Simulink® Coverage™

User's Guide

MATLAB®&SIMULINK®

MathWorks®

How to Contact MathWorks

	Latest news:	www.mathworks.com
	Sales and services:	www.mathworks.com/sales_and_services
	User community:	www.mathworks.com/matlabcentral
	Technical support:	www.mathworks.com/support/contact_us
	Phone:	508-647-7000

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

Contents

Setting Coverage Options

3

<div style="text-align: right">

Code Coverage

</div>

4

Results Review

6

Filtering in Simulink Coverage

7

Automating Model Coverage Tasks

8

Component Verification

9

Verification and Validation

10

Model Coverage Definition

Model Coverage

Model coverage helps you validate your model tests by measuring how thoroughly the model objects are tested. Model coverage calculates how much a model test case exercises simulation pathways through a model. It is a measure of how thoroughly a test case tests a model and the percentage of pathways that a test case exercises.

Model coverage analyzes the execution of the following types of model objects that directly or indirectly determine simulation pathways through your model:

- Simulink® blocks
- Models referenced in Model blocks
- The states and transitions of Stateflow® charts

During a simulation run, the tool records the behavior of the covered objects, states, and transitions. At the end of the simulation, the tool reports the extent to which the run exercised potential simulation pathways through each covered object in the model.

For the types of coverage that model coverage performs, see "Types of Model Coverage" on page 1-3. For an example of a model coverage report, see "Top-Level Model Coverage Report" on page 6-10.

The Simulink Coverage™ software can only collect model coverage for a model if its simulation mode is set to `Normal`. If the simulation mode is set to any other mode, model coverage is not measured during simulation.

If you have an Embedded Coder® license, you can also measure code coverage for code generated from models in software-in-the-loop (SIL) mode or processor-in-the-loop (PIL) mode. For the types of coverage that code coverage performs, see "Types of Code Coverage" on page 4-2. For an example of how to enable code coverage, see "Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode" on page 4-6

Types of Model Coverage

Simulink Coverage can perform several types of coverage analysis.

Execution Coverage (EC)

Execution coverage is the most basic form of coverage. For each item, execution coverage determines whether the item is executed during simulation.

Decision Coverage (DC)

Decision coverage analyzes elements that represent decision points in a model, such as a Switch block or Stateflow states. For each item, decision coverage determines the percentage of the total number of simulation paths through the item that the simulation traversed.

For an example of decision coverage data in a model coverage report, see "Decisions Analyzed" on page 6-23.

Condition Coverage (CC)

Condition coverage analyzes blocks that output the logical combination of their inputs (for example, the Logical Operator block) and Stateflow transitions. A test case achieves full coverage when it causes each input to each instance of a logic block in the model and each condition on a transition to be true at least once during the simulation, and false at least once during the simulation. Condition coverage analysis reports whether the test case fully covered the block for each block in the model.

When you collect coverage for a model, you may not be able to achieve 100% condition coverage. For example, if you specify to short-circuit logic blocks, by selecting **Treat Simulink Logic blocks as short-circuited** in the **Coverage** pane in the Configuration Parameters, you might not be able to achieve 100% condition coverage for that block. See "MCDC Analysis" on page 6-24 for more information.

For an example of condition coverage data in a model coverage report, see "Conditions Analyzed" on page 6-24.

Modified Condition/Decision Coverage (MCDC)

Modified condition/decision coverage analysis by the Simulink Coverage software extends the decision and condition coverage capabilities. It analyzes blocks that output the logical combination of their inputs and Stateflow transitions to determine the extent to which the test case tests the independence of logical block inputs and transition conditions.

- A test case achieves full coverage for a block when a change in one input, independent of any other inputs, causes a change in the block output.
- A test case achieves full coverage for a Stateflow transition when there is at least one time when a change in the condition triggers the transition for each condition.

If your model contains blocks that define expressions that have different types of logical operators and more than 12 conditions, the software cannot record MCDC coverage.

Because the Simulink Coverage MCDC coverage may not achieve full decision or condition coverage, you can achieve 100% MCDC coverage *without* achieving 100% decision coverage.

Some Simulink objects support MCDC coverage, some objects support only condition coverage, and some objects support only decision coverage. The table in "Model Objects That Receive Coverage" on page 2-2 lists which objects receive which types of model coverage. For example, the Combinatorial Logic block can receive decision coverage and condition coverage, but not MCDC coverage.

To achieve 100% MCDC coverage for your model, as defined by the DO-178C/DO-331 standard, in the **Coverage** pane of the Configuration Parameters, select "Modified Condition/Decision Coverage (MCDC)" on page 1-4 as the **Structural coverage level**.

When you collect coverage for a model, you may not be able to achieve 100% MCDC coverage. For example, if you specify to short-circuit logic blocks, you may not be able to achieve 100% MCDC coverage for that block.

If you run the test cases independently and accumulate all the coverage results, you can determine if your model adheres to the modified condition and decision coverage standard. For more information about the DO-178C/DO-331 standard, see "DO-178C/DO-331 Checks" (Simulink Check).

For an example of MCDC coverage data in a model coverage report, see "MCDC Analysis" on page 6-24. For an example of accumulated coverage results, see "Cumulative Coverage" on page 6-25.

Cyclomatic Complexity

Cyclomatic complexity is a measure of the structural complexity of a model. It approximates the McCabe complexity measure for code generated from the 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), model coverage uses the following 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 atomic subsystems and Stateflow charts.

For an example of cyclomatic complexity data in a model coverage report, see "Cyclomatic Complexity" on page 6-21.

Lookup Table Coverage

Lookup table coverage (LUT) examines blocks, such as the 1-D Lookup Table block, that output information from inputs in a table of inputs and outputs, interpolating between or extrapolating from table entries. Lookup table coverage records the frequency that table lookups use each interpolation interval. A test case achieves full coverage when it executes each interpolation and extrapolation interval at least once. For each lookup table block in the model, the coverage report displays a colored map of the lookup table, indicating each interpolation. If the total number of breakpoints of an n-D Lookup Table block exceeds 1,500,000, the software cannot record coverage for that block.

For an example of lookup table coverage data in a model coverage report, see "N-Dimensional Lookup Table" on page 6-27.

Note Configure lookup table coverage only at the start of a simulation. If you tune a parameter that affects lookup table coverage at run time, the coverage settings for the affected block are not updated.

Signal Range Coverage

Signal range coverage records the minimum and maximum signal values at each block in the model, as measured during simulation. Only blocks with output signals receive signal range coverage.

The software does not record signal range coverage for control signals, signals used by one block to initiate execution of another block. See "Control Signals".

If the total number of signals in your model exceeds 65535, or your model contains a signal whose width exceeds 65535, the software cannot record signal range coverage.

For an example of signal range coverage data in a model coverage report, see "Signal Range Analysis" on page 6-35.

Note When you create cumulative coverage for reusable subsystems or Stateflow constructs with single range coverage, the cumulative coverage has the largest possible range of signal values. For more information, see "Obtain Cumulative Coverage for Reusable Subsystems and Stateflow® Constructs" on page 5-34.

Signal Size Coverage

Signal size coverage records the minimum, maximum, and allocated size for all variable-size signals in a model. Only blocks with variable-size output signals are included in the report.

If the total number of signals in your model exceeds 65535, or your model contains a signal whose width exceeds 65535, the software cannot record signal size coverage.

For an example of signal size coverage data in a model coverage report, see "Signal Size Coverage for Variable-Dimension Signals" on page 6-36.

For more information about variable-size signals, see "Variable-Size Signal Basics".

Objectives and Constraints Coverage

The Simulink Coverage software collects model coverage data for the following Simulink Design Verifier™ blocks and MATLAB® for code generation functions:

Simulink Design Verifier blocks	MATLAB for code generation functions
Test Condition	sldv.condition
Test Objective	sldv.test
Proof Assumption	sldv.assume
Proof Objective	sldv.prove

If you do not have a Simulink Design Verifier license, you can collect model coverage for a model that contains these blocks or functions, but you cannot analyze the model using the Simulink Design Verifier software.

By adding one or more Simulink Design Verifier blocks or functions into your model, you can:

- Check the results of a Simulink Design Verifier analysis, run generated test cases, and use the blocks to observe the results.
- Define model requirements using the Test Objective block and verify the results with model coverage data that the software collected during simulation.
- Analyze the model, create a test harness, and simulate the harness with the Test Objective block to collect model coverage data.
- Analyze the model and use the Proof Assumption block to verify any counterexamples that the Simulink Design Verifier identifies.

If you specify to collect Simulink Design Verifier coverage:

- The software collects coverage for the Simulink Design Verifier blocks and functions.
- The software checks the data type of the signal that links to each Simulink Design Verifier block. If the signal data type is fixed point, the block parameter must also be fixed point. If the signal data type is not fixed point, the software tries to convert the block parameter data type. If the software cannot convert the block parameter data type, the software reports an error and you must explicitly assign the block parameter data type to match the signal.
- If your model contains a Verification Subsystem block, the software only records coverage for Simulink Design Verifier blocks in the Verification Subsystem block; it does not record coverage for any other blocks in the Verification Subsystem.

If you do not specify to collect Simulink Design Verifier coverage, the software does not check the data types for any Simulink Design Verifier blocks and functions in your model and does not collect coverage.

For an example of coverage data for Simulink Design Verifier blocks or functions in a model coverage report, see "Simulink Design Verifier Coverage" on page 6-37.

Saturate on Integer Overflow Coverage

Saturate on integer overflow coverage examines blocks, such as the Abs block, with the **Saturate on integer overflow** parameter selected. Only blocks with this parameter selected receive saturate on integer overflow coverage.

Saturate on integer overflow coverage records the number of times the block saturates on integer overflow.

A test case achieves full coverage when the blocks saturate on integer overflow at least once and does not saturate at least once.

For an example of saturate on integer overflow coverage data in a model coverage report, see "Saturate on Integer Overflow Analysis" on page 6-34.

Relational Boundary Coverage

Relational boundary coverage examines blocks, Stateflow charts, and MATLAB function blocks that have an explicit or implicit relational operation.

- Blocks such as Relational Operator and If have an explicit relational operation.
- Blocks such as Abs and Saturation have an implicit relational operation.

For these model objects, the metric records whether a simulation tests the relational operation with:

- Equal operand values.

  This part of relational boundary coverage applies only if both operands are integers or fixed-point numbers.
- Operand values that differ by a certain tolerance.

  This part of relational boundary coverage applies to all operands. For integer and fixed-point operands, the tolerance is fixed. For floating-point operands, you can either use a predefined tolerance or you can specify your own tolerance.

The tolerance value depends on the data type of both the operands. If both operands have the same type, the tolerance follows the following rules:

Data Type of Operand	Tolerance
Floating point, such as `single` or `double`	`max(absTol, relTol* max(|lhs|,|rhs|))` • `absTol` is an absolute tolerance value you specify. Default is `1e-05`. • `relTol` is a relative tolerance value you specify. Default is `0.01`. • `lhs` is the left operand and `rhs` the right operand. • `max(x,y)` returns x or y, whichever is greater.

Data Type of Operand	Tolerance
Fixed point	Value corresponding to least significant bit. For more information, see "Precision" (Fixed-Point Designer). To find the precision value, use the `lsb` (Fixed-Point Designer) function.
Integer	1
Boolean	N/A
Enum	N/A

If the two operands have different types, the tolerance follows the rules for the stricter type. If one of the operands is boolean, the tolerance follows the rules for the other operand. The strictness decreases in this order:

1   Floating point

2   Fixed point

3   Integer

If both operands are fixed point but have different precision, the smaller value of precision is used as tolerance.

You specify the value of absolute and relative tolerances for relational boundary coverage of floating point inputs when you select this metric in the **Coverage metrics** section in the "Coverage Pane" on page 3-2 of the Configuration Parameters dialog box.

For more information on:

- How this coverage metric appears in reports, see "Relational Boundary" on page 6-32.
- Which model objects receive this coverage, see the table in "Model Objects That Receive Coverage" on page 2-2.
- How to obtain coverage results from the MATLAB command-line, see "Collect Relational Boundary Coverage for Supported Block in Model".

Simulink Optimizations and Model Coverage

In the Configuration Parameters dialog box, there are three Simulink optimization parameters that can affect your model coverage data:

Inlined Parameters

To transform tunable model parameters into constant values for code generation, in the Configuration Parameters dialog box, on the **Math and Data Types** pane, set **Default parameter behavior** to `Inlined`.

When the parameters are transformed into constants, Simulink may eliminate certain decisions in your model. You cannot achieve coverage for eliminated decision, so the coverage report displays `0/0` for those decisions.

Block Reduction

To achieve faster execution during model simulation and in generated code, in the Configuration Parameters dialog box, select the **Block reduction** parameter. The Simulink software collapses certain groups of blocks into a single, more efficient block, or removes them entirely.

One of the model coverage options, **Force block reduction off**, allows you to ignore the **Block reduction** parameter when collecting model coverage.

If you do not select the **Block reduction** parameter, or if you select **Force block reduction off**, the Simulink Coverage software provides coverage data for every block in the model that collects coverage.

If you select the **Block reduction** parameter and do not set **Force block reduction off**, the coverage report lists the reduced blocks that would have collected coverage.

Conditional Input Branch Execution

The **Conditional input branch execution** parameter can cause lower than expected Simulink Coverage results.

Case 1: Upstream Switch Block Completely Optimized Out
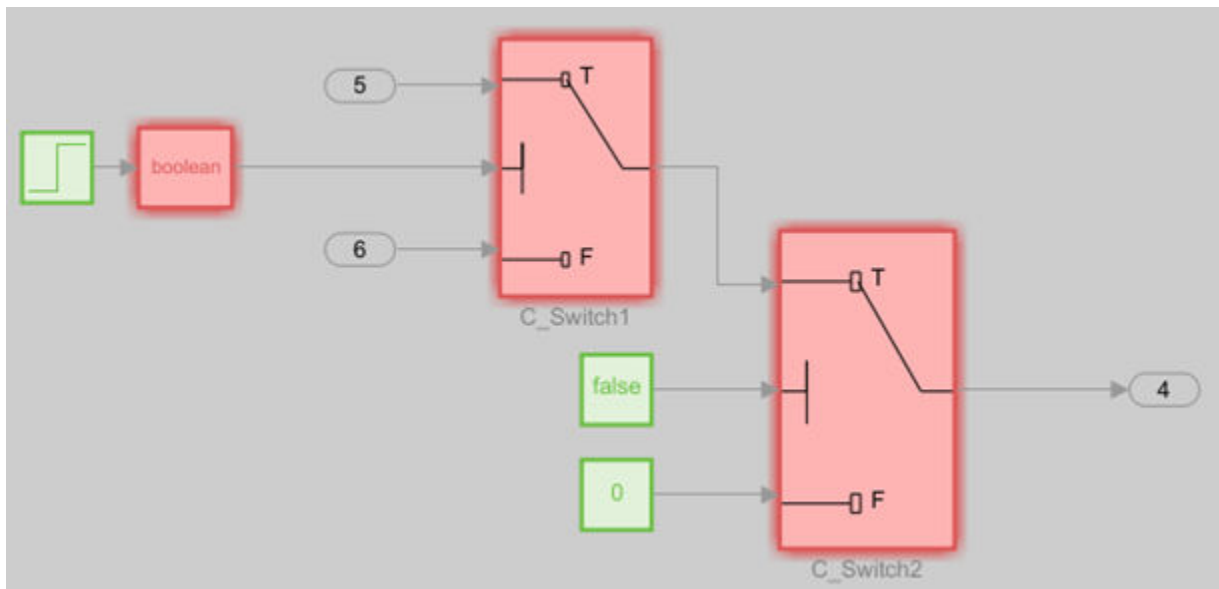
C_Switch1

C_Switch2

A Constant block set to `false` connected to the control input on C_Switch2 causes the `true` case of C_Switch2 to not occur. **Conditional input branch execution** optimizes C_Switch1 out as a result. Simulink Coverage reports 0% coverage on C_Switch1.

Because the C_Switch1 block is dead logic, the coverage report generates a Blocks Eliminated from Coverage Analysis section.

Blocks Eliminated from Coverage Analysis

# Model Object	Rationale
condInputBranchOptim_Example/C_Switch1	It might not be executed because of Conditional input branch optimization
condInputBranchOptim_Example/Data Type Conversion2	It might not be executed because of Conditional input branch optimization

Case 2: Upstream Switch Block Partially Optimized Out

A Step block converted to the boolean data type outputs `false` and `true` before and after the Step time, respectively.
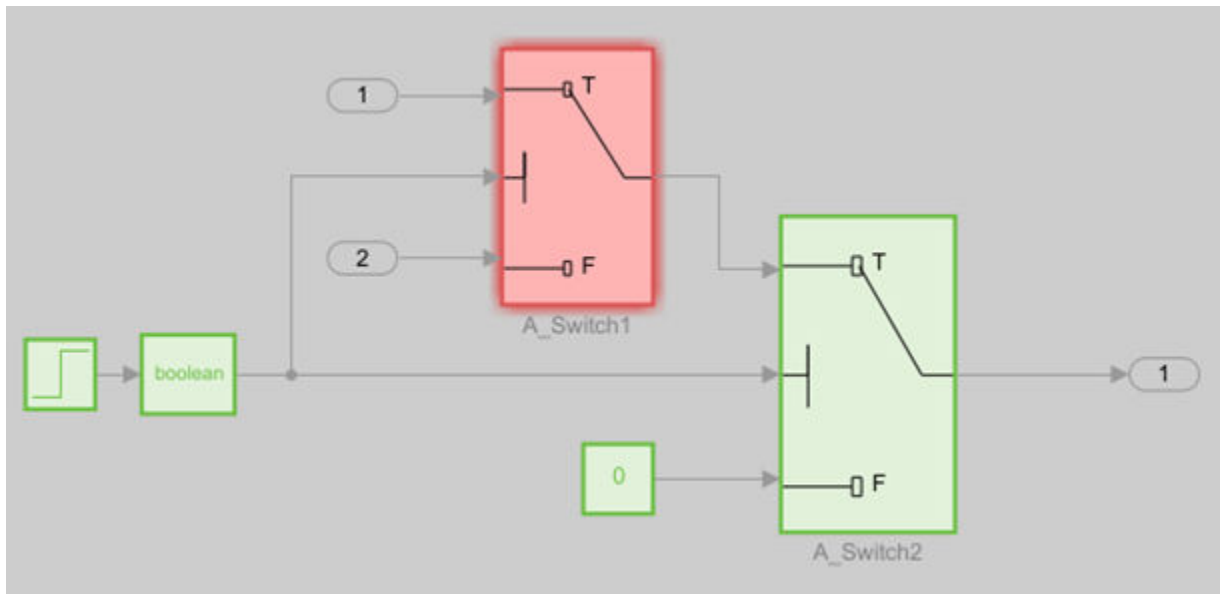
Disabling **Conditional input branch execution** provides full coverage. Enabling **Conditional input branch execution** provides partial coverage on A_Switch1 because A_Switch1 does not see a false case at the same time that A_Switch2 sees a true case. In other words, either both Switch blocks are true, or both are false. The false case of A_Switch1 does not affect the model. The coverage report correctly reports 50% coverage on A_Swtch1.

Address Incomplete Coverage

You can address incomplete coverage in models where the **Conditional input branch execution** parameter is selected by:

- Revising the model design. Incomplete coverage due to **Conditional input branch execution** could indicate a model design flaw.

- Justifying the missing coverage if the inaccessible logic in the model is intentional.

- Providing a more robust test case that can access all of the switch decisions.

- Clearing **Conditional input branch execution**. This eliminates the issue of incomplete Switch coverage, but does not address the inaccessible logic.

For usage details, see "Conditional input branch execution".

Limitations

Conditional input branch execution does not apply to Stateflow charts.

Model Objects That Receive Model Coverage

Model Objects That Receive Coverage

Certain Simulink objects can receive any type of model coverage. Other Simulink objects can receive only certain types of coverage, as the following table shows. Click a link in the first column to get more detailed information about coverage for specific model objects.

All Simulink objects can receive Execution coverage, except blocks that are not instrumented in model coverage:

- Merge Blocks
- Scope Blocks
- Outport Blocks
- Inport Blocks
- Width Blocks
- Display Blocks

For Stateflow states, events, and state temporal logic decisions, model coverage provides decision coverage. For Stateflow transitions, model coverage provides decision, condition, and MCDC coverage. Model coverage provides condition and MCDC coverage for logical expressions in assignment statements in states and transitions. For more information, see "Model Coverage for Stateflow Charts" on page 5-68.

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier	Saturate on Integer Overflow	Relational Boundary
"Abs" on page 2-6	●					●	●
"Bias" on page 2-6						●	
"Combinatorial Logic" on page 2-7	●	●					
"Compare to Constant" on page 2-7		●					●
"Compare to Zero" on page 2-7		●					●
"Data Type Conversion" on page 2-8						●	
"Dead Zone" on page 2-8	●					●	●
"Direct Lookup Table (n-D)" on page 2-8				●			
"Discrete Filter" on page 2-9						●	

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier	Saturate on Integer Overflow	Relational Boundary
"Discrete FIR Filter" on page 2-9						●	
"Discrete-Time Integrator" on page 2-9 (when saturation limits are enabled or reset)	●					●	
"Discrete Transfer Fcn" on page 2-10						●	
"Dot Product" on page 2-10						●	
"Enabled Subsystem" on page 2-10	●	●	●				
"Enabled and Triggered Subsystem" on page 2-11	●	●	●				
"Fcn" on page 2-11		●	●				●
"For Iterator, For Iterator Subsystem" on page 2-12	●						
"Gain" on page 2-12						●	
"If, If Action Subsystem" on page 2-12	●	●	●				●
"Index Vector" on page 2-13	●					●	
"Interpolation Using Prelookup" on page 2-13				●		●	
"Library-Linked Objects" on page 2-13	●	●	●	●	●		
"Logical Operator" on page 2-14		●	●				
"1-D Lookup Table" on page 2-14				●		●	

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier	Saturate on Integer Overflow	Relational Boundary
"2-D Lookup Table" on page 2-15				●		●	
"n-D Lookup Table" on page 2-15				●		●	
"Math Function" on page 2-15						●	
"MATLAB Function" on page 2-16	●	●	●				●
"MATLAB System" on page 2-16	●	●	●				
"MinMax" on page 2-16	●					●	
"Model" on page 2-16 See also "Triggered Models" on page 2-23.	●	●	●	●	●	●	●
"Multiport Switch" on page 2-17	●					●	
"Observer Model" on page 2-17	●	●	●	●	●	●	●
"PID Controller, PID Controller (2 DOF)" on page 2-17						●	
"Product" on page 2-17						●	
"Proof Assumption" on page 2-18					●		
"Proof Objective" on page 2-18					●		
"Rate Limiter" on page 2-18	● (Relative to slew rates)						●
"Relational Operator" on page 2-19		●					●
"Relay" on page 2-19	●						●

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier	Saturate on Integer Overflow	Relational Boundary
"C/C++ S-Function" on page 2-20	●	●	●				
"Saturation" on page 2-20	●						●
"Saturation Dynamic" on page 2-21						●	
"Simulink Design Verifier Functions in MATLAB Function Blocks" on page 2-21					●		
Stateflow charts on page 5-68	●	●	●				●
Stateflow state transition tables on page 5-80	●	●	●				●
"Sqrt, Signed Sqrt, Reciprocal Sqrt" on page 2-22						●	
"Sum, Add, Subtract, Sum of Elements" on page 2-22						●	
"Switch" on page 2-22	●					●	●
"SwitchCase, SwitchCase Action Subsystem" on page 2-22	●						
"Test Condition" on page 2-23					●		
"Test Objective" on page 2-23					●		
"Triggered Models" on page 2-23	●	●	●				
"Triggered Subsystem" on page 2-24	●	●	●				
"Truth Table" on page 2-24	●	●	●				

Model Object	Decision	Condition	MCDC	Lookup Table	Simulink Design Verifier	Saturate on Integer Overflow	Relational Boundary
"Unary Minus" on page 2-24						●	
Variant Subsystem, Variant Model, Variant Source, Variant Sink on page 5-89							
"Weighted Sample Time Math" on page 2-24						●	
"While Iterator, While Iterator Subsystem" on page 2-25	●						

Abs

The Abs block receives decision coverage. Decision coverage is based on:

- Input to the block being less than zero.
- Data type of the input signal.

For input to the block being less than zero, the decision coverage measures:

- The number of time steps that the block input is less than zero, indicating a true decision.
- The number of time steps the block input is not less than zero, indicating a false decision.

If you select the **Saturate on integer overflow** coverage metric, the Abs block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7.

If the input data type to the Abs block is `uint8`, `uint16`, or `uint32`, the Simulink Coverage software reports no coverage for the block. The software sets the block output equal to the block input without making a decision. If the input data type to the Abs block is Boolean, an error occurs.

The Abs block contains an implicit comparison of the input with zero. Therefore, if you select the **Relational Boundary** coverage metric, the Abs block receives relational boundary coverage. For more information, see "Relational Boundary Coverage" on page 1-7.

Bias

If you select the **Saturate on integer overflow** coverage metric, the Bias block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7.

Combinatorial Logic

The Combinatorial Logic block receives decision and condition coverage. Decision coverage is based on achieving each output row of the truth table. The decision coverage measures the number of time steps that each output row of the truth table is set to the block output.

The condition coverage measures the number of time steps that each input is false (equal to zero) and the number of times each input is true (not equal to zero). If the Combinatorial Logic block has a single input element, the Simulink Coverage software reports only decision coverage, because decision and condition coverage are equivalent.

If all truth table values are set to the block output for at least one time step, decision coverage is 100%. Otherwise, the software reports the coverage as the number of truth table values output during at least one time step, divided by the total number of truth table values. Because this block always has at least one value in the truth table as output, the minimum coverage reported is one divided by the total number of truth table values.

If all block inputs are false for at least one time step and true for at least one time step, condition coverage is 100%. Otherwise, the software reports the coverage as achieving a false value at each input for at least one time step, plus achieving a true value for at least one time step, divided by two raised to the power of the total number of inputs (i.e., $2\char`^number\_of\_inputs$). The minimum coverage reported is the total number of inputs divided by two raised to the power of the total number of inputs.

Compare to Constant

The Compare to Constant block receives condition coverage.

Condition coverage measures:

- the number of times that the comparison between the input and the specified constant was true.
- the number of times that the comparison between the input and the specified constant was false.

The Compare to Constant block contains a comparison of the input with a constant. Therefore, if you select the **Relational Boundary** coverage metric, the Compare to Constant block receives relational boundary coverage. For more information, see "Relational Boundary Coverage" on page 1-7.

Compare to Zero

The Compare to Zero block receives condition coverage.

Condition coverage measures:

- the number of times that the comparison between the input and zero was true.
- the number of times that the comparison between the input and zero was false.

The Compare to Zero block contains a comparison of the input with zero. Therefore, if you select the **Relational Boundary** coverage metric, the Compare to Zero block receives relational boundary coverage. For more information, see "Relational Boundary Coverage" on page 1-7.

Data Type Conversion

If you select the **Saturate on integer overflow** coverage metric, the Data Type Conversion block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7.

Dead Zone

The Dead Zone block receives decision coverage. The Simulink Coverage software reports decision coverage for these parameters:

- **Start of dead zone**
- **End of dead zone**

The **Start of dead zone** parameter specifies the lower limit of the dead zone. For the **Start of dead zone** parameter, decision coverage measures:

- The number of time steps that the block input is greater than or equal to the lower limit, indicating a true decision.
- The number of time steps that the block input is less than the lower limit, indicating a false decision.

The **End of dead zone** parameter specifies the upper limit of the dead zone. For the **End of dead zone**, decision coverage measures:

- The number of time steps that the block input is greater than the upper limit, indicating a true decision.
- The number of time steps that the block input is less than or equal to the upper limit, indicating a false decision.

When the upper limit is true, the software does not measure **Start of dead zone** coverage for that time step. Therefore, the total number of **Start of dead zone** decisions equals the number of time steps that the **End of dead zone** is false.

If you select the **Saturate on integer overflow** coverage metric, the Dead Zone block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7.

The Dead Zone block contains an implicit comparison of the input with an upper and lower limit value. Therefore, if you select the **Relational Boundary** coverage metric, the Dead Zone block receives relational boundary coverage. For more information, see "Relational Boundary Coverage" on page 1-7.

Direct Lookup Table (n-D)

The Direct Lookup Table (n-D) block receives lookup table coverage. For an *n*-dimensional lookup table, the number of output break points is the product of all the number of break points for each table dimension.

Lookup table coverage measures:

- The number of times during simulation that each combination of dimension input values is between each of the break points.

- The number of times during simulation that each combination of dimension input values is below the lowest break point and above the highest break point for each table dimension.

The total number of coverage points for an *n*-dimensional lookup table is the product of the number of break points in each table dimension plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

The software determines a percentage of total coverage by measuring the total interpolation and extrapolation points that achieve a measurement of at least one time step during simulation between a break point or beyond the end points.

Discrete Filter

If you select the **Saturate on integer overflow** coverage metric, the Discrete Filter block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7.

Discrete FIR Filter

If you select the **Saturate on integer overflow** coverage metric, the Discrete FIR Filter block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7.

Discrete-Time Integrator

The Discrete-Time Integrator block receives decision coverage. The Simulink Coverage software reports decision coverage for these parameters:

- **External reset**
- **Limit output**

If you set **External reset** to none, the Simulink Coverage software does not report decision coverage for the reset decision. Otherwise, the decision coverage measures:

- The number of time steps that the block output is reset, indicating a true decision.
- The number of time steps that the block output is not reset, indicating a false decision.

If you do not select **Limit output**, the software does not report decision coverage for that decision. Otherwise, the software reports decision coverage for the **Lower saturation limit** and the **Upper saturation limit**.

For the **Upper saturation limit**, decision coverage measures:

- The number of time steps that the integration result is greater than or equal to the upper limit, indicating a true decision.
- The number of time steps that the integration result is less than the upper limit, indicating a false decision.

For the **Lower saturation limit**, decision coverage measures

- The number of time steps that the integration result is less than or equal to the lower limit, indicating a true decision.
- The number of time steps that the integration result is greater than the lower limit, indicating a false decision.

For a time step when the upper limit is true, the software does not measure **Lower saturation limit** coverage. Therefore, the total number of lower limit decisions equals the number of time steps that the upper limit is false.

If you select the **Saturate on integer overflow** coverage metric, the Discrete-Time Integrator block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7.

Discrete Transfer Fcn

If you select the **Saturate on integer overflow** coverage metric, the Discrete Transfer Fcn block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7.

Dot Product

If you select the **Saturate on integer overflow** coverage metric, the Dot Product block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7.

Enabled Subsystem

The Enabled Subsystem block receives decision, condition, and MCDC coverage.

Decision coverage measures:

- The number of time steps that the block is enabled, indicating a true decision.
- The number of time steps that the block is disabled, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%.

The Simulink Coverage software measures condition coverage for the enable input only if the enable input is a vector. For the enable input, condition coverage measures the number of time steps each element of the enable input is true and the number of time steps each element of the enable input is false. The software reports condition coverage based on the total number of possible conditions and how many are true for at least one time step and how many are false for at least one time step.

The software measures MCDC coverage for the enable input only if the enable input is a vector. Because the enable of the subsystem is an OR of the vector inputs, MCDC coverage is 100% if, during at least one time step, each vector enable input is exclusively true and if, during at least one time step, all vector enable inputs are false. For MCDC coverage measurement, the software treats each element of the vector as a separate condition.

Enabled and Triggered Subsystem

The Enabled and Triggered Subsystem block receives decision, condition, and MCDC coverage. Decision coverage measures:

- The number of time steps that a trigger edge occurs while the block is enabled, indicating a true decision.
- The number of time steps that a trigger edge does not occur while the block is enabled, or the block is disabled, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%.

The software measures condition coverage for the enable input and for the trigger input separately:

- For the enable input, condition coverage measures the number of time steps the enable input is true and the number of time steps the enable input is false.
- For the trigger input, condition coverage measures the number of time steps the trigger edge occurs, indicating true, and the number of time steps the trigger edge does not occur, indicating false.

The software reports condition coverage based on the total number of possible conditions and how many conditions are true for at least one time step and how many are false for at least one time step. The software treats each element of a vector as a separate condition coverage measurement.

The software measures MCDC coverage for the enable input and for the trigger input in combination. Because the enable input of the subsystem is an AND of these two inputs, MCDC coverage is 100% if all of the following occur:

- During at least one time step, both inputs are true.
- During at least one time step, the enable input is true and the trigger edge is false.
- During one time step, the enable input is false and the trigger edge is true.

The software treats each vector element as a separate MCDC coverage measurement. It measures each trigger edge element against each enable input element. However, if the number of elements in both the trigger and enable inputs exceeds 12, the software does not report MCDC coverage.

Fcn

The Fcn block receives condition and MCDC coverage. The Simulink Coverage software reports condition or MCDC coverage for Fcn blocks only if the top-level operator is Boolean (&&, ||, or !).

Condition coverage is based on input values or arithmetic expressions that are inputs to Boolean operators in the block. The condition coverage measures:

- The number of time steps that each input to a Boolean operator is true (not equal to zero).
- The number of time steps that each input to a Boolean operator is false (equal to zero).

If all Boolean operator inputs are false for at least one time step and true for at least one time step, condition coverage is 100%. Otherwise, the software reports condition coverage based on the total number of possible conditions and how many are true for at least one time step and how many are false for at least one time step.

The software measures MCDC coverage for Boolean expressions within the Fcn block. If, during at least one time step, each condition independently sets the output of the expression to true and if, during at least one time step, each condition independently sets the output of the expression to false, MCDC coverage is 100%. Otherwise, the software reports MCDC coverage based on the total number of possible conditions and how many times each condition independently sets the output to true during at least one time step and how many conditions independently set the output to false during at least one time step.

If the Fcn block contains a relational operation and you select the **Relational Boundary** coverage metric, the Fcn block receives relational boundary coverage. For more information, see "Relational Boundary Coverage" on page 1-7.

For Iterator, For Iterator Subsystem

The For Iterator block and For Iterator Subsystem receive decision coverage. The Simulink Coverage software measures decision coverage for the loop condition value, which is determined by one of the following:

- The iteration value being at or below the iteration limit, indicated as true.
- The iteration value being above the iteration limit, indicated as false.

The software reports the total number of times that each loop condition evaluates to true and to false. If the loop condition evaluates to true at least once and false at least once, decision coverage is 100%. If no loop conditions are true, or if no loop conditions are false, decision coverage is 50%.

Gain

If you select the **Saturate on integer overflow** coverage metric, the Gain block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7.

If, If Action Subsystem

The If block that causes an If Action Subsystem to execute receives condition, decision, and MCDC coverage:

- The software measures decision coverage for the `if` condition and all `elseif` conditions defined in the If block.
- If the `if` condition or any of the `elseif` conditions contains a logical expression with multiple conditions, such as `u1 & u2 & u3`, the software also measures condition and MCDC coverage for each condition in the expression, `u1`, `u2`, and `u3` in the preceding example.

The software does not directly measure the `else` condition. When there are no `elseif` conditions, the `else` condition is the direct complement of the `if` condition, or the `else` condition is the direct complement of the last `elseif` condition.

The software reports the total number of time steps that each `if` and `elseif` condition evaluates to true and to false. If the `if` or `elseif` condition evaluates to true at least once, and evaluates to false at least once, decision coverage is 100%. If no `if` or `elseif` conditions are true, or if no `if` or `elseif` conditions are false, decision coverage is 50%. If the previous `if` or `elseif` condition never evaluates as false, an `elseif` condition can have 0% decision coverage.

The If block contains a comparison between its inputs. Therefore, if you select the **Relational Boundary** coverage metric, the If block receives relational boundary coverage. For more information, see "Relational Boundary Coverage" on page 1-7.

Index Vector

The Index Vector block receives decision coverage based on passing each element of the vector signal input to the output of the block.

If each vector index is passed to the block output for at least one time step, decision coverage is 100%. Otherwise, Simulink Coverage reports coverage as the percentage of the total number of vector indices in the input signal that passed through to the output.

If you select the **Saturate on integer overflow** coverage metric, the Index Vector block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7.

Interpolation Using Prelookup

The Interpolation Using Prelookup block receives lookup table coverage. For an n-D lookup table, the number of output break points equals the product of all the number of break points for each table dimension. The lookup table coverage measures:

- The number of times during simulation that each combination of dimension input values is between each of the break points.
- The number of times during simulation that each combination of dimension input values is below the lowest break point and above the highest break point for each table dimension.

The total number of coverage points for an n-dimensional lookup table is the product of the number of break points in each table dimension plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

The software determines a percentage of total coverage by measuring the total interpolation and extrapolation points that achieve a measurement of at least one time step during simulation between a break point or beyond the end points.

If you select the **Saturate on integer overflow**, the Interpolation Using Prelookup block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

Library-Linked Objects

Simulink blocks and Stateflow charts that are linked to library objects receive the same coverage that they would receive if they were not linked to library objects. The Simulink Coverage software records coverage individually for each library object in the model. If your model contains multiple instances of the same library object, each instance receives its own coverage data.

Logical Operator

The Logical Operator block receives condition and MCDC coverage. The Simulink Coverage measures condition coverage for each input to the block. The condition coverage measures:

- The number of time steps that each input is true (not equal to zero).
- The number of time steps that each input is false (equal to zero).

If all block inputs are false for at least one time step and true for at least one time step, the condition coverage is 100%. Otherwise, the condition coverage is based on the total number of possible conditions and how many are true at least one time step and how many are false at least one time step.

Simulink Coverage measures MCDC coverage for all inputs to the block. If, during at least one time step, each condition independently sets the output of the block to true and if, during at least one time step, each condition independently sets the output of the block to false, MCDC coverage is 100%. Otherwise, Simulink Coverage reports the MCDC coverage based on the total number of possible conditions and how many times each one of them independently set the output to true for at least one time step and how many independently set the output to false for at least one time step.

For a Logical Operator block to receive MCDC as part of a logic cascade, all of the following conditions must be met:

- Block input and output signals are all scalar
- Logic block operation is not XOR/NXOR
- Logic block has more than one input signal (unless its operation is NOT)

For more information about logical cascades, see "Logical Operator Cascade Patterns" on page 5-9.

1-D Lookup Table

The 1-D Lookup Table block receives lookup table coverage; for a one-dimensional lookup table, the number of input and output break points is equal. Lookup table coverage measures:

- The number of times during simulation that the input and output values are between each of the break points.
- The number of times during simulation that the input and output values are below the lowest break point and above the highest break point.

The total number of coverage points for a one-dimensional lookup table is the number of break points in the table plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

The software determines a percentage of total coverage by measuring the total interpolation and extrapolation points that achieve a measurement of at least one time step during simulation between a break point or beyond the end points.

If you select the **Saturate on integer overflow** coverage metric, the 1-D Lookup Table block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

2-D Lookup Table

The 2-D Lookup Table block receives lookup table coverage. For a two-dimensional lookup table, the number of output break points equals the number of row break points multiplied by the number of column inputs. Lookup table coverage measures:

- The number of times during simulation that each combination of row input and column input values is between each of the break points.
- The number of times during simulation that each combination of row input and column input values is below the lowest break point and above the highest break point for each row and column.

The total number of coverage points for a two-dimensional lookup table is the number of row break points in the table plus one, multiplied by the number of column break points in the table plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

If you select the **Saturate on integer overflow** coverage metric, the 2-D Lookup Table block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

n-D Lookup Table

The n-D Lookup Table block receives lookup table coverage. For an *n*-dimensional lookup table, the number of output break points equals the product of all the number of break points for each table dimension. Lookup table coverage measures:

- The number of times during simulation that each combination of dimension input values is between each of the break points.
- The number of times during simulation that each combination of dimension output values is below the lowest break point and above the highest break point for each table dimension.

The total number of coverage points for an *n*-dimensional lookup table is the product of the number of break points in each table dimension plus one. In the coverage report, an increasing white-to-green color scale, with six evenly spaced data ranges starting with zero, indicates the number of time steps that the software measures each interpolation or extrapolation point.

The software determines a percentage of total coverage by measuring the total interpolation and extrapolation points that achieve a measurement of at least one time step during simulation between a break point or beyond the end points.

If you select the **Saturate on integer overflow** coverage metric, the n-D Lookup Table block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

Math Function

If you select the **Saturate on integer overflow** coverage metric, the Math Function block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow

Coverage" on page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

MATLAB Function

For information about the type of coverage that the Simulink Coverage software reports for the MATLAB Function block, see "Model Coverage for MATLAB Functions" on page 5-45.

MATLAB System

Simulink Coverage records only Decision, Condition, and MCDC coverage for MATLAB System blocks.

MinMax

The MinMax block receives decision coverage based on passing each input to the output of the block.

For decision coverage based on passing each input to the output of the block, the coverage measures the number of time steps that the simulation passes each input to the block output. The number of decision points is based on the number of inputs to the block and whether they are scalar, vector, or matrix.

If all inputs are passed to the block output for at least one time step, the Simulink Coverage software reports the decision coverage as 100%. Otherwise, the software reports the coverage as the number of inputs passed to the output during at least one time step, divided by the total number of inputs.

If you select the **Saturate on integer overflow** coverage metric, the MinMax block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

Model

The Model block does not receive coverage directly; the model that the block references receives coverage. If the simulation mode for the referenced model is set to `Normal`, the Simulink Coverage software reports coverage for all objects within the referenced model that receive coverage. . If the simulation mode for the referenced model is set to `SIL` or `PIL` and you have Embedded Coder installed, the Simulink Coverage software reports coverage for the code generated from your model .If the simulation mode is set to a value other than `Normal`, `SIL`, or `PIL`, the software cannot measure coverage for the referenced model.

In the **Coverage** pane of the Configuration Parameters dialog box, select the referenced models for which you want to report coverage. The software generates a coverage report for each referenced model you select.

If your model contains multiple instances of the same referenced model, the software records coverage for all instances of that model where the simulation mode of the Model block is set to `Normal`. The coverage report for that referenced model combines the coverage data for all Normal mode instances of that model.

The coverage reports for all analyzed models in a model reference hierarchy are linked from a summary report.

Note For details on how to select referenced models to report coverage, see "Referenced Models" on page 3-3.

Multiport Switch

The Multiport Switch block receives decision coverage based on passing each input, excluding the first control input, to the output of the block.

For decision coverage based on passing each input, excluding the first control input, to the output of the block, the coverage measures the number of time steps that each input is passed to the block output. The number of decision points is based on the number of inputs to the block and whether the control input is scalar or vector.

If all inputs, excluding the first control input, are passed to the block output for at least one time step, decision coverage is 100%. Otherwise, the Simulink Coverage software reports coverage as the number of inputs passed to the output during at least one time step, divided by the total number of inputs minus one.

If you select the **Saturate on integer overflow** coverage metric, the Multiport Switch block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

Observer Model

The Observer Reference block does not receive coverage directly; the Observer model that the block references receives coverage metrics for the blocks inside that model. Only Observers models in `Normal` mode are analyzed for coverage.

You can select Observer models for coverage the same way you select referenced models. For more information about selecting models for analysis, see "Referenced Models" on page 3-3.

Only Observer models that you reference from the top model are active during a simulation and can receive coverage. Terminate Function blocks located inside Observer models do not receive coverage.

The coverage results for each Observer model are captured in separate `cvdata` objects. Each model referenced from an Observer model is considered an Observer model and has its own `cvdata` object. If you record coverage for multiple models in a model reference hierarchy, the results are collected in a `cv.cvdatagroup` object. The summary report links to the coverage reports for all analyzed models in the hierarchy.

PID Controller, PID Controller (2 DOF)

If you select the **Saturate on integer overflow** coverage metric, the PID Controller and PID Controller (2 DOF) blocks receive saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

Product

If you select the **Saturate on integer overflow** coverage metric, the Product block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on

page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

Proof Assumption

The Proof Assumption block receives Simulink Design Verifier coverage. Simulink Design Verifier coverage is based on the points and intervals defined in the block dialog box. Simulink Design Verifier coverage measures the number of time steps that each point or interval defined in the block is satisfied. The total number of objective outcomes is based on the number of points or intervals defined in the Proof Assumption block.

If all points and intervals defined in the block are satisfied for at least one time step, Simulink Design Verifier coverage is 100%. Otherwise, the Simulink Coverage software reports coverage as the number of points and intervals satisfied during at least one time step, divided by the total number of points and intervals defined for the block.

Proof Objective

The Proof Objective block receives Simulink Design Verifier coverage. Simulink Design Verifier coverage is based on the points and intervals defined in the block dialog box. Simulink Design Verifier coverage measures the number of time steps that each point or interval defined in the block is satisfied. The total number of objective outcomes is based on the number of points or intervals defined in the Proof Objective block.

If all points and intervals defined in the block are satisfied for at least one time step, Simulink Design Verifier coverage is 100%. Otherwise, the Simulink Coverage software reports coverage as the number of points and intervals satisfied during at least one time step, divided by the total number of points and intervals defined for the block.

Rate Limiter

The Rate Limiter block receives decision coverage. The Simulink Coverage software reports decision coverage for the **Rising slew rate** and **Falling slew rate** parameters.

For the **Rising slew rate**, decision coverage measures:

- The number of time steps that the block input changes more than or equal to the rising rate, indicating a true decision.
- The number of time steps that the block input changes less than the rising rate, indicating a false decision.

For the **Falling slew rate**, decision coverage measures:

- The number of time steps that the block input changes less than or equal to the falling rate, indicating a true decision.
- The number of time steps that the block input changes more than the falling rate, indicating a false decision.

The software does not measure **Falling slew rate** coverage for a time step when the **Rising slew rate** is true. Therefore, the total number of **Falling slew rate** decisions equals the number of time steps that the **Rising slew rate** is false.

