The class must be in a folder that has the same name as the package name.

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.

```
classdef TriggerBlockPosition < ModelAdvisor.EdittimeCheck
    properties
        TriggerBlock = [];
        position = [];
    end

 methods
        % Set Check ID and traversal type.
        function obj=TriggerBlockPosition(checkId)
            obj=obj@ModelAdvisor.EdittimeCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.ACTIVEGRAPH;
        end
```

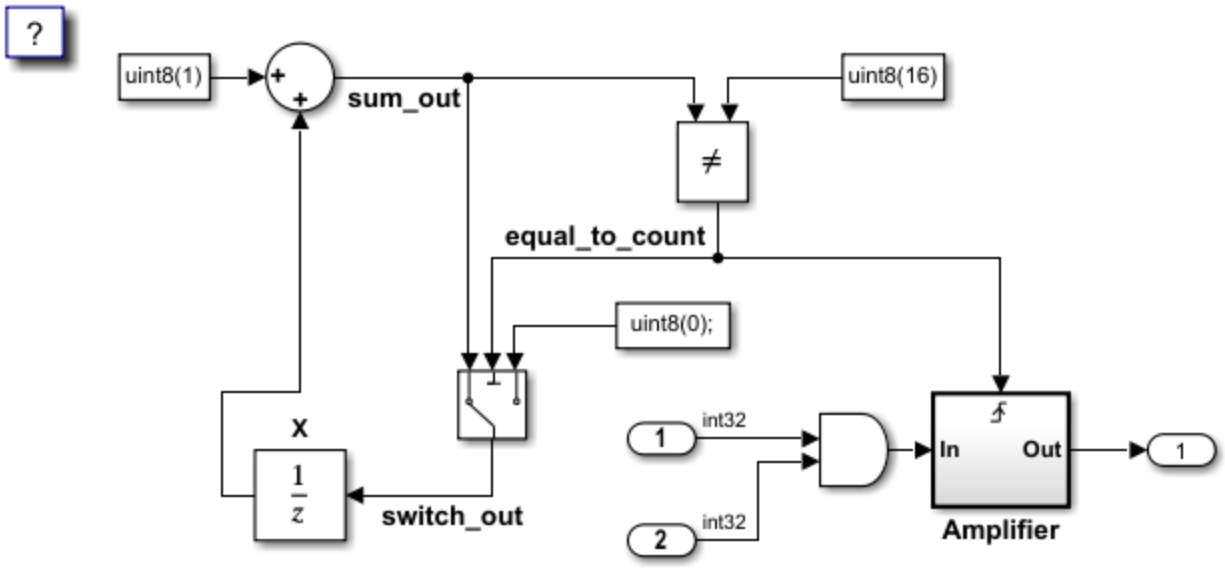
```
        function violation = blockDiscovered(obj, blk)
            violation = [];
            if strcmp(get_param(blk,'BlockType'),'TriggerPort')
                obj.TriggerBlock = blk;
            else
                h = get_param(blk,'Position');
                obj.position = [obj.position, 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
```

Refresh the Model Advisor to update the cache with the new check on the path.

```
Advisor.Manager.refresh_customizations
```

To use the check, copy the `AdvisorCustomizationExample.slx` model to your current working folder.

```
copyfile(fullfile(matlabroot,'examples','slcheck','main',...
'AdvisorCustomizationExample.slx'),'AdvisorCustomizationExample.slx','f');
```



Open the Model Advisor Configuration Editor by clicking the **Modeling** tab and selecting **Model Advisor > Configuration Editor** or by entering this command at the command prompt:

```
Simulink.ModelAdvisor.openConfigUI;
```

Create a custom configuration consisting of the custom edit-time check. Save the configuration as `my_config2.json`. Close the Model Advisor Configuration Editor. Set the custom configuration to the `my_config2.json` file.

```
ModelAdvisor.setModelConfiguration('AdvisorCustomizationExample', ''my_config2.json');
```

Turn on edit-time checking by clicking the **Modeling** tab and selecting **Model Advisor > Edit Time** checks. After 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');
```

To view the edit-time warnings, click the blocks highlighted in yellow.

In the **Amplifier** subsystem, the Trigger block does not produce an edit-time warning because it is the top-most block in the subsystem. If you move the Trigger block below another block, the produces an edit-time warning.

Version History

Introduced in R2022a

See Also

`ModelAdvisor.Check`

Topics

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

fix

Class: `ModelAdvisor.EdittimeCheck`
Package: `ModelAdvisor`

Specify algorithm for fixing edit-time check block violations

Syntax

```
success = fix(obj,violation)
```

Description

`success = fix(obj,violation)` is for specifying an algorithm that fixes each block `violation` that violates the custom edit-time check, `obj`. The blocks that violate the edit-time check are identified as part of the analysis done by the `blockDiscovered` and `finishedTraversal` methods.

Input Arguments

`obj` — Edit-time check object

object of a class that derives from the `ModelAdvisor.EdittimeCheck` class

Edit-time check object for which you want to provide a fix for block violations, specified as an object of a class that derives from the `ModelAdvisor.EdittimeCheck` class.

`violation` — Blocks that violate edit-time check

`ModelAdvisor.ResultDetail` | array of `ModelAdvisor.ResultDetail` objects

Blocks that violate the edit-time check, specified as a `ModelAdvisor.ResultDetail` object or an array of `ModelAdvisor.ResultDetail` objects.

Output Arguments

`success` — Whether a fix is successful

`true` | false

At the beginning of your algorithm, to indicate that a fix has not yet been applied, specify a value of `false`. At the end, to indicate that the fix has been successfully applied, specify a value of `true`.

Attributes

`Access`	`protected`

To learn about attributes of methods, see Method Attributes.

Examples

Create Custom Edit-time Check for Inport and Outport Blocks

Create a custom edit-time check that checks that Inport and Outport blocks have certain colors depending on their output data types.

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);
```

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 `PortColor` is the class name. Then, publish the check to a new folder in the Model Advisor. For this example, the folder name is **DEMO: Edit-time Checks**. For this example, create a `defineCheck` function and include the code below in it. Save the `defineCheck` function to your working folder.

```
function defineCheck
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');
```

Create a class that derives from the `ModelAdvisor.EdittimeCheck` abstract base class. For this example, create a class file named `PortColor.m`. Copy the code below into the `PortColor.m` file. Then, create a folder named `+MyEditTimeChecks` and save the `PortColor.m` file in that folder. The class must be in a folder that has the same name as the package name.

The `PortColor` class defines three methods: `PortColor`, `blockDiscovered`, and `fix`. The `PortColor` method sets the `CheckId` and `TraversalType` properties. This check has a traversal type of `edittimecheck.TraversalTypes.BLKITER` because the check must check newly added and edited blocks, but it does not have to check for affected blocks in the same subsystem or model as the edited or newly added blocks. The `blockDiscovered` method contains an algorithm that checks the color of Inport and Outport blocks. The `fix` method updates blocks that do not have correct colors.

```
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
        % Set the Check ID and traversal type.
        function obj=PortColor(checkId)
            obj=obj@ModelAdvisor.EdittimeCheck(checkId);
            obj.traversalType = edittimecheck.TraversalTypes.BLKITER;
        end
        % Specify the edit-time check algorithm using the blockDiscovered method.
```

```
function violation = blockDiscovered(obj, blk)
    violation = [];
    % To check when this check gets called, insert a breakpoint here.
    if strcmp(get_param(blk,'BlockType'),'Inport') || strcmp(get_param(blk,'BlockType'),'Outport')

        dataType = get_param(blk,'OutDataTypeStr');
        currentBgColor = get_param(blk,'BackgroundColor');

        if strcmp(dataType,'boolean')
                if ~strcmp(currentBgColor, 'orange')
                    % Create a violation object using the ModelAdvisor.ResultDetail class.
                    violation = ModelAdvisor.ResultDetail;
                    ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                    violation.CheckID = obj.checkId;
                    violation.Description = 'Inport/Outport blocks with Boolean outputs should be orange.';
                    violation.title = 'Port Block Color';
                    violation.ViolationType = 'Warning';
                end
        elseif any(strcmp({'single','double'},dataType))
                if ~strcmp(currentBgColor, 'green')
                    violation = ModelAdvisor.ResultDetail;
                    ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                    violation.CheckID = obj.checkId;
                    violation.Description = 'Inport/Outport blocks with floating-point outputs should be green.';
                    violation.title = 'Port Block Color';
                    violation.ViolationType = 'Warning';
                end
        elseif any(strcmp({'uint8','uint16','uint32','int8','int16','int32'}, dataType))
                if ~strcmp(currentBgColor, 'cyan')
                    violation = ModelAdvisor.ResultDetail;
                    ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                    violation.CheckID = obj.checkId;
                    violation.Description = 'Inport/Outport blocks with integer outputs should be cyan.';
                    violation.title = 'Port Block Color';
                    violation.ViolationType = 'Warning';
                end
        elseif contains(dataType,'Bus:')
            if ~strcmp(currentBgColor, 'lightBlue')
                violation = ModelAdvisor.ResultDetail;
                ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                violation.CheckID = obj.checkId;
                violation.Description = 'Inport/Outport blocks with bus outputs should be light blue.';
                violation.title = 'Port Block Color';
                violation.ViolationType = 'Warning';
            end
         elseif contains(dataType,'Enum:')
            if ~strcmp(currentBgColor, 'lightBlue')
                violation = ModelAdvisor.ResultDetail;
                ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                violation.CheckID = obj.checkId;
                violation.Description = 'Inport/Outport blocks with enumeration outputs should be light blue.';
                violation.title = 'Port Block Color';
                violation.ViolationType = 'Warning';
            end
        elseif contains(dataType, 'auto')
            if ~strcmp(currentBgColor, 'white')
                violation = ModelAdvisor.ResultDetail;
                ModelAdvisor.ResultDetail.setData(violation,'SID',Simulink.ID.getSID(blk));
                violation.CheckID = obj.checkId;
                violation.Description = 'Inport/Outport blocks with auto outputs should be white.';
                violation.title = 'Port Block Color';
                violation.ViolationType = 'Warning';
            end
        end
    end
end


% Optionally, provide a fix  for the violation object.
function success = fix(obj, violation)
    success = false;
    dataType = get_param(violation.Data,'OutDataTypeStr');
```

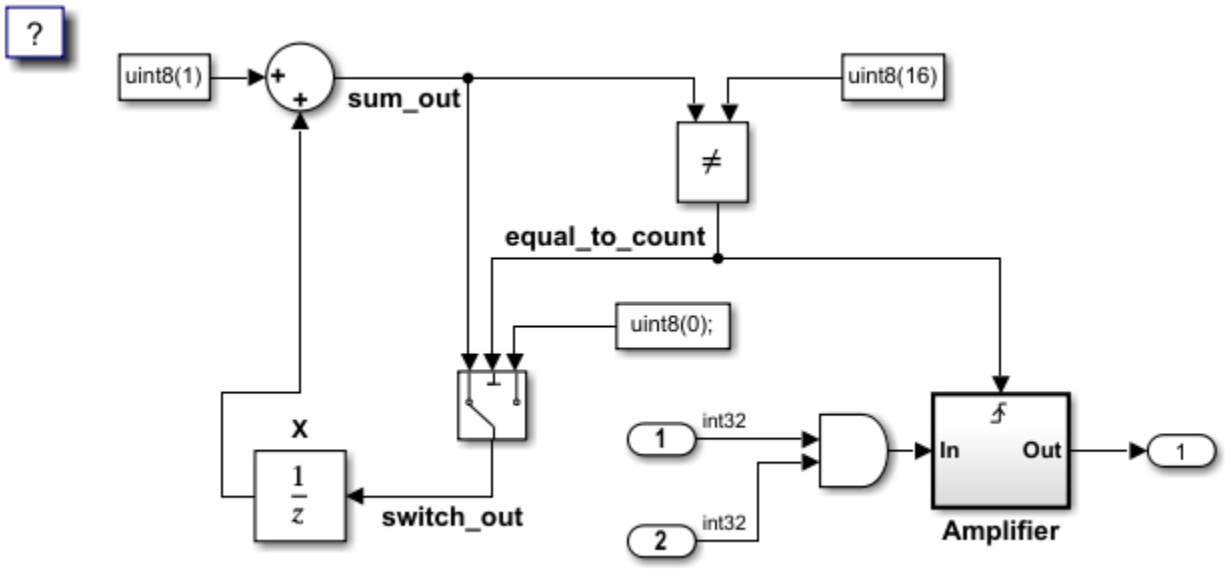
```
            if strcmp(dataType,'boolean')
                set_param(violation.Data,'BackgroundColor','orange');
            elseif any(strcmp({'single','double'},dataType))
                set_param(violation.Data,'BackgroundColor','green');
            elseif any(strcmp({'uint8','uint16','uint32','int8','int16','int32'}, 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
            success = true;
        end
    end
end
```

Refresh the Model Advisor to update the cache with the new check on the path.

```
Advisor.Manager.refresh_customizations
```

To use the check, copy the `AdvisorCustomizationExample.slx` model to your current working folder.

```
copyfile(fullfile(matlabroot,'examples','slcheck','main',...
'AdvisorCustomizationExample.slx'),'AdvisorCustomizationExample.slx','f');
```



Open the Model Advisor Configuration Editor by clicking the **Modeling** tab and selecting **Model Advisor > Configuration Editor** or by entering this command at the command prompt:

```
Simulink.ModelAdvisor.openConfigUI;
```

Create a custom configuration that consists of the custom edit-time check. Save the configuration as `my_config.json`. Close the Model Advisor Configuration Editor. Set the custom configuration to the `my_config.json` file.

```
ModelAdvisor.setModelConfiguration('AdvisorCustomizationExample', 'my_config.json');
```

Turn on edit-time checking by clicking the **Modeling** tab and selecting **Model Advisor > Edit-Time Checks**. The Configuration Parameters dialog box opens and you select the **Edit-Time Checks** parameter. Alternatively, you can enter this command at the command prompt:

```
edittime.setAdvisorChecking('AdvisorCustomizationExample','on');
```

To view the edit-time warnings, click the blocks 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.

To fix the edit-time warnings, in an edit-time check warning window, click **Fix**.

Version History

Introduced in R2022a

See Also

`ModelAdvisor.Check`

Topics

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

slreq.refreshSourceArtifactPath

Refresh artifact path information for Requirements Management Interface link set with migration error in Model Testing Dashboard

Syntax

```
slreq.refreshSourceArtifactPath(reqLinkSetPath)
```

Description

`slreq.refreshSourceArtifactPath(reqLinkSetPath)` refreshes the source artifact path information in a link set, `reqLinkSetPath`, that was created with the legacy Requirements Management Interface (RMI). If you create a requirement link set using the RMI, the link files may contain relative paths, which the Model Testing Dashboard cannot analyze. When the dashboard cannot analyze the relative paths in the requirement link set, the dashboard returns a migration error during artifact analysis. `slreq.refreshSourceArtifactPath` updates the link set to use absolute file paths and resolve the migration error.

Examples

Refresh the Path of a Requirement Link Set

Refresh the artifact paths in a link set created in the Requirements Management Interface.

Suppose that the requirements link set `'legacyReqLinks.slmx'` is in the directory `'C:\myProject\'` and was created in the legacy Requirements Management Interface (RMI). The Model Testing Dashboard cannot analyze this requirement link set.

Refresh the artifact paths.

```
slreq.refreshSourceArtifactPath('C:\myProject\legacyReqLinks.slmx')
```

Input Arguments

`reqLinkSetPath` — File path to requirement link set

character vector | string array

Path to requirement link set, specified as a character vector or string array.

Example: 'C:\myProject\legacyReqLinks.slmx'

Version History

Introduced in R2022a

See Also

sltest.testmanager.refreshTestRevisions

Refresh revision numbers for test cases and test suites in test file to track changes in Model Testing Dashboard

Syntax

```
sltest.testmanager.refreshTestRevisions(filename)
```

Description

`sltest.testmanager.refreshTestRevisions(filename)` refreshes the revision numbers for test cases and test suites in the `sltest.testmanager.TestFile` specified by `filename`. The Model Testing Dashboard is unable to isolate stale results to specific test cases or test suites if individual test cases, test suites, or test iterations were saved in R2021b or earlier. When you call `sltest.testmanager.refreshTestRevisions` on a test file, the function saves revision numbers for the test cases and test suites in the test file. If the test file is already open and dirty, the function updates the revision numbers, but does not save the test file.

Examples

Refresh Revision Numbers for Test Cases and Test Suites in Test File

Suppose you have a test file, `test_file.mldatx`, that contains test cases and test suites that do not have revision numbers. Use `sltest.testmanager.refreshTestRevisions` to save revision numbers for the test cases and test suites.

Call `sltest.testmanager.refreshTestRevisions` on the test file `test_file.mldatx`.

```
sltest.testmanager.refreshTestRevisions("test_file.mldatx");
```

Now the Model Testing Dashboard can isolate stale results to specific test cases or test suites inside the test file `test_file.mldatx`.

Input Arguments

`filename` — Filename of test file

string array

Filename of a test file, specified as a string.

Example: `"test_file.mldatx"`

Data Types: `string`

Version History

Introduced in R2022b

See Also

Topics
“Resolve Missing Artifacts, Links, and Results in the Model Testing Dashboard”

Model Advisor Checks

Simulink Check Checks

In this section...

Simulink Check Checks

Simulink Check checks facilitate designing and troubleshooting models from which code is generated for applications that must meet safety or mission-critical requirements and modeling guidelines.

For descriptions of the modeling standards checks, see

- “DO-178C/DO-331 Checks” on page 2-4
- “IEC 61508, IEC 62304, ISO 26262, ISO 25119, and EN 50128/EN 50657 Checks” on page 2-105
- “AUTOSAR Blockset Checks” (AUTOSAR Blockset)
- “Model Advisor Checks for MAB and JMAAB Compliance” on page 2-109
- “MISRA C:2012 Checks” on page 2-268
- “Secure Coding Checks for CERT C, CWE, and ISO/IEC TS 17961 Standards” on page 2-283

See Also

- “Run Model Advisor Checks and Review Results”

Requirements Toolbox Checks

Requirements Toolbox checks facilitate linking between requirements documentation and your model .

For descriptions of the requirements consistency checks, see “Requirements Consistency Checks” (Requirements Toolbox).

See Also

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

Modeling Standards Checks

Modeling standards checks facilitate designing and troubleshooting models from which code is generated for applications that must meet safety or mission-critical requirements or the global MathWorks Advisory Board (MAB) modeling guidelines.

A Simulink Check license is required to execute these MAB checks. Where applicable, additional license requirements are identified in the check-specific documentation.

For descriptions of the Model Advisor checks that verify compliance to the modeling standards, see:

- “DO-178C/DO-331 Checks” on page 2-4
- “IEC 61508, IEC 62304, ISO 26262, ISO 25119, and EN 50128/EN 50657 Checks” on page 2-105
- “DO-254 Checks” on page 2-267
- “Model Advisor Checks for MAB and JMAAB Compliance” on page 2-109

See Also

- “Run Model Advisor Checks and Review Results”

DO-178C/DO-331 Checks

In this section...

DO-178C/DO-331 Checks

DO-178C/DO-331 checks facilitate designing and troubleshooting models from which code is generated for applications that must meet safety or mission-critical requirements.

The Model Advisor performs a checkout of the Simulink Check license when you run the DO-178C/DO-331 checks.

These checks are qualified by the DO Qualification Kit for use in projects involving the DO-178 standard and related standards.

See Also

- “Run Model Advisor Checks and Review Results”
- “Qualified Model Advisor Checks” (DO Qualification Kit)

Display model version information

Check ID: `mathworks.do178.MdlChecksum`

Display model version information in your report.

Description

This check displays the following information for the current model:

- Version number
- Author
- Date
- Model checksum

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Could not retrieve model version and checksum information.	This summary is provided for your information. No action is required.

Capabilities and Limitations

- Does not run on library models.

- Does not allow exclusions of blocks or charts.

See Also

- “Reports for Code Generation” (Simulink Coder)
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178C Software Considerations in Airborne Systems and Equipment Certification and related standards

High Integrity System Modeling Checks

In this section...

In this section...

High-Integrity Systems Modeling Checks

High-Integrity Systems Modeling checks facilitate designing and troubleshooting models, subsystems, and the corresponding generated code for applications to comply with High-Integrity Systems Modeling Guidelines.

The Model Advisor performs a checkout of the Simulink Check license when you run the High-Integrity Systems Modeling checks.

High-Integrity Systems Modeling checks are classified in to the following groups:

Simulink

High-Integrity Systems Modelling checks for Simulink blocks and components.

For more information, see “Model Checks for High Integrity Systems Modeling”.

MATLAB

High-Integrity Systems Modelling checks for MATLAB code.

For more information, see “Model Checks for High Integrity Systems Modeling”.

Configuration

High-Integrity Systems Modelling checks for Configuration settings.

For more information, see “Model Checks for High Integrity Systems Modeling”.

Naming

High-Integrity Systems Modelling checks for Naming conventions.

For more information, see “Model Checks for High Integrity Systems Modeling”.

Requirements

High-Integrity Systems Modelling checks for Requirements Toolbox traceability.

For more information, see “Model Checks for High Integrity Systems Modeling”.

Code

High-Integrity Systems Modelling checks for generated code.

For more information, see “Model Checks for High Integrity Systems Modeling”.

Stateflow

High-Integrity Systems Modelling checks for Stateflow charts and components.

For more information, see “Model Checks for High Integrity Systems Modeling”.

Split Checks for High Integrity Systems Modeling

From R2018b and later, the following checks are not recommended for use. These checks are split into multiple checks that focus on a single action or operation. For more information, see the Split and New Checks table below.

Old Check Title	Split Check Titles
Check usage of Math Operations blocks	Check usage of Abs blocks

Old Check Title	Split Check Titles
	Check usage of Math Function blocks (rem and reciprocal functions)
	Check usage of Math Function blocks (log and log10 functions)
	Check usage of Assignment blocks
Check usage of Logic and Bit Operations blocks	Check for Relational Operator blocks that equate floating-point types
	Check usage of Relational Operator blocks
	Check usage of Logical Operator blocks
Check usage of Ports and Subsystems blocks	Check usage of While Iterator blocks
	Check sample time-dependent blocks
	Check usage of For Iterator blocks
	Check usage of If blocks and If Action Subsystem blocks
	Check usage Switch Case blocks and Switch Case Action Subsystem blocks
Check safety-related code generation settings	Check safety-related code generation settings for comments
	Check safety-related code generation interface settings
	Check safety-related code generation settings for code style
	Check safety-related code generation symbols settings
Check usage of Stateflow constructs	Check Stateflow charts for ordering of states and transitions
	Check Stateflow debugging options
	Check Stateflow charts for uniquely defined data objects
Check safety-related optimization settings	Check safety-related optimization settings for logic signals
	Check safety-related block reduction optimization settings
	Check safety-related optimization settings for application lifespan
	Check safety-related optimization settings for data initialization
	Check safety-related optimization settings for data type conversions
	Check safety-related optimization settings for division arithmetic exceptions

Check usage of standardized MATLAB function headers

Check ID: `mathworks.hism.himl_0001`

Description

This check inspects all MATLAB functions in the model, local functions, and referenced MATLAB files for standardized function headers and checks for these details:

- Function name
- Function description
- Description of input variables
- Description of output variables

Following is an example of how to define function headers:

%<Function Name> - Description of the function

%<Input variable 1> - Description of input variable 1

%<Input variable 2> - Description of input variable 2

%<Output variable 1> - Description of output variable 1

Available with Simulink Check.

Check Parameterization

You can set the following configuration through Model Advisor Configuration Editor.

Select the MATLAB Function header format type, and the custom header format and configure the description tags from the **Header format type**and **Custom header format**input parameters. The format for the **Custom header format** should be of a string type with comma separated tags.

By default, the **Header format type** is set to **Default** and the **Custom header format** parameter is **Description, Input,Output**.

For example, considering the default values (Description, Input, Output), each of the layer in the model should have the description format as following:

Description: <model description>

Input: <input information>

Output: <output information>

Results and Recommended Actions

Condition	Recommended Action
MATLAB functions use nonstandard function headers.	Consider adding a function header to the functions according to these guidelines: • Must be a valid MATLAB comment. • Must immediately follow the function signature. • Must have a "Function Description" section. • Must have an "Inputs Description" section. • Must have an "Outputs Description" section.

Capabilities and Limitations

- This check only analyzes the functions that are directly referenced by the Simulink model.
- You can configure the check to run on referenced MATLAB files using the input parameter **Check .m files referenced in the model** in the Configuration Editor. By default this parameter is selected.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- himl_0001: Usage of standardized MATLAB function headers

Check for MATLAB Function interfaces with inherited properties

Check ID: `mathworks.hism.himl_0002`

Identify MATLAB Functions that have inputs, outputs or parameters with inherited complexity or data type properties.

Description

The check identifies MATLAB Functions with inherited complexity or data type properties. A results table provides links to MATLAB Functions that do not pass the check, along with conditions triggering the warning.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
MATLAB Functions have inherited interfaces.	Explicitly define complexity and data type properties for inports, outports, and parameters of MATLAB Functions identified in the results. If applicable, using the **MATLAB Function Block Editor**, make the following modifications in the Property Inspector: • Change **Complexity** from `Inherited` to `On` or `Off`. • Change **Type** from `Inherit: Same as Simulink` to an explicit type.

Capabilities and Limitations

- This check only analyzes the functions that are directly referenced by the Simulink model.
- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- himl_0002: Strong data typing at MATLAB function boundaries

Check MATLAB Function metrics

Check ID: `mathworks.hism.himl_0003`

Display complexity and code metrics for MATLAB Functions. Report metric violations.

Description

This check provides complexity and code metrics for MATLAB Functions. The check reports two categories of metrics violations:

- Block-based metrics — Metrics for the overall code of the MATLAB Function block
- Function-based metrics — Metrics for each function of the block, presented individually

Available with Simulink Check.

Input Parameters

You can define the tolerance for these complexity parameters by using the Model Advisor Configuration Editor:

- **Maximum effective lines of code per function** — Effective lines do not include empty lines, comment lines, or lines with a function `end` keyword. The default value is `60`.
- **Minimum density of comments** — Density is ratio of comment lines to total lines of code. The default value is `0.2`.
- **Maximum cyclomatic complexity per function** — Cyclomatic complexity is the number of linearly independent paths through the source code. The default value is `15`.

Results and Recommended Actions

Condition	Recommended Action
MATLAB Function violates the complexity input parameters.	For the MATLAB Function: • If effective lines of code is too high, further divide the MATLAB Function. • If comment density is too low, add comment lines. • If cyclomatic complexity per function is too high, further divide the MATLAB Function.

Capabilities and Limitations

- This check only analyzes the functions that are directly referenced by the Simulink model.
- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- himl_0003: Limitation of MATLAB function complexity

Check MATLAB Code Analyzer messages

Check ID: `mathworks.hism.himl_0004`

Check MATLAB Functions for `%#codegen` directive, MATLAB Code Analyzer messages, and justification message IDs.

Description

Verifies `%#codegen` directive, MATLAB Code Analyzer messages, and justification message IDs for:

- MATLAB code in MATLAB Function blocks
- MATLAB functions defined in Stateflow charts
- Called MATLAB functions

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
For MATLAB code in MATLAB Function blocks, either of the following: • Code lines are not justified with a `%#ok` comment. • Codes lines justified with `%#ok` do not specify a message id.	• Implement MATLAB Code Analyzer recommendations. • Justify not following MATLAB Code Analyzer recommendations with a `%#ok` comment. • Specify justified code lines with a message id. For example, `%#ok<NOPRT>`.
For MATLAB functions defined in Stateflow charts, either of the following: • Code lines are not justified with a `%#ok` comment. • Codes lines justified with `%#ok` do not specify a message id.	• Implement MATLAB Code Analyzer recommendations. • Justify not following MATLAB Code Analyzer recommendations with a `%#ok` comment. • Specify justified code lines with a message id. For example, `%#ok<NOPRT>`.
For called MATLAB functions: • Code does not have the `%#codegen` directive. • Code lines are not justified with a `%#ok` comment. • Codes lines justified with `%#ok` do not specify a message id.	• Insert `%#codegen` directive in the MATLAB code. • Implement MATLAB Code Analyzer recommendations. • Justify not following MATLAB Code Analyzer recommendations with a `%#ok` comment. • Specify justified code lines with a message id. For example, `%#ok<NOPRT>`.

Capabilities and Limitations

- This check only analyzes the functions that are directly referenced by the Simulink model.
- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks or charts.

See Also

- himl_0004: MATLAB Code Analyzer recommendations for code generation

Check if/elseif/else patterns in MATLAB Function blocks

Check ID: `mathworks.hism.himl_0006`

Description

This check identifies the if/elseif/else patterns without appropriate else conditions in embedded MATLAB code.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Inappropriate if/elseif/else patterns are present in the embedded MATLAB code.	For every if/elseif/else pattern, add an else statement that includes at least one meaningful comment.

Capabilities and Limitations

- This check only analyzes the functions that are directly referenced by the Simulink model.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

See Also

- himl_0006: MATLAB code if / elseif / else patterns

Check switch statements in MATLAB Function blocks

Check ID: `mathworks.hism.himl_0007`

Description

This check identifies the switch/case/otherwise statements without appropriate conditions in embedded MATLAB code.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Switch statement does not have any otherwise condition.	Make sure that the switch statement has an otherwise condition.
Otherwise statement is left blank with no comments.	Make sure that the otherwise statement has at least one meaningful comment.
Switch statement has only one case statement.	Make sure that the switch statement has at least two case statements.

Capabilities and Limitations

- This check excludes a single **case** statement with a cell array of two or more elements.
- This check only analyzes the functions that are directly referenced by the Simulink model.
- Runs on library models.
- You can configure the check to run on referenced MATLAB files using the input parameter **Check .m files referenced in the model** in the Configuration Editor. By default this parameter is selected.

- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- himl_0007: MATLAB code switch / case / otherwise patterns

Check usage of relational operators in MATLAB Function blocks

Check ID: `mathworks.hism.himl_0008`

Description

This check inspects all MATLAB functions in the model, local functions, and referenced MATLAB files for the relational operator statements which operate on operands of different data types.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Relational operators in the function blocks operating on operands of different data types.	Type-cast the operands to be of the same data type.

Capabilities and Limitations

- This check only analyzes the functions that are directly referenced by the Simulink model.
- Does not run on library models.
- You can configure the check to run on referenced MATLAB files using the input parameter **Check .m files referenced in the model** in the Configuration Editor. By default this parameter is selected.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- himl_0008: MATLAB code relational operator data types

Check usage of logical operators and functions in MATLAB Function blocks

Check ID: `mathworks.hism.himl_0010`

Description

This check identifies the logical operators and functions operating on operands with numeric data types in MATLAB Function blocks.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Logical operators or functions used on operands with numeric data types.	Type-cast the operands to be of a logical data type.

Capabilities and Limitations

- This check only analyzes the functions that are directly referenced by the Simulink model.
- Does not run on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- You can configure the check to run on referenced MATLAB files using the input parameter **Check .m files referenced in the model** in the Configuration Editor. By default this parameter is selected.
- Allows exclusions of blocks and charts.

See Also

- himl_0010: MATLAB code with logical operators and functions

Check state machine type of Stateflow charts

Check ID: `mathworks.hism.hisf_0001`

Identify Stateflow Charts whose State Machine Type differs from the type set in the Model Advisor Configuration Editor.

Description

Compares the state machine type of all Stateflow charts to the type that you specify in the input parameters.

Available with Simulink Check.

This check requires a Stateflow license.

Input Parameters

Classic

Check whether all charts are Classic charts.

Mealy

Check whether all charts are Mealy charts.

Moore

Check whether all charts are Moore charts.

Results and Recommended Actions

Condition	Recommended Action
The input parameter is set to `Classic` and charts in the model use other state machine types.	For each chart, in the Chart Properties dialog box, specify **State Machine Type** to `Classic`.
The input parameter is set to `Moore` and charts in the model use other state machine types.	For each chart, in the Chart Properties dialog box, specify **State Machine Type** to `Moore`.
The input parameter is set to `Mealy` and charts in the model use other state machine types.	For each chart, in the Chart Properties dialog box, specify **State Machine Type** to `Mealy`.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- hisf_0001: State Machine Type

Check Stateflow charts for ordering of states and transitions

Check ID: `mathworks.hism.hisf_0002`

Identify Stateflow charts that have **User-specified state/transition execution order** cleared.

Description

Identify Stateflow charts that have **User-specified state/transition execution order** cleared, and therefore do not use explicit ordering of parallel states and transitions.

Available with Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
Stateflow charts have **User-specified state/ transition execution order** cleared.	For the specified charts, in the Chart Properties dialog box, select **User-specified state/ transition execution order**.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

Action Results

Clicking **Modify** selects **User-specified state/transition execution order** for the specified charts.

See Also

- hisf_0002: User-specified state/transition execution order

Check usage of recursions

Check ID: `mathworks.hism.hisf_0004`

Description

Identifies the usage of recursive behavior.

Note The undirected events are known to cause recursion in the generated code.

Available with Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
One or more entities were involved in recursions.	Remodel the entities to remove recursions.

Capabilities and Limitations

- This check can only be run from root level of the model.
- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.

- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- hisf_0004: Protect against recursive function calls to improve code compliance

Check Stateflow debugging options

Check ID: `mathworks.hism.hisf_0011`

Check the Stateflow debugging settings.

Description

Verify the following debugging settings.

- **Wrap on overflow**
- **Simulation range checking**
- **Detect Cycles**
- **Underspecification**
- **Overspecification**

Available with Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
Any of the following: • **Wrap on overflow** is not set to `error`. • **Simulation range checking** is not set to `error`. • Parameter **Underspecification** for the truth table is not set to `error` • Parameter **Overspecification** for the truth table is not set to `error`	In the Configuration Parameters dialog box, set: • **Wrap on overflow** to `error`, or set the parameter **IntegerOverflowMsg** to `error`. • **Simulation range checking** to `error`, or set the parameter **SignalRangeChecking** to `error`. • Set **Underspecification** to `error`. • Set **Overspecification** to `error`.
Inside a Stateflow chart, **Detect Cycles** is cleared.	In the model window, select **Debug > Diagnostics > Detect Cyclical Behavior**

Capabilities and Limitations

- Allows exclusions of blocks and charts.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

Action Results

Clicking **Modify** selects the specified debugging options.

See Also

- hisf_0011: Stateflow debugging settings

Check Stateflow charts for transition paths that cross parallel state boundaries

Check ID: `mathworks.hism.hisf_0013`

Identify transition paths that cross parallel state boundaries in Stateflow charts.

Description

Identify transition paths that cross parallel state boundaries in Stateflow charts. This check identifies transition paths that cross parallel state boundaries in Stateflow charts.

Available with Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
The Stateflow charts have transition paths that cross parallel state boundaries.	Modify the Stateflow charts so that transitions do not cross parallel state boundaries. For more information see, “Transition Between Operating Modes” (Stateflow).

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- hisf_0013: Usage of transition paths (crossing parallel state boundaries)

Check for inappropriate use of transition paths

Check ID: `mathworks.hism.hisf_0014`

Description

This check inspects the use of junctions inside states and identifies the junctions that lie on a path that goes in and out of a state.

Available with Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
One or more transition paths in the model traverses through a state without ending on a substate.	Remodel the junctions to avoid transition paths that go into and out of a state without ending on a substate.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

See Also

- hisf_0014: Usage of transition paths (passing through states)

Check Stateflow charts for strong data typing

Check ID: `mathworks.hism.hisf_0015`

Identify variables and parameters in expressions with different data types in Stateflow objects.

Description

To facilitate strong data typing, this check identifies the variables and parameters in expressions with different data types in Stateflow states and transitions.

Available with Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
The Stateflow objects have variables and parameters in expressions with different data types.	Explicitly cast variables and parameters in expressions to the same data types. For more information see, `cast`.

Capabilities and Limitations

- Does not run on library models.
- Allows exclusions of blocks and charts.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Does not analyze the type of literals in expressions in Stateflow objects. Explicitly casts types of literals to the intended data type.
- Does not flag expressions with true and false keywords. For more information, see “Reserved Keywords for Code Generation” (Embedded Coder).
- External functions written in C and embedded in Stateflow will not work with this check.

See Also

- hisf_0015: Strong data typing (casting variables and parameters in expressions)

Check naming of ports in Stateflow charts

Check ID: `mathworks.hism.hisf_0016`

Description

This check identifies the mismatches between names of Stateflow ports and associated signals. The reusable Stateflow blocks can have different port names.

Available with Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
Names of the input and output ports of Stateflow charts are not the same as the names of the signals connected.	Make sure that the names of the input and output ports of Stateflow charts are same as the names of the signals connected.

Capabilities and Limitations

- This check does not analyze port names of Stateflow Truth Tables or Stateflow State Transition Tables.

- This check considers reusable Stateflow charts as library linked charts and are not flagged.
- This check does not flag signals without names.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- hisf_0016: Stateflow port names

Check scoping of Stateflow data objects

Check ID: `mathworks.hism.hisf_0017`

Description

This check identifies the Stateflow data objects with local scope that are not scoped at the chart level or below.

Available with Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
One or more Stateflow data objects with local scope are not defined at the chart level or below.	Make sure to define all the Stateflow data objects with local scope at the chart level or below.

Capabilities and Limitations

- Does not analyze content of library linked blocks.
- Does not analyze content in masked subsystems.
- Allows exclusions of blocks and charts.

See Also

- hisf_0017: Stateflow data object scoping

Check assignment operations in Stateflow Charts

Check ID: `mathworks.hism.hisf_0065`

Identify assignment operations in Stateflow objects.

Description

This check identifies the assignment operations in Stateflow objects that implicitly cast integer and fixed-point arithmetic calculations to wider data types than the input data types.

This check identifies only the assignments with arithmetic operations.

Available with Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
The Stateflow object consists of assignment operations that cast integer and fixed-point calculations to wider data types than the input data types.	Explicitly replace assignment operator (=) to := operator in Stateflow objects.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- External functions written in C and embedded in Stateflow will not work with this check.

See Also

- hisf_0065: Type cast operations in Stateflow to improve code compliance

Check Stateflow charts for unary operators

Check ID: `mathworks.hism.hisf_0211`

Identify unary operators in Stateflow charts.

Description

This check identifies the unary minus operators on unsigned data types in Stateflow charts.

Available with Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
The Stateflow chart consists of a unary minus operator on unsigned data types.	Explicitly modify the unary operator on unsigned data types. For more information, see “Unary Operations and Actions” (Stateflow).

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Except for Shift and Divide operations, this check does not flag expressions with bitwise and arithmetic operators.

See Also

- hisf_0211: Protect against use of unary operators in Stateflow Charts to improve code compliance

Check usage of Abs blocks

Check ID: `mathworks.hism.hisl_0001`

Identify usage of Math Operation blocks that might impact safety.

Description

This check inspects the usage of the Abs block.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains an Absolute Value block that is operating on one of the following: • A boolean or an unsigned input data type. This condition results in unreachable simulation pathways through the model and might result in unreachable code • A signed integer value with the **Saturate on integer overflow** check box not selected. For signed data types, the absolute value of the most negative value is problematic because it is not representable by the data type. This condition results in an overflow in the generated code.	If the identified Absolute Value block is operating on a boolean or unsigned data type, do one of the following: • Change the input of the Absolute Value block to a signed input type. • Remove the Absolute Value block from the model. If the identified Absolute Value block is operating on a signed data type, in the **Block Parameters > Signal Attributes** dialog box, select **Saturate on integer overflow**.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts.

See Also

- hisl_0001: Usage of Abs block

Check usage of remainder and reciprocal operations

Check ID: `mathworks.sldv.hism.hisl_0002`

Description

Identifies usage of remainder and reciprocal operations that cause non-finite results.

Available with Simulink Check. This check requires a Simulink Design Verifier (SLDV) license.

Note: This check will perform SLDV analysis on the model.

Results and Recommended Actions

Condition	Recommended Action
Remainder and reciprocal operations in the model have inputs that can be equal to zero during simulation.	Make sure that the inputs to the remainder and reciprocal operations are not equal to zero.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- hisl_0002: Usage of Math Function blocks (rem and reciprocal)

Check usage of log and log10 operations

Check ID: `mathworks.sldv.hism.hisl_0004`

Description

Identifies log and log10 operations in the model that can cause nonfinite results.

Available with Simulink Check. This check requires a Simulink Design Verifier (SLDV) license.

Note: This check will perform SLDV analysis on the model.

Results and Recommended Actions

Condition	Recommended Action
Natural/base 10 logarithm (`Log` and `Log10`) operations in the model have inputs that can be less than or equal to zero during simulation.	Make sure that the input of `log` and `log 10` operations in the model are not less than or equal to zero.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- hisl_0004: Usage of Math Function blocks (natural logarithm and base 10 logarithm)

Check usage of While Iterator blocks

Check ID: `mathworks.hism.hisl_0006`

Description

This check inspects the usage of While Iterator blocks.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains a While Iterator block that has unlimited iterations. This condition can lead to infinite loops in the generated code.	For the identified While Iterator blocks: • Set the **Maximum number of iterations (-1 for unlimited)** parameter to a positive integer value. • Consider selecting the **Show iteration number port** check box and observe the iteration value during simulation.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- hisl_0006: Usage of While Iterator blocks

Check usage of For and While Iterator subsystems

Check ID: `mathworks.hism.hisl_0007`

Description

This check inspects the usage of time-dependent blocks in a For Iterator or While Iterator subsystem.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains one of the following time-dependent blocks in a For Iterator or While Iterator subsystem: • Discrete State-Space • Discrete-Time Integrator • Discrete FIR Filter • Discrete Filter • Discrete Transfer Fcn • Discrete Zero-Pole • Transfer Fcn First Order • Transfer Fnc Real Zero • Transfer Fcn Lead or Lag • Discrete Transfer Function with Initial Outputs • Discrete Transfer Function with Initial States • Discrete Zero-Pole with Initial Outputs • Discrete Zero-Pole with Initial States • Discrete Derivative	In the model or subsystem, consider removing the time-dependent blocks.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- hisl_0007: Usage of For Iterator or While Iterator subsystems

Check usage of For Iterator blocks

Check ID: `mathworks.hism.hisl_0008`

Description

This check inspects the usage of For Iterator blocks.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains a For Iterator block that has variable iterations. This condition can lead to unpredictable execution times or infinite loops in the generated code.	For the identified For Iterator blocks, do one of the following: • Set the **Iteration limit source** parameter to `internal`. • If the **Iteration limit source** parameter must be `external`, use a Constant, Probe, or Width block as the source. • Clear the **Set next i (iteration variable) externally** check box. • Consider selecting the **Show iteration variable** check box and observe the iteration value during simulation.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- hisl_0008: Usage of For Iterator Blocks

Check usage of If blocks and If Action Subsystem blocks

Check ID: `mathworks.hism.hisl_0010`

Description

This check inspects the usage of If blocks.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains an If block using Elseif expressions without an Else condition.	In the If block Block Parameters dialog box, select **Show else condition**. Connect the resulting Else output port to an If Action Subsystem block.

Condition	Recommended Action
The model or subsystem contains an If block with output ports that do not connect to If Action Subsystem blocks.	Verify that output ports of the If block connect to If Action Subsystem blocks.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- hisl_0010: Usage of If blocks and If Action Subsystem blocks

Check usage of Switch Case blocks and Switch Case Action Subsystem blocks

Check ID: `mathworks.hism.hisl_0011`

Description

This check inspects the usage of Switch Case blocks

The check flags Switch Case blocks that do not use integer data types or enumeration values for inputs. To comply with “hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks” – C, use an integer data type or an enumeration value for the inputs to Switch Case blocks.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains a Switch Case block without a default case.	In the Switch Case block Block Parameters dialog box, select **Show default case**. Connect the resulting default output port to a Switch Case Action Subsystem block.
The model or subsystem contains a Switch Case block with an output port that does not connect to a Switch Case Action Subsystem block.	Verify that output ports of the Switch Case blocks connect to Switch Case Action Subsystem blocks.
The model or subsystem contains a Switch Case block with non-integer or non-enum input port data types.	Make sure that input data type of the Switch Case blocks is integer or enum.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- hisl_0011: Usage of Switch Case blocks and Action Subsystem blocks

Check usage of conditionally executed subsystems

Check ID: `mathworks.hism.hisl_0012`

Description

This check identifies the blocks with incorrect sample times in conditionally executed subsystems and asynchronously executed sample time dependent blocks.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Sample time for the blocks is not set to `-1` in a conditionally executed subsystem: • If Action • Switch Case Action • Function-Call • Triggered • Enabled	Change the sample time for the blocks to be `-1`.

Condition	Recommended Action
The model contains asynchronously executed sample time dependent blocks: • Discrete State-Space • Discrete-Time Integrator • Discrete FIR Filter • Discrete Filter • Discrete Transfer Fcn • Discrete Zero-Pole • Transfer Fcn First Order • Transfer Fnc Real Zero • Transfer Fcn Lead or Lag • Discrete Transfer Function with Initial Outputs • Discrete Transfer Function with Initial States • Discrete Zero-Pole with Initial Outputs • Discrete Zero-Pole with Initial States • Discrete Derivative	Remodel to remove the sample time dependent blocks.

Capabilities and Limitations

- The asynchronously executed sample-time dependent blocks are flagged only if Triggered and Function-call blocks are present.
- Does not run on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

See Also

- hisl_0012: Usage of conditionally executed subsystems

Check safety-related diagnostic settings for data store memory

Check ID: `mathworks.hism.hisl_0013`

Check model configuration for diagnostic settings that apply to data store memory and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to data store memory are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects whether the model attempts to read data from a data store in which it has not stored data in the current time step is set to a value other than `Enable all as errors`. Reading data before it is written can result in use of stale data or data that is not initialized.	Set **Detect read before write** in the Configuration Parameters dialog box or set the parameter `ReadBeforeWriteMsg` to `Enable all as errors`.
The diagnostic that detects whether the model attempts to store data in a data store, after previously reading data from it in the current time step, is set to a value other than `Enable all as errors`. Writing data after it is read can result in use of stale or incorrect data.	Set **Detect write after read** in the Configuration Parameters dialog box or set the parameter `WriteAfterReadMsg` to `Enable all as errors`.
The diagnostic that detects whether the model attempts to store data in a data store twice in succession in the current time step is set to a value other than `Enable all as errors`. Writing data twice in one time step can result in unpredictable data.	Set **Detect write after write** in the Configuration Parameters dialog box or set the parameter `WriteAfterWriteMsg` to `Enable all as errors`.
The diagnostic that detects when one task reads data from a Data Store Memory block to which another task writes data is set to `none` or `warning`. Reading or writing data in different tasks in multitask mode can result in corrupted or unpredictable data.	Set **Multitask data store** in the Configuration Parameters dialog box or set the parameter `MultiTaskDSMMsg` to `error`.
The diagnostic detects that the parameter **Duplicate data store names** is not set to `error`.	Set **Duplicate data store names** in the Configuration Parameters dialog box or set the parameter `UniqueDataStoreMsg` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to data store memory and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0013: Usage of data store blocks

Check usage of Merge blocks

Check ID: `mathworks.hism.hisl_0015`

Description

This check identifies the Merge blocks that are not directly connected to a conditionally executed subsystem and have the **Allow unequal port widths** parameter set to on.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Merge block is not connected directly to a conditionally executed subsystem.	Remodel so that the Merge blocks are connected directly to conditionally executed subsystems.
The Merge block parameter **Allow unequal port widths** is set to **on**.	Set the Merge block parameter **Allow unequal port widths** to **off**
Multiple subsystem executes during a time step in conditionally executed subsystems.	Specify execution of the conditionally executed subsystems such that only one subsystem executes during a time step.
The Outport block parameter **Output when disabled** is not set to **held** for any conditionally executed subsystems in the model.	Set the Outport block parameter **Output when disabled** to **held** for each conditionally executed subsystem being merged.

Capabilities and Limitations

- This check will not check the execution order for conditionally executed subsystems.
- Does not run on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

See Also

- hisl_0015: Usage of Merge blocks

Check relational comparisons on floating-point signals

Check ID: `mathworks.hism.hisl_0016`

Description

Identifies the relational blocks or operations that perform equality or inequality comparisons on floating-point signals.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
One or more relational operations used in the model perform equality or inequality comparisons on floating-point signals.	For the identified blocks and expressions, do one of these: • Change the signal data type. • Remodel to eliminate using == or ~= operators on floating-point signals.
One ore more If blocks used in the model have `If` expressions or `Elseif` expressions that might cause floating point equality or inequality comparisons in generated code.	Modify the expressions to avoid floating point equality and inequality comparisons in generated code.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- “hisl_0016: Usage of blocks that compute relational operators”

  hisl_0016: Usage of equality and inequality comparisons on floating-point data types

Check usage of Relational Operator blocks

Check ID: `mathworks.hism.hisl_0017`

Description

This check inspects the usage of blocks that compute relational operators, including Relational Operator, Compare To Constant, Compare To Zero and, Detect Change blocks.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains a block computing a relational operator that is operating on different data types. The condition can lead to unpredictable results in the generated code.	For the identified blocks, use common data types as inputs. You can use Data Type Conversion blocks to change input data types.

Condition	Recommended Action
The model or subsystem contains a block computing a relational operator that does not have Boolean output. The condition can lead to unpredictable results in the generated code.	For the specified blocks, on the **Block Parameters > Signal Attributes** pane, set the **Output data type** to `boolean`.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- hisl_0017: Usage of blocks that compute relational operators (2)

Check usage of Logical Operator blocks

Check ID: `mathworks.hism.hisl_0018`

Identify usage of Logical Operator blocks that might impact safety.

Description

This check inspects the usage of Logical Operator blocks.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains a Logical Operator block that has inputs or outputs that are not Boolean inputs or outputs. The block might result in floating-point equality or inequality comparisons in the generated code.	• Modify the Logical Operator block so that all inputs and outputs are Boolean. On the **Block Parameters > Signal Attributes** pane, consider selecting **Require all inputs to have the same data type** and setting **Output data type** to `boolean`. • In the Configuration Parameters dialog box, consider selecting the **Implement logic signals as boolean data (vs. double)**.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.

- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- hisl_0018: Usage of Logical Operator block

Check usage of bit operation blocks

Check ID: `mathworks.hism.hisl_0019`

Identify bit operation blocks with signed data types as inputs

Description

This check identifies the use of the Bitwise Operations for the input and output data types. The check also looks at other bit operations blocks as following:

- Bit Clear
- Bit Set
- Bit Shift
- Bitwise operator
- Extract Bits
- Shift Arithmetic

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Bitwise Operations are used with signed data types.	Use unsigned data type for Bitwise operations.

Capabilities and Limitations

- Allows exclusions of blocks and charts.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

See Also

- hisl_0019: Usage of bitwise operations

Check for blocks not recommended for C/C++ production code deployment

Check ID: `mathworks.hism.hisl_0020`

Identify blocks not supported by code generation or not recommended for C/C++ production code deployment.

Description

This check partially identifies model constructs that are not recommended for C/C++ production code generation. For Simulink Coder™ and Embedded Coder®, these model construct identities appear in tables of Simulink Block Support (Simulink Coder). .

In some instances, this check flags blocks that are supported for code generation. For these blocks, you should review the footnote information that is provided in the support notes and adhere to the recommended action provided by the Model Advisor.

Available with a Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks that should not be used for production code deployment.	Consider replacing the blocks listed in the results. Click an element from the list of questionable items to locate condition.
The model or subsystem contains blocks that are supported but not recommended for production code generation.	Review the support notes and adhere to the recommended action provided by the Model Advisor.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems.
- Allows exclusions of blocks and charts.

See Also

- “hisl_0020: Blocks not recommended for MISRA C:2012 compliance”
- “Use Blocks and Products Supported for Code Generation” (Simulink Coder)
- “Use Blocks and Products Supported for Code Generation” (Simulink Coder)
- “Model Advisor Exclusion Overview”

Check for inconsistent vector indexing methods

Check ID: `mathworks.hism.hisl_0021`

Identify blocks with inconsistent indexing method.

Description

Using inconsistent block indexing methods can result in modeling errors. You should use a consistent vector indexing method for all blocks. The indexing methods are zero-based, one-based or user-specified.

Blocks that support configurable indexing:

- Assignment
- For Iterator
- Index Vector
- Multiport Switch
- Selector

Blocks that support only one-based indexing:

- Fcn (deprecated)
- MATLAB Function
- MATLAB System
- State Transition Table with MATLAB action language
- Test Sequence
- Stateflow chart with MATLAB action language
- Truth Table function with MATLAB action language

Blocks that supports only zero-based indexing:

- Stateflow chart with C action language
- Truth Table function with C action language
- State Transition Table with C action language

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks with inconsistent indexing methods. The indexing methods are zero-based, one-based or user-specified.	Modify the model to use a single consistent indexing method.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- hisl_0021: Consistent vector indexing method

Check data types for blocks with index signals

Check ID: `mathworks.hism.hisl_0022`

Description

This check Identifies the blocks with index signals that have data types other than integers or enum and are within the range of indexed values.

Available with Simulink Check.

Check Parameterization

You can use Model Advisor Configuration Editor to configure this check to include files with a `.m` extension in the analysis. To enable this feature, in the **Input Parameters** section, select **Check external .m files referenced in the model**.

Results and Recommended Actions

Condition	Recommended Action
One or more Simulink blocks in the model have index signals that have data types other than integer or enum.	Change the data type of block index signals to an integer or enum data type that covers the range of indexed values.
One or more MATLAB Function blocks have index variables with inappropriate data types.	Change the data type of index variables to an integer or enum data type that covers the range of indexed values.
One or more Stateflow charts in the model have index variables that have data types other than integer or enum.	Change the data type of index signals of the blocks to an integer or enum data type that covers the range of indexed values.

Capabilities and Limitations

- This check does not support dialog set indices.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- hisl_0022: Data type selection for index signals

Check usage of variant blocks

Check ID: `mathworks.hism.hisl_0023`

Check variant block parameters for settings that might result in code that does not trace to requirements.

Description

This check verifies that variant block parameters for code generation are set to trace to requirements.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The option to generate preprocessor conditionals is selected in one or more variant blocks in the model.	In order to simplify the tracing of code to requirements, consider clearing the option to generate preprocessor conditionals in variant blocks.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0023: Verification of variant blocks

Check for root Inports with missing properties

Check ID: `mathworks.hism.hisl_0024`

Description

Identifies the following:

- root level Inport blocks with missing or inherited sample times, data types or port dimensions for Simulink models.