If at least one time step is true and at least one time step is false, decision coverage for each of the two individual decisions for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

The Rate Limiter block implicitly compares the derivative of the input signal with an upper and lower limit value. Therefore, if you select the **Relational Boundary** coverage metric, the Rate Limiter block receives relational boundary coverage. For more information, see "Relational Boundary Coverage" on page 1-7.

Relational Operator

The Relational Operator block receives condition coverage.

Condition coverage measures:

- the number of times that the specified relational operation was true.
- the number of times that the specified relational operation was false.

The Relational Operator block contains a comparison between its inputs. Therefore, if you select the **Relational Boundary** coverage metric, the Relational Operator block receives relational boundary coverage. For more information, see "Relational Boundary Coverage" on page 1-7.

Relay

The Relay block receives decision coverage. The Simulink Coverage software reports decision coverage for the **Switch on point** and the **Switch off point** parameters.

For the **Switch on point**, decision coverage measures:

- The number of consecutive time steps that the block input is greater than or equal to the **Switch on point**, indicating a true decision.
- The number of consecutive time steps that the block input is less than the **Switch on point**, indicating a false decision.

For the **Switch off point**, decision coverage measures:

- The number of consecutive time steps that the block input is less than or equal to the **Switch off point**, indicating a true decision.
- The number of consecutive time steps that the block input is greater than the **Switch off point**, indicating a false decision.

The software does not measure **Switch off point** coverage for a time step when the switch on threshold is true. Therefore, the total number of **Switch off point** decisions equals the number of time steps that the **Switch on point** is false.

If at least one time step is true and at least one time step is false, decision coverage for each of the two individual decisions for the block is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%. The software treats each element of a vector or matrix as a separate coverage measurement.

The Relay block contains an implicit comparison of its second input with a threshold value. Therefore, if you select the **Relational Boundary** coverage metric, the Relay block receives relational boundary coverage. For more information, see "Relational Boundary Coverage" on page 1-7.

C/C++ S-Function

Model coverage is supported for C/C++ S-Functions. The coverage report for the model contains results for each instance of an S-Function block in the model. The results for an S-Function block link to a separate coverage report for the C/C++ code in the block.

To generate coverage report for S-Functions:

1 When creating the S-Functions, enable support for coverage. For more information, see "Make S-Function Compatible with Model Coverage" on page 5-59.

2 When generating the coverage report, enable support for S-Functions. For more information, see "Generate Coverage Report for S-Function" on page 5-60.

The following coverage types are reported for S-Functions:

- "Cyclomatic Complexity for Code Coverage" on page 4-4
- "Condition Coverage for Code Coverage" on page 4-2
- "Decision Coverage for Code Coverage" on page 4-3
- "Modified Condition/Decision Coverage (MCDC) for Code Coverage" on page 4-3
- "Relational Boundary for Code Coverage" on page 4-4
- Percentage of statements covered

The coverage data for S-Function blocks is obtained in the following way:

- The coverage result for a block is a weighted average of the result over all files in the block.

  For instance, an S-Function block has two files, `file1.c` and `file2.c`. The decision coverage for `file1.c` is 75% (3/4 outcomes covered) and that for `file2.c` is 50% (10/20 outcomes covered). The decision coverage for the block is $13/24 \approx 54$ %.

- For each file, the coverage result is a weighted average of the result over all functions in the file.
- For each function, the coverage result is a weighted average of the result over all statements in the function that receive that coverage.

Note Model coverage for S-Functions have the following restrictions:

- Only Level-2 C/C++ S-Functions are supported for coverage. For an example of a level-2 C S-Function, see "Create a Basic C MEX S-Function".
- C++ class templates are not instrumented for coverage.

Saturation

Simulink Coverage reports decision coverage for the **Upper limit** and **Lower limit** parameters of the Saturation block and relational boundary coverage for the Saturation block.

For the **Upper limit**, decision coverage measures:

- The number of time steps that the block input is greater than the upper limit, which indicates a true decision outcome and that the block output saturates at the upper limit.
- The number of time steps that the block input is less than or equal to the upper limit, which indicates a false decision outcome.

For the **Lower limit**, decision coverage measures:

- The number of time steps that the block input is greater than or equal to the lower limit, which indicates a true decision outcome.
- The number of time steps that the block input is less than the lower limit, which indicates a false decision outcome and that the block output saturates at the lower limit.

Simulink Coverage does not measure coverage for the **Lower limit** decision at a time step when the upper limit is true. Therefore, the total number of **Lower limit** decision evaluations is equal to the number of time steps that the **Upper limit** is false. Simulink Coverage reports 0% coverage for the **Lower limit** decision if the **Upper limit** decision is true at every time step.

For each decision, Simulink Coverage reports 100% coverage if the decision returns a true outcome for at least one time step and a false outcome for at least one time step. If a decision has no true outcomes or no false outcomes, Simulink Coverage reports 50% coverage for that decision. The decision coverage for the Saturation block is the sum of the coverage for the **Upper limit** and **Lower limit** decisions. For example, if a Saturation block receives 100% coverage for the **Upper limit** decision, and 50% coverage for the **Lower limit** decision, then the Saturation block receives a total of 75% decision coverage, with 3 out of 4 decision outcomes satisfied.

If the input signal to the Saturation block is a vector or matrix, each element is a separate coverage measurement. For example, an input signal that is a two element vector results in a total of eight decision outcomes within the Saturation block instead of four.

The Saturation block contains an implicit comparison of the input with an upper and lower limit value. Therefore, if you select the **Relational boundary** coverage metric in the **Coverage** pane of the Configuration Parameters window, the Saturation block receives relational boundary coverage. For more information, see "Relational Boundary Coverage" on page 1-7.

Saturation Dynamic

If you select the **Saturate on integer overflow** coverage metric, the Saturation Dynamic block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

Simulink Design Verifier Functions in MATLAB Function Blocks

The following functions in MATLAB Function blocks receive Simulink Design Verifier coverage:

- `sldv.condition`
- `sldv.test`
- `sldv.assume`
- `sldv.prove`

Each of these functions evaluates an expression *expr*, for example, `sldv.test(`*expr*`)`, where *expr* is any valid Boolean MATLAB expression. Simulink Design Verifier coverage measures the number of time steps that the expression *expr* evaluates to true.

If *expr* is true for at least one time step, Simulink Design Verifier coverage for that function is 100%. Otherwise, the Simulink Coverage software reports coverage for that function as 0%.

Sqrt, Signed Sqrt, Reciprocal Sqrt

If you select the **Saturate on integer overflow** coverage metric, the Sqrt, Signed Sqrt, and Reciprocal Sqrt blocks receive saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

Sum, Add, Subtract, Sum of Elements

If you select the **Saturate on integer overflow** coverage metric, the Sum, Add, Subtract, and Sum of Elements blocks receive saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

Switch

The Switch block receives decision coverage based on the control input to the block. Decision coverage measures:

- The number of time steps that the control input evaluates to true.
- The number of time steps the control input evaluates to false.

The number of decision points is based on whether the control input is scalar or vector.

If you select the **Saturate on integer overflow** coverage metric, the Switch block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

The Switch block contains an implicit comparison of its second input with a threshold value. Therefore, if you select the **Relational Boundary** coverage metric, the Switch block receives relational boundary coverage. For more information, see "Relational Boundary Coverage" on page 1-7.

SwitchCase, SwitchCase Action Subsystem

The SwitchCase block and SwitchCase Action Subsystem receive decision coverage. The Simulink Coverage software measures decision coverage individually for each switch case defined in the block and also for the default case. The number of decision outcomes is equal to the number of case conditions plus one for the `default` case, if one is defined.

The software reports the total number of time steps that each case evaluates to true. If each case, including the default case, evaluates to true at least once, decision coverage is 100%. The software determines the decision coverage by the number of cases that evaluate true for at least one time step divided by the total number of cases.

If the SwitchCase block does not contain a `default` case, the software measures decision coverage for the number of time steps in which none of the cases evaluated to true. In the coverage report, this coverage is reported as **implicit-default**.

Test Condition

The Test Condition block receives Simulink Design Verifier coverage. Simulink Design Verifier coverage is based on the points and intervals defined in the block dialog box. Simulink Design Verifier coverage measures the number of time steps that each point or interval defined in the block is satisfied. The total number of objective outcomes is based on the number of points or intervals defined in the Test Condition block.

If all points and intervals defined in the block are satisfied for at least one time step, Simulink Design Verifier coverage is 100%. Otherwise, the Simulink Coverage software reports coverage as the number of points and intervals satisfied during at least one time step, divided by the total number of points and intervals defined for the block.

Test Objective

The Test Objective block receives Simulink Design Verifier coverage. Simulink Design Verifier coverage is based on the points and intervals defined in the block dialog box. Simulink Design Verifier coverage measures the number of time steps that each point or interval defined in the block is satisfied. The total number of objective outcomes is based on the number of points or intervals defined in the Test Objective block.

If all points and intervals defined in the block are satisfied for at least one time step, Simulink Design Verifier coverage is 100%. Otherwise, the Simulink Coverage software reports coverage as the number of points and intervals satisfied during at least one time step, divided by the total number of points and intervals defined for the block.

Triggered Models

A Model block can reference a model that contains edge-based trigger ports at the root level of the model. Triggered models receive decision, condition, and MCDC coverage.

Decision coverage measures:

- The number of time steps that the referenced model is triggered, indicating a true decision.
- The number of time steps that the referenced model is not triggered, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage for the Model block that references the triggered model is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%.

Only if the trigger input is a vector, the Simulink Coverage software measures condition coverage for the trigger port in the referenced model. For the trigger port, condition coverage measures:

- The number of time steps that each element of the trigger port is true.
- The number of time steps that each element of the trigger port is false.

The software reports condition coverage based on the total number of possible conditions and how many are true for at least one time step and how many are false for at least one time step.

If the trigger port is a vector, the software measures MCDC coverage for the trigger port only. Because the trigger port of the referenced model is an OR of the vector inputs, if, during at least one time step, each vector trigger port is exclusively true and if, during at least one time step, all vector trigger port inputs are false, MCDC coverage is 100%. The software treats each element of the vector as a separate condition for MCDC coverage measurement.

Triggered Subsystem

The Triggered Subsystem block receives decision, condition, and MCDC coverage.

Decision coverage measures:

- The number of time steps that the block is triggered, indicating a true decision.
- The number of time steps that the block is not triggered, indicating a false decision.

If at least one time step is true and at least one time step is false, decision coverage is 100%. If no time steps are true, or if no time steps are false, decision coverage is 50%.

The Simulink Coverage software measures condition coverage for the trigger input only if the trigger input is a vector. For the trigger input, condition coverage measures:

- The number of time steps that each element of the trigger edge is true.
- The number of time steps that each element of the trigger edge is false.

The software reports condition coverage based on the total number of possible conditions and how many are true for at least one time step and how many are false for at least one time step.

If the trigger input is a vector, the software measures MCDC coverage for the trigger input only. Because the trigger edge of the subsystem is an OR of the vector inputs, if, during at least one time step, each vector trigger edge input is exclusively true and if, during at least one time step, all vector trigger edge inputs are false, MCDC coverage is 100%. The software treats each element of the vector as a separate condition for MCDC coverage measurement.

Truth Table

The Truth Table block is a Stateflow block that enables you to use truth table logic directly in a Simulink model. The Truth Table block receives condition, decision, and MCDC coverage. For more information on model coverage with Stateflow truth tables, see "Model Coverage for Stateflow Truth Tables" on page 5-82.

Unary Minus

If you select the **Saturate on integer overflow** coverage metric, the Unary Minus block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer Overflow Coverage" on page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

Weighted Sample Time Math

If you select the **Saturate on integer overflow** coverage metric, the Weighted Sample Time Math block receives saturate on integer overflow coverage. For more information, see "Saturate on Integer

Overflow Coverage" on page 1-7. The software treats each element of a vector or matrix as a separate coverage measurement.

While Iterator, While Iterator Subsystem

The While Iterator block and While Iterator Subsystem receive decision coverage. Decision coverage is measured for the `while` condition value, which is determined by the `while` condition being satisfied (true), or the `while` condition not being satisfied (false). Simulink Coverage software reports the total number of times that each `while` condition evaluates to true and to false. If the `while` condition evaluates to true at least once, and false at least once, decision coverage for the `while` condition is 100%. If no `while` conditions are true, or if no `while` conditions are false, decision coverage is 50%.

If the iteration limit is exceeded (true) or is not exceeded (false), the software measures decision coverage independently. If the iteration limit evaluates to true at least once, and false at least once, decision coverage for the iteration limit is 100%. If no iteration limits are true, or if no iteration limits are false, decision coverage is 50%. If you set **Maximum number of iterations** to -1 (no limit), the decision coverage for the iteration limit is true for all iterations and false for zero iterations, and decision coverage is 50%.

Model Objects That Do Not Receive Coverage

The Simulink Coverage software does not record Decision, Condition, or MCDC coverage for blocks that are not listed in "Model Objects That Receive Coverage" on page 2-2.

Note The software only records model coverage when the **Simulation mode** parameter is set to Normal. If you have Embedded Coder installed, the software can measure the coverage of code generated from models in SIL or PIL mode. For more information, see "Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode" on page 4-6.

The following table identifies specific model objects that do not receive coverage in certain conditions.

Model object	Does not receive coverage...
Logical Operator block	When the **Operator** parameter specifies XOR or NXOR and there are more than twelve scalar inputs or more than twelve elements in a vector input.
Model block	When the **Simulation mode** parameter specifies Accelerator. Coverage for Model blocks is the sum of the coverage data for the contents of the referenced model.
Protected model block	Coverage information is not provided for protected model blocks. See also "Model Protection" (Simulink Coder).
Subsystem block	When the **Read/Write Permissions** parameter is set to NoReadOrWrite.
Stateflow chart MATLAB Function block	When debugging/animation is not enabled for the model or object.
Virtual Blocks	Virtual blocks do not receive model coverage. For more information, see "Nonvirtual and Virtual Blocks".

Setting Coverage Options

Specify Coverage Options

Before starting a coverage analysis, you specify several coverage analysis options. On the **Apps** tab, select **Coverage Analyzer**. On the **Coverage** tab, select **Settings**.

Coverage Pane

On the **Coverage** pane in the Configuration Parameters dialog box, set the options for the coverage calculated during simulation.
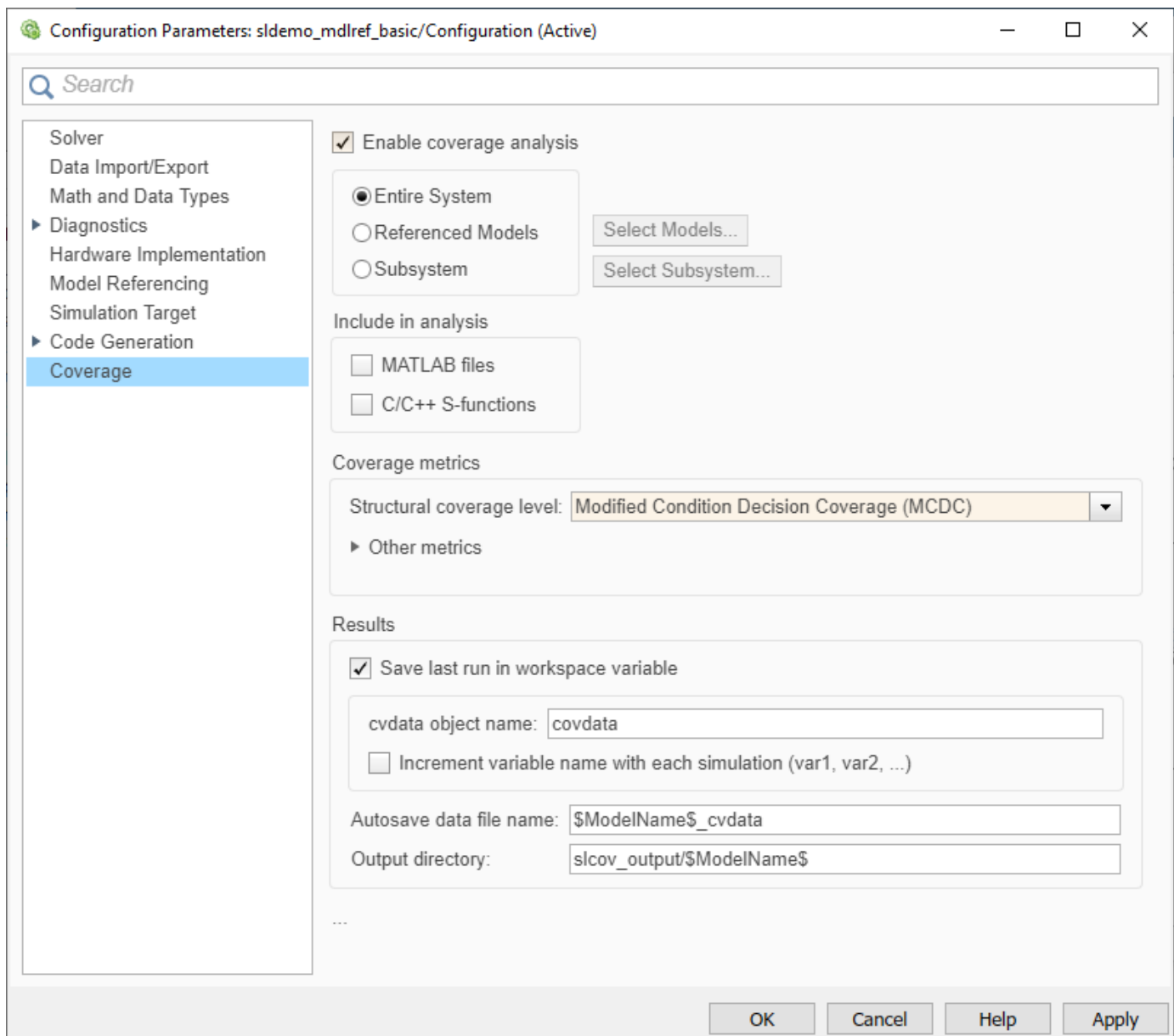
Enable coverage analysis

Gather specified coverage results during simulation and report the coverage. When you select **Enable coverage analysis**, these sections become available:

- "Scope of analysis" on page 3-3
- "Include in analysis" on page 3-5
- "Coverage metrics" on page 3-5

Scope of analysis

Specifies the systems for which the software gathers and reports coverage data. The options are:

- "Entire System" on page 3-3
- "Referenced Models" on page 3-3
- "Subsystem" on page 3-4

You must select **Enable coverage analysis** to specify the scope of analysis.

Entire System

By default, generates coverage data for the entire system. The coverage results include the top-level and all supported subsystems and model references.

Referenced Models

Collect coverage for the referenced models and Observer models that you select. By default this setting collects coverage for all referenced models where the simulation mode of the Model block is `Normal`, `Software-in-the-loop (SIL)`, or `Processor-in-the-loop (PIL)`, and for active Observer models where the simulation mode is `Normal`.

To specify the referenced models and Observer models for which Simulink Coverage collects coverage data:

1. Select **Enable coverage analysis**.
2. For the scope of analysis, select **Referenced Models**.
3. Click **Select Models**.
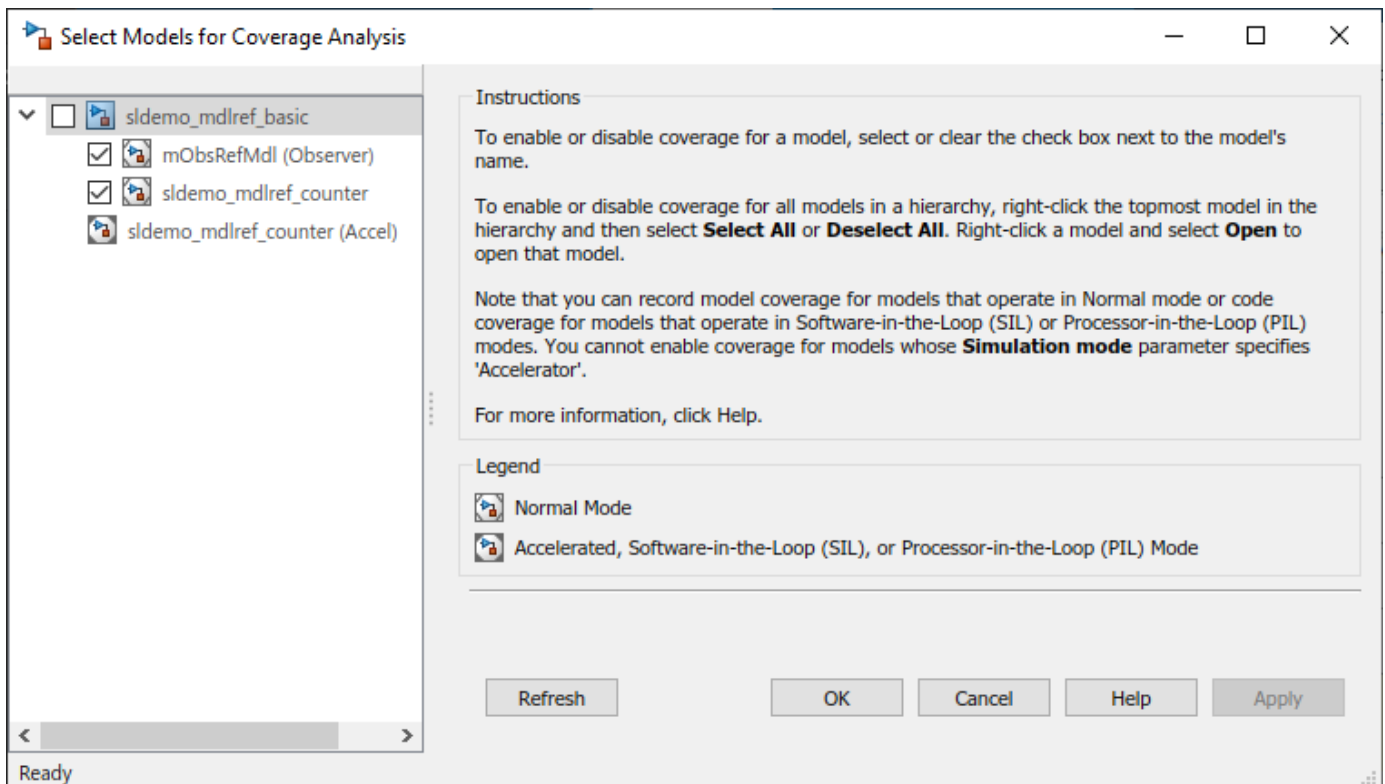
4   In the Select Models for Coverage Analysis dialog box, select the referenced models or Observer models for which you want to collect coverage. You can also select the top-level model.

The icon next to the model name indicates the simulation mode for that referenced model.

If you have multiple Model blocks that reference the same model and whose simulation modes are the same, selecting the check box for that model selects the check boxes for all instances of that model with the same simulation mode.

5   Click **OK**.

Subsystem

Coverage analysis collects coverage during simulation for the subsystem that you select. By default, generates coverage data for the entire model. To restrict coverage reporting to a particular subsystem:

1   In the Configuration Parameters dialog box, on the **Coverage** pane, select **Enable coverage analysis**.

2   Click **Select Subsystem**.
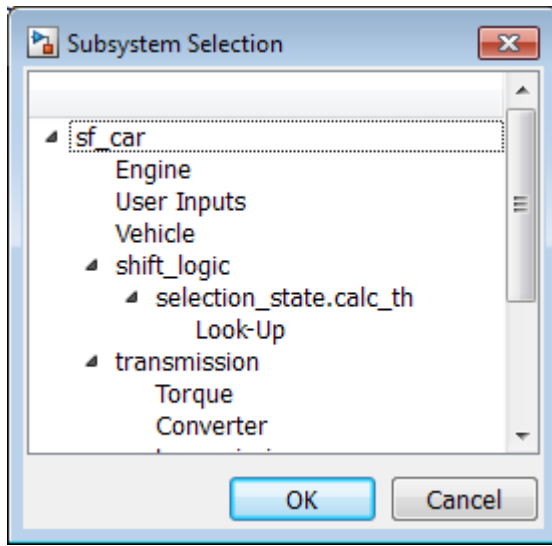
3 In the Subsystem Selection dialog box, select the subsystem for which you want to enable coverage reporting and click **OK**.

Include in analysis

The **Include in analysis** section contains two options:

- **MATLAB files** enables coverage for any external functions called by MATLAB functions in your model. You can define MATLAB functions in MATLAB Function blocks or in Stateflow charts.

  To select the **Coverage for MATLAB files** option, you must select **Enable coverage analysis**.

- **C/C++ S-functions** enables coverage for C/C++ S-Function blocks in your model. Coverage metrics are reported for the S-Function blocks and the C/C++ code in those blocks. For more information, see "Generate Coverage Report for S-Function" on page 5-60.

  You must select **Enable coverage analysis** to select the **Coverage for S-Functions** option.

Coverage metrics

Select the structural coverage level and other types of test case coverage analysis that you want the tool to perform (see "Types of Model Coverage" on page 1-3). Simulink Coverage gathers and reports those types of coverage for the subsystems, models, and referenced models that you specify.

The structural coverage levels are listed in order of strictness of test case coverage analysis:

- `Block Execution` — Enables "Execution Coverage (EC)" on page 1-3
- `Decision` — Enables "Execution Coverage (EC)" on page 1-3 and "Decision Coverage (DC)" on page 1-3
- `Condition Decision` — Enables "Execution Coverage (EC)" on page 1-3, "Decision Coverage (DC)" on page 1-3, and "Condition Coverage (CC)" on page 1-3
- `Modified Condition Decision Coverage (MCDC)` — enables "Execution Coverage (EC)" on page 1-3, "Decision Coverage (DC)" on page 1-3, "Condition Coverage (CC)" on page 1-3, and "Modified Condition/Decision Coverage (MCDC)" on page 1-4

Coverage metrics also includes **Other metrics**:

- "Lookup Table Coverage" on page 1-5
- "Signal Range Coverage" on page 1-5
- "Signal Size Coverage" on page 1-5
- "Objectives and Constraints Coverage" on page 1-6
- "Saturate on Integer Overflow Coverage" on page 1-7
- "Relational Boundary Coverage" on page 1-7

You must select **Enable coverage analysis** to select the coverage metrics.

Results

In the **Results** section of the Coverage Configuration Parameters, select the destination for coverage results. You must select **Enable coverage analysis** on the **Coverage** pane to set the **Results** options.

- **Save last run in workspace variable** — Saves the results of the last simulation run in a `cvdata` object in the workspace. Specify the workspace variable name in **cvdata object name**.
- **cvdata object name** — Name of the workspace variable where the results of the last simulation run are saved. You must select **Save last run in workspace variable** to specify the `cvdata` object name.
- **Increment variable name with each simulation (var1, var2, ...)** — Appends numerals to the workspace variable names for each new result so that earlier results are not overwritten. You must select **Save last run in workspace variable** to enable this option.
- **Autosave data file name** — Name of file to which coverage data results are saved. The default name is `$ModelName$_cvdata`. `$ModelName$` is the name of the model.
- **Output directory** — The folder where the coverage data is saved. The default location is `slcov_output/$ModelName$` in the current folder. `$ModelName$` is the name of the model.

See Also

Related Examples

- "Access, Manage, and Aggregate Coverage Results" on page 3-7

Access, Manage, and Aggregate Coverage Results

In this section...
"Accessing Coverage Data from the Results Explorer" on page 3-7
"Managing Coverage Data from the Results Explorer" on page 3-11
"Accumulating Coverage Data from the Results Explorer" on page 3-12

After you "Specify Coverage Options" on page 3-2 and collect coverage results, you can use the Results Explorer to access, manage, and aggregate the coverage data that you collect. After you aggregate the coverage results you need, you can then create a "Top-Level Model Coverage Report" on page 6-10 or "Export Model Coverage Web View" on page 6-39 using your aggregated coverage data.

Accessing Coverage Data from the Results Explorer

To open the Results Explorer after coverage analysis, in the **Coverage Analyzer** app, click on **Results Explorer**. The Results Explorer opens to show the most recent coverage run:

You can view the current data results summary from within the Results Explorer or click **Generate Report** to create a full coverage report. If you do not make any changes to your model after you

collect coverage, you do not need to re-simulate the model to generate a new coverage report. For more information on coverage reports, see "Top-Level Model Coverage Report" on page 6-10.

Click **Highlight model with coverage results** to provide highlighted results in your model that allow you to quickly see coverage results for model objects. For more information, see "Overview of Model Coverage Highlighting" on page 5-22.

Settings

In the coverage Results Explorer, you can access the data and reporting settings for your coverage data. To access these settings, click **Settings**.
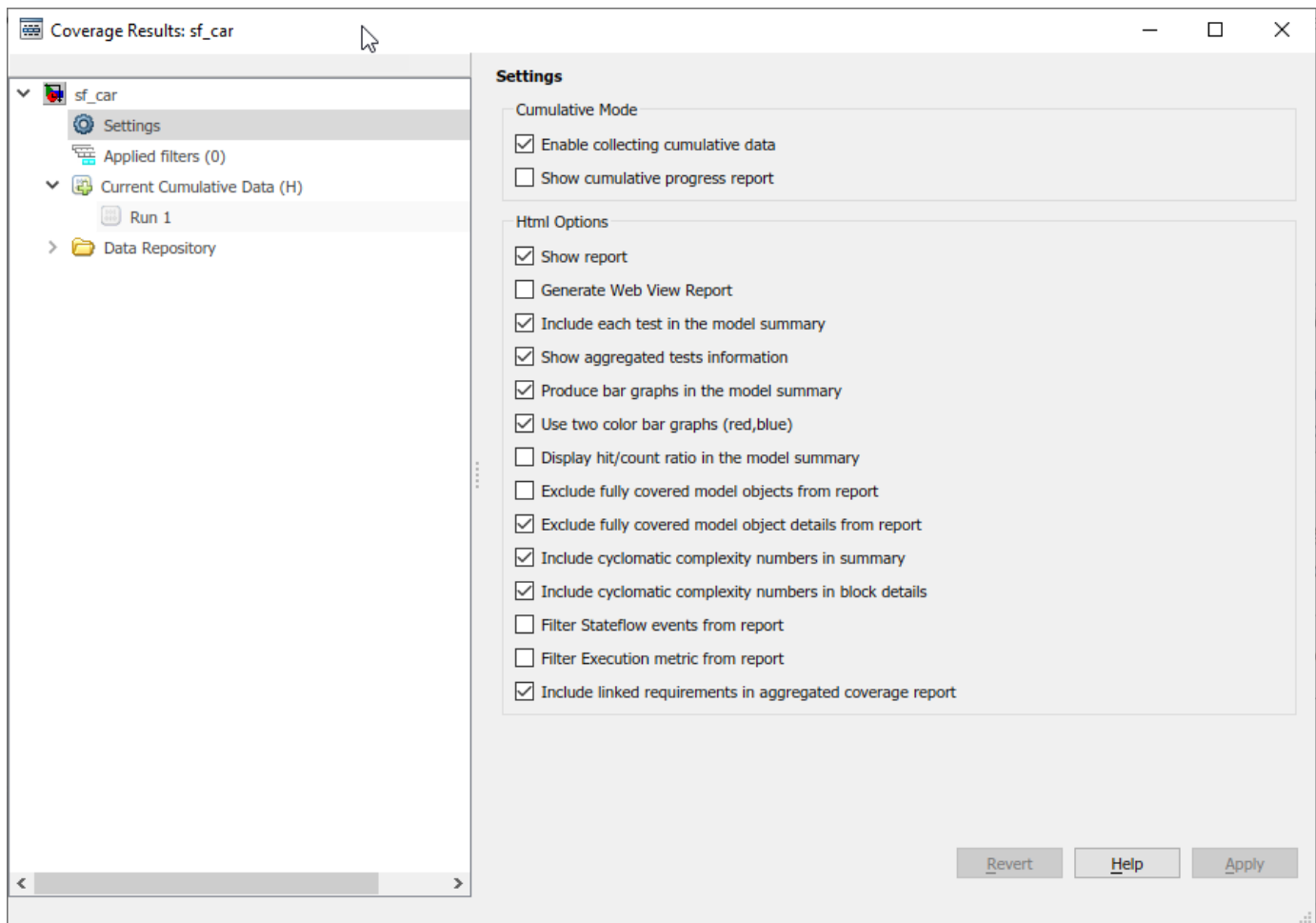
Option	Description
Enable collecting cumulative data	Aggregates coverage results from successive simulations, by default. You specify the name and output folder of the `.cvt` file in the in the "Results" on page 3-6 section of the Configuration Parameters dialog box. For more information, see "Cumulative Coverage Data" on page 3-14.

Option	Description
Show cumulative progress report	Shows the **Current Run** coverage results, the **Delta** of coverage compared to the previous cumulative data, and the total **Cumulative** data from all current cumulative data separately in the coverage reports. If you do not select this option, only the total **Cumulative** data from all current cumulative data are shown.
Show report	Opens a generated HTML coverage report in a MATLAB browser window at the end of model simulation. For more information, see "Top-Level Model Coverage Report" on page 6-10. You access the HTML report from the Simulink Coverage contextual tabs, which appear when you open the **Coverage Analyzer** app.
Generate Web View Report	Opens a generated Model Coverage Web View in a MATLAB browser window at the end of model simulation. For more information, see "Export Model Coverage Web View" on page 6-39.
Include each test in the model summary	At the top of the HTML report, the model hierarchy table includes columns listing the coverage metrics for each test. If you do not select this option, the model summary reports only the total coverage.
Show aggregated tests information	If you collect coverage for one or more subsystem harness, the **Aggregated Tests** section lists each unit test run. For more information, see "Aggregated Tests" on page 6-11.
Produce bar graphs in the model summary	Causes the model summary to include a bar graph for each coverage result for a visual representation of the coverage.
Use two color bar graphs (red, blue)	Red and blue bar graphs are displayed in the report instead of black and white bar graphs.
Display hit/count ratio in the model summary	Reports coverage numbers as both a percentage and a ratio, for example, 67% (8/12).
Exclude fully covered model objects from report	The coverage report includes only model objects that the simulation does not cover fully, useful when developing tests, because it reduces the size of the generated reports.
Exclude fully covered model object details from report	If you choose to include fully covered model objects in the report, the report does not include the details of the fully covered model objects

Option	Description
Include cyclomatic complexity numbers in summary	Includes the cyclomatic complexity (see "Types of Model Coverage" on page 1-3) of the model and its top-level subsystems and charts in the report summary. A cyclomatic complexity number shown in boldface indicates that the analysis considered the subsystem itself to be an object when computing its complexity. Boldface text can occur for atomic and conditionally executed subsystems and Stateflow Chart blocks.
Include cyclomatic complexity numbers in block details	Includes the cyclomatic complexity metric in the block details section of the report.
Filter Stateflow events from report	Excludes coverage data on Stateflow events.
Filter Execution metric from report	Excludes coverage data on Execution metrics
Include linked requirements in aggregate coverage report	If you run at least two test cases in Simulink Test™ that are linked to requirements in Simulink Requirements™, the aggregated coverage report details the links between model elements, test cases, and linked requirements. For more information, see "Requirement Testing Details" on page 6-20.

Creating and Managing Filters

You can create, load, or edit filters for the current coverage data from within the Results Explorer.

1   Open the Results Explorer.

2   Click the **Applied filters** tab.
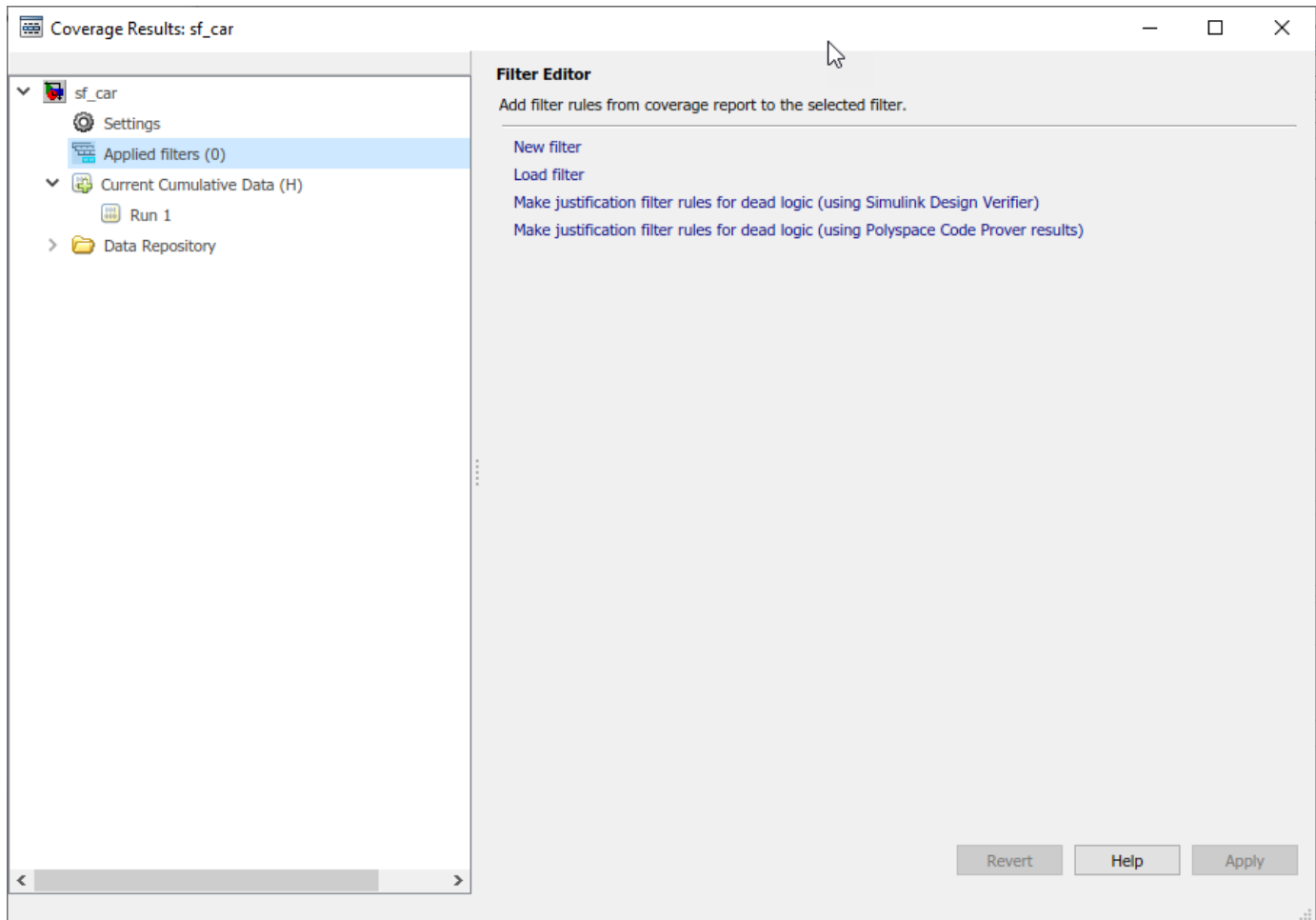
Option	Description
New filter	Creates a blank filter and opens the **Filter Editor** pane of the Coverage Results Explorer.
Load filter	Opens the **Load filter** selection window.
Make justification filter rules for dead logic (using Simulink Design Verifier)	Launches Simulink Design Verifier design error detection. For more information, see "Run a Design Error Detection Analysis" (Simulink Design Verifier).
Make justification filter rules for dead logic (using Polyspace® Code Prover™ results)	Opens a file selection window which asks for a Polyspace Code Prover results file.

For more information on filtering model objects, see "Creating and Using Coverage Filters" on page 7-11.

Managing Coverage Data from the Results Explorer

After you collect coverage, you can manage the coverage data from the Results Explorer. To view coverage data details, under **Current Cumulative Data**, click the coverage data of interest. You can

edit the description and tags for each run. Before you leave the coverage data details view, click **Apply** to apply your changes. Otherwise, the changes are reverted.

When you apply changes to coverage data, such as adding descriptions and tags, the data shows an asterisk next to its icon. To save these changes, right-click the data and click **Save modified coverage data**.

Accumulating Coverage Data from the Results Explorer

If you collect multiple coverage runs, each run is listed separately in the Data Repository. You can drag and drop runs from the Data Repository to the Current Cumulative Data to manage which runs to include in the cumulative coverage data. Alternatively, right-click runs in the Data Repository or the Current Cumulative Data to include or exclude them in the cumulative coverage data.
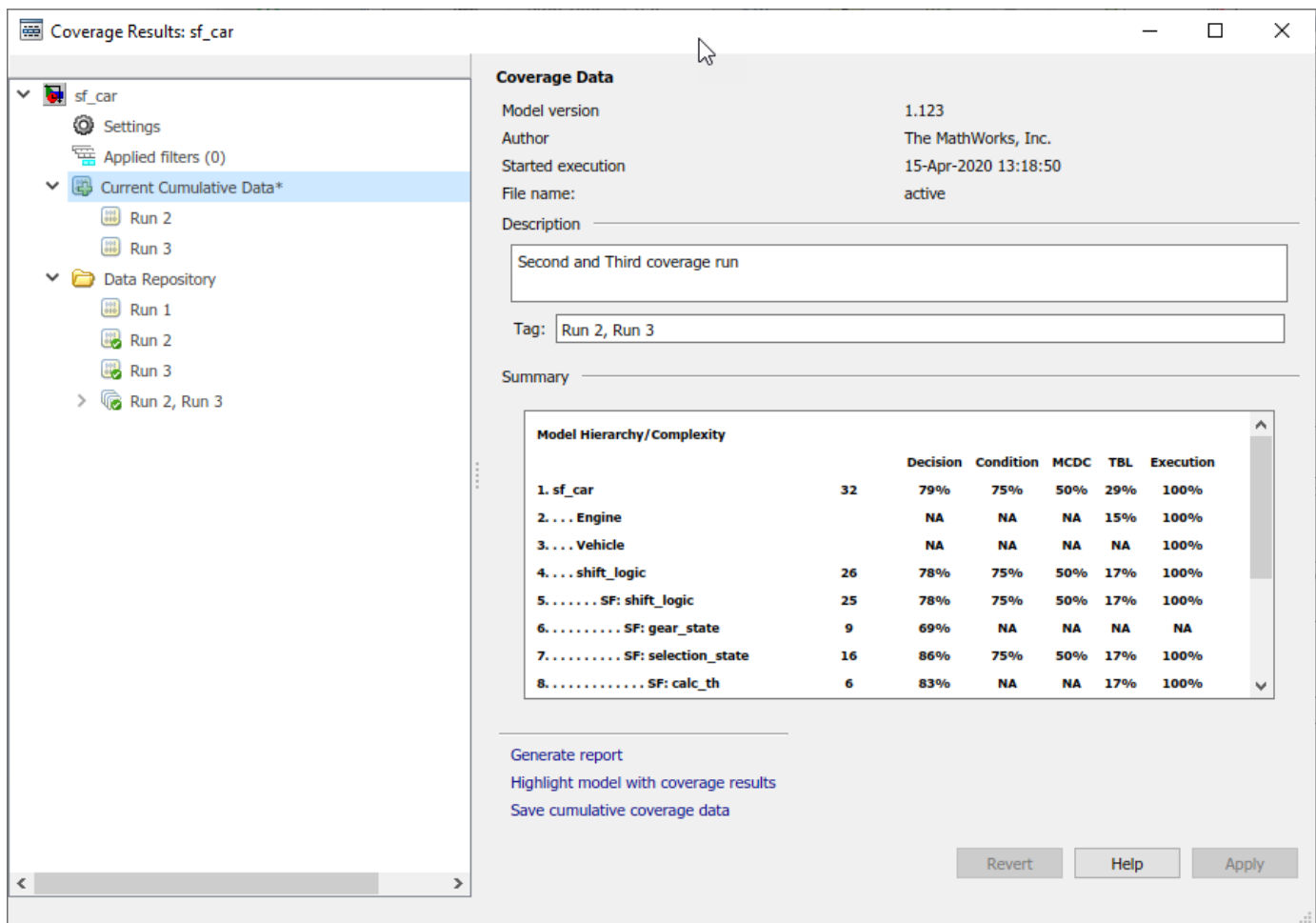
To save the current cumulative data set to a .cvt file, click **Save cumulative coverage data**. Alternatively, you can right-click the **Current Cumulative Data** and select **Save cumulative coverage data**.

Load Existing Coverage Data

The Data Repository contains the coverage data, which is saved to the Input folder. You specify the Input folder on the **Configuration Parameters dialog box > Coverage > "Results" on page 3-6 section**, in the **Output directory** field.
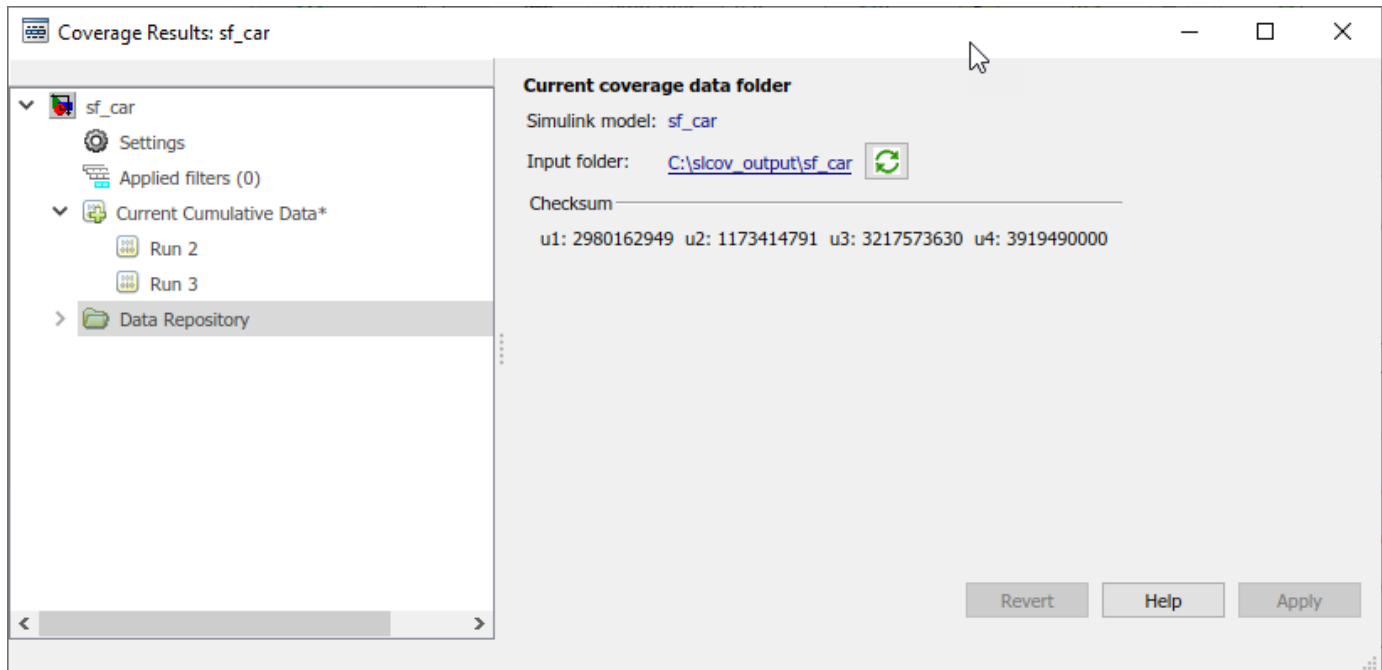
To synchronize the data in the input folder and the data in the Data Repository, click **Synchronize with the current coverage data folder** .