  Using root model Inport blocks that do not have defined sample time, data types or port dimensions can lead to undesired simulation results. Simulink back-propagates dimensions, sample times, and data types from downstream blocks unless you explicitly assign these values. You can specify Inport block properties with block parameters or Simulink signal objects that explicitly resolve to the connected signal lines.
- root level Input ports with missing or inherited data types or port dimensions for Architecture models.

When you run the check, a results table provides links to Inport blocks and signal objects that do not pass, along with conditions triggering the warning.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Missing port dimension — Model contains Inport blocks with inherited port dimensions.	For the listed Inport blocks and Simulink signal objects, specify port dimensions.
Missing signal data type — Model contains Inport blocks with inherited data types.	For the listed Inport blocks and Simulink signal objects, specify data types.
Missing port sample time — Model contains Inport blocks with inherited sample times.	For the listed Inport blocks and Simulink signal objects, specify sample times. The sample times for root Inports with bus type must match the sample times specified at the leaf elements of the bus object.
Implicit resolution to a Simulink signal object — Model contains Inport block signal names that implicitly resolve to a Simulink signal object in the base workspace, model workspace, or Simulink data dictionary.	For the listed Simulink signal objects, in the property dialog, select signal property **Signal name must resolve to Simulink signal object**. To set this option programmatically, use the port parameter **MustResolveToSignalObject**.
One or more Input ports of Architecture model do not have a data interface assigned to it.	Assign data interfaces to listed Input ports.

Capabilities and Limitations

- If the configuration parameter (**Configuration Parameters > Solver > Periodic sample time constraint**) is set to `Ensure sample time independent`, this check does not report warnings when input ports uses *inherited sample time*.
- For export-function models, *inherited sample time* is not flagged.
- Does not run on library models.
- Does not support exclusion in Architecture models.
- Allows exclusions of blocks and charts.

See Also

- hisl_0024: Inport interface definition

Check for root Inports with missing range definitions

Check ID: `mathworks.hism.hisl_0025`

Description

Identifies the following:

- root level Inport blocks with missing or erroneous minimum or maximum range values for Simulink models.

  The check identifies root level Inport blocks with missing or erroneous minimum or maximum range values. You can specify Inport block minimum and maximum values with block parameters or Simulink signal objects that explicitly resolve to the connected signal lines.

- root level Input ports with missing or erroneous minimum or maximum range values for Architecture models.

A results table provides links to Inport blocks and signal objects that do not pass the check, along with conditions triggering the warning.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Missing range — Model contains Inport blocks with numeric data types that have missing range parameters (minimum and/or maximum).	For the listed Inport blocks and Simulink signal objects, specify scalar minimum and maximum parameters.
Missing range(s) for bus object — Bus objects defining the Inport blocks have bus elements with missing ranges.	For the listed bus elements, to specify the model interface range, provide scalar minimum and maximum parameters.
Range specified will be ignored — Minimum or maximum values at Inports or Simulink signal objects are not supported for bus data types. The values are ignored during range checking.	To enable range checking, specify minimum and maximum signal values on the bus elements of the bus objects defining the data type.
No data type specified — Model contains Inport blocks or Simulink signal objects with inherited data types.	Specify one of the supported data types: • `Enum` • `Simulink.AliasType` • `Simulink.Bus` • `Simulink.NumericType` • build-in
Implicit resolution to a Simulink signal object — Model contains Inport block signal names that implicitly resolve to a Simulink signal object in the base workspace, model workspace, or Simulink data dictionary.	For the listed Simulink signal objects, in the property dialog, select signal property **Signal name must resolve to Simulink signal object**. To set this option programmatically, use the port parameter **MustResolveToSignalObject**.
One or more Input ports of Architecture model have missing or erroneous range definitions	Data Interface assigned with the listed Input ports has missing ranges. Provide minimum and maximum values to the data interface.

Capabilities and Limitations

- Does not run on library models.
- Does not support exclusion in Architecture models.
- Allows exclusions of blocks and charts.

See Also

- hisl_0025: Design min/max specification of input interfaces

Check for root Outports with missing range definitions

Check ID: `mathworks.hism.hisl_0026`

Description

Identifies the following:

- root level Outport blocks with missing or erroneous minimum or maximum range values for Simulink models.

  You can specify Outport block minimum and maximum values with block parameters or Simulink signal objects that explicitly resolve to the connected signal lines.
- root level Output ports with missing or erroneous minimum or maximum range values for Architecture models.

A results table provides links to Outport blocks that do not pass the check, along with conditions triggering the warning.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Missing range — Model contains Outport blocks with numeric data types that have missing range parameters (minimum and/or maximum).	For the listed Outport blocks and Simulink signal objects, specify scalar minimum and maximum parameters.
Missing range(s) for bus object — Bus objects defining the Outport blocks have leaf elements with missing ranges.	For the listed leaf elements, to specify the model interface range, provide scalar minimum and maximum parameters.
Range specified at Outport will be ignored — Minimum or maximum values at Outports or Simulink signal objects are not supported for bus data types. The values are ignored during range checking.	To enable range checking, specify minimum and maximum signal values on the leaf elements of the bus objects defining the data type.
No bus data type specified — Model contains Outport block or Simulink signal objects with inherited bus data types.	For the Outport blocks and Simulink signal objects, specify one of the supported data types: • `Enum` • `Simulink.AliasType` • `Simulink.Bus` • `Simulink.NumericType` • built-in
Implicit resolution to a Simulink signal object — Model contains Outport block signal names that implicitly resolve to a Simulink signal object in the base workspace, model workspace, or Simulink data dictionary.	For the listed Simulink signal objects, in the property dialog, select signal property **Signal name must resolve to Simulink signal object**. To set this option programmatically, use the port parameter **MustResolveToSignalObject**.
One or more Output ports of Architecture model have missing or erroneous range definitions.	Data interface assigned with the listed Output ports has missing ranges. Provide minimum and maximum values to the data interface.

Capabilities and Limitations

- This check does not flag ports with data type of Enum or Boolean when min max values are not set.
- Does not run on library models.
- Does not support exclusion in Architecture models.
- Allows exclusions of blocks and charts.

See Also

- hisl_0026: Design min/max specification of output interfaces

Check usage of Assignment blocks

Check ID: `mathworks.hism.hisl_0029`

Identify usage of Math Operation blocks that might impact safety.

Description

This check inspects the usage of the Assignment blocks.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem might contain Assignment blocks with incomplete array initialization (not in the iterator subsystem) that do not have block parameter **Action if any output element is not assigned** set to **Error** or **Warning**.	Set block parameter **Action if any output element is not assigned** to one of the recommended values: • **Error** • **Warning**
The model or subsystem might contain Assignment blocks in the iterator subsystem and the parameter **Action if any output element is not assigned** is not set to **Error**.	Set block parameter **Action if any output element is not assigned** to **Error**.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts.

See Also

- hisl_0029: Usage of Assignment blocks

Check model file name

Check ID: `mathworks.hism.hisl_0031`

Description

This check inspects the model file name to ensure that the name complies with the recommended guidelines.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The file name contains illegal characters.	Rename the file. Allowed characters are a–z, A–Z, 0–9, and underscore (_).
The file name starts with a number.	Make sure that the file name does not start with a number.
The file name starts with an underscore ("_").	Make sure that the file name does not start with an underscore ("_").
The file name ends with an underscore ("_").	Make sure that the file name does not end with an underscore ("_").
The file extension contains one or more underscores.	Change the file extension.
The file name has consecutive underscores.	Rename the file to eliminate trailing underscore.
The file name contains more than one dot (".").	Make sure that the file name does not have more than one dot (".").
The file name is a C/C++ or MATLAB keyword or built in function	Rename the file.
File name does not have more than 2 and less than 64 characters.	Make sure that the file name has a minimum of **2** and maximum of **64** supported characters.

Capabilities and Limitations

- Runs on library models.

See Also

- hisl_0031: Model file names

Check model object names

Check ID: `mathworks.hism.hisl_0032`

Check model object names.

Description

This check verifies that the following model object names comply with your own modeling guidelines or the high-integrity modeling guidelines. The check also verifies that the model object does not use a reserved name.

- Blocks
- Signals
- Parameters
- Buses
- Stateflow objects

Reserved names:

- MATLAB keywords
- Reserved keywords for C, C++, and code generation. For a complete list, see “Reserved Keywords” (Simulink Coder)
- `int8` , `uint8`
- `int16`, `uint16`
- `int32`, `uint32`
- `inf`, `Inf`
- `NaN`, `nan`
- `eps`
- `intmin`, `intmax`
- `realmin`, `realmax`
- `pi`
- `infinity`
- `Nil`

Note In some cases, the Model Advisor reports an issue in multiple subchecks of this check.

Available with Simulink Check.

Input Parameters

To specify the naming standard and model object names that the check flags, use the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and navigate to **Check model object names**. In the **Input Parameters** pane, for each of the model objects, select one of the following:

   - `MAB` to use the MAB naming standard. When you select `MAB`, the check uses the regular expression `(^.{32,}$)|([^a-zA-Z_0-9])|(^\d)|(^ )|(__)|(^_)|(_$)` to verify that names:

     - Use these characters: `a-z`, `A-Z`, `0-9`, and the underscore `(_)`.
     - Do not start with a number.

- Do not use underscores at the beginning or end of a string.
- Do not use more than one consecutive underscore.
- Use strings that are less than 32 characters.

- `Custom` to use your own naming standard. When you select `Custom`, you can enter your own **Regular expression for prohibited *<model object>* names**. For example, if you want to allow more than one consecutive underscore, enter `(^.{32,}$)|([^a-zA-Z_0-9])|(^\d)|(^ )|(^_)|(_$)`
- `None` if you do not want the check to verify the model object name

2 Click **Apply**.

3 Save the configuration. When you run the check using this configuration, the check uses the input parameters that you specified.

Results and Recommended Actions

Condition	Recommended Action
The model object names do not comply with the naming standard specified in the input parameters.	Update the model object names to comply with your own guidelines or the high-integrity guidelines.

Capabilities and Limitations

- Does not run on library models.
- Does not analyze content of library linked blocks.
- Does not analyze content in all masked subsystems.
- Allows exclusions of blocks and charts.

See Also

- “hisl_0032: Model object names”
- MAB guidelines:
  - jc_0201: Usable characters for subsystem names
  - jc_0211: Usable characters for Inport blocks and Outport block
  - jc_0231: Usable characters for block names
  - na_0019: Restricted variable names

Check usage of lookup table blocks

Check ID: `mathworks.hism.hisl_0033`

Check for lookup table blocks that do not generate out-of-range checking code.

Description

This check verifies that the following blocks generate code to protect against inputs that fall outside the range of valid breakpoint values:

- 1-D Lookup Table

- 2-D Lookup Table
- n-D Lookup Table
- Prelookup

This check also verifies that Interpolation Using Prelookup blocks generate code to protect against inputs that fall outside the range of valid index values.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The lookup table block does not generate out-of-range checking code.	Change the setting on the block dialog box so that out-of-range checking code is generated. • For the 1-D Lookup Table, 2-D Lookup Table, n-D Lookup Table, and Prelookup blocks, clear the check box for **Remove protection against out-of-range input in generated code**. • For the Interpolation Using Prelookup block, clear the check box for **Remove protection against out-of-range index in generated code**.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

Action Results

Clicking **Modify** verifies that lookup table blocks are set to generate out-of-range checking code.

See Also

- hisl_0033: Usage of Lookup Table blocks

Check usage of Signal Routing blocks

Check ID: `mathworks.hism.hisl_0034`

Identify usage of Signal Routing blocks that might impact safety.

Description

This check identifies model or subsystem Switch blocks that might generate code with inequality operations (~=) in expressions that contain a floating-point variable or constant.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains a Switch block that might generate code with inequality operations (~=) in expressions where at least one side of the expression contains a floating-point variable or constant. The Switch block might cause floating-point inequality comparisons in the generated code.	For the identified block, do one of the following: • For the control input block, change the **Data type** parameter setting. • Change the Switch block **Criteria for passing first input** parameter setting. This might change the algorithm.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts.

See Also

- hisl_0034: Usage of Signal Routing blocks

Check safety-related diagnostic settings for saving

Check ID: `mathworks.hism.hisl_0036`

Check model configuration for diagnostic settings that apply to saving model files

Description

This check verifies that model configuration parameters are set optimally for saving a model for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects whether a model contains disabled library links before the model is saved is set to `none` or `warning`. If this condition is undetected, incorrect code might be generated.	Set **Block diagram contains disabled library links** in the Configuration Parameters dialog box or set parameter `SaveWithDisabledLinkMsg` to `error`.

Condition	Recommended Action
The diagnostic that detects whether a model contains library links that are using parameters not in a mask before the model is saved is set to `none` or `warning`. If this condition is undetected, incorrect code might be generated.	Set **Block diagram contains parameterized library links** in the Configuration Parameters dialog box or set parameter `SaveWithParameterizedLinksMsg` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to saving a model file.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0036: Configuration Parameters > Diagnostics > Saving

Check safety-related model referencing settings

Check ID: `mathworks.hism.hisl_0037`

Check model configuration for model referencing settings that can impact safety.

Description

This check verifies that model configuration parameters for model referencing are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The referenced model is configured such that its target is rebuilt whenever you update, simulate, or generate code for the model, or if the Simulink software detects changes in known dependencies. These configuration settings can result in unnecessary regeneration of the code, resulting in changing only the date of the file and slowing down the build process when using model references.	Set **Rebuild** on the **Model Referencing** pane in the Configuration Parameters dialog box or set the parameter `UpdateModelReferenceTargets` to `Never` or `If any changes detected`.

Condition	Recommended Action
The diagnostic that detects whether a target needs to be rebuilt is set to `None` or `Warn if targets require rebuild`. For safety-related applications, an error should alert model developers that the parent and referenced models are inconsistent. This diagnostic parameter is available only if **Rebuild** is set to `Never`.	Set the configuration parameter **Never rebuild diagnostics** on the **Model Referencing** pane in the Configuration Parameters dialog box or set the parameter `CheckModelReferenceTargetMessage` to `error`.
The ability to pass scalar root input by value is off. This capability should be off because scalar values can change during a time step and result in unpredictable data. This parameter is only available when the config parameter **Total number of instances allowed per top model** is set to **One** or **Multiple** (`ModelReferenceNumInstancesAllowed` is `single` or `multi`).	Set **Pass fixed-size scalar root inputs by value for code generation** on the **Model Referencing** pane in the Configuration Parameters dialog box or set the parameter `ModelReferencePassRootInputsByReference` to `off`.
The model is configured to minimize algebraic loop occurrences. This configuration is incompatible with the recommended setting of **Single output/ update function** for embedded systems code.	In the Configuration Parameters dialog box, set **Minimize algebraic loop occurrences** or set parameter `ModelReferenceMinAlgLoopOccurrences` to `off`.

Action Results

Clicking **Modify Settings** configures model referencing settings that can impact safety.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0037: Configuration Parameters > Model Referencing

Check safety-related code generation settings for comments

Check ID: `mathworks.hism.hisl_0038`

Check model configuration for code generation settings that can impact safety.

Description

This check verifies that model configuration parameters for code generation are set optimally for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The option to include comments in the generated code is cleared. Comments provide good traceability between the code and the model.	Select **Include comments** (Simulink Coder) on the **Code Generation** > **Comments** pane in the Configuration Parameters dialog box or set the parameter `GenerateComments` to `on`.
The option to include comments that describe the code for blocks is cleared. Comments provide good traceability between the code and the model.	Select Simulink block comments (Simulink Coder) on the **Code Generation** > **Comments** pane in the Configuration Parameters dialog box or set the parameter `SimulinkBlockComments` to `on`.
The option to include comments that describe the code for blocks eliminated from a model is cleared. Comments provide good traceability between the code and the model.	Select **Show eliminated blocks** (Simulink Coder) on the **Code Generation** > **Comments** pane in the Configuration Parameters dialog box or set the parameter `ShowEliminatedStatement` to `on`.
The option to include the names of parameter variables and source blocks as comments in the model parameter structure declaration in *`model`*`_prm.h` is cleared. Comments provide good traceability between the code and the model.	Select **Verbose comments for 'Model default' storage class** (Simulink Coder) on the **Code Generation** > **Comments** pane in the Configuration Parameters dialog box or set the parameter `ForceParamTrailComments` to `on`.
The option to include requirement descriptions assigned to Simulink blocks as comments is cleared. Comments provide good traceability between the code and the model.	Select **Requirements in block comments** (Embedded Coder) on the **Code Generation** > **Custom comments** pane in the Configuration Parameters dialog box or set the parameter `ReqsInCode` to `on`.

Action Results

Clicking **Modify Settings** configures model code generation settings that can impact safety.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0038: Configuration Parameters > Code Generation > Comments
- “Model Configuration Parameters: Comments” (Simulink Coder)
- “Model Configuration Parameters: Code Generation Identifiers” (Simulink Coder)
- “Model Configuration Parameters: Code Generation Interface” (Simulink Coder)
- “Model Configuration Parameters: Code Style” (Embedded Coder)

Check safety-related code generation interface settings

Check ID: `mathworks.hism.hisl_0039`

Check model configuration for code generation settings that can impact safety.

Description

This check verifies that model configuration parameters for code generation are set optimally for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The option to generate nonfinite data and operations is selected. Support for nonfinite numbers is inappropriate for real-time embedded systems.	Clear **Support: non-finite numbers** (Simulink Coder) on the **Code Generation** > **Interface** pane in the Configuration Parameters dialog box or set the parameter `SupportNonFinite` to `off`.
The option to generate and maintain integer counters for absolute and elapsed time is selected. Support for absolute time is inappropriate for real-time safety-related systems.	Clear **Support: absolute time** (Embedded Coder) on the **Code Generation** > **Interface** pane in the Configuration Parameters dialog box or set the parameter `SupportAbsoluteTime` to `off`.
The option to generate code for blocks that use continuous time is selected. Support for continuous time is inappropriate for real-time safety-related systems.	Clear **Support: continuous time** (Embedded Coder) on the **Code Generation** > **Interface** pane in the Configuration Parameters dialog box or set the parameter `SupportContinuousTime` to `off`.
The option to generate code for noninlined S-functions is selected. This option requires support of nonfinite numbers, which is inappropriate for real-time safety-related systems.	Clear **Support: non-inlined S-functions** (Embedded Coder) in the Configuration Parameters dialog box or set the parameter `SupportNonInlinedSFcns` to `off`.
The option to generate model function calls compatible with the main program module of the pre-R2012a GRT target is selected. This option is inappropriate for real-time safety-related systems.	Clear **Classic call interface** (Simulink Coder) on the **Code Generation** > **Interface** pane in the Configuration Parameters dialog box or set the parameter `GRTInterface` to `off`.
The option to generate the *`model`*`_update` function is cleared. Having a single call to the output and update functions simplifies the interface to the real-time operating system (RTOS) and simplifies verification of the generated code.	Select **Single output/update function** (Simulink Coder) on the **Code Generation** > **Interface**pane in the Configuration Parameters dialog box or set the parameter `CombineOutputUpdateFcns` to `on`.
The option to generate the *`model`*`_terminate` function is selected. This function deallocates dynamic memory, which is unsuitable for real-time safety-related systems.	Clear **Terminate function** (Embedded Coder) on the **Code Generation** pane in the Configuration Parameters dialog box or set the parameter `IncludeMdlTerminateFcn` to `off`.
The option to log or monitor error status is cleared. If you do not select this option, the Simulink Coder product generates extra code that might not be reachable for testing.	Select **Remove error status field in real-time model data structure** (Embedded Coder) on the **Code Generation** > **Interface** pane in the Configuration Parameters dialog box or set the parameter `SuppressErrorStatus` to `on`.

Condition	Recommended Action
MAT-file logging is selected. This option adds extra code for logging test points to a MAT-file, which is not supported by embedded targets. Use this option only in test harnesses.	Clear **MAT-file logging** (Simulink Coder) in the Configuration Parameters dialog box or set the parameter `MatFileLogging` to `off`.

Action Results

Clicking **Modify Settings** configures model code generation settings that can impact safety.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0039: Configuration Parameters > Code Generation > Interface
- “Model Configuration Parameters: Comments” (Simulink Coder)
- “Model Configuration Parameters: Code Generation Identifiers” (Simulink Coder)
- “Model Configuration Parameters: Code Generation Interface” (Simulink Coder)
- “Model Configuration Parameters: Code Style” (Embedded Coder)

Check safety-related solver settings for simulation time

Check ID: `mathworks.hism.hisl_0040`

Check solver settings in the model configuration that apply to simulation time and might impact safety.

Description

This check verifies that the model solver configuration parameters pertaining to simulation time are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The solver setting to specify the start time for the simulation or generated code is set to a value other than `0.0`.	In the Configuration Parameters dialog box, set “Start time” or set the parameter `StartTime` to `0.0`.

Condition	Recommended Action
The solver setting to specify the stop time for the simulation or generated code is set to a negative value or a positive value greater than the value of “Application lifespan (days)”. By default, “Application lifespan (days)” is `auto`. If you do not change this setting, any positive value for “Stop time” is valid.	In the Configuration Parameters dialog box, set “Stop time” or set the parameter `StopTime` to a positive value that is less than the value of “Application lifespan (days)”.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to solvers and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.
- Analyzes content in masked subsystems.

See Also

- hisl_0040: Configuration Parameters > Solver > Simulation time

Check safety-related solver settings for solver options

Check ID: `mathworks.hism.hisl_0041`

Check solver settings in the model configuration that apply to solvers and might impact safety.

Description

This check verifies that the model solver configuration parameters pertaining to solvers are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The solver setting to specify the type of solver to simulate model is set to `Variable-step`.	In the Configuration Parameters dialog box, set “Type” or set the parameter `SolverType` to `Fixed-step`.
The solver setting to specify the solver to compute the states of the model during simulation or code generation is set to a value other than `Discrete(no continuous states)`.	In the Configuration Parameters dialog box, set “Solver” to `discrete(no continuous states)` or set the parameter `Solver` to `FixedStepDiscrete`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to solvers and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.
- Analyzes content in masked subsystems.

See Also

- hisl_0041: Configuration Parameters > Solver > Solver options

Check safety-related solver settings for tasking and sample-time

Check ID: `mathworks.hism.hisl_0042`

Check solver settings in the model configuration that apply to periodic sample time constraints and might impact safety.

Description

This check verifies that model configuration parameters are set optimally to ensure that the model operates at a specific set of prioritized periodic sample times for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Configuration parameter **Automatically handle rate transition for data transfer** is selected.	Clear **Automatically handle rate transition for data transfer** in the Configuration Parameters dialog box or set parameter `AutoInsertRateTranBlk` to `off`.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0042: Configuration Parameters > Solver > Tasking and sample time options

Check safety-related diagnostic settings for solvers

Check ID: `mathworks.hism.hisl_0043`

Check model configuration for diagnostic settings that apply to solvers and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to solvers are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic for detecting automatic breakage of algebraic loops is set to `none` or `warning`. The breaking of algebraic loops can affect the predictability of the order of block execution. For safety-related applications, a model developer needs to know when such breaks occur.	Set **Algebraic loop** on the **Diagnostics > Solver** pane in the Configuration Parameters dialog box or set the parameter `AlgebraicLoopMsg` to `error`. Consider breaking such loops explicitly with Unit Delay blocks so that the execution order is predictable. At a minimum, verify that the results of loops breaking automatically are acceptable.
The diagnostic for detecting automatic breakage of algebraic loops for Model blocks, atomic subsystems, and enabled subsystems is set to `none` or `warning`. The breaking of algebraic loops can affect the predictability of the order of block execution. For safety-related applications, a model developer needs to know when such breaks occur.	Set **Minimize algebraic loop** on the **Diagnostics > Solver** pane in the Configuration Parameters dialog box or set the parameter `ArtificialAlgebraicLoopMsg` to `error`. Consider breaking such loops explicitly with Unit Delay blocks so that the execution order is predictable. At a minimum, verify that the results of loops breaking automatically are acceptable.
The diagnostic for detecting potential conflict in block execution order is set to `none` or `warning`. For safety-related applications, block execution order must be predictable. A model developer needs to know when conflicting block priorities exist.	Set **Block priority violation** on the **Diagnostics > Solver** pane in the Configuration Parameters dialog box or set the parameter `BlockPriorityViolationMsg` to `error`.
The diagnostic for detecting whether the Simulink software automatically modifies the solver, step size, or simulation stop time is set to `none` or `warning`. Such changes can affect the operation of generated code. For safety-related applications, it is better to detect such changes so a model developer can explicitly set the parameters to known values.	Set **Automatic solver parameter selection** on the **Diagnostics > Solver** pane in the Configuration Parameters dialog box or set the parameter `SolverPrmCheckMsg` to `error`.
The diagnostic for detecting when a name is used for more than one state in the model is set to `none`. State names within a model should be unique. For safety-related applications, it is better to detect name clashes so a model developer can fix them.	Set **State name clash** on the **Diagnostics > Solver** pane in the Configuration Parameters dialog box or set the parameter `StateNameClashWarn` to `warning`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to solvers and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0043: Configuration Parameters > Diagnostics > Solver

Check safety-related diagnostic settings for sample time

Check ID: `mathworks.hism.hisl_0044`

Check model configuration for diagnostic settings that apply to sample time and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to sample times are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic for detecting when a source block, such as a Sine Wave block, inherits a sample time (specified as -1) is set to `none` or `warning`. The use of inherited sample times for a source block can result in unpredictable execution rates for the source block and blocks connected to it. For safety-related applications, source blocks should have explicit sample times to prevent incorrect execution sequencing.	Set **Source block specifies -1 sample time** on the **Diagnostics** > **Sample Time** pane in the Configuration Parameters dialog box or set the parameter `InheritedTslnSrcMsg` to `error`.
The diagnostic for detecting invalid data transfer between two blocks operating in multitasking mode is set to `none` or `warning`. Such data transfer should not be used for embedded real-time code.	Set **Multitask data transition** on the **Diagnostics** > **Sample Time** pane in the Configuration Parameters dialog box or set the parameter `MultiTaskRateTransMsg` to `error`.
The diagnostic for detecting subsystems that can cause data corruption or nondeterministic behavior is set to `none` or `warning`. This diagnostic detects whether conditionally executed multirate subsystems (enabled, triggered, or function-call subsystems) operate in multitasking mode. Such subsystems can corrupt data and behave unpredictably in real-time environments that allow preemption.	Set **Multitask conditionally executed subsystem** on the **Diagnostics** > **Sample Time** pane in the Configuration Parameters dialog box or set the parameter `MultiTaskCondExecSysMsg` to `error`.

Condition	Recommended Action
The diagnostic for checking sample time consistency between a Signal Specification block and the connected destination block is set to `none` or `warning`. An over-specified sample time can result in an unpredictable execution rate.	Set **Enforce sample times specified by Signal Specification blocks** on the **Diagnostics** > **Sample Time** pane in the Configuration Parameters dialog box or set the parameter `SigSpecEnsureSampleTimeMsg` to `error`.
The diagnostic detects that the parameter **Single task data transfer** is not set to `error`.	Set **Single task data transfer** in the Configuration Parameters dialog box or set the parameter `SingleTaskRateTransMsg` to `error`.
The diagnostic detects that the parameter **Tasks with equal priority** is not set to `error`.	Set **Tasks with equal priority** in the Configuration Parameters dialog box or set the parameter `TasksWithSamePriorityMsg` to `error`.
The diagnostic for detecting whether a model contains an S-function that has not been specified explicitly to inherit sample time is set to `none` or `warning`. These settings can result in unpredictable behavior. A model developer needs to know when such an S-function exists in a model so it can be modified to produce predictable behavior.	Set **Unspecified inheritability of sample time** in the Configuration Parameters dialog box or set parameter `UnknownTsInhSupMsg` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to sample time and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0044: Configuration Parameters > Diagnostics > Sample Time

Check safety-related optimization settings for logic signals

Check ID: `mathworks.hism.hisl_0045`

Check model configuration for optimization settings that can impact safety.

Description

This check verifies that model optimization configuration parameters are set optimally for generating code for a safety-related application. Although highly optimized code is desirable for most real-time systems, some optimizations can have undesirable side effects that impact safety.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Implementation of logic signals as Boolean data is cleared. Strong data typing is recommended for safety-related code.	Select Configuration Parameter **Implement logic signals as boolean data (vs. double)** or set the parameter `BooleanDataType` to `on`.

Action Results

Clicking **Modify Settings** configures model optimization settings that can impact safety.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0045: Configuration Parameters > Math and Data Types > Implement logic signals as Boolean data (vs. double)
- “Optimize Generated Code Using Minimum and Maximum Values” (Embedded Coder)

Check safety-related block reduction optimization settings

Check ID: `mathworks.hism.hisl_0046`

Check model configuration for optimization settings that can impact safety.

Description

This check verifies that model optimization configuration parameters are set optimally for generating code for a safety-related application. Although highly optimized code is desirable for most real-time systems, some optimizations can have undesirable side effects that impact safety.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Block reduction optimization is selected. This optimization can remove blocks from generated code, resulting in requirements without associated code and violations for traceability requirements.	Clear Configuration Parameter **Block reduction** or set parameter `BlockReduction` to `off`.

Action Results

Clicking **Modify Settings** configures model optimization settings that can impact safety.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0046: Configuration Parameters > Simulation Target > Block reduction

Check safety-related code generation settings for code style

Check ID: `mathworks.hism.hisl_0047`

Check model configuration for code generation settings that can impact safety.

Description

This check verifies that model configuration parameters for code generation are set optimally for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The option that specifies the style for parenthesis usage is set to `Minimum (Rely on C/C++ operators precedence)` or to `Nominal (Optimize for readability)`. For safety-related applications, explicitly specify precedence with parentheses.	Set parameter `ParenthesesLevel` to `Standards(Parentheses for Standards Compliance)` or `Maximum(Specify precedence with parentheses)`.
The option that specifies whether to preserve operand order is cleared. This option increases the traceability of the generated code.	Set parameter `PreserveExpressionOrder` to `on`.

Action Results

Clicking **Modify Settings** configures model code generation settings that can impact safety.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0047: Configuration Parameters > Code Generation > Code Style
- “Model Configuration Parameters: Comments” (Simulink Coder)
- “Model Configuration Parameters: Code Generation Identifiers” (Simulink Coder)
- “Model Configuration Parameters: Code Generation Interface” (Simulink Coder)
- “Model Configuration Parameters: Code Style” (Embedded Coder)

Check safety-related optimization settings for application lifespan

Check ID: `mathworks.hism.hisl_0048`

Check model configuration for optimization settings that can impact safety.

Description

This check verifies that model optimization configuration parameters are set optimally for generating code for a safety-related application. Although highly optimized code is desirable for most real-time systems, some optimizations can have undesirable side effects that impact safety.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model includes blocks that depend on elapsed or absolute time and is configured to minimize the amount of memory allocated for the timers. Such a configuration limits the number of days the application can execute before a timer overflow occurs. Many aerospace products are powered on continuously and timers should not assume a limited lifespan.	Set Configuration Parameter **Application lifespan (days)** or set the parameter `LifeSpan` to `inf`.

Action Results

Clicking **Modify Settings** configures model optimization settings that can impact safety.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0048: Configuration Parameters > Math and Data Types > Application lifespan (days)
- “Optimize Generated Code Using Minimum and Maximum Values” (Embedded Coder)

Check safety-related code generation identifier settings

Check ID: `mathworks.hism.hisl_0049`

Check model configuration for code generation settings that can impact safety.

Description

This check verifies that model configuration parameters for code generation are set optimally for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The minimum number of characters specified for generating name mangling strings is less than four. You can use this option to minimize the likelihood that parameter and signal names will change during code generation when the model changes. Use of this option assists with minimizing code differences between file versions, decreasing the effort to perform code reviews.	Set **Minimum mangle length** (Embedded Coder) on the **Code Generation** > **Identifier** pane in the Configuration Parameters dialog box or the parameter `MangleLength` to a value of 4 or greater.

Action Results

Clicking **Modify Settings** configures model code generation settings that can impact safety.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0049: Configuration Parameters > Code Generation > Identifiers
- “Model Configuration Parameters: Comments” (Simulink Coder)
- “Model Configuration Parameters: Code Generation Identifiers” (Simulink Coder)
- “Model Configuration Parameters: Code Generation Interface” (Simulink Coder)
- “Model Configuration Parameters: Code Style” (Embedded Coder)

Check safety-related optimization settings for data initialization

Check ID: `mathworks.hism.hisl_0052`

Check model configuration for optimization settings that can impact safety.

Description

This check verifies that model optimization configuration parameters are set optimally for generating code for a safety-related application. Although highly optimized code is desirable for most real-time systems, some optimizations can have undesirable side effects that impact safety.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The optimization that suppresses the generation of initialization code for root-level inports and outports that are set to zero is selected. For safety-related code, you should explicitly initialize all variables.	If you have an Embedded Coder license and are using an ERT-based system target file, clear Configuration Parameter **Remove root level I/O zero initialization** (Embedded Coder) or set the parameter `ZeroExternalMemoryAtStartup` to `on`. Alternatively, confirm that your compiler is standards-compliant, as most modern C compilers initialize global data to zero. Or, integrate external, handwritten code that initializes all I/O variables to zero explicitly.
The optimization that suppresses the generation of initialization code for internal work structures, such as block states and block outputs that are set to zero, is selected. For safety-related code, you should explicitly initialize every variable.	If you have an Embedded Coder license and are using an ERT-based system target file, clear Configuration Parameter **Remove internal data zero initialization** (Embedded Coder) or set the parameter `ZeroInternalMemoryAtStartup` to `on`. Alternatively, confirm that your compiler is standards-compliant, as most modern C compilers initialize global data to zero. Or, integrate external, handwritten code that initializes every state variable to zero explicitly.

Action Results

Clicking **Modify Settings** configures model optimization settings that can impact safety.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0052: Configuration Parameters > Optimization > Data initialization
- “Optimize Generated Code Using Minimum and Maximum Values” (Embedded Coder)

Check safety-related optimization settings for data type conversions

Check ID: `mathworks.hism.hisl_0053`

Check model configuration for optimization settings that can impact safety.

Description

This check verifies that model optimization configuration parameters are set optimally for generating code for a safety-related application. Although highly optimized code is desirable for most real-time systems, some optimizations can have undesirable side effects that impact safety.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The optimization that suppresses generation of code resulting from floating-point to integer conversions that wrap out-of-range values is cleared. You must avoid overflows for safety-related code. When this optimization is off and your model includes blocks that disable the **Saturate on overflow** parameter, the code generator wraps out-of-range values for those blocks. This can result in unreachable and, therefore, untestable code.	If you have a Simulink Coder license, select Configuration Parameter **Remove code from floating-point to integer conversions that wraps out-of-range values** (Simulink Coder) or set the parameter `EfficientFloat2IntCast` to `on`.

Action Results

Clicking **Modify Settings** configures model optimization settings that can impact safety.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0053: Configuration Parameters > Optimization > Remove code from floating-point to integer conversions that wraps out-of-range values

Check safety-related optimization settings for division arithmetic exceptions

Check ID: `mathworks.hism.hisl_0054`

Check model configuration for optimization settings that can impact safety.

Description

This check verifies that model optimization configuration parameters are set optimally for generating code for a safety-related application. Although highly optimized code is desirable for most real-time systems, some optimizations can have undesirable side effects that impact safety.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The optimization that suppresses generation of code that guards against division by zero for fixed-point data is selected. You must avoid division-by-zero exceptions in safety-related code.	If you have an Embedded Coder license and are using an ERT-based system target file, clear Configuration Parameter **Remove code that protects against division arithmetic exceptions** (Embedded Coder) or set the parameter `NoFixptDivByZeroProtection` to `off`.

Action Results

Clicking **Modify Settings** configures model optimization settings that can impact safety.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0054: Configuration Parameters > Optimization > Remove code that protects against division arithmetic exceptions
- “Optimize Generated Code Using Minimum and Maximum Values” (Embedded Coder)

Check safety-related optimization settings for specified minimum and maximum values

Check ID: `mathworks.hism.hisl_0056`

Check model configuration for optimization settings that can impact safety.

Description

This check verifies that model optimization configuration parameters are set optimally for generating code for a safety-related application. Although highly optimized code is desirable for most real-time systems, some optimizations can have undesirable side effects that impact safety.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The optimization that uses the specified minimum and maximum values for signals and parameters to optimize the generated code is selected. This might result in requirements without traceable code. (See DO-331 Section MB.6.3.4.e - Source code is traceable to low-level requirements.)	If you have an Embedded Coder license and are using an ERT-based system target file, clear Configuration Parameter **Optimize using the specified minimum and maximum values** (Embedded Coder), or parameter `UseSpecifiedMinMax` to `off`.

Action Results

Clicking **Modify Settings** configures model optimization settings that can impact safety.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- “Optimize Generated Code Using Minimum and Maximum Values” (Embedded Coder)
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178C Software Considerations in Airborne Systems and Equipment Certification and related standards
- hisl_0056: Configuration Parameters > Optimization > Optimize using the specified minimum and maximum values

Check Stateflow charts for uniquely defined data objects

Check ID: `mathworks.hism.hisl_0061`

Identify Stateflow charts that include data objects that are not uniquely defined.

Description

This check searches your model for local data in Stateflow charts that is not uniquely defined.

Available with Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
The Stateflow chart contains a data object identifier defined in two or more scopes.	For the identified chart, do one of the following: • Create a unique data object identifier within each of the scopes. • Create a unique data object identifier within the chart, at the parent level.

Capabilities and Limitations

- This check does not look for usage of unique identifiers for Simulink signals.
- Runs on library models.
- Does not analyze content of library linked blocks.
- Analyzes content in all masked subsystems.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0061: Unique identifiers for clarity

Check global variables in graphical functions

Check ID: `mathworks.hism.hisl_0062`

Description

This check Identifies the expressions that read and write to the same global data in a Stateflow.

This check requires Simulink Check and Stateflow licenses.

Results and Recommended Actions

Condition	Recommended Action
One or more expressions operate on graphical functions and global variables used within graphical functions.	Remodel the expressions so that the functions and the global variables are not used in the same expression.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- hisl_0062: Global variables in graphical functions

Check for length of user-defined object names

Check ID: `mathworks.hism.hisl_0063`

Description

This check inspects the length of the names of these user-defined objects against the Maximum Identifier length parameter in configuration settings:

- Subsystems with function name options set to User-specified.
- Data objects described in the guideline.
- Signal and parameter objects.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Length of the function name in a subsystem greater than the set threshold.	Change the function name in the Subsystem blocks to have a length less than the set threshold.
Data object names have a length greater than threshold.	Change the Data object names to have a length less than the set threshold.
Signal and Parameter names have a length greater than threshold.	Change the Signal and Parameter names to have a length less than the set threshold.

Capabilities and Limitations

- This check does not flag the signals that do not resolve to objects.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

See Also

- hisl_0063: Length of user-defined object names to improve MISRA C:2012 compliance

Check usage of Gain blocks

Check ID: `mathworks.hism.hisl_0066`

Description

This check identifies the Gain blocks with value that resolves to 1, an identity matrix, or a matrix of ones.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
One or more Gain blocks in the model resolve to 1 or an identity matrix.	Remodel the Gain blocks so that the gain value does not resolve to 1, an identity matrix, or a matrix of ones.

Capabilities and Limitations

- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- hisl_0066: Usage of Gain blocks

Check for divide-by-zero calculations

Check ID: `mathworks.hism.hisl_0067`

Description

Identifies the blocks that can result in divide-by-zero calculations.

Available with Simulink Check.

This check requires Simulink Design Verifier license.

Results and Recommended Actions

Condition	Recommended Action
One or more blocks in the model can result in divide-by-zero calculations.	Remodel to avoid divide-by-zero calculations.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

- Allows exclusions of blocks and charts.

See Also

- hisl_0067: Protect against divide-by-zero calculations

Check for model elements that do not link to requirements

Check ID: `mathworks.hism.hisl_0070`

Check whether Simulink model elements link to a requirements document.

Description

This check verifies whether model objects link to a document containing engineering requirements for traceability.

Available with Simulink Check.

This check requires a Requirements Toolbox license.

Check Parameterization

The following configuration can be set through Model Advisor Configuration Editor.

- The maximum number of requirement links per elements allowed can be set through the input parameter **Maximum number of requirement links per model elements**. By default, the value is set to **5**.
- Maximum number of child objects per linked component in Simulink allowed can be set through the input parameter **Maximum number of child objects per linked component in Simulink/ System Composer**. By default, the value is set to **100**.
- Maximum number of child objects per linked component in Stateflow allowed can be set through the input parameter **Maximum number of child objects per linked component in Stateflow**. By default, the value is set to **100**.
- Maximum number of lines of code per linked MATLAB function allowed can be set through the input parameter **Maximum number of lines of code per linked MATLAB function**. By default, the value is set to **200**.
- The model elements that can be exempted from requirement links can be set through the input parameter **List of model elements exempt from requirement links**.By default, the following list of blocks are included in the parameter. You can also add or remove the blocks as per your requirement.
  - Ground
  - Terminator
  - Inport
  - Outport
  - InportShadow
  - Goto
  - From

- Mux
- Demux
- CMBlock
- DocBlock
- System Requirements

Results and Recommended Actions

Condition	Recommended Action
Model elements do not link to a requirements document.	For each model element in the model, in the Model Editor, right-click the model element, select **Requirements** and specify a requirement.
Model elements have link requirements that exceed the set threshold.	make sure that model elements do not exceed the set threshold for maximum links to requirements.
Components in the model with links to requirements exceed the threshold for number of children.	Make sure that components with links to requirements do not exceed the threshold for number of children.

Capabilities and Limitations

- Runs on library models.
- Analyzes the content of library linked blocks. By default, the input parameter **Follow links** is set to `off`.
- Analyzes content in masked subsystems that have no workspaces and no dialogs. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.
- Does not allow the exclusion of Stateflow elements.
- This check verifies the requirements that are added to the MATLAB function headers or to the entire MATLAB function body only.
- The check report provides a link in the recommended actions tab for opening the Traceability Matrix.

Tip

Run this check from the top model or subsystem that you want to check.

See Also

- hisl_0070: Placement of requirement links in a model
- “Requirements Traceability” (Requirements Toolbox)
- Simulink Editor

Check safety-related settings for hardware implementation

Check ID: `mathworks.hism.hisl_0071`

Description

Identifies the inconsistencies or underspecifications of hardware attributes that can lead to non-optimal results.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
One or more parameters are not specified on the system.	Specify the parameters to ensure correct and efficient code generation for the target hardware.
One or more target specifications do not match.	Enable the parameter Test hardware is the same as production hardware or modify the target specifications to match.

Capabilities and Limitations

- Runs on library models.
- Allows exclusions of blocks and charts.

See Also

- hisl_0071: Configuration Parameters > Hardware Implementation >Inconsistent hardware implementation settings

Check data type of loop control variables

Check ID: `mathworks.hism.hisl_0102`

Description

This check identifies loop control variables using non-integer data types on the following:

- For iterator blocks.
- For loops in MATLAB function blocks.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
One or more For Iterator blocks are using non-integer data type for loop control counter variable.	Set the data type of loop control counter variable to an integer data type.
One or more For loops are using non-integer data type for loop control counter variable in MATLAB Function blocks.	Set the data type of loop control counter variable to an integer data type.

Capabilities and Limitations

- This check does not look at loop control variables inside Stateflow charts.
- Does not run on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- hisl_0102: Data type of loop control variables to improve MISRA C:2012 compliance

Check safety-related diagnostic settings for compatibility

Check ID: `mathworks.hism.hisl_0301`

Check model configuration for diagnostic settings that affect compatibility and that might impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to compatibility are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects when a block has not been upgraded to use features of the current release is set to `none` or `warning`. An S-function written for an earlier version might not be compatible with the current version and generated code could operate incorrectly.	Set **S-function upgrades needed** on the **Diagnostics** > **Compatibility** pane in the Configuration Parameters dialog box or set the parameter `SFcnCompatibilityMsg` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that affect compatibility and that might impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0301: Configuration Parameters > Diagnostics > Compatibility

Check safety-related diagnostic settings for parameters

Check ID: `mathworks.hism.hisl_0302`

Check model configuration for diagnostic settings that apply to parameters and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to parameters are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects when a parameter downcast occurs is set to `none` or `warning`. A downcast to a lower signal range can result in numeric overflows of parameters, resulting in unexpected behavior.	Set **Detect downcast** on the **Diagnostics** > **Data Validity** pane in the Configuration Parameters dialog box or set the parameter `ParameterDowncastMsg` to `error`.
The diagnostic that detects when a parameter underflow occurs is set to `none` or `warning`. When the data type of a parameter does not have enough resolution, the parameter value is zero instead of the specified value. This can lead to incorrect operation of generated code.	Set **Detect underflow** on the **Diagnostics** > **Data Validity** pane in the Configuration Parameters dialog box or set the parameter `ParameterUnderflowMsg` to `error`.
The diagnostic that detects when a parameter overflow occurs is set to `none` or `warning`. Numeric overflows can result in unexpected application behavior and should be detected and fixed in safety-related applications.	Set **Detect overflow** on the **Diagnostics** > **Data Validity** pane in the Configuration Parameters dialog box or set the parameter `ParameterOverflowMsg` to `error`.
The diagnostic that detects when a parameter loses precision is set to `none` or `warning`. Not detecting such errors can result in a parameter being set to an incorrect value in the generated code.	Set **Detect precision loss** on the **Diagnostics** > **Data Validity** pane in the Configuration Parameters dialog box or set the parameter `ParameterPrecisionLossMsg` to `error`.
The diagnostic that detects when an expression with tunable variables is reduced to its numerical equivalent is set to `none` or `warning`. This can result in a tunable parameter unexpectedly not being tunable in generated code.	Set **Detect loss of tunability** on the **Diagnostics** > **Data Validity** pane in the Configuration Parameters dialog box or set the parameter `ParameterTunabilityLossMsg` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to parameters and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0302: Configuration Parameters > Diagnostics > Data Validity > Parameters

Check safety-related diagnostic settings for Merge blocks

Check ID: `mathworks.hism.hisl_0303`

Check model configuration for diagnostic settings that apply to Merge blocks

Description

This check verifies that model configuration parameters are set optimally for Merge blocks for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects whether a model contains Merge blocks with more than one driving block executing at the same time step is set to `none` or `warning`.	In the Configuration Parameters dialog box, set “Detect multiple driving blocks executing at the same time step” or set the parameter `MergeDetectMultiDrivingBlocksExec` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to solvers and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- “hisl_0303: Configuration Parameters > Diagnostics > Data Validity > Merge blocks”

Check safety-related diagnostic settings for model initialization

Check ID: `mathworks.hism.hisl_0304`

In the model configuration, check diagnostic settings that affect model initialization and might impact safety.

Description

This check verifies that model diagnostic configuration parameters for initialization are optimally set to generate code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
In the Configuration Parameters dialog box, the "Underspecified initialization detection" diagnostic is set to `Classic`, ensuring compatibility with previous releases of Simulink. The "Check undefined subsystem initial output" diagnostic is cleared. This diagnostic specifies whether Simulink displays a warning if the model contains a conditionally executed subsystem, in which a block with a specified initial condition drives an Outport block with an undefined initial condition. A conditionally executed subsystem could have an output that is not initialized. If undetected, this condition can produce behavior that is nondeterministic.	Do one of the following: • In the Configuration Parameters dialog box, set **Underspecified initialization detection** to `Simplified`. • In the Configuration Parameters dialog box, set **Underspecified initialization detection** to `Classic` and select **Check undefined subsystem initial output**. • Set the parameter `CheckSSInitialOutputMsg` to `on`.
In the Configuration Parameters dialog box, the "Underspecified initialization detection" diagnostic is set to `Classic`, ensuring compatibility with previous releases of Simulink. This diagnostic detects potential initial output differences from earlier releases. A conditionally executed subsystem could have an output that is not initialized. If undetected, this condition can produce behavior that is nondeterministic.	Do one of the following: • In the Configuration Parameters dialog box, set **Underspecified initialization detection** to `Simplified`. • In the Configuration Parameters dialog box, set **Underspecified initialization detection** to `Classic`. • Set the parameter `CheckExecutionContextPreStartOutputMsg` to `on`.

Action Results

To configure the diagnostic settings that affect model initialization and might impact safety, click **Modify Settings**.

Subchecks depend on the results of the subchecks noted with **D** in the results table in the Model Advisor window.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- "hisl_0304: Configuration Parameters > Diagnostics > Data Validity > Model initialization"

Check safety-related diagnostic settings for data used for debugging

Check ID: `mathworks.hism.hisl_0305`

Check model configuration for diagnostic settings that apply to data used for debugging and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to debugging are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that enables model verification blocks is set to `Use local settings` or `Enable all`. Such blocks should be disabled because they are assertion blocks, which are for verification only. Model developers should not use assertions in embedded code.	In the Configuration Parameters dialog box, set **Model Verification block enabling** or set parameter `AssertControl` to `Disable All`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to data used for debugging and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- “hisl_0305: Configuration Parameters > Diagnostics > Data Validity > Debugging”

Check safety-related diagnostic settings for signal connectivity

Check ID: `mathworks.hism.hisl_0306`

Check model configuration for diagnostic settings that apply to signal connectivity and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to signal connectivity are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects virtual signals that have a common source signal but different labels is set to `none` or `warning`. This diagnostic pertains to virtual signals only and has no effect on generated code. However, signal label mismatches can lead to confusion during model reviews.	Set **Signal label mismatch** on the **Diagnostics** > **Connectivity** pane in the Configuration Parameters dialog box or set the parameter `SignalLabelMismatchMsg` to `error`.
The diagnostic that detects when the model contains a block with an unconnected input signal is set to `none` or `warning`. This must be detected because code is not generated for unconnected block inputs.	Set **Unconnected block input ports** on the **Diagnostics** > **Connectivity** pane in the Configuration Parameters dialog box or set the parameter `UnconnectedInputMsg` to `error`.
The diagnostic that detects when the model contains a block with an unconnected output signal is set to `none` or `warning`. This must be detected because dead code can result from unconnected block output signals.	Set **Unconnected block output ports** on the **Diagnostics** > **Connectivity** pane in the Configuration Parameters dialog box or set the parameter `UnconnectedOutputMsg` to `error`.
The diagnostic that detects unconnected signal lines and unmatched Goto or From blocks is set to `none` or `warning`. This error must be detected because code is not generated for unconnected lines.	Set **Unconnected line** on the **Diagnostics** > **Connectivity** pane in the Configuration Parameters dialog box or set the parameter `UnconnectedLineMsg` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to signal connectivity and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0306: Configuration Parameters > Diagnostics > Connectivity > Signals

Check safety-related diagnostic settings for bus connectivity

Check ID: `mathworks.hism.hisl_0307`

Check model configuration for diagnostic settings that apply to bus connectivity and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to bus connectivity are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects whether a Model block's root Outport block is connected to a bus but does not specify a bus object is set to `none` or `warning`. For a bus signal to cross a model boundary, the signal must be defined as a bus object for compatibility with higher level models that use a model as a reference model.	Set **Unspecified bus object at root Outport block** on the **Diagnostics** > **Connectivity** pane in the Configuration Parameters dialog box or set the parameter `RootOutportRequireBusObject` to `error`.
The diagnostic that detects whether the name of a bus element matches the name specified by the corresponding bus object is set to `none` or `warning`. This diagnostic prevents the use of incompatible buses in a bus-capable block such that the output names are inconsistent.	Set **Element name mismatch** on the **Diagnostics** > **Connectivity** pane in the Configuration Parameters dialog box or set the parameter `BusObjectLabelMismatch` to `error`.
The diagnostic that detects when some blocks treat a signal as a mux/vector, while other blocks treat the signal as a bus, is set to `none` or `warning`. When the Simulink software automatically converts a muxed signal to a bus, it is possible for an unintended operation or unpredictable behavior to occur.	Set **Bus signal treated as vector** on the **Diagnostics** > **Connectivity** pane in the Configuration Parameters dialog box to `error`, or the parameter `StrictBusMsg` to `ErrorOnBusTreatedAsVector`.
The diagnostic detects that the parameter **Non-bus signals treated as bus signals** is not set to `error`.	Set **Non-bus signals treated as bus signals** on the **Diagnostics** > **Connectivity** pane in the Configuration Parameters dialog box, or the parameter `NonBusSignalsTreatedAsBus` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to bus connectivity and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0307: Configuration Parameters > Diagnostics > Connectivity > Buses

Check safety-related diagnostic settings that apply to function-call connectivity

Check ID: `mathworks.hism.hisl_0308`

Check model configuration for diagnostic settings that apply to function-call connectivity and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to function-call connectivity are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that specifies whether the Simulink software has to compute inputs of a function-call subsystem directly or indirectly while executing the subsystem is set to `Use local settings` or `Disable all`. This diagnostic detects unpredictable data coupling between a function-call subsystem and the inputs of the subsystem in the generated code.	Set **Context-dependent inputs** on the **Diagnostics** > **Connectivity** pane in the Configuration Parameters dialog box or set the parameter `FcnCallInpInsideContextMsg` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to function-call connectivity and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0308: Configuration Parameters > Diagnostics > Connectivity > Function calls

Check safety-related diagnostic settings for type conversions

Check ID: `mathworks.hism.hisl_0309`

Check model configuration for diagnostic settings that apply to type conversions and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to type conversions are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects Data Type Conversion blocks when the type conversion is set to `none`. The Simulink software might remove unnecessary Data Type Conversion blocks from generated code, which might result in requirements without corresponding code. The removal of these blocks needs to be identified so model developers can explicitly remove the unnecessary blocks .	Set the **Unnecessary type conversions** Configuration Parameter or`UnnecessaryDatatypeConvMsg` parameter to `warning`.
The diagnostic that detects vector-to-matrix or matrix-to-vector conversions at block inputs is set to `none` or `warning`. When the Simulink software automatically converts between vector and matrix dimensions, unintended operations or unpredictable behavior can occur.	Set the **Vector/matrix block input conversion** Configuration Parameter or `VectorMatrixConversionMsg` parameter to `error`
The diagnostic that detects when a 32-bit integer value is converted to a floating-point value is set to `none`. This type of conversion can result in a loss of precision due to truncation of the least significant bits for large integer values.	Set the **32-bit integer to single precision float conversion** Configuration Parameter or `Int32ToFloatConvMsg` parameter to `warning`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to type conversions and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0309: Configuration Parameters > Diagnostics > Type Conversion

Check safety-related diagnostic settings for model referencing

Check ID: `mathworks.hism.hisl_0310`

Check model configuration for diagnostic settings that apply to model referencing and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to model referencing are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects port and parameter mismatches during model loading and updating is set to `none` or `warning`. If undetected, such mismatches can lead to incorrect simulation results because the parent and referenced models have different interfaces.	Set **Port and parameter mismatch** on the **Diagnostics > Model Referencing** pane in the Configuration Parameters dialog box or set the parameter `ModelReferenceIOMismatchMessage` to `error`.
The diagnostic that detects invalid internal connections to the current model's root-level Inport and Outport blocks is set to `none` or `warning`. When this condition is detected, the Simulink software might automatically insert hidden blocks into the model to fix the condition. The hidden blocks can result in generated code without traceable requirements. Setting the diagnostic to `error` forces model developers to fix the referenced models manually.	Set **Invalid root Inport/Outport block connection** on the **Diagnostics > Model Referencing** pane in the Configuration Parameters dialog box or set the parameter `ModelReferenceIOMessage` to `error`.
The diagnostic that detects whether To Workspace or Scope blocks are logging data in a referenced model is set to `none` or `warning`. Data logging is not supported for To Workspace and Scope blocks in referenced models.	Set **Unsupported data logging** on the **Diagnostics > Model Referencing** pane in the Configuration Parameters dialog box or set the parameter `ModelReferenceDataLoggingMessage` to `error`. To log data, remove the blocks and log the referenced model signals. For more information, see “Logging Referenced Model Signals”.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to model referencing and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0310: Configuration Parameters > Diagnostics > Model Referencing

Check safety-related diagnostic settings for Stateflow

Check ID: `mathworks.hism.hisl_0311`

Check safety-related diagnostic settings for Stateflow

Description

This check verifies that model configuration parameters are set optimally for Stateflow for a safety-related application.

Available with Simulink Check.

This check requires Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that detects whether a chart configuration leads to unwanted backtracking during simulation is set to `none` or `warning`.	In the Configuration Parameters dialog box, set "Unexpected backtracking" or set the parameter `SFUnexpectedBacktrackingDiag` to `error`.
The diagnostic that detects whether a chart configuration has blocks that connect to chart input ports do not initialize their outputs during initialization is set to `none` or `warning`.	In the Configuration Parameters dialog box, set "Invalid input data access in chart initialization" or set the parameter `SFInvalidInputDataAccessInChartInitDiag` to `error`.
The diagnostic that detects whether a chart has an unconditional default transition to a state or a junction is set to `none` or `warning`.	In the Configuration Parameters dialog box, set "No unconditional default transitions" or set the parameter `SFNoUnconditionalDefaultTransitionDiag` to `error`.
The diagnostic that detects whether a chart contains a transition that loops outside of the parent state or junction is set to `none` or `warning`.	In the Configuration Parameters dialog box, set "Transition outside natural parent" or set the parameter `SFTransitionOutsideNaturalParentDiag` to `error`.
The diagnostic that detects whether a chart is constructed on a valid execution path is set to `none` or `warning`.	In the Configuration Parameters dialog box, set "Unreachable execution path" or set the parameter `SFUnreachableExecutionPathDiag` to `error`.
The diagnostic detects that the parameter **Undirected event broadcasts** is set to `none` or `warning`.	Set **Undirected event broadcasts** in the Configuration Parameters dialog box or set the parameter `SFUndirectedBroadcastEventsDiag` to `error`.
The diagnostic detects that the parameter **Transition action specified before condition action** is set to `none` or `warning`.	Set **Transition action specified before condition action** in the Configuration Parameters dialog box or set the parameter `SFTransitionActionBeforeConditionDiag` to `error`.
The diagnostic that detects that the parameter **Read-before-write to output in Moore chart** is set to `none` or `warning`.	Set **Read-before-write to output in Moore chart** in the Configuration Parameters dialog box or set the parameter `SFOutputUsedAsStateInMooreChartDiag` to `error`.
The diagnostic detects that the parameter **Absolute time temporal value shorter than sampling period** is set to `none` or `warning`.	Set **Absolute time temporal value shorter than sampling period** in the Configuration Parameters dialog box or set the parameter `SFTemporalDelaySmallerThanSampleTimeDiag` to `error`.
The diagnostic detects that the parameter **Self transition on leaf state** is set to `none` or `warning`.	Set **Self transition on leaf state** in the Configuration Parameters dialog box or set the parameter `SFSelfTransitionDiag` to `error`.
The diagnostic detects that the parameter **Execute-at-Initialization disabled in presence of input events** is set to `none` or `warning`.	Set **Execute-at-Initialization disabled in presence of input events** in the Configuration Parameters dialog box or set the parameter `SFExecutionAtInitializationDiag` to `error`.

Condition	Recommended Action
The diagnostic detects that the parameter **Use of machine-parented data instead of Data Store Memory** is set to `none` or `warning`.	Set **Use of machine-parented data instead of Data Store Memory** in the Configuration Parameters dialog box or set the parameter `SFMachineParentedDataDiag` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to solvers and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0311: Configuration Parameters > Diagnostics > Stateflow

Check safety-related diagnostic settings for signal data

Check ID: `mathworks.hism.hisl_0314`

Check model configuration for diagnostic settings that apply to signal data and that can impact safety.

Description

This check verifies that model diagnostic configuration parameters pertaining to signal data are set optimally for generating code for a safety-related application.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The diagnostic that specifies how the Simulink software resolves signals associated with `Simulink.Signal` objects is set to `Explicit and implicit` or `Explicit and warn implicit`. For safety-related applications, model developers should be required to define signal resolution explicitly. (See DO-331, Section MB.6.3.3.b – Software architecture is consistent.)	Set **Signal resolution** on the **Diagnostics > Data Validity** pane in the Configuration Parameters dialog box or set the parameter `SignalResolutionControl` to `Explicit only`. This provides predictable operation by requiring users to define each signal and block setting that must resolve to `Simulink.Signal` objects in the workspace. Alternatively, to disable the use of `Simulink.Signal` objects, set the configuration parameter to `None`.

Condition	Recommended Action
The Product block diagnostic that detects a singular matrix while inverting one of its inputs in matrix multiplication mode is set to `none` or `warning`. Division by a singular matrix can result in numeric exceptions when executing generated code. This is not acceptable in safety-related systems. (See DO-331, Section MB.6.3.1.g – Algorithms are accurate, DO-331, Section MB.6.3.2.g – Algorithms are accurate, and MISRA C:2012, Dir 4.1.)	Set **Division by singular matrix** on the **Diagnostics** > **Data Validity** pane in the Configuration Parameters dialog box or set the parameter `CheckMatrixSingularityMsg` to `error`.
The diagnostic that detects when the Simulink software cannot infer the data type of a signal during data type propagation is set to `none` or `warning`. For safety-related applications, model developers must verify the data types of signals. (See DO-331, Section MB.6.3.1.e – High-level requirements conform to standards, and DO-331, Section MB.6.3.2.e – Low-level requirements conform to standards.)	Set **Underspecified data types** on the **Diagnostics** > **Data Validity** pane in the Configuration Parameters dialog box or set the parameter `UnderSpecifiedDataTypeMsg` to `error`.
The diagnostic that detects whether the value of a signal is too large to be represented by the signal data type is set to `none` or `warning`. Undetected numeric overflows can result in unexpected application behavior. (See DO-331, Section MB.6.3.1.g – Algorithms are accurate, DO-331, Section MB.6.3.2.g – Algorithms are accurate, and MISRA C:2012, Dir 4.1.)	Set **Wrap on overflow** on the **Diagnostics** > **Data Validity** pane in the Configuration Parameters dialog box or set the parameter `IntegerOverflowMsg` to `error`.
The diagnostic that detects whether the value of a signal is too large to be represented by the signal data type, resulting in a saturation, is set to `none` or `warning`. Undetected numeric overflows can result in unexpected application behavior. (See DO-331, Section MB.6.3.1.g – Algorithms are accurate, DO-331, Section MB.6.3.2.g – Algorithms are accurate, and MISRA C:2012, Dir 4.1.)	Set **Saturate on overflow** on the **Diagnostics** > **Data Validity** pane in the Configuration Parameters dialog box or set the parameter `IntegerSaturationMsg` to `error`.
The diagnostic that detects when the value of a block output signal is `Inf` or `NaN` at the current time step is set to `none` or `warning`. When this type of block output signal condition occurs, numeric exceptions can result, and numeric exceptions are not acceptable in safety-related applications. (See DO-331, Section MB.6.3.1.g – Algorithms are accurate, DO-331, Section MB.6.3.2.g – Algorithms are accurate, and MISRA C:2012, Dir 4.1.)	Set **Inf or NaN block output** on the **Diagnostics** > **Data Validity** pane in the Configuration Parameters dialog box or set the parameter `SignalInfNanChecking` to `error`.

Condition	Recommended Action
The diagnostic that detects Simulink object names that begin with `rt` is set to `none` or `warning`. This diagnostic prevents name clashes with generated signal names that have an `rt` prefix. (See DO-331, Section MB.6.3.1.e – High-level requirements conform to standards, and DO-331, Section MB.6.3.2.e – Low-level requirements conform to standards.)	Set **"rt" prefix for identifiers** on the **Diagnostics** > **Data Validity** pane in the Configuration Parameters dialog box or set the parameter `RTPrefix` to `error`.
The diagnostic that detects simulation range checking is set to `none` or `warning`. This diagnostic detects when signals exceed their specified ranges during simulation. Simulink compares the signal values that a block outputs with the specified range and the block data type. (See DO-331, Section MB.6.3.1.g – Algorithms are accurate, DO-331, Section MB.6.3.2.g – Algorithms are accurate, and MISRA C:2012, Dir 4.1.)	Set **Simulation range checking** on the **Diagnostics** > **Data Validity** pane in the Configuration Parameters dialog box or set the parameter `SignalRangeChecking` to `error`.

Action Results

Clicking **Modify Settings** configures model diagnostic settings that apply to signal data and that can impact safety.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- hisl_0314: Configuration Parameters > Diagnostics > Data Validity > Signals
- “Model Configuration Parameters: Data Validity Diagnostics”
- “View Diagnostics”

Check MATLAB functions not supported for code generation

Check ID: `mathworks.hism.himl_0012`

Description

This check identifies the MATLAB functions that are not supported for code generation.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
MATLAB functions that are not supported for code generation are in the model.	Avoid using MATLAB functions that are not supported for code generation.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- “himl_0012: Usage of MATLAB functions for code generation”

Metrics for generated code complexity

Check ID: `mathworks.hism.himl_0013`

Description

Identifies the usage of built-in MATLAB Functions with high code complexity of generated code.

Available with Simulink Check.

Input Parameters

For this check, you can set the following customizations using Model Advisor Configuration Editor:

- Set the threshold to flag MATLAB functions with high code complexity of generated code by using the **Complexity threshold** parameter. By default, the value is set to **40**.
- Exclude analyzing the MATLAB functions using the parameter **Functions excluded from analysis**. You can add multiple functions to the field by using a **comma** separator.

Results and Recommended Actions

Condition	Recommended Action
Some built-in MATLAB Functions used in the model might cause high code complexity when generating code.	• Functions with significant size and complexity must be reviewed to ensure if full potential of the function is required. • Use simpler alternatives to reduce code complexity, or add the functions to exclusion list in Model Advisor Configuration Editor.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- “himl_0013: Limitation of built-in MATLAB Function complexity”

Check for parameter tunability ignored for referenced models

Check ID: `mathworks.hism.hisl_0072`

Description

This check identifies the models parameter tunability information specified using Model Parameter Configuration dialog box.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Model contain parameter tunability information that Simulink ignores if the tunable parameters are defined using **Model Parameter Configuration** dialog box.	Use **Simulink.Parameter** object for defining the tunable parameters.

Capabilities and Limitations

- This check does not analyze models when the **Default parameter behavior** is set to `Tunable`.
- Runs on library models.
- Allows exclusions of blocks and charts.

See Also

- “hisl_0072: Usage of tunable parameters for referenced models”

Check usage of bit-shift operations

Check ID: `mathworks.hism.hisl_0073`

Description

Identifies blocks or expressions that perform bit-shift operations greater than the bit width of Input type that might result in violation of coding standards.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
One or more blocks or expressions in the model perform bit-shift operations greater than the bit width of Input type.	Remodel to avoid bit-shift operations greater than the bit width of Input type.

Capabilities and Limitations

- This check does not verify Expressions with signals as inputs. Such expressions must be verified manually.
- Runs on library models.
- Analyzes content in masked subsystems that have no workspaces and no dialogs. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes the content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- “hisl_0073: Usage of bit-shift operations”

Check safety-related diagnostic settings for variants

Check ID: `mathworks.hism.hisl_0074`

Description

Identifies the diagnostic settings in the model configuration that apply to variants.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The Configuration parameter **Variant condition mismatch at signal source and destination** in the model is not set to `Error`.	Set the Configuration parameters **Variant condition mismatch at signal source and destination** to `Error`.
The Configuration parameter **Arithmetic operations in variant conditions** in the model is not set to `Error`.	Set the Configuration parameters **Arithmetic operations in variant conditions** to `Error`.

Capabilities and Limitations

- Runs on library models.
- Allows exclusions of blocks and charts.

See Also

- “hisl_0074: Configuration Parameters > Diagnostics > Modeling issues related to variants”

Check usage of square root operations

Check ID: `mathworks.hism.hisl_0003`

Description

Identifies the square root operations with inputs that can be negative.

Available with Simulink Check. This check requires a Simulink Design Verifier (SLDV) license.

Note: This check will perform SLDV analysis on the model.

Results and Recommended Actions

Condition	Recommended Action
Square root operations in the model have inputs that can become negative during simulation.	Remodel to prevent the inputs of the square root operations from becoming negative.

Capabilities and Limitations

- Run on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.

See Also

- “hisl_0003: Usage of square root operations”

Check usage of Reciprocal Sqrt blocks

Check ID: `mathworks.hism.hisl_0028`

Description

Identifies Reciprocal Sqrt blocks with inputs that can become zero or negative.

Available with Simulink Check. This check requires a Simulink Design Verifier (SLDV) license.

Note: This check will perform SLDV analysis on the model.

Results and Recommended Actions

Condition	Recommended Action
One or more Reciprocal Sqrt blocks in the model have inputs that can become zero or negative during simulation.	Remodel to prevent the input of the Reciprocal Sqrt blocks from becoming zero or negative.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.

See Also

- “hisl_0028: Usage of Reciprocal Square Root blocks”

Check for disabled and parameterized library links

Check ID: `mathworks.hism.hisl_0075`

Description

Identifies the disabled and parameterized library links in the model.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
One or more blocks in the model are parameterized library links.	Use one of the following options to resolve the issue by right-clicking on the violated block and selecting an appropriate option from the **Library Link** menu: • Click **Push Link** to propagate the changes to the library. • Click **Select Restore Link** to discard the changes.
One or more blocks in the model are disabled library links.	Resolve the link using these steps: 1 Right-click the violated block in the Simulink diagram. 2 From the **Library Link** menu, select **Restore Link**.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- “hisl_0075: Usage of library links”

Check for unreachable and dead code

Check ID: `mathworks.hism.hisl_0101`

Description

Identifies the blocks and operations that results in unreachable and dead code.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
One or more blocks/operations in the model results in unreachable or dead code.	Remodel to protect against unreachable and dead code.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- “hisl_0101: Avoid operations that result in dead logic to improve code compliance”

Check for root Outports with missing properties

Check ID: `mathworks.hism.hisl_0077`

Description

Identifies the following:

- root level Outport blocks with missing or inherited sample times, data types or port dimensions for Simulink models.

  Using root model Outport blocks that do not have defined sample time, data types or port dimensions can lead to undesired simulation results. Simulink back-propagates dimensions, sample times, and data types from downstream blocks unless you explicitly assign these values. You can specify outport block properties with block parameters or Simulink signal objects that explicitly resolve to the connected signal lines.
- root level Output ports with missing or inherited data types or port dimensions for Architecture models.

When you run the check, a results table provides links to Outport blocks and signal objects that do not pass, along with conditions triggering the warning.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Missing port dimension — Model contains Outport blocks with inherited port dimensions.	For the listed Outport blocks and Simulink signal objects, specify port dimensions.
Missing signal data type — Model contains Outport blocks with inherited data types.	For the listed Outport blocks and Simulink signal objects, specify data types.
Missing port sample time — Model contains Outport blocks with inherited sample times.	For the listed Outport blocks and Simulink signal objects, specify sample times. The sample times for root Outports with bus type must match the sample times specified at the leaf elements of the bus object.
Implicit resolution to a Simulink signal object — Model contains Outport block signal names that implicitly resolve to a Simulink signal object in the base workspace, model workspace, or Simulink data dictionary.	For the listed Simulink signal objects, in the property dialog, select signal property **Signal name must resolve to Simulink signal object**. To set this option programmatically, use the port parameter **MustResolveToSignalObject**.
One or more Output ports of Architecture model do not have a data interface assigned to it.	Assign data interfaces to listed Output ports.

Capabilities and Limitations

- Allows exclusions of blocks and charts.
- Does not support exclusion in Architecture models.

See Also

- “hisl_0077: Outport interface definition”

Check type and size of condition expressions

Check ID: `mathworks.hism.himl_0011`

Description

This check evaluates the model to check that logical scalars are used for these condition expressions:

- `if` expressions
- `elseif` expressions
- `while` expressions
- Condition expressions of Stateflow transitions

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
`if` expression is not a logical scalar.	Change expression to a logical scalar.
`elseif` expression is not a logical scalar.	Change expression to a logical scalar.
`while` expression is not a logical scalar.	Change expression to a logical scalar.
Condition expression of Stateflow transition expression is not a logical scalar.	Change expression to a logical scalar.

Action Results

The results table identifies each expression that contains an expression that is not a logical scalar. To review the issue, select the corresponding function link from the result table. The Model Advisor opens and highlights the corresponding function definition or, for Stateflow, opens the chart and highlights the state in which the issue occurs.

Manually change the condition expressions to logical scalars. Save the model and rerun the check.

Capabilities and Limitations

- The conditions of Stateflow transitions and state actions are not checked when using C as the Stateflow action language.
- This check does not run on library models.
- Analyzes content in masked subsystems.
- Analyzes content of library-linked blocks.
- Unreachable invariant conditions and code fragments might not be analyzed.

See Also

- “himl_0011: Data type and size of condition expressions”
- “Represent Operating Modes by Using States” (Stateflow)
- “Transition Between Operating Modes” (Stateflow)
- “Differences Between MATLAB and C as Action Language Syntax” (Stateflow)

Check configuration parameters for MISRA C:2012

Check ID: `mathworks.misra.CodeGenSettings`

Identify configuration parameters that can impact MISRA C:2012 compliant code generation.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Math and Data Types	
Configuration parameter **Use division for fixed-point net slope computation** is not set to `On` or `Use division for reciprocals of integers only`.	Set **Use division for fixed-point net slope computation** to `On` or `Use division for reciprocals of integers only`.
Configuration parameter **Inf or NaN block output** is set to `None` or `error` and **Support non-finite numbers** is set to `on`. Configuration parameter **Inf or NaN block output** is set to `None` and **Support non-finite numbers** is set to `off`.	When **Support non-finite numbers** is: • `on`, set **Inf or NaN block output** to `warning` • `off`, set **Inf or NaN block output** to `warning` or `error`
Configuration parameter **Model Verification block enabling** is set to `Use local settings` or `Enable All`.	Set **Model Verification block enabling** to `Disable All`.
Configuration parameter **Undirected event broadcasts** is set to `none` or `warning`.	Set **Undirected event broadcasts** to `error`.
Configuration parameter **Wrap on overflow** is set to `None`	Set configuration parameter **Wrap on overflow** to `warning` or `error`.
Hardware Implementation	
Configuration parameter **Production hardware signed integer division rounds to** is set to `Undefined`	Set **Production hardware signed integer division rounds to** to `Zero` or `Floor`.
Configuration parameter **Shift right on a signed integer as arithmetic shift** is selected.	Clear **Shift right on a signed integer as arithmetic shift**.
Simulation Target	
Configuration parameter **Compile-time recursion limit for MATLAB functions** is set to a value other than `0`.	Set **Compile-time recursion limit for MATLAB functions** to `0`.
Configuration parameter **Dynamic memory allocation in MATLAB functions** is selected.	Clear **Dynamic memory allocation in MATLAB functions**.
Configuration parameter **Enable run-time recursion for MATLAB functions** is selected.	Clear **Enable run-time recursion for MATLAB functions**.
Code Generation	

Condition	Recommended Action
Configuration parameter **Bitfield declarator type specifier** is set to `uchar_T` when any of these parameters are selected: • **Pack Boolean data into bitfields** • **Use bitsets for storing state configuration** • **Use bitsets for storing Boolean data**	Set **Bitfield declarator type specifier** to `uint_T`.
Configuration parameter **Casting Modes** is not set to `Standards Compliant`.	Set **Casting Modes** to `Standards Compliant`.
Configuration parameter **Code replacement library** is not set to `None` or `AUTOSAR 4.0`.	Set **Code replacement library** to `None` or `AUTOSAR 4.0`
Configuration parameter **External mode** is selected.	Clear **External mode**.
Configuration parameter **Generate shared constants** is selected.	Clear **Generate shared constants**.
Configuration parameter **Include comments** is cleared.	Select **Include comments**.
Configuration parameter **MAT-file logging** is selected.	Clear **MAT-file logging**
For ERT-based target systems, configuration parameter **MATLAB user comments** is cleared.	Select **MATLAB user comments**.
A value for configuration parameter **Maximum identifier length** is not provided.	Set the value to the implementation-dependent limit. The default is `31`.
Configuration parameter **Parentheses level** is not set to `Standards(Parentheses for Standards Compliance)` or `Maximum(Specify precedence with parentheses)`.	Set **Parentheses level** to `Standards(Parentheses for Standards Compliance)` or `Maximum(Specify precedence with parentheses)`.
For ERT-based target systems, configuration parameter **Preserve static keyword in function declarations** is cleared when **File packaging format** is set to `Compact` or `Compact (with separate data file)`	Select **Preserve static keyword in function declarations**.
Configuration parameter **Replace multiplications by powers of two with signed bitwise shifts** is selected.	Clear **Replace multiplications by powers of two with signed bitwise shifts**.
Configuration parameter **Shared code placement** is set to `Auto`.	Set **Shared code placement** to `Shared location`
For ERT-based target systems, configuration parameter **Support continuous time** is selected	Clear **Support continuous time**.
For ERT-based target systems, configuration parameter **Support non-inlined S-functions** is selected	Clear **Support non-inlined S-functions**.

Condition	Recommended Action
Configuration parameter **System-generated identifiers** is set to `Classic`.	Set **System-generated identifiers** to `Shortened`.
Configuration parameter **System target file** is set to a GRT-based target.	Set **System target file** to an ERT-based target.
Configuration parameter **Use dynamic memory allocation for model initialization** is selected when **Code Interface Packaging** is set to `Reusable Function`.	Clear **Use dynamic memory allocation for model initialization**. **Note** Select only when **Code Interface Packaging** is set to `Reusable Function`.

Action Results

Clicking **Modify All** changes the parameter values to the recommended values.

Note When you click **Modify All** for models with a GRT-based target, the Model Advisor does not update the **System target file** configuration parameter to an ERT-based system.

Parameter subchecks depend on the results of the parameter noted with **D** in the results table. When the result is *D-Warning*, the **Current Value** column in the results table states *Prerequisite constraint not met* for the subchecks. After you change the parameter, rerun the check.

Note Some subchecks are specific to configuration parameters for ERT-based systems. These parameters are not updated when you click **Modify All** unless you change the model to an ERT-based system.

Capabilities and Limitations

Following parameters setting is informational in the check:

- `BooleansAsBitfields`
- `CodeInterfacePackaging`
- `ERTFilePackagingFormat`
- `SupportNonFinite`

This check does not review referenced models.

See Also

- hisl_0060: Configuration parameters that improve MISRA C:2012 compliance
- “MISRA C” (Embedded Coder)
- “MISRA C:2012 Compliance Considerations”

Check for blocks not recommended for MISRA C:2012

Check ID: `mathworks.misra.BlkSupport`

Identify blocks that are not supported or recommended for MISRA C:2012 compliant code generation.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Lookup Table blocks using cubic spline interpolation or extrapolation methods were found in the model or subsystem. Specific blocks are: • 1-D Lookup Table • 2-D Lookup Table • n-D Lookup Table	Consider other interpolation and extrapolation methods for the Lookup Table blocks.
Deprecated Lookup Table blocks were found in the model or subsystem. Specific blocks are: • Lookup Table • Lookup Table (2-D)	Consider replacing the deprecated Lookup Table blocks.
S-Function Builder blocks were found in the model or subsystem.	Consider replacing the S-Function Builder blocks with blocks recommended for production.
From Workspace blocks were found in the model or subsystem	Consider replacing the From Workspace blocks with blocks recommended for production.
String blocks were found in the model or subsystem. Specific blocks are: • Compose String • Scan String • String to Single • String to Double • To String	Consider replacing the String blocks with blocks recommended for production.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- “hisl_0020: Blocks not recommended for MISRA C:2012 compliance”
- “MISRA C” (Embedded Coder)
- “MISRA C:2012 Compliance Considerations”
- “Model Advisor Exclusion Overview”

IEC 61508, IEC 62304, ISO 26262, ISO 25119, and EN 50128/EN 50657 Checks

In this section...

IEC 61508, IEC 62304, ISO 26262, ISO 25119, and EN 50128/EN 50657 Checks

You can use these Model Advisor checks to facilitate designing and troubleshooting models, subsystems, and the corresponding generated code for applications to comply with IEC 61508-3, IEC 62304, ISO 26262-6, ISO 25119, or EN 50128, EN 50657, and MISRA C:2012 standards. They are certified by the IEC Certification Kit for use in development processes that must comply with IEC 61508, ISO 26262, EN 50128, EN 50657, ISO 25119, or derivative standards.

The Model Advisor performs a checkout of the Simulink Check license when you run the these checks.

Tips

If your model uses model referencing, run the IEC 61508, IEC 62304, ISO 26262, ISO 25119, or EN 50128/EN 50657 checks on all referenced models before running them on the top-level model.

Tips

If your model uses model referencing, run the IEC 61508, IEC 62304, ISO 26262 ISO 25119, or EN 50128/EN 50657 checks on all referenced models before running them on the top-level model.

See Also

- “Run Model Advisor Checks and Review Results”
- “Qualified Model Advisor Checks” (IEC Certification Kit)
- “Industry Standards” (Embedded Coder)

Display configuration management data

Check ID: `mathworks.iec61508.MdlVersionInfo`

Display model configuration and checksum information.

Description

This informer check displays the following information for the current model:

- Model version number
- Model author
- Date
- Model checksum

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Could not retrieve model version and checksum information.	This summary is provided for your information. No action is required.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- IEC 61508-3, Table A.8 (5) – Software configuration management
- IEC 62304-8 – Software configuration management process
- ISO 26262-8, Clause 7 – Configuration management
- EN 50128, Table A.9 (5) – Software Configuration Management
- “Manage Model Versions and Specify Model Properties”
- `Simulink.BlockDiagram.getChecksum` in the Simulink documentation
- `Simulink.SubSystem.getChecksum` in the Simulink documentation

Display model metrics and complexity report

Check ID: `mathworks.iec61508.MdlMetricsInfo`

Display number of elements and name, level, and depth of subsystems for the model or subsystem.

Description

The IEC 61508, ISO 26262, EN 50128, and EN 50657 standards recommend the usage of size and complexity metrics to assess the software under development. This check provides metrics information for the model. The provided information can be used to inspect whether the size or complexity of the model or subsystem exceeds given limits. The check displays:

- A block count for each Simulink block type contained in the given model, including library linked blocks.
- A count of Stateflow constructs in the given model (if applicable).
- Name, level, and depth of the subsystems contained in the given model (if applicable).
- The maximum subsystem depth of the given model.

Available with Simulink Check.

This check requires a Stateflow license.

Results and Recommended Actions

Condition	Recommended Action
N/A	This summary is provided for your information. No action is required.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.
- Does not allow exclusions of blocks or charts.

See Also

- IEC 61508-3, Table B.9 (1) - Software module size limit, Table B.9 (2) - Software complexity control
- IEC 62304, 5.5.3 - Software Unit acceptance criteria
- ISO 26262-6, Table 1 (1a) - Enforcement of low complexity, Table 3 (a) - Hierarchical structure of software components, Table 3 (b) - Restricted size of software components, and Table 3 (c) - Restricted size of interfaces
- EN 50128, Table A.12 (8) - Limited size and complexity of Functions, Subroutines and Methods and (9) Limited number of subroutine parameters
- EN 50657, Table A.12 (8) - Limited size and complexity of Functions, Subroutines and Methods and (9) Limited number of subroutine parameters
- `sldiagnostics` in the Simulink documentation
- “Cyclomatic Complexity for Stateflow Charts” (Simulink Coverage)

Check for unconnected objects

Check ID: `mathworks.iec61508.UnconnectedObjects`

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

Description

Unconnected objects are likely to cause problems propagating signal attributes such as data, type, sample time, and dimensions.

Ports connected to Ground or Terminator blocks pass this check.

Available with Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
There are unconnected lines, input ports, or output ports in the model or subsystem.	• Double-click an element in the list of unconnected items to locate the item in the model diagram. • Connect the objects identified in the results.

Capabilities and Limitations

- Runs on library models.
- Does not analyze content of library linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts.

See Also

- IEC 61508-3, Table A.3 (3) - Language subset
- IEC 62304, 5.5.3 - Software Unit acceptance criteria
- ISO 26262-6, Table 1 (1b) - Use of language subsets, Table 1 (1d) - Use of defensive implementation techniques
- EN 50128, Table A.4 (11) - Language Subset
- EN 50657, Table A.4 (11) - Language Subset
- “Signal Basics”

See Also

More About

- “Assess Requirements-Based Testing for ISO 26262”

Model Advisor Checks for MAB and JMAAB Compliance

Modeling Standards for MAB — Compliance Checks

You can use the Model Advisor to verify compliance of your model with the MathWorks Advisory Board (MAB) guidelines.

A Simulink Check license is required to execute these MAB checks. Where applicable, additional license requirements are identified in the check-specific documentation.

To access the checks in the Model Advisor, see “Accessing the MAB and JMAAB Model Advisor Checks”. To run the Model Advisor checks, see “Run Model Advisor Checks and Review Results”.

When executing the checks by using the **By Task** folder, MAB checks are classified into the following groups:

Naming Conventions

Checks that verify compliance of the model with MAB naming conventions guidelines.

For more information, see “Model Advisor Checks for MAB and JMAAB Guidelines”.

Simulink

Checks that verify compliance of the model with MAB guidelines for Simulink configuration parameters, diagram appearance, signals, block consistency, conditional subsystem relations, operation blocks, and other miscellaneous blocks.

For more information, see “Model Advisor Checks for MAB and JMAAB Guidelines”.

Stateflow

Checks that verify compliance of the model with MAB guidelines for Stateflow blocks/data/events, diagrams, conditional transition/Action, label descriptions, and other miscellaneous components.

For more information, see “Model Advisor Checks for MAB and JMAAB Guidelines”.

MATLAB

Checks that verify compliance of the model with MAB guidelines for MATLAB Functions.

For more information, see “Model Advisor Checks for MAB and JMAAB Guidelines”.

Modeling Standards for JMAAB — Compliance Checks

You can use the Model Advisor to verify compliance of your model with the Japan MATLAB Automotive Advisory Board (JMAAB) guidelines.

A Simulink Check license is required to execute these JMAAB checks. Where applicable, additional license requirements are identified in the check-specific documentation.

To access the checks in the Model Advisor, see “Accessing the MAB and JMAAB Model Advisor Checks”. To run the Model Advisor checks, see “Run Model Advisor Checks and Review Results”.

When executing the checks by using the **By Task** folder, JMAAB checks are classified into the following groups:

Naming Conventions

Checks related to the naming conventions.

For more information, see “Model Advisor Checks for MAB and JMAAB Guidelines”.

Model Architecture

Checks that verify compliance of the model with JMAAB model architecture guidelines.

For more information, see “Model Advisor Checks for MAB and JMAAB Guidelines”.

Model Configuration Options

Checks that verify compliance of the model with JMAAB configuration options guidelines.

For more information, see “Model Advisor Checks for MAB and JMAAB Guidelines”.

Simulink

Checks that verify compliance of the model with JMAAB guidelines for Simulink blocks and components.

For more information, see “Model Advisor Checks for MAB and JMAAB Guidelines”.

Stateflow

Checks that verify compliance of the model with JMAAB guidelines for Stateflow charts and components.

For more information, see “Model Advisor Checks for MAB and JMAAB Guidelines”.

MATLAB Functions

Checks that verify compliance of the model with JMAAB guidelines for MATLAB Functions.

For more information, see “Model Advisor Checks for MAB and JMAAB Guidelines”.

Check file names

Check ID: `mathworks.jmaab.ar_0001`

Description

Checks whether the file names meet the guideline standards.

Note

- This check only runs on the directory where the model is located. This behaviour does not change when the current directory is changed.
- This check does not run on the nested sub-directories.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub-IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b, c, d, e, f
- JMAAB — a, b, c, d, e, f

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
ar_0001_a: Characters allowed for file names	One or more files have invalid names.	Use alphanumeric characters and underscores in file names.
ar_0001_b: Number at the beginning	One or more files have numbers at the beginning of the file name.	Use alphabetic characters at the beginning of the file names.
ar_0001_c: Underscore at the beginning	One or more files have underscores at the beginning of the file name.	Use alphabetic characters at the beginning of the file names.
ar_0001_d: Underscore at the end	One or more files have underscores at the end of the file name.	Do not end the file names with underscores.
ar_0001_e: Consecutive underscores	One or more files have consecutive underscores in the file name.	Do not use consecutive underscores in the file names.
ar_0001_f: Single Reserved MATLAB word	One or more files have reserved MATLAB words as the file name.	Do not use reserved MATLAB word as the file name.
ar_0001_g: Identical file names on path	One or more files have names which are identical to files present in MATLAB path.	Use unique file names.

Capabilities and Limitations

- This check can be configured to run on the hidden folders by selecting the input parameter **Check hidden folders** in the Model Advisor Configuration Editor.
- This check can be configured to check the file names with specific extensions from the input parameter**File Extension** in the Model Advisor Configuration Editor.
- The check does not flag conflicts with C++ keywords.
- Runs on library models.

See Also

- MAB guideline ar_0001: Usable characters for file names
- JMAAB guideline ar_0001

Check folder names

Check ID: `mathworks.jmaab.ar_0002`

Description

Checks folder names to meet the guideline standards.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b, c, d, e, f
- JMAAB — a, b, c, d, e, f

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
ar_0002_a: Characters allowed for folder names	One or more folders have invalid names.	Use only alphanumeric characters and underscores in folder names.
ar_0002_b: Number at the beginning	One or more folders have numbers at the beginning of the folder name.	Use alphabetic characters at the beginning of the folder names.
ar_0002_c: Underscore at the beginning	One or more folders have underscores at the beginning of the folder name.	Use alphabetic characters at the beginning of the folder names.

Guideline Sub ID	Condition	Recommended Action
ar_0002_d: Underscore at the end	One or more folders have underscores at the end of the folder name.	Do not end the folder names with underscores.
ar_0002_e: Consecutive underscores	One or more folders have consecutive underscores in the folder name.	Do not use consecutive underscores in the folder names.
ar_0002_f: Single Reserved MATLAB word	One or more folders have reserved MATLAB words as the folder name.	Do not use reserved MATLAB word as the folder name.

Capabilities and Limitations

- You can configure this check to allow folder names starting with "+" or "@" using the input parameter **Allow MATLAB package** in the Model Advisor Configuration Editor.
- You can configure this check to run on the hidden folders by selecting the input parameter **Check hidden folders** in the Model Advisor Configuration Editor.
- Runs on library models.
- This check also runs on the nested sub directories.

See Also

- MAB guideline ar_0002: Usable characters for folder names
- JMAAB guideline ar_0002

Check length of model file name

Check ID: `mathworks.jmaab.jc_0241`

Description

Checks if the length of the model file name adheres to the maximum length restriction of 63 characters.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

You can configure the following input parameters to customize the check action:

Input Parameter	Value	JMAAB values (selected by default)
Minimum length	Minimum length for the folder path.	1
Maximum length	Maximum length for the folder path.	64

Results and Recommended Actions

Condition	Recommended Action
Model file name does not adhere to the length restriction of 63 characters.	Modify the model file name so that the length of the file name is not more than 63 characters.

See Also

- MAB guideline jc_0241: Length restriction for model file names
- JMAAB guideline jc_0241

Check length of folder name at every level of model path

Check ID: `mathworks.jmaab.jc_0242`

Description

Checks the length of the folder names at every level of the model path to see if all the folders in the path adhere to the maximum length restriction of 63 characters.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

You can configure the following input parameters to customize the check action:

Input Parameter	Value	JMAAB values (selected by default)
Minimum length	Minimum length for the folder path.	1
Maximum length	Maximum length for the folder path.	64

Input Parameter	Value	JMAAB values (selected by default)
Overall number of characters in path name	Total length (no of characters) for the folder path.	Disabled
Project Directory	Project path address.	Not configured

Results and Recommended Actions

Condition	Recommended Action
Length of the folder names at every level of the model path does not adhere to the length restriction of 63 characters.	Modify the folder names that do not meet the length restriction of 63 characters throughout the path.

See Also

- MAB guideline jc_0242: Length restriction for folder names
- JMAAB guideline jc_0242

Check Subsystem names

Check ID: `mathworks.jmaab.jc_0201`

Description

Identifies subsystem names with incorrect characters.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b, c, d, e, f
- JMAAB — a, b, c, d, e, f

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0201_a: Characters allowed for subsystem names	One or more subsystems have invalid names.	Use only alphanumeric characters and underscores in subsystem names.
jc_0201_b: Number at the beginning	One or more subsystems have numbers at the beginning of the subsystem name.	Use alphabetic characters at the beginning of the subsystem names.

Guideline Sub ID	Condition	Recommended Action
jc_0201_c: Underscore at the beginning	One or more subsystems have underscores at the beginning of the subsystem name.	Use alphabetic characters at the beginning of the subsystem names.
jc_0201_d: Underscore at the end	One or more subsystems have underscores at the end of the subsystem name.	Do not end the subsystem names with underscores.
jc_0201_e: Consecutive underscores	One or more subsystems have consecutive underscores in the subsystem name.	Do not use consecutive underscores in the subsystem names.
jc_0201_f: Single Reserved MATLAB word	One or more subsystems have reserved MATLAB words as the subsystem name.	Do not use reserved MATLAB word as the subsystem name.

Capabilities and Limitations

- This check is only applicable for Non-Virtual subsystems. If you want to run this check on Virtual Subsystems, select the input parameter **Check Virtual Subsystems** from the Model Advisor Configuration Editor.
- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Supports exclusions of blocks or charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0201: Usable characters for subsystem names
- JMAAB guideline jc_0201

Check character usage in block names

Check ID: `mathworks.jmaab.jc_0231`

Description

Identifies block names with incorrect characters.

Following are the special conditions the check verifies:

- If the block name is default block name and is hidden then the check will not report the block. This is because the block is a Simulink Library block.
- If the block name is default block name and is visible, then the check reports the block. This is because the block name now affects readability.

- If the block name is not default, and if it violates any conditions of the Sub-IDs, then the check reports even if it is visible or hidden.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b, c, d, e, f
- JMAAB — a, b, c, d, e, f

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0231_a: Characters allowed for block names	One or more blocks in the model have invalid names.	Use only alphanumeric characters and underscores in block name.
jc_0231_b: Number at the beginning	One or more blocks in the model have numbers at the beginning of the block name.	Use alphabetic characters at the beginning of the block name.
jc_0231_c: Underscore at the beginning	One or more blocks in the model have underscores at the beginning of the block name.	Use alphabetic characters at the beginning of the block name.
jc_0231_d: Underscore at the end	One or more blocks in the model have underscores at the end of the block name.	Do not end block names with underscores.
jc_0231_e: Consecutive underscores	One or more blocks in the model have consecutive underscores in the block name.	Do not use consecutive underscores in block name.
jc_0231_f: Single reserved MATLAB word	One or more blocks in the model use reserved MATLAB words as the block name.	Do not use reserved MATLAB word as block name.

Capabilities and Limitations

- Supports selection of Guideline Sub IDs.
- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Supports exclusions of blocks or charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0231: Usable characters for block names
- JMAAB guideline jc_0231

Check port block names

Check ID: `mathworks.jmaab.jc_0211`

Description

Identifies Inport or Outport block names with incorrect characters.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b, c, d, e, f
- JMAAB — a, b, c, d, e, f

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0211_a: Characters allowed for port block names	One or more port blocks have invalid names.	Use only alphanumeric characters and underscores in port block names.
jc_0211_b: Number at the beginning	One or more port blocks have numbers at the beginning of the port block name.	Use alphabetic characters at the beginning of the port block names.
jc_0211_c: Underscore at the beginning	One or more port blocks have underscores at the beginning of the port block name.	Use alphabetic characters at the beginning of the port block names.
jc_0211_d: Underscore at the end	One or more port blocks have underscores at the end of the port block name.	Do not end the port block names with underscores.
jc_0211_e: Consecutive underscores	One or more port blocks have consecutive underscores in the port block name.	Do not use consecutive underscores in the port block names.

Guideline Sub ID	Condition	Recommended Action
jc_0211_f: Single Reserved MATLAB word	One or more port blocks have reserved MATLAB words as the port block name.	Do not use reserved MATLAB word as the port block name.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Supports exclusions of blocks or charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0211: Usable characters for Inport blocks and Outport block
- JMAAB guideline jc_0211

Check length of subsystem names

Check ID: `mathworks.jmaab.jc_0243`

Description

Checks if the length of the subsystem names in the model adheres to the maximum length restriction of 63 characters.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

You can configure the following input parameters to customize the check action:

Input Parameter	Value	JMAAB values (selected by default)
Minimum length	Minimum length for the folder path.	1
Maximum length	Maximum length for the folder path.	64

Results and Recommended Actions

Condition	Recommended Action
Subsystem names in the model does not adhere to the length restriction of 63 characters.	Modify the subsystem block names so that the length of the subsystem name is not more than 63 characters.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0243: Length restriction for subsystem names
- JMAAB guideline jc_0243

Check length of block names

Check ID: `mathworks.jmaab.jc_0247`

Description

Checks if the length of the block names in the model adheres to the maximum length restriction of 63 characters.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

You can configure the following input parameters to customize the check action:

Input Parameter	Value	JMAAB values (selected by default)
Minimum length	Minimum length for the folder path.	1
Maximum length	Maximum length for the folder path.	64

Results and Recommended Actions

Condition	Recommended Action
Block names in the model does not adhere to the length restriction of 63 characters.	Modify the block names so that the length of the block names is not more than 63 characters.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0247: Length restriction for block names
- JMAAB guideline jc_0247

Check length of Inport and Outport names

Check ID: `mathworks.jmaab.jc_0244`

Description

Checks if the length of the inport and outport names adheres to the maximum length restriction of 63 characters.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

You can configure the following input parameters to customize the check action:

Input Parameter	Value	JMAAB values (selected by default)
Minimum length	Minimum length for the folder path.	1
Maximum length	Maximum length for the folder path.	64

Results and Recommended Actions

Condition	Recommended Action
Inport or outport block names in the Model does not adhere to the length restriction of 63 characters.	Modify the inport or the outport block names so that the length of the block name is not more than 63 characters.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0244: Length restriction for Inport and Outport names
- JMAAB guideline jc_0244

Check usable characters for signal names and bus names

Check ID: `mathworks.jmaab.jc_0222`

Description

Checks the signal and bus names in the model.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b, c, d, e, f
- JMAAB — a, b, c, d, e, f

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0222_a: Characters allowed for subsystem names	Signal or bus names in the model have invalid names.	Use only alphanumeric characters and underscores in signal names and bus names.
jc_0222_b: Number at the beginning	Signal or bus names in the model have numbers at the beginning of the name.	Use alphabetic characters at the beginning of the signal or bus names.
jc_0222_c: Underscore at the beginning	Signal or bus names in the model have underscores at the beginning of the name.	Use alphabetic characters at the beginning of the signal or bus names.
jc_0222_d: Underscore at the end	Signal or bus names in the model have underscores at the end of the signal or bus name.	Do not end the signal or bus names with underscores.
jc_0222_e: Consecutive underscores	Signal or bus names in the model have consecutive underscores in the name.	Do not use consecutive underscores in the signal or bus names.
jc_0222_f: Single Reserved MATLAB word	Signal or bus names in the model have reserved MATLAB words as the name.	Do not use reserved MATLAB word as the signal or bus names.

Capabilities and Limitations

- Runs on library models.
- Allows exclusions of blocks and charts.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

See Also

- MAB guideline jc_0222: Usable characters for signal and bus names
- JMAAB guideline jc_0222

Check usable characters for parameter names

Check ID: `mathworks.jmaab.jc_0232`

Description

Checks the parameter names in the model.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b, c, d, e, f
- JMAAB — a, b, c, d, e, f

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0232_a: Characters allowed for parameter names	The parameter name contains illegal characters.	Use only alphanumeric characters and underscores in parameter names.
jc_0232_b: Number at the beginning	The parameter name starts with a number.	Use alphabetic characters at the beginning of the parameter names.
jc_0232_c: Underscore at the beginning	The parameter name starts with an underscore ("_").	Use alphabetic characters at the beginning of the parameter names.
jc_0232_d: Underscore at the end	The parameter name ends with an underscore ("_").	Do not end parameter names with underscores.
jc_0232_e: Consecutive underscores	The parameter name has consecutive underscores.	Do not use consecutive underscores in the parameter names.
jc_0232_f: Single reserved MATLAB word	The parameter name is a reserved MATLAB words.	Do not use reserved MATLAB word as the parameter names.

Capabilities and Limitations

- Simulink Semantics limit the use of parameter names that use illegal characters, or start with a number, or start with a underscore, or start with a with reserved MATLAB words.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Supports exclusions of blocks or charts.

See Also

- MAB guideline jc_0232: Usable characters for parameter names
- JMAAB guideline jc_0232

Check length of signal and bus names

Check ID: `mathworks.jmaab.jc_0245`

Description

Checks if the length of the signal or bus names adheres to the maximum length restriction of 63 characters.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

You can configure the following input parameters to customize the check action:

Input Parameter	Value	JMAAB values (selected by default)
Minimum length	Minimum length for the folder path.	1
Maximum length	Maximum length for the folder path.	64

Results and Recommended Actions

Condition	Recommended Action
Signal or bus name in the model does not adhere to the length restriction of 63 characters.	Modify the signal or the bus names in the model so that the length of the names is not more than 63 characters.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0245: Length restriction for signal and bus names
- JMAAB guideline jc_0245

Check length of parameter names

Check ID: `mathworks.jmaab.jc_0246`

Description

Checks if the length of the parameter names in the model adheres to the maximum length restriction of 63 characters.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

You can configure the following parameters to customize the check action:

Input Parameter	Value	JMAAB values (selected by default)
Minimum length	Minimum length for the folder path.	1
Maximum length	Maximum length for the folder path.	64

Results and Recommended Actions

Condition	Recommended Action
Parameter names in the model does not adhere to the length restriction of 63 characters.	Modify the parameter names so that the length of the parameter names is not more than 63 characters.

Capabilities and Limitations

- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

- Does not support exclusions of blocks or charts.

See Also

- MAB guideline jc_0246: Length restriction for parameter name
- JMAAB guideline jc_0246

Check usable characters for Stateflow data names

Check ID: `mathworks.jmaab.jc_0795`

Description

Checks if the Stateflow data names in the model are using acceptable characters.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub-ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b, c, d
- JMAAB — a, b, c, d

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0795_a: Underscore at the beginning	The Stateflow data name starts with an underscore ("_").	Rename the Stateflow data.
jc_0795_b: Underscore at the end	The Stateflow data name ends with an underscore ("_").	Rename the Stateflow data.
jc_0795_c: Consecutive underscores	The Stateflow data name has consecutive underscores.	Rename the Stateflow data.
jc_0795_d: Single reserved MATLAB word	The Stateflow data name is solely a reserved MATLAB word.	Consider using a different name for Stateflow data.

Capabilities and Limitations

- JMAAB guideline, Version 5.1 limitation: This check does not flag the Stateflow data names with underscore at the beginning of the name. (Sub ID: a)
- Runs on library models.
- Allows exclusions of charts.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

See Also

- MAB guideline jc_0795: Usable characters for Stateflow data names
- JMAAB guideline jc_0795

Check length of Stateflow data name

Check ID: `mathworks.jmaab.jc_0796`

Description

Checks if the length of Stateflow data names are within the limit of 63 characters.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

You can customize the maximum length threshold by using the input parameter **Maximum length** from the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and search for check ID `jc_0796`.
2. From the **Standards** drop-down, select `Custom`.
3. Enter the desired maximum length to set in the **Maximum length** field.
4. Click **Apply** and save the configuration.

Results and Recommended Actions

Condition	Recommended Action
Length of Stateflow data names are not within the limit of 63 characters.	Consider using a different name for each Stateflow data name.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of charts.
- Analyzes content of library linked blocks.

See Also

- MAB guideline jc_0796: Length restriction for Stateflow data names
- JMAAB guideline jc_0796

Check duplication of Simulink data names

Check ID: `mathworks.jmaab.jc_0791`

Description

Checks for a duplicate definition of data names. Simulink data names must be unique across the base workspace, model workspace, and data dictionary.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub-ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b, c
- JMAAB — a, b, c

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0791_a: Check for repeated data names across base workspace and model workspace	Simulink data names are not unique across base workspace and model workspace.	Rename the repeated data name either in the base workspace or the model workspace.
jc_0791_b: Check for repeated data names across base workspace and data dictionary	Simulink data names are not unique across base workspace and data dictionary.	Rename the repeated data name either in the base workspace or the data dictionary.
jc_0791_c: Check for repeated data names across model workspace and data dictionary	Simulink data names are not unique across model workspace and data dictionary.	Rename the repeated data name either in the model workspace or the data dictionary.

Capabilities and Limitations

- Runs on library models.
- Does not allow exclusions of blocks and charts.

See Also

- MAB guideline jc_0791: Duplicate data name definitions

- JMAAB guideline jc_0791

Check unused data in Simulink Model

Check ID: `mathworks.jmaab.jc_0792`

Description

Identifies unused data in the model workspace and data dictionary.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub-ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b
- JMAAB — a, b

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0792_a: Check for unused data in Data Dictionary	One or more data variables in the data dictionary are unused.	Consider removing the unused data variables.
jc_0792_b: Check for unused data in Model Workspace	One or more data variables in the model workspace are unused.	Consider removing the unused data variables.

Capabilities and Limitations

- Does not run on library models.
- Allows exclusions of blocks or charts.

See Also

- MAB guideline jc_0792: Unused Data
- JMAAB guideline jc_0792

Check for unused data in Stateflow Charts

Check ID: `mathworks.jmaab.jc_0700`

Checks the state of the parameter **Unused data, events, messages and functions**.

Description

Identifies if the parameter **Unused data, events, messages and functions** is set to **None**. Unused data and events cannot exist in the Stateflow block.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
In a Stateflow block, the parameter **Unused data, events, messages and functions** is set to **None**.	Make sure to set the parameter to either **Warning** or **Error**.

Capabilities and Limitations

- Runs on library models.

See Also

- MAB guideline jc_0700: Unused data in Stateflow block
- JMAAB guideline jc_0700

Check usage of restricted variable names

Check ID: `mathworks.maab.na_0019`

Check for use of reserved keywords in MATLAB Function block variable names.

Description

Identifies variable names in MATLAB Function blocks that conflict with reserved C and C++ keywords. For a complete list of reserved keywords, see “Reserved Keywords” (Simulink Coder).

Avoid using variable names that conflict with MATLAB Functions, such as `conv`.

This check is case insensitive. For example, the check flags keywords `true`, `True`, `TRUE`, and `tRue`.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

You can use Model Advisor Configuration Editor to configure this check to include files with a `.m` extension in the analysis. To enable this feature, in the **Input Parameters** section, select **Check external .m files referenced in the model**.

Results and Recommended Actions

Condition	Recommended Action
Variable name conflicts with reserved keyword.	Consider using a different variable name that does not conflict with the reserved keywords.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline na_0019: Restricted variable names

Check Implement logic signals as Boolean data (vs. double)

Check ID: `mathworks.maab.jc_0011`

Check the optimization parameter for Boolean data types.

Description

Optimization for Boolean data types is required.

This check requires a Simulink Check license.

Note A prerequisite MAB guideline for this check is na_0002: Appropriate usage of basic logical and numerical operations.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Configuration setting for **Implement logic signals as Boolean data (vs. double)** is not set.	Select the **Implement logic signals as Boolean data (vs. double)** check box in the Configuration Parameters dialog box.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- MAB guideline jc_0011: Optimization parameters for Boolean data types
- JMAAB guideline jc_0011

Check Signed Integer Division Rounding mode

Check ID: `mathworks.jmaab.jc_0642`

Description

Identifies blocks whose parameter **Integer Rounding Mode** is set to **Simplest** when the configuration parameter **Signed Integer Division Rounds** is set to **Undefined**.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
If the parameter **Integer Rounding Mode** of the listed blocks is set to **Simplest** when the parameter **Signed Integer Division Rounds** is set to **Undefined**.	Set the parameter **Signed Integer Division Round** to a value that describes the rounding behavior of your production target or changing the **Integer Rounding Mode** of the listed blocks to a value other than **Simplest**.

Capabilities and Limitations

- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0642: Integer rounding mode setting
- JMAAB guideline jc_0642

Check diagnostic settings for incorrect calculation results

Check ID: `mathworks.jmaab.jc_0806`

Description

Identifies the status of the configuration parameters of the data validity diagnostic settings which detect incorrect calculation results.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline provides only one sub ID.