To load existing coverage data to the Data Repository:

1 Right-click the **Data Repository**.

2 Select **Load coverage data**.

3 Select existing coverage data for the current model and click **Open**.

Cumulative Coverage Data

On the **Coverage** pane in the Configuration Parameters dialog box, click the **...** to open the **Advanced parameters**. If you select **Enable cumulative data collection** and **Save cumulative results in workspace variable**, a coverage running total is updated with new results at the end of each simulation. However, if you change model or block settings between simulations that are incompatible with settings from previous simulations and affect the type or number of coverage points, the cumulative coverage data resets.

When you restore a running total from saved data, the saved results are reflected in the next cumulative report. If a running total exists when you restore a saved value, the existing value is overwritten.

Whenever you report on more than one single simulation, the coverage displayed for truth tables and lookup-table maps is based on the total coverage of all the reported runs. For cumulative reports, this information includes all the simulations where cumulative results are stored. For more information about managing cumulative results, see "Access, Manage, and Aggregate Coverage Results" on page 3-7.

You can make cumulative coverage results persist between MATLAB sessions. The `cvload` parameter `RESTORETOTAL` must be `1` to restore cumulative results. At the end of the sessions, use `cvsave` to save results to a file. At the beginning of the next session, use `cvload` to load the results.

When you save the coverage results to a file using `cvsave` and a model name argument, the file also contains the cumulative running total. When you load that file into the coverage tool using `cvload`, you can select whether you want to restore the running total from the file.

You can also calculate cumulative coverage results at the command line, through the + operator:

```
covdata1 = cvsim(test1);
covdata2 = cvsim(test2);
cvhtml('cumulative_report', covdata1 + covdata2);
```

Cumulative Coverage Analysis

This example illustrates the use of the Coverage Results Explorer to simplify the generation of cumulative coverage data and reports spanning a set of multiple coverage runs.

Open Example Model

This example uses the `slvnvdemo_ratelim_harness` model to explain the settings and options to accumulate coverage. Inside this model is an implementation of an Adjustable Rate Limiter. It uses three *Switch* blocks to control when the output should be limited and the type of limit to apply.

Inputs are produced using three *From Workspace* blocks: **gain**, **rising limit**, and **falling limit**. The values of the inputs are specified by six variables defined in the MATLAB® workspace: **t_gain**, **u_gain**, **t_pos**, **u_pos**, **t_neg**, and **u_neg**.

```
open_system('slvnvdemo_ratelim_harness');
```
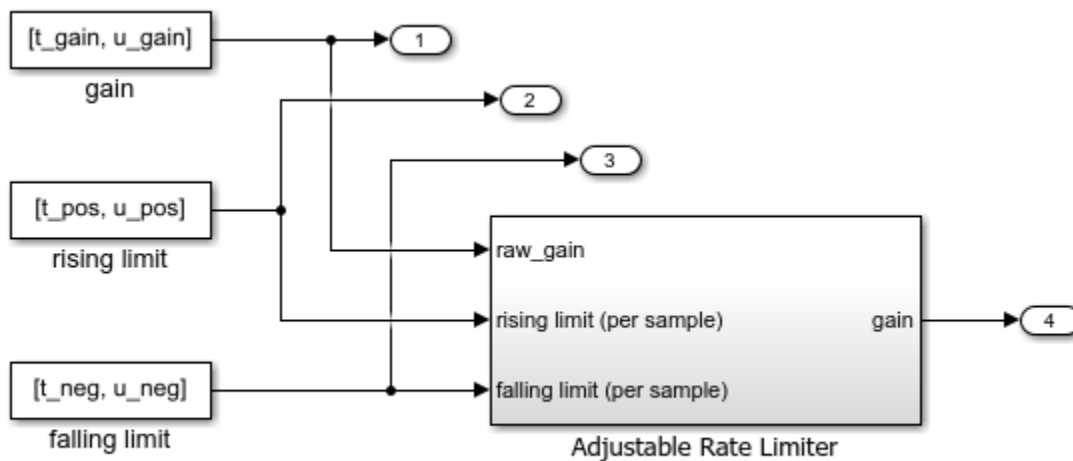

Copyright 1990-2006 The MathWorks Inc.

```
open_system('slvnvdemo_ratelim_harness/Adjustable Rate Limiter');
```
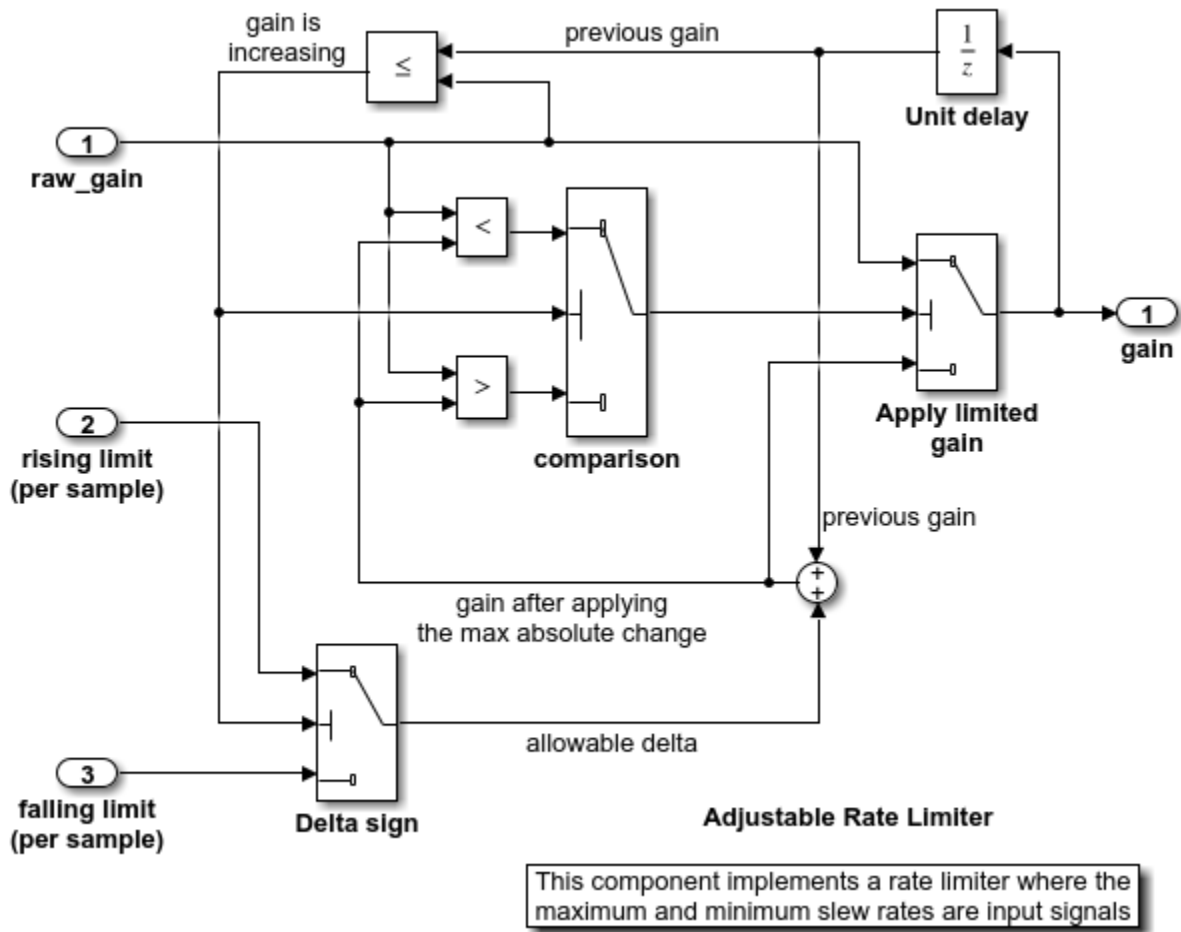
Enable Coverage Analysis

Start by opening the coverage settings. From the **Modeling** tab, select **Model Settings**.

To enable the coverage tool, select **Enable coverage analysis** in the **Coverage** pane. This setting enables the other options in the Coverage pane.

For this example, collect condition and decision coverage. Under the **Coverage metrics** panel, set the **Structural coverage level** to *Condition Decision*.
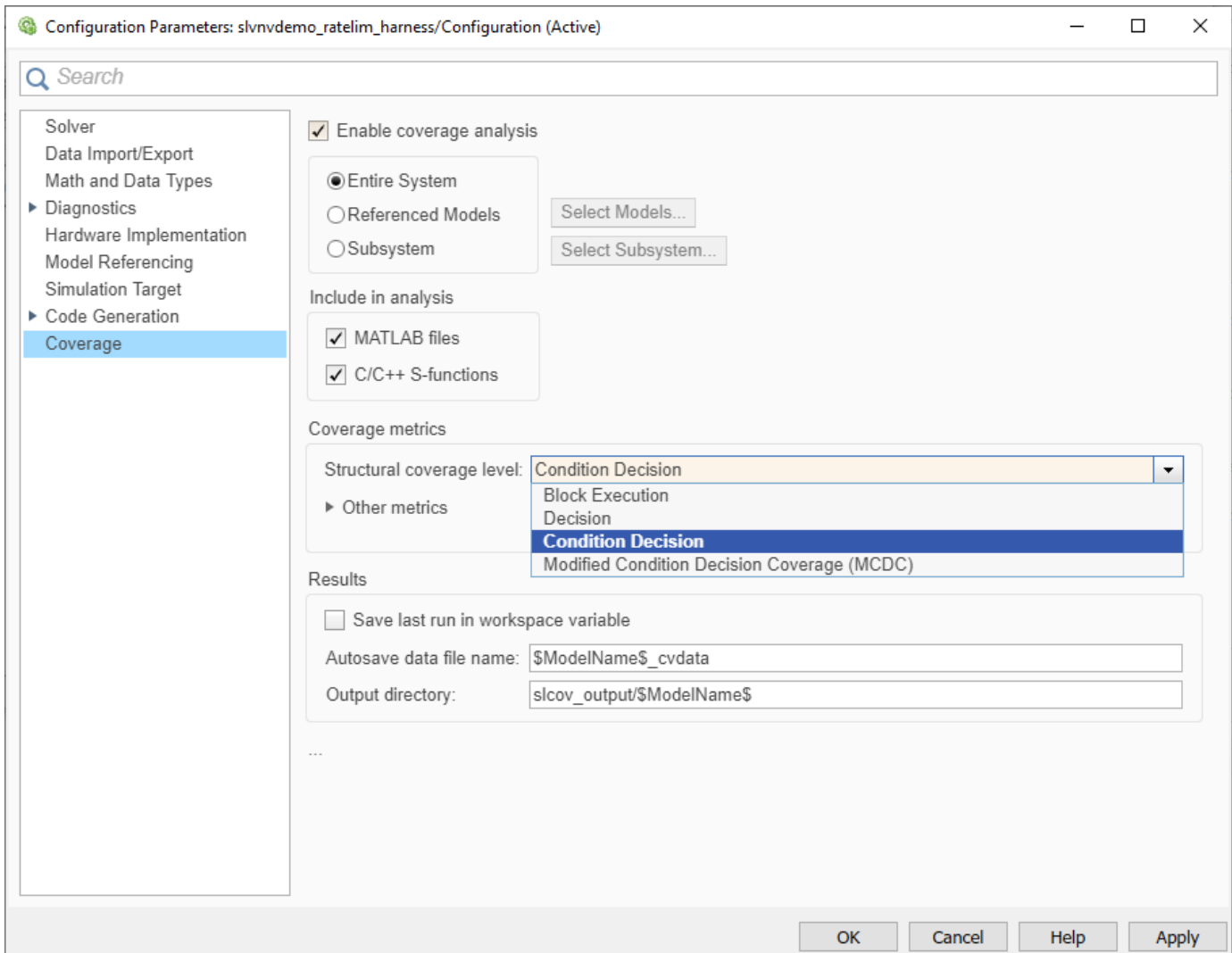
Click **OK** to apply your selected settings and close this dialog.

Simulate Model with First Test Case

The first test case exercises the scenario where the input values do not change rapidly. It uses a sine wave as the time varying signal and constants for rising and falling limits.

```
t_gain = (0:0.02:2.0)';
u_gain = sin(2*pi*t_gain);
```

Calculate the minimum and maximum change of the time varying input using the MATLAB `diff` function.

```
max_change = max(diff(u_gain))
min_change = min(diff(u_gain))
```

```
max_change =

    0.1253
```

```
min_change =

   -0.1253
```

Based on these minimum and maximum rates of change, set the rate limits to `1` and `-1`. As such, the rate of change of the input will be well within these limits for this test run.

```
t_pos = [0;2];
u_pos = [1;1];
t_neg = [0;2];
u_neg = [-1;-1];
```

Simulate the model with this first set of input variables by clicking the **Run (Coverage)** button.

Review First Test Case in Results Explorer

To open the Results Explorer, in the **Coverage Analyzer** app, click **Results Explorer**.

At this point the **Current Cumulative Data** contains just this first coverage run (tagged as *Run 1*). The Results Explorer initially shows information regarding this latest coverage run, including a summary of results for each enabled metric.

To keep track of the intent of this simulation, enter the text `"Test within rate limits"` in the **Description** field and click **Apply**.

Simulate Model with Second Test Case

The second test case complements the first case with a rising gain that exceeds the rate limit. After a second it increases the rate limit so that the gain changes are below that limit.

```
t_gain = [0;2];
u_gain = [0;4];
t_pos = [0;1;1;2];
u_pos = [1;1;5;5]*0.02;
t_neg = [0;2];
u_neg = [0;0];
```

Simulate the model with this second set of variables by clicking the **Run (Coverage)** button.

Generate Cumulative Progress Report for Second Test Case

Now that multiple coverage runs have been performed, you can generate cumulative coverage reports.

First, add a brief description of this run, as was done for the previous simulation. Enter the text "Test rising rate limit" in the **Description** field for *Run 2* and click **Apply**.
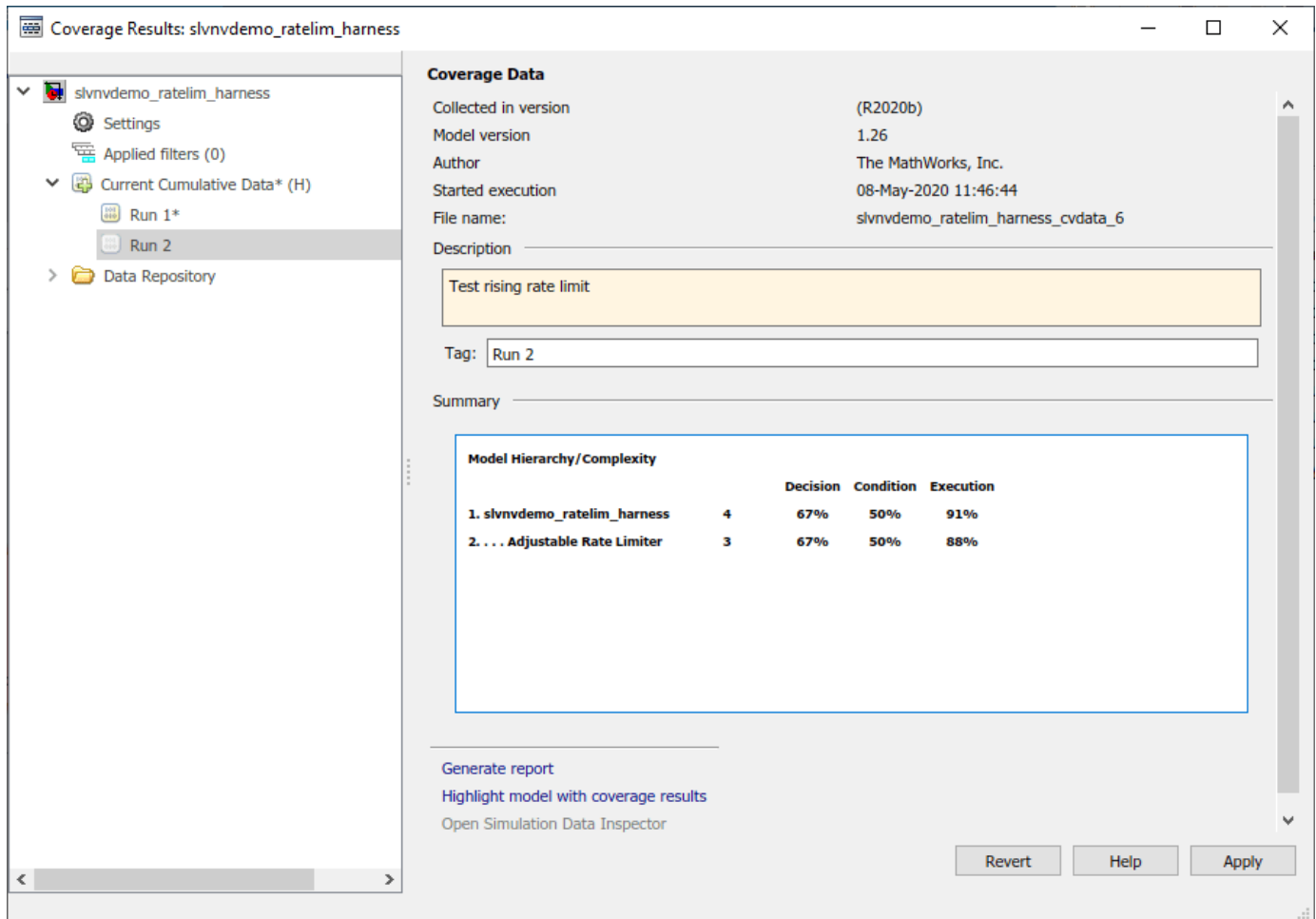
There are different formats of coverage reports that can be generated. To visualize how the most recent simulation affects the cumulative coverage results, you can generate a cumulative progress report.

In the Results Explorer, under **Settings**, select **Show cumulative progress report** and click **Apply**.
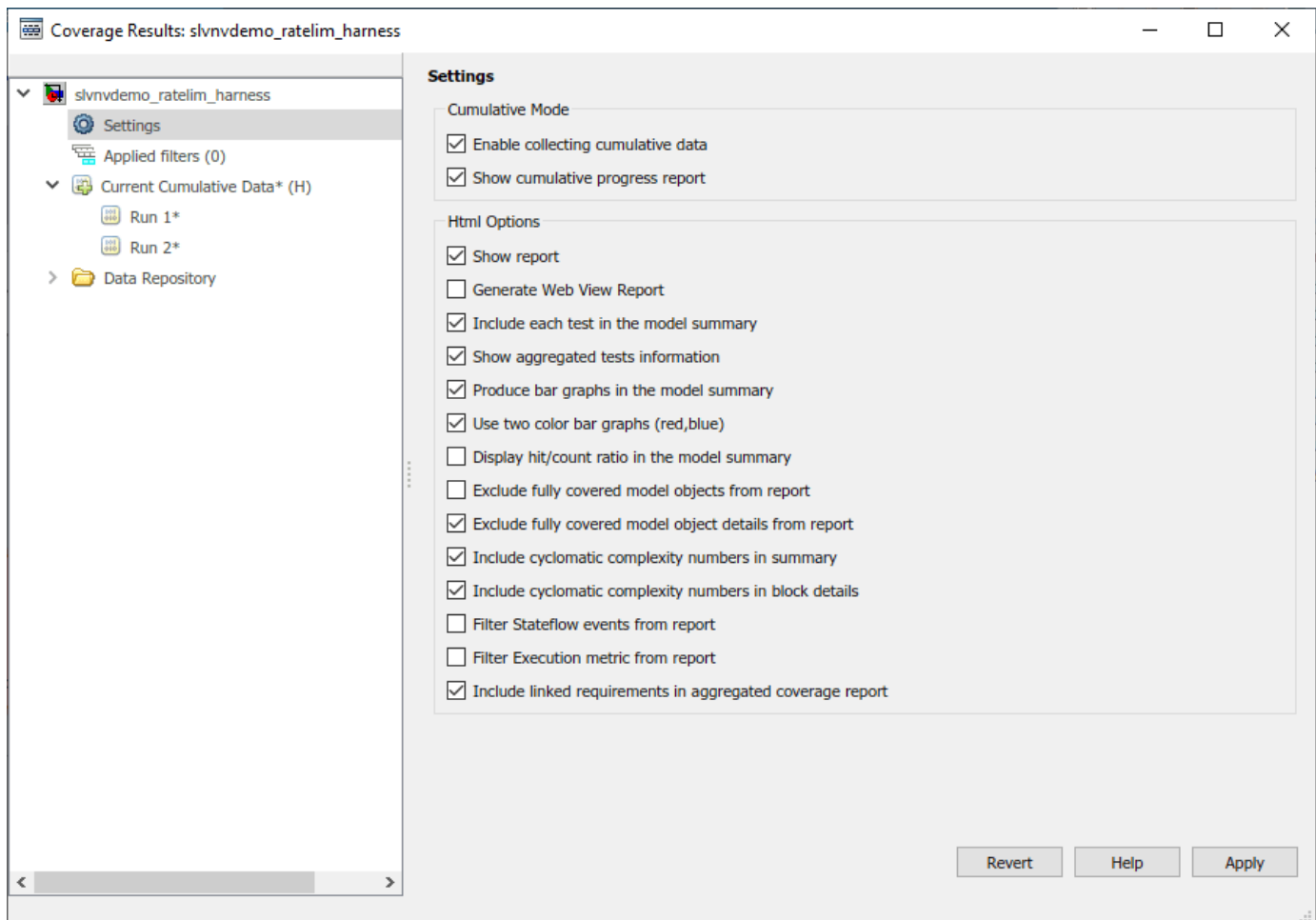
Click on **Current Cumulative Data** in the leftmost pane of the Results Explorer. Note that the **Summary** indicates the cumulative coverage results accumulated from *Run 1* and *Run 2*. Click on **Generate Report** to create the cumulative progress report.

The **Summary** section of the cumulative progress report has three columns: *Current Run, Delta,* and *Cumulative*. The *Current Run* column displays the coverage from the last simulation listed under **Current Cumulative Data** (which is *Run 2* in this case). The *Delta* column displays the coverage exposed by the current run that was not achieved in the cumulative results before this simulation. The *Cumulative* column gives the current cumulative coverage results.

Coverage Report for slvnvdemo_ratelim_harness

Table of Contents

Analysis Information

Coverage Data Information

Collected in version	(R2020b)

Model Information

Model version	1.26
Author	The MathWorks, Inc.
Last saved	Mon Aug 12 12:49:12 2019

Simulation Optimization Options

Default parameter behavior	tunable
Block reduction	off
Conditional branch optimization	on

Coverage Options

Analyzed model	slvnvdemo_ratelim_harness
Logic block short circuiting	off

Blocks Eliminated from Coverage Analysis

# Model Object	Rationale
slvnvdemo_ratelim_harness/Adjustable Rate Limiter/Relational Operator3	It might not be executed because of Conditional input branch optimization

Tests

Test#	Started execution	Ended execution	Description
Test 1	07-May-2020 16:51:11	07-May-2020 16:51:12	Current Run
Test 2	07-May-2020 16:48:40	07-May-2020 16:51:12	Delta
Test 3	07-May-2020 16:48:40	07-May-2020 16:51:12	Cumulative

Summary

Model Hierarchy/Complexity		Current Run Decision	Condition	Execution	Delta Decision	Condition	Execution	Cumulative Decision	Condition	Execution
1. slvnvdemo_ratelim_harness	4	67%	50%	91%	17%	17%	0%	100%	83%	100%
2. . . . Adjustable Rate Limiter	3	67%	50%	88%	17%	17%	0%	100%	83%	100%

Simulate Model with Third Test Case

The third test case is a mirror image of the second, with the rising gain replaced by a falling gain.

```
t_gain = [0;2];
u_gain = [-0.02;-4.02];
t_pos = [0;2];
u_pos = [0;0];
t_neg = [0;1;1;2];
u_neg = [-1;-1;-5;-5]*0.02;
```

Simulate the model with this third set of variables by clicking the **Run (Coverage)** button.

Generate Cumulative Progress Report for Third Test Case

Once again, add a brief description of the latest run. Enter the text "Test falling rate limit" in the **Description** field for *Run 3* and click **Apply**.

Navigate to **Current Cumulative Data** and click **Generate Report** to create a cumulative progress report for this latest run.

Notice that with this latest run, the cumulative results achieve full coverage for the Decision, Condition, and Execution metrics.

Coverage Report for slvnvdemo_ratelim_harness

Table of Contents

Analysis Information

Coverage Data Information

Collected in version	(R2020b)

Model Information

Model version	1.26
Author	The MathWorks, Inc.
Last saved	Mon Aug 12 12:49:12 2019

Simulation Optimization Options

Default parameter behavior	tunable
Block reduction	off
Conditional branch optimization	on

Coverage Options

Analyzed model	slvnvdemo_ratelim_harness
Logic block short circuiting	off

Blocks Eliminated from Coverage Analysis

# Model Object	Rationale
slvnvdemo_ratelim_harness/Adjustable Rate Limiter/Relational Operator1	It might not be executed because of Conditional input branch optimization

Tests

Test#	Started execution	Ended execution	Description
Test 1	07-May-2020 16:56:46	07-May-2020 16:56:46	Current Run
Test 2	07-May-2020 16:48:40	07-May-2020 16:56:46	Delta
Test 3	07-May-2020 16:48:40	07-May-2020 16:56:46	Cumulative

Summary

Model Hierarchy/Complexity		Current Run			Delta			Cumulative		
		Decision	Condition	Execution	Decision	Condition	Execution	Decision	Condition	Execution
1. slvnvdemo_ratelim_harness	4	67%	50%	91%	0%	17%	0%	100%	100%	100%
2. .... Adjustable Rate Limiter	3	67%	50%	88%	0%	17%	0%	100%	100%	100%

Refine Cumulative Dataset

If you determine that a particular coverage run is not necessary, you can exclude this run from the cumulative dataset and generate a new cumulative report.

In the Results Explorer, under **Current Cumulative Data**, right-click on *Run 1* and select **Exclude from cumulative data**.
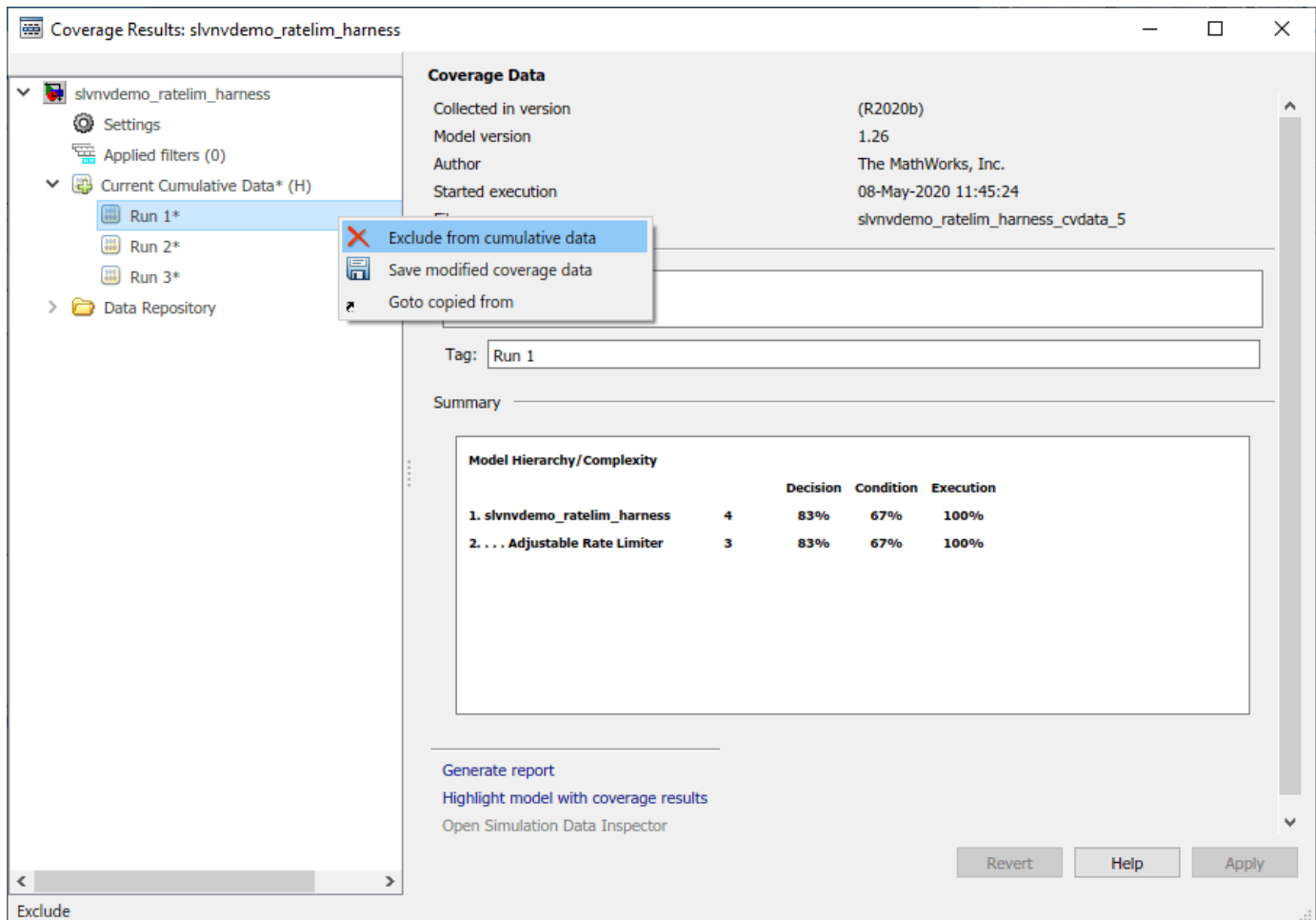
Generate Final Cumulative Coverage Report

Now that you have selected the desired subset of test runs, you can generate a coverage report for the accumulated results.

Navigate to **Settings**, deselect **Show cumulative progress report**, and then click **Apply**.

Navigate to **Current Cumulative Data** and click **Generate Report**.

The cumulative coverage report displays the results associated with the current cumulative data. Notice under the **Tests** section, there is a single test with the description `"Test rising rate limit,Test falling rate limit"`, indicating that this test contains the accumulated results from runs 2 and 3.

The **Summary** section shows that these cumulative results attain full coverage for all metrics analyzed.

Coverage Report for slvnvdemo_ratelim_harness

Table of Contents

Analysis Information

Coverage Data Information

Collected in version	(R2020b)

Model Information

Model version	1.26
Author	The MathWorks, Inc.
Last saved	Mon Aug 12 12:49:12 2019

Simulation Optimization Options

Default parameter behavior	tunable
Block reduction	off
Conditional branch optimization	on

Coverage Options

Analyzed model	slvnvdemo_ratelim_harness
Logic block short circuiting	off

Aggregated Tests

Run	Test Name	Description	Date
Model: "slvnvdemo_ratelim_harness"			
T1	Run 2	Test rising rate limit	07-May-2020 16:51:12
T2	Run 3	Rest falling rate limit	07-May-2020 16:56:46

Summary

Model Hierarchy/Complexity

		Decision	Condition	Execution
1. slvnvdemo_ratelim_harness	4	100%	100%	100%
2. .... Adjustable Rate Limiter	3	100%	100%	100%

Obtain Cumulative Coverage for Reusable Subsystems and Stateflow® Constructs

This example shows how to create and view cumulative coverage results for a model with a reusable subsystem.

Simulink® Coverage™ provides cumulative coverage for multiple instances of identically configured:

- Reusable subsystems
- Stateflow™ constructs

To obtain cumulative coverage, you add the individual coverage results at the command line. You can get cumulative coverage results for multiple instances across models and test harnesses by adding the individual coverage results.

Open example model

At the MATLAB® command line, type:

```
model = 'slvnvdemo_cv_mutual_exclusion';
open_system(model);
```


Copyright 1990-2019 The MathWorks Inc.

This model has two instances of a reusable subsystem. The instances are named Subsystem 1 and Subsystem 2.

Get decision coverage for Subsystem 1

Execute the commands for Subsystem 1 decision coverage:

```
testobj1 = cvtest([model '/Subsystem 1']);
testobj1.settings.decision = 1;
covobj1 = cvsim(testobj1);
```

Get decision coverage for Subsystem 2

Execute the commands for Subsystem 2 decision coverage:

```
testobj2 = cvtest([model '/Subsystem 2']);
testobj2.settings.decision = 1;
covobj2 = cvsim(testobj2);
```

Add coverage results for Subsystem 1 and Subsystem 2

Execute the command to create cumulative decision coverage for Subsystem 1 and Subsystem 2:

```
covobj3 = covobj1 + covobj2;
```

Generate coverage report for Subsystem 1

Create an HTML report for Subsystem 1 decision coverage:

```
cvhtml('subsystem1',covobj1)
```

The report indicates that decision coverage is 50% for Subsystem 1. The `true` condition for `enable logical value` is not analyzed.

Generate coverage report for Subsystem 2

Create an HTML report for Subsystem 2 decision coverage:

```
cvhtml('subsystem2',covobj2)
```

The report indicates that decision coverage is 50% for Subsystem 2. The `false` condition for `enable logical value` is not analyzed.

Generate coverage report for cumulative coverage of Subsystem 1 and Subsystem 2

Create an HTML report for cumulative decision coverage for Subsystem 1 and Subsystem 2:

```
cvhtml('cum_subsystem',covobj3)
```

Cumulative decision coverage for reusable subsystems Subsystem 1 and Subsystem 2 is 100%. Both the `true` and `false` conditions for `enable logical value` are analyzed.

Saturation on Integer Overflow Coverage

Simulate this model to collect and report Saturate on integer overflow coverage.

Enabling Saturation on integer overflow

To enable the coverage metric **Saturation on integer overflow**:

Click on the *Modeling* tab in the toolstrip. Click on *Model Settings*.

On the left pane, click on *Coverage*. Ensure that *Enable coverage analysis* is checked.

Expand the *Other metrics* drop down list. Check the box next to *Saturation on integer overflow*.

What Saturation on integer overflow does

The coverage tool identifies all the blocks that have the Saturation on integer overflow parameter enabled. After simulating the model, the tool reports the number of times each block saturates on integer overflow.

In this example, the test harness supplies the Test Unit with an input to reach full coverage on one of the Sum blocks in the Controller subsystem.

The other two Sum blocks in the Controller subsystem do not generate true cases because they do not reach their saturation thresholds.

When to use Saturation on integer overflow

Saturation on integer overflow coverage helps to identify missing tests for blocks and blocks that do not need the saturation on integer overflow parameter enabled, optimizing design efficiency.

For more information, see "Saturate on Integer Overflow Coverage" on page 1-7.

Saturation on Integer Overflow Coverage Example

Copyright 2006-2012 The MathWorks, Inc.
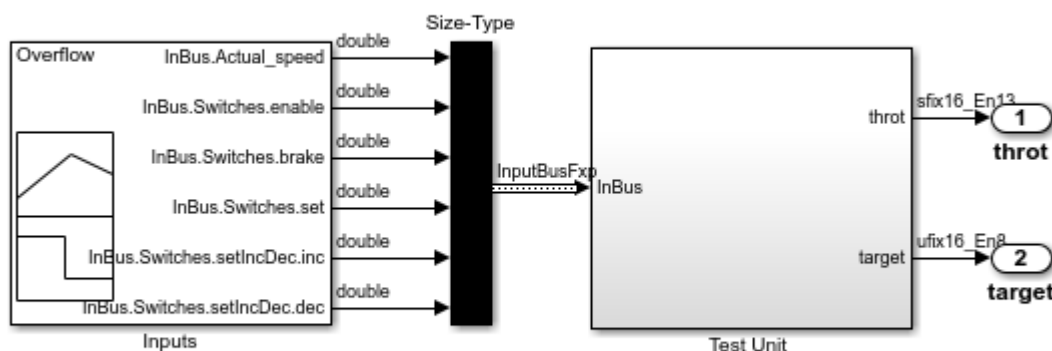
Code Coverage

Types of Code Coverage

If you have Embedded Coder, Simulink Coverage can perform several types of code coverage analysis for models in software-in-the-loop (SIL) mode, processor-in-the-loop (PIL) mode, and for the code within supported S-Function blocks.

Statement Coverage for Code Coverage

Statement coverage determines the number of source code statements that execute when the code runs. Use this type of coverage to determine whether every statement in the program has been invoked at least once.

Statement coverage = (Number of executed statements / Total number of statements) *100

Statement Coverage Example

This code snippet contains five statements. To achieve 100% statement coverage, you need at least three test cases. Specifically, tests with positive *x* values, negative *x* values, and *x* values of zero.

```
if (x > 0)
  printf( "x is positive" );
else if (x < 0)
  printf( "x is negative" );
else
  printf( "x is 0" );
```

Condition Coverage for Code Coverage

Condition coverage analyzes statements that include conditions in source code. Conditions are C/C++ Boolean expressions that contain relation operators (<, >, <=, or >=), equation operators (!= or ==), or logical negation operators (!), but that do not contain logical operators (&& or ||). This type of coverage determines whether every condition has been evaluated to all possible outcomes at least once.

Condition coverage = (Number of executed condition outcomes / Total number of condition outcomes) *100

Condition Coverage Example

In this expression:

```
y = x<=5 && x!=7;
```

there are these conditions:

```
x<=5
x!=7
```

Decision Coverage for Code Coverage

Decision coverage analyzes statements that represent decisions in source code. Decisions are Boolean expressions composed of conditions and one or more of the logical C/C++ operators && or ||. Conditions within branching constructs (if/else, while, do-while) are decisions. Decision coverage determines the percentage of the total number of decision outcomes the code exercises during execution. Use this type of coverage to determine whether all decisions, including branches, in your code are tested.

Note The decision coverage definition for DO-178C compliance differs from the Simulink Coverage definition. For decision coverage compliance with DO-178C, select the `Condition Decision` structural coverage level for Boolean expressions not containing && or || operators.

Decision coverage = (Number of executed decision outcomes / Total number of decision outcomes) *100

Decision Coverage Example

This code snippet contains three decisions:

```
y = x<=5 && x!=7;                    // decision #1

if( x > 0 )                    // decision #2
  printf( "decision #2 is true" );
else if( x < 0 && y )              // decision #3
  printf( "decision #3 is true" );
else
  printf( "decisions #2 and #3 are false" );
```

Modified Condition/Decision Coverage (MCDC) for Code Coverage

Modified condition/decision coverage (MCDC) is the extent to which the conditions within decisions are independently exercised during code execution.

*   All conditions within decisions have been evaluated to all possible outcomes at least once.
*   Every condition within a decision independently affects the outcome of the decision.

MCDC coverage = (Number of conditions evaluated to all possible outcomes affecting the outcome of the decision / Total number of conditions within the decisions) *100

Modified Condition/Decision Coverage Example

For this decision:

```
X || ( Y && Z )
```

the following set of test cases delivers 100% MCDC coverage.

	X	Y	Z
Test case #1	0	0	1
Test case #2	0	1	0
Test case #3	0	1	1
Test case #4	1	0	1

Cyclomatic Complexity for Code Coverage

Cyclomatic complexity is a measure of the structural complexity of code that uses the McCabe complexity measure. To compute the cyclomatic complexity of code, code coverage uses this formula:

$$c = \sum_{1}^{N}(o_n - 1)$$

N is the number of decisions in the code. o_n is the number of outcomes for the nth decision point. Code coverage adds 1 to the complexity number for each C/C++ function.

Coverage Example

For this code snippet, the cyclomatic complexity is 3:

```
void evalNum(int x)
{
  if (x > 0)                          // decision #1
    printf( "x is positive" );
  else if (x < 0)                     // decision #2
    printf( "x is negative" );
  else
    printf( "x is 0" );
}
```

The code contains one function that has two decision points. Each decision point has two outcomes. Using the preceding formula, N is 2, o_1 is 2, and o_2 is 2. Code coverage uses the formula with these decisions and outcomes and adds 1 for the function. The cyclomatic complexity for this code snippet is:

$$c = (o_1 - 1) + (o_2 - 1) + 1 = (2 - 1) + (2 - 1) + 1 = 3$$

Relational Boundary for Code Coverage

Relational boundary code coverage examines code that has relational operations. Relational boundary code coverage metrics align with those for model coverage, as described in "Relational Boundary Coverage" on page 1-7. Fixed-point values in your model are integers during code coverage.

Function Coverage

Function coverage determines whether all the functions of your code have been called during simulation. For instance, if there are ten unique functions in your code, function coverage checks if all ten functions have been executed at least once during simulation.

Function Call Coverage

Function call coverage determines whether all function call-sites in your code have been executed during simulation. For instance, if functions are called twenty times in your code, function call coverage checks if all twenty function calls have been executed during simulation.

Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode

If you have Embedded Coder and Simulink Coverage, you can analyze coverage for generated code during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation.

In this section...
"Enable SIL or PIL Code Coverage for a Model" on page 4-6
"Review the Coverage Results for Models in SIL or PIL Mode" on page 4-6
"Limitations" on page 4-7

Enable SIL or PIL Code Coverage for a Model

To record SIL or PIL code coverage for a model:

1 Ensure you have a supported compiler installed. For a list of supported compilers, see Supported and Compatible Compilers.

2 In the Configuration Parameters dialog box, on the left pane, click **Code Generation**. From the list, select **Verification**.

3 Under **Code profiling**, set **Measure function execution times** to `Off`.

4 Under **Code coverage for SIL or PIL**, set **Third-party tool** to `None (use Simulink Coverage)`.

5 Enable coverage for a model in SIL or PIL mode or a reference model in SIL or PIL mode.

6 Run a SIL or PIL simulation.

> **Note** The **Coverage (Run)** button in the Coverage toolstrip forces a Normal mode simulation for the top-level model and does not yield SIL or PIL code coverage.

To run a SIL or PIL simulation:

a In the Simulink window, click **Apps** and then click **SIL/PIL Manager**.

b In the **SIL/PIL** tab, click **Automated Verification > SIL/PIL Simulation Only**.

c Click **Run SIL/PIL**.

Review the Coverage Results for Models in SIL or PIL Mode

To view the code coverage information in the Code view, from the drop-down list to the right of the search box, select **Show code coverage**. If the option is disabled, then on the **Coverage** tab, click **Coverage Highlighting**. The code displays highlighting and annotations that show code coverage information. You can navigate from the code to the associated model blocks by using the links in the line numbers, code elements, and comments.

Coverage annotation

Links to model element

Tooltip with code coverage results
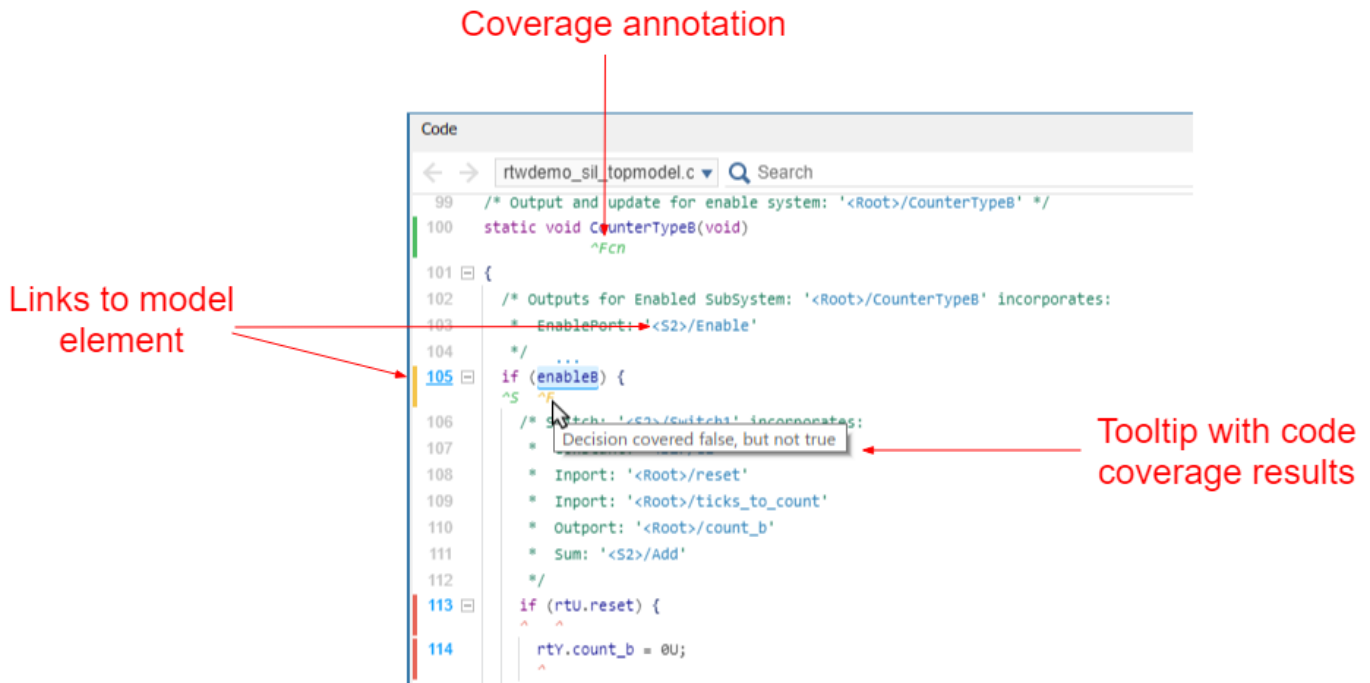
At the bottom of the Code view, the coverage section shows a summary of the code coverage report.