For reference, the MAB guideline Sub-ID(s) that NA-MAAB and JMAAB modeling standards organizations recommend are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
The model configuration parameter **Division by singular matrix** (CheckMatrixSingularityMsg) is not set to **error**.	Set the input parameter value to **error**.

Condition	Recommended Action
The model configuration parameter **Inf or NaN block output** (SignalInfNanChecking) is not set to **error**.	Set the input parameter value to **error**.
The model configuration parameter **Wrap on overflow** (IntegerOverflowMsg) is not set to **error**.	Set the input parameter value to **error**.
The model configuration parameter **Saturate on overflow** (IntegerSaturationMsg) is not set to **error**.	Set the input parameter value to **error**.

See Also

- MAB guideline jc_0806: Detecting incorrect calculation results
- JMAAB guideline jc_0806

Check model diagnostic parameters

Check ID: `mathworks.maab.jc_0021`

Check the model diagnostic configuration parameter settings.

Description

Model Advisor checks that these diagnostics configuration parameters are set as defined in the Results and Recommended Actions section below:

- “Algebraic loop”
- “Minimize algebraic loop”
- “Multitask data transfer”
- “Inf or NaN block output”
- “Duplicate data store names”
- “Unconnected block input ports”
- “Unconnected block output ports”
- “Unconnected line”
- “Unspecified bus object at root Outport block”
- “Element name mismatch”

This check requires a Simulink Check license.

Check Parameterization

This Model Advisor check is not applicable for JMAAB modeling guidelines.

This check does not include sub-checks.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — Not supported

Results and Recommended Actions

Condition	Recommended Action
Algebraic loop is set to none.	Set **Algebraic loop** to `error` or `warning`. Otherwise, Simulink might attempt to automatically break the algebraic loops, which can impact the execution order of the blocks.
Minimize algebraic loop is set to none.	Set **Minimize algebraic loop** to `error` or `warning`. Otherwise, Simulink might attempt to automatically break the algebraic loops for reference models and atomic subsystems, which can impact the execution order for those models or subsystems.
Inf or NaN block output is set to none	Set **Inf or NaN block output** to `error` or `warning`. Otherwise, numerical exceptions occur in the generated code
Duplicate data store names is set to none.	Set **Duplicate data store names** to `error` or `warning`. Otherwise, non-unique variable names exist in the generated code.
Unconnected block input ports is set to none.	Set **Unconnected block input ports** to `error` or `warning`. Otherwise, code cannot be generated.
Unconnected block output ports is set to none.	Set **Unconnected block output ports** to `error` or `warning`. Otherwise, dead code results.
Unconnected line is set to none.	Set **Unconnected line** to `error` or `warning`. Otherwise, code cannot be generated.
Unspecified bus object at root Outport block is set to none.	Set **Unspecified bus object at root Outport block** to `error` or `warning`. Otherwise, the result is an unspecified interface when the model is referenced from another model.
Element name mismatch is set to none.	Set **Element name mismatch** to `error` or `warning`. Otherwise, the result is an unintended interface in the generated code.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks or charts.

See Also

- MAB guideline jc_0021: Model diagnostic settings

Check for Simulink diagrams using nonstandard display attributes

Check ID: `mathworks.maab.na_0004`

Check model appearance setting attributes.

Description

Model appearance settings are required to conform to the guidelines when the model is released.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
The toolbar is not visible.	Select **Modeling > Environment > Toolstrip**.
The status bar is not visible.	Select **Modeling > Environment > Status Bar**.
Sample Time > Colors is selected.	Clear **Debug > Information Overlays > Colors**.
Wide Nonscalar Lines is cleared.	Select **Debug > Information Overlays > Nonscalar Signals**.
Viewer Indicators is cleared.	Select **Debug > Information Overlays > Logging & Viewers**.
Testpoint Indicators is cleared.	Select **Debug > Information Overlays > Testpoint**.
Port Data Types is selected.	Clear • **Debug > Information Overlays > Base Data Type** and • **Debug > Information Overlays > Alias Data Type** .
Storage Class is selected.	Clear **Debug > Information Overlays > Storage Class Indicator**.
Signal Dimensions is selected.	Clear **Debug > Information Overlays > Signal Dimensions**.
Signal Data Ranges is selected.	Clear **Debug > Information Overlays > Signal Data Ranges**.
Model Browser is selected.	Clear **Modeling > Environment > Model Browser**.

Condition	Recommended Action
Sorted Execution Order is selected.	Clear **Debug > Information Overlays > Execution Order**.
Model Block Version is selected.	Clear **Debug > Information Overlays > Ref. Model Version**.
Model Block I/O Mismatch is selected.	Clear **Debug > Information Overlays > Ref. Model I/O Mismatch**.
Library Links is set `Disabled`, `User Defined`, or `All`.	Select **Debug > Information Overlays > Show All Links**.
Linearization Indicators is cleared.	Select **Debug > Information Overlays > Linearization Indicators**.
Block backgrounds are not white.	For each listed diagram, click the block and select **Format > Background** and select color from the drop-down list.
Block foregrounds are not black.	Select **Format > Foreground** and select color from the drop-down list.
Diagrams do not have white backgrounds.	Select **Format > Background** and select color from the drop-down list.
Diagrams do not have zoom factor set to 100%.	For each listed diagram, select **Modeling > Environment > Zoom > Normal View (100%)**.

Action Results

Clicking **Modify** updates the display attributes to conform to the guideline.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems that have no workspaces and no dialogs. By default, the input parameter **Look under masks** is set to `graphical`.
- Supports exclusions of blocks or charts.

See Also

- MAB guideline: na_0004: Simulink model appearance settings
- JMAAB guideline: na_0004

Check Model font settings

Check ID: `mathworks.jmaab.db_0043`

Description

Check for difference in font and font sizes.

For font, this check runs on the following Simulink and Stateflow elements:

Simulink elements:

- Block
- Signal
- Annotation

Stateflow Elements:

- State
- Box
- Simulink Functions
- Embedded MATLAB functions
- Annotations
- Truth Table
- Charts and Sub-charts
- Transitions

Available with Simulink Check.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b, c, d
- JMAAB — a, b, c, d

To customize the text elements in your model, use the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and search for check ID `db_0043`.
2. Use the lists under **Input Parameters** to customize the font elements in your Simulink models and Stateflow charts. Note that when you select `Default`, the check flags different fonts/styles/size that are used in your model.
3. Click **Apply** and save the configuration.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
db_0043_a: Check font and font style in Simulink block and signal names	Font settings of one or more Simulink blocks or signal names are different from input parameters.	Change font settings of block and signal names as per input parameters.

Guideline Sub ID	Condition	Recommended Action
db_0043_b: Check font size in Simulink block and signal names	Font settings of one or more Simulink blocks or signal names are different from input parameters.	Change font size of block and signal names as per input parameters.
db_0043_c: Check font and font style in Stateflow objects	Font settings of one or more Stateflow objects are different from input parameters.	Change font settings of Stateflow objects as per input parameters.
db_0043_d: Check font size in Stateflow objects	Font settings of one or more Stateflow objects are different from input parameters.	Change font size of Stateflow objects as per input parameters.

Capabilities and Limitations

- Runs on library models.
- Does not analyze content of library linked blocks.
- Analyzes content in masked subsystems that have no workspaces and no dialogs.
- Allows exclusions of blocks and charts.

Action Results

Click **Modify all Fonts** to change the font and font size of text elements in the model to the values selected in the input parameters.

For the input parameters, if you specify `Common`, clicking **Modify all Fonts** changes the font and font sizes of text elements in the model to the most commonly used fonts, font sizes, or font styles.

See Also

- MAB guideline db_0043: Model font and font size
- JMAAB guideline db_0043
- “Specify Fonts in Models”

Check whether block names appear below blocks

Check ID: `mathworks.maab.db_0142`

Check whether block names appear below blocks.

Description

Identifies and reports if the block name does not appear below the block.

Note For vertically oriented blocks, this check reports the block names if they are not placed at right side of the block.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Blocks have names that do not appear below the blocks.	Set the name of the block to appear below the blocks.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline db_0142: Position of block names in the Simulink documentation.
- JMAAB guideline db_0142

Check the display attributes of block names

Check ID: `mathworks.maab.jc_0061`

Check the display attributes of subsystem and block names.

Description

Display the name of a block or subsystem when it provides a description that is used to identify its function.

Otherwise, when the function of a block is easily identified from its visual appearance, do not display the name. Such blocks include:

- From
- Goto
- Ground

- MinMax
- Multiport Switch
- Product
- Relational Operator
- Switch
- Terminator
- Unit Delay
- Compare To Constant
- Compare To Zero
- Saturation
- ModelReference
- Logic
- Trigonometry
- Sum
- Merge

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

To customize the blocks and masks that are checked during the analysis, use the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and search for check ID `jc_0061`.
2. Under **Input Parameters**, select `Custom` from the **Standards** list.
3. Use the **Block Types list** table to delete or add a block and/or mask type.
4. Click **Apply** and save the configuration.

Results and Recommended Actions

Condition	Recommended Action
Name is displayed and obvious from the block appearance.	Hide the block name by selecting **Format** > **Hide Automatic Block Name**.

Condition	Recommended Action
Name is not descriptive. Specifically, the block name is: • Not obvious from the block appearance. • The default name appended with an integer.	Modify the block name to provide descriptive information, or hide the block name by selecting **Format > Hide Automatic Block Name**.
Name is descriptive and not displayed. Descriptive names are: • Provided for blocks that are not obvious from the block appearance. • Not a default name appended with an integer.	Modify the blocks to show the block name by deselecting **Format > Hide Automatic Block Name**.
Check does not evaluate my custom blocks and masks.	Use the Model Configuration Editor to add your custom checks and blocks.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0061: Display of block names
- JMAAB guideline jc_0061

Check for nondefault block attributes

Check ID: `mathworks.maab.db_0140`

Identify blocks that use nondefault block parameter values that are not displayed in the model diagram.

Description

Model diagrams should display block parameters that have values other than default values. One way of displaying this information is by using the **Block Annotation** tab in the Block Properties dialog box.

For a list of block parameter default values, see “Block-Specific Parameters”.

Tip If you use the `add_block` function with `'built-in/`*`blocktype`*`'` as a source block path name for Simulink built-in blocks, some default parameter values of some blocks are different from the defaults that you get if you added those blocks interactively by using Simulink.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

To customize the block parameters for this check, use the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and search for check ID `db_0140`.
2. Under **Input Parameters**, select `Custom` from the **Standard** drop-down.
3. Use the **List of Block Types** table to delete or add a parameter for the block.
4. Click **Apply** and save the configuration.

Results and Recommended Actions

Condition	Recommended Action
Block parameters that have values other than default values, and the values are not in the model display.	In the Block Properties dialog box, use the **Block Annotation** tab to add block parameter annotations.

Action Results

You can use the **Add non-default values into block annotation** button to add an annotation to the block display that specifies the nondefault block parameter that was flagged in the analysis. Rerun the check; the block is no longer flagged.

Capabilities and Limitations

- Only customizable for block-specific parameters, see “Block-Specific Parameters”.
- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems that have no workspaces and no dialog boxes.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline db_0140: Display of block parameters
- JMAAB guideline: db_0140

Check Model Description

Check ID: `mathworks.jmaab.jc_0603`

Description

Identifies the layers in a model that have inconsistent description formatting for the following model elements:

- Annotations
- Model Info Block
- DocBlock Block

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a, b

You can select the description block type and configure the description tags from the **Description Block Type** and **Description Tags** input parameters. The format for the **Description Tags** should be of a string type with comma separated tags.

By default, the **Description Block Type** is set to **Annotation** and the **Description Tags** parameter is **Input:,Description:,Output:**.

For example, considering the default values (Input:, Description:, Output:), each of the layer in the model should have the description format as following:

Input: <input information>

Description: <model description>

Output: <output information>

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0603_a: Check for layer description at each layer	One or more layers in the model do not have model descriptions.	Add model descriptions at all the layers in the model.
jc_0603_b: Check for consistent layer description	One or more layers in the model do not have consistent model description formatting.	Make sure to have consistent format for the model descriptions at all the layers in the model.

Capabilities and Limitations

- Runs on library models.

- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Supports exclusions of blocks or charts.

See Also

- MAB guideline jc_0603: Model description
- JMAAB guideline jc_0603

Check if blocks are shaded in the model

Check ID: `mathworks.jmaab.jc_0604`

Description

Checks if block shading is used in the model.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Block shading is turned **on**.	Consider turning **off** the **DropShadow** property in blocks for better readability.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- MAB guidelines jc_0604: Using block shadow

- JMAAB guideline jc_0604

Check for unconnected signal lines and blocks

Check ID: `mathworks.jmaab.db_0081`

Check whether model has unconnected input ports, output ports, or signal lines.

Description

Unconnected blocks and signal lines should be connected to Terminator or Ground blocks.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b
- JMAAB — a, b

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
db_0081_a: Check for unconnected signal lines	One or more signal lines in the model are not connected.	Connect the signal lines to the correct source or destination block. If the destination block is not known, use a Terminator or Ground block to terminate the line.
db_0081_b: Check for unconnected subsystems and basic blocks	One or more blocks in the model are not connected.	Connect the blocks to the correct source or destination block. If the destination block is not known, use a Terminator or Ground block to terminate the line.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline db_0081: Unconnected signals and blocks
- JMAAB guideline db_0081

Check signal line connections

Check ID: `mathworks.jmaab.db_0032`

Description

Checks if the Simulink signals that are intersecting and overlapping adhere to recommended guidelines.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a1/a2, b, c, e
- JMAAB — a1/a2, b, c, d, e

Note Subchecks db_0032_a1, db_0032_b, db_0032_c, db_0032_d, and db_0032_e are selected by default.

The input parameter **Signal crossing threshold (in %)** allows you to set a condition to fail the check if the set threshold is met. Signal crossing threshold is a percentage of signal violations to the total number of Signal Lines in the model.

Example:

If the **Signal crossing threshold (in %)** is set to 50%, then the check will only fail when number of violations crosses 50% of total number of Signal lines in the block diagram.

If the **Signal crossing threshold (in %)** is set to 0%, any single Violation will cause the Check to fail whereas, for 100% threshold, all Signal Lines in the model must be violating the guideline for the Check to fail.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
db_0032_a1: Check signal intersections	Simulink signals intersect each other.	Make sure that the signals do not intersect with each other.

Guideline Sub ID	Condition	Recommended Action
db_0032_a2: Check if Line crossing style is set to "Line hop"	Line crossing style preference is not set to "Line hop".	Set Line crossing style preference to "Line hop".
db_0032_b: Check signal overlaps	Simulink signals overlap each other.	Make sure that the signals do not overlap with each other.
db_0032_c: Check if signals are intersecting blocks</entry>	Simulink signals are drawn over a Simulink block.	Make sure that the signals are not drawn over any Simulink blocks.
db_0032_d: Check if signal lines are split into multiple sublines	Signal lines are split into multiple sublines.	Reposition the signals to avoid splitting of signal lines.
db_0032_e: Check if signals are drawn as slanting lines	Signals are drawn as slanting lines in the diagram.	Make sure to draw signals as vertical or horizontal lines.

Capabilities and Limitations

- Signal hop preference is considered.
- Block label overlaps are not analyzed.
- Simulink signals that split into more than two signals at a single branch are considered.
- Runs on library models.
- Allows exclusions of subsystems.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline db_0032: Signal line connections
- JMAAB guideline db_0032

Check signal flow in model

Check ID: `mathworks.maab.db_0141`

Description

Identify subsystems with inappropriate signal flow. The signal flow in the models and subsystems should be from left to right.

This check requires a Simulink Check license.

Results and Recommended Actions

Guideline Sub-ID	Condition	Recommended Action
db_0141_a: Check placement of sequential block	The placement of blocks in subsystems is distorted.	Ensure that the signal flow in the mentioned subsystems is from left to right. All sequential blocks, except the blocks on feedback path, must be placed from left to right. All blocks, except the blocks on feedback path, should be oriented to the right.
db_0141_b: Check placement of parallel blocks	One or more groups of blocks are not arranged from top to bottom.	Arrange the groups of blocks or subsystems vertically from top to bottom.

Capabilities and Limitations

- This check does not verify the conditions in guideline sub-id C.
- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline db_0141: Signal flow in Simulink models
- JMAAB guideline db_0141

Check usage of tunable parameters in blocks

Check ID: `mathworks.maab.db_0110`

Check whether tunable parameters specify expressions, data type conversions, or indexing operations.

Description

To make a parameter tunable, you must enter the basic block without the use of MATLAB calculations or scripting. For example, omit:

- Expressions
- Data type conversions
- Selections of rows or columns

Supported blocks include:

- Backlash
- Bias
- Combinatorial Logic
- Constant
- Dead Zone
- Derivative
- Discrete-Time Integrator
- Gain
- Hit Crossing
- Initial Condition (IC)
- Integrator
- n-D Lookup Table
- Magnitude-Angle to Complex
- Memory
- Permute Dimensions
- Quantizer
- Rate Limiter
- Rate Transition
- Real-Imag to Complex
- Relay
- Saturation
- Sine
- State-Space
- Switch
- Transport Delay
- Unit Delay
- VariableTransportDelay
- lookup
- lookup2D

Available with Simulink Check.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Blocks have a tunable parameter that specifies an expression, data type conversion, or indexing operation.	In each case, move the calculation outside of the block, for example, by performing the calculation with a series of Simulink blocks, or pre-compute the value as a new variable.

Capabilities and Limitations

- This check does not flag if block parameters consist of expressions that contain literals.
- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.
- Does not evaluate mask parameters.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline db_0110: Block parameters
- JMAAB guideline db_0110

Check connections between structural subsystems

Check ID: `mathworks.jmaab.jc_0171`

Description

Checks the signal flow when using Goto and From blocks.

This check identifies the subsystems connected to each other that use Goto and From blocks in feed-forward and feedback loops that do not have at least one signal line for each direction.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a, b

You can configure the check to run on the Variant Subsystems, and Conditionally Executed Subsystems by selecting the **Variant Subsystems**,and **Conditionally Executed Subsystems** options respectfully from the Model Advisor Configuration Editor. By default, these options are enabled.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0171_a: Check direct connections between structural subsystems	One or more structural subsystems do not have direct connections between them.	Consider making at least one direct connection between the listed subsystems in the given direction (Subsystem 1 to Subsystem 2) with the exception of memory, delay and bus creator/selector blocks.
jc_0171_b: Check for unused inputs inside structural subsystems	One or more structural subsystems have unused inputs.	Consider removing the unused inputs inside the structural subsystems. Reducing the number of unnecessary connections makes connection relationships clearer.

Capabilities and Limitations

- This check does not report issues with guideline sub-id b (jc_0171_b: Check for unused inputs inside structural subsystems) for Charts and MATLAB Function blocks.
- The check allows the connections made from the referenced subsystems within another subsystem.
- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0171: Clarification of connections between structural subsystems
- JMAAB guideline jc_0171

Check for consistency in model element names

Check ID: `mathworks.jmaab.jc_0602`

Description

Checks if the model elements connected to a signal are following consistent naming.

The following names must be matched exactly when directly connected with signal lines:

- Inport block name
- Outport block name
- Structural subsystem input port label name

- Structural subsystem output port label name
- From tag name
- Goto tag name
- Signal line signal name

Exception 1: The name of a signal line connected to one of the below subsystems can have a different name to that of the subsystem port label name:

- Reusable subsystems
- Subsystems linked to a library

Exception 2: If a combination of Inport blocks, Outport blocks, and other blocks have the same block name, use a suffix or prefix for the Inport and Outport blocks for consistent naming. This can be configured through Model Advisor Configuration Editor.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

You can configure the following parameters to customize the check action:

Input Parameter	Value
Consistent naming format for combination of Inport and Outport blocks	Prefix(<Prefix_tag><Signal_name><Sequence>) or Suffix(<Signal_name><Sequence><Suffix_tag>)
Prefix/Suffix tag for Inports	IN_ (Default value, can be configured)
Prefix/Suffix tag for Outports	OUT_ (Default value, can be configured)

Results and Recommended Actions

Condition	Recommended Action
One or more model elements are not consistent with the connected signal name.	Consider renaming the deviating model elements to match the signal name or to be consistent with Inport/Outport blocks.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks or charts.

See Also

- MAB guideline jc_0602: Consistency in model element names
- JMAAB guideline jc_0602

Check trigger signal names

Check ID: `mathworks.jmaab.jc_0281`

Description

Identifies the trigger blocks where the origin of the trigger signal and the destination have similar names.

Note The check compares the names using **Levenshtein distance**.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a1/a2/a3/a4, b1/b2/b3/b4

Note Sub-checks jc_0281_a1 and jc_0281_b1 are selected by default.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0281_a1: Check names of the origin of the trigger signal and the conditional input block	The name of the block at the origin of the trigger signal and the conditional input block name at the destination are dissimilar.	The name of the block at the origin of the trigger signal and the conditional input block name at the destination must include the same name.
jc_0281_a2: Check names of the trigger signal and the conditional subsystem	The name of the block at the origin of the trigger signal and the conditional subsystem name at the destination are dissimilar.	The name of the block at the origin of the trigger signal and the conditional subsystem name at the destination must include the same name.

Guideline Sub ID	Condition	Recommended Action
jc_0281_a3: Check names of the origin of the trigger signal and the conditional input block	The name of the trigger signal and the conditional input block name at the destination are dissimilar.	The name of the trigger signal and the conditional input block name at the destination must include the same name.
jc_0281_a4: Check names of the trigger signal and the conditional subsystem	The name of the trigger signal and the conditional subsystem name at the destination must include the same name.	The name of the trigger signal and the conditional subsystem name at the destination are dissimilar.
jc_0281_b1: Check names of the origin of the trigger signal and the Stateflow block event	The name of the block at the origin of the trigger signal and the Stateflow block event name at the destination are dissimilar.	The name of the block at the origin of the trigger signal and the Stateflow block event name at the destination must include the same name.
jc_0281_b2: Check names of the origin of the trigger signal and the Chart name	The name of the block at the origin of the trigger signal and the Chart name at the destination are dissimilar.	The name of the block at the origin of the trigger signal and the Chart name at the destination must include the same name.
jc_0281_b3: Check names of trigger signal and the Stateflow block event	The name of the trigger signal and the Stateflow block event name at the destination are dissimilar.	The name of the trigger signal and the Stateflow block event name at the destination must include the same name.
jc_0281_b4: Check names of the trigger signal and the Chart	The name of the trigger signal and the Chart name at the destination are dissimilar.	The name of the trigger signal and the Chart name at the destination must include the same name.

Capabilities and Limitations

- This check flags Trigger and Enable block names only.
- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Supports exclusions of blocks or charts.

See Also

- MAB guideline jc_0281: Trigger signal names
- JMAAB guideline jc_0281

Check for mixing basic blocks and subsystems

Check ID: `mathworks.maab.db_0143`

Check for systems that mix primitive blocks and subsystems.

Description

You must design each level of a model with building blocks of the same type, for example, only subsystems or only primitive (basic) blocks. If you mask your subsystem and set MaskType to a nonempty string, the Model Advisor treats the subsystem as a basic block.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

To customize the blocks and masks that are checked during the analysis, use the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and search for check ID `db_0143`.
2. Under **Input Parameters**, select `Custom` from the **Standards** list.
3. In the **Treat blocktype list as** list, select `Allowed` to include the blocks/masks in the analysis or `Prohibited` to exclude the blocks/masks from the analysis.
4. Use the **Block Types list** table to delete or add a block and/or mask type.
5. Click **Apply** and save the configuration.

Results and Recommended Actions

Condition	Recommended Action
A level in the model includes subsystem blocks and primitive blocks.	Move nonvirtual blocks into the subsystem.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline db_0143: Usable block types in model hierarchy
- JMAAB guideline db_0143

Check for avoiding algebraic loops between subsystems

Check ID: `mathworks.jmaab.jc_0653`

Description

Checks the placement of the Delay blocks in algebraic loops between subsystems.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Delay blocks in feedback loops are placed inside subsystem.	Delay blocks in feedback loops must be placed outside of their parent subsystem.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0653: Delay block layout in feedback loops
- JMAAB guideline jc_0653

Check for prohibited sink blocks

Check ID: `mathworks.maab.hd_0001`

Check for prohibited Simulink sink blocks.

Description

You must design controller models from discrete blocks. Sink blocks, such as the Scope block, are not allowed in controller models.

This check requires a Simulink Check license.

Check Parameterization

This Model Advisor check is not applicable for JMAAB modeling guidelines.

This check does not include sub-checks

For reference, the MAB guideline sub-ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — Not supported

To customize the blocks and masks that are checked during the analysis, use the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and search for check ID `hd_0001`.
2. Under **Input Parameters**, select `Custom` from the **Standards** list.
3. In the **Treat blocktype list as** list, select `Allowed` to include the blocks/masks in the analysis or `Prohibited` to exclude the blocks/masks from the analysis.
4. Use the **Block Types list** table to delete or add a block and/or mask type.
5. Click **Apply** and save the configuration.

Results and Recommended Actions

Condition	Recommended Action
Sink blocks are not permitted in discrete controllers.	Remove sink blocks from the model.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline: hd_0001: Prohibited Simulink sinks

Check usage of vector and bus signals

Check ID: `mathworks.jmaab.na_0010`

Check usage of buses and Mux blocks.

Description

This check verifies the usage of buses and Mux blocks.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub-IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor configuration editor to specify which sub-IDs (one or multiple) to execute.

For reference, the MAB guideline sub-ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a, b, c, d

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
na_0010_a: Check usage of Mux and Demux blocks	Mux and Demux blocks are not used to generate and decompose vectors.	Use Mux and Demux blocks to generate and decompose vectors respectively.
na_0010_b: Check inputs to Mux blocks	Mux blocks have inputs which are not scalars or vectors.	Use only scalar or vector signals as inputs to Mux blocks.
na_0010_c: Check bus signals treated as vectors	One or more configuration parameters are set inappropriately.	Set the configuration parameters to the recommended value.
na_0010_d: Check usage of bus signals	One or more blocks are not supported for use with bus signals.	Use buses only with bus-supported blocks.

Capabilities and Limitations

- The check does not flag when blocks other than Demux and Mux are used to generate or decompose a vector. For example, a Selector block used instead of Demux or a Vector Concatenate used instead of Mux block. (Sub ID: a)
- This check supports auto fix mechanism, you can now click on the **Modify** button to fix the errors displayed in the report.
- Does not run on library models.
- Allows exclusions of blocks or charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline na_0010: Usage of vector and bus signals

- JMAAB guideline na_0010
- “Composite Interfaces”

Check signal line labels

Check ID: `mathworks.jmaab.jc_0008`

Check the labeling on signal lines.

Description

Use a label to identify:

- Signals originating from the following blocks (the block icon exception noted below applies to all blocks listed, except Inport, Bus Selector, Demux, and Selector):
  - Bus Selector block (tool forces labeling)
  - Chart block (Stateflow)
  - Constant block
  - Data Store Read block
  - Demux block
  - From block
  - Inport block
  - Selector block
  - Subsystem block

  Block Icon Exception If a signal label is visible in the display of the icon for the originating block, you do not have to display a label for the connected signal unless the signal label is required elsewhere due to a rule for signal destinations.
- Signals connected to one of the following destination blocks (directly or indirectly with a basic block that performs an operation that is not transformative):
  - Bus Selector block (tool forces labeling)
  - Chart block (Stateflow)
  - Data Store Write block
  - Goto block
  - Mux block
  - Outport block
  - Subsystem block
- Any signal of interest.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

To customize the list of block types to check output signal labels and list of block types to check input signal labels use the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and search for check ID `db_0140`.
2. Under **Input Parameters**, select `Custom` from the **Standard** drop-down.
3. Use the **List of block types to check output signal labels** table and **List of block types to check input signal labels** tables to delete or add a parameter for the block.
4. Click **Apply** and save the configuration.

Results and Recommended Actions

Condition	Recommended Action
Signals coming from Bus Selector, Chart, Constant, Data Store Read, Demux, From, Inport, or Selector blocks are not labeled.	Label the signal.

A warning might be thrown if you specify signal propagation from **Information Overlays** tab on the toolstrip.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions.

See Also

- MAB guideline jc_0008: Definition of signal names
- JMAAB guideline jc_0008
- “Signal Basics”

Check for propagated signal labels

Check ID: `mathworks.jmaab.jc_0009`

Check for propagated labels on signal lines.

Description

You should propagate a signal label from its source rather than enter the signal label explicitly (manually) if the signal originates from:

- An Inport block in a nested subsystem. However, if the nested subsystem is a library subsystem, you can explicitly label the signal coming from the Inport block to accommodate reuse of the library block.
- A basic block that performs a non-transformative operation.
- A Subsystem or Stateflow Chart block. However, if the connection originates from the output of an instance of the library block, you can explicitly label the signal to accommodate reuse of the library block.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a,b

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0009_a: Check Signal name propagation for subsystems	The model includes subsystems that do not have propagated signal labels.	Add labels and enable signal propagation by selecting **Show propagated signal** parameter for signals.
	One or more subsystems in the model display empty propagated signals.	Disable signal propagation by un-selecting **Show propagated signal** parameter, or if required, add a new label.
	One or more subsystems in the model do not display propagated signals but have signal names.	Remove labels and enable signal propagation by selecting **Show propagated signal** parameter for signals.
jc_0009_b: Check Signal name propagation for connection blocks	One or more connection blocks in the model do not propagate signals even when source block has labels.	Enable signal propagation by selecting **Show propagated signal** parameter for signals.
	One or more connection blocks in the model display empty propagated signals.	Disable signal propagation by un-selecting **Show propagated signal** parameter, or if required, add a new label.

Guideline Sub ID	Condition	Recommended Action
	One or more connection blocks in the model do not display propagated signals but have signal names.	Remove labels and enable signal propagation by selecting **Show propagated signal** parameter for signals.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions.

See Also

- MAB guideline jc_0009: Signal name propagation
- JMAAB guideline jc_0009
- “Signal Basics”

Check position of signal labels

Check ID: `mathworks.jmaab.db_0097`

Identify inappropriately placed signal labels for signals and buses.

Description

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b, c
- JMAAB — a, b, c

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
db_0097_a: Check overlap of signal labels	One or more signals in the model have labels which overlap other objects.	Place the signal labels so that it is readable.

Guideline Sub ID	Condition	Recommended Action
db_0097_b: Check position of signal labels	One or more signals in the model have labels placed at the top of signal line.	Place the signal labels underneath the signal lines.
db_0097_c: Check location of signal labels	One or more signals in the model do not have labels located at the origin of the signal line.	Place the signal labels at the origin of the signal line.

Capabilities and Limitations

- The modify action for this check currently addresses the issue by changing the location and correcting the flip format of the signal label. This action will not modify the position of the signal line to correct the overlap of signal labels.
- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content in all masked files.
- Support exclusions of blocks or charts.

See Also

- MAB guideline db_0097: Position of labels for signals and buses
- JMAAB guideline db_0097

Check signal line labels

Check ID: `mathworks.maab.na_0008`

Check the labeling on signal lines.

Description

Use a label to identify:

- Signals originating from the following blocks (the block icon exception noted below applies to all blocks listed, except Inport, Bus Selector, Demux, and Selector):
  - Bus Selector block (tool forces labeling)
  - Chart block (Stateflow)
  - Constant block
  - Data Store Read block
  - Demux block
  - From block
  - Inport block
  - Selector block

- Subsystem block

Block Icon Exception

- For Constant and Data Store Read blocks, if a signal label is visible in the display of the originating block, you do not have to display a label for the connected signal unless the signal label is required elsewhere due to a rule for signal destinations.
- For Subsystem, Chart, and From blocks, if a signal label is visible in the display of the originating block, and signal Propagation for the signal is enabled by signal property `Show propagated signals`, you do not have to display a label for the connected signal.

- Signals connected to one of the following destination blocks (directly or indirectly with a basic block that performs an operation that is not transformative):
  - Bus Selector block (tool forces labeling)
  - Chart block (Stateflow)
  - Data Store Write block
  - Goto block
  - Mux block
  - Outport block
  - Subsystem block
- Any signal of interest.

This check requires a Simulink Check license.

Check Parameterization

To customize the blocks and masks for this check, use the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and search for check ID `na_0008`.
2. Under **Input Parameters**, select `Custom` from the **Standards** list.
3. Add or delete the blocks and/or masks from the **List of block types to check output signal labels** and **List of block types to check input signal labels** tables.
4. Click **Apply** and save the configuration.

Results and Recommended Actions

Condition	Recommended Action
Signals coming from Bus Selector, Chart, Constant, Data Store Read, Demux, From, Inport, or Selector blocks are not labeled.	Label the signal.
Blocks from the list below that receive signals are not labeled: Outport, Goto, DataStore, BusCreator, Mux, or SubSystem.	Label the signal.

Capabilities and Limitations

- Runs on library models.

- Allows exclusions of blocks or charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline: na_0008: Display of labels on signals
- “Signal Names and Labels”.

Check for propagated signal labels

Check ID: `mathworks.maab.na_0009`

Description

Check for propagated labels on signal lines.

You should propagate a signal label from its source rather than enter the signal label explicitly (manually) if the signal originates from:

- An Inport block in a nested subsystem. However, if the nested subsystem is a library subsystem, you can explicitly label the signal coming from the Inport block to accommodate reuse of the library block.
- A basic block that performs a non-transformative operation (The output data type of these blocks remains the same as input).
- A Subsystem or Stateflow Chart block. However, if the connection originates from the output of an instance of the library block, you can explicitly label the signal to accommodate reuse of the library block.

This check requires a Simulink Check license.

Check Parameterization

This Model Advisor check is not applicable for JMAAB modeling guidelines.

This check does not include sub-checks

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — Not supported

Results and Recommended Actions

Condition	Recommended Action
The model includes signal labels that were entered explicitly, but should be propagated.	Use the open angle bracket (<) character to mark signal labels that should be propagated and remove the labels that were entered explicitly.

Capabilities and Limitations

- This check should not be run on models for AUTOSAR.
- Does not run on library models.
- Analyzes content in masked subsystems that have no workspaces and no dialogs. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks or charts.

See Also

- MAB guideline: na_0009: Entry versus propagation of signal labels
- “Signal Names and Labels”.

Check block orientation

Check ID: `mathworks.jmaab.jc_0110`

Checks blocks with changed orientation.

Description

Identifies the blocks that are reversed or with rotated orientation. This check excludes Unit Delay and Delay blocks.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Some blocks in the model have rotated or reversed orientation.	Flip or rotate these blocks to be oriented toward the right.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0110: Direction of block
- JMAAB guideline jc_0110

Check Indexing Mode

Check ID: `mathworks.jmaab.db_0112`

Description

Identifies blocks and charts with inconsistent Indexing mode.

Blocks with either 0 or 1 based indexing:

- Assignment
- For Iterator Subsystem
- Find
- Multiport Switch
- Selector

Blocks with default 1 based indexing:

- MATLAB Function
- Fcn
- MATLAB System
- Truth Table
- State Transition Table
- Test Sequence

Note

- For Simulink, there are no blocks with default zero-based indexing.
- For Stateflow, if action language for a chart is set to MATLAB, it is `1` based indexing, if the action language is set to C it is `0` based indexing.

This check requires Simulink Check and Stateflow licenses.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a1/a2
- JMAAB — a1/a2

Note Sub-check db_0112_a1 is selected as the default

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
db_0112_a1: Check for Zero-Based Indexing Mode	One or more blocks or charts in the model have One-Based Indexing mode or Specify indices mode.	Consider remodeling by using Zero-Based Indexing.
db_0112_a2: Check for One-Based Indexing Mode.	One or more blocks or charts in the model have Zero-Based Indexing mode or Specify indices mode.	Consider remodeling by using One-Based Indexing.

Capabilities and Limitations

- Runs on library models.
- Allows exclusions.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline db_0112: Usage of index
- JMAAB guideline db_0112

Check if tunable block parameters are defined as named constants

Check ID: `mathworks.jmaab.jc_0645`

Description

Checks if the tunable block parameters are defined as named constants.

Block parameters that are targets of calibration must be defined as named constants. Except for these parameters:

- Initial value when set to **0**
- Increment, decrement value when set to **1**
- Arithmetic expressions

This table lists the Simulink blocks and respective parameters supported :

Blocks	Supported Parameters
Backlash	**BacklashWidth**,**InitialOutput**
Bias	**Bias**
Combinatorial Logic	**TruthTable**
Constant	**Value**
Dead Zone	**LowerValue**,**UpperValue**
Derivative	**LinearizePole**
Discrete-Time Integrator	**gainval**,**InitialCondition**,**UpperSaturationLimit**,**LowerSaturationLimit**
Gain	**Gain**
Hit Crossing	**HitCrossingOffset**
Initial Condition (IC)	**Value**
Integrator	**InitialCondition**,**UpperSaturationLimit**,**LowerSaturationLimit**,**AbsoluteTolerance**
n-D Lookup Table	
Magnitude-Angle to Complex	**ConstantPart**
Memory	**X0**
Permute Dimensions	**Order**
Quantizer	**QuantizationInterval**
Rate Limiter	**RisingSlewLimit**,**FallingSlewLimit**,**InitialCondition**
Rate Transition	**X0**
Real-Imag to Complex	**ConstantPart**
Relay	**OnSwitchValue**,**OffSwitchValue**,**OnOutputValue**,**OffOutputValue**
Saturation	**UpperLimit**,**LowerLimit**
Sine	**Amplitude**,**Bias**,**Frequency**,**Phase**,**Samples**,**Offset**

Blocks	Supported Parameters
State-Space	**A**,**B**,**C**,**D**,**X0**,**AbsoluteTolerance**
Switch	**Threshold**
Transport Delay	**DelayTime**,**InitialOutput**,**BufferSize**,**PadeOrder**
Unit Delay	**InitialCondition**

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Improper usage of tunable block parameter values.	Change the tunable block parameter literal values to named constants.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0645: Parameter definition for calibration
- JMAAB guideline jc_0645

Check for sample time setting

Check ID: `mathworks.jmaab.jc_0641`

Description

Check if the sample time property of a block is set to -1 (inherited).

Note The check will not flag Enable and Trigger ports if they are inside a subsystem. This is as the sample time setting cannot be edited when they are inside subsystem. They will be flagged only if they are at the model root.

Following are the exceptions for this check:

- Inport block
- Outport block
- Atomic subsystem
- Blocks with state variables, such as Unit Delay, Delay, and Memory blocks
- Signal conversion blocks, such as Data Type Conversion and Rate Transition blocks
- Blocks that do not have external inputs, such as Constant block
- Stateflow charts

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Sample time property of a block is not set to -1 (inherited).	Change the sample time to -1 (inherited).

Capabilities and Limitations

- This check will not flag Enable and Trigger ports if they are inside a subsystem.
- This check allows you to use the Auto-Fix option to update the sample time of the flagged blocks to -1(inherited).
- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0641: Sample time setting

- JMAAB guideline jc_0641

Check usage of fixed-point data type with non-zero bias

Check ID: `mathworks.jmaab.jc_0643`

Check blocks with whose output signal data type is fixed-point and bias is not zero.

Description

For blocks that have a fixed-point data type for their output signals, check that block parameter **Bias** is set to `0`.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
In the Data Type Assistant, **Mode** is set to `Fixed Point` but the value for **Bias** is not `0`.	Change block parameter **Bias** to `0`.

Capabilities and Limitations

- Does not run on library models.
- Supports exclusions.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0643: Fixed-point setting
- JMAAB guideline jc_0643

Check type setting by data objects

Check ID: `mathworks.jmaab.jc_0644`

Description

Identifies the blocks in Simulink that violate the type setting if the signal objects are used (if signal data type is set in signal object, then it must not be set on the block side).

This check exempts:

- Data type conversion block.
- Type setting using **fixdt**.
- Double and Boolean types.
- Reusable internal part of a function (atomic subsystem).
- Block output data type set to **Inherit via backpropagation**.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Signal data type is set to different types in signal objects and in the block.	Set the output data type of the blocks either to **auto** or **Inherit via back propagation**.

Capabilities and Limitations

- Does not run on library models.
- Allows exclusions of blocks and charts.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

See Also

- MAB guideline jc_0644: Type setting
- JMAAB guideline jc_0644

Check position of conditional blocks and iterator blocks

Check ID: `mathworks.jmaab.db_0146`

Check the position of Trigger and Enable blocks.

Description

Locate blocks that define subsystems as conditional or iterative at the top of the subsystem diagram.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b
- JMAAB — a, b

The default block position can be configured by using the parameter **Block Position** from the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and search for check ID `db_0146`.
2. Enter the position of the block to check in the **Block Position** field.

   By default, this parameter is set to **Top**.
3. Click **Apply** and save the configuration.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
db_0146_a: Block layout in conditional subsystem	Trigger, Enable, and Action Port blocks are not at the top of the subsystem diagram.	Move the Trigger, Enable, and Action Port blocks to the top of the subsystem diagram.
db_0146_b: Block layout in iterative subsystem	For Each, For Iterator, and While Iterator blocks are not in the same location on the subsystem diagram.	Move the For Each, For Iterator, and While Iterator blocks so they are at a uniform location on the subsystem diagram.

Capabilities and Limitations

- The Sub-ID `B` of this check `db_0146_b` does not flag the For Each Subsystem, For Iterator Subsystem, and While Iterator Subsystem blocks when they have Inports or Outport blocks.
- Runs on library models.
- Allows exclusions of blocks and charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline db_0146: Block layout in conditional subsystems
- JMAAB guideline db_0146

Check undefined initial output for conditional subsystems

Check ID: `mathworks.jmaab.jc_0640`

Description

Checks that the initial output value for all Outports and Merge blocks connected to a conditional subsystem are explicitly defined.

Note This check identifies Outport blocks and Merge blocks connected to Conditional subsystems in your model that can cause problems if you use the classic initialization mode.

To set classic initialization mode:

1. On the **Modeling** tab, in**Setup** section, select **Model Settings**.
2. In the Configuration Parameters dialog box, use the search box and enter **Underspecified initialization detection**.
3. From the drop-down list, select **Classic**.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
The initial output for all Outports and Merge blocks connected to a Conditional subsystem are not explicitly defined.	For a Conditional subsystem, explicitly define the initial output value for all Outports and Merge blocks connected to the Conditional subsystem.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0640: Initial value settings for Outport blocks in conditional subsystems
- JMAAB guideline jc_0640

Check usage of Merge block

Check ID: `mathworks.jmaab.jc_0659`

Identifies the blocks present in between a conditional subsystem and a merge block.

Description

Merge blocks must have direct connections from conditionally executed subsystems. While using a Merge block take the following into consideration:

- No blocks must be present in between the Merge and Conditionally executed subsystem blocks, including a virtual subsystem that does not affect the function of Merge block.
- The Merge block can be nested inside any number of subsystems, if the preceding condition is satisfied.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more blocks inserted in between a Merge and a Conditional Subsystem block.	Make direct connections from Conditional Subsystem blocks to Merge blocks.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0659: Usage restrictions of signal lines input to Merge blocks
- JMAAB guideline jc_0659

Check logical expressions in If blocks

Check ID: `mathworks.maab.na_0003`

Check If blocks for inappropriate construct of primary expressions in a logical expression.

Description

Identifies instances in an If block where primary expressions are complex.

Primary expressions are defined as:

- An input
- A constant
- A constant parameter
- A parenthesized expression containing no operators other than zero or <, >, >=, <=, ==, ~=, |, &, and ~

Examples of primary expressions include:

- `u1`
- `5`
- `K`
- `(u1 > 0)`
- `(u1 <= G)`
- `(u1 > U2)`
- `(~u1)`

Examples of acceptable logical expressions exceptions include:

- `u1 | u2`
- `((u1 > 0) & (u1 < 20)`
- `(u1 > 0) & (u2 < u3)`
- `(u1 > 0) & (~u2)`

This table provides examples of unacceptable logical expressions.

Primary Expression	Reasoning
`u1 & u2 | u3`	Too many primary expressions.

Primary Expression	Reasoning
`u1 & (u2 | u3)`	Unacceptable operator within primary expression.
`(u1 > 0) & (u1 < 20) & (u2 > 5)`	Too many primary expressions that are not inputs.
`(u1 > 0) & ((2* u2) > 6)`	Unacceptable operator within primary expression.

Exception

A logical expression can contain more than two primary expressions when both these conditions are met:

- The primary expressions are all inputs.
- Only one type of logical operator is present.

Examples of acceptable exceptions include:

- `u1 | u2 | u3 | u4 | u5`
- `u1 & u2 & u3 & u4`

Simple "If" Expressions

In the literal interpretation of guideline na_0003, expression `u1 < u2` is a violation. However, the expression follows the commonly used "If" expression template (<Primary Expression><Operator><Primary Expression>). So, when logical operators are not used and only one relational operator is present, the expression satisfies guideline na_0003 and `u1 < u2` is NOT a violation.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action

Logical expression contains more than two primary expressions that consist of a constant, constant parameter, and input.	Consider one of the following:
	• Make primary expressions an input and either:
	  • Use parenthesized expressions with one relational operator type
	  • Construct a simple "If" express using template <Primary Expression><Logical Operator><Primary Expression>
	• Reduce the number of primary expressions to two or less.
	• Construct the logical expression using logical blocks other than the If block.

Logical expression contains more than two parenthesized expressions that use multiple relational operators	Consider one of the following: • Use only one type of relational operator. Acceptable logical operators include <, >, >=, <=, ==, ~=, |, &, and ~. The primary expression must consist of inputs only. • Reduce the number of parenthesized expressions to two or less. • Construct the logical expression using logical blocks other than the If block.
Parenthesized expression includes a relational operator other than zero or <, >, >=, <=, ==, ~=, |, &, or ~.	Consider one of the following: • Use relational operator <, >, >=, <=, ==, ~=, |, &, or ~ within the parenthesized expression. • Construct the logical expression using logical blocks other than the If block.

Capabilities and Limitations

- Does not flag logical expressions that use only one of these relative operators <, >, >=, <=, ==, ~=, |, &, and ~
- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline na_0003: Usage of If blocks
- JMAAB guideline na_0003

Check default/else case in Switch Case blocks and If blocks

Check ID: `mathworks.jmaab.jc_0656`

Description

Checks the **default/else** case in Switch Case blocks and If blocks.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Improper usage of Switch and If blocks.	Consider setting the **default/else** case option in Switch Case blocks and If blocks to **on**.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0656: Usage of Conditional Control blocks
- JMAAB guideline jc_0656

Check fundamental logical and numerical operations

Check ID: `mathworks.jmaab.na_0002`

Checks data types in numerical and logic blocks.

Description

Checks the data types for logical and numerical blocks and identifies when the data type is not appropriate for the operation.

The data type for logical blocks should be Boolean. Logic blocks include:

- Logical Operator (AND, OR, NOT)
- Enable (port)
- Trigger (port)

The data type for numerical blocks should be non-Boolean. Numerical blocks include:

- Complex to Real-Imag
- Product
- Dot Product
- Gain
- Sign
- Slider Gain
- Sum

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b
- JMAAB — a, b

To customize the blocks and masks to include in the analysis for this check, use the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and search for check ID `na_0002`.
2. Under **Input Parameters**, add or remove blocks and/or masks from the **Blocks for Numerical Operations** table or **Blocks for Logical Operations** table.
3. Click **Apply** and save the configuration.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
na_0002_a: Check input data types of blocks meant for numerical operations	Data type for a numerical operation blocks is Boolean.	Consider having non-Boolean inputs for the numerical operation blocks.
na_0002_b: Check input data types of blocks meant for logical operations	Data type for a logical operation blocks is not Boolean.	Consider having Boolean inputs for the logical operation blocks.

Capabilities and Limitations

- Does not run on library models.
- Allows exclusions of blocks and charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline na_0002: Appropriate usage of basic logical and numerical operations
- JMAAB guideline na_0002

Check usage of Sum blocks

Check ID: `mathworks.jmaab.jc_0121`

Description

Identifies the violations of the guideline found with the usage of the Sum block.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a, b, c

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0121_a: Check shape of Sum block	One or more Sum blocks are "round" shaped but are not part of a feedback loop.	Set the shape of Sum block to "rectangular".
jc_0121_b: Check first input of Sum block	One or more Sum blocks don't have '+' sign as first input and are not part of a feedback loop.	Set first input to Sum block to '+' sign.
jc_0121_c: Check number of inputs for Sum block	One or more Sum blocks have more than two inputs.	Set Sum block to have no more than two inputs.

Capabilities and Limitations

- Runs on library models.
- Supports exclusions of blocks or charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0121: Usage of add and subtraction blocks
- JMAAB guideline jc_0121

Check operator order of Product blocks

Check ID: `mathworks.jmaab.jc_0610`

Description

Checks the operator order of product blocks.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a, b

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0610_a: Check first input to Product block	Improper usage of operator order of Product blocks.	Change the first input in Product block to multiplication('*').
jc_0610_b: Check number of inputs to Product blocks	Product blocks have invalid number of inputs.	Consider having not more than two inputs for the Product blocks.

Capabilities and Limitations

- Supports selection of Guideline Sub IDs.
- Runs on library models.

- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0610: Operator order for multiplication and division block
- JMAAB guideline jc_0610

Check signs of input signals in product blocks

Check ID: `mathworks.jmaab.jc_0611`

Check the sign bit for the input signal data types in product blocks.

Description

For product blocks, this check identifies if the same sign bit is used for input signals with fixed-point data types. Sign bits are either `signed` or `unsigned`.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Input signal data types have different sign bits.	Update the production block so the sign bit for the input signal data types match.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0611: Input sign for multiplication and division blocks
- JMAAB guideline jc_0611

Check for parentheses in Fcn block expressions

Check ID: `mathworks.jmaab.jc_0622`

Description

Checks the use of parentheses in Fcn block expressions. Parentheses must be used to define the operator precedence.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Improper usage of Fcn block expressions.	Resolve the operator precedence in Fcn block expressions by adding parentheses.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Does not allow exclusions of blocks and charts.

See Also

- MAB guideline jc_0622: Usage of Fcn blocks
- JMAAB guideline jc_0622

Check icon shape of Logical Operator blocks

Check ID: `mathworks.jmaab.jc_0621`

Description

Checks icon shape of Logical Operator blocks. Icon shape of Logical Operator should be rectangular.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Improper setting of icon shape for Logical Operator blocks.	Change the icon shape of Logical Operator blocks to rectangular for readability.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0621: Usage of Logical Operator blocks
- JMAAB guideline jc_0621

Check usage of Relational Operator blocks

Check ID: `mathworks.maab.jc_0131`

Check the position of Constant blocks used in Relational Operator blocks.

Description

When the relational operator is used to compare a signal to a constant value, the constant input should be the second, lower input.

This check requires a Simulink Check license.

Available with Simulink Check.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Relational Operator blocks have a Constant block on the first, upper input.	Move the Constant block to the second, lower input.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems that have no workspaces and no dialogs.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0131: Usage of Relational Operator blocks
- JMAAB guideline jc_0131

Comparing floating point types in Simulink

Check ID: `mathworks.jmaab.jc_0800`

Description

Checks if equivalence comparison is done on floating-point numbers.

This check looks for these blocks:

- Relational Operator
- Compare To Zero

- Compare To Constant

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more instances of equivalence comparison for floating-point numbers is observed.	Make sure to avoid the use of equivalence comparisons for floating-point numbers.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0800: Comparing floating-point types in Simulink
- JMAAB guideline jc_0800

Check usage of Lookup Tables

Check ID: `mathworks.jmaab.jc_0626`

Checks for the correct parameter settings in Lookup Tables to prevent unexpected results.

Description

Checks n-D Lookup (1-D,2-D, and n-D) tables for the following parameters to ensure that the values adhere to the corresponding recommendations.

- InterpMethod
- ExtrapMethod
- UseLastTableValue

Checks Dynamic Lookup Tables for the parameter **LookUpMeth** and ensures that the values adhere to the recommendation.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b
- JMAAB — a, b

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0626_a: Check Lookup Method settings for Dynamic Lookup table blocks	The parameter **Lookup Method** in the dynamic lookup table is set to other than **Interpolation-Use End Values**.	Set the parameter to the recommended value **Interpolation-Use End Values**.
jc_0626_b: Check Lookup Method settings for n-D Lookup table blocks	The parameter **Interpolation method** in the n-D lookup table is not set to **Linear point-slope** or .**Linear Lagrange**.	Update the parameter settings with one of the following recommended values • **Linear point-slope** • **Linear Lagrange**
	The parameter **Extrapolation method** in the n-D lookup table is set to **Cubic spline** or **Linear**.	Set the parameter to the recommended value **Clip**.
	The parameter **Use last table value for inputs at or above last breakpoint** in the n-D lookup table is set to **off**.	Set the parameter to the recommended value **on**.

Capabilities and Limitations

- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0626: Usage of Lookup Table blocks
- JMAAB guideline jc_0626

Check usage of Memory and Unit Delay blocks

Check ID: `mathworks.jmaab.jc_0623`

Checks Memory and Unit Delay blocks with inappropriate sample time.

Description

This check identifies these conditions:

- Memory blocks with a discrete sample time.
- Delay and Unit Delay blocks with a non-discrete sample time.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub-ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Memory blocks have sample time that is not continuous.	Use Unit Delay block instead of Memory block.
Unit Delay blocks have non-discrete sample time.	Use Memory block instead of Unit Delay block.

Capabilities and Limitations

- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0623: Usage of continuous-time Delay blocks and discrete-time Delay blocks
- JMAAB guideline jc_0623

Check for cascaded Unit Delay blocks

Check ID: `mathworks.jmaab.jc_0624`

Description

Identifies cascaded and tapped pattern of Unit Delay blocks.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a, b

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0624_a: Check Delay blocks usage in tapped pattern	Set of Unit Delay blocks in the model can be replaced by Tapped Delay block.	Consider replacing cascaded Unit Delay blocks with Tapped Delay block.
jc_0624_b: Check usage of cascaded Delay blocks	Set of Delay blocks can be replaced by a single Delay block.	Consider replacing cascaded Delay blocks with a Delay block.

Capabilities and Limitations

- Runs on library models.
- Supports exclusions of blocks or charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0624: Usage of Tapped Delay blocks/Delay blocks
- JMAAB guideline jc_0624

Check usage of Discrete-Time Integrator block

Check ID: `mathworks.jmaab.jc_0627`

Check usage of Discrete-Time Integrator block.