Code coverage summary

Tooltip with code coverage results

Limitations

Coverage for models in SIL and PIL mode has these limitations:

- The model must meet the requirements listed in "Enable SIL or PIL Code Coverage for a Model" on page 4-6.
- Code coverage results must not include external C/C++ files in read-only folders.
- The **Coverage (Run)** button in the Coverage toolstrip forces a Normal simulation and will not yield SIL or PIL code coverage.

See Also

Related Examples

- "Custom Toolchain Directives Required for Code Coverage and Execution Profiling" (Embedded Coder)
- "Software-in-the-Loop Code Coverage" on page 4-22
- "SIL/PIL Manager Verification Workflow" (Embedded Coder)

Collect Code Coverage Metrics with Simulink® Coverage™
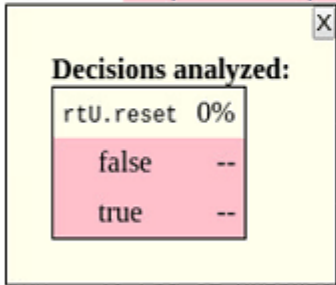
This example shows how to collect code coverage metrics during a software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation with Simulink® Coverage™.

You use the code coverage tool and code coverage report to view the recorded code coverage for a SIL simulation.

```
 99  /* Output and update for enable system: '<Root>/CounterTypeB' */
100  static void CounterTypeB(void)
101  {
102    /* Outputs for Enabled SubSystem: '<Root>/CounterTypeB' incorporates:
103     *  EnablePort: '<S2>/Enable'
104     */
105    if (enableB) {
106      /* Switch: '<S2>/Switch1' incorporates:
107       *  Constant: '<S2>/C1'
108       *  Inport: '<Root>/reset'
109       *  Inport: '<Root>/ticks to count'
110       *  Outport: '<Root>/count b'
111       *  Sum: '<S2>/Add'
112       */
113      if (rtU.reset) {
```

Decisions analyzed:

rtU.reset	0%
false	--
true	--

```
          uint8_T)(uint32_T)((uint32_T)rtU.ticks_to_count + (uint32_T)
          n: '<S2>/Switch1' */
123      /* End of Outputs for SubSystem: '<Root>/CounterTypeB' */
124  }
125
126  /* Model step function */
127  void rtwdemo_sil_topmodel_step(void)
128  {
129    /* Logic: '<Root>/Logical Operator2' incorporates:
130     *  Inport: '<Root>/count enable'
131     *  Inport: '<Root>/counter mode'
132     *  Logic: '<Root>/Logical Operator'
133     */
```

In this example, you measure model coverage during a simulation in normal mode, repeat the same simulation in SIL mode, and compare the recorded metrics from both simulations.

Compare model coverage and code coverage results by using the hyperlinks in the model coverage and code coverage reports.

For more examples of measuring SIL and PIL simulations, see "Test Generated Code with SIL and PIL Simulations" (Embedded Coder).

Initial Setup

Open the model.

```
model = 'rtwdemo_sil_topmodel';
close_system(model,0)
open_system(model)
```

Remove any existing build folders.

```
buildFolder = RTW.getBuildDir(model);
if isfolder(buildFolder.BuildDirectory)
    rmdir(buildFolder.BuildDirectory,'s');
end
```

Configure the model for coverage collection.

```
set_param(model, 'CovEnable', 'on')
clear covCumulativeData
```

Set up the input data.

```
T = 0.1; % sample time
[ticks_to_count, reset, counter_mode, count_enable, ...
    counter_mode_values_run1, counter_mode_values_run2, ...
    count_enable_values_run1, count_enable_values_run2] = ...
    rtwdemo_sil_topmodel_data(T);
```

Run the First Simulation in Normal Mode

After the simulation completes, the model coverage report opens. To navigate from blocks in the model to the corresponding sections of the coverage report, use the coverage display window.

```
counter_mode.signals.values = counter_mode_values_run1;
count_enable.signals.values = count_enable_values_run1;
set_param(model, 'SimulationMode', 'normal');
```

Use the Simulation Data Inspector to view and compare simulation results.

```
Simulink.sdi.view;
Simulink.sdi.clear;
```

Run the simulation.

```
simout_normal_run1 = sim(model, 'ReturnWorkspaceOutputs', 'on');
```

Highlight the model.

```
cvmodelview(simout_normal_run1.covdata);
```

Capture the results.

```
Simulink.sdi.createRun('Run 1 (normal mode)', 'namevalue',...
                       {'simout_normal_run1'}, {simout_normal_run1});
```

Run the Second Simulation in Normal Mode

For the first simulation, the report shows that the model achieved less than 100% MCDC coverage. Run a second simulation with different input signals to increase the level of MCDC coverage to 100%. The model coverage report is configured to show cumulative coverage across both simulation runs.

```
counter_mode.signals.values = counter_mode_values_run2;
count_enable.signals.values = count_enable_values_run2;
set_param(model, 'SimulationMode', 'normal');

simout_normal_run2 = sim(model, 'ReturnWorkspaceOutputs', 'on');

cvmodelview(simout_normal_run2.covdata);

Simulink.sdi.createRun('Run 2 (normal mode)', 'namevalue',...
                       {'simout_normal_run2'}, {simout_normal_run2});
```

Configure the Model to Measure Code Coverage

Before running a SIL simulation, configure the model to collect code coverage metrics.

```
coverageSettings = get_param(model, 'CodeCoverageSettings');
coverageSettings.CoverageTool = 'Simulink Coverage';
coverageSettings.TopModelCoverage = 'on';
set_param(model, 'CodeCoverageSettings', coverageSettings);
```

Run the First Simulation in SIL Mode

You can use the same input signals in the SIL simulation that you used during the first simulation run in normal mode.

Run the first simulation in SIL mode.

```
counter_mode.signals.values = counter_mode_values_run1;
count_enable.signals.values = count_enable_values_run1;
set_param(model, 'SimulationMode', 'software-in-the-loop');
set_param(model, 'CodeExecutionProfiling', 'off');
set_param(model, 'CodeProfilingInstrumentation', 'off');
simout_sil_run1 = sim(model, 'ReturnWorkspaceOutputs', 'on');

### Starting build procedure for: rtwdemo_sil_topmodel
### Successful completion of build procedure for: rtwdemo_sil_topmodel

Build Summary

Top model targets built:

Model                   Action                          Rebuild Reason
==================================================================================================
rtwdemo_sil_topmodel  Code generated and compiled  Code generation information file does not exis

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 37.841s
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Updating code generation report with SIL files ...
### Starting SIL simulation for component: rtwdemo_sil_topmodel
```

```
### Stopping SIL simulation for component: rtwdemo_sil_topmodel
### Completed code coverage analysis

cvmodelview(simout_sil_run1.covdata);
Simulink.sdi.createRun('Run 1 (SIL mode)', 'namevalue',...
                       {'simout_sil_run1'}, {simout_sil_run1});
```

When the simulation completes, view the code coverage results on the model by using coverage highlighting. To see the SIL code coverage summary for a model element, place your cursor over the model element.
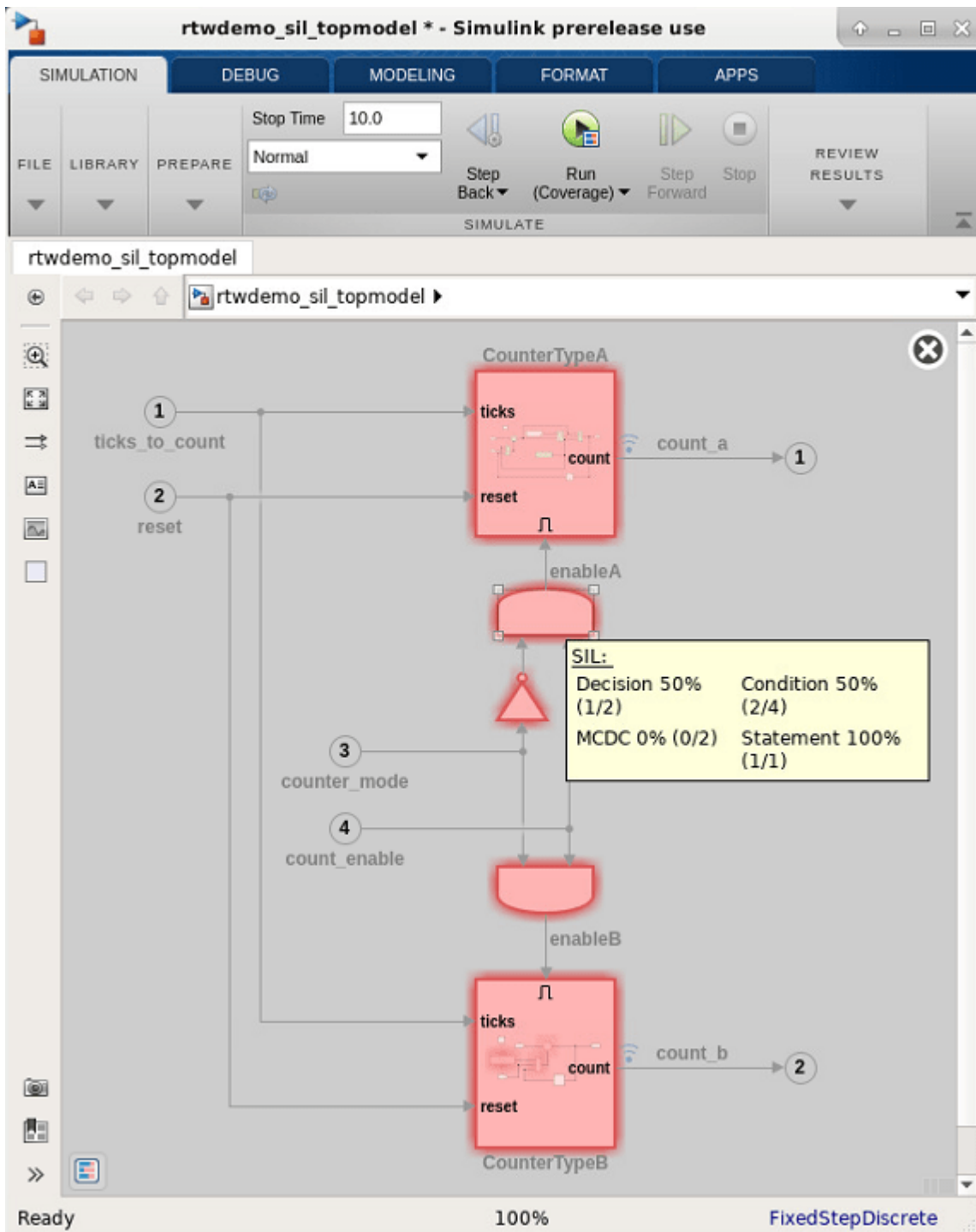
You can also view the code coverage results in the HTML code coverage report. The summary section shows that all functions have been called, but the SIL simulation run did not achieve full coverage for decision, condition, or MCDC coverage.

```
cvhtml('codecovreport',simout_sil_run1.covdata);
```
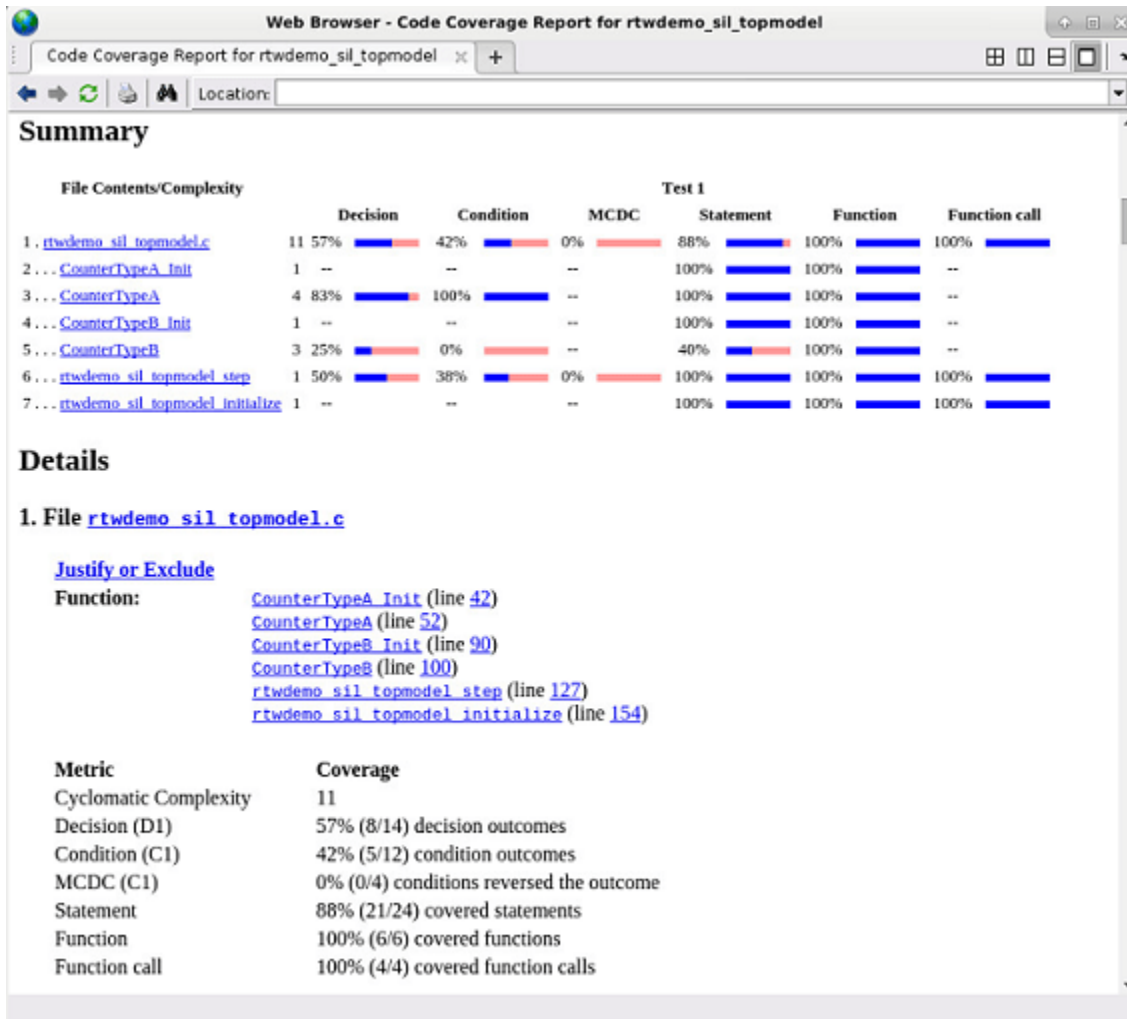
To navigate to the corresponding model elements in the block diagram, use the hyperlinks in the code coverage report

Run the Second Simulation in SIL mode

Use the same input signals in the SIL simulation that you used in the second simulation run in normal mode.

```
counter_mode.signals.values = counter_mode_values_run2;
count_enable.signals.values = count_enable_values_run2;
set_param(model, 'SimulationMode', 'software-in-the-loop');
set_param(model, 'CodeExecutionProfiling', 'off');
set_param(model, 'CodeProfilingInstrumentation', 'off');
simout_sil_run2 = sim(model, 'ReturnWorkspaceOutputs', 'on');

### Starting build procedure for: rtwdemo_sil_topmodel
### Generated code for 'rtwdemo_sil_topmodel' is up to date because no structural, parameter or o
### Successful completion of build procedure for: rtwdemo_sil_topmodel

Build Summary
```

```
Top model targets built:

Model                    Action          Rebuild Reason
==============================================================================
rtwdemo_sil_topmodel  Code compiled  Compilation artifacts were out of date.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 4.417s
### Preparing to start SIL simulation ...
### Starting SIL simulation for component: rtwdemo_sil_topmodel
### Stopping SIL simulation for component: rtwdemo_sil_topmodel
### Completed code coverage analysis
```

```
Simulink.sdi.createRun('Run 2 (SIL mode)', 'namevalue',...
                       {'simout_sil_run2'}, {simout_sil_run2});
```

Highlight the model using the cumulative coverage from both SIL mode simulations to see that the generated code from the model achieved full coverage.

```
cvmodelview(simout_sil_run1.covdata + simout_sil_run2.covdata);
```
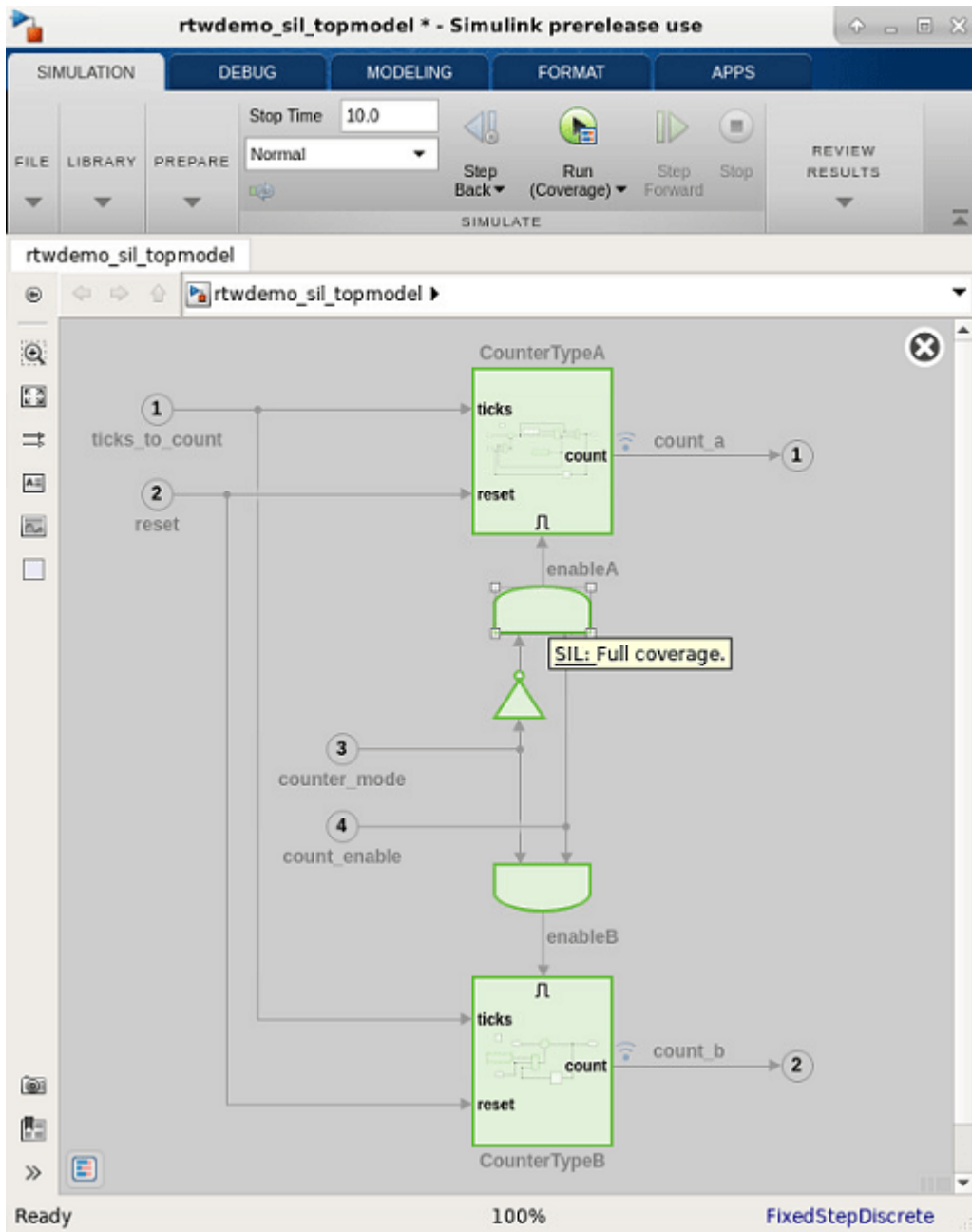
Compare Metrics from the Normal and SIL Simulations

The Simulation Data Inspector opens automatically after each run, which allows you to view and analyze the results. To confirm that the logged signals for the SIL and normal mode runs are identical, review the information in the Compare and Inspect panes.

Specify Code Coverage Options

Simulink Coverage provides three modes of code coverage analysis. For general coverage options, see "Specify Coverage Options" on page 3-2.

In this section...
"Models with S-Function Blocks" on page 4-17
"Models with Software-in-the-Loop and Processor-in-the-Loop Mode Blocks" on page 4-17
"Models with MATLAB Function Blocks" on page 4-18

Models with S-Function Blocks

Configure an S-Function block for coverage based on how you created it. For more information, see "Coverage for Custom C/C++ Code in Simulink Models" on page 5-59.

Note If you have software-in-the-loop or processor-in-the-loop blocks in your model, set the options described in "Models with Software-in-the-Loop and Processor-in-the-Loop Mode Blocks" on page 4-17.

Models with Software-in-the-Loop and Processor-in-the-Loop Mode Blocks

1   Open the Configuration Parameters. In the **Modeling** tab, click **Model Settings**.

2   Before setting code coverage options, on the **Code Generation** pane in the Configuration Parameters dialog box, set the **System target file** in the **Target selection** menu to `ert.tlc`.

3   In the Configuration Parameters dialog box, on the left pane, click **Code Generation**. From the list, select **Verification**.

4   Select the code coverage tool from the **Code coverage for SIL or PIL** tab.

You can measure code coverage using these tools:

- Simulink Coverage code coverage tool
- BullseyeCoverage
- LDRA tool suite

BullseyeCoverage and the LDRA tool suite are third-party tools supported by Embedded Coder. For more information on third-party code coverage tool support, see "Code Coverage Tool Support" (Embedded Coder). To set code coverage options, click **Configure**. If you select `None (use Simulink Coverage)` as the code coverage tool, the software opens the **Coverage** pane when you click **Configure**.

Using Simulink Coverage for code coverage means that you can analyze coverage results, justify missing coverage, and generate more test cases from within the Simulink environment.

Models with MATLAB Function Blocks

When you record coverage for models containing MATLAB Function blocks, code coverage is recorded for the code within the MATLAB Function blocks. To include MATLAB Function blocks in your analysis:

1 In the Simulink Editor, select **Model Settings** on the **Modeling** tab.

2 In the Configuration Parameters dialog box, on the **Coverage** pane, under **Include in analysis**, select **MATLAB files**.

See Also

More About

- "Create and Run Test Cases" on page 5-2
- "Types of Coverage Reports" on page 6-2
- "View Coverage Results for Custom C/C++ Code in S-Function Blocks" on page 5-61
- "Coverage Filtering" on page 7-2

Coverage for Models with Code Blocks and Simulink Blocks

In this section...
"Set Up the Model to Record Coverage" on page 4-19
"Record Coverage" on page 4-20
"Review Results by Generating a Coverage Report" on page 4-20
"Justify Missing Coverage" on page 4-20

In this example, you record coverage for a model which contains a combination of code blocks and other Simulink blocks.

Set Up the Model to Record Coverage

1   Open the model.

```
open_system('ex_cc_cruise_control_doublepress_sfun');
```
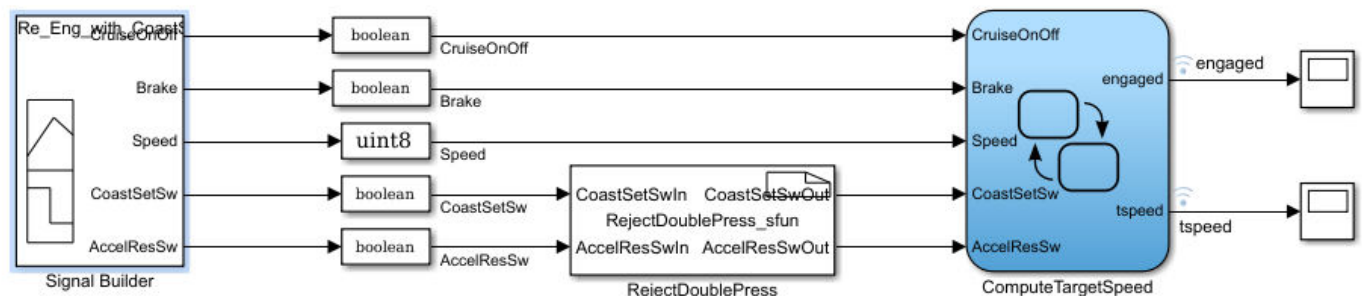

The model is a cruise control system that consists of test cases and input signals from a Signal Builder block. The signals from the Signal Builder act as inputs to the Stateflow chart `ComputeTargetSpeed`, which engages or disengages the cruise control system and sets the target speed, `tspeed`.

2   In the Simulink Editor, select **Model Settings** on the **Modeling** tab. Before setting code coverage options, on the **Code Generation** pane in the Configuration Parameters dialog box, set the **System target file** in the **Target selection** menu to `ert.tlc`. Navigate to the **Verification** tab of the **Code Generation** pane. From the **Code coverage for SIL or PIL** tab, select `None (use Simulink Coverage)` as the code coverage tool.

3   In the **Coverage** pane, set the options for coverage calculated during simulation.

   1   Select **Enable coverage analysis**.

   2   In the **Include in analysis** section, ensure that **C/C++ S-Functions** is selected.

   3   In the **Coverage metrics** section, select `Modified Condition Decision (MCDC)` as the **Structural coverage level**. Apply the changes by clicking **Apply**.

4   Open the `RejectDoublePress` S-Function Builder block. In the **Build options** of the **Build Info** tab, select **Enable support for coverage**. To build the S-Function, click **Build** .

Note   To build the S-Function, you must have a compiler installed. For more information on supported compilers for various platforms, see Supported and Compatible Compilers.

Record Coverage

1 Open the Signal Builder block.

```
open_system('ex_cc_cruise_control_doublepress_sfun/Signal Builder');
```
2 The Signal Builder consists of eight signal groups with five signals each. In this example, we

simulate all the signal groups and record coverage. Click ▶ **Run all and produce coverage** to start recording coverage. At the end of the simulation, the Coverage Results Explorer opens, showing the results for the latest coverage analysis. The blocks in the model are highlighted in different colors corresponding to the level of coverage achieved by each block.

Review Results by Generating a Coverage Report

The Coverage Results Explorer offers several options for displaying and reporting coverage results. Select the `Not_Engaged_with_Enable` group in the **Current Cumulative Data** tab of the left pane. Click the **Generate report** link at the bottom of the Coverage Results Explorer to generate an HTML coverage report in the built-in MATLAB web browser. The coverage report lists model coverage for Simulink model blocks and code coverage for code blocks.

Scroll down to view the coverage metrics for the S-Function block in the coverage report. Click the **Detailed Report** link to open the code coverage report for the S-Function block. For more details on the code coverage report for S-Function blocks, see "View Coverage Results for Custom C/C++ Code in S-Function Blocks" on page 5-61.

Justify Missing Coverage

In this example, we justify coverage for one input signal group by creating a coverage filter. In the code coverage report for the S-Function block created in "Review Results by Generating a Coverage Report" on page 4-20, scroll down to Decision/Condition 2.1 `!(CoastSetSwIn[0] && AccelResSwIn[0])`. This condition is never `False` for the current test case. We can therefore justify this condition in our coverage analysis.

1 Click the **Justify or Exclude** link under the detailed results for this condition. The **Filter** tab of the Coverage Results Explorer opens, and the rule filtering this transition is added. Change the **Mode** for this rule to `Justified` and enter a description for the **Rationale**, such as "expression cannot be false". Click **Apply** to apply the changes.
2 After you click **Apply**, the **Generate report** link becomes available. Click the link to generate the report with the updated coverage filter. The new code coverage report for the `RejectDoublePress` S-Function block lists the excluded condition under **Objects Filtered from Coverage Analysis**. The detailed results for the condition `!(CoastSetSwIn[0] && AccelResSwIn[0])` show that missing coverage for this condition has been justified. The justified objects are treated as satisfied when reporting coverage percentages and appear light blue in the "Coverage Summary" on page 6-12.

Summary

File Contents/Complexity		Current Run				Delta				Cumulative			
		Decision	Condition	MCDC	Statement	Decision	Condition	MCDC	Statement	Decision	Condition	MCDC	Statement
1. RejectDoublePress_sfun_wrapper.c	1	67%	67%	17%	100%	33%	33%	33%	0%	100%	83%	67%	100%
2. ... RejectDoublePress_sfun_Outputs_wrapper	1	67%	67%	17%	100%	33%	33%	33%	0%	100%	83%	67%	100%

For more information on coverage filters, see "Coverage Filtering" on page 7-2.

See Also

"Types of Coverage Reports" on page 6-2 | "Creating and Using Coverage Filters" on page 7-11 | "Coverage for Custom C/C++ Code in Simulink Models" on page 5-59

Software-in-the-Loop Code Coverage

This example shows how to use a model reference in either SIL or Normal simulation mode to collect model or code coverage metrics with Simulink® Coverage™.

Model Reference SIL Code Coverage Example
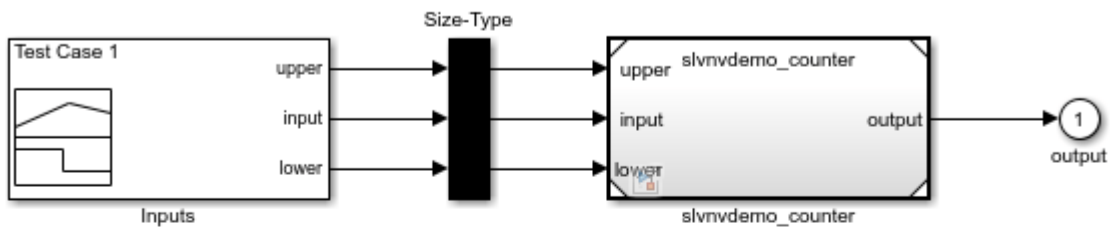
This example shows how to use a model reference in either SIL or Normal simulation mode to collect model or code coverage metrics with Simulink Coverage.

To enable model coverage, double-click on the left blue box "Configure for Normal Mode" and then run a simulation. To enable SIL code coverage, double-click on the right blue box "Configure for SIL Mode" and then run a simulation.

Note: To run this example, you must have Embedded Coder installed.

Configure
for Normal Mode
(double-click)

Configure for Model Coverage
in Normal Mode

Configure
for SIL Mode
(double-click)

Configure for Model Coverage
in SIL Mode

Copyright 2015 The MathWorks, Inc.

Use Justification Rules to Filter Code Coverage Outcomes

This example shows how to filter code coverage outcomes in the coverage report after collecting coverage for a model in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode.
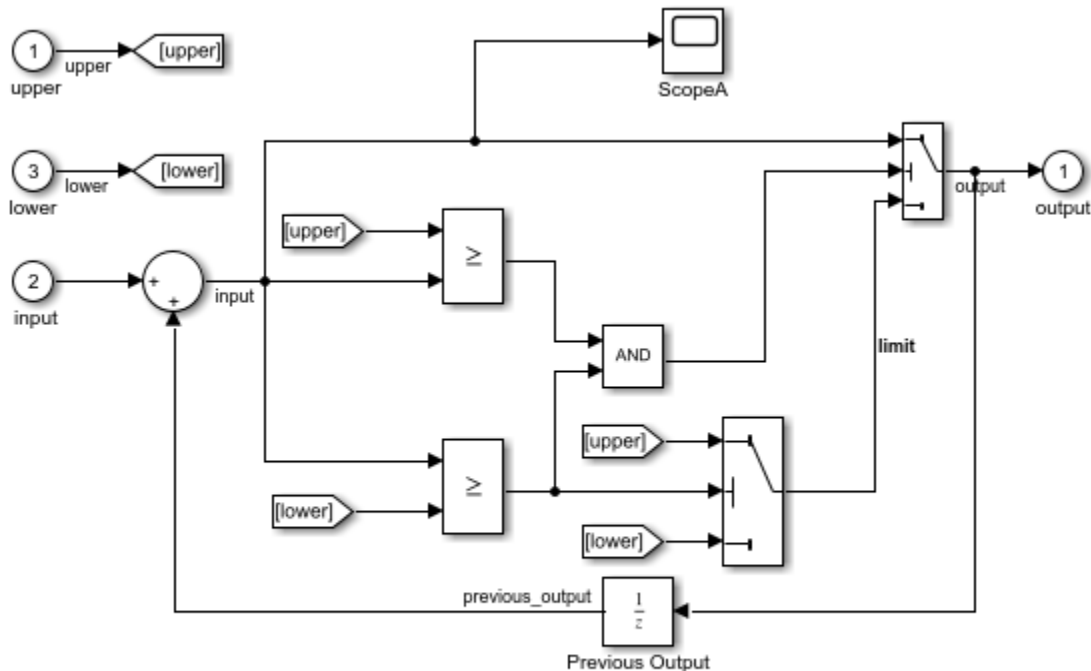
Copyright 2015 The MathWorks, Inc.

Generate Code Coverage Data

First, put the model into SIL/PIL mode. In the Simulink® window, click **Apps** and, under **Code Verification, Validation, and Test**, click **SIL/PIL Manager**. On the **SIL/PIL** tab, change **Automated Verification** to **SIL/PIL Simulation Only**.

In this example model, coverage is enabled by default. If you are using your own model, enable coverage in the **Configuration Parameters** window. For more information about coverage settings, see "Specify Coverage Options" on page 3-2.

Simulate the model and collect coverage by clicking **Run SIL/PIL**. When you simulate the model, a docked pane opens next to the Simulink® model. Click the **Coverage Details** tab to see the code coverage report.

```
### Starting build procedure for: slvnvdemo_counter
### Successful completion of build procedure for: slvnvdemo_counter

Build Summary

Top model targets built:

Model                Action                    Rebuild Reason
========================================================================================================
slvnvdemo_counter  Code generated and compiled  Code generation information file does not exist.

1 of 1 models built (0 models already up to date)
Build duration: 0h 0m 55.067s
### Preparing to start SIL simulation ...
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Updating code generation report with SIL files ...
### Starting SIL simulation for component: slvnvdemo_counter
### Stopping SIL simulation for component: slvnvdemo_counter
### Completed code coverage analysis
```

Justify Missing Code Coverage Using Coverage Filters

If your model has unreachable logic that is intentional, such as defensive model design or exception handling, you can justify this missing coverage using coverage filters.

The **Summary** section of the code coverage report links to each source file and function. In this example, click `slvnvdemo_counter_step`. The code coverage report jumps to the function named `slvnvdemo_counter_step`. In section 2.1, you can see that both conditions inside the decision (!(slvnvdemo_counter_U.upper >=z rtb_input)) || (!rtb_inputGElower)| are false for all time steps.

2.1. Decision/Condition (!(slvnvdemo_counter_U.upper >= rtb_input)) || (!rtb_inputGElower) (line 58)

Justify or Exclude

Function:	slvnvdemo_counter_step
Model Objects:	And, Switch, upper GE input
Uncovered Links:	← →

Metric	Coverage
Decision	50% (1/2) decision outcomes
Condition	50% (2/4) condition outcomes
MCDC	0% (0/2) conditions reversed the outcome

Decisions analyzed

(!(slvnvdemo_counter_U.upper >= rtb_input)) || (!rtb_inputGElower)	50%
false	51/51
true	0/51 ⊞

Conditions analyzed

Description	True	False
slvnvdemo_counter_U.upper >= rtb_input	51	0 ⊞
rtb_inputGElower	51	0 ⊞

To justify a missing coverage outcome, in the **Conditions analyzed** table, next to the condition `slvnvdemo_counter_U.upper >= rtb_input`, click the **Add justification rule** icon. The Coverage Results Explorer opens and creates a filter rule to justify the `false` outcome of the condition `slvnvdemo_counter_U.upper >= rtb_input`.
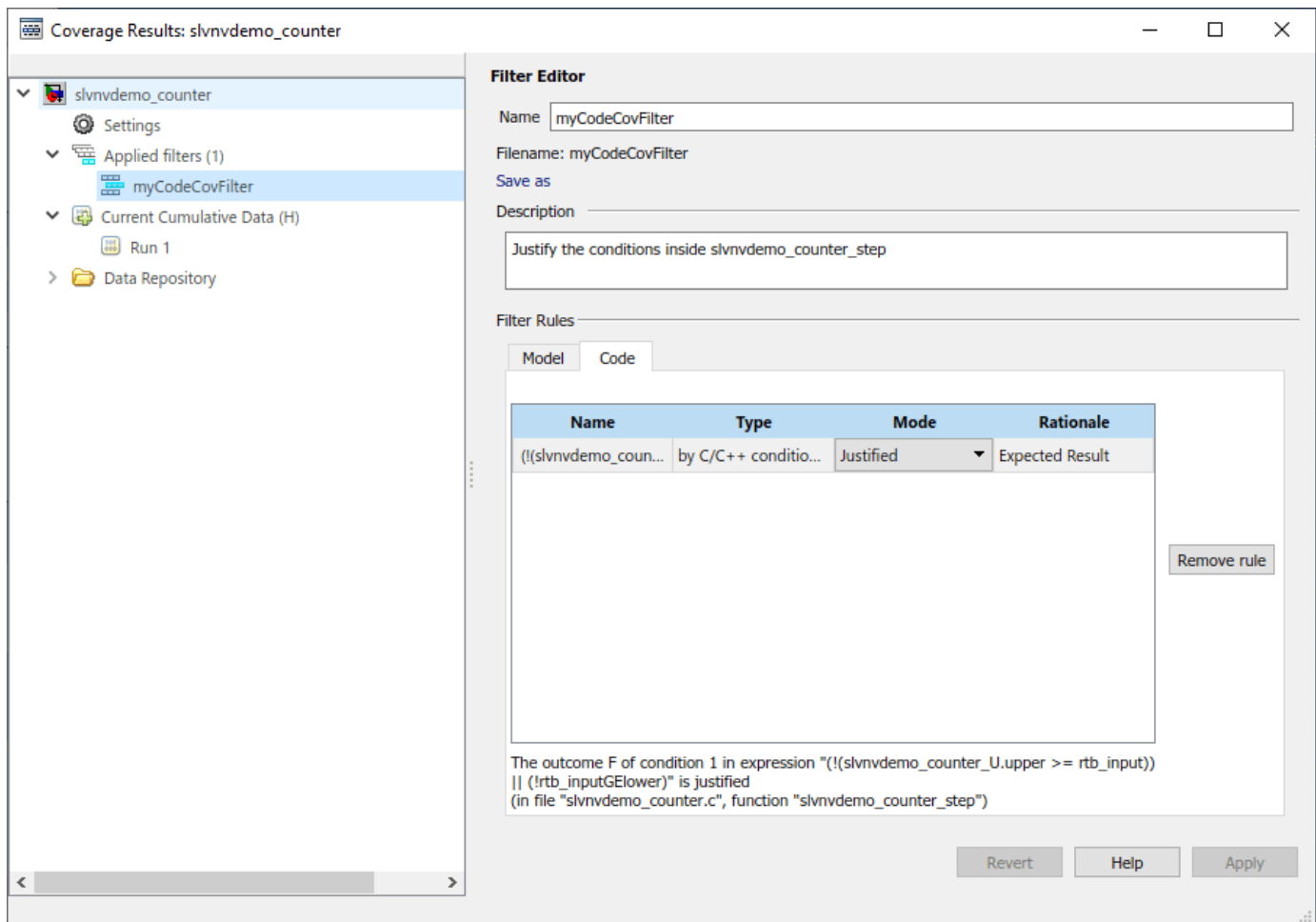
In the Filter Editor pane, set the **Name** field to `myCodeCovFilter`. You can set the **Description** field to any descriptive text. The **Filter Rules** section has two tabs, **Model** and **Code**. In this case, the filter appears on the **Code** tab because you are filtering from the code coverage report. You can double-click the **Rationale** field to add a reason, for example "Expected result."

Near the top of the Filter Editor, under the **Filename** field, click **Save as**. In the Save filter window, name the filter file `myCodeCovFilter`. Note that the filter name and the filter file name do not have to be the same.

When you save the filter, the code coverage report updates and displays the justified outcome.

Conditions analyzed

Description	True	False
slvnvdemo_counter_U.upper >= rtb_input	102 T1	J1.
rtb_inputGElower	102 T1	0

Justify the false case of the second condition by clicking the **Add justification rule** filter next to `rtb_inputGElower` and following the steps listed above. This second rule is added to the same filter file that you created for the first rule.
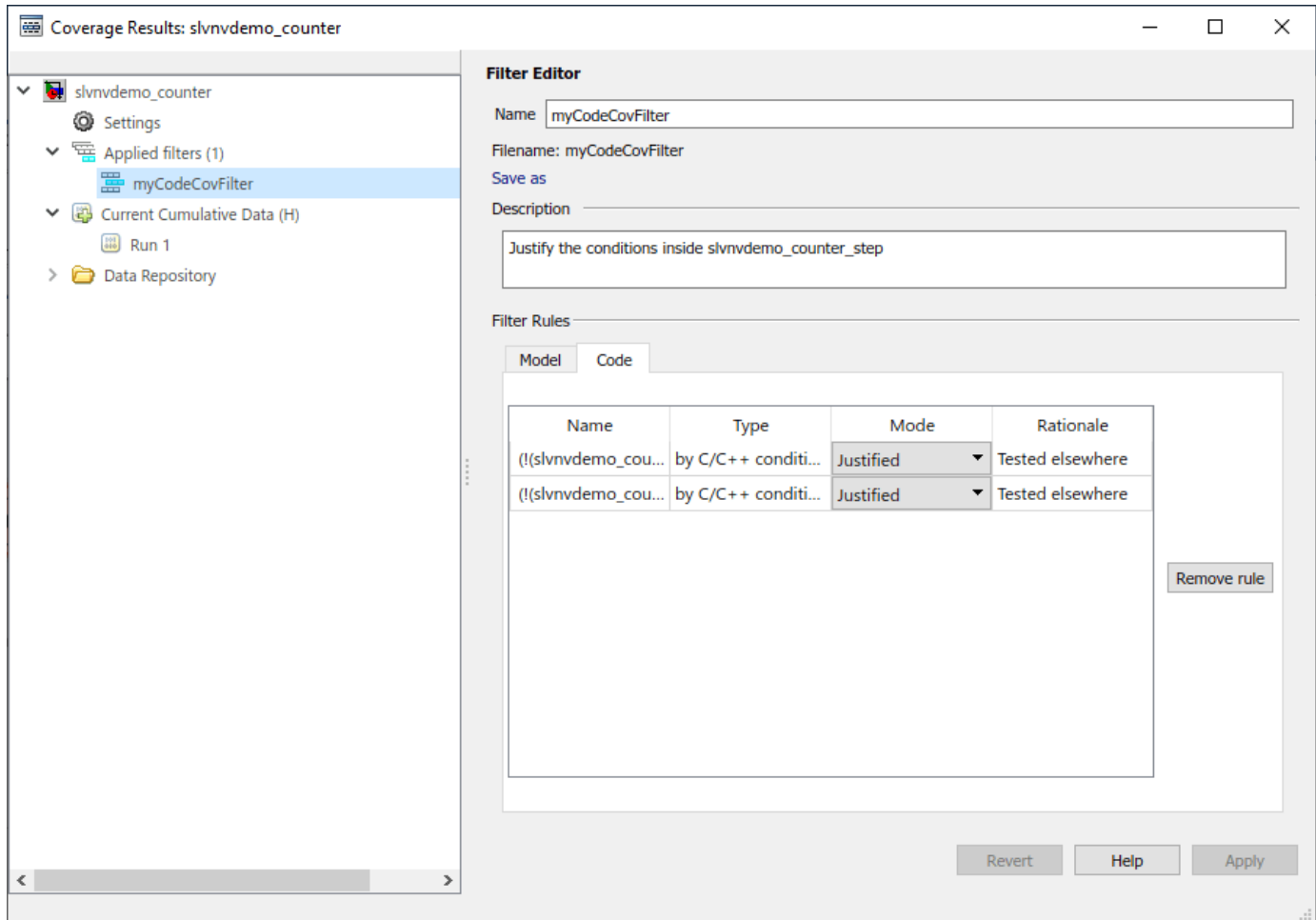
Conditions analyzed

Description	True	False
slvnvdemo_counter_U.upper >= rtb_input	102 T1	J1.
rtb_inputGElower	102 T1	J2.

You can create a new code coverage report after applying coverage filters by clicking **Current Cumulative Data (H)** in the left pane of the Coverage Results Explorer, and then clicking **Generate report** at the bottom of the **Coverage Data** pane. This link creates a standalone report which you can use for archiving or sharing reports.