Description

For Discrete-Time Integrator blocks, check:

- Block parameter **Limit output** is selected.
- Saturation limits is defined using a `Simulink.Parameter` or `MPT.Parameter` object whose data type is `auto`.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub-ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a, b

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0627_a: Check Saturation limit settings of Discrete-Time Integrator blocks	Block parameter **Limit output** is cleared	Select the Discrete-Time Integrator block parameter **Limit output**.
jc_0627_b: Check Saturation limit settings of Discrete-Time Integrator blocks	Saturation limit is defined by a Parameter object whose data type is not `auto`	Change the data type for the Parameter object to `auto`.

Capabilities and Limitations

- Runs on library models.
- Supports exclusions.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0627: Usage of Discrete-Time Integrator blocks
- JMAAB guideline jc_0627

Check usage of the Saturation blocks

Check ID: `mathworks.jmaab.jc_0628`

Description

This check identifies:

- The Saturation or Saturation Dynamic blocks with any type casting operations. The check compares that the compiled input and output data types match or checks that **Output data type** is set to **Inherit: Same as input** and **Inherit: Same as second input** for Saturation and Saturation Dynamic blocks respectively.
- If the upper limit is not set to less the maximum value of the output data type (intmax, realmax) and the lower limit is not set to greater than the minimum value of the output data type (intmin, -realmax).

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
The input and output data types are different.	Make sure that the **Output data type** is set to **Inherit: Same as input** and **Inherit: Same as second input** for Saturation and Saturation Dynamic blocks respectively.
The upper limit and lower limit values of the blocks are not set to adhered values.	Do any of the following: • Set the **upper limit** of the output data type to less than the maximum value. • Set the **lower limit** of the output data type to greater than the minimum value.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0628: Usage of Saturation blocks
- JMAAB guideline jc_0628

Check output data type of operation blocks

Check ID: `mathworks.jmaab.jc_0651`

Description

Checks if the model adheres to the guidelines for implementing type conversion.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more Operation blocks found that explicitly specify output data type.	Instead of explicitly specifying output data type on operation blocks, use Data Type Conversion block when changing the data type of the block output signal.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks or charts.

See Also

- MAB guideline jc_0651: Implementing a type conversion
- JMAAB guideline jc_0651

Check position of Inport and Outport blocks

Check ID: `mathworks.jmaab.db_0042`

Description

Check whether the model contains ports with invalid position and configuration.

In models, ports must comply with the following rules:

- Place Inport blocks for a subsystem on the left side of the diagram compared to all other blocks and lines connected to the subsystem. You can move the Inport blocks to the right only to prevent signal crossings.
- Place Inport blocks for a subsystem on the left side of the diagram compared with all other blocks and lines connected to the subsystem. You can move the Inport block to the right only to prevent signal crossings.
- Avoid using duplicate Inport blocks at the subsystem level if possible.
- Do not use duplicate Inport blocks at the root level.

Available with Simulink Check.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b
- JMAAB — a, b, c

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
db_0042_a: Check positions of Inport blocks	Inport blocks are not placed to left side of the diagram.	Place the Inport blocks to the left side of the diagram. Block placement causing signal overlaps can be excluded.
db_0042_b: Check positions of Outport blocks	Outport blocks are not placed to right side of the diagram.	Place the Outport blocks to the right side of the diagram. Block placement causing signal overlaps can be excluded.
db_0042_c: Check usage of Duplicate Inport blocks	Ports are duplicate Inport blocks.	• If the duplicate Inport blocks are in a subsystem, remove them where possible. • If the duplicate Inport blocks are at the root level, remove them.

Capabilities and Limitations

- Runs on library models.
- Allows exclusions of blocks and charts.
- In models with multiple subsystems, this check does not flag if all the Inport and Outport blocks are not aligned in a straight line.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to on.

- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline db_0042: Usage of Inport and Outport blocks
- JMAAB guideline db_0042

Check display for port blocks

Check ID: `mathworks.maab.jc_0081`

Check the **Icon display** setting for Inport and Outport blocks.

Description

The **Icon display** setting is required.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
The **Icon display** setting is not set.	Set the **Icon display** to `Port number` for the specified Inport and Outport blocks.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems that have no workspaces and no dialogs.
- Allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0081: Inport and Outport block icon display

- JMAAB guideline jc_0081

Check scope of From and Goto blocks

Check ID: `mathworks.maab.na_0011`

Check the scope of From and Goto blocks.

Description

You can use global scope for controlling flow. However, From and Goto blocks must use local scope for signal flows.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
From and Goto blocks are not configured with local scope.	• Make sure that the ports are connected. • Change the scope of the specified blocks to local.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline na_0011: Scope of Goto and From blocks
- JMAAB guideline na_0011

Check for usage of Data Store Memory blocks

Check ID: `mathworks.jmaab.jc_0161`

Description

This check identifies these conditions:

- Data Store Memory blocks must be used only if the block data is used for code generation or execution.
- Data Store Memory blocks must be defined at smallest scope level.

  For example, consider a model that contains subsystem **A**, which contains two subsystems, **B** and **C**. If you add a Data Store Read block to subsystem B and a Data Store Write block to subsystem C, you must add a Data Store Memory block to subsystem A, which is the smallest common scope for both blocks, and not at the model level.

Note To access the Data Store Memory block, you can place the Data Store Read and Data Store Write blocks in the same subsystem or levels below in the model hierarchy. The models that violate this are not in the scope of this check.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a, b

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0161_a: Check Data Store Memory block definition	One or more Data Store Memory blocks in the model are not defined at smallest scope level.	Consider moving the Data Store Memory blocks to smallest scope level.
jc_0161_b: Check usage of data in Data Store Memory block	Data in some Data Store Memory blocks in the model are not used for execution and code generation.	Use Data Store Memory blocks only if its data is used for code generation or execution.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

- Allows exclusions.

See Also

- MAB guideline jc_0161: Definition of Data Store Memory blocks
- JMAAB guideline jc_0161

Check usage of Switch blocks

Check ID: `mathworks.maab.jc_0141`

Check usage of Switch blocks.

Description

Verifies that the Switch block control input (the second input) is a Boolean value and that the block is configured to pass the first input when the control input is nonzero.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
The Switch block control input (second input) is not a Boolean value.	Change the data type of the control input to Boolean.
The Switch block is not configured to pass the first input when the control input is nonzero.	Set the block parameter **Criteria for passing first input** to u2 ~=0.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems that have no workspaces and no dialogs.
- Allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking. However, edit-time checking for this check does not verify that the data type of the control input is a Boolean value.

See Also

- MAB guideline jc_0141: Usage of the Switch blocks
- JMAAB guideline jc_0141

Check input and output datatype for Switch blocks

Check ID: `mathworks.jmaab.jc_0650`

Check whether the input and output data types for data ports are the same for switching function blocks.

Description

For Switch, Multiport Switch, and Index Vector blocks, check that the input and output data ports have the same data type.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Input and output data ports have different data types.	Change the input or output data port so the data type is the same for both.

Capabilities and Limitations

- Does not run on library models.
- Allows exclusions.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0650: Block input/output data type with switching function
- JMAAB guideline jc_0650

Check settings for data ports in Multiport Switch blocks

Check ID: `mathworks.jmaab.jc_0630`

Description

Identifies the Multiport Switch blocks that violate data port settings.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub-ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, c
- JMAAB — a, b, c

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0630_a: Check number of inputs to Multiport Switch block	**Switch block** or a **Multiport Switch block** in the model have fewer than two data ports.	Make sure that the **Switch blocks** or a **Multiport Switch block** in the model have data ports that are greater than two.
jc_0630_b: Check input type to Multiport Switch block	Data type of **control port** on the **Multiport Switch** is not set to **unsigned integer**.	Change the data type of the **control port** on the **Multiport Switch** to **unsigned integer**.
jc_0630_c: Check data port order of Multiport Switch block	Multiport Switch blocks have incorrect settings.	Set the Multiport Switch block setting **Data port for default case** to **Additional data port**, and **Diagnostics for default case** to **None**.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content of library linked blocks.

Edit-Time Checking

This check is supported by edit-time checking. However, edit-time checking for this check does not verify compliance with jc_0630 Sub ID b.

See Also

- MAB guideline jc_0630: Usage of Multiport Switch blocks
- JMAAB guideline jc_0630

Check for missing ports in Variant Subsystems

Check ID: `mathworks.jmaab.na_0020`

Description

Checks for number of inputs/outputs to a Variant Subsystem.

This check requires a Simulink Check license.

Check Parameterization

You can configure the check to allow the Variant Subsystem to have different number of outports than the Choice Subsystems by selecting the input parameter **Check for parameter 'Specify output when unconnected' on Variant Subsystem outports** in Model Advisor Configuration Editor.

Note: The outports of Variant Subsystem must have the input parameter **specify output when unconnected** selected.

Results and Recommended Actions

Condition	Recommended Action
One or more Variant Subsystems have different number of inputs/outputs on their subordinate subsystems.	Consider having same number of inputs/outputs on Variant Subsystems and their subordinate subsystems.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks or charts.

See Also

- MAB guideline na_0020: Number of inputs to variant subsystems
- JMAAB guideline na_0020

Check use of default variants

Check ID: `mathworks.maab.na_0036`

Check use of default variants in a variant subsystem.

Description

Checks Variant Subsystem, Variant Source, Variant Sink, and variant Model blocks in a variant subsystem for a default variant.

This check requires a Simulink Check license.

Check Parameterization

To set the active variant as the default variant, use the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and search for check ID `na_0036`.
2. Under **Input Parameters**, select **Check use of 'Allow zero active variant controls' option**.
3. For each Output ports of the variant subsystem, set the following block parameters:
   - Select **Specify output when source is unconnected**
   - Enter a **Constant value**
   - Set the output block parameter **Data type** to `Inherit: auto`
4. Click **Apply** and save the configuration.

Results and Recommended Actions

Condition	Recommended Action
The subsystem does not contain a default variant.	Set block parameter **Variant control** to `(default)`.
Block parameter **Variant Control** is set to `Variant`.	To set the active variant as the default variant. 1 **Variant Control** is set to `Variant` 2 Open the variant block and select block parameter **Allow zero active variant controls**. 3 For output ports of the variant subsystem: • Set **Specify output when source is unconnected** to `true` • Provide a valid value in **Constant value** • Set **Output Data type** to `Inherit: auto`

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts
- Allows syntax highlighting

See Also

- MAB guideline na_0036: Default variant
- JMAAB guideline na_0036

Check use of single variable variant conditionals

Check ID: `mathworks.maab.na_0037`

Check use of single variables in conditional expressions

Description

Checks Variant Subsystem, Variant Source, Variant Sink, and variant Model blocks for conditional expressions that have more than one variable.

Note Guideline na_0037 states that default variants are an exception to the recommendation of writing variant conditional expressions using multiple variable with a single condition. You can define a default by:

- Selecting `(default)` in the block parameter **Variant control**.
- Specifying an exhaustive condition.

This check cannot differentiate between defaults that are defined using an exhaustive condition.

This check requires a Simulink Check license.

Examples for expected check behaviour for acceptable variant conditions include:

- Pass case 1: Only one variant condition has multiple variables with single condition (same condition or single condition, `x==0` is assumed as default case):
  - `var1 IN==0 && OUT==0`
  - `var2 IN==1`
- Pass case 2: Variant conditions with same variable with multiple condition:
  - `var1 IN==0 && IN==1`
  - `var2 OUT==1 && OUT==2`

Examples for expected check behaviour for violating variant conditions include:

- Violation case 1: More than one variant condition has multiple variables with single condition:
  - `var1 IN==0 && OUT==0`
  - `var2 IN==1 && OUT==1`
- Violation case 2: More than one variant condition has multiple variables with multiple condition:
  - `var1 IN==1 && OUT==2`
  - `var2 IN==2`

- Violation case 3: One of the variant conditions is `default` and other one has multiple variables with single condition. This is not acceptable case as more than one default variant is not allowed:
  - `var1 IN==0 && OUT==0`
  - `var2 default`

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Conditional expression contains more than one condition variable.	Consider updating your model so that only one variant is used.
Conditional expression variable or `Simulink.Variant` object is not found in the workspace.	Consider defining a variant in your model.
Check does not execute on my variant subsystem.	Clear **Override variant conditions and use following variant** for the variant subsystem.

Capabilities and Limitations

- Does not check default variants.
- Does not run on the Variant subsystem when you select **Override variant conditions and use following variant**
- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in all masked subsystems.

See Also

- MAB guideline na_0037: Use of single variable for variant condition
- JMAAB guideline na_0037

Check for names of Stateflow ports and associated signals

Check ID: `mathworks.maab.db_0123`

Check for mismatches between Stateflow ports and associated signal names.

Description

The name of Stateflow input and output should be the same as the corresponding signal.

Available with Simulink Check.

This check requires a Stateflow license.

Check Parameterization

This Model Advisor check is not applicable for JMAAB modeling guidelines.

This check does not include sub-checks

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — Not supported

Results and Recommended Actions

Condition	Recommended Action
Signals have names that differ from the corresponding Stateflow ports.	Change the names of either the signals or the Stateflow ports.

Capabilities and Limitations

- Does not flag name mismatches for reusable Stateflow charts in libraries.
- Does not flag Stateflow ports when the corresponding signal does not have a label.
- Does not run on library models.
- Does not analyze content of library linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts. Exclusions will not work for library linked charts.

See Also

- MAB guideline db_0123: Stateflow port names

Check definition of Stateflow data

Check ID: `mathworks.jmaab.db_0125`

Description

Identifies the Scope value set on Stateflow data defined at machine level.

This check requires a Simulink Check and Stateflow licenses.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b, c, d
- JMAAB — a, b, c, d

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
db_0125_a: Check for Stateflow data with Scope set to Local at machine level	Stateflow data with Local Scope defined at machine level	Consider not defining data with Local Scope at machine level.
db_0125_b: Check for Stateflow data with Scope set to Constant at machine level	Stateflow data with Constant Scope defined machine level	Consider not defining data with Constant Scope at machine level.
db_0125_c: Check for Stateflow data with Scope set to Parameter at machine level	Stateflow data with Parameter Scope defined at machine level	Consider not defining data with Parameter Scope at machine level.
db_0125_d: Check for duplicate Stateflow Data names with Scope set to Local on a Stateflow block hierarchy	Stateflow data defined in a chart have multiple definition on the same Stateflow block hierarchy	Consider using unique Stateflow data names in the hierarchy.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions.

See Also

- MAB guideline db_0125: Stateflow local data
- JMAAB guideline db_0125

Check definition of Stateflow events

Check ID: `mathworks.jmaab.db_0126`

Description

Stateflow events should be defined at the smallest possible scope of usage.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more Stateflow events used in a chart are not defined at the same level in the hierarchy.	Consider defining the Stateflow events at the smallest scope of usage.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Support exclusions at chart level.

See Also

- MAB guideline db_0126: Defining Stateflow events
- JMAAB guideline db_0126

Check usable number for first index

Check ID: `mathworks.jmaab.jc_0701`

Description

Identifies if the first index of arrays in Stateflow is not set to either **0** or **1**.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a1/a2
- JMAAB — a1/a2

Note Sub-check jc_0701_a1 is selected by default.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0701_a1: Check if first index of Stateflow data is set to 0	One or more Stateflow data have first index set to a value other than 0.	Make Sure to set the first index value to 0.
jc_0701_a2: Check if first index of Stateflow data is set to 1	One or more Stateflow data have first index set to a value other than 1.	Make Sure to set the first index value to 1.

Capabilities and Limitations

- Supports selection of Guideline Sub IDs.
- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of charts.

See Also

- MAB guideline jc_0701: Usable number for first index
- JMAAB guideline jc_0701

Check execution timing for default transition path

Check ID: `mathworks.jmaab.jc_0712`

Description

Identifies the state of the parameter **Execute (enter) Chart At Initialization**. This parameter requires many other considerations to produce consistent results.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
The parameter **Execute (enter) Chart At Initialization** is selected.	Make sure to clear the selection.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0712: Execution timing for default transition path
- JMAAB guideline jc_0712

Check scope of data in parallel states

Check ID: `mathworks.jmaab.jc_0722`

Description

The scope of local variables must be set as restricted to one parallel state unless that same data is required by two or more parallel states.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
The scope of Stateflow data (local variables) is not restricted to a parallel state when the same data is not required by multiple parallel states.	Restrict the scope of Stateflow data (local variables) to only one parallel state.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of charts.

See Also

- MAB guideline jc_0722: Local data definition in parallel states
- JMAAB guideline jc_0722

Check for unconnected objects in Stateflow Charts

Check ID: `mathworks.jmaab.jc_0797`

Description

Checks for unconnected objects in Stateflow Charts and Identifies dangling transitions and unconnected Stateflow States and Junctions in Stateflow Charts.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub-ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b
- JMAAB — a, b

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0797_a: Check unconnected transitions	One or more transitions in the chart are unconnected.	Consider remodeling to connect the dangling transitions.

Guideline Sub ID	Condition	Recommended Action
jc_0797_b: Check unconnected states and junctions	One or more states and junctions are unconnected.	Consider remodeling to connect the unconnected States and Junctions.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0797: Unconnected transitions / states / connective junctions
- JMAAB guideline jc_0797

Check for state in state machines

Check ID: `mathworks.jmaab.db_0137`

Description

Identifies states with OR(exclusive) type decomposition with only one sub-state.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more states with OR(exclusive) type decomposition have only one sub-state in the model.	Remove the sub-state or add another state.

Capabilities and Limitations

- Runs on library models.

- Allows exclusions of blocks and charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline db_0137: States in state machines
- JMAAB guideline db_0137

Check usage of parallel states

Check ID: `mathworks.jmaab.jc_0721`

Description

Parallel states must not be used for the purpose of grouping that is the substates of parallel states must not be parallel states.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Parallel states are only for grouping.	Substates of the parallel states must not be parallel (do not use for grouping).

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

- Allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0721: Usage of parallel states
- JMAAB guideline jc_0721

Check for Stateflow transition appearance

Check ID: `mathworks.jmaab.db_0129`

Description

Checks and reports Stateflow transitions that are visually overlapping other Stateflow objects.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b, c, d, e
- JMAAB — a, b, c, d, e

Self transitions can be allowed to flag with this check by selecting the parameter **Check for Self Transitions** from the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and search for check ID `db_0129`.
2. Select the parameter **Check for Self Transitions**.
3. Click **Apply** and save the configuration.

Results and Recommended Actions

Guideline Sub IDs	Condition	Recommended Action
db_0129_a: Check for transition lines that cross over one another	Transition lines cross over one another.	Consider remodeling so that transition lines do not cross over one another.
db_0129_b: Check for transition lines that overlap one another	Transition lines overlap with other transition lines.	Consider remodeling so that transition lines do not overlap with other transition lines.

Guideline Sub IDs	Condition	Recommended Action
db_0129_c: Check transition lines that cross over other Stateflow objects	Transition lines cross over other Stateflow objects.	Consider remodeling so that transitions do not cross over other Stateflow objects.
db_0129_d: Check orientation of transition lines	Transitions are not vertical or horizontal and/or diagonal for flow chart loops.	Consider remodeling using either horizontal or vertical transitions only and diagonal transitions for flow chart loops.
db_0129_e: Check for unnecessary connective junctions	One or more charts use unnecessary connective junctions.	Consider avoiding unnecessary connective junctions.

Capabilities and Limitations

- This check does not flag the transitions that are overlapped by the state labels exceeding the boundary of the state.
- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline db_0129: Stateflow transition appearance
- JMAAB guideline db_0129

Check default transition placement in Stateflow charts

Check ID: `mathworks.jmaab.jc_0531`

Description

Checks the default transition placement in Stateflow charts.

This check requires a Simulink Check and Stateflow license.

Results and Recommended Actions

Guideline Sub-ID	Condition	Recommended Action
jc_0531_a: Check for default transitions at every level	One or more states or junctions in the model do not have any default transitions at their level.	Make sure that there is at least one default transition at every level.
jc_0531_b: Check for parallel states having default transitions	One or more parallel states in the model have default transitions.	Default transitions must not be used for parallel states.

Guideline Sub-ID	Condition	Recommended Action
jc_0531_c: Check for multiple default transitions at same level	One or more states in the model have multiple default transitions at the same level.	Make sure that multiple default transitions are not included in the same level.
jc_0531_d: Check for default transitions not connected to top of state or junction	One or more default transitions in the model are not connected to the upper part of the state or junction.	Default transitions must be directly connected vertically to the upper part of the state or junction.
jc_0531_e: Check for default transitions not positioned at top left within the same level	One or more destination states or junctions of default transitions in the model are not on the top left.	The default transitions must be positioned on the top left within the same level.
jc_0531_f: Check for default transitions exceeding state boundaries	One or more default transitions in the model exceed state boundaries.	The default transition must not exceed state boundaries.
jc_0531_g: Check for unconditional default transitions.	One or more default transitions in the model do not have a single non-guard path to any state at the level in the chart.	Set **No unconditional default transitions** parameter to recommended value 'error'.

Capabilities and Limitations

- Runs on library models.
- Allows exclusions of blocks and charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0531: Default transition
- JMAAB guideline jc_0531
- “Syntax for States and Transitions” (Stateflow)

Check usage of transitions to external states

Check ID: `mathworks.jmaab.jc_0723`

Description

Identifies transitions in Stateflow Charts that end on external child states.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more transitions end on external child states.	Consider remodeling to avoid use of transitions ending on external child states.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Supports exclusions of charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0723: Prohibited direct transition from external state to child state
- JMAAB guideline jc_0723

Check for unexpected backtracking in state transitions

Check ID: `mathworks.jmaab.jc_0751`

Description

Checks unexpected backtracking in state transitions. Configuration parameter for **Unexpected backtracking (SFUnexpectedBacktrackingDiag)** must be set to **error**.

Note

- This check looks for backtracking issues.
- This check does not look for connective junctions used for separating the complex conditions.
- Splitting complex conditions does not always result in backtracking issues.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Backtracking is undetected during the state transition.	Set configuration parameter for **Unexpected backtracking (SFUnexpectedBacktrackingDiag)** to **error**.

Capabilities and Limitations

- Runs on library models.
- Does not analyze content of library-linked blocks.
- Does not analyze content in masked subsystems.
- Does not allow exclusions of blocks and charts.

See Also

- MAB guideline jc_0751: Backtracking prevention in state transition
- JMAAB guideline jc_0751

Check starting point of internal transition in Stateflow

Check ID: `mathworks.jmaab.jc_0760`

Description

Identifies if in all state charts and flow charts, internal transitions from state boundaries must start from the left edge of the state.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Starting point of one or more internal transitions from state boundaries of state charts or flow charts does not start from the left edge of the state.	Make sure that in all the state charts and flow charts, internal transitions from state boundaries must start from the left edge of the state.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0760: Starting point of internal transition
- JMAAB guideline jc_0760

Check usage of internal transitions in Stateflow states

Check ID: `mathworks.jmaab.jc_0763`

Description

Identifies the Stateflow states that uses multiple internal transitions.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub-ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a1/a2
- JMAAB — a1/a2

Note Sub-check jc_0763_a1 is selected by default.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0763_a1: Check for multiple internal transitions	One or more Stateflow states have multiple internal transitions.	Remodel to avoid the use of multiple internal transitions.
jc_0763_a2: Check order of multiple internal transitions	One or more Stateflow states have multiple internal transitions not placed in order of execution.	Consider placing internal transitions from top to bottom in the order of execution.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0763: Usage of multiple internal transitions
- JMAAB guideline jc_0763

Check prohibited combination of state action and flow chart

Check ID: `mathworks.jmaab.jc_0762`

Description

Checks if state actions within states and flow chart statements are used in combination.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Stateflow states combine state action and flow chart.	Separate state actions and flow chart statements into different states.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0762: Prohibition of state action and flow chart combination
- JMAAB guideline jc_0762

Check transitions in Stateflow Flow charts

Check ID: `mathworks.jmaab.db_0132`

Description

Check transition orientations in flow charts.

The following rules apply to transitions in flow charts:

- Draw transition conditions horizontally.
- Draw transitions with a condition action vertically.
- Junctions in flow charts should have a default exit transition.
- Transitions in flow charts should not combine condition and action.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub-ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b
- JMAAB — a, b

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
db_0132_a: Check usage of transition actions in Stateflow Flow charts	One or more transition actions are used in flow charts in the model.	Do not use transition actions in flow charts.
db_0132_b: Check for transition orientation in Flow charts	One or more transitions with condition expressions are not drawn horizontally.	Make sure to draw transitions with condition expressions horizontally.
	One or more transitions with condition actions are not drawn vertically.	Make sure to draw transitions with condition actions vertically.
	One or more transitions have both condition expressions and condition actions.	Do not use condition expressions and condition actions in the same transition.

Capabilities and Limitations

- The check only flags flow charts containing loop constructs if the transition violates the orientation rule.
- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline db_0132: Transitions in flow charts
- JMAAB guideline db_0132

Check usage of unconditional transitions in flow charts

Check ID: `mathworks.jmaab.jc_0773`

Description

Identifies unconditional transitions in flow charts.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b
- JMAAB — a, b

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0773_a: Check for the presence of unconditional transition originating from a Stateflow junction with conditional transition	One or more Stateflow junctions do not have unconditional transitions originating from them.	Consider adding an unconditional transition to the junction.
jc_0773_b: Check execution order of unconditional transitions	One or more Stateflow junctions have unconditional transitions that are not executed last.	Consider setting the order of execution of the unconditional transition from the junction to the highest value.

Capabilities and Limitations

- Runs on library models.
- Supports exclusions of blocks or charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0773: Unconditional transition of a flow chart
- JMAAB guideline jc_0773

Check terminal junctions in Stateflow

Check ID: `mathworks.jmaab.jc_0775`

Description

Identifies the usage of terminal junctions in flow charts.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub-ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a1/a2
- JMAAB — a1/a2

Note Sub-check jc_0775_a1 is selected by default.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0775_a1: Check for usage of only one terminal junction	One or more Stateflow containers have more than one terminal junction.	Consider using only one terminal junction.
jc_0775_a2: Check for usage of one terminal junction with one unconditional transition as input	One or more Stateflow containers have either more than one terminal junction or a terminal junction without one unconditional transition.	Consider using only one terminal junction with one unconditional transition as input.

Capabilities and Limitations

- Runs on library models.
- Supports exclusions of blocks or charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0775: Terminating junctions in flow charts
- JMAAB guideline jc_0775

Check usage of Stateflow comments

Check ID: `mathworks.jmaab.jc_0738`

Description

Identifies the comments that contains newline(s) or nested in the middle in Stateflow.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a, b

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0738_a: If the action language is "C", comment nesting must not be used	On or more comments in Stateflow objects are nested.	Comments in Stateflow must not be nested.
jc_0738_b: If the action language is "C", comments must not contain newline(s) in the middle	One or more comments in Stateflow objects contain newline(s) in the middle.	Comments in Stateflow must not contain newline(s) in the middle.

Capabilities and Limitations

- Runs on library models.
- Allows exclusions of blocks and charts.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

See Also

- MAB guideline jc_0738: Usage of Stateflow comments
- JMAAB guideline jc_0738

Check Stateflow chart action language

Check ID: `mathworks.jmaab.jc_0790`

Description

Checks if the action language of Stateflow charts is set to C.

This check requires Simulink Check and Stateflow licenses.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Action language of one or more Stateflow charts is not set to C.	Set all Stateflow charts action language to C.

Capabilities and Limitations

- Runs on library models.
- Allows exclusions of charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0790: Action language of Chart block
- JMAAB guideline jc_0790

Check usage of numeric literals in Stateflow

Check ID: `mathworks.jmaab.jc_0702`

Description

Identifies the use of numeric literals in Stateflow states and transitions.

Note Following are the exceptions for this check:

- Initial value set to**0**.
- Increment, decrement value set to **1**.

This check requires Simulink Check and Stateflow licenses.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more expressions in Stateflow states and Stateflow transitions contain numeric literals.	Consider remodeling to use named parameters and constants instead of numeric literals.

Capabilities and Limitations

- Runs on library models.
- Supports exclusions of blocks or charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0702: Use of named Stateflow parameters and constants
- JMAAB guideline jc_0702

Check for pointers in Stateflow charts

Check ID: `mathworks.maab.jm_0011`

Identify pointer operations on custom code variables.

Description

Pointers to custom code variables are not allowed.

This check requires a Simulink Check and Stateflow license.

This check requires a license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Custom code variables use pointer operations.	Modify the specified chart to remove the dependency on pointer operations.

Capabilities and Limitations

- Applies only to Stateflow charts that use C as the action language.
- Runs on library models.
- Does not analyze content of library linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jm_0011: Pointers in Stateflow
- JMAAB guideline jm_0011

Check for usage of events and broadcasting events in Stateflow charts

Check ID: `mathworks.jmaab.jm_0012`

Identify undirected event broadcasts that might cause recursion during simulation and generate inefficient code.

Description

Event broadcasts in Stateflow charts must be directed.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a1/a2/a3

Note Sub-check jm_0012_a1 is selected by default.

Results and Recommended Actions

Guideline Sub IDs	Condition	Recommended Action
jm_0012_a1: Check if Stateflow events are used only in the output of Stateflow blocks	Stateflow events are not used in the output of Stateflow blocks.	Change the scope of above listed Stateflow events to output.
jm_0012_a2: Check if Stateflow broadcast events are directed using "send" syntax	Stateflow broadcast events are not directed using "send" syntax.	Use "send(event_name, state_name)" syntax to broadcast Stateflow events.
jm_0012_a3: Check if Stateflow broadcast events are directed using qualified event name	Stateflow events are not directed using qualified event name.	Use "send(state_name.event_name)" syntax to broadcast Stateflow events.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jm_0012: Usage restrictions of events and broadcasting events
- JMAAB guideline jm_0012
- “Broadcast Local Events to Synchronize Parallel States” (Stateflow)

Check order of state action types

Check ID: `mathworks.jmaab.jc_0733`

Description

Identifies state actions that are out of order in Stateflow states.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a, b
- JMAAB — a, b

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0733_a: Check basic state action types	One or more Stateflow states have basic state action types written out of order.	Consider ordering the state actions in the order of entry (en), during (du), and exit (ex).
jc_0733_b: Check combined state action types	One or more Stateflow states have combined state action types written out of order.	Consider ordering the state actions in the order of entry (en), during (du), and exit (ex).

Capabilities and Limitations

- Runs on library models.
- Supports exclusions of charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0733: Order of state action types
- JMAAB guideline jc_0733

Check repetition of Action types

Check ID: `mathworks.jmaab.jc_0734`

Identifies repeated Action types in a Stateflow state.

Description

The action types (entry (en), during (du), exit (ex), en, du:, du, ex:, en, ex:, en, du, ex:) must not be described two or more times in a Stateflow state.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more action types is used multiple times in a Stateflow state.	Merge the actions types so that each of the action types is defined only once in a Stateflow state.

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of charts.

See Also

- MAB guideline jc_0734: Number of state action types
- JMAAB guideline jc_0734

Check if state action type 'exit' is used in the model

Check ID: `mathworks.jmaab.jc_0740`

Description

Checks if Stateflow exit actions are used in the model.

This check requires Simulink Check and Stateflow licenses.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more Stateflow states use **exit** action type.	Consider removing state action type **exit** in Stateflow states.

Capabilities and Limitations

- Runs on library models.
- Supports exclusions of charts.

- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0740: Limitation on use of exit state action
- JMAAB guideline jc_0740

Check updates to variables used in state transition conditions

Check ID: `mathworks.jmaab.jc_0741`

Description

Checks if the variables used in state transition conditions perform an update by "during" state action type.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more variables in the state transition condition performs an update by "during" state action type.	Make sure that the variables used in state transition conditions do not perform an update by "during" state action type.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0741: Timing to update data used in state chart transition conditions
- JMAAB guideline jc_0741

Check usage of transition conditions in Stateflow transitions

Check ID: `mathworks.jmaab.jc_0772`

Description

Identifies the transitions sourced from a state and unconditional Stateflow transitions with higher priority than conditional transitions.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Stateflow transitions found with higher priority than conditional transitions.	Change the execution order of the transitions or add an execution condition.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0772: Execution order and transition conditions of transition lines
- JMAAB guideline jc_0772

Check condition actions and transition actions in Stateflow

Check ID: `mathworks.jmaab.jc_0753`

Description

Checks if the use of condition actions or transition actions are uniform within the same chart.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub-ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a1/a2
- JMAAB — a1/a2

Note Sub-check jc_0753_a1 is selected by default.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0753_a1: Check transition actions in Stateflow charts	The following Stateflow charts use transition actions.	Do not use transition actions in Stateflow charts.
jc_0753_a2: Check usage of condition actions and transition actions within same Stateflow chart	Condition actions and transition actions are mixed within the same chart.	Use of condition actions or transition actions must be uniform within the same chart.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0753: Condition actions and transition actions in Stateflow
- JMAAB guideline jc_0753

Check for MATLAB expressions in Stateflow charts

Check ID: `mathworks.jmaab.db_0127`

Description

Identifies the Stateflow objects that use MATLAB expressions that are not suitable for code generation.

This check requires a Simulink Check license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a1/a2
- JMAAB — a1/a2

Note Sub-check db_0127_a1 is selected by default.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
db_0127_a1: Check for MATLAB expressions in Stateflow charts	One or more Stateflow objects in the model use MATLAB expressions.	Consider remodeling by replacing all MATLAB expressions in Stateflow objects.
db_0127_a2: Check for MATLAB expressions in Stateflow charts not accessed through MATLAB function	One or more Stateflow objects in the model use MATLAB expressions that are not accessed through MATLAB function.	Consider remodeling so that MATLAB expressions are accessed through MATLAB functions in Stateflow objects.

Capabilities and Limitations

- Applies only to Stateflow charts that use C as the action language.
- Runs on library models.
- Allows exclusions.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline db_0127: Limitation on MATLAB commands in Stateflow blocks

- JMAAB guideline db_0127

Check usage of floating-point expressions in Stateflow charts

Check ID: `mathworks.maab.jc_0481`

Identify equal to operations (==) in expressions where at least one side of the expression is a floating-point variable or constant.

Description

Do not use equal to operations with floating-point data types. You can use equal to operations with integer data types.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Expressions use equality comparison operations (==, !=, ~=) where at least one side of the expression is a floating-point variable or constant.	Modify the specified expressions to avoid equality comparison operations between floating-point expressions.
The Model Advisor could not determine the data types in expressions with equality operations.	To allow Model Advisor to determine the data types, consider explicitly typecasting the specified expressions.

Capabilities and Limitations

- Does not run on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0481: Use of hard equality comparisons for floating point numbers in Stateflow
- JMAAB guideline jc_0481

Check Stateflow operators

Check ID: `mathworks.jmaab.na_0001`

Description

Identifies the usage of operators in Stateflow.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a, b1/b2/b3, c

Note Sub-checks na_0001_a, na_0001_b1, and na_0001_c are selected by default.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
na_0001_a: Usage of bitwise operators in Stateflow	One or more expressions have incorrect usage of bitwise operators.	Consider using bitwise operators ('&', '|', '^', '~') for bit operations only.
na_0001_b1: Usage of inequality operator (~=) in Stateflow	One or more expressions have incorrect usage of inequality operators.	Consider using '~=' for inequality operations.
na_0001_b2: Usage of inequality operator (!=) in Stateflow	One or more expressions have incorrect usage of inequality operators.	Consider using '!=' for inequality operations.
na_0001_b3: Usage of inequality operator (<>) in Stateflow	One or more expressions have incorrect usage of inequality operators.	Consider using '<>' for inequality operations.
na_0001_c: Usage of logical negation operator in Stateflow	One or more scenarios have incorrect usage of logical negation operator.	Consider using '!' for logical negation operations

Capabilities and Limitations

- Applies only to charts that use C as the action language.
- Does not run on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to on.

- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline na_0001: Standard usage of Stateflow operators
- JMAAB guideline na_0001

Check prohibited comparison operation of logical type signals

Check ID: `mathworks.jmaab.jc_0655`

Description

Identifies the Boolean type transitions in Stateflow charts that use either comparison with numbers or logical values (true or false), or use negation operators (! or ~) variably in the model.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Boolean type transactions are compared with numbers or logical values (true or false).	Make sure that the Boolean type transactions are not compared with numbers or logical values.

Capabilities and Limitations

- Runs on library models.
- Allows exclusions of blocks and charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0655: Prohibition of logical value comparison in Stateflow
- JMAAB guideline jc_0655

Check usage of unary minus operations in Stateflow charts

Check ID: `mathworks.maab.jc_0451`

Identify unary minus operations applied to unsigned integers in Stateflow objects.

Description

Do not perform unary minus operations on unsigned integers in Stateflow objects.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Unary minus operations are applied to unsigned integers in Stateflow objects.	Modify the specified objects to remove dependency on unary minus operations.
The Model Advisor could not determine the data types in expressions with unary minus operations.	To allow Model Advisor to determine the data types, consider explicitly typecasting the specified expressions.

Capabilities and Limitations

- Does not run on library models.
- Does not analyze content of library linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0451: Use of unary minus on unsigned integers
- JMAAB guideline jc_0451

Check for implicit type casting in Stateflow

Check ID: `mathworks.jmaab.jc_0802`

Description

Identifies implicit type casting in Stateflow.

This check requires Simulink Check and Stateflow licenses.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more operations and/or function calls in Stateflow charts have data type mismatch.	All operations and function calls must be made between variables of the same data type. If the data types are different, the variables need to be explicitly type cast to match data types.

Capabilities and Limitations

- This check will not analyze the enumeration types in Stateflow if the enumeration types are not specified with full class name.
- This check does not analyze the Fixed-Point Context-Sensitive Constants.
- Does not run on library models.
- Allows exclusions of blocks or charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0802: Prohibited use of implicit type casting in Stateflow
- JMAAB guideline jc_0802

Check uniqueness of Stateflow State and Data names

Check ID: `mathworks.jmaab.jc_0732`

Description

Checks if in a single Stateflow chart, the Stateflow data name, Stateflow state name, and Stateflow event name are identical.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
If the Stateflow Data name and the Stateflow State name have the same name in a Stateflow Chart.	Rename either of the Stateflow Data name or Stateflow State name to not to be identical names.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of charts.

See Also

- MAB guideline jc_0732: Distinction between state names, data names, and event names
- JMAAB guideline jc_0732

Check uniqueness of State names

Check ID: `mathworks.jmaab.jc_0730`

Description

State names must be unique in charts, with the exception of Atomic subcharts. I.e. Atomic Subcharts are treated as different container so they can share State Names with other states outside of the subchart.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
In a Stateflow chart, two or more Stateflow states have the same name.	Rename the Stateflow states so that there are no identical names in the Stateflow chart.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of charts.

See Also

- MAB guideline jc_0730: Unique state name in Stateflow blocks
- JMAAB guideline jc_0730

Check usage of State names

Check ID: `mathworks.jmaab.jc_0731`

Checks for slashes (/) in the state names.

Description

Checks if slashes (/) are included in state names. After the state name is defined, add a new line for describing any executable statements. A slash (/) is required only when describing executable statements in continuation after state names.

This check requires Simulink Check and Stateflow licenses.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Slash is in the state name.	Remove the slash from the state name and make sure to start a new line for any executable statements.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of charts.

See Also

- MAB guideline jc_0731: State name format
- JMAAB guideline jc_0731

Check entry formatting in State blocks in Stateflow charts

Check ID: `mathworks.jmaab.jc_0501`

Description

Note Prerequisite: Run syntax checking before running this check.

Identify missing line breaks between entry action (`en`), during action (`du`), and exit action (`ex`) entries in states. Identify missing line breaks after semicolons (`;`) in statements.

Start a new line after the `entry`, `during`, and `exit` entries, and after the completion of a statement ";".

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
An `entry` (`en`) is not on a new line.	Add a new line after the `entry`.
A `during` (`du`) is not on a new line.	Add a new line after the `during`.
An `exit` (`ex`) is not on a new line.	Add a new line after the `exit`.
Multiple statements found on one line.	Add a new line after each statement.

Capabilities and Limitations

- Runs on library models.
- Does not analyze content of library linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MAB guideline jc_0501: Format of entries in a State block
- JMAAB guideline jc_0501

Check indentation of code in Stateflow states

Check ID: `mathworks.jmaab.jc_0736`

Check for uniform indentation of label Strings in Stateflow States and Transitions.

Description

Checks if the indentations in the Stateflow blocks are described uniformly and adhere to the following recommendations:

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a, b, c

You can use the input parameter **Number of single-byte spaces** to set the threshold to the desired value. By default the value is set to **1**.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0736_a: Check for uniform indentation of action language in Stateflow states	One or more Stateflow states in the model do not have uniform indentation.	Consider deleting the blank spaces before state action types and adding the exact number of single-byte spaces as defined in the input parameter before executable statements.
jc_0736_b: Check for uniform spacing of transition action types	One or more Stateflow transitions in the model do not have uniform spacing.	Consider not adding blank spaces before '[' of a transition condition, '{' of a transition action and '/' of the event in a transition.
jc_0736_c: Check for uniform spacing of transition actions	One or more Stateflow transitions in the model do not have uniform spacing.	Consider adding the exact number of single-byte spaces as defined in the input parameter after the '/' of a transition action.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of charts.

See Also

- MAB guideline jc_0736: Uniform indentations in Stateflow blocks
- JMAAB guideline jc_0736

Check for usage of text inside states

Check ID: `mathworks.jmaab.jc_0739`

Description

Identifies the Stateflow states with text exceeding the boundary of the state.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more Stateflow states in the model have text exceeding the boundary of the state.	Make sure that the text inside the Stateflow state does not go outside the boundary of the state.

Capabilities and Limitations

- This check flags Stateflow states in the model that have a newline character in the text. This happens even if the text inside the Stateflow state is contained within the boundary of the state.
- Runs on library models.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0739: Describing text inside states
- JMAAB guideline jc_0739

Check placement of Label String in Transitions

Check ID: `mathworks.jmaab.jc_0770`

Description

Checks the placement of the Stateflow Transition labels. The Stateflow signal label must always be at the origin of the signal or at the midpoint of the signal transition line.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a1/a2

Note Sub-check jc_0770_a1 is selected by default.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0770_a1: Transition labels should be placed near the point of origin of the transition	One or more Stateflow transitions do not have label string placed near the origin of the transition.	Make sure to place the Stateflow labels near the point of origin of the transition.
jc_0770_a2: Transition labels should be placed near center of the transition	One or more Stateflow transitions do not have label string placed near the center of the transition.	Make sure to place the Stateflow labels near the mid-point (center) of the transition.

Capabilities and Limitations

- Runs on library models.
- Allows exclusions of blocks and charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0770: Position of transition label
- JMAAB guideline jc_0770

Check position of comments in transition labels

Check ID: `mathworks.jmaab.jc_0771`

Description

Identifies comments in transition labels that are not positioned uniformly.

This check requires Simulink Check and Stateflow licenses.

Check Parameterization

This check contains sub-checks that correspond to the sub IDs that are specified in the MAB and JMAAB modeling guidelines. You can use the Model Advisor Configuration Editor to specify which sub IDs (one or multiple) to execute.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a1/a2
- JMAAB — a1/a2

Note Sub-check jc_0771_a1 is selected by default.

Results and Recommended Actions

Guideline Sub ID	Condition	Recommended Action
jc_0771_a1: Comments in transition labels must be uniformly positioned on top	One or more comments in transition labels are not positioned at the top.	Comments in transition labels must be positioned above transition conditions, condition actions, transition actions, and Stateflow events.
jc_0771_a2: Comments in transition labels must be uniformly positioned at the bottom	One or more comments in transition labels are not positioned at the bottom.	Comments in transition labels must be positioned below transition conditions, condition actions, transition actions, and Stateflow events.

Capabilities and Limitations

- Runs on library models.
- Supports exclusions of blocks or charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0771: Comment position in transition labels
- JMAAB guideline jc_0771

Check usage of parentheses in Stateflow transitions

Check ID: `mathworks.jmaab.jc_0752`

Description

Checks if a new line is started before and after parentheses for condition actions in Stateflow transitions.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Condition actions in Stateflow transitions are written beside parenthesis.	Start new line before and after parentheses for condition actions in Stateflow transitions.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0752: Condition action in transition label
- JMAAB guideline jc_0752

Check for comments in unconditional transitions

Check ID: `mathworks.jmaab.jc_0774`

Description

Identifies the comments in unconditional transitions without action statements.

This check requires Simulink Check and Stateflow licenses.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more unconditional transitions without action statements do not have comments.	Consider adding a comment explaining the unconditional transition.

Capabilities and Limitations

- Runs on library models.

- Supports exclusions of blocks or charts.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

See Also

- MAB guideline jc_0774: Comments for through transition
- JMAAB guideline jc_0774

Check return value assignments in Stateflow graphical functions

Check ID: `mathworks.maab.jc_0511`

Identify graphical functions with multiple assignments of return values in Stateflow charts.

Description

The return value from a Stateflow graphical function must be set in only one place.

This check requires a Simulink Check and Stateflow license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — No recommendations
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
The return value from a Stateflow graphical function is assigned in multiple places.	Modify the specified graphical function so that its return value is set in one place.

Capabilities and Limitations

- Runs on library models.
- Does not analyze content of library linked blocks.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline jc_0511: Return values from a graphical function
- JMAAB guideline jc_0511

- “Reuse Logic Patterns by Defining Graphical Functions” (Stateflow).

Check usage of Simulink function in Stateflow

Check ID: `mathworks.jmaab.na_0042`

Description

Checks the usage of Simulink functions in Stateflow.

This check requires Simulink Check and Stateflow licenses.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Input or output arguments passed to Simulink Functions are not data or variables of **Local** scope.	Modify the chart data or variables scope to **Local**.
Input or output arguments passed to Simulink Functions use data or variables of **output** scope.	Input or output arguments passed to Simulink Functions should be a mixture of data or variables of **Local** scope and/or **Input** scope.
Simulink functions are not called from multiple locations within the chart.	Make sure Simulink functions are reused.

Capabilities and Limitations

- This check does not analyze if Simulink functions are called every time step.
- Runs on library models.
- Allows exclusions of blocks and charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline na_0042: Usage of Simulink functions
- JMAAB guideline na_0042

Check use of Simulink in Stateflow charts

Check ID: `mathworks.maab.na_0039`

Checks for Stateflow charts that are nested in Simulink functions used in the root Stateflow chart.

Description

Checks Simulink functions in the root Stateflow chart and identifies Stateflow charts that are nested within these functions.

This check requires Simulink Check and Stateflow licenses.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
The Simulink function has a nested Stateflow chart.	Consider modifying your root Stateflow chart so the Simulink function does not contain a nested Stateflow chart.

Capabilities and Limitations

- Does not check Stateflow states.
- Runs on library models.
- Allows exclusions of blocks and charts.
- Allows syntax highlighting.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline na_0039: Limitation on Simulink functions in Chart blocks
- JMAAB guideline na_0039

Check MATLAB Function metrics

Check ID: `mathworks.maab.himl_0003`

Display complexity and code metrics for MATLAB Functions. Report metric violations.

Description

This check provides complexity and code metrics for MATLAB Functions. The check additionally reports metric violations.

A results table provides links to MATLAB Functions that violate the complexity input parameters.

This check requires a Simulink Check license.

Check Parameterization

This Model Advisor check is not applicable for JMAAB modeling guidelines.

This check does not include sub-checks

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — Not supported

To specify the complexity thresholds, use the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and search for check ID `himl_0003`.
2. Under **Input Parameters**, select **Check .m files referenced in the model** to include files with a `.m` extension in the analysis.
3. Define these complexity metrics:
   - **Maximum effective lines of code per function** — Provide the maximum effective lines of code per function. Effective lines do not include empty lines, comment lines, or lines with a function `end` keyword.
   - **Minimum density of comments** — Provide minimum density of comments. Density is ratio of comment lines to total lines of code.
   - **Maximum cyclomatic complexity per function** — Provide maximum cyclomatic complexity per function. Cyclomatic complexity is the number of linearly independent paths through the source code.
4. Click **Apply** and save the configuration.

Results and Recommended Actions

Condition	Recommended Action
MATLAB Function violates the complexity input parameters.	For the MATLAB Function: • If effective lines of code is too high, further divide the MATLAB Function. • If comment density is too low, add comment lines. • If cyclomatic complexity per function is too high, further divide the MATLAB Function.

Capabilities and Limitations

- Runs on library models.
- Does not analyze content of library linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline na_0016: Source lines of MATLAB Functions
- MAB guideline na_0018: Number of nested if/else and case statement

Check MATLAB code for global variables

Check ID: `mathworks.maab.na_0024`

Check for global variables in MATLAB code.

Description

Verifies that global variables are not used in any of the following:

- MATLAB code in MATLAB Function blocks
- MATLAB functions defined in Stateflow charts
- Called MATLAB functions

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Global variables are used in one or more of the following: • MATLAB code in MATLAB Function blocks • MATLAB functions defined in Stateflow charts • Called MATLAB functions	Replace global variables with signal lines, function arguments, or persistent data.

Capabilities and Limitations

- Runs on library models.
- Does not analyze content of library linked blocks.
- Analyzes content in all masked subsystems.
- Does not allow exclusions of blocks or charts.

See Also

- MAB guideline na_0024: Shared data in MATLAB functions
- JMAAB guideline na_0024

Check usage of enumerated values

Check ID: `mathworks.maab.na_0031`

Description

Identifies the enumeration classes used in the model without a default value specification.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
Enumeration classes in the model do not have a default value specification.	Make sure that the enumeration classes used in the model have a **getDefaultValue** method implementation as a static method of the class.

Capabilities and Limitations

- Does not run on library models.
- Does not allow exclusions of blocks and charts.
- Analyzes content of library linked blocks.

See Also

- MAB guideline na_0031: Definition of default enumerated value
- JMAAB guideline na_0031

Check input and output settings of MATLAB Functions

Check ID: `mathworks.maab.na_0034`

Identify MATLAB Functions that have inputs, outputs or parameters with inherited complexity or data type properties.

Description

The check identifies MATLAB Functions with inherited complexity or data type properties. A results table provides links to MATLAB Functions that do not pass the check, along with conditions triggering the warning.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
MATLAB Functions have inherited interfaces with one or more of these conditions: • **Complexity** is `Inherited`. • **Type** is `Inherit: Same as Simulink`. • **Size** is `-1 (Inherited)`.	Explicitly define complexity and data type properties for inports, outports, and parameters of MATLAB Function identified in the results. If applicable, using the **MATLAB Function Block Editor**, make the following modifications in the Ports and Data Manager: • Change **Complexity** from `Inherited` to `On` or `Off`. • Change **Type** from `Inherit: Same as Simulink` to an explicit type. • Change **Size** from `-1 (Inherited)` to an explicit size.

Capabilities and Limitations

- Runs on library models.
- Does not analyze content of library linked blocks.
- Analyzes content in all masked subsystems.
- Allows exclusions of blocks and charts.

See Also

- MAB guideline na_0034: MATLAB Function block input/output settings
- JMAAB guideline na_0034

Check the number of function calls in MATLAB Function blocks

Check ID: `mathworks.maab.na_0017`

Description

Checks whether number of function calls in MATLAB Function blocks is less than the set threshold. By default, the limit is set to three.

This check requires a Simulink Check license.

Check Parameterization

This Model Advisor check is not applicable for JMAAB modeling guidelines.

This check does not include sub-checks

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — Not supported

You can customize the function call levels threshold by using the input parameter **Function Call Level** from the Model Advisor Configuration Editor.

1. Open the Model Configuration Editor and search for check ID `na_0017`.
2. Enter the desired function call level to set in the **Function Call Level** field.

   By default, this parameter is set to **3**.
3. Click **Apply** and save the configuration.

Results and Recommended Actions

Condition	Recommended Action
Number of function calls in MATLAB Function blocks is greater than the set threshold. By default, the set threshold is three. The set threshold can be modified by using the input parameter **Function Call Level** in the configuration editor.	Reduce the number of function calls from MATLAB Function blocks to be less than the set threshold.

Capabilities and Limitations

- Recursive function calls are only counted once.
- Inline class methods are not analyzed.

- Runs on library models.
- Allows exclusions of blocks and charts.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

See Also

- MAB guideline na_0017: Number of called function levels

Check usage of character vector inside MATLAB Function block

Check ID: `mathworks.maab.na_0021`

Check for use of character vectors in MATLAB Function blocks.

Description

Identifies character vectors that are used in MATLAB Function blocks.

MATLAB Functions store strings as character arrays. Due to lack of dynamic memory allocation, the arrays cannot be re-sized to accommodate a string value of different length. Strings are not a supported data type in Simulink, so MATLAB Function blocks cannot pass the string data outside the block.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

You can use Model Advisor Configuration Editor to configure this check to include files with a `.m` extension in the analysis. To enable this feature, in the **Input Parameters** section, select **Check external .m files referenced in the model**.

Results and Recommended Actions

Condition	Recommended Action
MATLAB Function block contains a character vector.	Consider using enumerations instead of character vectors.

Capabilities and Limitations

- This check does flags direct string substitutions only.
- Does not flag strings in MATLAB
- Does not flag character vectors that are hard-coded into the class definition.
- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

See Also

- MAB guideline na_0021: Strings in MATLAB functions
- JMAAB guideline na_0021

Check usage of recommended patterns for Switch/Case statements

Check ID: `mathworks.maab.na_0022`

Check for use of non-constant variables in Switch/Case statements.

Description

In generated code, MATLAB Function block inputs are passed as functional arguments. This check evaluates the Switch/Case statements in the generated code to determine if non-constant values are being used in the Case argument.

This check requires a Simulink Check license.

Check Parameterization

This Model Advisor check is not applicable for JMAAB modeling guidelines.

This check does not include sub-checks

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — Not supported

You can use Model Advisor Configuration Editor to configure this check to include files with a `.m` extension in the analysis. To enable this feature, in the **Input Parameters** section, select **Check external .m files referenced in the model**.

Results and Recommended Actions

Condition	Recommended Action
Non-constant variables are used in the Switch/ Case statement.	Consider defining the input variable as a constant.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.

See Also

- MAB guideline: na_0022: Recommended patterns for Switch/Case statements

Check for use of C-style comment symbols

Check ID: `mathworks.jmaab.jc_0801`

Description

Identifies the usage of C-style (/* and */) comments in CGT files and MPT objects.