Coverage Results: slvnvdemo_counter — □ ✕

- ∨ 🖳 slvnvdemo_counter
  - ⚙ Settings
  - ∨ 🗂 Applied filters (1)
    - 🗂 myCodeCovFilter
  - ∨ 🗐 Current Cumulative Data (H)
    - 🔢 Run 1
  - \> 📁 Data Repository

Coverage Data

Collected in version	(R2021a)
Model version	4.0
Author	The MathWorks, Inc.
Started execution	09-Jan-2021 21:03:09
File name:	active

Description

Tag: Run 1

Summary

File Contents/Complexity		Test 1				
		Decision	Condition	MCDC	Statement	Function
1 . slvnvdemo_counter.c	5	25%	83%	0%	84%	100%
2 ... slvnvdemo_counter_step	3	25%	83%	0%	73%	100%
3 ... slvnvdemo_counter_initialize	1	--	--	--	100%	100%
4 ... slvnvdemo_counter_terminate	1	--	--	--	100%	100%

Last report: slvnvdemo_counter_active_cov

Remove highlight

[Revert] [Help] [Apply]

The summary section of the code coverage report reflects the improved condition coverage due to the filter rules.

Summary

File Contents/Complexity		Test 1				
		Decision	Condition	MCDC	Statement	Function
1 . slvnvdemo_counter.c	5	25%	83%	0%	84%	100%
2 ... slvnvdemo_counter_step	3	25%	83%	0%	73%	100%
3 ... slvnvdemo_counter_initialize	1	--	--	--	100%	100%
4 ... slvnvdemo_counter_terminate	1	--	--	--	100%	100%

Additionally, the code coverage report now shows a section titled **Objects Filtered from Coverage Analysis** that displays the filter rules and rationales.

Objects Filtered from Coverage Analysis

Filter myCodeCovFilter

File   myCodeCovFilter.cvf

Description Justify the conditions inside slvnvdemo_counter_step

Code	Rationale
J1. The outcome F of condition 1 in decision (`!(slvnvdemo_counter_U.upper >= rtb_input)) || (!rtb_inputGElower`) (line 58)	Tested elsewhere
J2. The outcome F of condition 2 in decision (`!(slvnvdemo_counter_U.upper >= rtb_input)) || (!rtb_inputGElower`) (line 58)	Tested elsewhere

Coverage Collection During Simulation

Create and Run Test Cases

To create and run test cases, model coverage provides the MATLAB commands `cvtest` and `cvsim`. The `cvtest` command creates test cases that the `cvsim` command runs.

You can also run the coverage tool interactively:

1 Open the `ExtractingDetailedCoverageDataExample` example using `openExample`.

```
openExample('slcoverage/ExtractingDetailedCoverageDataExample');
```

2 Open the `slvnvdemo_cv_small_controller` model.

3 In the Simulink Editor, select **Model Settings** on the **Modeling** tab.

In the Configuration Parameters dialog box, on the "Coverage Pane" on page 3-2, select **Enable coverage analysis**, which enables the coverage settings.

4 Under **Coverage metrics**, select the types of coverage that you want to record in the coverage report. Click **OK**.

5 Simulate the model.

Simulink Coverage saves coverage data for the current run in the workspace object `covdata` and cumulative coverage data in `covCumulativeData`, by default if you simulate using the **Run** button. Simulink Coverage also saves these results to a `.cvt` file by default. At the end of the simulation, the data appears in an HTML report that opens next to your model. For more information on coverage data settings, see "Specify Coverage Options" on page 3-2.

You cannot run simulations if you select both the model coverage reporting and acceleration options. If you set the simulation mode to **Accelerator**, Simulink Coverage does not record coverage.

When you perform coverage analysis, you cannot select both block reduction and conditional branch input optimization, because they interfere with coverage recording. See "Simulink Optimizations and Model Coverage" on page 1-9 for more information.

Modified Condition and Decision Coverage (MCDC) Definitions in Simulink Coverage

Simulink Coverage by default uses the masking modified condition and decision coverage (MCDC) definition for recording MCDC coverage results. Although you can change the MCDC definition that Simulink Coverage uses during analysis to the unique-cause MCDC definition, there are some differences in how Simulink Coverage records coverage for models depending on which definition you use.

In this section...
"Differences between Masking MCDC and Unique-Cause MCDC in Simulink Coverage Coverage Analysis" on page 5-3
"Certification Considerations for MCDC Coverage" on page 5-4
"Setting the (MCDC) Definition Used for Simulink Coverage Coverage Analysis" on page 5-4
"Modified Condition and Decision Coverage in Simulink Design Verifier" on page 5-5

Differences between Masking MCDC and Unique-Cause MCDC in Simulink Coverage Coverage Analysis

Masking MCDC accounts for the masking of conditions in subexpressions, allowing for an increased number of satisfied MCDC objectives compared to the unique-cause definition of MCDC. As a result, some Simulink models that receive less than complete MCDC coverage using the unique-cause MCDC definition receive increased coverage when using the masking MCDC definition. Consider the following example, where two inputs to a Stateflow chart, condition A and condition C, cannot change independently:
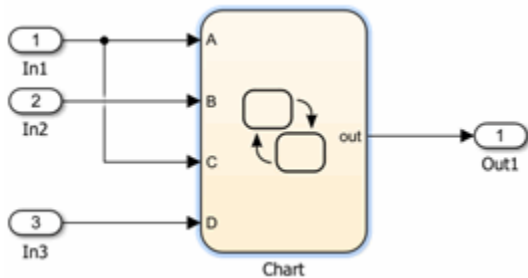
This input dependence results in dependent conditions for the expression contained within the Stateflow chart:
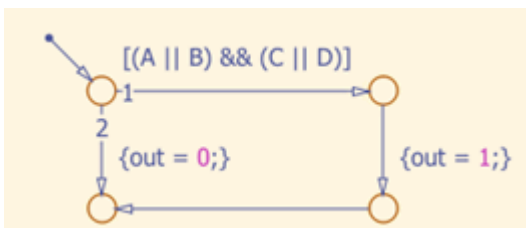
For the expression (A||B)&&(C||D), changing the value of condition C also changes the value of condition A. Due to the interdependence of conditions A and C, unique-cause MCDC for condition C cannot be achieved:

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition	True Out	False Out
(A ‖ B) && (C ‖ D)		
A	TxTx	FFxx
B	FTFT	FFxx
C	TxTx	(TxFF)
D	FTFT	FTFF

However, masking MCDC for condition C can be achieved, because masking MCDC allows the value of condition A to change in the independence pair for condition C, as long as the subexpression (A‖B) remains true:

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition	True Out	False Out
(A ‖ B) && (C ‖ D)		
A	TxTx	FFxx
B	FTFT	FFxx
C	TxTx	FTFF
D	FTFT	FTFF

Certification Considerations for MCDC Coverage

DO-248C Discussion Paper #13 "Discussion of Statement Coverage, Decision Coverage and Modified Condition/Decision Coverage" states that masking MCDC is acceptable for meeting the MCDC objective of DO-178B certification.

Setting the (MCDC) Definition Used for Simulink Coverage Coverage Analysis

By default, Simulink Coverage uses the masking MCDC definition during coverage analysis. There are two ways to change the MCDC definition used for Simulink Coverage coverage analysis:

Use the Model Configuration Parameters to Set the MCDC Definition Used

1 Open the **Configuration Parameters** dialog box.

2 Set the `CovMcdcMode` parameter to `Masking` or `Unique-Cause`.

Use the cvtest Object to Set the MCDC Definition Used

Create a `cvtest` object for your model to set the `mcdcMode` to `'Masking'` or `'UniqueCause'`:

```
cvt = cvtest(model)
cvt.options.mcdcMode = 'UniqueCause'
covdata = cvsim(cvt)
```

Modified Condition and Decision Coverage in Simulink Design Verifier

Setting `CovMcdcMode` to `'UniqueCause'` can result in differences between MCDC reporting in Simulink Coverage and test generation in Simulink Design Verifier. Simulink Design Verifier always uses the masking MCDC definition for test case generation. For more information, see "Modified Condition and Decision Coverage in Simulink Design Verifier" on page 5-6.

See Also

More About

- "MCDC" (Simulink Design Verifier)

Modified Condition and Decision Coverage in Simulink Design Verifier

Depending on the settings you apply for Simulink Coverage coverage recording, there can be a difference between the definition of modified condition and decision (MCDC) coverage used for model coverage analysis in Simulink Coverage and the definition used for test case generation analysis in Simulink Design Verifier.

MCDC Definitions for Simulink Coverage and Simulink Design Verifier

Simulink Design Verifier and Simulink Coverage represent MCDC objectives in two different ways:

- Simulink Coverage treats each condition of a logical expression as an MCDC objective.
- Simulink Design Verifier treats the true and false halves of each independence pair as separate MCDC objectives.

The Simulink Design Verifier Results window shows **Justified** for any justified MCDC objectives. Click on the corresponding **View** link to see the filter rule in the Simulink Design Verifier Analysis Filter window.

Unsatisfiable or undecided MCDC objectives include a **Justify** link. Click on this link to create a corresponding filter rule. Because every MCDC objective in Simulink Coverage corresponds to two MCDC objectives in Simulink Design Verifier, the Simulink Design Verifier MCDC objectives are justified in pairs.

For example, in the image below, when you click on the **Justify** link for the MCDC expression `expression for output with input port 4` **false**, creates a filter rule that justifies this MCDC objective as well as the MCDC objective for when that expression is **true**.

Simulink Design Verifier always uses the masking MCDC definition for test case generation. By default, Simulink Coverage also uses the masking MCDC definition when recording coverage. However, if you set the `CovMcdcMode` model configuration parameter to `'UniqueCause'`, Simulink Coverage instead uses the unique-cause MCDC definition when recording coverage. For information

on the differences between the masking MCDC definition and the unique-cause MCDC definition, see "Modified Condition and Decision Coverage (MCDC) Definitions in Simulink Coverage" on page 5-3.

Setting the `CovMcdcMode` model configuration parameter to `'UniqueCause'` can result in differences between MCDC reporting in Simulink Coverage and test generation in Simulink Design Verifier. An example of this difference can be seen in analysis results for logical expressions containing a mixture of `AND` and `OR` operators, as in this Stateflow transition.
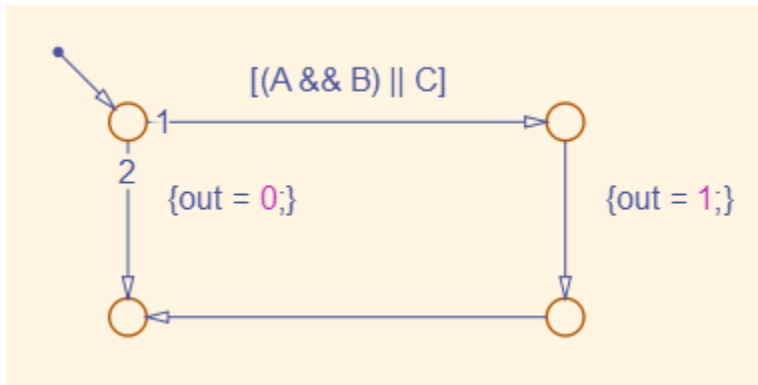
Given that A, B, and C are each separate inputs, there are five possible ways to evaluate the condition on the Stateflow transition, shown in the following table.

	A	B	C	(A && B) || C
1	F	x	F	F
2	F	x	T	T
3	T	F	F	F
4	T	F	T	T
5	T	T	x	T

Satisfying MCDC for a Boolean variable requires a pair of condition evaluations, showing that a change in that variable alone changes the evaluation of the entire expression. In this example, MCDC can be satisfied for C with either the pair 1, 2 or the pair 3, 4. In both of those cases, the value of the expression changed because the value of C changed, while all other variable values stayed the same.

Each pair has a different set of values for A and B which are held constant, but each pair contains one evaluation where C and `out` are true and one evaluation where C and `out` are false. To satisfy MCDC for C, Simulink Design Verifier test generation analysis accepts any pair containing one evaluation of true values and one evaluation of false values for C and `out`. In this example, Simulink Design Verifier test generation analysis accepts not only pair 1, 2 and pair 3, 4 but also pair 1, 4 and pair 2, 3. Simulink Coverage model coverage analysis using the unique-cause MCDC definition is satisfied only by pair 1, 2 or by pair 3, 4.

The preceding example assumes that A, B, and C are all separate inputs. When input A is constrained to be the same value as C, as in this model, only a subset of condition evaluations are possible.
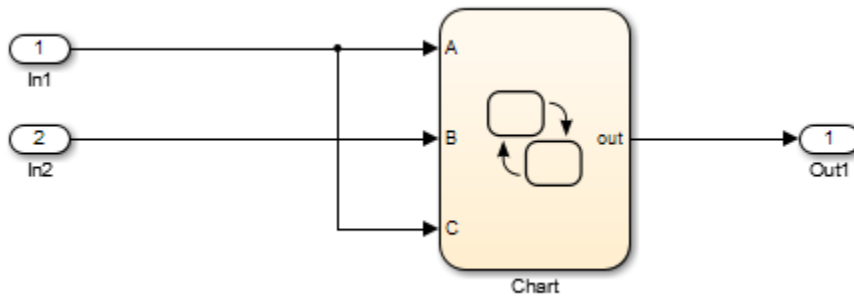
This subset of condition evaluations for the Stateflow transition is shown in the following table.

	A	B	C	(A && B) || C
1	F	x	F	F
4	T	F	T	T
5	T	T	x	T

Evaluations 2 and 3 are no longer possible, so neither pair 1, 2 nor pair 3, 4 is possible. As a result, unique-cause MCDC for C can no longer be satisfied in Simulink Coverage model coverage analysis. Since pair 1, 4 is still possible, however, Simulink Design Verifier test generation analysis reports that MCDC for C is satisfiable.

The complexity of MCDC analysis for logical expressions with a mixture of AND and OR operators causes this difference between results from Simulink Coverage set to unique-cause MCDC analysis and Simulink Design Verifier. The defaultCovMcdcMode model configuration parameter value of `'Masking'` does not cause this discrepancy. However, if you require the use of unique-cause MCDC analysis in Simulink Coverage, you can minimize this effect by using the IndividualObjectives test suite optimization for test generation analysis in Simulink Design Verifier For more information, see the Tip section of "Test suite optimization" (Simulink Design Verifier).

See Also

More About

- "MCDC" (Simulink Design Verifier)

Logical Operator Cascade Patterns

This model includes various patterns of cascaded Logical Operator blocks. This example illustrates the criteria by which logic block cascades are identified for the purpose of model coverage analysis for the MCDC metric.

Logical Operator Cascade Patterns

This model includes various patterns of cascaded Logical Operator blocks. This example illustrates the criteria by which logic block cascades are identified for the purpose of model coverage analysis for the MCDC metric.

Simulate the model to generate a coverage report. Review the MCDC results and note how certain blocks are combined based on the criteria described below.

XOR cannot be included in cascade for MCDC. These blocks will be analyzed individually for MCDC.

Signal branches. Downstream blocks won't be included in this cascade.

Cascade may cross boundaries of virtual subsystems and virtual connections.

Cascade 1

~(C1 && C2 && C3)

Cascade 2

~(C1 || C2) && C3

AND Block has a vector input so will not be included in the cascade.

For a Logical Operator block to receive MCDC as part of a logic cascade, the following conditions must be met:
- Block input and output signals are all scalar
- Logic block operation is not XOR/NXOR
- Logic block has more than one input signal (unless its operation is NOT)

Copyright 2016-2021 The MathWorks Inc.

Analyzing MCDC for Cascaded Logic Blocks

This example illustrates how Simulink® Coverage™ records the MCDC metric for a cascade of Logical Operator blocks.

Example Model

In Simulink, there are various ways to implement Boolean logic, such as through the use of an `if` statement in a MATLAB Function block, a conditional transition in a Stateflow Chart, or a combination of multiple Logical Operator blocks connected together in a cascade.

The example model `slvnvdemo_cv_logic_cascade` implements the same Boolean expression through the use of MATLAB code in a MATLAB Function block as well as with a cascade of Logical Operator blocks.

Use the following command to open the model `slvnvdemo_cv_logic_cascade`:
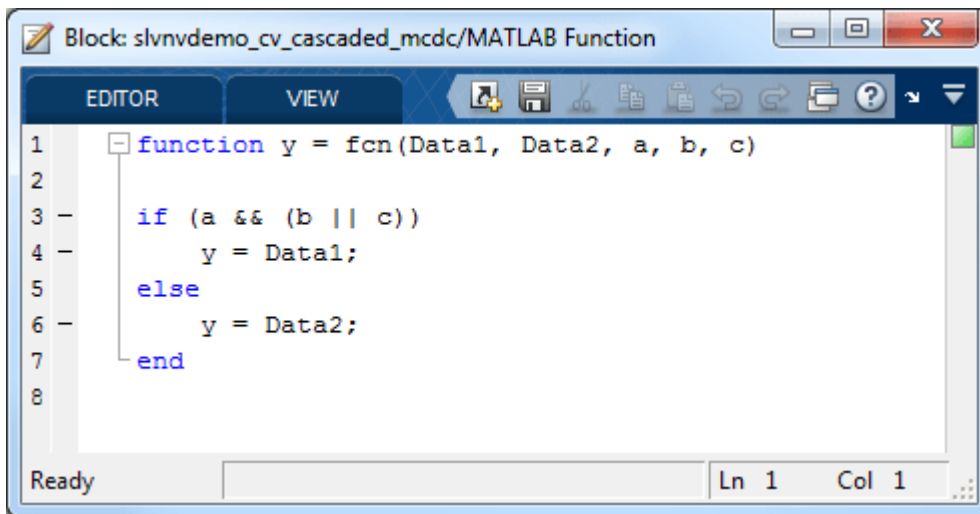
`open_system('slvnvdemo_cv_logic_cascade');`
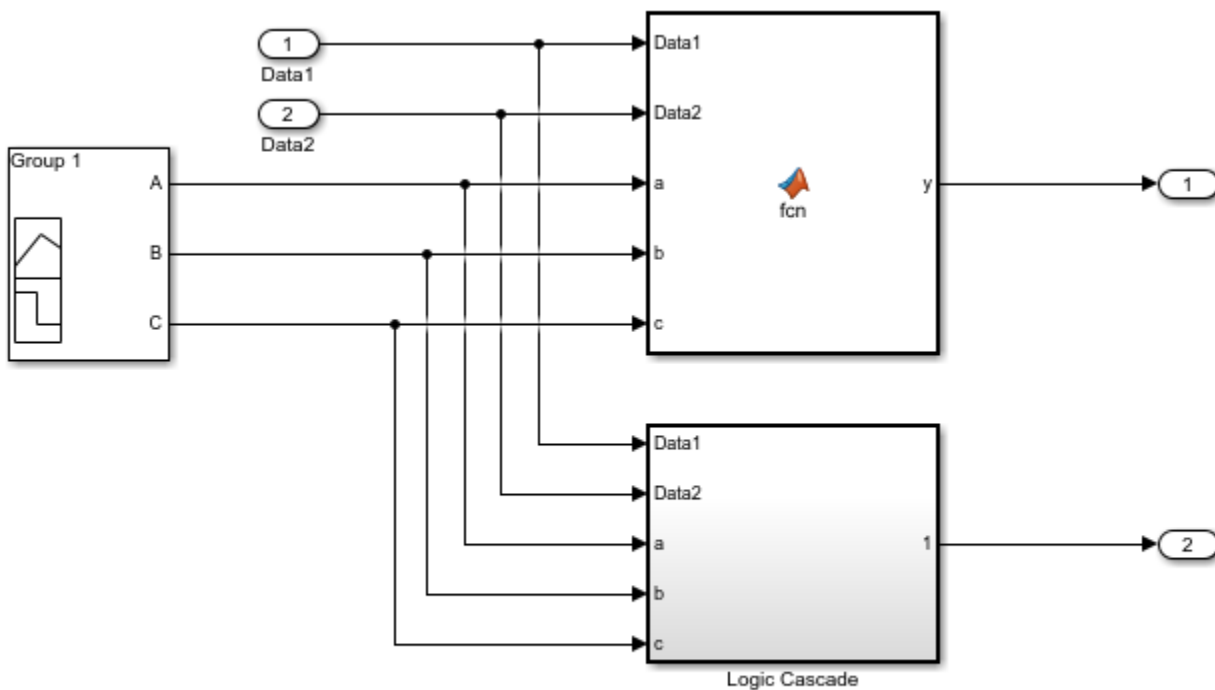
Copyright 2016 The MathWorks Inc.

Open the MATLAB Function block to see the associated function.

`open_system('slvnvdemo_cv_logic_cascade/MATLAB Function')`

In the MATLAB Function block, if *(a && (b || c))* is true, then the signal **Data1** will be output; otherwise, the signal **Data2** is output.

Open the subsystem 'Logic Cascade' using the following command and note that this subsystem implements the exact same logic using Logical Operator blocks and a Switch.

```
open_system('slvnvdemo_cv_logic_cascade/Logic Cascade');
```

Finally, open the Signal Builder and note that there are three combinations given for the Boolean inputs **a**, **b**, and **c**. These combinations are *FFF*, *TFT*, and *TTT*.

```
open_system('slvnvdemo_cv_logic_cascade/Signal Builder');
```
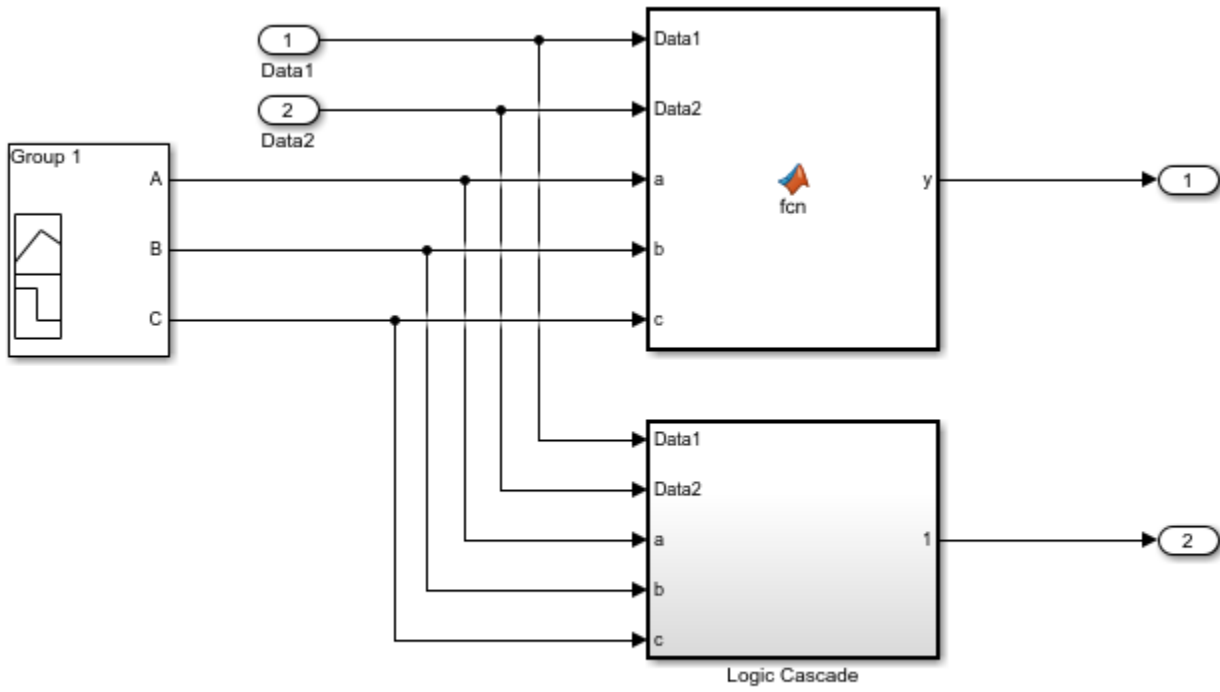
Copyright 2016 The MathWorks Inc.
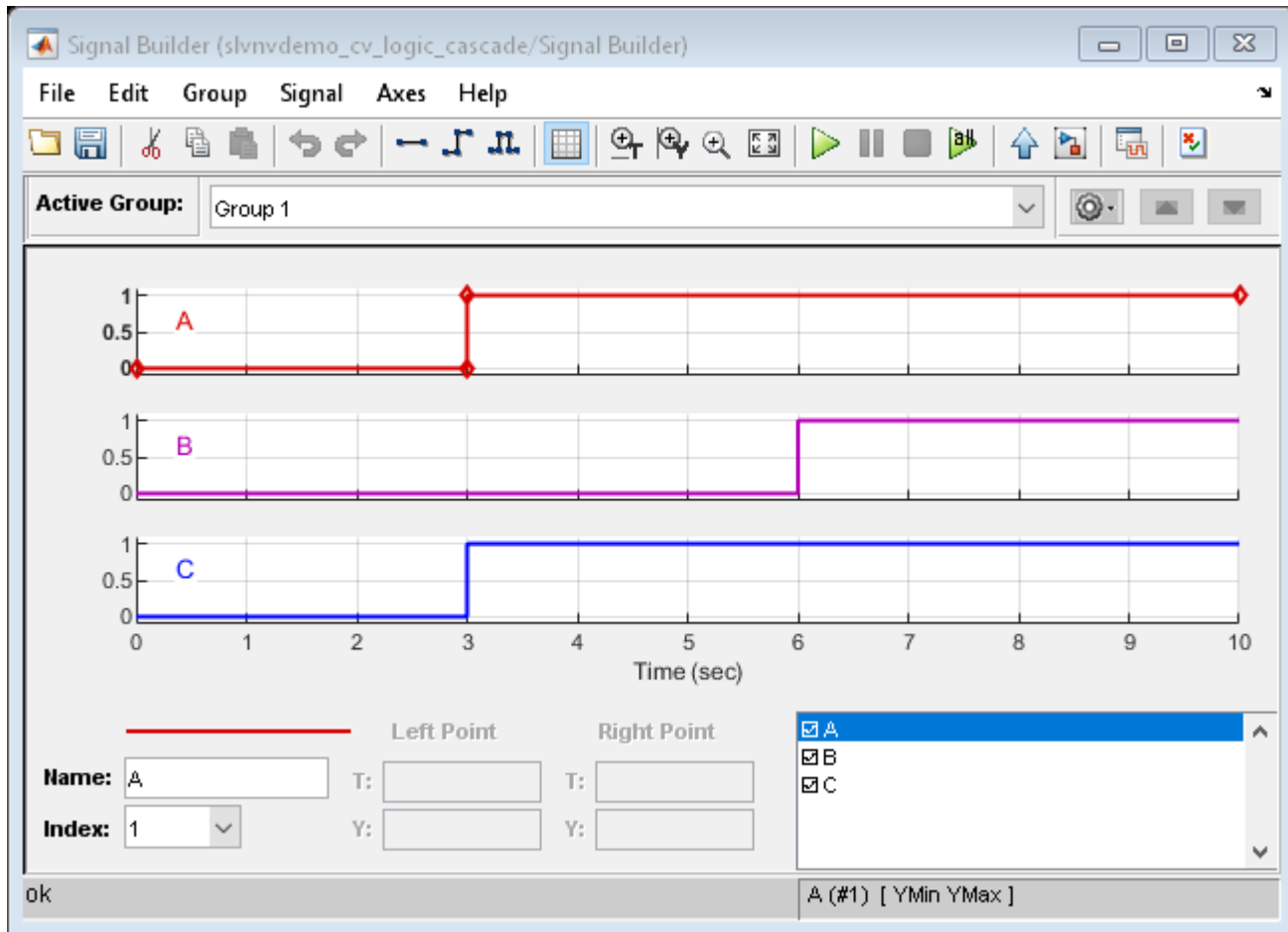
Close the Signal Builder.

```
close_system('slvnvdemo_cv_logic_cascade/Signal Builder', 0);
```

Comparing MCDC Results in the Coverage Report

Simulate the model and generate a Coverage Report.

```
testObj = cvtest('slvnvdemo_cv_logic_cascade');
testObj.settings.decision = 1;
testObj.settings.condition = 1;
testObj.settings.mcdc = 1;
covdata = cvsim(testObj); % Simulate for coverage
cvhtml('exampleReport.html',covdata); % Generate Coverage Report
```

MCDC Results for MATLAB Function block

In the generated report, navigate to the details for the MATLAB Function block.

MATLAB Function "fcn"

Justify or Exclude

Parent: slvnvdemo_cv_logic_cascade/MATLAB Function

Uncovered Links: ◀

Metric	Coverage
Cyclomatic Complexity	4
Condition	83% (5/6) condition outcomes
Decision	100% (3/3) decision outcomes
MCDC	33% (1/3) conditions reversed the outcome

```
1   function y = fcn(Data1, Data2, a, b, c)
2
3   if (a && (b || c))
4       y = Data1;
5   else
6       y = Data2;
7   end
8
```

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition	True Out	False Out
a && (b || c)		
a	TFT	Fxx
b	TTx	(TFF)
c	TFT	(TFF)

The MCDC results for the `if` statement in the MATLAB Function block are as would be expected, given the specified inputs.

MCDC Results for Logic Cascade

Next examine the results for the logic cascade. Recall that this combination of blocks implements the same logic as the MATLAB code in the MATLAB Function block; therefore, we would expect that the MCDC results would be the same, as well.

Let's first take a look at the upstream `Or_Block`.

Logic block "Or_Block"

Justify or Exclude

Parent: slvnvdemo_cv_logic_cascade/Logic Cascade

Metric	Coverage
Cyclomatic Complexity	0
Condition	100% (4/4) condition outcomes
MCDC	see And_Block
Execution	100% (1/1) objective outcomes

Notice that the MCDC summary for this block has a link with the text `"see And_Block"`, referring to the Logical Operator at the root of the cascade. Click on this link to be taken to the section of the report showing results for this block.

Logic block "And_Block"

Justify or Exclude

Parent:	slvnvdemo_cv_logic_cascade/Logic Cascade
Uncovered Links:	➡

Metric	**Coverage**
Cyclomatic Complexity	0
Condition	75% (3/4) condition outcomes
MCDC	33% (1/3) conditions reversed the outcome
Execution	100% (1/1) objective outcomes

Conditions analyzed

Description	True	False
input port 1	8	3
input port 2	8	0 ⊞

MC/DC analysis (combinations in parentheses did not occur)
Includes 2 blocks

Decision/Condition	True Out	False Out
C1 && (C2 || C3)		
C1 (And_Block In1)	TFT	Fxx
C2 (Or_Block In1)	TTx	(TFF)
C3 (Or_Block In2)	TFT	(TFF)

The Logical Operator block at the root of the cascade (in this case And_Block) reports the MCDC results for the entire cascade.

The details for the MCDC analysis of the cascade first show a link illustrating how many blocks are included in the cascade. Clicking on the link "Includes 2 blocks" will bring up the model and highlight the two blocks included in the cascade (Or_Block and And_Block).

This section of the report then shows the Boolean expression represented by the cascade, in this case *C1 && (C2 || C3)*, where C1, C2, and C3 are the conditions which correspond to the three inputs to the cascade. For each condition, the table illustrates the associated block and its input (shown in parenthesis) as well as the MCDC result. These results indicate that the input combinations TTx, Fxx,

and TFT have all been exercised, but TFF has not. This matches the expectation given the inputs generated by the Signal Builder (TTT, FFF, and TFT).

Furthermore, as expected, both the Boolean expression and MCDC results shown for this cascade match what was shown for the `if` statement implementing the equivalent logic in the MATLAB Function block.

Coverage Informer and Model Coloring

Display coverage results on the model using the following command:

`cvmodelview(covdata);`
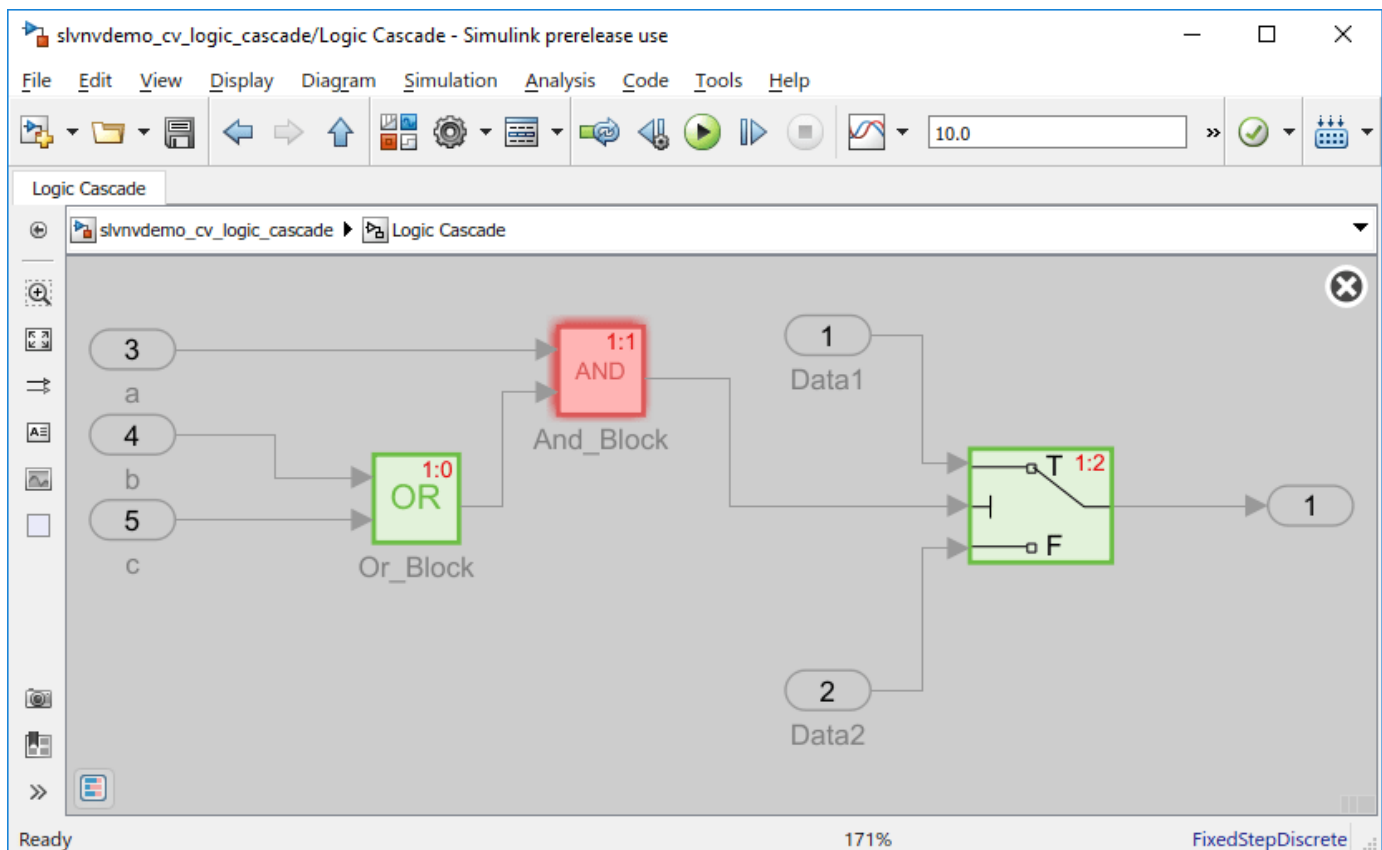
As was shown in the Coverage Report, MCDC objectives are not recorded for the individual Logical Operator blocks in a cascade; rather, MCDC objectives are recorded for the Boolean expression represented by the combination of blocks in the cascade, and results are reported on the final block in the cascade. The highlighting of the model reflects this, as well. Given the input combinations FFF, TFT, and TTT for the three inputs **a**, **b**, and **c**, `Or_Block` receives full coverage, because all of the block's Condition coverage objectives have been satisfied. However, because there are MCDC objectives associated with this cascade which have not been satisfied, `And_block` (the final block in the cascade) is highlighted in red.

Hover over `And_block` for more information.
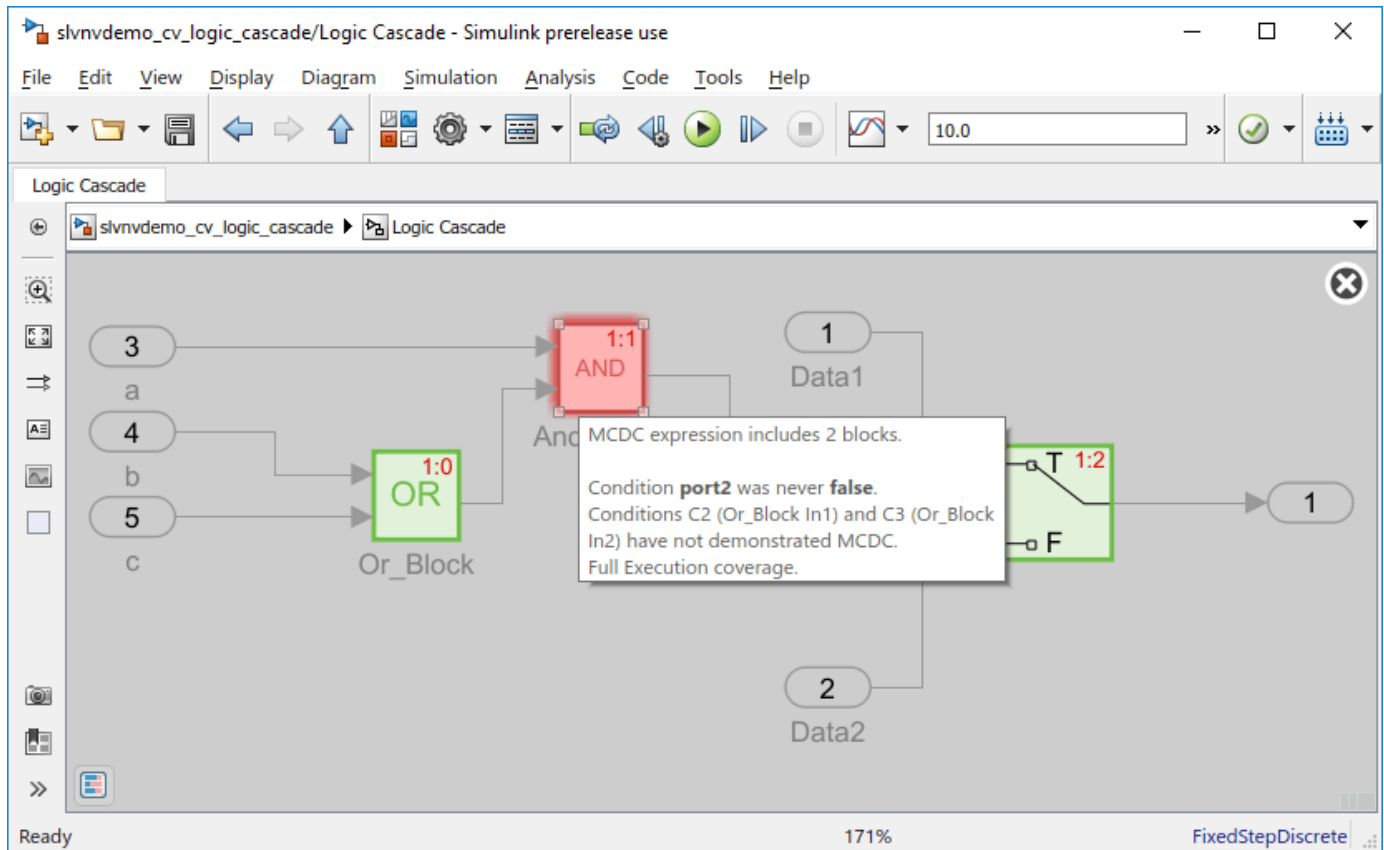
The tooltip correctly reports that this block does not receive full coverage, because some MCDC objectives for the cascade are not satisfied.

Command Line

You can also retrieve the MCDC results for the logic block cascade from the MATLAB command line using `mcdcinfo`. Again, MCDC objectives for the cascade will be found on the final block in the cascade.

```
[coverage_casc, description_casc] = mcdcinfo(covdata, 'slvnvdemo_cv_logic_cascade/Logic Cascade//
description_casc.condition(1)
description_casc.condition(2)
description_casc.condition(3)


coverage_casc =

     1     3


description_casc =

  struct with fields:

           text: 'C1 && (C2 || C3)'
      condition: [1x3 struct]
     isFiltered: 0
```

```
       filterRationale: ''
     justifiedCoverage: 0


ans =

  struct with fields:

               text: 'C1 (And_Block In1)'
           achieved: 1
           trueRslt: 'TFT'
          falseRslt: 'Fxx'
         isFiltered: 0
        isJustified: 0
     filterRationale: ''
      trueExecutedIn: []
     falseExecutedIn: []


ans =

  struct with fields:

               text: 'C2 (Or_Block In1)'
           achieved: 0
           trueRslt: 'TTx'
          falseRslt: '(TFF)'
         isFiltered: 0
        isJustified: 0
     filterRationale: ''
      trueExecutedIn: []
     falseExecutedIn: []


ans =

  struct with fields:

               text: 'C3 (Or_Block In2)'
           achieved: 0
           trueRslt: 'TFT'
          falseRslt: '(TFF)'
         isFiltered: 0
        isJustified: 0
     filterRationale: ''
      trueExecutedIn: []
     falseExecutedIn: []
```

Other blocks that are members of the cascade will not exhibit MCDC objectives.

```
[coverage_or, description_or] = mcdcinfo(covdata, 'slvnvdemo_cv_logic_cascade/Logic Cascade/Or_B'


coverage_or =

     []
```

```
description_or =

    []
```

Short-Circuiting of Boolean Expressions for MCDC

In example model `slvnvdemo_cv_logic_cascade`, coverage settings are set such that Logical Operator blocks are treated as short-circuiting.

Due to this setting, when analyzing a cascade of Logical Operator blocks, the operators in the corresponding Boolean expression are treated as short-circuiting for the purposes of MCDC. As illustrated by the results shown above, this means that MCDC recognizes short-circuiting that occurs both within and across Logical Operator blocks. As such, the MCDC results for the cascade of Logical Operator blocks matches those of the `if` statement in the MATLAB Function block, as the latter is always treated as short-circuiting.

Short-circuiting within a block

Notice that in the example above, the True Out MCDC objective outcome for `C2` is `TTx`, indicating that when `C1` and `C2` are both true, `C3` is inconsequential due to short-circuiting within the `Or_Block`.

Short-circuiting across multiple blocks

Furthermore, consider the False Out MCDC objective outcome for `C1`, `Fxx`. This outcome illustrates how MCDC analysis recognizes short-circuiting across blocks. Because the first input to `And_Block` is false, the second input is short-circuited. Subsequently, for the purposes of MCDC, this short-circuits `Or_Block` (and both of its inputs) entirely. The short-circuiting behavior of MCDC for logic block cascades occurs based on the precedence of operations in the corresponding Boolean expression (regardless of the execution order of the Logical Operator blocks during simulation).

Non-short-circuiting Boolean expressions

You can also treat the Boolean expression represented by a cascade of Logical Operator blocks as non-short-circuiting during MCDC analysis, provided that the masking definition of MCDC is being used. To do so, set the parameter **CovLogicBlockShortCircuit** to `"off"` and ensure that **CovMcdcMode** is set to `"Masking"`. These are, in fact, the default settings for these parameters when creating a new model.

Note, if **CovLogicBlockShortCircuit** is `"off"` and **CovMcdcMode** is set to `"UniqueCause"` then the Logical Operator blocks in a cascade will be analyzed individually for the purposes of MCDC, and MCDC for the Boolean expression represented by the cascade as a whole will not be calculated.

Notice that when the cascade in this example is not treated as short-circuiting, some MCDC objectives are no longer satisfied by the given inputs.

```
set_param('slvnvdemo_cv_logic_cascade', 'CovLogicBlockShortCircuit', 'off');
set_param('slvnvdemo_cv_logic_cascade', 'CovMcdcMode', 'Masking');
covdata_non_sc = cvsim('slvnvdemo_cv_logic_cascade'); % Simulate for coverage with logic block sh
cvhtml('exampleReport_non_sc.html',covdata_non_sc); % Generate Coverage Report
```

Logic block "And_Block"

Justify or Exclude

Parent: slvnvdemo_cv_logic_cascade/Logic Cascade

Uncovered Links: ➡

Metric	Coverage
Cyclomatic Complexity	0
Condition	100% (4/4) condition outcomes
MCDC	0% (0/3) conditions reversed the outcome
Execution	100% (1/1) objective outcomes

MC/DC analysis (combinations in parentheses did not occur)
Includes 2 blocks

Decision/Condition	True Out	False Out
C1 && (C2 || C3)		
C1 (And_Block In1)	TFT	(FTT)
C2 (Or_Block In1)	(TTF)	(TFF)
C3 (Or_Block In2)	TFT	(TFF)

View Coverage Results in Simulink Canvas

In this section...
"Overview of Model Coverage Highlighting" on page 5-22
"Enable Coverage Highlighting" on page 5-22
"View Coverage Details" on page 5-24

Overview of Model Coverage Highlighting

When you simulate a Simulink model, you can configure your model to provide visual results that enable you to see which objects failed to record 100% coverage. After the simulation:

- In the model window, model objects are highlighted in certain colors according to what coverage was recorded:

  - Green indicates that an object received full coverage during simulation.
  - Green with a dashed border indicates that an object had incomplete coverage that you justified.
  - Red indicates that an object received incomplete coverage.
  - Gray with a dashed border indicates that you excluded an object from coverage.
  - Objects with no color highlighting did not receive coverage.

- When you place your cursor over a colored object, you see a tooltip with details about the coverage recorded for that block. For subsystems and Stateflow charts, the coverage tooltip lists the summary coverage for all objects in that subsystem or chart. For other blocks, the coverage tooltip lists specific details about the objects that did not receive 100% coverage.

The simulation highlights blocks that received these types of model coverage:

- "Execution Coverage (EC)" on page 1-3
- "Decision Coverage (DC)" on page 1-3
- "Condition Coverage (CC)" on page 1-3
- "Modified Condition/Decision Coverage (MCDC)" on page 1-4
- "Relational Boundary Coverage" on page 1-7
- "Saturate on Integer Overflow Coverage" on page 1-7
- "Objectives and Constraints Coverage" on page 1-6

Enable Coverage Highlighting

Your model will receive coverage highlighting if you simulate the model using the **Run** button. After simulation, you can see which model objects received full, partial, or no coverage.

If you simulate without the **Run** button, or load coverage data, you can click **Highlight model with coverage results** in the Results Explorer to enable model coverage highlighting. To open the results explorer, in the **Apps** tab, select **Coverage Analyzer**. Then click **Results Explorer**. For more information, see "Accessing Coverage Data from the Results Explorer" on page 3-7. You can also use `cvmodelview` to enable model highlighting.

Highlighted Coverage Results

Examples of highlighted model objects in colors that correspond to the recorded coverage are:

Green: Full Coverage

The Switch block received 100% coverage, as indicated by the green highlighting and the information in the coverage tooltip.
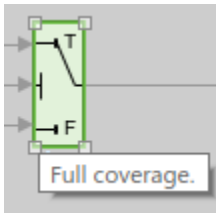
Green with Dashed Border: Justified Coverage

The Relational Operator block received justified coverage, as indicated by the green highlighting with a dashed border and the information in the coverage tooltip.
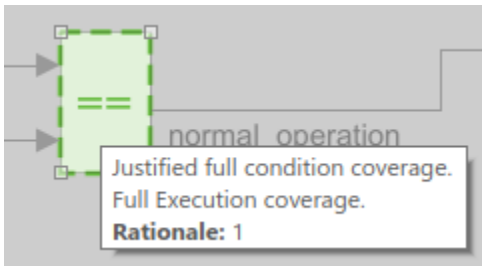
Red: Partial Coverage

The `shift_logic` Stateflow chart received this coverage:
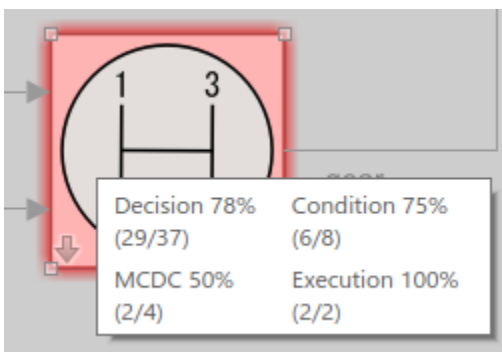
Inside the `shift_logic` Stateflow chart, the `gear_state` substate was never `fourth`.
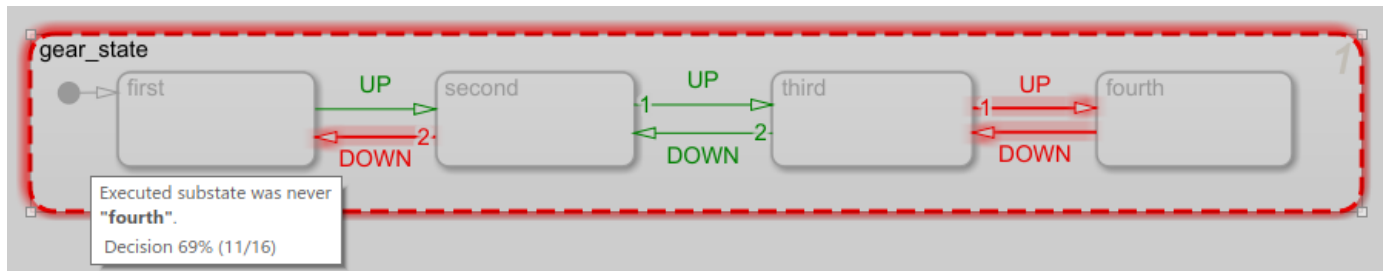
Two of the data ports in the Multiport Switch block were never executed.
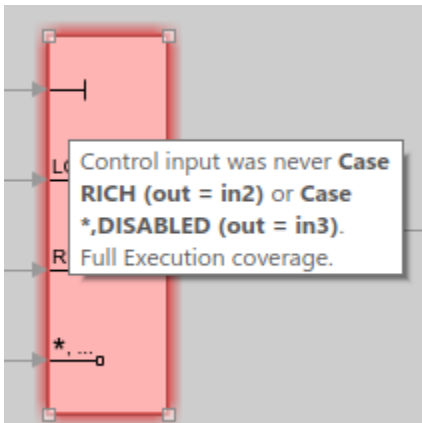
Gray with Dashed Border: Filtered Coverage

The fuel_rate_control subsystem is highlighted in gray because it was excluded from coverage recording.
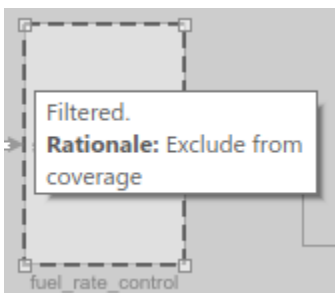
No Coloring: Coverage Not Recorded

The Inport block is not highlighted because it does not receive coverage recording.

View Coverage Details

After you highlight coverage results on the model, you can view coverage details for each model element in the **Coverage Details** window. To open the **Coverage Details** window, click the

Coverage Details icon in the lower-left corner of the Simulink block diagram, and then click **Open Coverage Details**:
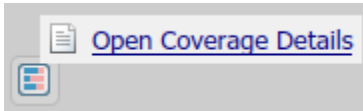
You can then click a model object to view its coverage details.

Model Coverage for Multiple Instances of a Referenced Model

In this section...
"About Coverage for Model Blocks" on page 5-26
"Record Coverage for Multiple Instances of a Referenced Model" on page 5-26

About Coverage for Model Blocks

Model blocks do not receive coverage directly; if you set the simulation mode of the Model block to `Normal`, SIL, or PIL, the Simulink Coverage software records coverage for the model referenced from the Model block. If the simulation mode for the Model block is anything other than `Normal`, SIL, or PIL, the software does not record coverage for the referenced model.

Your Simulink model can contain multiple Model blocks with the same simulation mode that reference the same model. When the software records coverage, each instance of the referenced model can be exercised with different inputs or parameters, possibly resulting additional coverage data for the referenced model.

The Simulink Coverage software records coverage for all instances of the referenced model with the same simulation mode and combines the coverage data for that referenced model in the final results.

Record Coverage for Multiple Instances of a Referenced Model

To see how this works, simulate a model twice. The first time, you record coverage for one Model block in `Normal` simulation mode. The second time, you record coverage for two Model blocks in `Normal` simulation mode. Both Model blocks reference the same model.

- "Record Coverage for the First Instance of the Referenced Model" on page 5-26
- "Record Coverage for the Second Instance of the Referenced Model" on page 5-30

Record Coverage for the First Instance of the Referenced Model

Record coverage for one Model block.

1 Open your top-level model. This example uses the `sldemo_mdlref_datamngt` model:
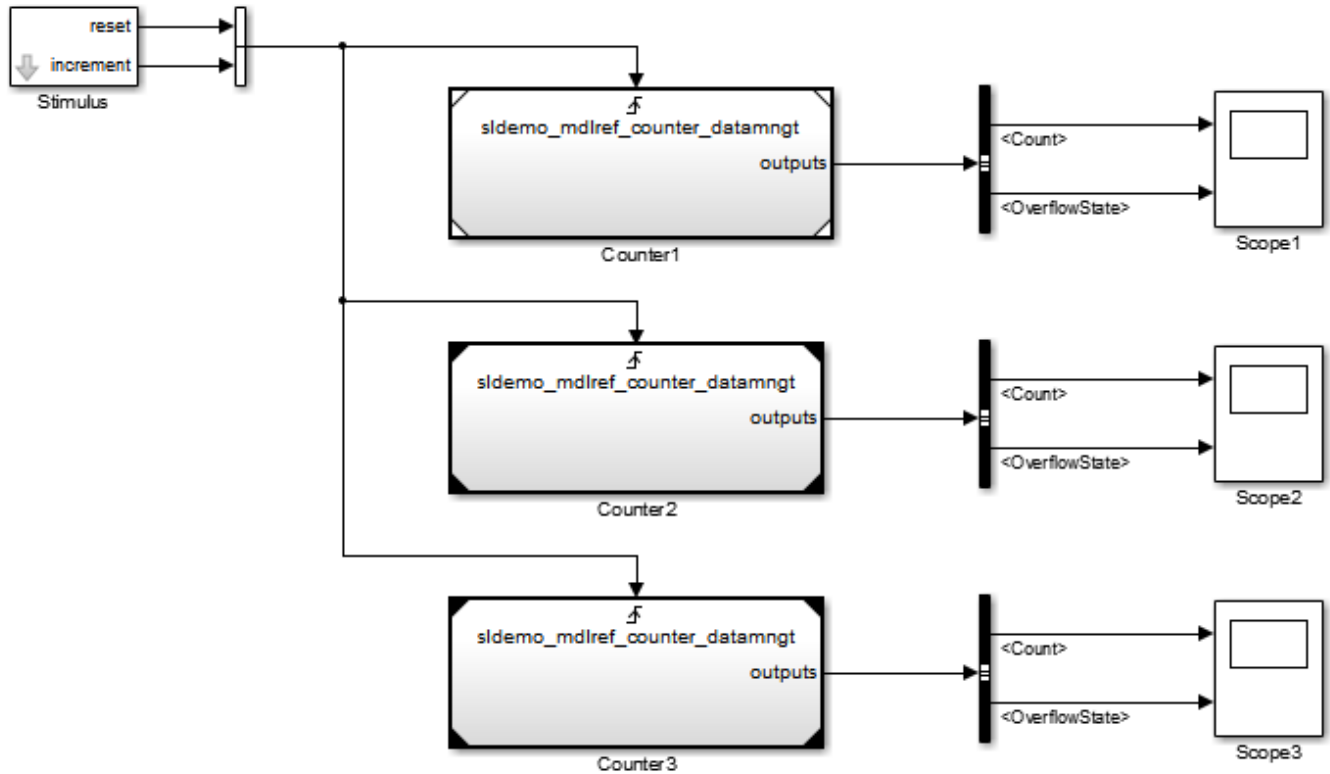
This model contains three Model blocks that reference the
`sldemo_mdlref_counter_datamngt` example model. The corners of each Model block indicate
the value of their **Simulation mode** parameter:

- Counter1 — Simulation mode: `Normal`
- Counter2 — Simulation mode: `Accelerator`
- Counter3 — Simulation mode: `Accelerator`

2 Configure your model to record coverage during simulation:

a In the Simulink Editor, select **Model Settings** on the **Modeling** tab.

b On the **Coverage** pane of the Configuration Parameters dialog box, select:

- **Enable coverage analysis**
- **Referenced Models**

c Click **Select Models**. In the Select Models for Coverage Analysis dialog box, you can select
only those referenced models whose simulation mode is `Normal`, `SIL`, or `PIL`. In this
example, only the first Model block that references `sldemo_mdlref_counter_datamngt` is
available for recording coverage.
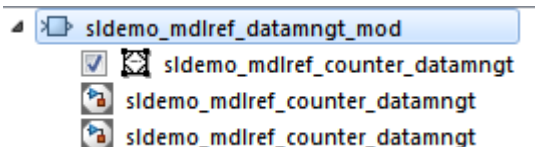
    d   Click **OK** to exit the Select Models for Coverage Analysis dialog box.

3   Click **OK** to save your coverage settings and exit the Configuration Parameters dialog box.

4   Simulate your model.

When the simulation is complete, the HTML coverage report opens. In this example, the coverage data for the referenced model, `sldemo_mdlref_counter_datamngt`, shows that the model achieved 69% coverage.

5   Click the hyperlink in the report for the referenced model.

The detailed coverage report for the referenced model opens, and the referenced model appears with highlighting to show coverage results.
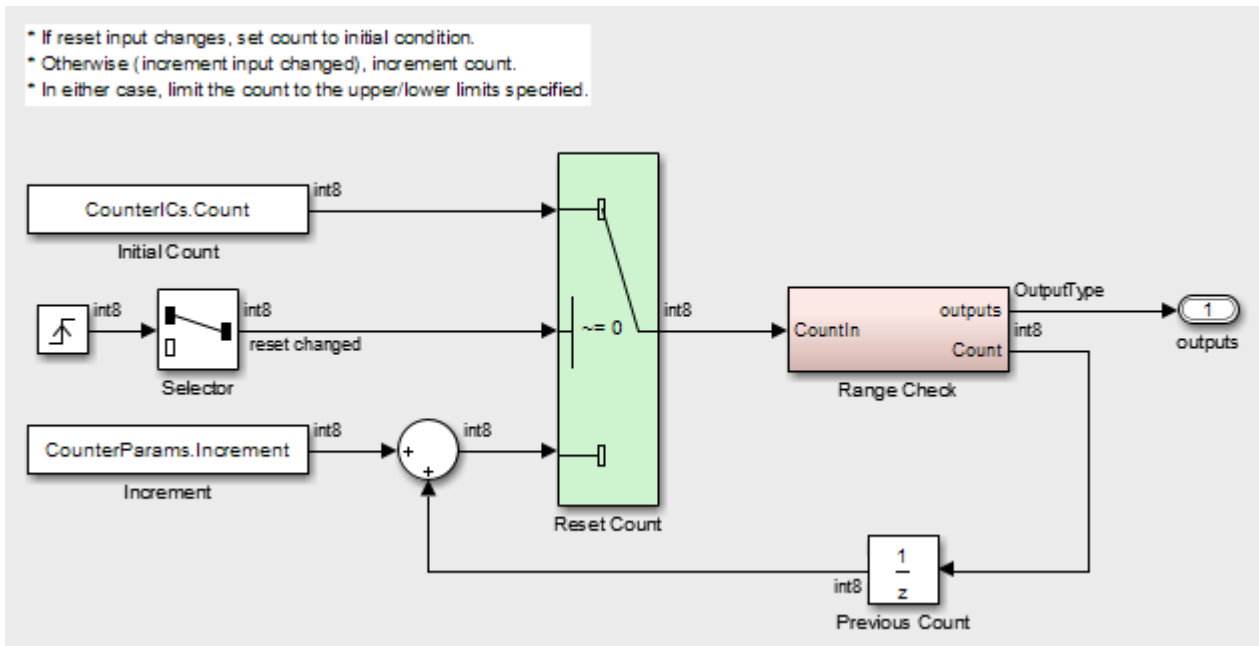
Note the following about the coverage for the Range Check subsystem in this example:

- The Saturate Count block executed 100 times. This block has four Boolean decisions. Decision coverage was 50%, because two of the four decisions were never recorded:

  - The decision `input > lower limit` was never `false`.
  - The decision `input >= upper limit` was never `true`.

Saturate block "Saturate Count"

Parent: sldemo_mdlref_counter_datamngt/Range Check

Uncovered Links: ➡

Metric	Coverage
Cyclomatic Complexity	2
Decision	50% (2/4) decision outcomes

Decisions analyzed:

input > lower limit	50%
false	0/50
true	50/50

input >= upper limit	50%
false	50/50
true	0/50

- The `DetectOverflow` function executed 50 times. This script has five decisions. The `DetectOverflow` script achieved 60% coverage because two of the five decisions were never recorded:

  - The expression `count >= CounterParams.UpperLimit` was never `true`.
  - The expression `count > CounterParams.LowerLimit` was never `false`.

MATLAB Function "DetectOverflow"

Parent:	sldemo_mdlref_counter_datamngt/Range Check/Detect Overflow
Uncovered Links:	←

Metric	**Coverage**
Cyclomatic Complexity	3
Decision	60% (3/5) decision outcomes

```
 1  function result = DetectOverflow(count, CounterParams)
 2  % DETECTOVERFLOW  Check count
 3  %#codegen
 4
 5  if (count >= CounterParams.UpperLimit)
 6     result = SlDemoRangeCheck.UpperLimit;
 7  elseif (count > CounterParams.LowerLimit)
 8     result = SlDemoRangeCheck.InRange;
 9  else
10     result = SlDemoRangeCheck.LowerLimit;
11  end
12
```

#1: function result = DetectOverflow(count, CounterParams)

Decisions analyzed:

function result = DetectOverflow(count, CounterParams)	100%
executed	50/50

#5: if (count >= CounterParams.UpperLimit)

Decisions analyzed:

if (count >= CounterParams.UpperLimit)	50%
false	50/50
true	0/50

#7: elseif (count > CounterParams.LowerLimit)

Decisions analyzed:

elseif (count > CounterParams.LowerLimit)	50%
false	0/50
true	50/50

Record Coverage for the Second Instance of the Referenced Model

Record coverage for two Model blocks. Set the simulation mode of a second Model block to `Normal` and simulate the model. In this example, the Counter2 block adds to the coverage for the model referenced from both Model blocks.

1   In the Simulink Editor for your top-level model, right-click a second Model block and select **Block Parameters (ModelReference)**.

   The Function Block Parameters dialog box opens.

2   Set the **Simulation mode** parameter to `Normal`.

3   Click **OK** to save your change and exit the Function Block Parameters dialog box.

The corners of the Model block change to indicate that the simulation mode for this block is `Normal`, as in the example below.

Counter 1

Counter2

Counter3
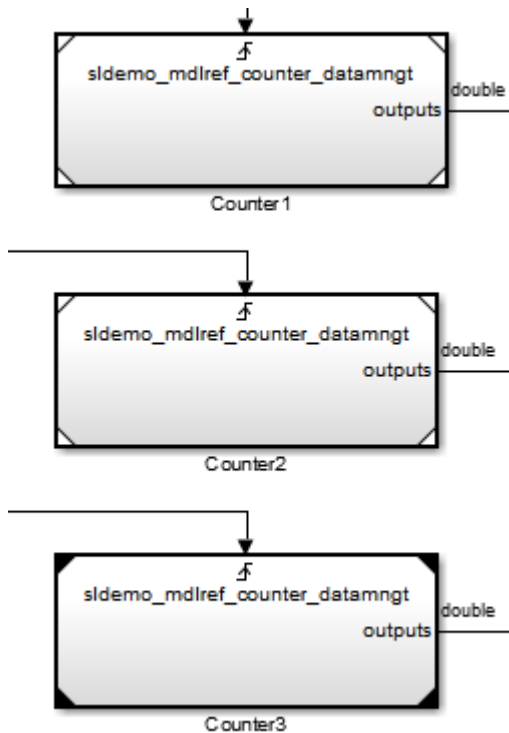
4 To make sure that the software records coverage for both instances of this model:

   a In the Simulink Editor, select **Model Settings** on the **Modeling** tab.

   b On the **Coverage** pane, select **Enable coverage analysis**.

   c Select **Referenced Models** and click **Select Models**.

      In the **Select Models for Coverage Analysis** dialog box, verify that both instances of the referenced model are selected. In this example, the list now looks like the following.
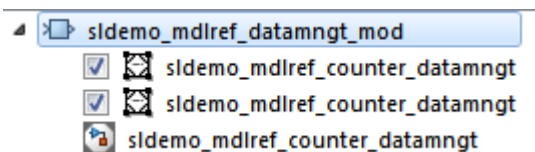
      If you have multiple instances of a referenced model in `Normal` mode, you can choose to record coverage for all of them or none of them.

   d Click **OK** to close the Select Models for Coverage Analysis dialog box.

5 Simulate your model again.

6 When the simulation is complete, open the HTML coverage report.

In this example, the referenced model achieved 85% coverage. Note the following about the coverage data for the Range Check subsystem:

- The Saturate Count block executed 179 times. The simulation of the Counter2 block executed the Saturate Count block an additional 79 times, for a total of 179 executions.

  The decision `input >= upper limit` was `true` 21 times during this simulation, compared to 0 during the first simulation. The fourth decision `input > lower limit` was still never `false`. Three out of four decisions were recorded during simulation, so this block achieved 75% coverage.

Saturate block "Saturate Count"

Parent:	sldemo_mdlref_counter_datamngt/Range Check
Uncovered Links:	➡

Metric	**Coverage**
Cyclomatic Complexity	2
Decision	75% (3/4) decision outcomes

Decisions analyzed:

input > lower limit	50%
false	0/79
true	79/79
input >= upper limit	100%
false	79/100
true	21/100

- The `DetectOverflow` function executed 100 times. The simulation of the Counter2 block executed the `DetectOverflow` function an additional 50 times.

  The `DetectOverflow` function has five decisions. The expression `count >= CounterParams.UpperLimit` was `true` 21 times during this simulation, compared to 0 during the first simulation. The expression `count > CounterParams.LowerLimit` was never `false`. Four out of five decisions were recorded during simulation, so the `DetectOverflow` function achieved 80% coverage.

MATLAB Function "DetectOverflow"

Parent:	sldemo_mdlref_counter_datamngt/Range Check/Detect Overflow
Uncovered Links:	←

Metric	**Coverage**
Cyclomatic Complexity	3
Decision	80% (4/5) decision outcomes

```
 1  function result = DetectOverflow(count, CounterParams)
 2  % DETECTOVERFLOW   Check count
 3  %#codegen
 4
 5  if (count >= CounterParams.UpperLimit)
 6      result = SlDemoRangeCheck.UpperLimit;
 7  elseif (count > CounterParams.LowerLimit)
 8      result = SlDemoRangeCheck.InRange;
 9  else
10      result = SlDemoRangeCheck.LowerLimit;
11  end
12
```

#1: function result = DetectOverflow(count, CounterParams)

Decisions analyzed:

function result = DetectOverflow(count, CounterParams)	100%
executed	100/100

#5: if (count >= CounterParams.UpperLimit)

Decisions analyzed:

if (count >= CounterParams.UpperLimit)	100%
false	79/100
true	21/100

#7: elseif (count > CounterParams.LowerLimit)

Decisions analyzed:

elseif (count > CounterParams.LowerLimit)	50%
false	0/79
true	79/79

Obtain Cumulative Coverage for Reusable Subsystems and Stateflow® Constructs

This example shows how to create and view cumulative coverage results for a model with a reusable subsystem.

Simulink® Coverage™ provides cumulative coverage for multiple instances of identically configured:

- Reusable subsystems
- Stateflow™ constructs

To obtain cumulative coverage, you add the individual coverage results at the command line. You can get cumulative coverage results for multiple instances across models and test harnesses by adding the individual coverage results.

Open example model

At the MATLAB® command line, type:

```
model = 'slvnvdemo_cv_mutual_exclusion';
open_system(model);
```


Copyright 1990-2019 The MathWorks Inc.

This model has two instances of a reusable subsystem. The instances are named Subsystem 1 and Subsystem 2.

Get decision coverage for Subsystem 1

Execute the commands for Subsystem 1 decision coverage:

```
testobj1 = cvtest([model '/Subsystem 1']);
testobj1.settings.decision = 1;
covobj1 = cvsim(testobj1);
```

Get decision coverage for Subsystem 2

Execute the commands for Subsystem 2 decision coverage:

```
testobj2 = cvtest([model '/Subsystem 2']);
testobj2.settings.decision = 1;
covobj2 = cvsim(testobj2);
```

Add coverage results for Subsystem 1 and Subsystem 2

Execute the command to create cumulative decision coverage for Subsystem 1 and Subsystem 2:

```
covobj3 = covobj1 + covobj2;
```

Generate coverage report for Subsystem 1

Create an HTML report for Subsystem 1 decision coverage:

```
cvhtml('subsystem1',covobj1)
```

The report indicates that decision coverage is 50% for Subsystem 1. The `true` condition for `enable logical value` is not analyzed.

Generate coverage report for Subsystem 2

Create an HTML report for Subsystem 2 decision coverage:

```
cvhtml('subsystem2',covobj2)
```

The report indicates that decision coverage is 50% for Subsystem 2. The `false` condition for `enable logical value` is not analyzed.

Generate coverage report for cumulative coverage of Subsystem 1 and Subsystem 2

Create an HTML report for cumulative decision coverage for Subsystem 1 and Subsystem 2:

```
cvhtml('cum_subsystem',covobj3)
```

Cumulative decision coverage for reusable subsystems Subsystem 1 and Subsystem 2 is 100%. Both the `true` and `false` conditions for `enable logical value` are analyzed.

Trace Coverage Results to Requirements

If you run test cases in Simulink Test that are linked to requirements in Simulink Requirements, the aggregated coverage report details the requirements implemented by each model element and the tests that verify those requirements.

Prerequisites for Tracing Requirements Links

To view linked requirements details in your coverage report, you must:

- Link to test cases from requirements in Simulink Requirements. For more information, see "Link to Test Cases from Requirements" (Simulink Requirements) and "Perform Functional Testing and Analyze Test Coverage" on page 10-9.
- Run your test cases through the Simulink Test Manager. For more information, see "Requirements-Based Testing for Model Development" (Simulink Test).
- Record the aggregated coverage results for at least two test cases.

This example shows how to view the links between test cases, model elements, and linked requirements in a coverage report.

Open the slreqCCProjectStart Project and Load Test Cases

1 Open the `slreqCCProjectStart` project.

   `slreqCCProjectStart`
2 Load the `DriverSwRequest_Tests.mldatx` test data suite and open the Simulink Test Manager.

   `sltest.testmanager.load('DriverSwRequest_Tests.mldatx')`
   `sltest.testmanager.view`
3 In the Simulink Test Manager, click the `DriverSwRequest_Tests` test file.
4 To enable decision coverage collection for the test case, in the right pane under **Coverage Settings**:

   - Select **Record coverage for system under test**.
   - Under **Coverage Metrics**, select **Decision**.
   - Save your changes.
5 Run the loaded test cases.

   `resultObj = sltest.testmanager.run`
6 When the test finishes, navigate to the test case results in the Test Manager. The Aggregated Coverage Results section displays the coverage for the analyzed model.

AGGREGATED COVERAGE RESULTS

ANALYZED MODEL	REPORT	COMPLEXI...	DECISION	EXECUTION
crs_controller/DriverSwRequest		12	95%	100%

Add Tests for Missing Coverage   Export

7   Click **Report** to create a coverage report.

The coverage report shows requirements details for each model element, including linked requirements, which tests verify the requirements, and which runs are associated with each verification test.

Switch block "Switch1"

Justify or Exclude

Requirement Testing Details

Implemented Requirements	Verified by Tests	Associated Runs
Enable Switch Detection	Enable button	U1.1

Parent: crs_controller/DriverSwRequest

Metric	Coverage
Cyclomatic Complexity	1
Decision	100% (2/2) decision outcomes
Execution	100% (1/1) objective outcomes

Decisions analyzed

logical trigger input	100%
false (output is from 3rd input port)	1607/1608 U1.1
true (output is from 1st input port)	1/1608 U1.1

5-37

The **Decisions analyzed** section links to the first test case that reached each decision. To see other test cases that also reached a decision, hover over the listed test case. For more information, see "Trace Coverage Results to Associated Test Cases" on page 5-41.

See Also

More About

- "Requirement Testing Details" on page 6-20
- "Link to Test Cases from Requirements" (Simulink Requirements)
- "Perform Functional Testing and Analyze Test Coverage" on page 10-9

Assess Coverage Results from Requirements-Based Tests

You can scope coverage results to linked requirements-based tests from the Simulink Test Manager. The aggregated coverage results are scoped such that each test only contributes coverage for the corresponding model elements that implement the requirements verified by that test.

Rationale for Scoping Coverage Results to Linked Requirements-Based Tests

If your model-based design workflow requires that models are fully exercised by requirements-based tests, you can scope your coverage results to only those outcomes exercised by requirements-based tests. As an example, DO-178C suggests that structural coverage information collected during requirements-based testing should confirm that the degree of structural coverage is appropriate and satisfies the software requirements. When you enable **Scope coverage results to linked requirements**, the aggregated coverage results are scoped such that each test only contributes coverage for the corresponding model elements that implement the requirements verified by that test.

You define requirements and link them to model elements and tests by using Simulink Requirements. Scoping coverage results to linked requirements allows you to produce evidence that your model coverage comes from the intended requirements-based tests and is not a side effect of an unrelated test. Scoping coverage results to linked requirements can also reveal inadequate requirement linking or testing gaps that might otherwise be difficult to detect in aggregated coverage results.

Prerequisites for Scoping Coverage Results to Linked Requirements-Based Tests

To scope coverage results to linked requirements, you must:

- Have licenses for Simulink Test and Simulink Coverage.
- Link requirements in Simulink Requirements to model elements and to test cases in Simulink Test that verify the requirements. For more information on creating requirements links, see "Link Blocks and Requirements" (Simulink Requirements).

    Note You cannot create or edit requirements links or view detailed information about the requirements without a Simulink Requirements license.

- Collect coverage by using the Simulink Test Manager, and enable **Scope coverage results to linked requirements** for the aggregated coverage results. For more information on setting up coverage collection in the Simulink Test Manager, see "Collect Coverage in Tests" (Simulink Test).

Coverage Reporting for Aggregated Coverage Results Scoped to Linked Requirements

The following coverage report shows requirements testing details and coverage details for a MultiPortSwitch block called MPSwitch1.

In the example above, MPSwitch1 implements Requirement 1, which is verified by Testcase 1. Therefore, Testcase 1 attempts to provide full coverage for MPSwitch1. Scoping coverage results to linked requirements makes it easier to assess the extent to which MPSwitch1 was exercised by Testcase 1 when viewing aggregated coverage results.

The first decision outcome is successfully exercised by Testcase 1 and is reported as satisfied. The second decision outcome is not exercised by Testcase 1, but is reached by a test unrelated to Requirements 1. The coverage report therefore reports this decision as not satisfied.

The third decision outcome is not exercised by any test and is therefore reported as not satisfied.

Example

For an example of how to scope coverage results to linked requirements from the Simulink Test Manager, see "Test Coverage for Requirements-Based Testing" (Simulink Test).

See Also

More About
- "Link Blocks and Requirements" (Simulink Requirements)
- "Collect Coverage in Tests" (Simulink Test)

Trace Coverage Results to Associated Test Cases

If you record aggregated coverage results for test cases in Simulink Test with your model in Normal or SIL/PIL mode, the aggregated coverage report links to the test cases associated with each model element.

Prerequisites for Tracing Associated Test Cases to Coverage Results

To view associated test cases in your coverage report, you must record aggregated coverage results for at least two test cases through the Simulink Test Manager, or produce a coverage report for cumulative coverage results from the Results Explorer. For more information, see "Perform Functional Testing and Analyze Test Coverage" on page 10-9.

Note Test case traceability and unit test aggregation for MCDC coverage are only supported for Masking Mode. Unique-cause MCDC is not supported for these features.

Aggregate Unit-Level Coverage Data into Top-Level Model Coverage

This example shows how to generate an aggregated coverage report that includes results from both integration and unit tests.

Load the Test Cases into the Simulink® Test™ Manager

The `slcovTestTraceabilityExample.mldatx` test data is configured to record decision coverage.

```
sltest.testmanager.load('slcovTestTraceabilityExample.mldatx');
sltest.testmanager.view
```

Run the Test Cases

From the Simulink Test Manager, select the `Combined Integration and Unit Tests` test suite and click **Run**. This test suite contains two sub-suites, `Integration Tests` and `Unit Tests`. Alternatively, run the following command:

```
results = sltest.testmanager.run;
```

Access the Coverage Report for the Integration Tests

From the **Results and Artifacts** pane of the Simulink Test Manager, select the results for `Integration Tests`. From the **Aggregated Coverage Results** section, click the **Report** button.

The coverage report for this test suite only shows coverage results for the integration tests.

Aggregated Tests

Run	Test Name	Date
Model: "slcovSerialSwitchUnits"		
T1	Switches Integration Test - In Range	12-Jul-2019 10:52:24
T2	Switches Integration Test - Out of Range	12-Jul-2019 10:52:29

View Subsystem Details

View the coverage details for the subsystem SwitchUnit2. Notice that this subsystem does not receive full coverage. The first three decision outcomes are covered by integration test run T1. The fourth decision outcome for the MPSwitch block cannot be satisfied in the integrated system.

MultiPortSwitch block "MPSwitch"

Justify or Exclude

Parent: slcovSerialSwitchUnits/SwitchUnit2

Metric	Coverage
Cyclomatic Complexity	3
Decision	75% (3/4) decision outcomes

Decisions analyzed

truncated input value	75%
= 1 (output is from input port 1)	4/22 T1
= 2 (output is from input port 2)	4/22 T1
= 3 (output is from input port 3)	14/22 T1
= * (output is from input port 4)	0/22

Access the Coverage Report for the Unit Tests

From the **Results and Artifacts** pane of the Simulink Test Manager, select the results for Unit Tests. From the **Aggregated Coverage Results** section, click the **Report** button.

The coverage report for this test suite only shows coverage results for the unit tests of the SwitchUnit2 subsystem that were recorded by using subsystem test harnesses.

Aggregated Tests

Run	Test Name	Date
Subsystem: "/SwitchUnit2"		
U1.1	Switch2 Unit Test - In Range	17-Jul-2019 13:06:17
U1.2	Switch2 Unit Test - Out of Range	17-Jul-2019 13:06:18

View Subsystem Details

View the coverage details for the subsystem SwitchUnit2. Notice that this subsystem does receive full coverage from the unit tests.

MultiPortSwitch block "MPSwitch"

Justify or Exclude
Parent: slcovSerialSwitchUnits/SwitchUnit2

Metric	Coverage
Cyclomatic Complexity	3
Decision	100% (4/4) decision outcomes
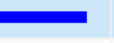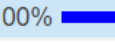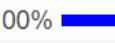
Decisions analyzed

truncated input value	100%
= 1 (output is from input port 1)	4/22 U1.1
= 2 (output is from input port 2)	4/22 U1.1
= 3 (output is from input port 3)	3/22 U1.1
= * (output is from input port 4)	11/22 U1.2

Locate the Combined Unit-Level and System-Level Coverage Report

From the **Results and Artifacts** pane of the Simulink Test Manager, select the results for `Combined Integration and Unit Tests`. The results show two coverage reports available--one report for the SwitchUnit2 subsystem tested by the unit tests and one report for the top-level model that incorporates results from both the unit and integration tests.

▼ AGGREGATED COVERAGE RESULTS   ?

ANALYZED MODEL	REPORT	COMPLEXI...	DECISION	EXECUTION	＋
slcovSerialSwitchUnits	⌐	8	100%	100%	
slcovSerialSwitchUnits/SwitchUnit2	⌐	4	100%	100%	

＋ Add Tests for Missing Coverage   ⌐ Export

Access Aggregated Coverage Report for the Top-Level Model

When you click the **Report** button for the top-level model, Simulink Coverage aggregates the integration and unit tests into a system-level coverage report.

Aggregated Tests

Run	Test Name	Date
Subsystem: "/SwitchUnit2"		
U1.1	Switch2 Unit Test - In Range	12-Jul-2019 10:55:17
U1.2	Switch2 Unit Test - Out of Range	12-Jul-2019 10:55:17
Model: "slcovSerialSwitchUnits"		
T1	Switches Integration Test - In Range	12-Jul-2019 10:55:14
T2	Switches Integration Test - Out of Range	12-Jul-2019 10:55:15

View Subsystem Details

Notice that the subsystem receives full coverage. The first three decision outcomes for the MPSwitch MultiPortSwitch block are covered by the integration test run T1. The fourth decision outcome for the MPSwitch MultiPortSwitch block is covered by unit test run U1.2.

MultiPortSwitch block "MPSwitch"

Justify or Exclude

Parent: slcovSerialSwitchUnits/SwitchUnit2

Metric	Coverage
Cyclomatic Complexity	3
Decision	100% (4/4) decision outcomes

Decisions analyzed

truncated input value	100%
= 1 (output is from input port 1)	8/44 T1
= 2 (output is from input port 2)	8/44 T1
= 3 (output is from input port 3)	17/44 T1
= * (output is from input port 4)	11/44 U1.2

See Also

More About

- "Perform Functional Testing and Analyze Test Coverage" on page 10-9
- "Aggregated Tests" on page 6-11

Model Coverage for MATLAB Functions

In this section...

About Model Coverage for MATLAB Functions

The Simulink Coverage software simulates a Simulink model and reports model coverage data for the decisions and conditions of code in MATLAB Function blocks. Model coverage only supports coverage for MATLAB functions configured for code generation.

For example, consider the following `if` statement:

```
if (x > 0 || y > 0)
    reset = 1;
```

The `if` statement contains a decision with two conditions ($x > 0$ and $y > 0$). The Simulink Coverage software verifies that all decisions and conditions are taken during the simulation of the model.

Types of Model Coverage for MATLAB Functions

The types of model coverage that the Simulink Coverage software records for MATLAB functions configured for code generation are:

- "Decision Coverage" on page 5-45
- "Condition and MCDC Coverage" on page 5-46
- "Simulink Design Verifier Coverage" on page 5-46
- "Relational Boundary Coverage" on page 5-46

Decision Coverage

During simulation, the following MATLAB Function block statements are tested for decision coverage:

- Function header — Decision coverage is 100% if the function or local function is executed.
- `if` — Decision coverage is 100% if the `if` expression evaluates to `true` at least once, and `false` at least once.
- `switch` — Decision coverage is 100% if every `switch` case is taken, including the fall-through case.
- `for` — Decision coverage is 100% if the equivalent loop condition evaluates to `true` at least once, and `false` at least once.
- `while` — Decision coverage is 100% if the equivalent loop condition evaluates to `true` at least once, and evaluates to `false` at least once.

Condition and MCDC Coverage

During simulation, in the MATLAB Function block function, the following logical conditions are tested for condition and MCDC coverage:

- `if` statement conditions
- Logical expressions in assignment statements

Simulink Design Verifier Coverage

The following MATLAB functions are active in code generation and in Simulink Design Verifier:

- `sldv.condition`
- `sldv.test`
- `sldv.assume`
- `sldv.prove`

When you specify the **Objectives and Constraints** coverage metric in the **Coverage** pane of the Configuration Parameters dialog box, the Simulink Coverage software records coverage for these functions.

Each of these functions evaluates an expression *expr*, for example, `sldv.test(`*expr*`)`, where *expr* is a valid Boolean MATLAB expression. Simulink Design Verifier coverage measures the number of time steps that the expression *expr* evaluates to `true`.

If *expr* is `true` for at least one time step, Simulink Design Verifier coverage for that function is 100%. Otherwise, the Simulink Coverage software reports coverage for that function as 0%.

For an example of coverage data for Simulink Design Verifier functions in a coverage report, see "Simulink Design Verifier Coverage" on page 6-37.

Relational Boundary Coverage

If the MATLAB function block contains a relational operation, the relational boundary coverage metric applies to this block.

If the MATLAB function block calls functions containing relational operations multiple times, the relational boundary coverage reports a cumulative result over all instances where the function is called. If a relational operation in the function uses operands of different types in the different calls, relational boundary coverage uses tolerance rules for the stricter operand type. For instance, if a relational operation uses `int32` operands in one call, and `double` operands in another call, relational boundary coverage uses tolerance rules for `double` operands.

For information on the tolerance rules and the order of strictness of types, see "Relational Boundary Coverage" on page 1-7.

How to Collect Coverage for MATLAB Functions

When you simulate your model, the Simulink Coverage software can collect coverage data for MATLAB functions configured for code generation. You enable model coverage from the **Coverage** app.

You collect model coverage for MATLAB functions as follows:

- Functions in a MATLAB Function block
- Functions in an external MATLAB file

  To collect coverage for an external MATLAB file, **Coverage** pane of the Configuration Parameters dialog box, select **Coverage for MATLAB files**.

- Simulink Design Verifier functions:

  - `sldv.condition`
  - `sldv.test`
  - `sldv.assume`
  - `sldv.prove`

  To collect coverage for these functions, on the **Coverage** pane of the Configuration Parameters dialog box, select the **Objectives and Constraints** coverage metric.

The following section provides model coverage examples for each of these situations.

Examples: Model Coverage for MATLAB Functions

- "Model Coverage for MATLAB Function Blocks" on page 5-47
- "Model Coverage for MATLAB Functions in an External File" on page 5-54
- "Model Coverage for Simulink Design Verifier MATLAB Functions" on page 5-55

Model Coverage for MATLAB Function Blocks

Simulink Coverage software measures model coverage for functions in a MATLAB Function block.

The following model contains two MATLAB functions in its MATLAB Function block:
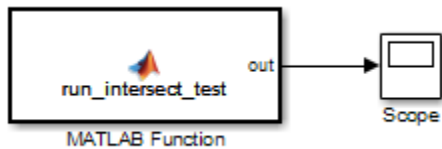
In the Configuration Parameters dialog box, on the **Solver** pane, under **Solver selection**, the simulation parameters are set as follows:

- **Type** — `Fixed-step`
- **Solver** — `discrete (no continuous states)`
- **Fixed-step size (fundamental sample time)** — `1`

The MATLAB Function block contains two functions:

- The top-level function, `run_intersect_test`, sends the coordinates for two rectangles, one fixed and the other moving, as arguments to `rect_intersect`.
- The local function, `rect_intersect`, tests for intersection between the two rectangles. The origin of the moving rectangle increases by `1` in the `x` and `y` directions with each time step.

The coordinates for the origin of the moving test rectangle are represented by persistent data `x1` and `y1`, which are both initialized to `-1`. For the first sample, `x1` and `y1` both increase to `0`. From then on, the progression of rectangle arguments during simulation is as shown in the following graphic.
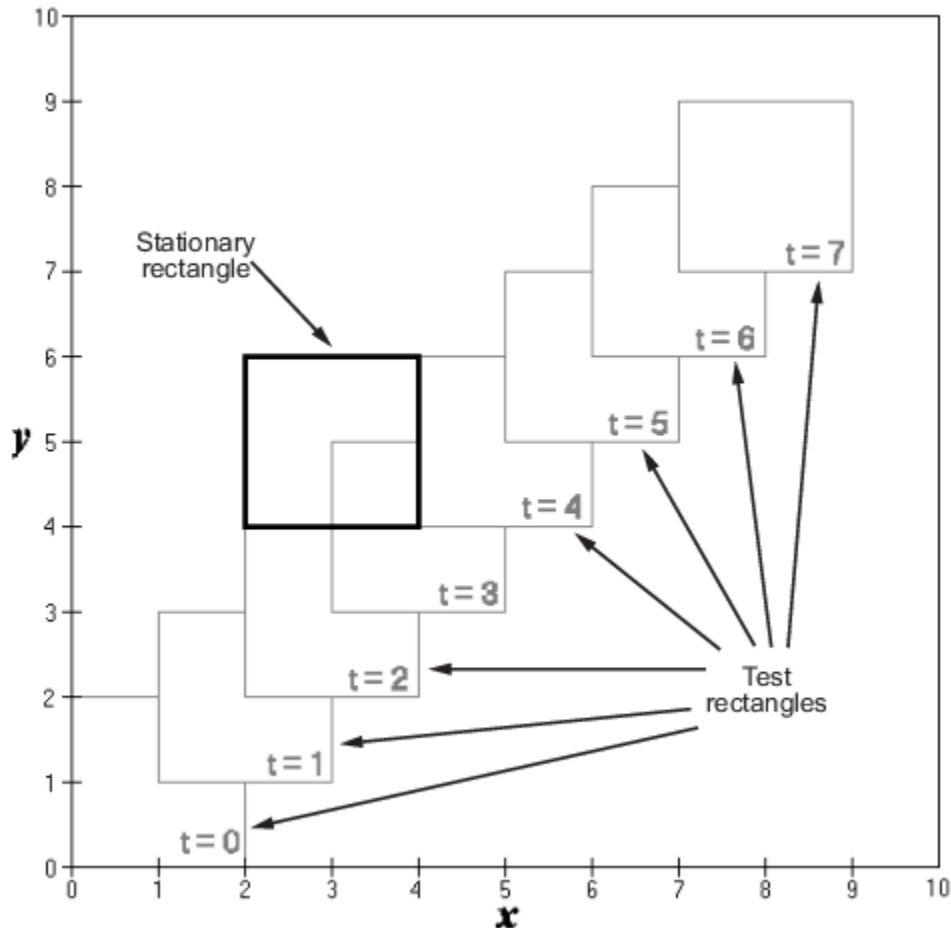
The fixed rectangle is shown in bold with a lower-left origin of (`2,4`) and a width and height of 2. At time `t = 0`, the first test rectangle has an origin of (`0,0`) and a width and height of 2. For each succeeding sample, the origin of the test rectangle increments by (`1,1`). The rectangles at sample times `t = 2`, 3, and 4 intersect with the test rectangle.

The local function `rect_intersect` checks to see if its two rectangle arguments intersect. Each argument consists of coordinates for the lower-left corner of the rectangle (origin), and its width and height. `x` values for the left and right sides and `y` values for the top and bottom are calculated for each rectangle and compared in nested `if-else` decisions. The function returns a logical value of 1 if the rectangles intersect and 0 if they do not.

Scope output during simulation, which plots the return value against the sample time, confirms the intersecting rectangles for sample times 2, 3, and 4 .
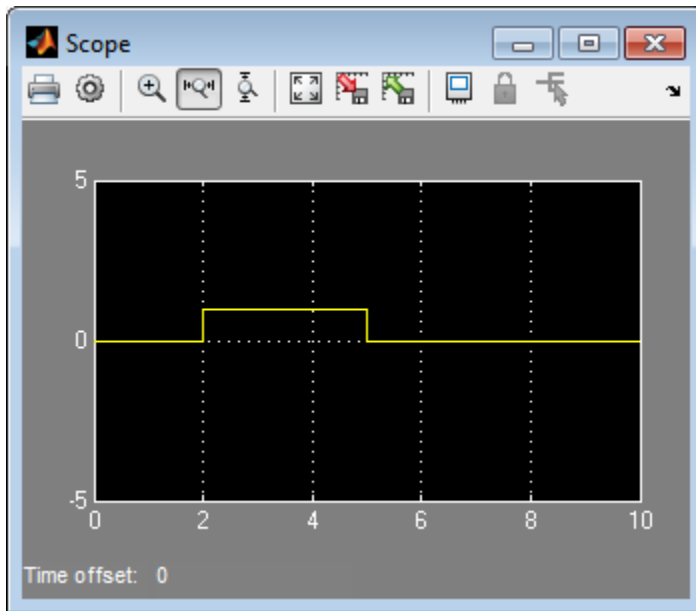
After the simulation, the model coverage report appears in a browser window. After the summary in the report, the Details section of the model coverage report reports on each parts of the model.