Available with Simulink Check.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more entities use C-style comments.	Consider removing C-style comments in CGT files and MPT objects.

Capabilities and Limitations

- Does not run on library models.
- Does not analyze content of library linked blocks.
- Does not analyze content in masked subsystems.

- Does not allow exclusions of blocks or charts.

See Also

- MAB guideline jc_0801: Prohibited use of the /* and */ comment symbols
- JMAAB guideline jc_0801

Check usage of graphical functions in Stateflow

Check ID: `mathworks.jmaab.jc_0804`

Description

Identifies the graphical function calls made inside the graphical function.

This check requires Simulink Check and Stateflow licenses.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more calls to graphical functions are made inside a graphical function.	Remove calls to graphical functions made inside the graphical function.

Capabilities and Limitations

- Does not run on library models.
- Allows exclusions of blocks or charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.

See Also

- jc_0804: Prohibited use of recursive calls with graphical functions
- JMAAB guideline jc_0804

Check for division by zero in Simulink

Check ID: `mathworks.jmaab.jc_0794`

Description

Identifies division operations in Simulink that result in a divide-by-zero error.

This check requires Simulink Check, and Simulink Design Verifier (SLDV) licenses.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — a

Results and Recommended Actions

Condition	Recommended Action
One or more blocks performing division can result in a divide-by-zero error.	Remodel to avoid a divide-by-zero error.

Capabilities and Limitations

- Does not run on library models.
- Allows exclusions of blocks or charts.
- Analyzes content of library-linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `graphical`.

See Also

- MAB guideline jc_0794: Division in Simulink
- JMAAB guideline jc_0794

Check lines of code in MATLAB Functions

Check ID: `mathworks.jmaab.na_0016`

Description

Identify MATLAB Functions with high number of effective lines of code.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — Not Supported

Under **Input Parameters**, select **Check external .m files referenced in the model** to include files with a `.m` extension in the analysis.

Results and Recommended Actions

Condition	Recommended Action
One or more MATLAB Functions have many effective lines of code.	Remodel to reduce the number effective lines of code per MATLAB Function.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks or charts.

See Also

- MAB guideline na_0016: Source lines of MATLAB Functions

Check nested conditions in MATLAB Functions

Check ID: `mathworks.jmaab.na_0018`

Description

Checks for number of inputs/outputs to a Variant Subsystem.

This check requires a Simulink Check license.

Check Parameterization

This check does not include sub-checks because the MAB modeling guideline only provides one sub ID.

For reference, the MAB guideline sub ID(s) that are recommended for use by the NA-MAAB and JMAAB modeling standards organizations are:

- NA-MAAB — a
- JMAAB — Not Supported

Under **Input Parameters**, select **Check external .m files referenced in the model** to include files with a `.m` extension in the analysis.

Results and Recommended Actions

Condition	Recommended Action
One or more MATLAB Function found with deeply nested if/else and case statements.	Remodel to reduce the number of deeply nested conditional statements.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks. By default, the input parameter **Follow links** is set to `on`.
- Analyzes content in masked subsystems. By default, the input parameter **Look under masks** is set to `all`.
- Allows exclusions of blocks or charts.

See Also

- MAB guideline na_0018: Number of nested if/else and case statement

DO-254 Checks

In this section...

Modeling Standards for DO-254 Overview

DO-254 checks facilitate designing and troubleshooting models from which code is generated for applications that must meet safety or mission-critical requirements.

The Model Advisor performs a checkout of the Simulink Check license when you run the DO-254 checks.

These checks are qualified by the DO Qualification Kit for use in projects involving the DO-254 standard and related standards.

See Also

- “Simulink Checks”
- “Simulink Coder Checks” (Simulink Coder)
- Radio Technical Commission for Aeronautics (RTCA) for information on the DO-178C Software Considerations in Airborne Systems and Equipment Certification and related standards

Modeling Standards for DO-254

MISRA C:2012 Checks

In this section...

You can check that your model or subsystem has a likelihood of generating MISRA C:2012 compliant code.

See Also

- "Run Model Advisor Checks and Review Results"
- "Qualified Model Advisor Checks" (IEC Certification Kit)
- "Qualified Model Advisor Checks" (DO Qualification Kit)

Check usage of Assignment blocks

Check ID: `mathworks.misra.AssignmentBlocks`

Identify Assignment blocks that do not have block parameter **Action if any output element is not assigned** set to **Error** or **Warning**.

Description

This check applies to the Assignment block that is available in the Simulink block library under **Simulink** > **Math Operations**.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem might contain Assignment blocks with incomplete array initialization that do not have block parameter **Action if any output element is not assigned** set to **Error** or **Warning**.	Set block parameter **Action if any output element is not assigned** to one of the recommended values: • **Error**, if Assignment block is not in an Iterator subsystem. • **Warning**, if Assignment block is in an Iterator subsystem.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems.
- If you have a Simulink Check license, allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking. However, the following check condition is not supported because edit-time checking is unable to determine whether the Assignment block is in an Iterator subsystem.

Set block parameter **Action if any output element is not assigned** to one of the recommended values:

- **Error**, if Assignment block is not in an Iterator subsystem.
- **Warning**, if Assignment block is in an Iterator subsystem.

See Also

- MISRA C:2012, Rule 9.1
- ISO/IEC TS 17961: 2013, uninitref
- CERT C, EXP33-C
- CWE, CWE-908
- “hisl_0029: Usage of Assignment blocks”
- “MISRA C” (Embedded Coder)
- “MISRA C:2012 Compliance Considerations”

Check for blocks not recommended for MISRA C:2012

Check ID: `mathworks.misra.BlkSupport`

Identify blocks that are not supported or recommended for MISRA C:2012 compliant code generation.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Lookup Table blocks using cubic spline interpolation or extrapolation methods were found in the model or subsystem. Specific blocks are: • 1-D Lookup Table • 2-D Lookup Table • n-D Lookup Table	Consider other interpolation and extrapolation methods for the Lookup Table blocks.
Deprecated Lookup Table blocks were found in the model or subsystem. Specific blocks are: • Lookup Table • Lookup Table (2-D)	Consider replacing the deprecated Lookup Table blocks.
S-Function Builder blocks were found in the model or subsystem.	Consider replacing the S-Function Builder blocks with blocks recommended for production.
From Workspace blocks were found in the model or subsystem	Consider replacing the From Workspace blocks with blocks recommended for production.
String blocks were found in the model or subsystem. Specific blocks are: • Compose String • Scan String • String to Single • String to Double • To String	Consider replacing the String blocks with blocks recommended for production.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- “hisl_0020: Blocks not recommended for MISRA C:2012 compliance”
- “MISRA C” (Embedded Coder)
- “MISRA C:2012 Compliance Considerations”
- “Model Advisor Exclusion Overview”

Check for unsupported block names

Check ID: `mathworks.misra.BlockNames`

Identify block names containing `/`.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Block names containing `/` were found in the model or subsystem.	Remove `/` from the block name.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems.
- If you have a Simulink Check license, allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MISRA C:2012, Rule 3.1
- “MISRA C” (Embedded Coder)
- “MISRA C:2012 Compliance Considerations”

Check configuration parameters for MISRA C:2012

Check ID: `mathworks.misra.CodeGenSettings`

Identify configuration parameters that can impact MISRA C:2012 compliant code generation.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Math and Data Types	
Configuration parameter **Use division for fixed-point net slope computation** is not set to `On` or `Use division for reciprocals of integers only`.	Set **Use division for fixed-point net slope computation** to `On` or `Use division for reciprocals of integers only`.
Configuration parameter **Inf or NaN block output** is set to `None` or `error` and **Support non-finite numbers** is set to `on`. Configuration parameter **Inf or NaN block output** is set to `None` and **Support non-finite numbers** is set to `off`.	When **Support non-finite numbers** is: • `on`, set **Inf or NaN block output** to `warning` • `off`, set **Inf or NaN block output** to `warning` or `error`
Configuration parameter **Model Verification block enabling** is set to `Use local settings` or `Enable All`.	Set **Model Verification block enabling** to `Disable All`.
Configuration parameter **Undirected event broadcasts** is set to `none` or `warning`.	Set **Undirected event broadcasts** to `error`.
Configuration parameter **Wrap on overflow** is set to `None`	Set configuration parameter **Wrap on overflow** to `warning` or `error`.
Hardware Implementation	
Configuration parameter **Production hardware signed integer division rounds to** is set to `Undefined`	Set **Production hardware signed integer division rounds to** to `Zero` or `Floor`.
Configuration parameter **Shift right on a signed integer as arithmetic shift** is selected.	Clear **Shift right on a signed integer as arithmetic shift**.
Simulation Target	
Configuration parameter **Compile-time recursion limit for MATLAB functions** is set to a value other than `0`.	Set **Compile-time recursion limit for MATLAB functions** to `0`.
Configuration parameter **Dynamic memory allocation in MATLAB functions** is selected.	Clear **Dynamic memory allocation in MATLAB functions**.
Configuration parameter **Enable run-time recursion for MATLAB functions** is selected.	Clear **Enable run-time recursion for MATLAB functions**.
Code Generation	

Condition	Recommended Action
Configuration parameter **Bitfield declarator type specifier** is set to `uchar_T` when any of these parameters are selected: • **Pack Boolean data into bitfields** • **Use bitsets for storing state configuration** • **Use bitsets for storing Boolean data**	Set **Bitfield declarator type specifier** to `uint_T`.
Configuration parameter **Casting Modes** is not set to `Standards Compliant`.	Set **Casting Modes** to `Standards Compliant`.
Configuration parameter **Code replacement library** is not set to `None` or `AUTOSAR 4.0`.	Set **Code replacement library** to `None` or `AUTOSAR 4.0`
Configuration parameter **External mode** is selected.	Clear **External mode**.
Configuration parameter **Generate shared constants** is selected.	Clear **Generate shared constants**.
Configuration parameter **Include comments** is cleared.	Select **Include comments**.
Configuration parameter **MAT-file logging** is selected.	Clear **MAT-file logging**
For ERT-based target systems, configuration parameter **MATLAB user comments** is cleared.	Select **MATLAB user comments**.
A value for configuration parameter **Maximum identifier length** is not provided.	Set the value to the implementation-dependent limit. The default is `31`.
Configuration parameter **Parentheses level** is not set to `Standards(Parentheses for Standards Compliance)` or `Maximum(Specify precedence with parentheses)`.	Set **Parentheses level** to `Standards(Parentheses for Standards Compliance)` or `Maximum(Specify precedence with parentheses)`.
For ERT-based target systems, configuration parameter **Preserve static keyword in function declarations** is cleared when **File packaging format** is set to `Compact` or `Compact (with separate data file)`	Select **Preserve static keyword in function declarations**.
Configuration parameter **Replace multiplications by powers of two with signed bitwise shifts** is selected.	Clear **Replace multiplications by powers of two with signed bitwise shifts**.
Configuration parameter **Shared code placement** is set to `Auto`.	Set **Shared code placement** to `Shared location`
For ERT-based target systems, configuration parameter **Support continuous time** is selected	Clear **Support continuous time**.
For ERT-based target systems, configuration parameter **Support non-inlined S-functions** is selected	Clear **Support non-inlined S-functions**.

Condition	Recommended Action
Configuration parameter **System-generated identifiers** is set to `Classic`.	Set **System-generated identifiers** to `Shortened`.
Configuration parameter **System target file** is set to a GRT-based target.	Set **System target file** to an ERT-based target.
Configuration parameter **Use dynamic memory allocation for model initialization** is selected when **Code Interface Packaging** is set to `Reusable Function`.	Clear **Use dynamic memory allocation for model initialization**. **Note** Select only when **Code Interface Packaging** is set to `Reusable Function`.

Action Results

Clicking **Modify All** changes the parameter values to the recommended values.

Note When you click **Modify All** for models with a GRT-based target, the Model Advisor does not update the **System target file** configuration parameter to an ERT-based system.

Parameter subchecks depend on the results of the parameter noted with **D** in the results table. When the result is *D-Warning*, the **Current Value** column in the results table states *Prerequisite constraint not met* for the subchecks. After you change the parameter, rerun the check.

Note Some subchecks are specific to configuration parameters for ERT-based systems. These parameters are not updated when you click **Modify All** unless you change the model to an ERT-based system.

Capabilities and Limitations

Following parameters setting is informational in the check:

- `BooleansAsBitfields`
- `CodeInterfacePackaging`
- `ERTFilePackagingFormat`
- `SupportNonFinite`

This check does not review referenced models.

See Also

- hisl_0060: Configuration parameters that improve MISRA C:2012 compliance
- “MISRA C” (Embedded Coder)
- “MISRA C:2012 Compliance Considerations”

Check for equality and inequality operations on floating-point values

Check ID: `mathworks.misra.CompareFloatEquality`

Identify equality and inequality operations on floating-point values.

Description

The check flags sources causing equality or inequality operations on floating-point values.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

The check does not flag blocks with equality or inequality operations on floating-point values if they are justified with a Polyspace® annotation. When you run the check, the **Blocks with justification** table lists blocks with equality or inequality operations that have a justification.

Available with Embedded Coder and Simulink Check.

Note Stateflow license is needed if the model contains Stateflow elements.

Results and Recommended Actions

Condition	Recommended Action
Model object has an equality or inequality operation on a floating-point value.	Consider using non-floating-point values for equality or inequality operations.

Capabilities and Limitations

You can:

- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C:2012, Dir 1.1
- CERT C, FLP02-C
- CWE, CWE-1077
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)
- “MISRA C” (Embedded Coder)

Check for bitwise operations on signed integers

Check ID: `mathworks.misra.CompliantCGIRConstructions`

Identify Simulink blocks that contain bitwise operations on signed integers.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

Available with Embedded Coder and Simulink Check.

Note Stateflow license is needed if the model contains Stateflow elements.

Results and Recommended Actions

Condition	Recommended Action
The model has blocks that contain bitwise operations on signed integers.	Consider using unsigned integers for bitwise operations.

Capabilities and Limitations

You can:

- The check assumes that code is generated for the whole model. When code is generated by a subsystem build or export functions, the check can product incorrect results.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C:2012, Rule 10.1
- CERT C, INT13-C
- CWE, CWE-682
- “hisl_0060: Configuration parameters that improve MISRA C:2012 compliance”
- “MISRA C:2012 Compliance Considerations”
- “MISRA C” (Embedded Coder)

Check for recursive function calls

Check ID: `mathworks.misra.RecursionCompliance`

Identify recursive function calls in Stateflow charts.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications. The check flags charts that have recursive function calls.

Available with Embedded Coder and Simulink Check.

Note Stateflow license is needed if the model contains Stateflow elements.

Results and Recommended Actions

Condition	Recommended Action
Chart has a recursive function call.	Remove recursive function call.

See Also

- MISRA C:2012, Rule 17.2
- “Avoid Unwanted Recursion in a Chart” (Stateflow)

Check for switch case expressions without a default case

Check ID: `mathworks.misra.SwitchDefault`

Identify switch case expressions that do not have a default case.

Description

The check flags model objects that have switch case expressions without a default case.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

The check does not flag blocks without default cases if they are justified with a Polyspace annotation. When you run the check, the **Blocks with justification** table lists blocks without default cases that have a justification.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Model object has a switch case expression without a default case.	For Switch Case blocks, consider selecting block parameter **Show default case** to explicitly specify a default case.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MISRA C:2012, Rule 16.4
- ISO/IEC TS 17961: 2013, swtchdflt
- CERT C, MSC01-C
- CWE, CWE-478
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)

- “MISRA C” (Embedded Coder)

Check for blocks not recommended for C/C++ production code deployment

Check ID: `mathworks.codegen.PCGSupport`

Identify blocks not supported by code generation or not recommended for C/C++ production code deployment.

Description

This check partially identifies model constructs that are not recommended for C/C++ production code generation. For Simulink Coder and Embedded Coder, these model construct identities appear in tables of Simulink Block Support (Simulink Coder).

In some instances, this check flags blocks that are supported for code generation. For these blocks, you should review the footnote information that is provided in the support notes and adhere to the recommended action provided by the Model Advisor.

Following the recommendations of this check increases the likelihood of generating code that complies with the CERT C, CWE, and ISO/IEC TS 17961 standards.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks that should not be used for production code deployment.	Consider replacing the blocks listed in the results. Click an element from the list of questionable items to locate condition.
The model or subsystem contains blocks that are supported but not recommended for production code generation.	Review the support notes and adhere to the recommended action provided by the Model Advisor.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Analyze content of library linked blocks.
- Analyze content in masked subsystems.
- Exclude blocks and charts if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- “Use Blocks and Products Supported for Code Generation” (Simulink Coder)

- “Model Advisor Exclusion Overview”
- Secure Coding Standards (Embedded Coder) “Secure Coding” (Embedded Coder)

Check for missing error ports for AUTOSAR receiver interfaces

Check ID: `mathworks.misra.AutosarReceiverInterface`

Identify AUTOSAR receiver interface inports that do not have matching error ports.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications. The check flags AUTOSAR receiver interfaces inports that are missing error ports. The following table identifies the AUTOSAR data access mode types for receiver interface ports that are flagged by the check when the corresponding error port is missing.

AUTOSAR Data Access Mode Type	Flagged by Check?
`ImplicitReceive`	Yes
`ExplicitReceive`	Yes
`QueuedExplicitReceive`	No
`ErrorStatus`	No
`ModeReceive`	No
`IsUpdated`	No
`EndToEndRead`	Yes
`ExplicitReceiveByVal`	No
`otherwise`	No

The check does not flag missing error ports when they are justified with a Polyspace annotation. When you run the check, the **Blocks with justification** table lists the missing error ports that have a justification.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
AUTOSAR receiver interface inport does not have a matching error port.	Add missing error port and map to the corresponding AUTOSAR receiver interface inport.
AUTOSAR receiver interface ports do not have a matching error port when data access mode is `ImplicitReceive`, `ExplicitReceive`, or `EndToEndRead`.	Add missing error port and map to the corresponding AUTOSAR receiver interface inport.

Capabilities and Limitations

You can:

- Analyzes top layer/root level models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C: 2012, Directive 4.7
- “MISRA C” (Embedded Coder)
- “Model Advisor Exclusion Overview”
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)
- “Configure AUTOSAR Elements and Properties” (AUTOSAR Blockset)
- “AUTOSAR Component Configuration” (AUTOSAR Blockset)

Check for missing const qualifiers in model functions

Check ID: `mathworks.misra.ModelFunctionInterface`

Identify missing const qualifiers in input data pointers.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications. The check flags input data pointers that do not have a const qualifier.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
A const qualifier is not defined for the input data pointer.	Consider adding a const qualifier to the input data pointer.

See Also

- MISRA C:2012, Rule 8.13
- “MISRA C” (Embedded Coder)

Check integer word length

Check ID: `mathworks.misra.IntegerWordLengths`

Identify integer word lengths that do not comply with hardware implementation settings

Description

The check flags integers whose word lengths exceed the number of bits permitted via the hardware implementation settings.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

Available with Embedded Coder and Simulink Check.

Note Stateflow license is needed if the model contains Stateflow elements.

Results and Recommended Actions

Condition	Recommended Action
Model object contains integer word lengths that are not compliant with hardware implementation settings.	Update the integer so its length does not exceed the permitted number of bits. You can view the permitted number of bits in the Configuration Parameters dialog box, on the **Hardware Implementation** > **Device details** pane.

Capabilities and Limitations

You can:

- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C:2012, Rule 10.1
- CERT C, INT13-C
- CWE, CWE-682
- “MISRA C” (Embedded Coder)
- “Model Advisor Exclusion Overview”

Check bus object names that are used as bus element names

Check ID: `mathworks.misra.BusElementNames`

Identify bus object names that are used as bus element names.

Description

Using this check increases the likelihood of generating code for embedded applications that is compliant with MISRA C:2012. The check flags instances where a Simulink.Bus object name is used as the Simulink.Bus element name.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
A bus object name is being used as a bus element name.	Change either the flagged bus object name or the bus element name so that they are not identical.

See Also

- MISRA C:2012, Rule 5.6
- MISRA AC AGC, Rule 5.3
- “MISRA C” (Embedded Coder)

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

In this section...

These checks are used to validate that code generated by Embedded Coder complies with the CERT C, CWE, and ISO/IEC TS 17961 (Embedded Coder) secure coding standards.

See Also

- “Run Model Advisor Checks and Review Results”

Check configuration parameters for secure coding standards

Check ID: `mathworks.security.CodeGenSettings`

Identify configuration parameters that might impact compliance with secure coding standards.

Description

Following the recommendations of this check increases the likelihood of generating code that complies with CERT C, CWE, ISO/IEC TS 17961 secure coding standards.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Diagnostics	

Condition	Recommended Action
Configuration parameter **Inf or NaN block output** is set to `None` or `error` and **Support non-finite numbers** is set to `on`. Configuration parameter **Inf or NaN block output** is set to `None` and **Support non-finite numbers** is set to `off`.	When **Support non-finite numbers** is: • `on`, set **Inf or NaN block output** to `warning`. • `off`, set **Inf or NaN block output** to `warning` or `error`.
Configuration parameter **Model Verification block enabling** is set to `Use local settings` or `Enable All`.	Set **Model Verification block enabling** to `Disable All`.
Configuration parameter **Undirected event broadcasts** is set to `none` or `warning`.	Set **Undirected event broadcasts** to `error`.
Configuration parameter **Wrap on overflow** is set to `none`.	Set **Wrap on overflow** to `warning` or `error`.
Hardware Implementation	
Configuration parameter **Production hardware signed integer division rounds to** is set to `Undefined`.	Set **Production hardware signed integer division rounds to** to `Zero` or `Floor`.
Configuration parameter **Shift right on a signed integer as arithmetic shift** is selected.	Clear **Shift right on a signed integer as arithmetic shift**.
Simulation Target	
Configuration parameter **Compile-time recursion limit for MATLAB functions** is set to a value other than `0`.	Set **Compile-time recursion limit for MATLAB functions** to `0`.
Configuration parameter **Dynamic memory allocation in MATLAB functions** is selected.	Clear **Dynamic memory allocation in MATLAB functions**.
Configuration parameter **Enable run-time recursion for MATLAB functions** is selected.	Clear **Enable run-time recursion for MATLAB functions**.
Code Generation	
Configuration parameter **Code replacement library** is not set to `None` or `AUTOSAR 4.0`.	Set **Code replacement library** to `None` or `AUTOSAR 4.0`.
Configuration parameter **External mode** is selected.	Clear **External mode**.
Configuration parameter **Include comments** is cleared.	Select **Include comments**.
Configuration parameter **MAT-file logging** is selected.	Clear **MAT-file logging**.
For ERT-based target systems, configuration parameter **MATLAB user comments** is cleared.	Select **MATLAB user comments**.
Configuration parameter **Replace multiplications by powers of two with signed bitwise shifts** is selected.	Clear **Replace multiplications by powers of two with signed bitwise shifts**.

Condition	Recommended Action
For ERT-based target systems, configuration parameter **Support continuous time** is selected	Clear **Support continuous time**.
For ERT-based target systems, configuration parameter **Support non-inlined S-functions** is selected	Clear **Support non-inlined S-functions**.
Configuration parameter **System target file** is set to a GRT-based target.	Set **System target file** to an ERT-based target.
Configuration parameter **Use dynamic memory allocation for model initialization** is selected.	Clear **Use dynamic memory allocation for model initialization**. **Note** Select only when **Code Interface Packaging** is set to `Reusable Function`.

Action Results

Clicking **Modify All** changes the parameter values to the recommended values.

Note When you click **Modify All** for models with a GRT-based target, the Model Advisor does not update the **System target file** configuration parameter to an ERT-based system.

Parameter subchecks depend on the results of the parameter noted with **D** in the results table. When the result is *D-Warning*, the **Current Value** column in the results table states *Prerequisite constraint not met* for the subchecks. After you change the parameter, rerun the check.

Note Some subchecks are specific to configuration parameters for ERT-based systems. These parameters are not updated when you click **Modify All** unless you change the model to an ERT-based system.

See Also

“MISRA C” (Embedded Coder)

Check for blocks not recommended for C/C++ production code deployment

Check ID: `mathworks.codegen.PCGSupport`

Identify blocks not supported by code generation or not recommended for C/C++ production code deployment.

Description

This check partially identifies model constructs that are not recommended for C/C++ production code generation. For Simulink Coder and Embedded Coder, these model construct identities appear in tables of Simulink Block Support (Simulink Coder).

In some instances, this check flags blocks that are supported for code generation. For these blocks, you should review the footnote information that is provided in the support notes and adhere to the recommended action provided by the Model Advisor.

Following the recommendations of this check increases the likelihood of generating code that complies with the CERT C, CWE, and ISO/IEC TS 17961 standards.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem contains blocks that should not be used for production code deployment.	Consider replacing the blocks listed in the results. Click an element from the list of questionable items to locate condition.
The model or subsystem contains blocks that are supported but not recommended for production code generation.	Review the support notes and adhere to the recommended action provided by the Model Advisor.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Analyze content of library linked blocks.
- Analyze content in masked subsystems.
- Exclude blocks and charts if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- “Use Blocks and Products Supported for Code Generation” (Simulink Coder)
- “Model Advisor Exclusion Overview”
- Secure Coding Standards (Embedded Coder) “Secure Coding” (Embedded Coder)

Check for blocks not recommended for secure coding standards

Check ID: `mathworks.security.BlockSupport`

Identify blocks not recommended for compliance with secure coding standards.

Description

Following the recommendations of this check increases the likelihood of generating code that complies with CERT C, CWE, ISO/IEC TS 17961 secure coding standards.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Lookup Table blocks using cubic spline interpolation or extrapolation methods were found in the model or subsystem. Specific blocks are: • 1-D Lookup Table • 2-D Lookup Table • n-D Lookup Table	Consider other interpolation and extrapolation methods for the Lookup Table blocks.
Deprecated Lookup Table blocks were found in the model or subsystem. Specific blocks are: • Lookup Table • Lookup Table (2-D)	Consider replacing the deprecated Lookup Table blocks.
S-Function Builder blocks were found in the model or subsystem.	Consider replacing the S-Function Builder blocks with blocks recommended for production.
From Workspace blocks were found in the model or subsystem	Consider replacing the From Workspace blocks with blocks recommended for production.
String blocks were found in the model or subsystem. Specific blocks are: • Compose String • Scan String • String to Single • String to Double • To String	Consider replacing the String blocks with blocks recommended for production.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- “Model Advisor Exclusion Overview”
- “MISRA C” (Embedded Coder)

Check usage of Assignment blocks

Check ID: `mathworks.misra.AssignmentBlocks`

Identify Assignment blocks that do not have block parameter **Action if any output element is not assigned** set to **Error** or **Warning**.

Description

This check applies to the Assignment block that is available in the Simulink block library under **Simulink > Math Operations**.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
The model or subsystem might contain Assignment blocks with incomplete array initialization that do not have block parameter **Action if any output element is not assigned** set to **Error** or **Warning**.	Set block parameter **Action if any output element is not assigned** to one of the recommended values: • **Error**, if Assignment block is not in an Iterator subsystem. • **Warning**, if Assignment block is in an Iterator subsystem.

Capabilities and Limitations

- Runs on library models.
- Analyzes content of library linked blocks.
- Analyzes content in masked subsystems.
- If you have a Simulink Check license, allows exclusions of blocks and charts.

Edit-Time Checking

This check is supported by edit-time checking. However, the following check condition is not supported because edit-time checking is unable to determine whether the Assignment block is in an Iterator subsystem.

Set block parameter **Action if any output element is not assigned** to one of the recommended values:

- **Error**, if Assignment block is not in an Iterator subsystem.
- **Warning**, if Assignment block is in an Iterator subsystem.

See Also

- MISRA C:2012, Rule 9.1
- ISO/IEC TS 17961: 2013, uninitref
- CERT C, EXP33-C
- CWE, CWE-908

- “hisl_0029: Usage of Assignment blocks”
- “MISRA C” (Embedded Coder)
- “MISRA C:2012 Compliance Considerations”

Check for switch case expressions without a default case

Check ID: `mathworks.misra.SwitchDefault`

Identify switch case expressions that do not have a default case.

Description

The check flags model objects that have switch case expressions without a default case.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

The check does not flag blocks without default cases if they are justified with a Polyspace annotation. When you run the check, the **Blocks with justification** table lists blocks without default cases that have a justification.

Available with Embedded Coder and Simulink Check.

Results and Recommended Actions

Condition	Recommended Action
Model object has a switch case expression without a default case.	For Switch Case blocks, consider selecting block parameter **Show default case** to explicitly specify a default case.

Capabilities and Limitations

You can:

- Run this check on your library models.
- Exclude blocks and charts from this check if you have a Simulink Check license.

Edit-Time Checking

This check is supported by edit-time checking.

See Also

- MISRA C:2012, Rule 16.4
- ISO/IEC TS 17961: 2013, swtchdflt
- CERT C, MSC01-C
- CWE, CWE-478
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)
- “MISRA C” (Embedded Coder)

Check for bitwise operations on signed integers

Check ID: `mathworks.misra.CompliantCGIRConstructions`

Identify Simulink blocks that contain bitwise operations on signed integers.

Description

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

Available with Embedded Coder and Simulink Check.

Note Stateflow license is needed if the model contains Stateflow elements.

Results and Recommended Actions

Condition	Recommended Action
The model has blocks that contain bitwise operations on signed integers.	Consider using unsigned integers for bitwise operations.

Capabilities and Limitations

You can:

- The check assumes that code is generated for the whole model. When code is generated by a subsystem build or export functions, the check can product incorrect results.
- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C:2012, Rule 10.1
- CERT C, INT13-C
- CWE, CWE-682
- “hisl_0060: Configuration parameters that improve MISRA C:2012 compliance”
- “MISRA C:2012 Compliance Considerations”
- “MISRA C” (Embedded Coder)

Check for equality and inequality operations on floating-point values

Check ID: `mathworks.misra.CompareFloatEquality`

Identify equality and inequality operations on floating-point values.

Description

The check flags sources causing equality or inequality operations on floating-point values.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

The check does not flag blocks with equality or inequality operations on floating-point values if they are justified with a Polyspace annotation. When you run the check, the **Blocks with justification** table lists blocks with equality or inequality operations that have a justification.

Available with Embedded Coder and Simulink Check.

Note Stateflow license is needed if the model contains Stateflow elements.

Results and Recommended Actions

Condition	Recommended Action
Model object has an equality or inequality operation on a floating-point value.	Consider using non-floating-point values for equality or inequality operations.

Capabilities and Limitations

You can:

- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C:2012, Dir 1.1
- CERT C, FLP02-C
- CWE, CWE-1077
- “Annotate Code and Hide Known or Acceptable Results” (Polyspace Bug Finder)
- “MISRA C” (Embedded Coder)

Check integer word length

Check ID: `mathworks.misra.IntegerWordLengths`

Identify integer word lengths that do not comply with hardware implementation settings

Description

The check flags integers whose word lengths exceed the number of bits permitted via the hardware implementation settings.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

Available with Embedded Coder and Simulink Check.

Note Stateflow license is needed if the model contains Stateflow elements.

Results and Recommended Actions

Condition	Recommended Action
Model object contains integer word lengths that are not compliant with hardware implementation settings.	Update the integer so its length does not exceed the permitted number of bits. You can view the permitted number of bits in the Configuration Parameters dialog box, on the **Hardware Implementation** > **Device details** pane.

Capabilities and Limitations

You can:

- Exclude blocks and charts from this check if you have a Simulink Check license.

See Also

- MISRA C:2012, Rule 10.1
- CERT C, INT13-C
- CWE, CWE-682
- “MISRA C” (Embedded Coder)
- “Model Advisor Exclusion Overview”

Detect Dead Logic

Check ID: `mathworks.sldv.deadlogic`

Identify logic that stays inactive during simulation.

Description

This check identifies portions of your model that stay inactive during simulation.

You can run a more detailed analysis that identifies both dead logic and active logic using Simulink Design Verifier design error detection. For more information, see “Detect Dead Logic Caused by an Incorrect Value” (Simulink Design Verifier).

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards

Results and Recommended Actions

Result	Recommended Action
Failed, model incompatible	Resolve the model incompatibility. See: • "Supported and Unsupported Simulink Blocks in Simulink Design Verifier" (Simulink Design Verifier) • "Support Limitations for Model Blocks" (Simulink Design Verifier) • "Support Limitations for Simulink Software Features" (Simulink Design Verifier) • "Support Limitations for Stateflow Software Features" (Simulink Design Verifier) • "Support Limitations for MATLAB for Code Generation" (Simulink Design Verifier) Also see "Handle Incompatibilities with Automatic Stubbing" (Simulink Design Verifier).
Dead logic found in model	Simulink Design Verifier proved that these decision and condition outcomes cannot occur and are dead logic in the model. Dead logic can also be a side effect of specified constraints on parameters or specified minimum and maximum constraints on input ports. In rare cases, dead logic can result from approximations performed by Simulink Design Verifier. It is possible that there are objectives that this analysis did not decide. To extend the results of this analysis, use Simulink Design Verifier design error detection to also identify active logic. From the Simulink Editor, select **Apps > Design Verifier > Settings**. In the **Configuration Parameters** window, from **Design Verifier > Design Error Detection** pane, select both **Dead logic** and **Identify active logic**.
Dead logic not found in model	Simulink Design Verifier did not find dead logic in the model. It is possible that there are objectives that this analysis did not decide. To extend the results of this analysis, use Simulink Design Verifier design error detection to also identify active logic. From the Simulink Editor, select **Apps > Design Verifier > Settings**. In the **Configuration Parameters** window, from **Design Verifier > Design Error Detection** pane, select both **Dead logic** and **Identify active logic**.

See Also

- MISRA C:2012: Rule 2.1
- CERT C, MSC07-C
- CWE, CWE-561
- “Run Model Advisor Checks”
- “Secure Coding” (Embedded Coder)
- “Detect Dead Logic Caused by an Incorrect Value” (Simulink Design Verifier)
- “Design Verifier Pane: Design Error Detection” (Simulink Design Verifier)

Detect Integer Overflow

Check ID: `mathworks.sldv.integeroverflow`

Detects integer or fixed-point data overflow errors in your model

Description

This check identifies operations that exceed the data type range for integer or fixed-point operations.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

Results and Recommended Actions

Result	Recommended Action
Failed, model incompatible	Resolve the model incompatibility. See • “Supported and Unsupported Simulink Blocks in Simulink Design Verifier” (Simulink Design Verifier) • “Support Limitations for Model Blocks” (Simulink Design Verifier) • “Support Limitations for Simulink Software Features” (Simulink Design Verifier) • “Support Limitations for Stateflow Software Features” (Simulink Design Verifier) • “Support Limitations for MATLAB for Code Generation” (Simulink Design Verifier) Also see “Handle Incompatibilities with Automatic Stubbing” (Simulink Design Verifier).
Integer overflow found in model	To view the conditions that cause the integer overflow, create a harness model. When you simulate the harness, the inputs replicate the error. Click **View test case** in the Model Advisor report.

See Also

- MISRA C:2012: Directive 4.1
- ISO/IEC TS 17961: 2013, intoflow
- CERT C, INT30-C and INT32-C
- CWE, CWE-190
- “Secure Coding” (Embedded Coder)
- “Detect and Address Bugs” (Simulink Design Verifier)
- “Detect Integer Overflow and Division-by-Zero Errors” (Simulink Design Verifier)

Detect Division by Zero

Check ID: `mathworks.sldv.divbyzero`

Detects division-by-zero errors in your model

Description

This check identifies operations in your model that cause division-by-zero errors.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

Results and Recommended Actions

Result	Recommended Action
Failed, model incompatible	Resolve the model incompatibility. See • “Supported and Unsupported Simulink Blocks in Simulink Design Verifier” (Simulink Design Verifier) • “Support Limitations for Model Blocks” (Simulink Design Verifier) • “Support Limitations for Simulink Software Features” (Simulink Design Verifier) • “Support Limitations for Stateflow Software Features” (Simulink Design Verifier) • “Support Limitations for MATLAB for Code Generation” (Simulink Design Verifier) Also see “Handle Incompatibilities with Automatic Stubbing” (Simulink Design Verifier).
Division by zero found in model	To view the conditions that cause the division by zero, create a harness model. When you simulate the harness, the inputs replicate the error. Click **View test case** in the Model Advisor report.

See Also

- MISRA C:2012: Directive 4.1
- ISO/IEC TS 17961: 2013, diverr
- CERT C, INT33-C and FLP03-C
- CWE, CWE-369
- “Secure Coding” (Embedded Coder)
- “Detect and Address Bugs” (Simulink Design Verifier)
- “Detect Integer Overflow and Division-by-Zero Errors” (Simulink Design Verifier)

Detect Out Of Bound Array Access

Check ID: `mathworks.sldv.arraybounds`

Detects operations that access outside the bounds of an array index

Description

This check detects instances of out of bound array access in Simulink Design Verifier.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C, CWE, ISO/IEC TS 17961 standards.

Results and Recommended Actions

Result	Recommended Action
Failed, model incompatible	Resolve the model incompatibility. See • “Supported and Unsupported Simulink Blocks in Simulink Design Verifier” (Simulink Design Verifier) • “Support Limitations for Model Blocks” (Simulink Design Verifier) • “Support Limitations for Simulink Software Features” (Simulink Design Verifier) • “Support Limitations for Stateflow Software Features” (Simulink Design Verifier) • “Support Limitations for MATLAB for Code Generation” (Simulink Design Verifier) Also see “Handle Incompatibilities with Automatic Stubbing” (Simulink Design Verifier).
Out of bound array access found in model	To view the conditions that cause the out of bound array access, create a harness model. When you simulate the harness, the inputs replicate the error. Click **View test case** in the Model Advisor report.

See Also

- MISRA C:2012: Rule 18.1
- ISO/IEC TS 17961: 2013, invptr
- CERT C, ARR30-C
- CWE, CWE-118
- “Secure Coding” (Embedded Coder)
- “Detect and Address Bugs” (Simulink Design Verifier)
- “Detect Out of Bound Array Access Errors” (Simulink Design Verifier)

Detect Specified Minimum and Maximum Value Violations

Check ID: `mathworks.sldv.minmax`

Detect signals which exceed specified minimum and maximum values

Description

This analysis checks the specified minimum and maximum values (the design ranges) on intermediate signals throughout the model and on the output ports. If the analysis detects that a signal exceeds the design range, the results identify where in the model the errors occurred.

Following the recommendations of this check increases the likelihood of generating MISRA C:2012 compliant code for embedded applications, as well as code that complies with the CERT C and CWE standards.

Results and Recommended Actions

Result	Recommended Action
Failed, model incompatible	Resolve the model incompatibility. See • “Supported and Unsupported Simulink Blocks in Simulink Design Verifier” (Simulink Design Verifier) • “Support Limitations for Model Blocks” (Simulink Design Verifier) • “Support Limitations for Simulink Software Features” (Simulink Design Verifier) • “Support Limitations for Stateflow Software Features” (Simulink Design Verifier) • “Support Limitations for MATLAB for Code Generation” (Simulink Design Verifier) Also see “Handle Incompatibilities with Automatic Stubbing” (Simulink Design Verifier).

Result	Recommended Action
Violation of minimum and/or maximum found in model	To view the conditions that cause the violation, create a harness model. When you simulate the harness, the inputs replicate the error. Click **View test case** in the Model Advisor report.

See Also

- MISRA C:2012: Directive 4.1
- CERT C, API00-C
- CWE, CWE-628
- “Secure Coding” (Embedded Coder)
- “Detect and Address Run-Time Errors” (Simulink Design Verifier)
- “Check for Specified Minimum and Maximum Value Violations” (Simulink Design Verifier)

Model Metrics

Model Metrics

Model metrics analyze your model and help you assess your model with regard to size, architecture, readability, and compliance to standards. Simulink Check provides the metrics for these metric types:

- “Size Metrics” on page 2-300
- “Architecture Metrics” on page 2-301
- “Compliance Metrics” on page 2-301
- “Readability Metrics” on page 2-302

Using the Metrics Dashboard, you can collect and view model metrics to get an assessment of your project quality status. For more information, see “Collect and Explore Metric Data by Using the Metrics Dashboard”.

You can use the model metric API to run the model metrics programmatically and export the results to a file. For more information, see “Collect Model Metrics Programmatically”.

For your company guidelines and standards, you can also use the model metric API to create your own model metrics, compute those metrics, and export the metric data. For more information, see “Create a Custom Model Metric for Nonvirtual Block Count”.

To obtain the metric IDs for the available metrics in your metric engine, use `slmetric.metric.getAvailableMetrics`.

```
availableMetricIDs = slmetric.metric.getAvailableMetrics()
```

```
availableMetricIDs =

  26×1 cell array

    {'mathworks.metrics.CloneContent'                          }
    {'mathworks.metrics.CloneDetection'                        }
    {'mathworks.metrics.CyclomaticComplexity'                  }
    {'mathworks.metrics.DescriptiveBlockNames'                 }
    {'mathworks.metrics.DiagnosticWarningsCount'               }
    {'mathworks.metrics.ExplicitIOCount'                       }
    {'mathworks.metrics.FileCount'                             }
    {'mathworks.metrics.IOCount'                               }
    {'mathworks.metrics.LayerSeparation'                       }
    {'mathworks.metrics.LibraryContent'                        }
    {'mathworks.metrics.LibraryLinkCount'                      }
    {'mathworks.metrics.MatlabCodeAnalyzerWarnings'            }
    {'mathworks.metrics.MatlabFunctionCount'                   }
    {'mathworks.metrics.MatlabLOCCount'                        }
    {'mathworks.metrics.ModelAdvisorCheckCompliance.hisl_do178'}
    {'mathworks.metrics.ModelAdvisorCheckCompliance.maab'      }
    {'mathworks.metrics.ModelAdvisorCheckIssues.hisl_do178'    }
    {'mathworks.metrics.ModelAdvisorCheckIssues.maab'          }
    {'mathworks.metrics.ModelFileCount'                        }
    {'mathworks.metrics.ParameterCount'                        }
    {'mathworks.metrics.SimulinkBlockCount'                    }
```

```
{'mathworks.metrics.StateflowChartCount'                  }
{'mathworks.metrics.StateflowChartObjectCount'            }
{'mathworks.metrics.StateflowLOCCount'                    }
{'mathworks.metrics.SubSystemCount'                       }
{'mathworks.metrics.SubSystemDepth'                       }
```

`slmetric.metric.getAvailableMetrics()` returns the available metrics in your metrics engine, including custom metrics. For information on how to configure and use custom metrics to customize the dashboard, see “Customize Metrics Dashboard Layout and Functionality”.

Size Metrics

To collect metric data on a model or subsystem, run these metrics.

Metric	Description
“Simulink block metric” on page 2-302	Calculates the number of blocks in the model.
“Subsystem metric” on page 2-303	Calculates the number of subsystems in the model.
“Library link metric” on page 2-304	Calculates the number of library-linked blocks in the model.
“Effective lines of MATLAB code metric” on page 2-305	Calculates the number of effective lines of MATLAB code.
“Stateflow chart objects metric” on page 2-306	Calculates the number of Stateflow objects.
“Lines of code for Stateflow blocks metric” on page 2-307	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
“Subsystem depth metric” on page 2-308	Calculates the subsystem depth of the model.
“Input output metric” on page 2-309	Calculates the number of inputs and outputs in your model.
“Explicit input output metric” on page 2-311	Calculates the number of inputs and outputs in your model.
“File metric” on page 2-311	Calculates the number of model and library files.
“MATLAB Function metric” on page 2-312	Calculates the number of MATLAB Function blocks in your model.
“Model file count” on page 2-313	Calculates the number of model files.
“Parameter metric” on page 2-313	Calculates the number of instances of data objects that parameterize the behavior of a model.
“Stateflow chart metric” on page 2-314	Calculates the number of Stateflow charts in your model.

For more information on model metrics, see “Collect Model and Testing Metrics”.

Architecture Metrics

To learn more about the architecture for a model or subsystem, run these metrics.

Metric	Description
“Cyclomatic complexity metric” on page 2-315	Calculates the cyclomatic complexity of the model.
“Clone content metric” on page 2-317	Calculates the fraction of total number of subcomponents that are clones.
“Clone detection metric” on page 2-317	Calculates the number of clones in components across the model hierarchy.
“Library content metric” on page 2-318	Calculates the fraction of total number of components that are linked library blocks.

For more information on model metrics, see “Collect Model and Testing Metrics”.

Compliance Metrics

To determine if your model or subsystem is compliant with standards and guidelines, run one or more of these metrics.

Metric	Description
“MATLAB code analyzer warnings” on page 2-321	Determines warnings for MATLAB code blocks in your model.
“Diagnostic warnings metric” on page 2-310	Calculates the number of diagnostic warnings reported.
“Model Advisor Check Compliance for High-Integrity Systems” on page 2-321	Returns the fraction of checks the model passes from Model Advisor DO-178C/DO-331 Standards.
“Model Advisor Check Compliance for Modeling Standards for MAB” on page 2-322	Returns the fraction of checks the model passes from Model Advisor MAB Standard.
“Model Advisor Check Issues for High-Integrity Systems” on page 2-323	Reports the number of issues from Model Advisor DO-178C/ DO-331 Standards.
“Model Advisor Check Issues for MAB Standards” on page 2-324	Reports the number of issues from Model Advisor MAAB Standard.

For information on compliance metrics that obtain compliance and issues metric data on your Model Advisor configuration, see “Compliance Metrics for Model Advisor Configurations” on page 2-325.

For more information on model metrics, see “Collect Model and Testing Metrics”.

Readability Metrics

Run these metrics to determine readability for a model or subsystem.

Metric	Description
"Nondescriptive block name metric" on page 2-318	Determines nondescriptive Inport, Outport, and Subsystem block names.
"Data and structure layer separation metric" on page 2-320	Calculates the data and structure layer separation.

For more information on model metrics, see "Collect Model and Testing Metrics".

Simulink block metric

Metric Type: Size

Metric ID: `mathworks.metrics.SimulinkBlockCount`

Model Advisor Check ID: `mathworks.metricchecks.SimulinkBlockCount`

Calculate the number of Simulink blocks in the model.

Description

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`.

Model Advisor Check

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 `sldiagnostics`. 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.

Capabilities and Limitations

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Subsystem metric

Metric Type: Size

Metric ID: `mathworks.metrics.SubSystemCount`

Model Advisor Check ID: `mathworks.metricchecks.SubSystemCount`

Display number of subsystems in the model.

Description

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`.

Model Advisor Check

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.

Capabilities and Limitations

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.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Library link metric

Metric Type: Size

Metric ID: `mathworks.metrics.LibraryLinkCount`

Model Advisor Check ID: `mathworks.metricchecks.LibraryLinkCount`

Display number of library links in the model.

Description

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`.

Model Advisor Check

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.

Capabilities and Limitations

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.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Effective lines of MATLAB code metric

Metric Type: Size

Metric ID: `mathworks.metrics.MatlabLOCCount`

Model Advisor Check ID: `mathworks.metricchecks.MatlabLOCCount`

Display number of effective lines of MATLAB code.

Description

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`.

Model Advisor Check

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.

Capabilities and Limitations

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.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Stateflow chart objects metric

Metric Type: Size

Metric ID: `mathworks.metrics.StateflowChartObjectCount`

Model Advisor Check ID: `mathworks.metricchecks.StateflowChartObjectCount`

Display the number of Stateflow objects in each chart.

Description

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`.

Model Advisor Check

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.

Capabilities and Limitations

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Lines of code for Stateflow blocks metric

Metric Type: Size

Metric ID: `mathworks.metrics.StateflowLOCCount`

Model Advisor Check ID: `mathworks.metricchecks.StateflowLOCCount`

Display the number of effective lines of code for Stateflow blocks.

Description

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`.

Model Advisor Check

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.

Capabilities and Limitations

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Subsystem depth metric

Metric Type: Size

Metric ID: `mathworks.metrics.SubSystemDepth`

Model Advisor Check ID: `mathworks.metricchecks.SubSystemDepth`

Calculates the depth of the hierarchical children of a subsystem or model.

Description

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`.

Model Advisor Check

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.

Capabilities and Limitations

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Input output metric

Metric Type: Size

Metric ID: `mathworks.metrics.IOCount`

Display number of inputs and outputs in the model.

Description

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`.

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.

Capabilities and Limitations

The metric:

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Diagnostic warnings metric

Metric Type: Compliance

Metric ID: `mathworks.metrics.DiagnosticWarningsCount`

Calculate the number of diagnostic warnings reported during a model update for simulation.

Description

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`.

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.

Capabilities and Limitations

- If specified, analyzes the content of library-linked blocks or referenced models.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Explicit input output metric

Metric Type: Size

Metric ID: `mathworks.metrics.ExplicitIOCount`

Display number of inputs and outputs in the model, excluding From and Goto blocks.

Description

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`.

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.

Capabilities and Limitations

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.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

File metric

Metric Type: Size

Metric ID: `mathworks.metrics.FileCount`

Calculates the number of model and library files used by a specific component and its subcomponents.

Description

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`.

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.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

MATLAB Function metric

Metric Type: Size

Metric ID: `mathworks.metrics.MatlabFunctionCount`

Calculates the number of MATLAB Function blocks inside a component.

Description

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`.

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.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Model file count

Metric Type: Size

Metric ID: `mathworks.metrics.ModelFileCount`

Calculate the number of model files.

Description

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`.

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.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Parameter metric

Metric Type: Size

Metric ID: `mathworks.metrics.ParameterCount`

Calculate number of instances of parameter data

Description

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`.

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.

Capabilities and Limitations

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.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Stateflow chart metric

Metric Type: Size

Metric ID: `mathworks.metrics.StateflowChartCount`

Calculate the number of Stateflow charts at any component level.

Description

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`.

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.

Capabilities and Limitations

- Runs on library models.
- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Cyclomatic complexity metric

Metric Type: Architecture

Metric ID: `mathworks.metrics.CyclomaticComplexity`

Model Advisor Check ID: `mathworks.metricchecks.CyclomaticComplexity`

Display the cyclomatic complexity of the model.

Description

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:

$$c = \sum_{1}^{N} (o_n - 1)$$

N is the number of decision points that the object represents and o_n is the number of outcomes for the nth 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`.

Model Advisor Check

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.

Capabilities and Limitations

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.

See Also

- “Compare Model Complexity and Code Complexity Metrics”
- “Collect Model and Testing Metrics”
- “Cyclomatic Complexity for Stateflow Charts” (Simulink Coverage)
- “Specify Coverage Options” (Simulink Coverage)

Clone content metric

Metric Type: Architecture

Metric ID: `mathworks.metrics.CloneContent`

Calculates the fraction of total number of subcomponents that are clones.

Description

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”.

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`.

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.

Capabilities and Limitations

- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Clone detection metric

Metric Type: Architecture

Metric ID: `mathworks.metrics.CloneDetection`

Calculate the number of clones in a model.

Description

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`.

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.

Capabilities and Limitations

- Analyzes content in masked subsystems.
- If specified, analyzes the content of library-linked blocks or referenced models.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Library content metric

Metric Type: Architecture

Metric ID: `mathworks.metrics.LibraryContent`

Calculates the fraction of total number of components that are linked library blocks.

Description

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`.

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

Capabilities and Limitations

- If specified, analyzes the content of library-linked blocks or referenced models.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”.

Nondescriptive block name metric

Metric Type: Readability

Metric ID: `mathworks.metrics.DescriptiveBlockNames`

Model Advisor Check ID: `mathworks.metricchecks.DescriptiveBlockNames`

Display nondescriptive Inport, Outport, and Subsystem block names.

Description

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`.

Model Advisor Check

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

Capabilities and Limitations

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.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”

Data and structure layer separation metric

Metric Type: Readability

Metric ID: `mathworks.metrics.LayerSeparation`

Model Advisor Check ID: `mathworks.metricchecks.LayerSeparation`

Display data and structure layer separation.

Description

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`.

Model Advisor Check

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.

Capabilities and Limitations

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.

See Also

For more information on model metrics, see “Collect Model and Testing Metrics”

MATLAB code analyzer warnings

Metric Type: Compliance

Metric ID: `mathworks.metrics.MatlabCodeAnalyzerWarnings`

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`.

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.

Capabilities and Limitations

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

See Also

- “Collect Model and Testing Metrics”
- “Check Code for Errors and Warnings Using the Code Analyzer”

Model Advisor Check Compliance for High-Integrity Systems

Metric Type: Compliance

Family ID: `mathworks.metrics.ModelAdvisorCheckCompliance`

Metric ID: `mathworks.metrics.ModelAdvisorCheckCompliance.hisl_do178`

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`.

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.

Capabilities and Limitations

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.

See Also

- “Collect Model and Testing Metrics”
- “Model Checks for DO-178C/DO-331 Standard Compliance”

Model Advisor Check Compliance for Modeling Standards for MAB

Metric Type: Compliance

Family ID: `mathworks.metrics.ModelAdvisorCheckCompliance`

Metric ID: `mathworks.metrics.ModelAdvisorCheckCompliance.maab`

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`.

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.

Capabilities and Limitations

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.

See Also

- “Collect Model and Testing Metrics”
- “Model Advisor Checks for MAB and JMAAB Compliance” on page 2-109

Model Advisor Check Issues for High-Integrity Systems

Metric Type: Compliance

Family ID: `mathworks.metrics.ModelAdvisorCheckIssues`

Metric ID: `mathworks.metrics.ModelAdvisorCheckIssues.hisl_do178`

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`.

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.

Capabilities and Limitations

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.

See Also

- “Collect Model and Testing Metrics”
- “Model Checks for DO-178C/DO-331 Standard Compliance”

Model Advisor Check Issues for MAB Standards

Metric Type: Compliance

Family ID: `mathworks.metrics.ModelAdvisorCheckIssues`

Metric ID: `mathworks.metrics.ModelAdvisorCheckIssues.maab`

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`.

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.

Capabilities and Limitations

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.

See Also

- “Collect Model and Testing Metrics”
- “Model Advisor Checks for MAB and JMAAB Compliance” on page 2-109

Compliance Metrics for Model Advisor Configurations

The Metrics Dashboard and metric APIs can obtain compliance and issues metric data for your Model Advisor configuration or for an existing check group, such as the MISRA checks. To set up your own Model Advisor configuration, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”.

You can configure Model Advisor compliance metrics and Model Advisor compliance issues metrics.