The model coverage report for the MATLAB Function block shows that the block itself has no decisions of its own apart from its function.

The following sections examine the model coverage report for the example model in reverse function-block-model order. Reversing the order helps you make sense of the summary information at the top of each section.

Coverage for the MATLAB Function run_intersect_test

Model coverage for the MATLAB Function block function `run_intersect_test` appears under the linked name of the function. Clicking this link opens the function in the editor.

Below the linked function name is a link to the model coverage report for the parent MATLAB Function block that contains the code for `run_intersect_test`.

MATLAB Function "run_intersect_test"

Parent: ex_mc_eml_intersecting_rectangles/MATLAB Function
Uncovered Links:

Metric	Coverage
Cyclomatic Complexity	7
Decision	100% (8/8) decision outcomes
Condition	88% (7/8) condition outcomes
MCDC	75% (3/4) conditions reversed the outcome

The top half of the report for the function summarizes its model coverage results. The coverage metrics for `run_intersect_test` include decision, condition, and MCDC coverage. You can best understand these metrics by examining the code for `run_intersect_test`.

```
 1  function out = run_intersect_test
 2  % Call rect_intersect to see if a moving test rectangle
 3  % and a stationary rectangle intersect
 4
 5  persistent x1 y1;
 6  if isempty(x1)
 7      x1 = -1, y1 = -1;
 8  end
 9
10  x1 = x1 + 1;
11  y1 = y1 + 1;
12  out = rect_intersect([x1 y1 2 2]', [2 4 2 2]');
13
14  function out = rect_intersect(rect1, rect2);
15  % Return 1 if two rectange arguments intersect, 0 if not.
16
17  left1 = rect1(1);
18  bottom1 = rect1(2);
19  right1 = left1 + rect1(3);
20  top1 = bottom1 + rect1(4);
21
22  left2 = rect2(1);
23  bottom2 = rect2(2);
24  right2 = left2 + rect2(3);
25  top2 = bottom2 + rect2(4);
26
27  if (top1 < bottom2 || top2 < bottom1)
28      out = 0;
29  else
30      if (right1 < left2 || right2 < left1)
31          out = 0;
32      else
33          out = 1;
34      end
35  end
```

Lines with coverage elements are marked by a highlighted line number in the listing:

- Line 1 receives decision coverage on whether the top-level function `run_intersect_test` is executed.
- Line 6 receives decision coverage for its `if` statement.
- Line 14 receives decision coverage on whether the local function `rect_intersect` is executed.
- Lines 27 and 30 receive decision, condition, and MCDC coverage for their `if` statements and conditions.

  Each of these lines is the subject of a report that follows the listing.

  The condition `right1 < left2` in line 30 is highlighted in red. This means that this condition was not tested for all of its possible outcomes during simulation. Exactly which of the outcomes was not tested is in the report for the decision in line 30.

The following sections display the coverage for each `run_intersect_test` decision line. The coverage for each line is titled with the line itself, which if clicked, opens the editor to the designated line.

Coverage for Line 1

The coverage metrics for line 1 are part of the coverage data for the function `run_intersect_test`.

The first line of every MATLAB function configured for code generation receives coverage analysis indicative of the decision to run the function in response to a call. Coverage for `run_intersect_test` indicates that it executed at least once during simulation.

#1: function out = run_intersect_test

Decisions analyzed:

function out = run_intersect_test	100%
executed	11/11

Coverage for Line 6

The *Decisions analyzed* table indicates that the decision in line 6, `if isempty(x1)`, executed a total of eight times. The first time it executed, the decision evaluated to `true`, enabling `run_intersect_test` to initialize the values of its persistent data. The remaining seven times the decision executed, it evaluated to `false`. Because both possible outcomes occurred, decision coverage is 100%.

#6: if isempty(x1)

Decisions analyzed:

if isempty(x1)	100%
false	10/11
true	1/11

Coverage for Line 14

The *Decisions analyzed* table indicates that the local function `rect_intersect` executed during testing, thus receiving 100% coverage.

#14: function out = rect_intersect(rect1, rect2);

Decisions analyzed:

function out = rect_intersect(rect1, rect2);	100%
executed	11/11

Coverage for Line 27

The *Decisions analyzed* table indicates that there are two possible outcomes for the decision in line 27: `true` and `false`. Five of the eight times it was executed, the decision evaluated to `false`. The remaining three times, it evaluated to `true`. Because both possible outcomes occurred, decision coverage is 100%.

The *Conditions analyzed* table sheds some additional light on the decision in line 27. Because this decision consists of two conditions linked by a logical OR (`||`) operation, only one condition must evaluate `true` for the decision to be `true`. If the first condition evaluates to `true`, there is no need to evaluate the second condition. The first condition, `top1 < bottom2`, was evaluated eight times, and was `true` twice. This means that the second condition was evaluated only six times. In only one case was it `true`, which brings the total `true` occurrences for the decision to three, as reported in the *Decisions analyzed* table.

MCDC coverage looks for decision reversals that occur because one condition outcome changes from `T` to `F` or from `F` to `T`. The *MCDC analysis* table identifies all possible combinations of outcomes for the conditions that lead to a reversal in the decision. The character `x` is used to indicate a condition outcome that is irrelevant to the decision reversal. Decision-reversing condition outcomes that are not achieved during simulation are marked with a set of parentheses. There are no parentheses, therefore all decision-reversing outcomes occurred and MCDC coverage is complete for the decision in line 27.

#27: if (top1 < bottom2 || top2 < bottom1)

Decisions analyzed:

if (top1 < bottom2 || top2 < bottom1)	100%
false	5/11
true	6/11

Conditions analyzed:

Description:	True	False
top1 < bottom2	2	9
top2 < bottom1	4	5

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition:	True Out	False Out
top1 < bottom2 || top2 < bottom1		
top1 < bottom2	Tx	FF
top2 < bottom1	FT	FF

Coverage for Line 30

The line 30 decision, `if (right1 < left2 || right2 < left1)`, is nested in the `if` statement of the line 27 decision and is evaluated only if the line 27 decision is `false`. Because the line 27

decision evaluated `false` five times, line 30 is evaluated five times, three of which are `false`. Because both the `true` and `false` outcomes are achieved, decision coverage for line 30 is 100%.

Because line 30, like line 27, has two conditions related by a logical OR operator (`||`), condition 2 is tested only if condition 1 is `false`. Because condition 1 tests `false` five times, condition 2 is tested five times. Of these, condition 2 tests `true` two times and `false` three times, which accounts for the two occurrences of the `true` outcome for this decision.

Because the first condition of the line 30 decision does not test `true`, both outcomes do not occur for that condition and the condition coverage for the first condition is highlighted with a rose color. MCDC coverage is also highlighted in the same way for a decision reversal based on the `true` outcome for that condition.

#30: if (right1 < left2 || right2 < left1)

Decisions analyzed:

if (right1 < left2 || right2 < left1)	100%
false	3/5
true	2/5

Conditions analyzed:

Description:	True	False
right1 < left2	0	5
right2 < left1	2	3

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition:	True Out	False Out
right1 < left2 || right2 < left1		
right1 < left2	(Tx)	FF
right2 < left1	FT	FF

Coverage for run_intersect_test

On the *Details* tab, the metrics that summarize coverage for the entire `run_intersect_test` function are reported and repeated as shown.

```
MATLAB Function "run_intersect_test"

    Parent:                    ex_mc_eml_intersecting_rectangles/MATLAB Function
    Uncovered Links:

    Metric                     Coverage
    Cyclomatic Complexity      7
    Decision                   100% (8/8) decision outcomes
    Condition                  88% (7/8) condition outcomes
    MCDC                       75% (3/4) conditions reversed the outcome
```

The results summarized in the coverage metrics summary can be expressed in the following conclusions:

- There are eight decision outcomes reported for `run_intersect_test` in the line reports:

  - One for line 1 (executed)
  - Two for line 6 (`true` and `false`)
  - One for line 14 (executed)
  - Two for line 27 (`true` and `false`)
  - Two for line 30 (`true` and `false`).

  The decision coverage for each line shows 100% decision coverage. This means that decision coverage for `run_intersect_test` is eight of eight possible outcomes, or 100%.

- There are four conditions reported for `run_intersect_test` in the line reports. Lines 27 and 30 each have two conditions, and each condition has two condition outcomes (`true` and `false`), for a total of eight condition outcomes in `run_intersect_test`. All conditions tested positive for both the `true` and `false` outcomes except the first condition of line 30 (`right1 < left2`). This means that condition coverage for `run_intersect_test` is seven of eight, or 88%.

- The MCDC coverage tables for decision lines 27 and 30 each list two cases of decision reversal for each condition, for a total of four possible reversals. Only the decision reversal for a change in the evaluation of the condition `right1 < left2` of line 30 from `true` to `false` did not occur during simulation. This means that three of four, or 75% of the possible reversal cases were tested for during simulation, for a coverage of 75%.

Model Coverage for MATLAB Functions in an External File

Using the same model in "Model Coverage for MATLAB Function Blocks" on page 5-47, suppose the MATLAB functions `run_intersect_test` and `rect_intersect` are stored in an external MATLAB file named `run_intersect_test.m`.

To collect coverage for MATLAB functions in an external file, on the **Coverage** pane of the Configuration Parameters dialog box, select **Coverage for MATLAB files**.

After simulation, the model coverage report summary contains sections for the top-level model and for the external function.

Coverage by Model

	Complexity	Condition	Decision	MCDC
TOTAL COVERAGE		88%	100%	75%
1. . . . run_intersect_test	5	88%	100%	75%
2. . . . intersecting_rectangles1	3	--	100%	--

The model coverage report for `run_intersect_test.m` reports the same coverage data as if the functions were stored in the MATLAB Function block.

For a detailed example of a model coverage report for a MATLAB function in an external file, see "External MATLAB File Coverage Report" on page 6-3.

Model Coverage for Simulink Design Verifier MATLAB Functions

If the MATLAB code includes any of the following Simulink Design Verifier functions configured for code generation, you can measure coverage:

- `sldv.condition`
- `sldv.test`
- `sldv.assume`
- `sldv.prove`

For this example, consider the following model that contains a MATLAB Function block.

The MATLAB Function block contains the following code:

```
function y = fcn(u)
% This block supports MATLAB for code generation.

sldv.condition(u > -30)
sldv.test(u == 30)
y = 1;
```

To collect coverage for Simulink Design Verifier MATLAB functions, on the **Coverage** pane in the Configuration Parameters dialog box, under **Other metrics**, select **Objectives and Constraints**.

After simulation, the model coverage report listed coverage for the `sldv.condition` and `sldv.test` functions. For `sldv.condition`, the expression `u > -30` evaluated to `true` 51 times. For `sldv.test`, the expression `u == 30` evaluated to `true` 51 times.

eM Function "fcn"

Parent:	ex_mc_eml_sldv_blocks/MATLAB Function

Metric	Coverage
Cyclomatic Complexity	1
Decision	100% (1/1) decision outcomes
Test Objective	100% (1/1) objective outcomes
Test Condition	100% (1/1) objective outcomes

```
1  function y = fcn(u)
2  % This block supports MATLAB for code generation.
3
4  sldv.condition(u > -30)
5  sldv.test(u == 30)
6  y = 1;
```

#1: function y = fcn(u)

Decisions analyzed:

function y = fcn(u)	100%
executed	51/51

#4: sldv.condition(u > -30)

Test Condition analyzed:

sldv.condition(u > -30)	51/51

#5: sldv.test(u == 30)

Test Objective analyzed:

sldv.test(u == 30)	51/51

For an example of model coverage data for Simulink Design Verifier blocks, see "Objectives and Constraints Coverage" on page 1-6.

Coverage for MATLAB® Function Blocks

This example model explains how Model Coverage relates to MATLAB code inside a MATLAB Function Block.

MATLAB(R) Function Block Model Coverage Examples

This example model explains how Model Coverage relates to MATLAB code inside a MATLAB Function Block.

Start the simulation from the toolbar or the command line. The Coverage Details Pane containing the coverage report will automatically open at the end of the simulation. To open the Results Explorer, go to the Coverage tab of the Ribbon and click Results Explorer.

Function coverage

Functions in the MATLAB Function Block have model coverage decisions that indicate if the function was called.

If coverage

if statements in the MATLAB Function Block have model coverage decisions that indicate if the *if* statement was taken as true and false. Full coverage requires at least one execution where the *if* statement is true and at least one execution where the statement was false. Coverage requirements do not change when an *else* statement is added.

and-or coverage

if statements in the MATLAB Function Block that have && and || in their expressions have model coverage conditions. Full condition coverage requires that each condition evaluate to true at least once and evaluate to false at least once. Full MCDC coverage requires that each condition independently changes the *if* statement value.

Switch-case coverage

switch-case constructs in the MATLAB Function Block have model coverage decisions that indicate which *case* statements have executed. Full coverage requires that each *case* statement (and the *otherwise* case) be taken at least once. Even if the *otherwise* is not written there must still be an execution where no *case* statement is taken to achieve full coverage.

While coverage

while statements in the MATLAB Function Block have model coverage decisions that indicate if the expression was evaluated to true and false. Full coverage requires at least one evaluation where the expression is true and at least one evaluation where the expression is false.

For coverage

for statements in the MATLAB Function Block have model coverage decisions that indicate if the loop expression was evaluated to true and false. Full coverage requires at least one evaluation where the loop expression is true and at least one evaluation where the loop expression is false.

Assignment coverage

Logical expressions in assignment statements in the MATLAB Function Block that contain && and || have model coverage conditions. Full condition coverage requires that each condition evaluate to true at least once and evaluate to false at least once. Full MCDC coverage requires that each condition independently changes the expression's value.

Copyright 1990-2017 The MathWorks, Inc.

Coverage for Custom C/C++ Code in Simulink Models

When you record coverage for models containing supported C/C++ S-Functions, MATLAB Function blocks that call external C/C++ code, C Caller blocks with C/C++ code , or Stateflow charts that integrate custom C/C++ code for simulation, coverage is recorded for the C/C++ code within the C/C++ S-Functions, MATLAB Function blocks, or Stateflow charts. The coverage results for the custom code can be viewed in the same report as the rest of the model. For each S-Function block, MATLAB Function block, or Stateflow chart, the report links to a detailed coverage report for the C/C++ code in the block.

Enable Code Coverage for Custom C/C++ code in MATLAB Function Blocks, C Caller Blocks, and Stateflow Charts

To enable code coverage for custom C/C++ code in your Simulink model:

1 On the **Simulation Target** pane of the Configuration Parameters, select **Import custom code**.

2 On the **Simulation Target** pane of the Configuration Parameters, select **Enable custom code analysis**.

Simulink Coverage records code coverage for custom C/C++ code in MATLAB Function blocks, C Caller blocks, and Stateflow charts.

Code Coverage for S-Functions

Make S-Function Compatible with Model Coverage

If you use the `legacy_code` function, S-Function Builder block or `mex` function to create your S-Functions, adapt your method appropriately to make the S-Function compatible with model coverage.

For more information on the three approaches, see "Implement C/C++ S-Functions".

- "S-Function Using legacy_code Function" on page 5-59
- "S-Function Using S-Function Builder" on page 5-59
- "S-Function Using mex Function" on page 5-60

S-Function Using legacy_code Function

1 Initialize a MATLAB structure with fields that represent Legacy Code Tool properties.

```
def = legacy_code('initialize')
```

2 To enable model coverage, turn on the option `def.Options.supportCoverage`.

```
def.Options.supportCoverageAndDesignVerifier = true;
```

3 Use the structure `def` in the usual way to generate an S-function. For an example, see "Coverage for S-Functions" on page 5-65.

S-Function Using S-Function Builder

1 Copy an instance of the S-Function Builder block from the **User-Defined Functions** library in the Library Browser into the your model.

2 Double-click the block to open the S-Function Builder dialog box.

3   On the **Build Info** tab, select **Enable support for coverage**.

S-Function Using mex Function

If you use the `mex` function to compile and link your source files, use the `slcovmex` function instead. The `slcovmex` function compiles your source code and also makes it compatible with coverage.

This function has the same syntax and takes the same options as the `mex` function. In addition, you can provide some options relevant for model coverage. For more information, see `slcovmex`.

Generate Coverage Report for S-Function

1   In the Simulink Editor, select **Model Settings** on the **Modeling** tab.

2   On the **Coverage** pane of the Configuration Parameters dialog box, select **C/C++ S-functions**.

When you run a simulation, the coverage report contains coverage metrics for C/C++ S-Function blocks in your model. For each S-Function block, the report links to a detailed coverage report for the C/C++ code in the block.

See Also

Related Examples
*   "View Coverage Results for Custom C/C++ Code in S-Function Blocks" on page 5-61

More About
*   "C/C++ S-Function" on page 2-20

View Coverage Results for Custom C/C++ Code in S-Function Blocks

This example shows how to view coverage results for the C/C++ code in S-Function blocks in your model. To view coverage results for the C/C++ code in the blocks:

- Enable support for S-Function coverage. For more information, see "Coverage for Custom C/C++ Code in Simulink Models" on page 5-59.
- Run simulation and view the coverage report.

  The coverage results for S-Function blocks can be viewed in the same report as the rest of the model. For each S-Function block, the report links to a detailed coverage report for the C/C++ code in the block.

To view the full code coverage report used in this example, follow the steps in "Coverage for S-Functions" on page 5-65.

1 In the coverage report, view the coverage metrics for the S-Function block.

S-Function block "sldemo_sfun_counterbus"

Parent:	sldemo_lct_bus/TestCounter
Uncovered Links:	➡

Metric	Coverage
Cyclomatic Complexity	3
Condition	67% (4/6) condition outcomes
Decision	75% (3/4) decision outcomes
MCDC	50% (1/2) conditions reversed the outcome

Detailed Report:	sldemo_lct_bus_sldemo_sfun_counterbus_instance_1_cov.html

For more information on the coverage report format, see "Top-Level Model Coverage Report" on page 6-10.

2 Select the **Detailed Report** link. The code coverage report for the S-Function block opens.

3 Select each of the links in **Table Of Contents** to navigate to various sections of the report.

Code Coverage Report for S-Function sldemo_sfun_counterbus

Table Of Contents

Section Title	Purpose		
Analysis information	Contains information such as time when model was created and last modified, and file size.		
Tests	Contains information about the simulation such as start and end time.		
Summary	Contains coverage information about the files and functions in the S-Function block. For each file and function, the percentage coverage is displayed. The coverage types relevant for the code are the following:		
	Coverage Type		**Label**
	"Cyclomatic Complexity for Code Coverage" on page 4-4		**Complexity**
	"Condition Coverage for Code Coverage" on page 4-2		**Condition**.
	"Decision Coverage for Code Coverage" on page 4-3		**Decision**
	"Modified Condition/Decision Coverage (MCDC) for Code Coverage" on page 4-3		**MCDC**
	"Relational Boundary for Code Coverage" on page 4-4		**Relational Boundary**
	Percentage of statements covered		**Stmt**
Details	Contains coverage information about the statements that receive condition, decision or MCDC coverage. The information is grouped by file and function.		
Code	Contains the C/C++ code. Statements that are not covered are highlighted in pink.		

4 In the **Summary** section, select each file or function name to see details of coverage for statements in the file or function.

File Contents	Complexity	Decision	Condition	MCDC	Stmt
1. counterbus.c	3	75% ▬	67% ▬	50% ▬	90% ▬
2...counterbusFcn	3	75% ▬	67% ▬	50% ▬	90% ▬

5 The condition, decision or MCDC outcomes that were not tested during simulation are highlighted in pink. Within the details for a file or function, scroll down to note these cases and investigate them further.

2.1 Decision/Condition (u1->limits.upper_saturation_limit >= limit) && inputGElower

Function: counterbusFcn (line 6)

Metric	Coverage
Decision	100% (2/2) decision outcomes
Condition	75% (3/4) condition outcomes
MCDC	50% (1/2) conditions reversed the outcome

Decisions analyzed:

(u1->limits.upper_saturation_limit >= limit) && inputGElower	100%
false	61/201
true	140/201

Conditions analyzed:

Description:	True	False
u1->limits.upper_saturation_limit >= limit	140	61
inputGElower	140	0

6 To obtain an overview of the statements that were not covered, navigate to the **Code** section. This section contains your code with the uncovered statements highlighted in pink.

Code

```
1   /* Copyright 2005-2006 The MathWorks, Inc. */
2
3
4   #include "counterbus.h"
5
6   void counterbusFcn(COUNTERBUS *u1, int32_T u2, COUNTERBUS *y1, int32_T *y2)
7   {
8       int32_T limit;
9       boolean_T inputGElower;
10
11      limit = u1->inputsignal.input + u2;
12
13      inputGElower = (limit >= u1->limits.lower_saturation_limit);
14
15      if((u1->limits.upper_saturation_limit >= limit) && inputGElower) {
16          *y2 = limit;
17      } else {
18
19          if(inputGElower) {
20              limit = u1->limits.upper_saturation_limit;
21          } else {
22              limit = u1->limits.lower_saturation_limit;
23          }
24          *y2 = limit;
25      }
26
27      y1->inputsignal.input = *y2;
28      y1->limits = u1->limits;
29
30  }
31
```

See Also

More About

- "C/C++ S-Function" on page 2-20

Coverage for S-Functions

This example shows how to configure an S-Function generated with the Legacy Code Tool to be compatible with coverage. The model coverage tool supports S-Functions that are:

- Generated with the Legacy Code Tool, with `def.Options.supportCoverage` set to `true`,
- Generated with the SFunctionBuilder, with **Enable support for coverage** selected on the **Build Info** tab of the SFunctionBuilder dialog box, or
- Compiled with the `slcovmex` function.

Open Example Model

The example model slcoverage_lct_bus contains an S-Function generated with the Legacy Code Tool. The S-Function has constructs that receive decision, condition, and MCDC coverage.

Open slcoverage_lct_bus

Configure S-Function to Be Compatible with Model Coverage

The legacy source code in the files counterbus.h, and counterbus.c implements the same algorithm as in slcoverage_lct_bus/slCounter. The Legacy Code Tool data structure is defined as follows:

```
load_system('slcoverage_lct_bus');
open_system('slcoverage_lct_bus/TestCounter');
load slcoverage_lct_data.mat

def = legacy_code('initialize');
def.SFunctionName = 'slcoverage_sfun_counterbus';
def.OutputFcnSpec = 'void counterbusFcn(COUNTERBUS u1[1], int32 u2, COUNTERBUS y1[1], int32 y2[1
def.HeaderFiles   = {'counterbus.h'};
def.SourceFiles   = {'counterbus.c'};
```


To make this S-Function compatible with model coverage, enable the following option:

```
def.Options.supportCoverage = true;
```

Generate and compile the S-Function using the `legacy_code` function:

```
legacy_code('generate_for_sim', def);
```

```
### Start Compiling slcoverage_sfun_counterbus
mex -IC:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\9\tpf9c986ce\ex71096464 -c C:\TEMP\Bdoc21b_1757077_30
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
mex -IC:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\9\tpf9c986ce\ex71096464 C:\TEMP\Bdoc21b_1757077_3096\
```

```
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
mex -IC:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\9\tpf9c986ce\ex71096464 -c C:\TEMP\Bdoc21b_1757077_30
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
mex -IC:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\9\tpf9c986ce\ex71096464 C:\TEMP\Bdoc21b_1757077_3096\
Building with 'Microsoft Visual C++ 2019 (C)'.
MEX completed successfully.
### Finish Compiling slcoverage_sfun_counterbus
### Exit
```

Enable S-Function Coverage

To enable coverage collection for S-Functions, select **C/C++ S-Functions** in the **Coverage pane of the Configurations Parameters** dialog box. Alternatively, set the option through the command line:

```
set_param('slcoverage_lct_bus',...
          'CovMetricStructuralLevel', 'MCDC',...
          'RecordCoverage', 'on',...
          'CovSFcnEnable', 'on',...
          'CovSaveSingleToWorkspaceVar','on'...
          );
```

Run Simulation and Produce Coverage Report

Once you enable coverage data collection, coverage information is automatically recorded when you simulate the model. At the end of the simulation, you can generate an HTML report of coverage information, which is displayed in the built-in MATLAB® web browser.

```
sim('slcoverage_lct_bus', 'StopTime', '20');
cvhtml('coverageResults', covdata);
```

Extract Information from Coverage Data Objects

The `cvdata` object can be used to extract coverage information for S-Functions, just like any other supported model element. For instance, the `decisioninfo` command extracts coverage information from a block path or a block handle. The output is a vector containing the satisfied and total outcomes for a single model object.

```
cov = decisioninfo(covdata, 'slcoverage_lct_bus/TestCounter/slcoverage_sfun_counterbus')


cov =

     3     4
```

You then use this coverage information to calculate the percentage of covered model objects:

```
percentCov = 100 * (cov(1)/cov(2))


percentCov =

    75
```

S-Function coverage is fully compatible with the model coverage commands, such as `decisioninfo`, `conditioninfo`, and `mcdcinfo`.

Model Coverage for Stateflow Charts

How Model Coverage Reports Work for Stateflow Charts

A model coverage report is generated automatically if you simulate your model using the **Run** button. If you did not use the **Run** button, or you loaded coverage data without simulating the model, generate a Model Coverage report using `cvhtml`. For Stateflow charts, Simulink Coverage records the execution of the chart itself and the execution of states, transition decisions, and individual conditions that compose each decision. After simulation ends, the model coverage reports on how thoroughly a model was tested. The report shows:

- How many times each exclusive sub-state is executed or exited from its parent superstate and entered due to parent superstate history
- How many times each transition decision has been evaluated as true or false
- How many times each condition has been evaluated as true or false

Note To measure model coverage data for a Stateflow chart, you must:

- Have a Stateflow license.
- Have debugging/animation enabled for the chart.

Specify Coverage Report Settings for Stateflow Charts

Specify coverage recording settings from the **Coverage** pane of the Configuration Parameters dialog box.

Enabling coverage analysis also enables the selection of different coverage metrics. The following sections address only coverage metrics that affect reports for Stateflow charts. These metrics include decision coverage, condition coverage, and MCDC coverage.

Cyclomatic Complexity for Stateflow Charts

Cyclomatic complexity is a measure of the complexity of a software module based on its edges, nodes, and components within a control-flow chart. It provides an indication of how many times you need to test the module.

The calculation of cyclomatic complexity is as follows:

```
CC = E - N + p
```

where `CC` is the cyclomatic complexity, `E` is the number of edges, `N` is the number of nodes, and `p` is the number of components.

Within the Model Coverage tool, each decision is exactly equivalent to a single control flow node, and each decision outcome is equivalent to a control flow edge. Any additional structure in the control-flow chart is ignored since it contributes the same number of nodes as edges and therefore has no effect on the complexity calculation. Therefore, you can express cyclomatic complexity as follows:

```
CC = OUTCOMES - DECISIONS + p
```

For analysis purposes, each chart counts as a single component.

Decision Coverage for Stateflow Charts

Decision coverage interprets a model execution in terms of underlying decisions where behavior or execution must take one outcome from a set of mutually exclusive outcomes.

Note Full coverage for an object of decision means that every decision has had at least one occurrence of each of its possible outcomes.

Decisions belong to an object making the decision based on its contents or properties. The following table lists the decisions recorded for model coverage for the Stateflow objects owning them. The sections that follow the table describe these decisions and their possible outcomes.

Object	Possible Decisions
Chart	If a chart is a triggered Simulink block, it must decide whether or not to execute its block. If a chart contains exclusive (OR) substates, it must decide which of its states to execute.
State	If a state is a superstate containing exclusive (OR) substates, it must decide which substate to execute. If a state has on *event name* actions (which might include temporal logic operators), the state must decide whether or not to execute the actions.
Transition	If a transition is a conditional transition, it must decide whether or not to exit its active source state or junction and enter another state or junction.

Chart as a Triggered Simulink Block Decision

If the chart is a triggered block in a Simulink model, the decision to execute the block is tested. If the block is not triggered, there is no decision to execute the block, and the measurement of decision coverage is not applicable (NA).

Chart Containing Exclusive OR Substates Decision

If the chart contains exclusive (OR) substates, the decision on which substate to execute is tested. If the chart contains only parallel AND substates, this coverage measurement is not applicable (NA).

Superstate Containing Exclusive OR Substates Decision
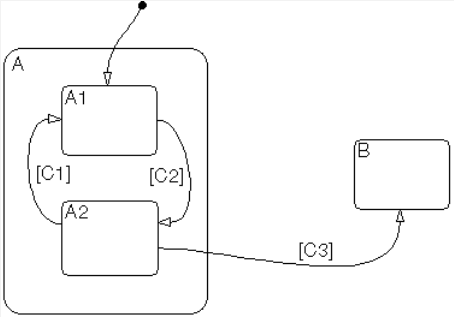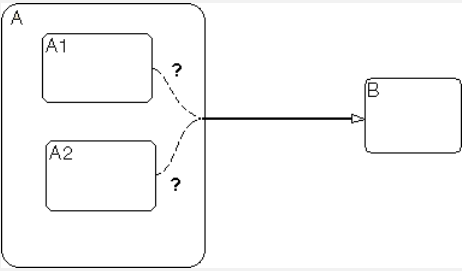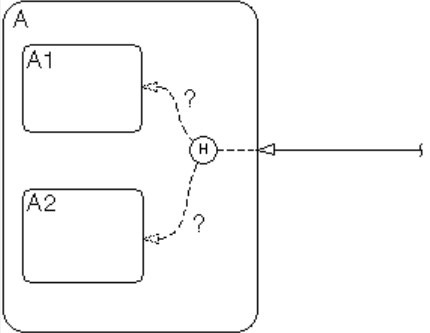
Since a chart is hierarchically processed from the top down, procedures such as exclusive (OR) substate entry, exit, and execution are sometimes decided by the parenting superstate.

Note Decision coverage for superstates applies only to exclusive (OR) substates. A superstate makes no decisions for parallel (AND) substates.

Since a superstate must decide which exclusive (OR) substate to process, the number of decision outcomes for the superstate is the number of exclusive (OR) substates that it contains. In the

examples that follow, the choice of which substate to process can occur in one of three possible contexts.

Note Implicit transitions appear as dashed lines in the following examples.

Context	Example	Decisions That Occur
Active call	States A and A1 are active. 	• The parent of states A and B must decide which of these states to process. This decision belongs to the parent. Since A is active, it is processed. • State A, the parent of states A1 and A2, must decide which of these states to process. This decision belongs to state A. Since A1 is active, it is processed. During processing of state A1, all outgoing transitions are tested. This decision belongs to the transition and not to the parent state A. In this case, the transition marked by condition C2 is tested and a decision is made whether to take the transition to A2 or not.
Implicit substate exit	A transition takes place whose source is superstate A and whose destination is state B. 	If the superstate has two exclusive (OR) substates, it is the decision of superstate A which substate performs the implicit transition from substate to superstate.
Substate entry with a history junction	A history junction records which substate was last active before the superstate was exited. 	If that superstate becomes the destination of one or more transitions, the history junction decides which previously active substate to enter.

For more information, see "State Details Report Section" on page 5-76.

State with On Event_Name Action Statement Decision

A state that has an on *event_name* action statement must decide whether to execute that statement based on the reception of a specified event or on an accumulation of the specified event when using temporal logic operators.

Conditional Transition Decision

A conditional transition is a transition with a triggering event and/or a guarding condition. In a conditional transition from one state to another, the decision to exit one state and enter another is credited to the transition itself.

Note Only conditional transitions receive decision coverage. Transitions without decisions are not applicable to decision coverage.

Condition Coverage for Stateflow Charts

Condition coverage reports on the extent to which all possible outcomes are achieved for individual subconditions composing a transition decision or for logical expressions in assignment statements in states and transitions.

For example, for the decision [A & B & C] on a transition, condition coverage reports on the true and false occurrences of each of the subconditions A, B, and C. This results in eight possible outcomes: true and false for each of three subconditions.

Outcome	A	B	C
1	T	T	T
2	T	T	F
3	T	F	T
4	T	F	F
5	F	T	T
6	F	T	F
7	F	F	T
8	F	F	F

For more information, see "Transition Details Report Section" on page 5-78.

MCDC Coverage for Stateflow Charts

The Modified Condition Decision/Coverage (MCDC) option reports a test's coverage of occurrences in which changing an individual subcondition within a logical expression results in changing the entire expression from true to false or false to true.

For example, if a transition executes on the condition [C1 & C2 & C3 | C4 & C5], the MCDC report for that transition shows actual occurrences for each of the five subconditions (C1, C2, C3, C4, C5) in which changing its result from true to false is able to change the result of the entire condition from true to false.

Relational Boundary Coverage for Stateflow Charts

If a transition in a Stateflow chart involves a relational operation, it receives relational boundary coverage. For more information, see "Relational Boundary Coverage" on page 1-7.

Simulink Design Verifier Coverage for Stateflow Charts

You can use the following Simulink Design Verifier functions inside Stateflow charts:

- `sldv.condition`
- `sldv.test`
- `sldv.assume`
- `sldv.prove`

If you do not have a Simulink Design Verifier license, you can collect model coverage for a Stateflow chart containing these functions, but you cannot analyze the model using the Simulink Design Verifier software.

When you specify the **Objectives and Constraints** coverage metric in the **Coverage** pane of the Configuration Parameters dialog box, the Simulink Coverage software records coverage for these functions.

Each of these functions evaluates an expression *expr*, for example, `sldv.test(`*expr*`)`, where *expr* is any valid Boolean MATLAB expression. Simulink Design Verifier coverage measures the number of time steps that the expression *expr* evaluates to `true`.

If *expr* is `true` for at least one time step, Simulink Design Verifier coverage for that function is 100%. Otherwise, the Simulink Coverage software reports coverage for that function as 0%.

Consider a model that contains this Stateflow chart:
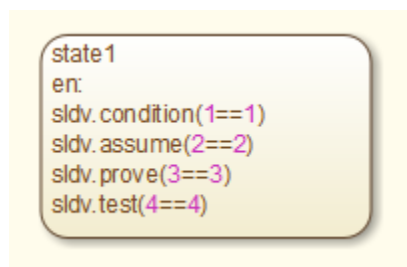
To collect coverage for Simulink Design Verifier functions, on the **Coverage** pane in the Configuration Parameters dialog box, select **Objectives and Constraints**.

After simulation, the model coverage report lists coverage for the `sldv.condition`, `sldv.assume`, `sldv.prove`, and `sldv.test` functions.

Metric	Coverage
Cyclomatic Complexity	0
Proof Assumption	100% (1/1) objective outcomes
Test Condition	100% (1/1) objective outcomes
Proof Objective	100% (1/1) objective outcomes
Test Objective	100% (1/1) objective outcomes

Proof Assumption analyzed:

sldv.assume(2==2)	1/1

Test Condition analyzed:

sldv.condition(1==1)	1/1

Proof Objective analyzed:

sldv.prove(3==3)	1/1

Test Objective analyzed:

sldv.test(4==4)	1/1

Model Coverage Reports for Stateflow Charts

- "Summary Report Section" on page 5-73
- "Subsystem and Chart Details Report Sections" on page 5-74
- "State Details Report Section" on page 5-76
- "Transition Details Report Section" on page 5-78

The following sections of a Model Coverage report were generated by simulating the `sf_boiler` model, which includes the Bang-Bang Controller chart. The coverage metrics for **MCDC** are enabled for this report.

Summary Report Section

The Summary section shows coverage results for the entire test and appears at the beginning of the Model Coverage report.

Summary

Model Hierarchy/Complexity:		Test 1		
		D1	C1	MCDC
1. sf_boiler	20	89%	71%	43%
2. . . . Bang-Bang Controller	16	95%	71%	43%
3. . . . . . . SF: Bang-Bang Controller	15	95%	71%	43%
4. . . . . SF: Heater	12	94%	71%	43%
5. . . . . SF: Off	2	100%	75%	50%
6. . . . . SF: On	4	88%	NA	NA
7. . . . . SF: flash_LED	1	100%	NA	NA
8. . . . . SF: turn_boiler	1	100%	NA	NA
9. . . . Boiler Plant model	3	67%	NA	NA
10. . . . . . . digital thermometer	2	50%	NA	NA
11. . . . . ADC	2	50%	NA	NA

Each line in the hierarchy summarizes the coverage results at that level and the levels below it. You can click a hyperlink to a later section in the report with the same assigned hierarchical order number that details that coverage and the coverage of its children.

The top level, `sf_boiler`, is the Simulink model itself. The second level, Bang-Bang Controller, is the Stateflow chart. The next levels are superstates within the chart, in order of hierarchical containment. Each superstate uses an SF: prefix. The bottom level, Boiler Plant model, is an additional subsystem in the model.

Subsystem and Chart Details Report Sections

When recording coverage for a Stateflow chart, the Simulink Coverage software reports two types of coverage for the chart—Subsystem and Chart.

- *Subsystem* — This section reports coverage for the chart:

  - *Coverage (this object)*: Coverage data for the chart as a container object

  - *Coverage (inc.) descendants*: Coverage data for the chart and the states and transitions in the chart.

If you click the hyperlink of the subsystem name in the section title, the Bang-Bang Controller block is highlighted in the block diagram.

Decision coverage is not applicable (NA) because this chart does not have an explicit trigger. Condition coverage and MCDC are not applicable (NA) for a chart, but apply to its descendants.

2. SubSystem block "Bang-Bang Controller"

Parent:	/sf_boiler
Child Systems:	Bang-Bang Controller

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	16
Condition (C1)	NA	71% (10/14) condition outcomes
Decision (D1)	NA	95% (21/22) decision outcomes
MCDC (C1)	NA	43% (3/7) conditions reversed the outcome

- *Chart* — This section reports coverage for the chart:

  - *Coverage (this object)*: Coverage data for the chart and its inputs

  - *Coverage (inc.) descendants*: Coverage data for the chart and the states and transitions in the chart.

  If you click the hyperlink of the chart name in the section title, the chart opens in the Stateflow Editor.

  Decision coverage is listed appears for the chart and its descendants. Condition coverage and MCDC are not applicable (NA) for a chart, but apply to its descendants.

3. Chart "Bang-Bang Controller"

Parent:	sf_boiler/Bang-Bang Controller
Child Systems:	Heater, flash_LED, turn_boiler

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	15
Condition (C1)	NA	71% (10/14) condition outcomes
Decision (D1)	100% (2/2) decision outcomes	95% (21/22) decision outcomes
MCDC (C1)	NA	43% (3/7) conditions reversed the outcome

Decisions analyzed:

Substate executed	100%
State "Off"	1160/1400
State "On"	240/1400

State Details Report Section

For each state in a chart, the coverage report includes a *State* section with details about the coverage recorded for that state.

In the sf_boiler model, the state On resides in the box Heater. On is a superstate that contains:

- Two substates HIGH and NORM
- A history junction
- The function warm
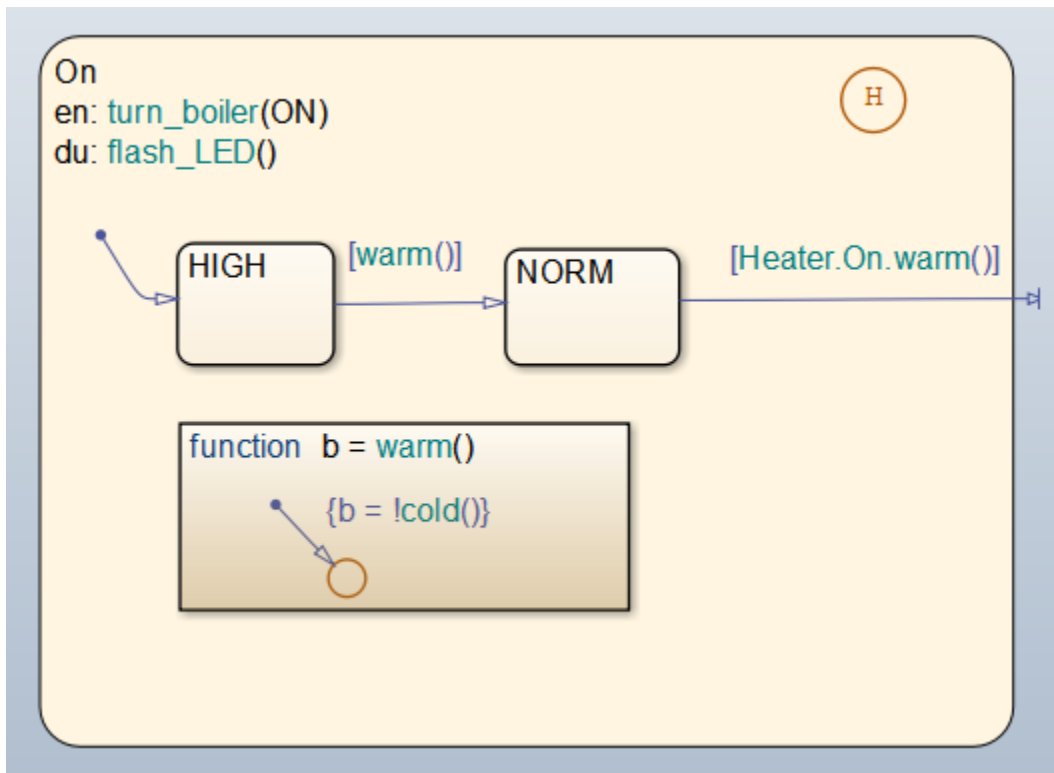
The coverage report includes a *State* section on the state On.

6. State "On"

Justify or Exclude

Parent: sf_boiler/Bang-Bang Controller.Heater

Uncovered Links: ➡

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	3	4
Decision	83% (5/6) decision outcomes	88% (7/8) decision outcomes

Decisions analyzed

Substate executed	100%
State "HIGH"	150/233
State "NORM"	83/233
Substate exited when parent exits	50%
State "HIGH"	7/7
State "NORM"	0/7
Previously active substate entered due to history	100%
State "HIGH"	7/28
State "NORM"	21/28

The decision coverage for the `On` state tests the decision of which substate to execute.

The three decisions are listed in the report:

- Under *Substate executed*, which substate to execute when `On` executes.
- Under *Substate exited when parent exited*, which substate is active when `On` exits. `NORM` is listed as never being active when `On` exits because the coverage tool sees the supertransition from `NORM` to `Off` as a transition from `On` to `Off`.
- Under *Previously active substate entered due to history*, which substate to reenter when `On` re-executes. The history junction records the previously active substate.

Because each decision can result in either `HIGH` or `NORM`, the total possible outcomes are 3 × 2 = 6. The results indicate that five of six possible outcomes were tested during simulation.

Cyclomatic complexity and decision coverage also apply to descendants of the `On` state. The decision required by the condition `[warm()]` for the transition from HIGH to NORM brings the total possible decision outcomes to 8. Condition coverage and MCDC are not applicable (`NA`) for a state.

Note Nodes and edges that make up the cyclomatic complexity calculation have no direct relationship with model objects (states, transitions, and so on). Instead, this calculation requires a graph representation of the equivalent control flow.

Transition Details Report Section

Reports for transitions appear under the report sections of their owning objects. Transitions do not appear in the model hierarchy of the Summary section, since the hierarchy is based on superstates that own other Stateflow objects.

Transition "after(40,sec) [cold()]" from "Off" to "On"

Parent:	sf_boiler/Bang-Bang Controller.Heater
Uncovered Links:	⬅ ➡

Metric	Coverage
Cyclomatic Complexity	3
Condition (C1)	67% (4/6) condition outcomes
Decision (D1)	100% (2/2) decision outcomes
MCDC (C1)	33% (1/3) conditions reversed the outcome

Decisions analyzed:

Transition trigger expression	100%
false	1131/1160
true	29/1160

Conditions analyzed:

Description:	True	False
Condition 1, "sec"	1160	0
Condition 2, "after(40,sec)"	29	1131
Condition 3, "cold()"	29	0

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition:	True Out	False Out
Transition trigger expression		
Condition 1, "sec"	TTT	(Fxx)
Condition 2, "after(40,sec)"	TTT	TFx
Condition 3, "cold()"	TTT	(TTF)

The decision for this transition depends on the time delay of 40 seconds and the condition `[cold()]`. If, after a 40 second delay, the environment is cold (`cold() = 1`), the decision to execute this transition and turn the Heater on is made. For other time intervals or environment conditions, the decision is made not to execute.

For decision coverage, both true and false outcomes occurred. Because two of two decision outcomes occurred, coverage was full or 100%.

Condition coverage shows that only 4 of 6 condition outcomes were tested. The temporal logic statement `after(40,sec)` represents two conditions: the occurrence of `sec` and the time delay `after(40,sec)`. Therefore, three conditions on the transition exist: `sec`, `after(40,sec)`, and `cold()`. Since each of these decisions can be true or false, six possible condition outcomes exist.

The **Conditions analyzed** table shows each condition as a row with the recorded number of occurrences for each outcome (true or false). Decision rows in which a possible outcome did not occur are shaded. For example, the first and the third rows did not record an occurrence of a false outcome.

In the MCDC report, all sets of occurrences of the transition conditions are scanned for a particular pair of decisions for each condition in which the following are true:

- The condition varies from true to false.
- All other conditions contributing to the decision outcome remain constant.
- The outcome of the decision varies from true to false, or the reverse.

For three conditions related by an implied AND operator, these criteria can be satisfied by the occurrence of these conditions.

Condition Tested	True Outcome	False Outcome
1	**T**TT	**F**xx
2	T**T**T	T**F**x
3	TT**T**	TT**F**

Notice that in each line, the condition tested changes from true to false while the other condition remains constant. Irrelevant contributors are coded with an "x" (discussed below). If both outcomes occur during testing, coverage is complete (100%) for the condition tested.

The preceding report example shows coverage only for condition 2. The false outcomes required for conditions 1 and 3 did not occur, and are indicated by parentheses for both conditions. Therefore, condition rows 1 and 3 are shaded. While condition 2 has been tested, conditions 1 and 3 have not and MCDC is 33%.