Model Advisor Compliance Metrics

Metric Type: Compliance

Family ID: `mathworks.metrics.ModelAdvisorCheckCompliance`

Metric ID: **<Family ID>.<**Model Advisor **Check Group ID>**

Use this metric to calculate the fraction of Model Advisor checks that pass for the selected group of checks.

For Model Advisor compliance metrics, the **Family ID** is `mathworks.metrics.ModelAdvisorCheckCompliance`.

To obtain the Model Advisor **Check Group ID**, 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 information on the Model Advisor Configuration Editor, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”.

For an example of how to use Model Advisor compliance metrics, see “Customize Metrics Dashboard Layout and Functionality”.

Model Advisor Compliance Issues Metrics

Metric Type: Compliance

Family ID: `mathworks.metrics.ModelAdvisorCheckIssues`

Metric ID: **<Family ID>.<**Model Advisor **Check Group ID>**

Use this metric to calculate the number of issues reported by the selected group of Model Advisor checks. This metric counts each Model Advisor check, in the selected group of Model Advisor checks, that produces a warning or failure.

For Model Advisor compliance issues metrics, the **Family ID** is `mathworks.metrics.ModelAdvisorCheckIssues`.

To obtain the Model Advisor **Check Group ID**, 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 information on the Model Advisor Configuration Editor, see “Use the Model Advisor Configuration Editor to Customize the Model Advisor”.

For an example of how to use Model Advisor compliance metrics, see “Customize Metrics Dashboard Layout and Functionality”.

See Also

Related Examples

- “Collect Model Metrics Using the Model Advisor”
- “Collect Model Metrics Programmatically”
- “Model Metric Data Aggregation”
- “Create a Custom Model Metric for Nonvirtual Block Count”
- “Customize Metrics Dashboard Layout and Functionality”

Model Transformer Tasks

Model Transformer Tasks

In this section...

Use the Model Transformer app to refactor a model to implement variants, eliminate eligible data store blocks, and improve the simulation and code efficiency of table lookup operations. You can perform the steps in the Model Transformer all at once or one step at a time.

Transformations

You can use the Model Transformer app to replace:

- Qualifying modeling patterns with variant blocks.
- Data store blocks with blocks that make data dependency explicit
- Lookup Table blocks with Interpolation using Prelookup blocks
- Modeling patterns with multiple Interpolation using Prelookup blocks into a single Interpolation using Prelookup block

If you want to perform all the transformations, for each step, specify the input parameters. Then, click the **Run Selected Checks** button. After you run each check, create new models with the transformations by clicking the **Refactor Model** buttons.

If you want to perform one transformation at a time, you can individually select the checks.

Replace Modeling Patterns with Variant Blocks

Select the **Transform model to variant system** button to identify system constants to use in variant transformations and blocks that qualify for transformation into Variant Subsystem or Variant Source blocks. These transformations are possible:

- If an If block connects to one or more If Action Subsystem blocks and each If Action Subsystem block has one output port, replace this modeling pattern with a subsystem and a Variant Source block.
- If an If block connects to an If Action Subsystem block that has no output port or two or more output ports, replace this modeling pattern with a Variant Subsystem block.
- If a Switch Case block connects to one or more Switch Case Action Subsystem blocks and each Switch Case Action Subsystem block has one output port, replace this modeling pattern with a subsystem block and a Variant Source block.
- If a Switch Case block connects to a Switch Case Action Subsystem block that has no output port or two or more output ports, 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.

Note For some modelling patterns and settings, the Model Transformer cannot perform each one of the preceding transformations.

A system constant is a control input or part of an arithmetic expression that forms the control input to Multiport Switch or Switch blocks and the inputs to If or Switch Case blocks. You must use Constant blocks and a combination of blocks that form a supported MATLAB expression as the control input of these blocks. In the Constant block, the **Constant value** parameter is the system constants. In the transformed model, system constants are part of condition expressions in Variant Source or Variant Subsystem blocks.

After you run this transformation, the **Result** table displays each modeling pattern with a hyperlink to the corresponding location in the model. If you do not want the Model Transformer to perform a transformation, clear the check box next to the pattern.

Click **Refactor Model** to create a model that contains the transformations. The app creates the transformed model in a folder with the name `m2m_<original model name>`.

For an example that converts blocks in a model to variant block, see “Transform Model to Variant System”.

Eliminate Data Store Blocks

Select **Eliminate data store blocks** to identify Data Store Memory, Data Store Read, and Data Store Write blocks that qualify for elimination. Click the **Refactor Model** button to create a model that replaces these blocks with either a direct signal line, Delay block, or Merge block.

Replacing these blocks improves model readability by making the data dependency explicit. 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 app 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 app replaces these blocks with a direct connection.
- For bus signals, if the write to bus elements executes before the read of the bus, the app 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 app replaces the Data Store Read and Data Store Write blocks with a direct connection and a Bus Selector block.
- For conditionally executed subsystems, the app replaces the Data Store Read and Data Store Write blocks with a direct connection and a Merge block.

The Model Transformer app only eliminates local data stores that Data Store Memory blocks define. The app does not eliminate global data stores. For the Data Store Memory block, on the **Signal Attributes** tab of the Block Parameters dialog box, the **Data store name must resolve to Simulink signal object** parameter must be cleared.

After you run this transformation, the **Result** table displays hyperlinks to the corresponding Data Store Memory, Data Store Read, and Data Store Write blocks. If you do not want the Model Transformer to perform a transformation, clear the check box next to the Data Store Memory block.

Click **Refactor Model** to create a model that contains the transformations. The app creates the transformed model in a folder with the name `m2m_<original model name>`.

For an example that replaces Data Store blocks in a model, see “Replace Data Store Blocks”.

Transform Table Lookup Blocks to Prelookup and Interpolation Using Prelookup Blocks

Select **Transform table lookup into prelookup and interpolation** to identify n-D Lookup Table blocks to transform into shared Prelookup blocks and Interpolation Using Prelookup blocks. Eliminating redundant Prelookup blocks in a model improves the simulation speed for linear interpolation.

The Model Transformer identifies where multiple n-D Lookup Table blocks:

- Use the same input signal for the Lookup Table blocks
- Have the same breakpoint specification, values, and data types
- Have the same algorithm parameters in the block parameters dialog box
- Have the same data type for fraction parameters

After you run this transformation, the **Result** table displays the identified blocks, their algorithm parameters, and the model, libraries, and referenced models. Select the blocks you want to transform, then click **Run This Check**.

For an example about the transformation, see “Improve Efficiency of Simulation by Optimizing Prelookup Operation of Lookup Table Blocks”. For more information about n-D Lookup Table blocks, see n-D Lookup Table.

Replace Interpolation Using Prelookup Blocks

Select **Common source interpolation transform** to identify multiple Interpolation Using Prelookup blocks that qualify for transformation with a single Interpolation Using Prelookup block. Eliminating the redundant Interpolation Using Prelookup blocks improves the code efficiency of the model.

The Model Transformer identifies when a model contains 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 switch block
- Have the same breakpoint specification, values, and data types
- Have the same algorithm parameters
- Have the same data type for fraction parameters

The Model Transformer app works if the properties of Interpolation Using Prelookup blocks are same except for **Table data**.

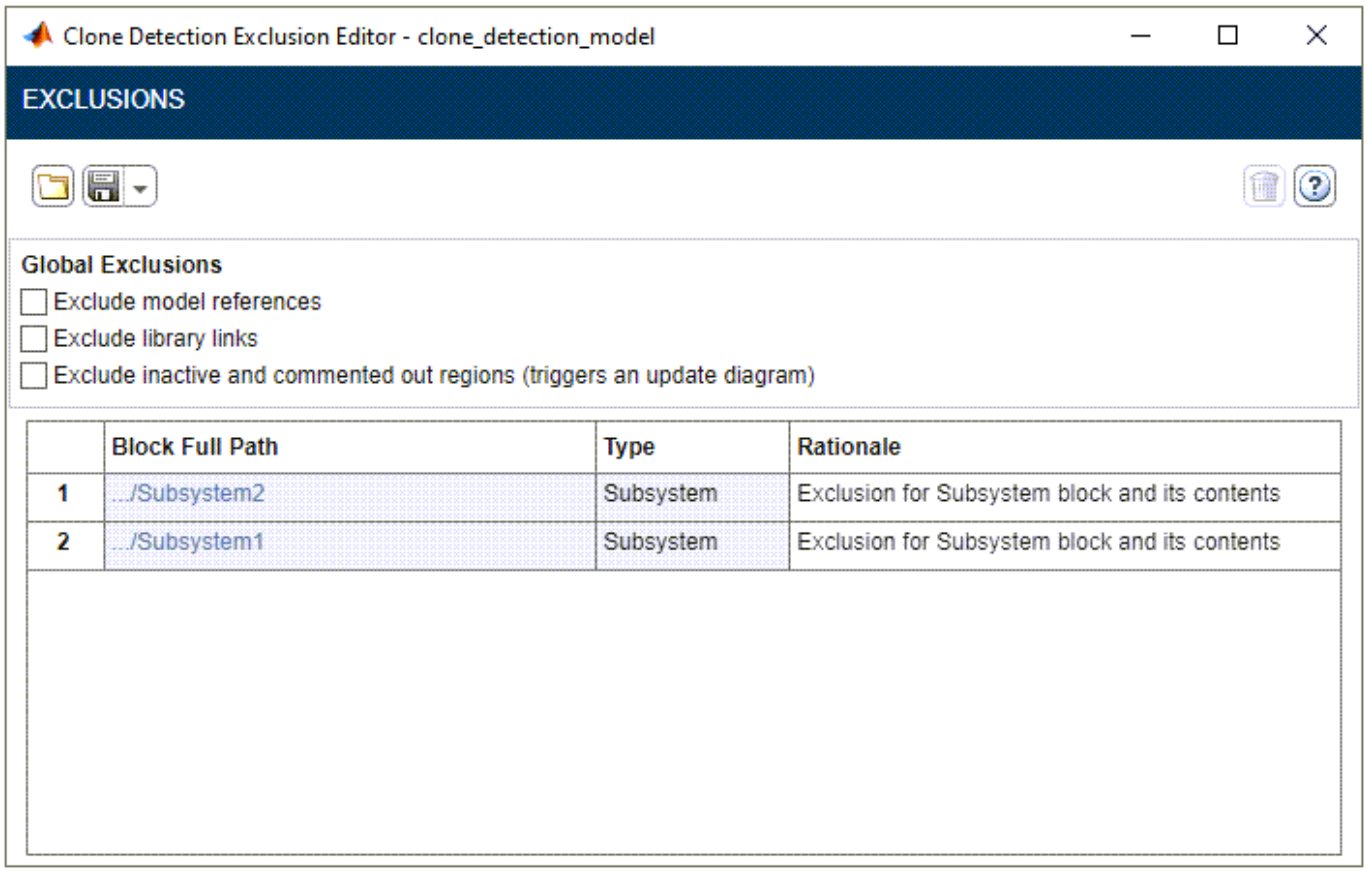
After you run this transformation, the **Result** table displays the identified blocks, their algorithm parameters, and the model, libraries, and referenced models. Select the blocks you want to transform, then click **Run This Check**.

For an example about the transformation, see “Improve Code Efficiency by Merging Multiple Interpolation Using Prelookup Blocks”. For more information about Interpolation Using Prelookup blocks, see “Using the Prelookup and Interpolation Blocks”.

Clone Detection Tasks

Exclude Components from Clone Detection

To save time during model development, you can limit the scope of the clone detection analysis to exclude Subsystem and Model blocks from clone detection. To exclude a subsystem or referenced model, right-click the subsystem or referenced model and select **Identify Modeling Clones > Subsystem and its contents > Add to exclusions**.



After you specify the Subsystem or Model blocks to exclude, the Clone Detector App uses the exclusion information to exclude blocks during analysis. By default, the exclusion information is stored in the model SLX file. Alternatively, you can store the information in an separate exclusion file. Unless you specify a different folder, the Clone Detection Exclusion Editor saves the 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 analysis excludes the blocks specified in the exclusion file.

To view exclusion information for a model, right-click a Subsystem or Model block and select **Identify Modeling Clones > Open Clone Detection Exclusion Editor**. For each subsystem or referenced model that you exclude from detection, you can use the **Rationale** field to provide a reason for excluding the block.

You can also select these properties in the Clone Detection Exclusion Editor:

- **Exclude model references** - Exclude model references.
- **Exclude library links** - Exclude subsystems that are external library links.
- **Exclude inactive and commented out regions** - Exclude modeling subsystems that are inactive during simulation, such as variant subsystems.

See Also

Related Examples

- “Enable Component Reuse by Using Clone Detection”
- “Replace Exact Clones with Subsystem Reference”

5

Model Testing Metrics

Model Testing Metrics

The Model Testing Dashboard collects metric data from the model design and testing artifacts in a project, such as requirements, models, and test results. Use the metric data to assess the status and quality of your model testing. 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-178. Use the widgets in the Model Testing Dashboard to see high-level metric results and testing gaps, as described in “Explore Status and Quality of Testing Activities Using Model Testing Dashboard”.

Alternatively, you can use the API functions to collect metric results programmatically. When using the API, use the metric IDs to refer to each metric. See “Collect Metrics on Model Testing Artifacts Programmatically” for an example of how to collect these metrics programmatically. You can use the function `getAvailableMetricIds` to return a list of available metric identifiers.

The model testing metrics provide:

Metrics for Requirements Linked to Tests

The metrics associated with the **Requirements Linked to Tests** section of the dashboard include:

Metric	Description
“RequirementWithTestCase” on page 5-7	Determine whether a requirement is linked to test cases.
“RequirementWithTestCasePercentage” on page 5-7	Calculate the percentage of requirements that are linked to test cases.
“RequirementWithTestCaseDistribution” on page 5-8	Distribution of the number of requirements linked to test cases compared to the number of requirements that are missing test cases.
“TestCasesPerRequirement” on page 5-10	Count the number of test cases linked to each requirement.
“TestCasesPerRequirementDistribution” on page 5-10	Distribution of the number of test cases linked to each requirement.

For more information, see “Requirements Linked to Tests” on page 5-6.

Metrics for Tests Linked to Requirements

The metrics associated with the **Tests Linked to Requirements** section of the dashboard include:

Metric	Description
“TestCaseWithRequirement” on page 5-12	Determine whether a test case is linked to requirements.
“TestCaseWithRequirementPercentage” on page 5-13	Calculate the fraction of test cases that are linked to requirements.
“TestCaseWithRequirementDistribution” on page 5-14	Distribution of the number of test cases linked to requirements compared to the number of test cases that are missing links to requirements.
“RequirementsPerTestCase” on page 5-16	Count the number of requirements linked to each test case.
“RequirementsPerTestCaseDistribution” on page 5-17	Distribution of the number of requirements linked to each test case.

For more information, see “Tests Linked to Requirements” on page 5-11.

Metrics for Test Case Breakdown

The metrics associated with the **Test Case Breakdown** section of the dashboard include:

Metric	Description
“TestCaseType” on page 5-18	Return the type of the test case.
“TestCaseTypeDistribution” on page 5-19	Distribution of the types of the test cases for the unit.
“TestCaseTag” on page 5-20	Return the tags for a test case.
“TestCaseTagDistribution” on page 5-21	Distribution of the tags of the test cases for the unit.

For more information, see “Test Case Breakdown” on page 5-18.

Metrics for Model Test Status

The metrics associated with the **Model Test Status** section of the dashboard include:

Metric	Description
“TestCaseStatus” on page 5-22	Return the status of the test case result.
“TestCaseStatusPercentage” on page 5-23	Calculate the fraction of test cases that passed.
“TestCaseStatusDistribution” on page 5-24	Distribution of the statuses of the test case results for the unit.
“TestCaseVerificationStatus” on page 5-26	Determine whether a test case has pass/fail criteria such as verify statements, verification blocks, custom criteria, and logical or temporal assessments.

Metric	Description
“TestCaseVerificationStatusDistribution” on page 5-27	Distribution of the number of test cases that do not have pass/fail criteria compared to the number of test cases that do have pass/fail criteria.

For more information, see “Model Test Status” on page 5-22.

Metrics for Model Coverage for the Unit

The **Model Coverage** section of the dashboard shows the aggregated coverage for the unit.

The metrics associated with the **Model Coverage** section of the dashboard include:

Metric	Description
“ExecutionCoverageBreakdown” on page 5-28	Overall model execution coverage achieved, justified, or missed by the tests in the unit.
“DecisionCoverageBreakdown” on page 5-29	Overall model decision coverage achieved, justified, or missed by the tests in the unit.
“ConditionCoverageBreakdown” on page 5-30	Overall model condition coverage achieved, justified, or missed by the tests in the unit.
“MCDCCoverageBreakdown” on page 5-31	Overall model modified condition and decision coverage (MC/DC) achieved, justified, or missed by the tests in the unit.

For more information, see “Model Coverage for the Unit” on page 5-28.

Metrics for Model Coverage for Each Model in the Unit

When you click on a bar in the **Model Coverage** bar chart, the **Metric Details** show the coverage for each model in the unit.

The metrics associated with the model coverage for each model in the unit include:

Metric	Description
“ExecutionCoverageFragment” on page 5-33	Execution coverage for each model in the unit.
“DecisionCoverageFragment” on page 5-33	Decision coverage for each model in the unit.
“ConditionCoverageFragment” on page 5-34	Condition coverage for each model in the unit.
“MCDCCoverageFragment” on page 5-35	Modified condition/decision coverage (MC/DC) for each model in the unit.

For more information, see “Model Coverage for each Model in the Unit” on page 5-32.

Metrics for Requirements-Based Tests for the Unit

The **Achieved Coverage Ratio** section of the dashboard shows the sources of achieved coverage for the unit. The **Requirements-Based Tests** section shows how much of the overall achieved coverage comes from requirements-based tests.

The metrics associated with the **Requirements-Based Tests** section of the dashboard include:

Metric	Description
"RequirementsExecutionCoverageBreakdown" on page 5-37	Fraction of the overall achieved execution coverage that comes from requirements-based tests in the unit.
"RequirementsDecisionCoverageBreakdown" on page 5-38	Fraction of the overall achieved decision coverage that comes from requirements-based tests in the unit.
"RequirementsConditionCoverageBreakdown" on page 5-39	Fraction of the overall achieved condition coverage that comes from requirements-based tests in the unit.
"RequirementsMCDCCoverageBreakdown" on page 5-40	Fraction of the overall achieved MC/DC coverage that comes from requirements-based tests in the unit.

For more information, see "Requirements-Based Tests for the Unit" on page 5-36.

Metrics for Requirements-Based Tests for Each Model in the Unit

When you click on a bar in the **Requirements-Based Tests** section, the **Metric Details** show the coverage ratio for each model in the unit. For requirements-based tests, the coverage ratio is the percentage of the overall achieved coverage that comes from requirements-based tests.

The metrics associated with requirements-based coverage for each model in the unit include:

Metric	Description
"RequirementsExecutionCoverageFragment" on page 5-41	Fraction of the overall achieved execution coverage that comes from requirements-based tests in each model in the unit.
"RequirementsDecisionCoverageFragment" on page 5-42	Fraction of the overall achieved decision coverage that comes from requirements-based tests in each model in the unit.
"RequirementsConditionCoverageFragment" on page 5-43	Fraction of the overall achieved condition coverage that comes from requirements-based tests in each model in the unit.
"RequirementsMCDCCoverageFragment" on page 5-44	Fraction of the overall achieved MC/DC coverage that comes from requirements-based tests in each model in the unit.

For more information, see "Requirements-Based Tests for each Model in the Unit" on page 5-41.

Metrics for Unit-Boundary Tests for the Unit

The **Achieved Coverage Ratio** section of the dashboard shows the sources of achieved coverage for the unit. The **Unit-Boundary Tests** section shows how much of the overall achieved coverage comes from unit-boundary tests.

The metrics associated with the **Unit-Boundary Tests** section of the dashboard include:

Metric	Description
"UnitBoundaryExecutionCoverageBreakdown" on page 5-45	Fraction of the overall achieved execution coverage that comes from unit-boundary tests in the unit.
"UnitBoundaryDecisionCoverageBreakdown" on page 5-45	Fraction of the overall achieved decision coverage that comes from unit-boundary tests in the unit.
"UnitBoundaryConditionCoverageBreakdown" on page 5-46	Fraction of the overall achieved condition coverage that comes from unit-boundary tests in the unit.
"UnitBoundaryMCDCCoverageBreakdown" on page 5-47	Fraction of the overall achieved MC/DC coverage that comes from unit-boundary tests in the unit.

For more information, see "Unit-Boundary Tests for the Unit" on page 5-44.

Metrics for Unit-Boundary Tests for Each Model in the Unit

When you click on a bar in the **Unit-Boundary Tests** section, the **Metric Details** show the coverage ratio for each model in the unit. For unit-boundary tests, the coverage ratio is the percentage of the overall achieved coverage that comes from unit-boundary tests.

The metrics associated with unit-boundary coverage for each model in the unit include:

Metric	Description
"UnitBoundaryExecutionCoverageFragment" on page 5-49	Fraction of the overall achieved execution coverage that comes from unit-boundary tests in each model in the unit.
"UnitBoundaryDecisionCoverageFragment" on page 5-49	Fraction of the overall achieved decision coverage that comes from unit-boundary tests in each model in the unit.
"UnitBoundaryConditionCoverageFragment" on page 5-50	Fraction of the overall achieved condition coverage that comes from unit-boundary tests in each model in the unit.
"UnitBoundaryMCDCCoverageFragment" on page 5-51	Fraction of the overall achieved MC/DC coverage that comes from unit-boundary tests in each model in the unit.

For more information, see "Unit-Boundary Tests for each Model in the Unit" on page 5-48.

Requirements Linked to Tests

The metrics associated with the **Requirements Linked to Tests** section of the dashboard are:

RequirementWithTestCase

Determine whether a requirement is linked to test cases.

Metric Information
Metric ID: `RequirementWithTestCase`
Description: Use this metric to determine whether a requirement is linked to a test case 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`. 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 test cases in the project. • `1` — The requirement is linked to at least one test case with a link where the **Type** is set to `Verifies`.
Capabilities and Limitations: 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 test cases in the project where the link type is set to `Verifies`, including links to test cases that test other models or subsystems. For each requirement that is linked to test cases, check that the links are to test cases that run on the unit that implements the requirement.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

RequirementWithTestCasePercentage

Calculate the percentage of requirements that are linked to test cases.

Metric Information
Metric ID: `RequirementWithTestCasePercentage`
Description: This metric counts the fraction of requirements that are linked to at least one test case 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 a unit with a link where the **Type** is set to `Implements`. This metric calculates the results by using the results of the `RequirementWithTestCase` metric. To collect data for this metric: • In the Model Testing Dashboard, view the **Requirements with Tests** widget. • Use `getMetrics` with the metric ID `RequirementWithTestCasePercentage`. 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 fraction structure that contains these fields: • `Numerator` — The number of implemented requirements that are linked to at least one test case. • `Denominator` — The total number of functional requirements implemented in the unit with a link where the **Type** is set to `Implements`.
Compliance Thresholds: The default compliance thresholds for this metric are: • `Compliant` — `100%` of unit requirements are linked to test cases • `Non-Compliant` — Less than `100%` of unit requirements are linked to test cases • `Warning` — None
Capabilities and Limitations: 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 test cases in the project where the link type is set to `Verifies`, including links to test cases that test other models or subsystems. For each requirement that is linked to test cases, check that the links are to test cases that run on the unit that implements the requirement.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

RequirementWithTestCaseDistribution

Distribution of the number of requirements linked to test cases compared to the number of requirements that are missing test cases.

Metric Information
Metric ID: `RequirementWithTestCaseDistribution`
Description: Use this metric to count the number of requirements that are linked to test cases and the number of requirements that are missing links to test cases. 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 case 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. 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. • `BinEdges` — The logical output results of the `RequirementWithTestCase` metric, returned as a vector with entries `0` (`false`) and `1` (`true`). The first bin includes requirements that are not linked to test cases. The second bin includes requirements that are linked to at least one test case.
Compliance Thresholds: The default compliance thresholds for this metric are: • `Compliant` — `0` requirements are missing links to test cases • `Non-Compliant` — `1` or more requirements are missing links to test cases • `Warning` — None
Capabilities and Limitations: 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 test cases in the project where the link type is set to `Verifies`, including links to test cases that test other models or subsystems. For each requirement that is linked to test cases, check that the links are to test cases that run on the unit that implements the requirement.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

TestCasesPerRequirement

Count the number of test cases linked to each requirement.

Metric Information
Metric ID: `TestCasesPerRequirement`
Description: Use this metric to count the number of test cases 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 case is linked to a requirement if it has a link where the **Type** is set to `Verifies`. 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.
Capabilities and Limitations: 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 test cases in the project where the link type is set to `Verifies`, including links to test cases that test other models or subsystems. For each requirement that is linked to test cases, check that the links are to test cases that run on the unit that implements the requirement.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

TestCasesPerRequirementDistribution

Distribution of the number of test cases linked to each requirement.

Metric Information
Metric ID: `TestCasesPerRequirementDistribution`

Metric Information
Description: This metric returns a distribution of the number of test cases linked to each requirement. Use this metric to determine if requirements are linked to a disproportionate number of test cases. 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 case 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 `Test cases per requirement` metric. 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 test cases 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.
Capabilities and Limitations: 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 test cases in the project where the link type is set to `Verifies`, including links to test cases that test other models or subsystems. For each requirement that is linked to test cases, check that the links are to test cases that run on the unit that implements the requirement.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

Tests Linked to Requirements

The metrics associated with the **Tests Linked to Requirements** section of the dashboard are:

TestCaseWithRequirement

Determine whether a test case is linked to requirements.

Metric Information
Metric ID: `TestCaseWithRequirement`
Description: Use this metric to determine whether a test case is linked to a requirement with a link where the **Type** is set to `Verifies`. The metric analyzes only test cases that run on the model or subsystems in the unit for which you collect metric data. 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 case is not linked to requirements that are implemented in the unit. • `1` — The test case is linked to at least one requirement with a link where the **Type** is set to `Verifies`.

Metric Information

Capabilities and Limitations:
 	
The metric:
 	
• Analyzes only test cases 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 case that is linked to requirements, check that the links are to requirements that are implemented by the unit that the test case runs on.

See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

TestCaseWithRequirementPercentage

Calculate the fraction of test cases that are linked to requirements.

Metric Information
Metric ID: `TestCaseWithRequirementPercentage`
Description: This metric counts the fraction of test cases that are linked to at least one requirement with a link where the **Type** is set to `Verifies`. The metric analyzes only test cases that run on the model or subsystems in the unit for which you collect metric data. This metric calculates the results by using the results of the `TestCaseWithRequirement` metric. To collect data for this metric: • In the Model Testing Dashboard, view the **Tests with Requirements** widget. • Use `getMetrics` with the metric ID `TestCaseWithRequirementPercentage`. 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 fraction structure that contains these fields: • `Numerator` — The number of test cases that are linked to at least one requirement with a link where the **Type** is set to `Verifies`. • `Denominator` — The total number of test cases that test the unit.

Metric Information
Compliance Thresholds: The default compliance thresholds for this metric are: • `Compliant` — `100%` of test cases are linked to requirements • `Non-Compliant` — Less than `100%` of test cases are linked to requirements • `Warning` — None

Capabilities and Limitations:
 	
The metric:
 	
• Analyzes only test cases 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 case that is linked to requirements, check that the links are to requirements that are implemented by the unit that the test case runs on.

See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

TestCaseWithRequirementDistribution

Distribution of the number of test cases linked to requirements compared to the number of test cases that are missing links to requirements.

Metric Information
Metric ID: `TestCaseWithRequirementDistribution`

Metric Information
Description: Use this metric to count the number of test cases that are linked to requirements and the number of test cases that are missing links to requirements. The metric analyzes only test cases that run on the model or subsystems in the unit for which you collect metric data. A test case 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. 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 test cases in each bin, returned as an integer vector. • `BinEdges` — The logical output results of the `TestCaseWithRequirement` metric, returned as a vector with entries `0` (`false`) and `1` (`true`). The first bin includes test cases that are not linked to requirements. The second bin includes test cases that are linked to at least one requirement.
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

Metric Information

Capabilities and Limitations:
 	
The metric:
 	
• Analyzes only test cases 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 case that is linked to requirements, check that the links are to requirements that are implemented by the unit that the test case runs on.

See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

RequirementsPerTestCase

Count the number of requirements linked to each test case.

Metric Information
Metric ID: `RequirementsPerTestCase`
Description: Use this metric to count the number of requirements linked to each test case. The metric analyzes only test cases that run on the model or subsystems in the unit for which you collect metric data. A test case is linked to a requirement if it has a link where the **Type** is set to `Verifies`. 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.

Metric Information

Capabilities and Limitations:
 	
The metric:
 	
• Analyzes only test cases 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 case that is linked to requirements, check that the links are to requirements that are implemented by the unit that the test case runs on.

See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

RequirementsPerTestCaseDistribution

Distribution of the number of requirements linked to each test case.

Metric Information
Metric ID: `RequirementsPerTestCaseDistribution`
Description: This metric returns a distribution of the number of requirements linked to each test case. Use this metric to determine if test cases are linked to a disproportionate number of requirements. The metric analyzes only test cases that run on the model or subsystems in the unit for which you collect metric data. A test case 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 case` metric. 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.

Metric Information
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` — Bin edges for the number of requirements linked to each test case, 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.

Capabilities and Limitations:
 	
The metric:
 	
• Analyzes only test cases 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 case that is linked to requirements, check that the links are to requirements that are implemented by the unit that the test case runs on.

See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

Test Case Breakdown

The metrics associated with the **Test Case Breakdown** section of the dashboard are:

TestCaseType

Return the type of the test case.

Metric Information
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. 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
Capabilities and Limitations: The metric includes only test cases in the project that test the model or subsystems in the unit for which you collect metric data.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

TestCaseTypeDistribution

Distribution of the types of the test cases for the unit.

Metric Information
Metric ID: `TestCaseTypeDistribution`

Metric Information
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. 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 toolstrip.
Capabilities and Limitations: The metric includes only test cases in the project that test the model or subsystems in the unit for which you collect metric data.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

TestCaseTag

Return the tags for a test case.

Metric Information
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. 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.
Capabilities and Limitations: The metric includes only test cases in the project that test the model or subsystems in the unit for which you collect metric data.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

TestCaseTagDistribution

Distribution of the tags of the test cases for the unit.

Metric Information
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. 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.

Metric Information
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.
Capabilities and Limitations: The metric includes only test cases in the project that test the model or subsystems in the unit for which you collect metric data.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

Model Test Status

The metrics associated with the **Model Test Status** section of the dashboard are:

- “TestCaseStatus” on page 5-22
- “TestCaseStatusPercentage” on page 5-23
- “TestCaseStatusDistribution” on page 5-24
- “TestCaseVerificationStatus” on page 5-26
- “TestCaseVerificationStatusDistribution” on page 5-27

TestCaseStatus

Return the status of the test case result.

Metric Information
Metric ID: `TestCaseStatus`

Metric Information
Description: This metric returns the status of the test case result. A test status is passed, failed, disabled, or untested. To collect data for this metric: • In the Model Testing Dashboard, click a widget in the **Model Test Status** section to display the results in a table. • Use `getMetrics` with the metric ID `TestCaseStatus`. 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 case failed. • `1` — The test case passed. • `2` — The test case was disabled. • `3` — The test case was not run (untested).
Capabilities and Limitations: The metric: • Includes only test cases in the project that test the model or subsystems in the unit for which you collect metric data. • Does not count the status of test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. The metric shows these test cases as untested. • Reflects the status of the whole test case if the test case includes multiple iterations.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

TestCaseStatusPercentage

Calculate the fraction of test cases that passed.

Metric Information
Metric ID: `TestCaseStatusPercentage`

Metric Information
Description: This metric counts the fraction of test cases that passed in the test results. This metric calculates the results by using the results of the `TestCaseStatus` metric. To collect data for this metric: • In the Model Testing Dashboard, in the **Model Test Status** section, place your cursor over the **Passed** widget. • Use `getMetrics` with the metric ID `TestCaseStatusPercentage`. 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 fraction structure that contains these fields: • `Numerator` — The number of test cases that passed. • `Denominator` — The total number of test cases that test the unit.
Compliance Thresholds: The default compliance thresholds for this metric are: • `Compliant` — `100%` of test cases passed • `Non-Compliant` — Less than `100%` of test cases passed • `Warning` — None
Capabilities and Limitations: The metric: • Includes only test cases in the project that test the model or subsystems in the unit for which you collect metric data. • Does not count the status of test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. The metric shows these test cases as untested. • Reflects the status of the whole test case if the test case includes multiple iterations.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

TestCaseStatusDistribution

Distribution of the statuses of the test case results for the unit.

Metric Information
Metric ID: `TestCaseStatusDistribution`

Metric Information
Description: This metric returns a distribution of the status of the results of test cases that run on the unit. A test status is passed, failed, disabled, or untested. This metric returns the result as a distribution of the results of the `Test case type` metric. To collect data for this metric: • In the Model Testing Dashboard, use the widgets in the **Model Test Status** section to see the results. • Use `getMetrics` with the metric ID `TestCaseStatusDistribution`. 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 `TestCaseStatus` metric, returned as an integer vector. The integer outputs represent the test result statuses:
 	
  • `0` — The test case failed.
  • `1` — The test case passed.
  • `2` — The test case was disabled.
  • `3` — The test case was not run (untested).

Compliance Thresholds:
 	
The default compliance thresholds for this metric are:
 	
• `Compliant`:
 	
  • `0` unit tests are untested.
  • `0` unit tests failed.
  • `0` unit tests are disabled.
• `Non-Compliant` — `1` or more unit tests are untested, disabled, or have failed.
• `Warning` — None

Capabilities and Limitations: The metric: • Includes only test cases in the project that test the model or subsystems in the unit for which you collect metric data. • Does not count the status of test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. The metric shows these test cases as untested. • Reflects the status of the whole test case if the test case includes multiple iterations.

Metric Information
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

TestCaseVerificationStatus

Determine whether a test case has pass/fail criteria such as verify statements, verification blocks, custom criteria, and logical or temporal assessments.

Metric Information
Metric ID: `TestCaseVerificationStatus`
Description: Use this metric to determine whether a test case has pass/fail criteria. A test case 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 case 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 case is missing pass/fail criteria. • `1` — The test case has pass/fail criteria. • `2` — The test case was not run.
Capabilities and Limitations: The metric: • Includes only test cases 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 test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. The metric shows these test cases as **Missing Pass/Fail Criteria**. • Reflects the status of the whole test case if the test case includes multiple iterations.

Metric Information
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

TestCaseVerificationStatusDistribution

Distribution of the number of test cases that do not have pass/fail criteria compared to the number of test cases that do have pass/fail criteria.

Metric Information
Metric ID: `TestCaseVerificationStatusDistribution`
Description: Use this metric to count the number of test cases that do not have pass/fail criteria and the number of test cases that do have pass/fail criteria. A test case 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 case This metric returns the result as a distribution of the results of the `TestCaseVerificationStatusDistribution` metric. 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 test cases 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 case verification statuses:
 	
  • `0` — The test case is missing pass/fail criteria.
  • `1` — The test case has pass/fail criteria.
  • `2` — The test case was not run.

Metric Information
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
Capabilities and Limitations: The metric: • Includes only test cases 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 test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. The metric shows these test cases as **Missing Pass/Fail Criteria**. • Reflects the status of the whole test case if the test case includes multiple iterations.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

Model Coverage for the Unit

The **Model Coverage** section of the dashboard shows the aggregated coverage for the unit.

The metrics associated with the **Model Coverage** section of the dashboard are:

- “ExecutionCoverageBreakdown” on page 5-28 for the execution coverage
- “DecisionCoverageBreakdown” on page 5-29 for the decision coverage
- “ConditionCoverageBreakdown” on page 5-30 for the condition coverage
- “MCDCCoverageBreakdown” on page 5-31 for the MC/DC coverage

ExecutionCoverageBreakdown

Overall model execution coverage achieved, justified, or missed by the tests in the unit.

Metric Information
Metric ID: `ExecutionCoverageBreakdown`

Metric Information
Description: This metric returns the model execution coverage measured in the test results, aggregated across the unit. The metric result includes the percentage of execution coverage achieved by the test cases, the percentage of coverage justified in coverage filters, and the percentage of execution coverage missed by the tests. To collect data for this metric: • In the Model Testing Dashboard, in the **Model Coverage** section, place your cursor over the bars in the **Execution** widget. • Use `getMetrics` with the metric ID `ExecutionCoverageBreakdown`. 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 double vector that contains these elements. • `Value(1)` — The percentage of execution coverage achieved by the tests. • `Value(2)` — The percentage of execution coverage justified by coverage filters. • `Value(3)` — The percentage of execution coverage missed by the tests.
Compliance Thresholds: The default compliance thresholds for this metric are: • `Compliant` — Test results return `0%` missed coverage • `Non-Compliant` — Test results return missing coverage • `Warning` — None
Capabilities and Limitations: The metric: • Returns aggregated coverage results. • Does not include coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. • Returns 100% coverage for models that do not have execution points.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

DecisionCoverageBreakdown

Overall model decision coverage achieved, justified, or missed by the tests in the unit.

Metric Information
Metric ID: `DecisionCoverageBreakdown`

Metric Information
Description: This metric returns the model decision coverage measured in the test results, aggregated across the unit. The metric result includes the percentage of decision coverage achieved by the test cases, the percentage of coverage justified in coverage filters, and the percentage of decision coverage missed by the tests. To collect data for this metric: • In the Model Testing Dashboard, in the **Model Coverage** section, place your cursor over the bars in the **Decision** widget. • Use `getMetrics` with the metric ID `DecisionCoverageBreakdown`. 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 double vector that contains these elements: • `Value(1)` — The percentage of decision coverage achieved by the tests. • `Value(2)` — The percentage of decision coverage justified by coverage filters. • `Value(3)` — The percentage of decision coverage missed by the tests.
Compliance Thresholds: The default compliance thresholds for this metric are: • `Compliant` — Test results return `0%` missed coverage • `Non-Compliant` — Test results return missing coverage • `Warning` — None
Capabilities and Limitations: The metric: • Returns aggregated coverage results. • Does not include coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. • Returns 100% coverage for models that do not have decision points.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

ConditionCoverageBreakdown

Overall model condition coverage achieved, justified, or missed by the tests in the unit.

Metric Information
Metric ID: `ConditionCoverageBreakdown`

Metric Information
Description: This metric returns the model condition coverage measured in the test results, aggregated across the unit. The metric result includes the percentage of condition coverage achieved by the test cases, the percentage of coverage justified in coverage filters, and the percentage of condition coverage missed by the tests. To collect data for this metric: • In the Model Testing Dashboard, in the **Model Coverage** section, place your cursor over the bars in the **Condition** widget. • Use `getMetrics` with the metric ID `ConditionCoverageBreakdown`. 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 double vector that contains these elements: • `Value(1)` — The percentage of condition coverage achieved by the tests. • `Value(2)` — The percentage of condition coverage justified by coverage filters. • `Value(3)` — The percentage of condition coverage missed by the tests.
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
Capabilities and Limitations: The metric: • Returns aggregated coverage results. • Does not include coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. • Returns 100% coverage for models that do not have condition points.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

MCDCCoverageBreakdown

Overall model modified condition and decision coverage (MC/DC) achieved, justified, or missed by the tests in the unit.

Metric Information
Metric ID: `MCDCCoverageBreakdown`
Description: This metric returns the modified condition and decision (MC/DC) measured in the test results, aggregated across the unit. The metric result includes the percentage of MC/DC coverage achieved by the test cases, the percentage of coverage justified in coverage filters, and the percentage of MC/DC coverage missed by the tests. To collect data for this metric: • In the Model Testing Dashboard, in the **Model Coverage** section, place your cursor over the bars in the **MC/DC** widget. • Use `getMetrics` with the metric ID `MCDCCoverageBreakdown`. 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 double vector that contains these elements: • `Value(1)` — The percentage of MC/DC coverage achieved by the tests. • `Value(2)` — The percentage of MC/DC coverage justified by coverage filters. • `Value(1)` — The percentage of MC/DC coverage missed by the tests.
Compliance Thresholds: The default compliance thresholds for this metric are: • `Compliant` — Test results return `0%` missed coverage • `Non-Compliant` — Test results return missing coverage • `Warning` — None
Capabilities and Limitations: The metric: • Returns aggregated coverage results. • Does not include coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. • Returns 100% coverage for models that do not have condition/decision points.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

Model Coverage for each Model in the Unit

When you click on a bar in the **Model Coverage** bar chart, the **Metric Details** show the coverage for each model in the unit.

The metrics associated with the model coverage for each model in the unit are:

- “ExecutionCoverageFragment” on page 5-33 for the execution coverage
- “DecisionCoverageFragment” on page 5-33 for the decision coverage
- “ConditionCoverageFragment” on page 5-34 for the condition coverage
- “MCDCCoverageFragment” on page 5-35 for the MC/DC coverage

ExecutionCoverageFragment

Execution coverage for each model in the unit.

Metric Information
Metric ID: `ExecutionCoverageFragment`
Description: This metric returns the model execution coverage measured in the test results for each model in the unit. The metric result includes the percentage of execution coverage achieved by the test cases, the percentage of coverage justified in coverage filters, and the percentage of execution coverage missed by the tests. To collect data for this metric: • In the Model Testing Dashboard, in the **Model Coverage** section, click one of the bars in the **Execution** widget. • Use `getMetrics` with the metric ID `ExecutionCoverageFragment`. 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 double vector that contains these elements. • `Value(1)` — The percentage of execution coverage achieved by the tests. • `Value(2)` — The percentage of execution coverage justified by coverage filters. • `Value(3)` — The percentage of execution coverage missed by the tests.
Compliance Thresholds: This metric does not have predefined thresholds.
Capabilities and Limitations: The metric: • Does not include coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. • Returns 100% coverage for models that do not have execution points.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

DecisionCoverageFragment

Decision coverage for each model in the unit.

Metric Information
Metric ID: `DecisionCoverageFragment`
Description: This metric returns the model decision coverage measured in the test results for each model in the unit. The metric result includes the percentage of decision coverage achieved by the test cases, the percentage of coverage justified in coverage filters, and the percentage of decision coverage missed by the tests. To collect data for this metric: • In the Model Testing Dashboard, in the **Model Coverage** section, click one of the bars in the **Decision** widget. • Use `getMetrics` with the metric ID `DecisionCoverageFragment`. 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 double vector that contains these elements: • `Value(1)` — The percentage of decision coverage achieved by the tests. • `Value(2)` — The percentage of decision coverage justified by coverage filters. • `Value(3)` — The percentage of decision coverage missed by the tests.
Compliance Thresholds: This metric does not have predefined thresholds.
Capabilities and Limitations: The metric: • Does not include coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. • Returns 100% coverage for models that do not have decision points.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

ConditionCoverageFragment

Condition coverage for each model in the unit.

Metric Information
Metric ID: `ConditionCoverageFragment`

Metric Information
Description: This metric returns the model condition coverage measured in the test results for each model in the unit. The metric result includes the percentage of condition coverage achieved by the test cases, the percentage of coverage justified in coverage filters, and the percentage of condition coverage missed by the tests. To collect data for this metric: • In the Model Testing Dashboard, in the **Model Coverage** section, click one of the bars in the **Condition** widget. • Use `getMetrics` with the metric ID `ConditionCoverageFragment`. 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 double vector that contains these elements: • `Value(1)` — The percentage of condition coverage achieved by the tests. • `Value(2)` — The percentage of condition coverage justified by coverage filters. • `Value(3)` — The percentage of condition coverage missed by the tests.
Compliance Thresholds: This metric does not have predefined thresholds.
Capabilities and Limitations: The metric: • Does not include coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. • Returns 100% coverage for models that do not have condition points.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

MCDCCoverageFragment

Modified condition/decision coverage (MC/DC) for each model in the unit.

Metric Information
Metric ID: `MCDCCoverageFragment`

Metric Information
Description: This metric returns the modified condition and decision (MC/DC) measured in the test results for each model in the unit. The metric result includes the percentage of MC/DC coverage achieved by the test cases, the percentage of coverage justified in coverage filters, and the percentage of MC/DC coverage missed by the tests. To collect data for this metric: • In the Model Testing Dashboard, in the **Model Coverage** section, click one of the bars in the **MC/DC** widget. • Use `getMetrics` with the metric ID `MCDCCoverageFragment`. 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 double vector that contains these elements: • `Value(1)` — The percentage of MC/DC coverage achieved by the tests. • `Value(2)` — The percentage of MC/DC coverage justified by coverage filters. • `Value(1)` — The percentage of MC/DC coverage missed by the tests.
Compliance Thresholds: This metric does not have predefined thresholds.
Capabilities and Limitations: The metric: • Does not include coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode. • Returns 100% coverage for models that do not have condition/decision points.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

Requirements-Based Tests for the Unit

The **Achieved Coverage Ratio** section of the dashboard shows the sources of achieved coverage for the unit. The **Requirements-Based Tests** section shows how much of the overall achieved coverage comes from requirements-based tests.

The metrics associated with the **Requirements-Based Tests** section of the dashboard are:

- “RequirementsExecutionCoverageBreakdown” on page 5-37 for the requirements-based execution coverage
- “RequirementsDecisionCoverageBreakdown” on page 5-38 for the requirements-based decision coverage

- “RequirementsConditionCoverageBreakdown” on page 5-39 for the requirements-based condition coverage
- “RequirementsMCDCCoverageBreakdown” on page 5-40 for the requirements-based MC/DC coverage

RequirementsExecutionCoverageBreakdown

Fraction of the overall achieved execution coverage that comes from requirements-based tests.

Metric Information
Metric ID: `RequirementsExecutionCoverageBreakdown`
Description: This metric returns the fraction of overall achieved execution coverage that comes from requirements-based tests. 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
Capabilities and Limitations: The metric: • Analyzes the overall aggregated coverage results. • Does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Metric Information
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

RequirementsDecisionCoverageBreakdown

Fraction of the overall achieved decision coverage that comes from requirements-based tests.

Metric Information
Metric ID: `RequirementsDecisionCoverageBreakdown`
Description: This metric returns the fraction of overall achieved decision coverage that comes from requirements-based tests. 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
Capabilities and Limitations: The metric: • Analyzes the overall aggregated coverage results. • Does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Metric Information
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

RequirementsConditionCoverageBreakdown

Fraction of the overall achieved condition coverage that comes from requirements-based tests.

Metric Information
Metric ID: `RequirementsConditionCoverageBreakdown`
Description: This metric returns the fraction of overall achieved condition coverage that comes from requirements-based tests. 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
Capabilities and Limitations: The metric: • Analyzes the overall aggregated coverage results. • Does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Metric Information
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

RequirementsMCDCCoverageBreakdown

Fraction of the overall achieved MC/DC coverage that comes from requirements-based tests.

Metric Information
Metric ID: `RequirementsMCDCCoverageBreakdown`
Description: This metric returns the fraction of overall achieved MC/DC coverage that comes from requirements-based tests. 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 `RequirementsMCDCCoverageBreakdown`. 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
Capabilities and Limitations: The metric: • Analyzes the overall aggregated coverage results. • Does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.

Metric Information
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

Requirements-Based Tests for each Model in the Unit

When you click on a bar in the **Requirements-Based Tests** section, the **Metric Details** show the coverage ratio for each model in the unit. For requirements-based tests, the coverage ratio is the percentage of the overall achieved coverage that comes from requirements-based tests. Requirements-based tests are test cases that are linked to at least one requirement in the project.

The metrics associated with requirements-based coverage for each model in the unit are:

- “RequirementsExecutionCoverageFragment” on page 5-41 for the requirements-based execution coverage
- “RequirementsDecisionCoverageFragment” on page 5-42 for the requirements-based decision coverage
- “RequirementsConditionCoverageFragment” on page 5-43 for the requirements-based condition coverage
- “RequirementsMCDCCoverageFragment” on page 5-44 for the requirements-based MC/DC coverage

RequirementsExecutionCoverageFragment

Fraction of the overall achieved execution coverage that comes from requirements-based tests.

Metric Information
Metric ID: `RequirementsExecutionCoverageFragment`
Description: This metric returns the fraction of overall achieved execution coverage that comes from requirements-based tests. 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.

Metric Information
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.
Capabilities and Limitations: The metric does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

RequirementsDecisionCoverageFragment

Fraction of the overall achieved decision coverage that comes from requirements-based tests.

Metric Information
Metric ID: `RequirementsDecisionCoverageFragment`
Description: This metric returns the fraction of overall achieved decision coverage that comes from requirements-based tests. 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.

Metric Information
Compliance Thresholds: This metric does not have predefined thresholds.
Capabilities and Limitations: The metric does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

RequirementsConditionCoverageFragment

Fraction of the overall achieved condition coverage that comes from requirements-based tests.

Metric Information
Metric ID: `RequirementsConditionCoverageFragment`
Description: This metric returns the fraction of overall achieved condition coverage that comes from requirements-based tests. 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.
Capabilities and Limitations: The metric does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

RequirementsMCDCCoverageFragment

Fraction of the overall achieved modified condition and decision (MC/DC) coverage that comes from requirements-based tests.

Metric Information
Metric ID: `RequirementsMCDCCoverageFragment`
Description: This metric returns the fraction of overall achieved MC/DC coverage that comes from requirements-based tests. 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.
Capabilities and Limitations: The metric does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

Unit-Boundary Tests for the Unit

The **Achieved Coverage Ratio** section of the dashboard shows the sources of achieved coverage for the unit. The **Unit-Boundary Tests** section shows how much of the overall achieved coverage comes from unit-boundary tests.

The metrics associated with the **Unit-Boundary Tests** section of the dashboard are:

- “UnitBoundaryExecutionCoverageBreakdown” on page 5-45 for the unit-boundary execution coverage

- “UnitBoundaryDecisionCoverageBreakdown” on page 5-45 for the unit-boundary decision coverage
- “UnitBoundaryConditionCoverageBreakdown” on page 5-46 for the unit-boundary condition coverage
- “UnitBoundaryMCDCCoverageBreakdown” on page 5-47 for the unit-boundary MC/DC coverage

UnitBoundaryExecutionCoverageBreakdown

Fraction of the overall achieved execution coverage that comes from unit-boundary tests.

Metric Information
Metric ID: `UnitBoundaryExecutionCoverageBreakdown`
Description: This metric returns the fraction of overall achieved execution coverage that comes from unit-boundary tests. 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.
Capabilities and Limitations: The metric: • Analyzes the overall aggregated coverage results. • Does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

UnitBoundaryDecisionCoverageBreakdown

Fraction of the overall achieved decision coverage that comes from unit-boundary tests.

Metric Information
Metric ID: `UnitBoundaryDecisionCoverageBreakdown`
Description: This metric returns the fraction of overall achieved decision coverage that comes from unit-boundary tests. 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 toolstrip.
Capabilities and Limitations: The metric: • Analyzes the overall aggregated coverage results. • Does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

UnitBoundaryConditionCoverageBreakdown

Fraction of the overall achieved condition coverage that comes from unit-boundary tests.

Metric Information
Metric ID: `UnitBoundaryConditionCoverageBreakdown`

Metric Information
Description: This metric returns the fraction of overall achieved condition coverage that comes from unit-boundary tests. 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.
Capabilities and Limitations: The metric: • Analyzes the overall aggregated coverage results. • Does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

UnitBoundaryMCDCCoverageBreakdown

Fraction of the overall achieved MC/DC coverage that comes from unit-boundary tests.

Metric Information
Metric ID: `UnitBoundaryMCDCCoverageBreakdown`

Metric Information
Description: This metric returns the fraction of overall achieved MC/DC coverage that comes from unit-boundary tests. 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.
Capabilities and Limitations: The metric: • Analyzes the overall aggregated coverage results. • Does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

Unit-Boundary Tests for each Model in the Unit

When you click on a bar in the **Unit-Boundary Tests** section, the **Metric Details** show the coverage ratio for each model in the unit. For unit-boundary tests, the coverage ratio is the percentage of the overall achieved coverage that comes from unit-boundary tests. Unit-boundary tests are test cases that test the whole unit (and not just lower-level subsystems of the unit).

The metrics associated with unit-boundary coverage for each model in the unit are:

• “UnitBoundaryExecutionCoverageFragment” on page 5-49 for the unit-boundary execution coverage
• “UnitBoundaryDecisionCoverageFragment” on page 5-49 for the unit-boundary decision coverage

- “UnitBoundaryConditionCoverageFragment” on page 5-50 for the unit-boundary condition coverage
- “UnitBoundaryMCDCCoverageFragment” on page 5-51 for the unit-boundary MC/DC coverage

UnitBoundaryExecutionCoverageFragment

Fraction of the overall achieved execution coverage that comes from unit-boundary tests.

Metric Information
Metric ID: `UnitBoundaryExecutionCoverageFragment`
Description: This metric returns the fraction of overall achieved execution coverage that comes from unit-boundary tests. 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.
Capabilities and Limitations: The metric does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

UnitBoundaryDecisionCoverageFragment

Fraction of the overall achieved decision coverage that comes from unit-boundary tests.

Metric Information
Metric ID: `UnitBoundaryDecisionCoverageFragment`

Metric Information
Description: This metric returns the fraction of overall achieved decision coverage that comes from unit-boundary tests. 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.
Capabilities and Limitations: The metric does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

UnitBoundaryConditionCoverageFragment

Fraction of the overall achieved condition coverage that comes from unit-boundary tests.

Metric Information
Metric ID: `UnitBoundaryConditionCoverageFragment`

Metric Information
Description: This metric returns the fraction of overall achieved condition coverage that comes from unit-boundary tests. 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.
Capabilities and Limitations: The metric does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

UnitBoundaryMCDCCoverageFragment

Fraction of the overall achieved MC/DC coverage that comes from unit-boundary tests.

Metric Information
Metric ID: `UnitBoundaryMCDCCoverageFragment`

Metric Information
Description: This metric returns the fraction of overall achieved MC/DC coverage that comes from unit-boundary tests. 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.
Capabilities and Limitations: The metric does not analyze coverage from test cases that run in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

Coverage Fragments

Metric Information
Metric ID:
Description:
Results:
Compliance Thresholds:
Capabilities and Limitations:
See Also: For an example of collecting metrics programmatically, see “Collect Metrics on Model Testing Artifacts Programmatically”.

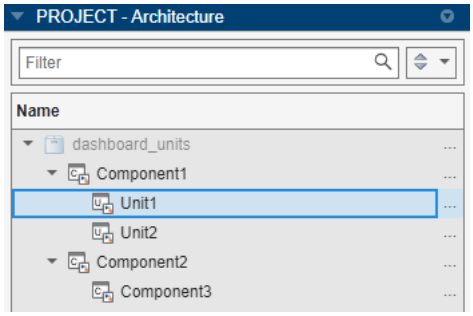
Artifact Tracing

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 test cases, 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”.

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.

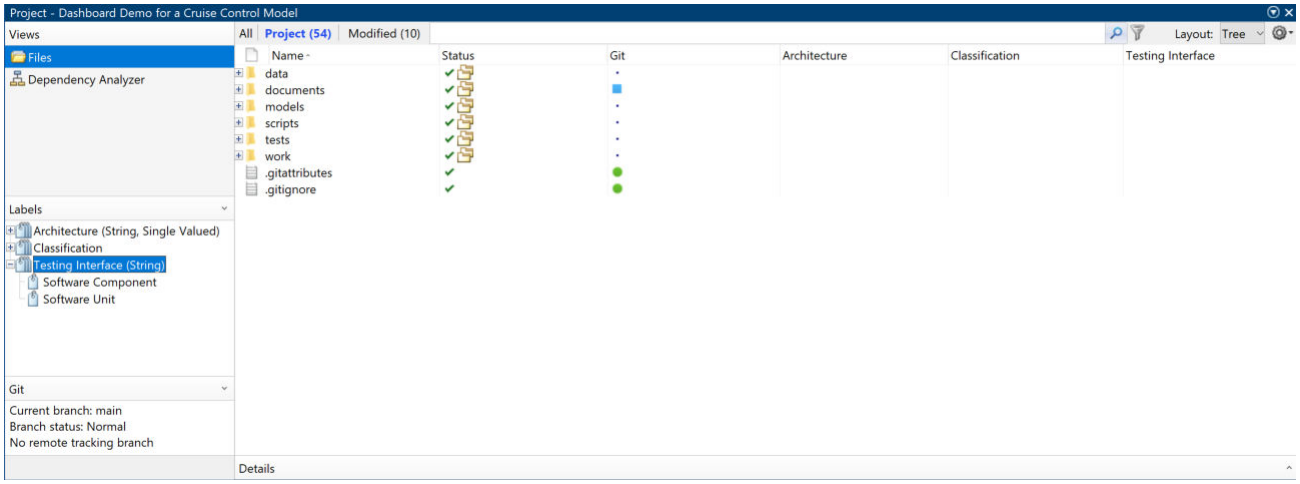
In the Dashboard window, in the **Project** panel, the component icon indicates a component. To see the units under a component, expand the component node by clicking the arrow next to the component icon.

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”.

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.

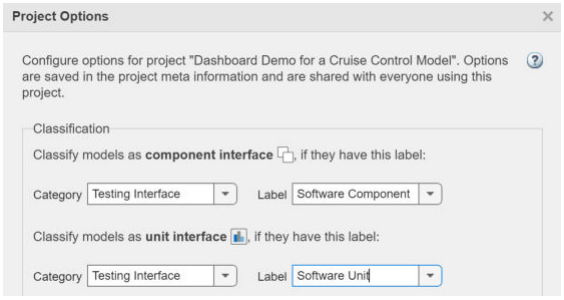
5. Label the models in the project as components and units. In the project pane, right-click a model and click **Add label**. In the dialog box, select the label and click **OK**. For this example, apply these labels:
   - `db_Controller` — Software Component
   - `db_ControlMode` — Software Unit
   - `db_DriverSwRequest` — Software Unit
   - `db_LightControl` — Software Unit
   - `db_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`.

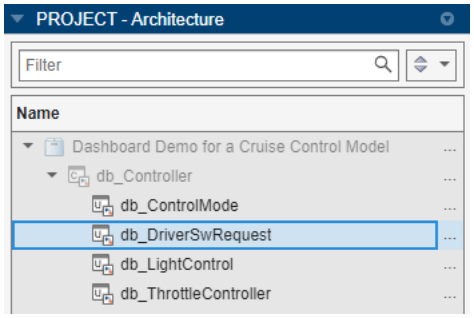


9 Click **Apply**. The dashboard updates the traceability information in the **Project** panel and organizes the models under the component models that reference them. If a model is not referenced by a component, it appears at the top level with the components.

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.

Trace Artifacts to Units and Components

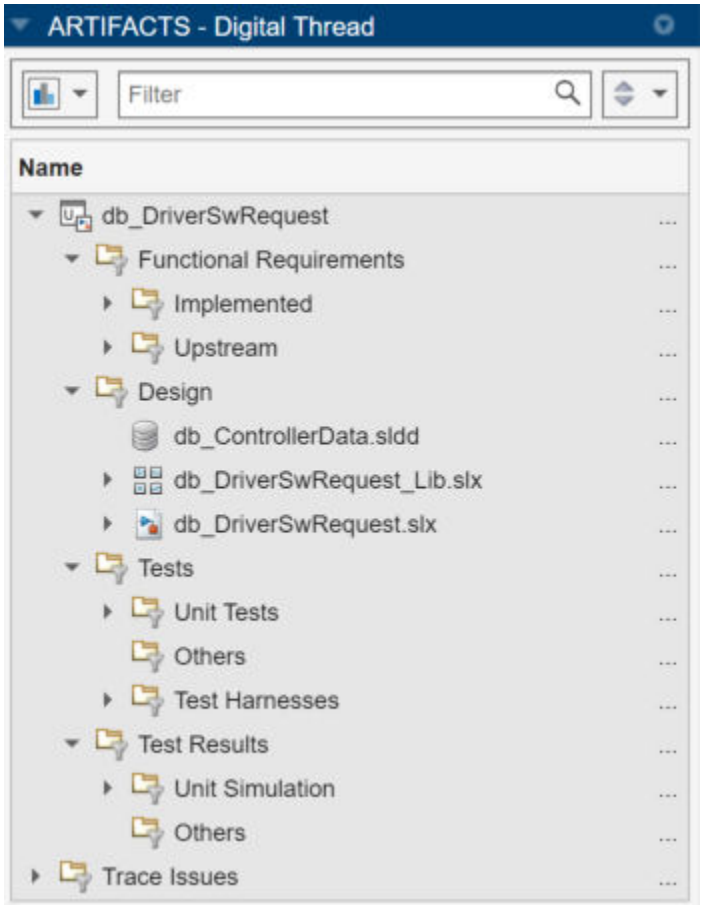
To determine which artifacts are in the scope of a unit or component, the dashboard analyzes the traceability links between the artifacts, software unit models, and component models in the project. The **Project** panel lists the units, organized by the components that reference them.



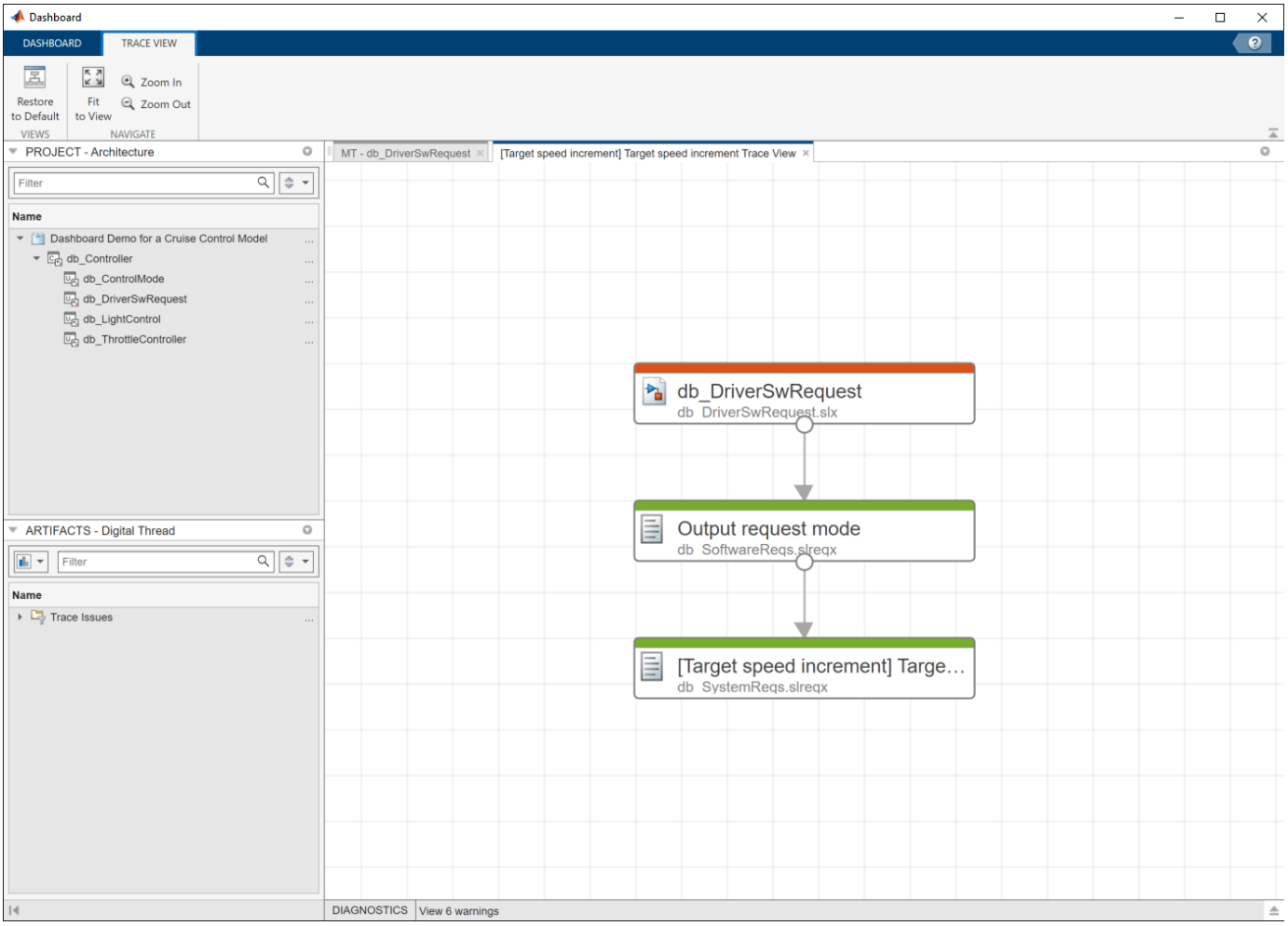
When you select a unit or component in the **Project** panel, the **Artifacts** panel shows the artifacts that trace to the selected unit or component. Traced artifacts include:

- Functional Requirements
- Design Artifacts
- Tests

- Test Results



To see the traceability path that the dashboard found from an artifact to its unit or component, right-click the artifact and click **View trace to dashboard**. A trace view opens in a new tab in the dashboard. The trace view shows the connections and intermediate artifacts that the dashboard traced from the unit or component to the artifact. To see the type of traceability that connects two artifacts, place your cursor over the arrow that connects the artifacts. The traceability relationship is either one artifact containing the other or one artifact tracing to the other. For example, for the unit **db_DriverSwRequest**, expand **Functional Requirements > Upstream > db_SystemReqs.slreqx**. Right-click the requirement for **Target speed increment** and click **View trace to dashboard**. The trace view shows that the unit `db_DriverSwRequest` traces to the implemented functional requirement `Output request mode`, which traces to the upstream functional requirement `Target speed increment`.



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”.

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, at the bottom of the dashboard dialog, click **Diagnostics**.

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" on page 5-60 folder. If a requirement does not appear in the **Artifacts** panel when you expect it to, see "Requirement Missing from Artifacts Panel".

For more information on how the dashboard traces dependencies between project files, see "Digital Thread".

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".

If an artifact does not appear in the **Design** folder when you expect it to, see "Resolve Missing Artifacts, Links, and Results in the Model Testing Dashboard" or "Resolve Missing Artifacts and Results in the Model Maintainability Dashboard". For more information on how the dashboard traces dependencies between project files, see "Digital Thread".

Tests

The folder **Tests** shows test cases and test harnesses that trace to the selected unit.

When you collect metric results for a unit, the dashboard analyzes only the test cases for unit tests. The folder **Tests** contains subfolders to help identify whether a test case is considered a unit test and which test harnesses trace to the unit:

- **Unit Tests** — Test cases that the dashboard considers as unit tests. A unit test directly tests either the entire unit or lower-level elements in the unit, like subsystems. The dashboard uses these tests in the metrics for the unit.
- **Others** — Test cases that trace to the unit but that the dashboard does not consider as unit tests. For example, the dashboard does not consider tests on a library to be unit tests. The dashboard does not use these tests in the metrics for the unit.
- **Test Harnesses** — Test harnesses that trace to the unit or lower-level elements in the unit. Double-click a test harness to open it.

Create test cases in a test suite file by using Simulink Test. If a test case does not trace to a unit, it appears in the “Trace Issues” on page 5-60 folder. If a test case does not appear in the **Artifacts** panel when you expect it to, see “Test Case Missing from Artifacts Panel”. For troubleshooting test cases in metric results, see “Fix a test case that does not produce metric results”.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread”.

Test Results

When you collect metric results for a unit, the dashboard analyzes only the test results from unit tests. The folder **Test Results** contains two subfolders to help identify which test results are from unit tests:

- **Unit Simulation** — Simulation results from unit tests. The dashboard uses these results in the metrics for the unit.

  The following types of test results are shown:

  -  Saved test results — results that you have collected in the Test Manager and have exported to a results file.
  -  Temporary test results — results that you have collected in the Test Manager but have not exported to a results file. When you export the results from the Test Manager the dashboard analyzes the saved results instead of the temporary results. Additionally, the dashboard stops recognizing the temporary results when you close the project or close the result set in the Simulink Test Result Explorer. If you want to analyze the results in a subsequent test session or project session, export the results to a results file.
- **Others** — Results that are not simulation results, are not from unit tests, or are only reports. For example, SIL results are not simulation results. The dashboard does not use these results in the metrics for the unit.

If a test result does not trace to a unit, it appears in the “Trace Issues” on page 5-60 folder. If a test result does not appear in the **Artifacts** panel when you expect it to, see “Test Result Missing from Artifacts Panel”. For troubleshooting test results in dashboard metric results, see “Fix a test result that does not produce metric results”.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread”.

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 in the Model Testing Dashboard”.
- **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 case. For example, if a test case produces a result, but you delete the test case, the dashboard cannot trace the results to the test case.

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”.

Artifact Errors

The folder **Errors** appears if artifacts returned errors when the dashboard performed artifact analysis. These are some errors that artifacts might return during traceability analysis:

- An artifact returns an error if it has unsaved changes when traceability analysis starts.
- A test results file returns an error if it was saved in a previous version of Simulink.
- A model returns an error if it is not on the search path.

Open these artifacts and fix the errors. The dashboard 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.

Diagnostics

To see details about artifacts that cause errors, warnings, and informational messages during analysis, at the bottom of the dashboard dialog, click **Diagnostics**. You can filter the diagnostic messages by their type: **Error**, **Warning**, and **Info**. You can also clear the messages from the viewer.

The diagnostic messages show:

- Modeling constructs that the dashboard does not support
- Links that the dashboard does not trace
- Test harnesses or cases that the dashboard does not support
- Test results missing coverage or simulation results
- Artifacts that return errors when the dashboard loads them
- Information about model callbacks that the dashboard deactivates
- Files that have file shadowing or path traceability issues
- Artifacts that are not on the path and are not considered during tracing

Metric Results Report Generation

Save your metric results by creating a report.

Create a Metric Result Report

In the **Settings** section, you can specify which dashboard you want to create a report for.

In the **Output Options** section, you can specify:

- **File Format**—The file type for the generated report. The generated report can be an HTML file or a PDF.
- **File Name**—The full file name for the generated report.
- **Launch Report**—When the check box is selected, the dashboard automatically opens the generated report.

Click the **Create** button to generate a metric result report. The report file contains the metric results from the dashboard. Saving the metric results in a report file allows you to access the results without opening the project and the dashboard.

To programmatically create a report, use the function `generateReport`.

Trace Artifacts

Trace Artifacts to Units and Components for Analysis

To determine which artifacts are in the scope of a unit and a component, the app analyzes the traceability links between the artifacts and the software units and components in the project. As you edit and save the artifacts in your project, the app tracks your changes and indicates if the traceability data in the **Artifacts** panel is outdated.

Click **Trace Artifacts** to update the traceability data. For more information, see “Digital Thread”.

Collect Metric Results

Collect Metric Results from Analysis

The dashboards can collect metric results for the units and components listed in the **Project** panel. If metric data was previously collected for a unit or component, the dashboard populates from the existing data. If artifacts in the project change after the results are collected, the dashboard might indicate that some dashboard widgets show stale data which does not reflect the changes.

Click **Collect** to collect metric data for the unit or component and to update the stale widgets with data from the current artifacts.

For more information, see “Explore Status and Quality of Testing Activities Using Model Testing Dashboard” and “Monitor the Complexity of Your Design Using the Model Maintainability Dashboard”.

Fix Issues and Trace Artifacts

Fix Issues and Trace Artifacts for Model Testing Analysis

If an artifact returns an error during traceability analysis, the **Artifacts** panel includes the artifact in the **Errors** folder. To see details about the errors that the dashboard finds during artifact analysis, open the **Diagnostics** pane at the bottom of the Dashboard window.

These are some errors that artifacts might cause during traceability analysis:

- An artifact causes an error if it has unsaved changes when traceability analysis starts.
- A test results file causes an error if it was saved in a previous version of Simulink.
- A model causes an error if it is not on the search path.

For information on how to resolve common traceability issues, see “Resolve Missing Artifacts, Links, and Results in the Model Testing Dashboard” or “Resolve Missing Artifacts and Results in the Model Maintainability Dashboard”.

Enable Artifact Tracing

Trace Pending Artifacts to Units and Components for Analysis

As you edit and save the artifacts in your project, the dashboard needs to track these changes to capture recent artifact changes in the metrics and to detect stale results.

The dashboard tracks tool outputs, such as test results from Simulink Test, to detect outdated metric results.

For more information, see “Enable Artifact Tracing for the Project” and “Digital Thread”.

Unanalyzed Artifacts

Unanalyzed Artifacts During First-Time Setup

If you have not opened the Dashboard window for a project, the dashboard performs an initial artifact analysis called **First-Time Setup**. If you click **Cancel** during the initial artifact analysis, unanalyzed artifacts in the project appear in the **Unanalyzed** folder in the **Artifacts** panel.

Click **Collect Results > Trace Artifacts** to perform the initial artifact analysis on the unanalyzed artifacts in the project.

Implemented Requirements

Implemented Requirements

The folder **Implemented** contains functional requirements that are directly linked to the unit with a link **Type** of `Implements`. The dashboard uses these requirements in the metrics for the unit.

Use the Requirements Toolbox to create or import the requirements in a requirements file (`.slreqx`). If a requirement does not trace to a unit, it appears in the “Trace Issues” on page 5-60 folder. If a requirement does not appear in the **Artifacts** panel when you expect it to, see “Requirement Missing from Artifacts Panel”.

You can edit the link type in the **Requirements Editor**. In the **Requirements Editor**, click **Show Links**. Select a link and, in the **Details** pane, under **Properties**, select the desired link type from the **Type** list.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread”.

Upstream Requirements

Upstream Requirements

Upstream Requirements

The folder **Upstream** contains 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.

Use the Requirements Toolbox to create or import the requirements in a requirements file (`.slreqx`). If a requirement does not trace to a unit, it appears in the “Trace Issues” on page 5-60 folder. If a requirement does not appear in the **Artifacts** panel when you expect it to, see “Requirement Missing from Artifacts Panel”.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread”.

Unit Tests

Unit Tests

The folder **Unit Tests** contains test cases that the dashboard considers as unit tests. A unit test directly tests either the entire unit model or the model subsystems. The dashboard uses these tests in the metrics for the unit.

Create test cases in a test suite file by using Simulink Test. If a test case does not trace to a unit, it appears in the “Trace Issues” on page 5-60 folder. If a test case does not appear in the **Artifacts** panel when you expect it to, see “Test Case Missing from Artifacts Panel”. For troubleshooting test cases in metric results, see “Fix a test case that does not produce metric results”.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread”.

Other Tests

Others

The folder **Others** contains test cases that trace to the unit but that the dashboard does not consider as unit tests. For example, the dashboard does not consider tests on a library to be unit tests. The dashboard does not use these tests in the metrics for the unit.

Create test cases in a test suite file by using Simulink Test. If a test case does not trace to a unit, it appears in the “Trace Issues” on page 5-60 folder. If a test case does not appear in the **Artifacts** panel when you expect it to, see “Test Case Missing from Artifacts Panel”. For troubleshooting test cases in metric results, see “Fix a test case that does not produce metric results”.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread”.

Test Harnesses

Test Harnesses

The folder **Test Harnesses** contains test harnesses that trace to the unit or unit subsystems. Double-click a test harness to open it.

Create test cases in a test suite file by using Simulink Test. If a test case does not trace to a unit, it appears in the “Trace Issues” on page 5-60 folder. If a test case does not appear in the **Artifacts** panel when you expect it to, see “Test Case Missing from Artifacts Panel”. For troubleshooting test cases in metric results, see “Fix a test case that does not produce metric results”.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread”.

Unit Simulation

Unit Simulation

The folder **Unit Simulation** contains simulation results from unit tests. The dashboard uses these results in the metrics for the unit.

The following types of test results are shown:

-  Saved test results — results that you have collected in the Test Manager and have exported to a results file.
-  Temporary test results — results that you have collected in the Test Manager but have not exported to a results file. When you export the results from the Test Manager the dashboard analyzes the saved results instead of the temporary results. Additionally, the dashboard stops recognizing the temporary results when you close the project or close the result set in the Simulink Test Result Explorer. If you want to analyze the results in a subsequent test session or project session, export the results to a results file.

If a test result does not trace to a unit, it appears in the “Trace Issues” on page 5-60 folder. If a test result does not appear in the **Artifacts** panel when you expect it to, see “Test Result Missing from Artifacts Panel”. For troubleshooting test results in dashboard metric results, see “Fix a test result that does not produce metric results”.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread”.

Other Results

Others

The folder **Others** contains 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” on page 5-60 folder. If a test result does not appear in the **Artifacts** panel when you expect it to, see “Test Result Missing from Artifacts Panel”. For troubleshooting test results in dashboard metric results, see “Fix a test result that does not produce metric results”.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread”.

Unexpected Implementation Links

Unexpected Implementation Links

The folder **Unexpected Implementation Links** contains 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`.

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, see the troubleshooting solutions in “Trace Issues”.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread”.

Unresolved Links

Unresolved and Unsupported Links

The folder **Unresolved and Unsupported Links** contains 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 and it does not, see the troubleshooting solutions in “Resolve Missing Artifacts, Links, and Results in the Model Testing Dashboard”.

Note that functional requirements must be directly linked to the unit or component with a link **Type** of `Implements`. You can edit the link type in the **Requirements Editor**. In the **Requirements Editor**, click **Show Links**. Select a link and, in the **Details** pane, under **Properties**, select the desired link type from the **Type** list.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread”.

Untraced Tests

Untraced Tests

Untraced Tests

The folder **Untraced Tests** contains tests that execute on models or subsystems that are not on the project path.

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, see the troubleshooting solutions in “Trace Issues”.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread”.

Untraced Results

Untraced Results

The folder **Untraced Results** contains results that the dashboard cannot trace to a test case. For example, if a test case produces a result, but you delete the test case, the dashboard cannot trace the results to the test case.

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, see the troubleshooting solutions in “Trace Issues”.

For more information on how the dashboard traces dependencies between project files, see “Digital Thread”.

Refresh Tasks

Refresh Tasks

As you edit and save the artifacts in your project, the Process Advisor tracks your changes and identifies outdated information. For example:

- 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.
- If you make a change to your `processmodel.m` file, the Process Advisor may need to add or remove the tasks shown in the Process Advisor. For example, if you import a new task into your `processmodel.m` file, the Process Advisor needs to refresh the tasks shown in the Process Advisor.

Click **Refresh Tasks** to refresh the tasks and task results shown in the Process Advisor.

6

Model Maintainability Metrics

Model Maintainability Metrics

The Model Maintainability Dashboard is a model design dashboard that collects metric data from the model design artifacts in a project, such as MATLAB code, Simulink models, and Stateflow charts. Use the metric data to assess the maintainability and complexity of the units and components in your design across the model development lifecycle. Each metric in the dashboard measures a different aspect of the maintainability of your design. Use the widgets in the Model Maintainability Dashboard to see high-level metric results and to gauge the complexity of the units and components in your design.

Alternatively, you can use the API functions to collect metric results programmatically. When using the API, use the metric identifiers (metric IDs) to refer to each metric. You can use the function `getAvailableMetricIds` to return a list of available metric identifiers.

Overall Design Cyclomatic Complexity

Metric ID: `slcomp.OverallCyclomaticComplexity`

Determine the overall design cyclomatic complexity for a unit or component.

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 6-12
- “Stateflow Decision Count” on page 6-26
- “MATLAB Decision Count” on page 6-30

- plus one (for the default path)

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.

Capabilities and Limitations

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.

See Also

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

Layer Depth

Metric ID: `slcomp.LayerDepth`

Determine how many layers deep a model component is in the model hierarchy.

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

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.

Capabilities and Limitations

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.

See Also

For information on finding the maximum layer depth for a unit or component, see “Maximum Layer Depth” on page 6-4.

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

Maximum Layer Depth

Metric ID: `slcomp.MaxLayerDepth`

Determine the maximum number of layers in the model hierarchy of a unit or component.

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

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.

Capabilities and Limitations

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.

See Also

For information on finding the layer depth for each layer of a unit or component, see “Layer Depth” on page 6-3.

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

Layer Breadth

Metric ID: `slcomp.LayerBreadth`

Determine the number of child model components that each artifact contains.

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

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.

Capabilities and Limitations

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.

See Also

For information on finding the maximum layer breadth for a unit or component, see “Maximum Layer Breadth” on page 6-6.

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

Maximum Layer Breadth

Metric ID: `slcomp.MaxLayerBreadth`

Determine the maximum number of child model components that a single model layer in the unit or component contains.

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

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.

Capabilities and Limitations

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.

See Also

For information on finding the layer breadth for each layer of a unit or component, see “Layer Breadth” on page 6-5.

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

Input and Output Component Interface Ports

Metric ID: `slcomp.InterfacePorts`

Determine the number of input ports and output ports to the component interface.

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

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]`.

Capabilities and Limitations

The metric returns the number of component interfaces.

See Also

For information on finding the number of interface signals for a unit or component, see “Input and Output Component Interface Signals” on page 6-9.

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

Input and Output Component Interface Signals

Metric ID: `slcomp.ComponentInterfaceSignals`

Determine the number of input signals and output signals that connect to the component interface.

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

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]`.

Capabilities and Limitations

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.

See Also

For information on finding the number of component interfaces, see “Input and Output Component Interface Ports” on page 6-8.

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

Design Cyclomatic Complexity Breakdown

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 **Design Cyclomatic Complexity Breakdown** section of the dashboard shows the sources of design cyclomatic complexity for a component or unit in your design.

In the **Design Cyclomatic Complexity Breakdown** section of the dashboard:

- The **Complexity** column shows:
  - “Simulink Design Cyclomatic Complexity” on page 6-11
  - “Stateflow Design Cyclomatic Complexity” on page 6-25

    - “MATLAB Design Cyclomatic Complexity” on page 6-28
- The **Distribution** column shows:
    - “Simulink Decision Distribution” on page 6-24
    - “Stateflow Decision Distribution” on page 6-27
    - “MATLAB Decision Distribution” on page 6-32

When you drill in to a widget in the **Design Cyclomatic Complexity Breakdown** section of the dashboard, you can view the **Metric Details** associated with the metrics for:

- “Simulink Decision Count” on page 6-12
- “Stateflow Decision Count” on page 6-26
- “MATLAB Decision Count” on page 6-30

Simulink Design Cyclomatic Complexity

Metric ID: `slcomp.SLCyclomaticComplexity`

Determine the design cyclomatic complexity of the Simulink model components in your design.

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 6-12, 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 6-13.

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 ≥ 0` and the output of the Abs block is equal to the input of the Abs block

Capabilities and Limitations

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.

See Also

For information on how the metric calculates the number of Simulink decisions, see “Simulink Decision Count” on page 6-12.

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

Simulink Decision Count

Metric ID: `slcomp.SimulinkDecisions`

Determine the number of Simulink decisions in each layer of a unit or component.

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	For the decision count, 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 1.
		• 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`
		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).

Cominatorial Logic	Is equal to the number of rows in the `Truth table` parameter) minus `1`	The decision count depends on the number of rows in the truth table. 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.

Block	Decision Count	Description

Dead Zone	2	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.
		
		For the decision count, the Dead Zone block can be treated as an `if-elseif-else` statement:
		
		• If `U >= LL` and `U <= UL`, the output is `0`.
		• If `U > UL`, the output is `U - UL`.
		• Otherwise, the output is `U - LL`.
		
		`if ((U >= LL) & (U <= UL)) % one decisi`
		`  output = 0;`
		`elseif (U > UL) % one decision`
		`  output = U - UL;`
		`else % default path, zero decisions`
		`  output = U - LL;`
		`end`
		
		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).

Block	Decision Count	Description
Discrete-Time Integrator	Is equal to either 0, 2, 3, or 5	The decision count depends on the **External reset** parameter and the **Limit output** parameter. • 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	For the decision count, the Enabled Subsystem block can be treated as an `if-else` statement: • If the subsystem is enabled, the subsystem executes. • Otherwise, by default, the subsystem does not execute. 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).

Block	Decision Count	Description
Enabled and Triggered Subsystem	1	For the decision count, the Enabled and Triggered Subsystem block can be treated as an `if-else` statement: • If the subsystem is enabled and triggered, the subsystem executes. • Otherwise, by default, the subsystem does not execute. 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).
For Iterator Subsystem	1	For the decision count, the For Iterator block can be treated as an `if-else` statement: • 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. 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).

Block	Decision Count	Description

If	Is equal to the number of `if` and `elseif` expressions	Each `if` statement represents a decision and each `elseif` statement represents a decision.
		
		The decision count is equal to the number of `if` and `elseif` statements.
		
		For example, the following code contains one `if` statement and one `elseif` statement and therefore has a decision count of 2:
		
		`if (u > 0) % one decision`
		`  y = 1;`
		`elseif (u < 0) % one decision`
		`  y = 2;`
		`else % default path, zero decisions`
		`  y = 0;`
		`end`

Index Vector	1	The Index Vector block is associated with a decision count of 1. 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. 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.

Block	Decision Count	Description
Multiport Switch	Is equal to the number of data ports minus 1	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. 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. For example, if the number of data inputs is 3, then the decision count is 2. If there is only one data port, the Multiport Switch block acts as an Index Vector block and has a decision count of 1.

Block	Decision Count	Description
Rate Limiter	2	The output of the Rate Limiter block is determined by comparing *rate* to the **Rising slew rate** (R) and **Falling slew rate** (F) parameters. For the decision count, the Rate Limiter block can be treated as an `if-elseif-else` statement: • If *rate* is greater than R, the output is calculated using the rising slew rate. • If *rate* is less than F, the output is calculated using the falling slew rate. • Otherwise, the *rate* is between the bounds of R and F and the change in output is equal to the change in input. 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).

Block	Decision Count	Description
Relay	2	The output of the Relay block is determined by the **Switch off point** and the **Switch on point** parameters. For the decision count, the Relay block can be treated as an `if-elseif-else` statement: • 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. 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).

Block	Decision Count	Description
Saturation	2	The output of the Saturation block is determined by comparing the input to the **Upper limit** and **Lower limit** parameters. For the decision count, the Saturation block can be treated as an `if-elseif-else` statement: • 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. 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).

Block	Decision Count	Description

Switch	1	The Switch block is associated with a decision count of 1:
		• 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.
		For example, the following code shows a decision count of 1:
		`switch (u2 > 0)`
		`  case 1 % one decision`
		`  y = u1;`
		`  otherwise % default path, zero decisi`
		`  y = u3;`
		`end`

Switch Case	Is equal to the number of outports minus 1	The `default` case represents the default behavior (no decisions made). But each subsequent `case` represents a decision.
		The decision count is equal to the number of outports minus 1.
		For example, the following code represents a Switch Case block with 3 outputs and has a decision count of 2:
		`switch u1`
		`  case 1 % one decision`
		`  y = port1;`
		`  case 2 % one decision`
		`  y = port2;`
		`  otherwise % default path, zero decisi`
		`  y = port3;`
		`end`

Block	Decision Count	Description
Triggered Subsystem	1	For the decision count, the Triggered Subsystem block can be treated as an `if-else` statement: • If the subsystem is triggered, the subsystem executes. • Otherwise, by default, the subsystem does not execute. 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).

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” on page 6-13.

Capabilities and Limitations

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.

See Also

For related metrics, see:

- “Simulink Design Cyclomatic Complexity” on page 6-11
- “Simulink Decision Distribution” on page 6-24

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

Simulink Decision Distribution

Metric ID: `slcomp.SimulinkDecisionDistribution`

Determine the distribution of the number of Simulink decisions in a unit or component.

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” on page 6-12.

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

For information on how the metric calculates the number of Simulink decisions, see “Simulink Decision Count” on page 6-12.

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

Stateflow Design Cyclomatic Complexity

Metric ID: `slcomp.SFCyclomaticComplexity`

Determine the design cyclomatic complexity of the Stateflow components in your design.

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 6-26, 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 6-26.

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 6-26.

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.

Capabilities and Limitations

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.

See Also

For information on how the number of Stateflow decisions is calculated, see “Stateflow Decision Count” on page 6-26.

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

Stateflow Decision Count

Metric ID: `slcomp.StateflowDecisions`

Determine the number of Stateflow decisions in each layer of a unit or component.

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 transitions with conditions plus the number of states minus 1 (This includes decisions in Entry, During, and Exit Actions.)
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” on page 6-26.

See Also

For related metrics, see:

- “Stateflow Design Cyclomatic Complexity” on page 6-25
- “Stateflow Decision Distribution” on page 6-27

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

Stateflow Decision Distribution

Metric ID: `slcomp.StateflowDecisionsDistribution`

Determine the distribution of the number of Stateflow decisions in each layer of a unit or component.

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 6-26.

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

For information on how the number of Stateflow decisions is calculated, see “Stateflow Decision Count” on page 6-26.

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

MATLAB Design Cyclomatic Complexity

Metric ID: `slcomp.MatlabCyclomaticComplexity`

Determine the design cyclomatic complexity of the MATLAB code in your design.

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 6-30, plus one for the default path. The default path is only counted once per unit or component.

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 ≥ 0` and the output of the MATLAB Function block is equal to the input of the MATLAB Function block

Capabilities and Limitations

The metric includes the decisions associated with logical operations that might be short-circuited during code execution.

See Also

For information on how the number of MATLAB decisions is calculated, see “MATLAB Decision Count” on page 6-30.

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

MATLAB Decision Count

Metric ID: `slcomp.MATLABDecisions`

Determine the number of decisions in MATLAB code associated with a unit or component.

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 `otherwise` 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 `parfor` statements	Each `parfor` statement represents a decision in the code.
`try`	Is equal to the number of `try` statements	Each `try` statement represents a decision in the code.
`while`	Is equal to the number of `while` statements	Each `while` statement represents a decision in the code.

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.

Capabilities and Limitations

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`.
- Counts decisions from `if`, `elseif`, `while`, `for`, `parfor`, `try`, and `case` statements.
- Ignores `else`, `otherwise`, and `catch` statements because they form the default path.

See Also

For related metrics, see:

- “MATLAB Design Cyclomatic Complexity” on page 6-28
- “MATLAB Decision Distribution” on page 6-32

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

MATLAB Decision Distribution

Metric ID: `slcomp.MATLABDecisionsDistribution`

Determine the distribution of the number of decisions in MATLAB code.

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 6-30.

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

For information on how the number of MATLAB decisions is calculated, see “MATLAB Decision Count” on page 6-30.

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

Simulink Architecture

The **Simulink Architecture** section of the dashboard shows the number of Simulink blocks, signal lines, and Goto blocks in the layers of your unit or component.

In the **Simulink Architecture** section of the dashboard:

- The **Count** column shows:
  - “Overall Blocks” on page 6-34
  - “Overall Signal Lines” on page 6-36
  - “Overall Goto Blocks” on page 6-38
- The **Distribution** column shows:
  - “Simulink Blocks Distribution” on page 6-35
  - “Simulink Signal Lines Distribution” on page 6-37

- “Simulink Goto Blocks Distribution” on page 6-39

When you drill in to a widget in the **Simulink Architecture** section of the dashboard, you can view the **Metric Details** associated with the metrics for:

- “Simulink Blocks” on page 6-34
- “Simulink Signal Lines” on page 6-37
- “Simulink Goto Blocks” on page 6-39

Overall Blocks

Metric ID: `slcomp.OverallBlocks`

Determine the overall number of blocks in a unit or component.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Simulink Blocks” on page 6-34
- “Simulink Blocks Distribution” on page 6-35

Simulink Blocks

Metric ID: `slcomp.SimulinkBlocks`

Determine the number of Simulink blocks, excluding Inport, Outport, and Goto blocks, in each layer of a unit or component.

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.

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.

Capabilities and Limitations

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.

See Also

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Overall Blocks” on page 6-34
- “Simulink Blocks Distribution” on page 6-35

Simulink Blocks Distribution

Metric ID: `slcomp.BlocksDistribution`

Determine the distribution of the number of Simulink blocks, excluding Inport, Outport, and Goto blocks, in each layer of a unit or component.

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.

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.

Capabilities and Limitations

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.

See Also

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Overall Blocks” on page 6-34
- “Simulink Blocks” on page 6-34

Overall Signal Lines

Metric ID: `slcomp.OverallSignalLines`

Determine the overall number of signal lines in a unit or component.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Simulink Signal Lines” on page 6-37
- “Simulink Signal Lines Distribution” on page 6-37

Simulink Signal Lines

Metric ID: `slcomp.SimulinkSignalLines`

Determine the number of signal lines in each layer of a unit or component.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Overall Signal Lines” on page 6-36
- “Simulink Signal Lines Distribution” on page 6-37

Simulink Signal Lines Distribution

Metric ID: `slcomp.SignalLinesDistribution`

Determine the distribution of the number of signal lines in a unit or component.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Overall Signal Lines” on page 6-36
- “Simulink Signal Lines” on page 6-37

Overall Goto Blocks

Metric ID: `slcomp.OverallGotos`

Determine the overall number of Goto blocks in a unit or component.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Simulink Goto Blocks” on page 6-39
- “Simulink Goto Blocks Distribution” on page 6-39

Simulink Goto Blocks

Metric ID: `slcomp.SimulinkGotos`

Determine the number of Simulink Goto blocks in each layer of a unit or component.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Overall Goto Blocks” on page 6-38
- “Simulink Goto Blocks Distribution” on page 6-39

Simulink Goto Blocks Distribution

Metric ID: `slcomp.GotosDistribution`

Determine the distribution of the number of Simulink Goto blocks in each layer of a unit or component.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Overall Goto Blocks” on page 6-38
- “Simulink Goto Blocks” on page 6-39

Stateflow Architecture

The **Stateflow Architecture** section of the dashboard shows the number of Stateflow transitions and states in the layers of your unit or component.

In the **Stateflow Architecture** section of the dashboard:

- The **Count** column shows:
  - “Overall Transitions” on page 6-41
  - “Overall States” on page 6-42
- The **Distribution** column shows:

- “Stateflow Transitions Distribution” on page 6-42
- “Stateflow States Distribution” on page 6-44

When you drill in to a widget in the **Stateflow Architecture** section of the dashboard, you can view the **Metric Details** associated with the metrics for:

- “Stateflow Transitions” on page 6-41
- “Stateflow States” on page 6-43

Overall Transitions

Metric ID: `slcomp.OverallTransitions`

Determine the overall number of transitions in a unit or component.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Stateflow Transitions” on page 6-41
- “Stateflow Transitions Distribution” on page 6-42

Stateflow Transitions

Metric ID: `slcomp.StateflowTransitions`

Determine the number of Stateflow transitions in each layer of a unit or component.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Overall Transitions” on page 6-41
- “Stateflow Transitions Distribution” on page 6-42

Stateflow Transitions Distribution

Metric ID: `slcomp.TransitionsDistribution`

Determine the distribution of the number of Stateflow transitions in each layer of a unit or component.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Overall Transitions” on page 6-41
- “Stateflow Transitions” on page 6-41

Overall States

Metric ID: `slcomp.OverallStates`

Determine the overall number of states in a unit or component.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Stateflow States” on page 6-43
- “Stateflow States Distribution” on page 6-44

Stateflow States

Metric ID: `slcomp.StateflowStates`

Determine the number of Stateflow states in each layer of a unit or component.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Overall States” on page 6-42
- “Stateflow States Distribution” on page 6-44

Stateflow States Distribution

Metric ID: `slcomp.StatesDistribution`

Determine the distribution of the number of Stateflow states in each layer of a unit or component.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Overall States” on page 6-42
- “Stateflow States” on page 6-43

MATLAB Architecture

The **MATLAB Architecture** section of the dashboard shows the number of effective lines of MATLAB code in your unit or component.

In the **MATLAB Architecture** section of the dashboard:

- The **Count** column shows:
  - “Overall MATLAB Executable Lines of Code (eLOC)” on page 6-45
- The **Distribution** column shows:
  - “MATLAB Effective Lines of Code (eLOC) Distribution” on page 6-46

When you drill in to a widget in the **MATLAB Architecture** section of the dashboard, you can view the **Metric Details** associated with the metric for:

- “MATLAB Effective Lines of Code (eLOC)” on page 6-45

Overall MATLAB Executable Lines of Code (eLOC)

Metric ID: `slcomp.OverallMATLABeLOC`

Determine the overall number of executable lines of MATLAB code in a unit or component.

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

To view the number of lines of MATLAB code associated with each model component, see “MATLAB Effective Lines of Code (eLOC)” on page 6-45.

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

MATLAB Effective Lines of Code (eLOC)

Metric ID: `slcomp.MATLABeLOC`

Determine the number of effective lines of MATLAB code.

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 6-45.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Overall MATLAB Executable Lines of Code (eLOC)” on page 6-45
- “MATLAB Effective Lines of Code (eLOC) Distribution” on page 6-46

MATLAB Effective Lines of Code (eLOC) Distribution

Metric ID: `slcomp.MATLABeLOCDistribution`

Determine the distribution of the number of effective lines of MATLAB code.

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 6-45.

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

For an example of collecting metrics programmatically, see “Collect Model Maintainability Metrics Programmatically”.

- “Overall MATLAB Executable Lines of Code (eLOC)” on page 6-45
- “MATLAB Effective Lines of Code (eLOC)” on page 6-45

Apps

Clone Detector

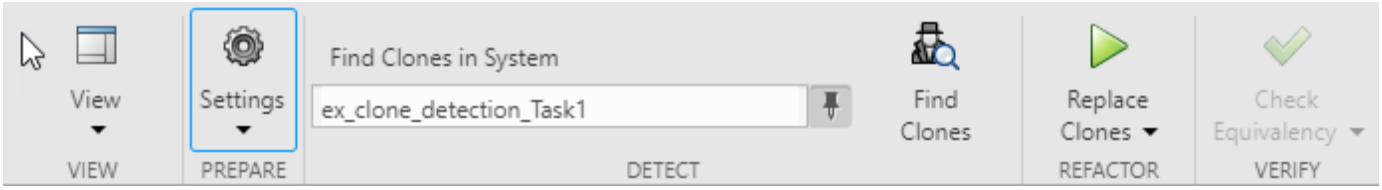
Enable model refactorization and subsystem reuse in models by identifying and replacing clones

Description

The **Clone Detector** is a tool that identifies and replaces clones, which are modeling patterns that have identical block types and connections. The tool identifies clones across referenced model boundaries. You can refactor your model by replacing the clones with library links or Subsystem Reference blocks, which enables you to reuse components.

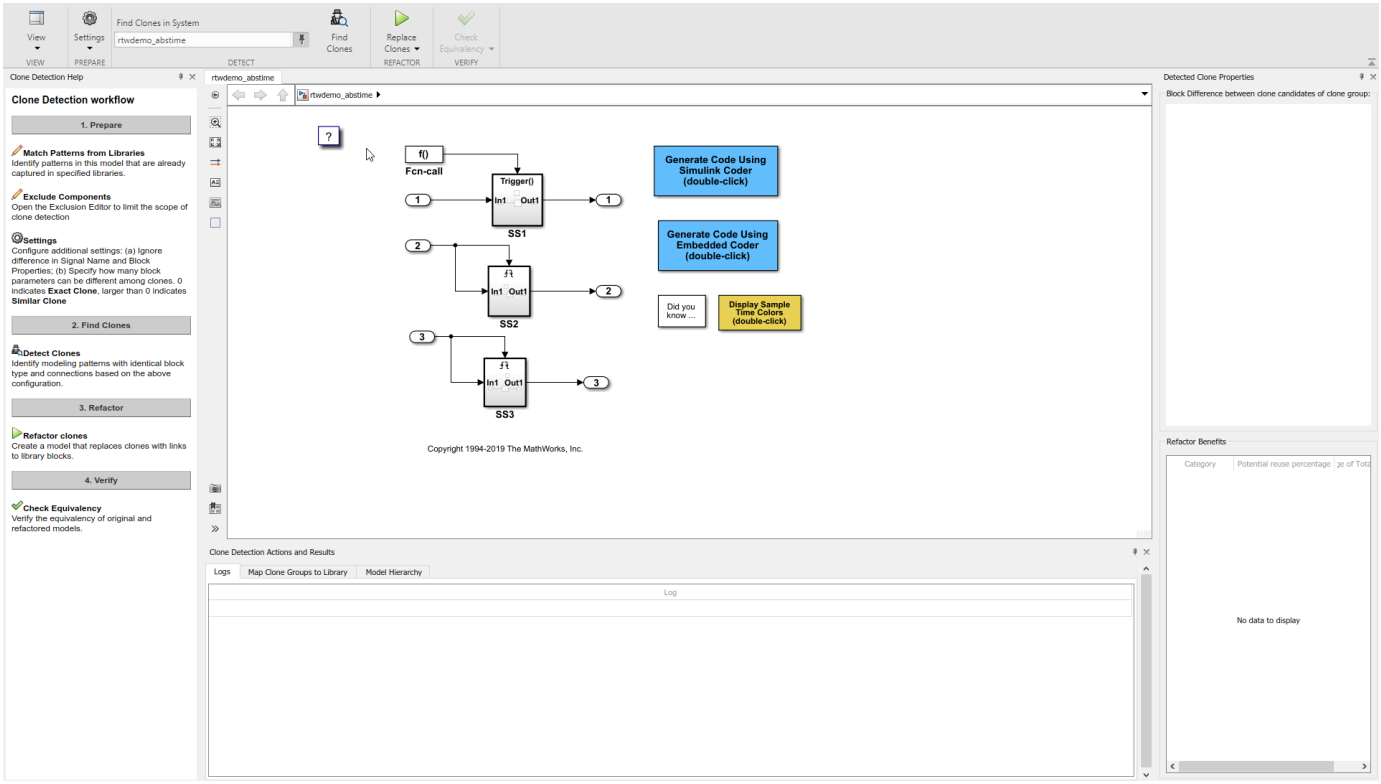
The Clone Detector interface steps you through the process of identifying and refactoring clones. You can:

- Access clone reuse percentages in the model to determine the benefits of refactoring.
- Vary parameter tolerance to identify similar and exact clones.
- Identify the differences in a clone from the baseline subsystem.
- Access a log of clone detection results.
- Use a Simulink Test license to access an embedded Test Manager that allows you to verify the equivalence of the refactored model and the original model.



To identify and replace clones in the model:

1. Open the model `rtwdemo_abstime`.
2. Save the model to your working folder.
3. On the **Apps** tab, click **Clone Detector**.
4. In the Clone Detector tab, on the **View** section, you can select **Help** pane, **Properties** to show the Detected Clone Properties pane, or **Results** to show the Clone Detection Actions and Results pane.
5. Click **Find Clones** to identify clones.
6. Click the Clone Detection Action and Results pane and browse the results to view the identified clones.
7. Click **Replace Clones** to replace the clone groups. A backup model with the original layout is saved. Click the **Restore** button in the clone detection logs to revert to the original model.
8. Click **Check Equivalency** to open the Test Manager. This tests whether the refactored model is functionally equivalent to the original model.



Open the Clone Detector App

In the **Apps** gallery, click **Clone Detector**.

Examples

- “Custom Libraries”
- “Generate Reusable Code from Library Subsystems Shared Across Models” (Simulink Coder)
- “Replace Exact Clones with Subsystem Reference”
- “Enable Component Reuse by Using Clone Detection”

Version History

Introduced in R2019b

See Also

Topics

“Custom Libraries”
“Generate Reusable Code from Library Subsystems Shared Across Models” (Simulink Coder)
“Replace Exact Clones with Subsystem Reference”
“Enable Component Reuse by Using Clone Detection”

Model Transformer

Enable model transformation by identifying and refactoring the modelling patterns to optimize the models

Description

The **Model Transformer** is a tool to refactor a model to implement variants, eliminate eligible data store blocks, and improve the simulation and code efficiency of table lookup operations. You can perform the steps in the Model Transformer all at once or one step at a time.

You can use the Model Transformer app to replace:

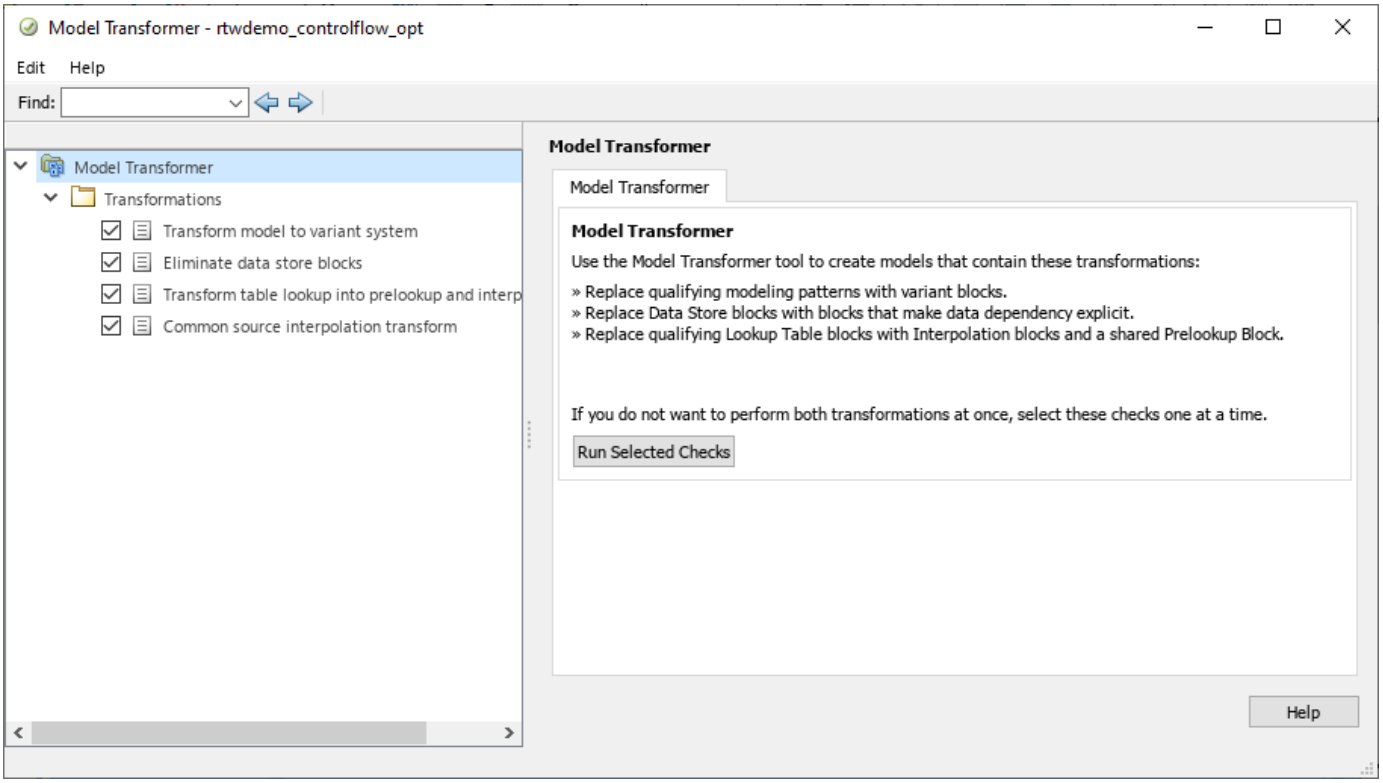
- Qualifying modeling patterns with variant blocks.
- Data store blocks with blocks that make data dependency explicit.
- Lookup Table blocks with Interpolation using Prelookup blocks.
- Modeling patterns with multiple Interpolation using Prelookup blocks into a single Interpolation using Prelookup block.

The Model Transformer tool provides the following checks:

- **Transform model to variant system**
- **Eliminate data store blocks**
- **Transform table lookup into prelookup and interpolation**
- **Common source interpolation transform**

If you want to perform all the transformations, for each step, specify the input parameters. Then, click the **Run Selected Checks** button. After you run each check, create new models with the transformations by clicking the **Refactor Model** buttons.

If you want to perform one transformation at a time, you can individually select the checks.



Open the Model Transformer App

In the **Apps** gallery, click **Model Transformer**.

Examples

- “Transform Model to Variant System”
- “Replace Data Store Blocks”
- “Improve Efficiency of Simulation by Optimizing Prelookup Operation of Lookup Table Blocks”
- “Improve Code Efficiency by Merging Multiple Interpolation Using Prelookup Blocks”

Version History

Introduced in R2019b

See Also

Topics

“Transform Model to Variant System”
“Replace Data Store Blocks”
“Improve Efficiency of Simulation by Optimizing Prelookup Operation of Lookup Table Blocks”
“Improve Code Efficiency by Merging Multiple Interpolation Using Prelookup Blocks”