For some decisions, the values of some conditions are irrelevant under certain circumstances. For example, in the decision [C1 & C2 & C3 | C4 & C5] the left side of the | is false if any one of the conditions C1, C2, or C3 is false. The same applies to the right side result if either C4 or C5 is false. When searching for matching pairs that change the outcome of the decision by changing one condition, holding some of the remaining conditions constant is irrelevant. In these cases, the MCDC report marks these conditions with an "x" to indicate their irrelevance as a contributor to the result. These conditions appear as shown.

Consider the first matched pair. Since condition 1 is true in the **True** outcome column, it must be false in the matching **False** outcome column. This makes the conditions C2 and C3 irrelevant for the false outcome since C1 & C2 & C3 is always false if C1 is false. Also, since the false outcome is required to evaluate to false, the evaluation of C4 & C5 must also be false. In this case, a match was found with C4 = F, making condition C5 irrelevant.

Model Coverage for Stateflow State Transition Tables

State transition tables are an alternative way of expressing modal logic in Stateflow. Stateflow charts represent modal logic graphically, and state transition tables can represent equivalent modal logic in tabular form. For more information, see "State Transition Tables" (Stateflow).

Coverage results for state transition tables are the same as coverage results for equivalent Stateflow charts, except for a slight difference that arises in coverage of temporal logic. For example, consider the temporal logic expression `after(4, tick)` in the Mode Logic chart of the `slvnvdemo_covfilt` example model.
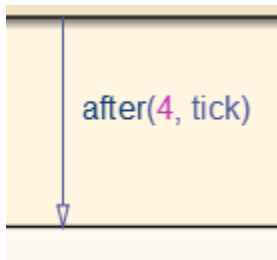
In chart coverage, the `after(4, tick)` transition represents two conditions: the occurrence of `tick` and the time delay `after(4, tick)`. Since the temporal event `tick` is never false, the first condition is not satisfiable, and you cannot record 100% condition and MCDC coverage for the transition `after(4, tick)`.

In state transition table coverage, the `after(4, tick)` transition represents a single decision, with no subcondition for the occurrence of `tick`. Therefore, only decision coverage is recorded.

For state transition tables containing temporal logic decisions, as in the above example, condition coverage and MCDC is not recorded.

Model Coverage for Stateflow Atomic Subcharts

In a Stateflow chart, an atomic subchart is a graphical object that allows you to reuse the same state or subchart across multiple charts and models.

When you specify to record coverage data for a model during simulation, the Simulink Coverage software records coverage for any atomic subcharts in your model. The coverage data records the execution of the chart itself, and the execution of states, transition decisions, and individual conditions that compose each decision in the atomic subchart.

Simulate the `sf_atomic_subcharts_map_iodata_fixed` example model and record decision coverage:

1   Open the Stateflow example **Map Input and Output Data for an Atomic Subchart**.

    `openExample(``'stateflow/AtomicSubchartInOutDataExample'``);`

    Open the `sf_atomic_subcharts_map_iodata_fixed` model.

    `open_system(``'sf_atomic_subcharts_map_iodata_fixed'``)`

    This model contains two Sine Wave blocks that supply input signals to the Stateflow chart. This chart contains two atomic subcharts—A and B—that are linked from the same library chart, also named A. The library chart contains the following objects:
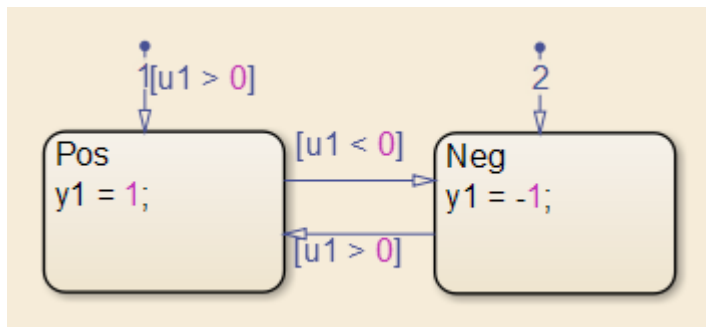
2   In the Simulink Editor, select **Model Settings** on the **Modeling** tab. Select the **Coverage** pane of the Configuration Parameters dialog box.

3   Select **Enable coverage analysis** and then select **Entire System**.

4   Click **OK** to close the Configuration Parameters dialog box.

5   Simulate the `sf_atomic_subcharts_map_iodata_fixed` model by clicking the **Run** button.

    When the simulation completes, the coverage report opens.

The report provides coverage data for atomic subcharts A and B in the following forms:

•   For the atomic subchart instance and its contents. Decision coverage is not applicable (NA) because this chart does not have an explicit trigger.

4. Atomic Subchart "A"

Justify or Exclude

Parent:	sf_atomic_subcharts_map_iodata_fixed/Chart
Child Systems:	A

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	4
Decision	NA	88% (7/8) decision outcomes

- For the library chart A and its contents. The chart itself achieves 100% coverage on the input u1, and 88% coverage on the states and transitions inside the library chart.

5. Chart "A"

Justify or Exclude

Parent:	sf_atomic_subcharts_map_iodata_fixed/Chart.A

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	4
Decision	100% (2/2) decision outcomes	88% (7/8) decision outcomes

Decisions analyzed

Substate executed	100%
State "Neg"	4/10
State "Pos"	6/10

Atomic subchart B is a copy of the same library chart A. The coverage of the contents of subchart B is identical to the coverage of the contents of subchart A.

Model Coverage for Stateflow Truth Tables

- "Types of Coverage in Stateflow Truth Tables" on page 5-82
- "Analyze Coverage in Stateflow Truth Tables" on page 5-83

Types of Coverage in Stateflow Truth Tables

Simulink Coverage reports model coverage for the decisions the objects make in a Stateflow chart during simulation. The report includes coverage for the decisions the truth table functions make.

For this type of truth table...	The report includes coverage data for...
Stateflow Classic	Conditions only.
MATLAB	Conditions and only those actions that have decision points. **Note** With the MATLAB for code generation action language, you can specify decision points in actions using control flow constructs, such as loops and switch statements.

Note To measure model coverage data for a Stateflow truth table, you must have a Stateflow license. For more information about Stateflow truth tables, see "Obtain Cumulative Coverage for Reusable Subsystems and Stateflow® Constructs" on page 5-34.

Analyze Coverage in Stateflow Truth Tables

If you have a Stateflow license, you can generate a model coverage report for a truth table.

Consider the following model.
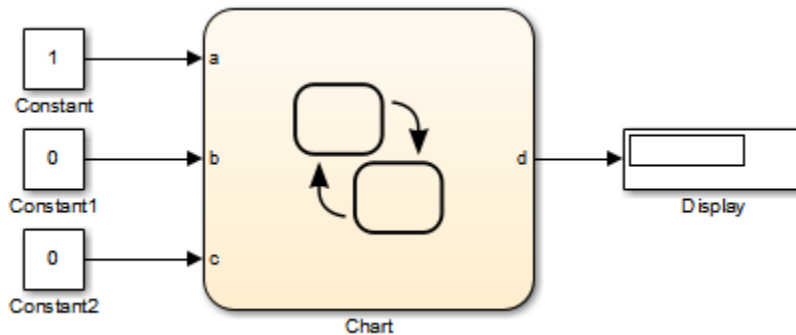
The Stateflow chart contains the following truth table:

Condition Table

	DESCRIPTION	CONDITION	D1	D2	D3	D4
1	x is equal to 1	XEQ1: x == 1	T	F	F	-
2	y is equal to 1	YEQ1: y == 1	F	T	F	-
3	z is equal to 1	ZEQ1: z == 1	F	F	T	-
		ACTIONS: SPECIFY A ROW FROM THE ACTION TABLE	A1	A2	A3	A4

Action Table

	DESCRIPTION	ACTION
1	Initial action: Display message	INIT: ml.disp('truth table ttable entered');
2	set t to 1	A1: t=1;
3	set t to 2	A2: t=2;
4	set t to 3	A3: t=3;
5	set t to 4	A4: t=4;
6	Final action: Display message	FINAL: ml.disp('truth table ttable exited');

When you simulate the model and collect coverage, the model coverage report includes the following data:

4. Truth Table "ttable"

Justify or Exclude

Parent: ex_first_truth_table/Chart

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	9
Condition	NA	17% (3/18) condition outcomes
Decision	NA	17% (1/6) decision outcomes
MCDC	NA	0% (0/9) conditions reversed the outcome

Condition table analysis (missing values are in parentheses)

x is equal to 1	XEQ1: x == 1	T (F)	F (TF)	F (TF)	-
y is equal to 1	YEQ1: y == 1	F (T)	T (TF)	F (TF)	-
z is equal to 1	ZEQ1: z == 1	F (T)	F (TF)	T (TF)	-
	Actions	A1 (F)	A2 (TF)	A3 (TF)	A4

The **Coverage (this object)** column shows no coverage. The reason is that the container object for the truth table function—the Stateflow chart—does not decide whether to execute the `ttable` truth table.

The **Coverage (inc. descendants)** column shows coverage for the truth table contents. Coverage for the descendants in the **Coverage (inc. descendants)** column includes coverage for the conditions and decisions of the logical expressions represented by the truth table.

Coverage for the decisions and their individual conditions in the `ttable` truth table function are as follows:

Coverage	Explanation
No model coverage for the default decision, **D4**	All logic that leads to taking a default decision is based on a false outcome for all preceding decisions. This means that the default decision requires no logic, so there is no model coverage.

5-85

Coverage	Explanation
17% (1/6) decision coverage	The three constants that are inputs to the truth table (1, 0, 0) cause only decision *D1* to be true. These inputs satisfy only one of the six decisions (*D1* through *D3*, T or F). Because each condition can have an outcome value of T or F, three conditions can have six possible values.
17% (3/18) condition coverage	Three decisions *D1*, *D2*, and *D3* have condition coverage, because the set of inputs (1, 0, 0) make only decision *D1* true.
0% (0/9) MCDC coverage	MCDC coverage looks for decision reversals that occur because one condition outcome changes from T to F or F to T. The simulation tests only one set of inputs, so the model reverses no decisions.
Missing coverage	The red letters T and F indicate that model coverage is missing for those conditions. For decision *D1*, only the T decision is satisfied. For decisions *D2*, *D3*, and *D4*, none of the conditions are satisfied.

Model Coverage Display for Stateflow Charts

Simulink Coverage displays model coverage results for individual blocks directly in Stateflow charts. When you simulate your model with coverage enabled, the model displays:

• Highlighting for Stateflow elements that receive model coverage during simulation
• A context-sensitive display of summary model coverage information for each object

For details on enabling coverage highlighting, see "Enable Coverage Highlighting" on page 5-22.

Display Model Coverage with Model Coloring

When you enable coverage and simulate the model with the **Run** button, the model highlights individual Stateflow elements receiving coverage. If you run your model using `sim` the model does not display coverage results by default. In this case, you can see the model highlighting by using `cvmodelview`.

1. Open the `sf_car` model from "Simulate Chart as a Simulink Block With Local Events" (Stateflow).
2. In the **Modeling** tab, click **Model Settings**.
3. In the **Coverage** pane of the Configuration Parameters dialog box, select **Enable coverage analysis**.
4. In the **Coverage metrics** section, set **Structural coverage level** to `Modified Condition Decision Coverage (MCDC)`.
5. Click **OK**.
6. Simulate the model by clicking the **Run (Coverage)** button.
7. Open the shift_logic Stateflow chart.

After simulation ends, the model highlights the chart objects that were analyzed for coverage.
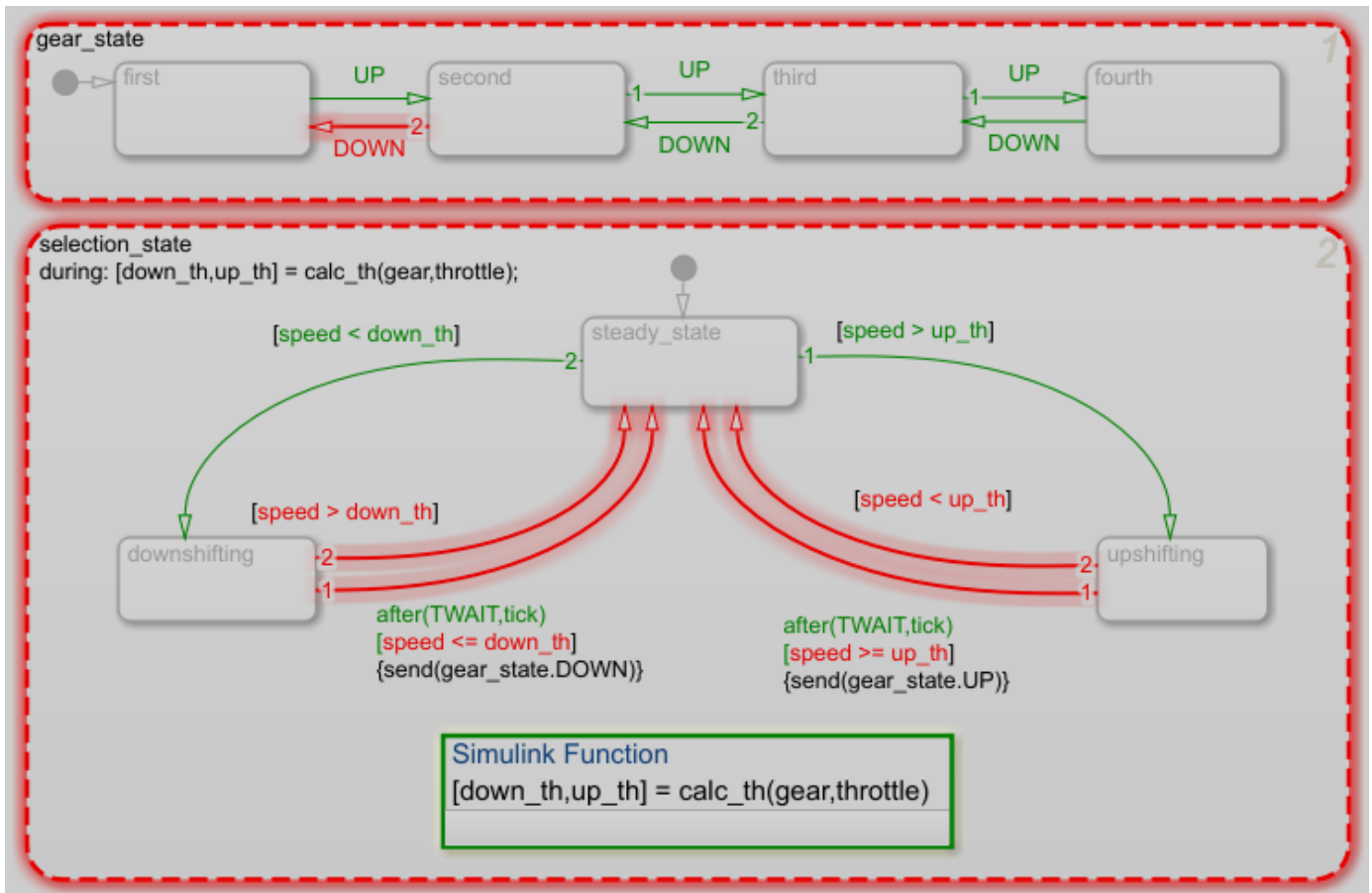
The colors indicate the completeness of coverage analysis:

- Green border for full coverage
- Red border for partial or missing coverage
- Light grey for elements not analyzed for coverage

States that include executable code and conditional transitions that use MATLAB as the action language display granular text coloring based on which outcomes are satisfied. Green indicates satisfied outcomes and red indicates unsatisfied outcomes. For example, consider the following chart:
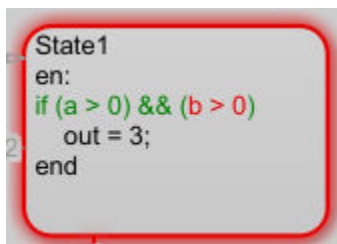
In this example, the `if` statement has evaluated to both true and false and therefore has full decision coverage. Within the statement, condition `a > 0` evaluated to both true and false and has full

condition coverage. Condition $b > 0$, however, evaluated to true but not false and therefore has only partial condition coverage.

Code Coverage for C/C++ code in Stateflow Charts

Simulink Coverage can record code coverage if your Stateflow chart contains custom C/C++ code. For more information, see "Coverage for Custom C/C++ Code in Simulink Models" on page 5-59.

Model Coverage for Variant Blocks

Simulink Coverage can analyze and report coverage data for models containing variant blocks, such as Variant Source or Variant Subsystem blocks. When you simulate a model containing one or more variant blocks with coverage enabled, Simulink Coverage reports the coverage results depending on the type of variant block and the **Variant activation time** parameter.

For Variant Source and Variant Sink blocks, Simulink Coverage analyzes the upstream and downstream blocks of the variant choices, respectively. The Variant Source and Variant Sink block itself does not receive coverage. For Variant Subsystem and Variant Model blocks, Simulink Coverage analyzes the contents of the subsystem or model blocks for the selected metrics.

## Update-Time and Compile-Time Variants

If you set the **Variant activation time** parameter of a variant block to `update diagram`, `update diagram analyze all choices`, or `code compile`, Simulink Coverage reports coverage only for the variant choice that is active during the simulation.

Tip You cannot aggregate coverage for simulations with different active variant choices for update-time or compile-time variants. If you want to test all possible variant choices, consider setting **Variant activation time** to `startup`.

For example, suppose you have a model that contains a Variant Subsystem block with two variant choices. The **Variant activation time** parameter of the Variant Subsystem block is set to `update diagram`.
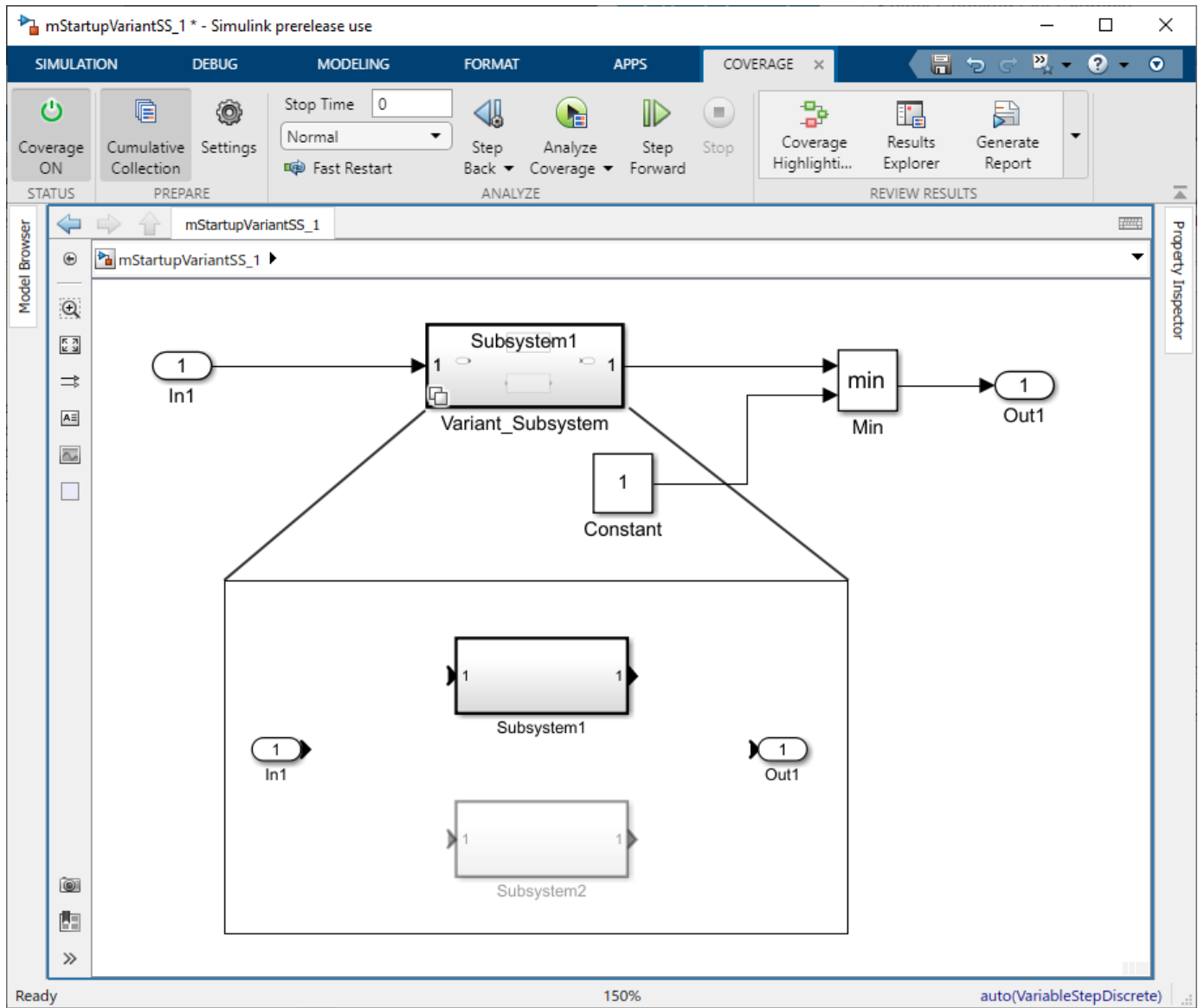
If `Subsystem1` is active for the simulation, you receive coverage results for `Subsystem1` and not `Subsystem2`.
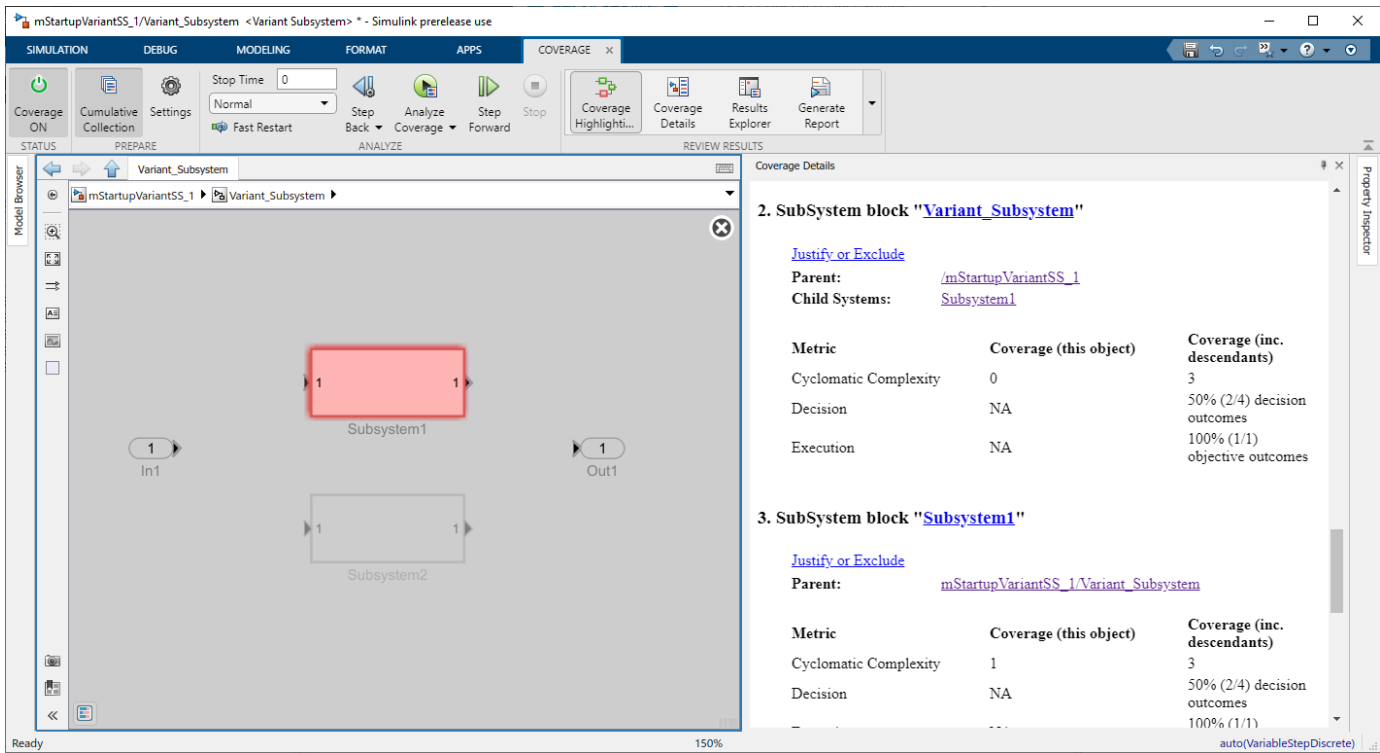
In the coverage results, `Subsystem1` is red, which indicates that it received incomplete coverage, and `Subsystem2` is grey which indicates that it was not analyzed. Additionally, the coverage report summary omits the variant choices that were not active.

When you set the **Variant activation time** parameter of a variant block to `update diagram`, `update diagram analyze all choices`, or `code compile`, changing the active variant is a structural change to the model. Because the coverage data for a model is only valid as long as the model is not changed or closed, you cannot aggregate coverage for the same model simulated with different active variants. If you want to test multiple active variants and aggregate coverage for them, change the **Variant activation time** parameter to `startup`.

Startup Variants

If you set the **Variant activation time** parameter of a variant block to `startup`, Simulink Coverage reports coverage for all variant choices, even if they are not active during the simulation. You can also aggregate coverage for simulations with different active variant choices.

For example, if you modify the model in the previous example and set the **Variant activation time** to `startup` and Subsystem1 is active for the simulation, you get the same coverage for `Subsystem1` as the case with the update time variant. However, Simulink Coverage also reports on the inactive variant, `Subsystem2`.
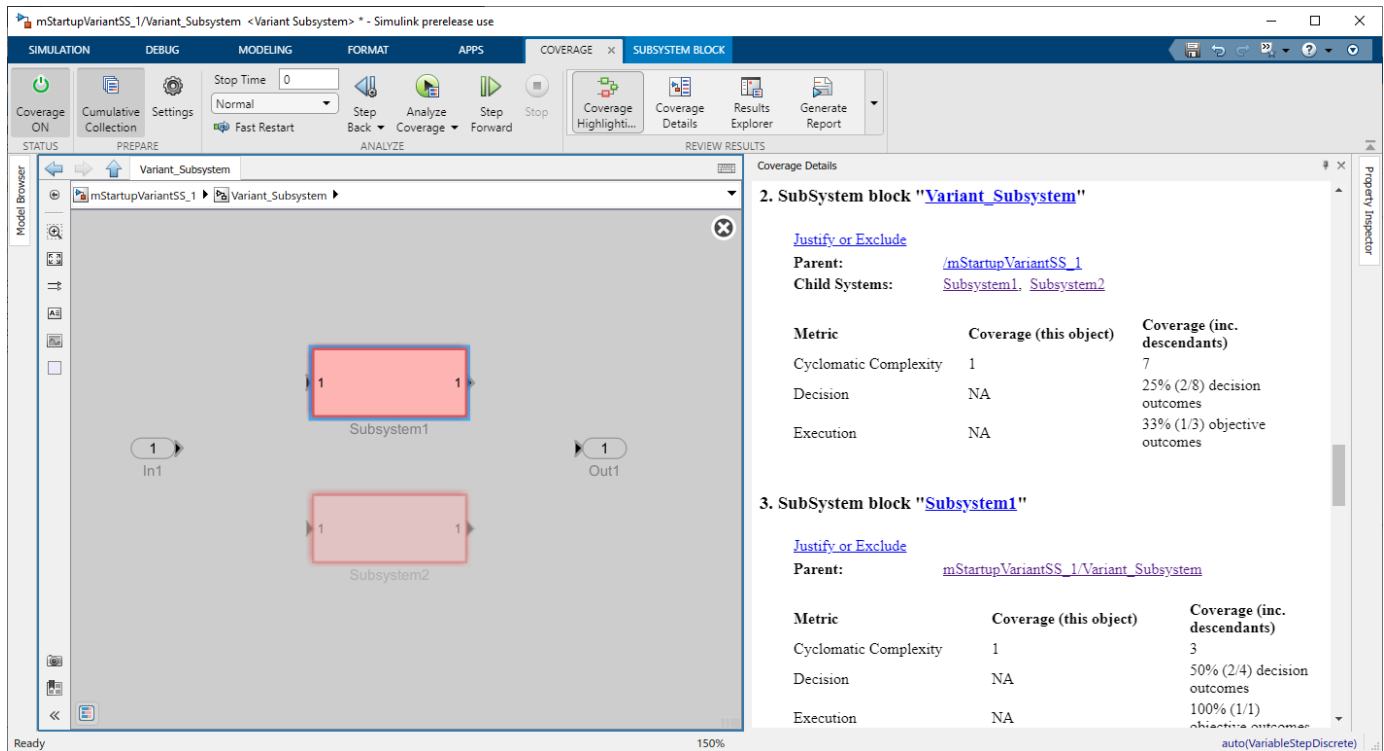
In the coverage results, `Subsystem1` and `Subsystem2` are both red which indicates incomplete coverage. The coverage report summary includes all variant choices.

The inactive variants receive 0% coverage because they were not executed during the simulation. To increase the reported coverage for `Variant_Subsystem`, you can simulate the model again with `Subsystem2` as the active variant and then aggregate the coverage results.

Aggregated Tests

Run	Test Name	Date
Model: "mStartupVariantSS_1"		
T1	Run 14	23-Jun-2021 16:29:15
T2	Run 15	23-Jun-2021 16:49:21

Summary

Model Hierarchy/Complexity

		Decision	Execution
1. mStartupVariantSS_1	9	60%	100%
2. ... Variant_Subsystem	7	50%	100%
3. ...... Subsystem1	3	50%	100%
4. ...... Subsystem2	3	50%	100%

If you want to use startup variant blocks, but you do not want to see inactive variants reported with 0% coverage, you can create exclusion filter rules. For more information about creating coverage filter rules, see "Create, Edit, and View Coverage Filter Rules" on page 7-6.

See Also

Related Examples

- "Variant Systems"
- "Cumulative Coverage Analysis" on page 3-15
- "Model Objects That Receive Coverage" on page 2-2
- "Creating and Using Coverage Filters" on page 7-11
- "Model Objects That Receive Coverage" on page 2-2

See Also
Variant Subsystem, Variant Model | Variant Source | Variant Sink

Collect Coverage for Multiple Simulations by Using Design Studies

This example shows how to collect coverage when running multiple simulations. To learn more about running multiple simulations, see "Run Multiple Simulations".

The `slvnvdemo_powerwindow_parsim` model contains a power window controller and a low-order plant model. A Stateflow® chart implements the controller.

Simulink Coverage
Power Window Controller Hybrid System Model
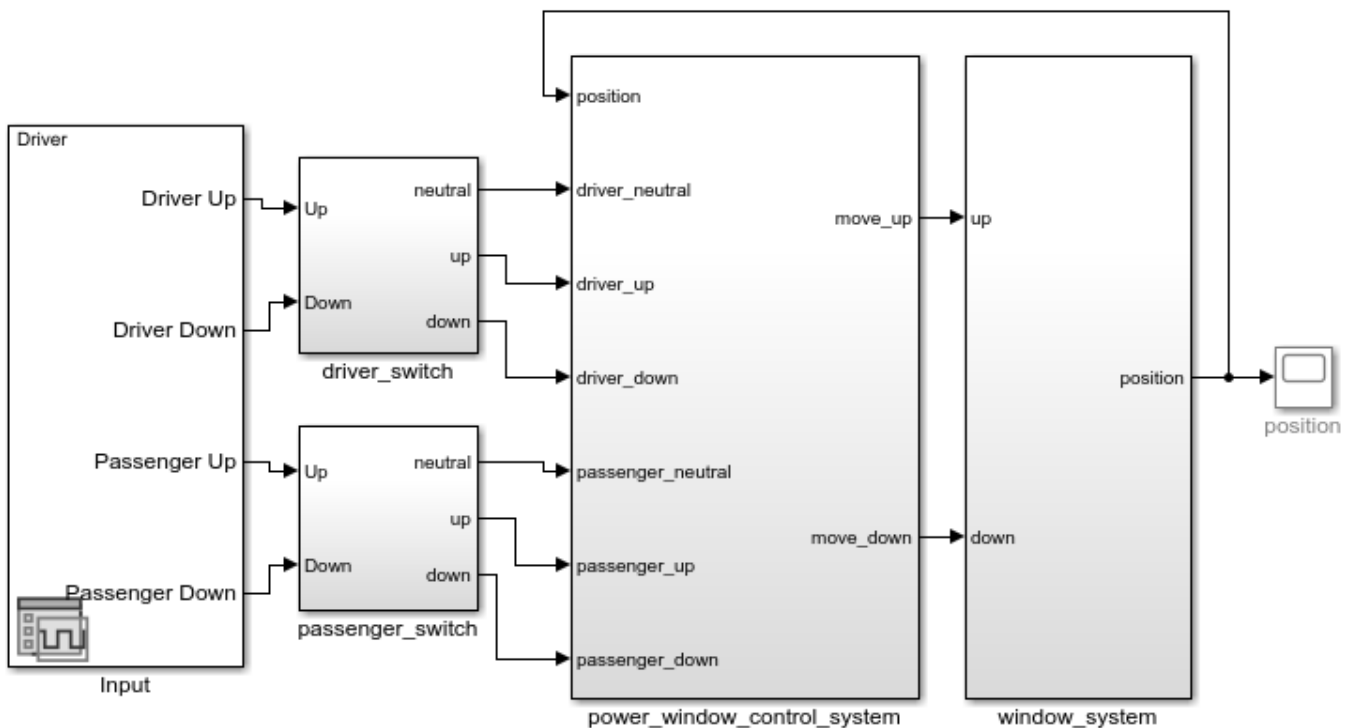
Copyright 1990-2018 The MathWorks, Inc.

Load the Design Study

Design studies allow you to run a series of simulations where each simulation changes one parameter at a time. You can collect coverage on the simulations to test different sets of model inputs without manually running each simulation. For more information about creating design studies, see "Configure and Run Simulations with Multiple Simulations Panel".

To open the **Multiple Simulations** pane, on the **Simulation** tab, under **Prepare**, click **Multiple Simulations**.

To load the design study file, click the **Load file containing design studies** icon at the top of the **Multiple Simulations** pane. In the **Select File to Open** window, select `slvnvdemo_powerwindow_multisim.mldatx`.

This file contains two design studies:

The **Driver Scenario** design study contains two simulations. In both simulations, the Signal Editor block `Input` parameter **Active Scenario** is set to `Driver`. In the first simulation, the **Active Signal** parameter is set to `Driver Up`, and in the second simulation the parameter is set to `Driver Down`. These settings simulate the driver-side power window receiving input to raise and lower the window.

The **Passenger Scenario** design study contains two simulations. In both simulations, the Signal Editor block `Input` parameter **Active Scenario** is set to `Passenger`. In the first simulation, the **Active Signal** parameter is set to `Passenger Up`, and in the second simulation the parameter is set to `Passenger Down`. These settings simulate the passenger-side power window receiving input to raise and lower the window.

Enable Coverage Analysis

Next, enable coverage for the model. On the **Modeling** tab, click **Model Settings**. In the left pane, click **Coverage** and then select **Enable coverage analysis**. Click **OK**.

Run Simulations and Review Coverage Results

In the **Multiple Simulations** pane, select **Driver Scenario**. On the **Apps** tab, select **Coverage Analyzer**. In the **Coverage** tab, click **Analyze Coverage > Run All (Coverage)**

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 12).
```

After the simulations complete, the **Coverage Details** pane opens and displays the aggregated coverage results for the two simulations in the Driver Scenario.

The coverage results for all of the simulations are populated into the Coverage Results Explorer. To open the Coverage Results Explorer, on the **Coverage** tab, click **Results Explorer**.
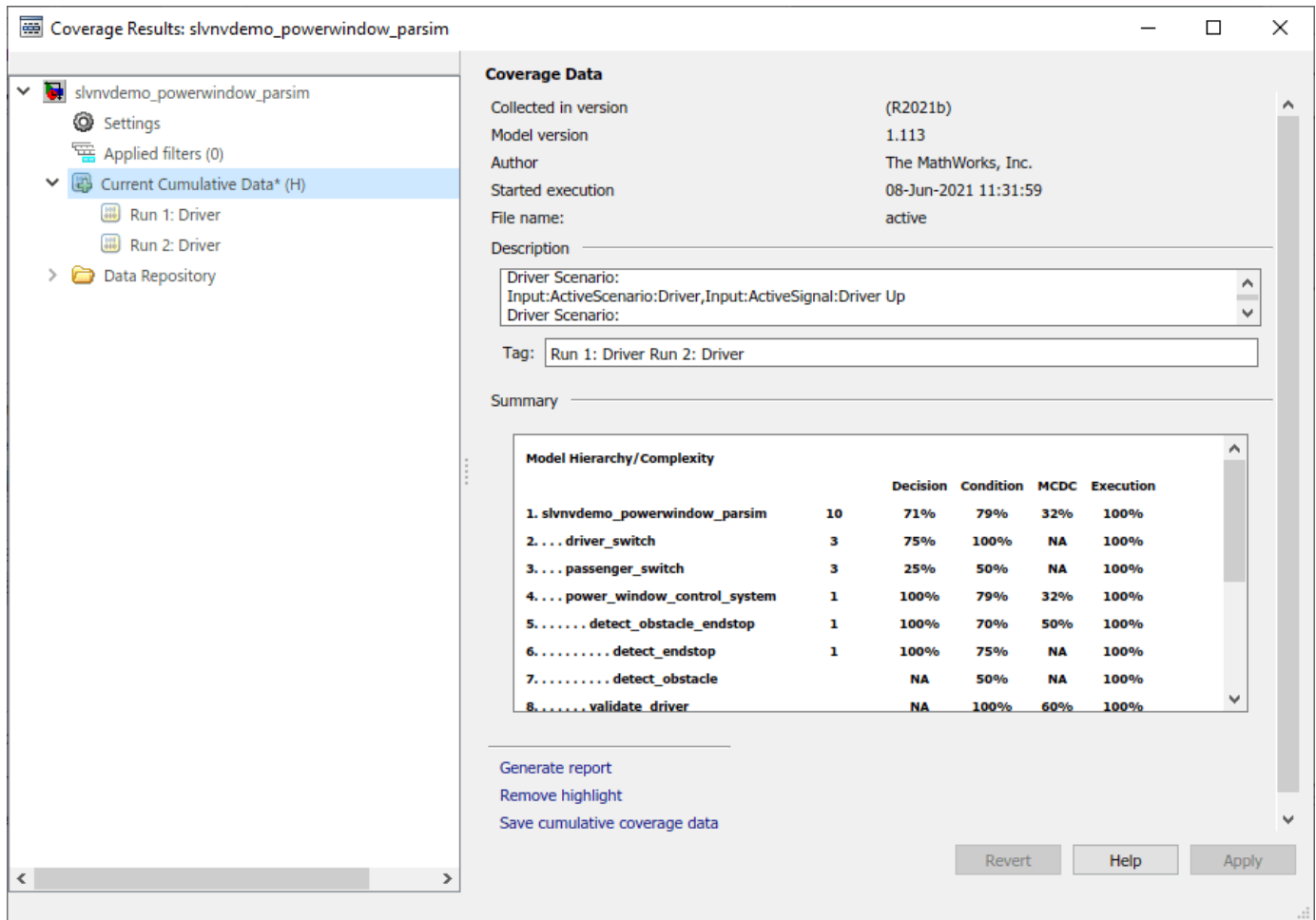
In the left pane of the Coverage Results Explorer, you can see the two simulations for the Driver Scenario design study and the current cumulative results. In the left pane, click **Current Cumulative Data**. In the **Coverage Data** pane, the description lists the design study name and a list of the block and parameter names specified in the design study. The **Tag** field lists the scenario specified for each run.

In the Simulink® window, in the **Multiple Simulations** pane, select **Passenger Scenario**. This action automatically clears the **Driver Scenario** design study because you can only select one design study at a time. On the **Coverage** tab, click **Analyze Coverage > Run All (Coverage)**.

The **Coverage Details** pane updates with aggregated data fromd the two additional simulations in the Passenger Scenario design study. Scroll to the **Aggregated Tests** section to see four simulations: two from the Driver Scenario design study, and two from the Passenger Scenario design study.

In the Coverage Results Explorer, click **Current Cumulative Data** to see that the additional simulations improved the coverage results.

Additionally, you can click each run to see that the aggregated coverage data displays more complete coverage than either of the individual runs. The different coverage results indicate that the different simulations satisfied different coverage outcomes.

To generate a standalone report for these results, in the Simulink® window, on the **Coverage** tab, click **Generate Report**. Alternatively, in the Coverage Results Explorer, in the left pane, click **Current Cumulative Data** and then click **Generate report**.

See Also
sim | parsim

Related Examples
- "Run Multiple Simulations"

Results Review

Types of Coverage Reports

If you simulate your model with coverage enabled using the **Run** button, or you generate a report from the Results Explorer, Simulink Coverage creates one or more model coverage reports after a simulation.

Report Type	Description	HTML Report File Name
"Top-Level Model Coverage Report" on page 6-10	Provides coverage information for all model elements, including the model itself.	*model_name*_cov.html
"Model Summary Report" on page 6-2	Provides links to coverage results for referenced models and external MATLAB files in the model hierarchy. Created when the top-level model includes Model blocks or calls one or more external files.	*model_name* active_summary_cov.html
"Model Reference Coverage Report" on page 6-3	Created for each referenced model in the model hierarchy; has the same format as the model coverage report.	*reference_model_name* _cov.html
"External MATLAB File Coverage Report" on page 6-3	Provides detailed coverage information about any external MATLAB file that the model calls. There is one report for each external file called from the model.	*MATLAB_file_name* _cov.html
"Subsystem Coverage Report" on page 6-7	Model coverage report includes only coverage results for the subsystem, if you select one.	*model_name*_cov.html; *model_name* is the name of the top-level model
"Code Coverage Report" on page 6-9	Provides coverage information for C/C++ code in S-Function blocks, or for models in SIL mode.	*model_name_block_name* _instance_n_cov.html, or *model_name*_cov.html

Model Summary Report

If the top-level model contains Model blocks or calls external files, the software creates a model summary coverage report named *model_name*_active_summary_cov.html. The title of this report is *Coverage by Model*.

The summary report lists and provides links to coverage reports for Model block referenced models and external files called by MATLAB code in the model. For more information, see "External MATLAB File Coverage Report" on page 6-3.

The following graphic shows an example of a model summary report. It contains links to the model coverage report (mExternalMfile), a report for the Model block (mExternalMfileRef), and three external files called from the model (externalmfile,I externalmfile1, andexternalmfile2).

Coverage Report by Model

Top Model: mExternalMfile

	Complexity	Decision	Condition	MCDC
TOTAL COVERAGE		90%	75%	25%
1. . . . mExternalMfile	5	50%	--	--
2. . . . externalmfile1	5	88%	75%	0%
3. . . . mExternalMfileRef	3	100%	--	--
4. . . . externalmfile	5	100%	75%	50%
5. . . . externalmfile2	2	100%	--	--

The following models have signal range coverage:

mExternalMfile

mExternalMfileRef

Model Reference Coverage Report

If your top-level model references a model in a Model block, the software creates a separate report, named *reference_model_name*_cov.html, that includes coverage for the referenced model. This report has the same format as the "Top-Level Model Coverage Report" on page 6-10. Coverage results are recorded as if the referenced model was a standalone model; the report gives no indication that the model is referenced in a Model block.

External MATLAB File Coverage Report

If your top-level model calls any external MATLAB files, select **MATLAB files** on the **Coverage** pane in the Configuration Parameters dialog box. The software creates a report, named *MATLAB_file_name*_cov.html, for each distinct file called from the model. When there are several calls to a given file from the model, the software creates only one report for that file, but it accumulates coverage from all the calls to the file. The external MATLAB file coverage report does not include information about what parts of the model call the external file.

The first section of the external MATLAB file coverage report contains summary information about the external file, similar to the model coverage report.

Coverage Report for externalmfile1

Table of Contents

Analysis Information

Coverage Data Information

Collected in version	(R2020b)

MATLAB Function File Information

Last saved	09-Jun-2020 13:57:49

Simulation Optimization Options

Default parameter behavior	tunable
Block reduction	forced off
Conditional branch optimization	on

Coverage Options

Analyzed model	externalmfile1
Logic block short circuiting	off
MCDC mode	masking

Tests

Test#	Started execution	Ended execution
Test 1	09-Jun-2020 13:58:10	09-Jun-2020 13:58:11

Summary

Model Hierarchy/Complexity		Test 1		
		Decision	Condition	MCDC
1. externalmfile1	6	88%	25%	0%

The *Details* section reports coverage for the external file and the function in that file.

Details

1. MATLAB Function file "externalmfile1"

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	6
Condition	NA	25% (1/4) condition outcomes
Decision	NA	88% (7/8) decision outcomes
MCDC	NA	0% (0/2) conditions reversed the outcome

MATLAB Function "externalmfile1"

Justify or Exclude

Parent: externalmfile1

Metric	Coverage
Cyclomatic Complexity	5
Condition	25% (1/4) condition outcomes
Decision	88% (7/8) decision outcomes
MCDC	0% (0/2) conditions reversed the outcome

The *Details* section also lists the content of the file, highlighting the code lines that have decision points or function definitions.

```
 1  %#eml
 2  function y = externalmfile1(u)
 3
 4  %    Copyright 2008 The MathWorks, Inc.
 5
 6  if u>1 && u<5
 7      a = 2;
 8  else
 9      a = 3;
10  end
11
12  for i=1:5
13      a = a+2;
14  end
15
16  y = a+localtest(a);
17
18  [x,y] = pol2cart(u,u);
19  [y2,y3] = cart2pol(x,y);
20
21  function y = localtest(u)
22
23  y = 0;
24  flg = true;
25  while flg
26      u = u/2;
27      y = y+1;
28      flg = u>2;
29  end
30
```

Coverage results for each of the highlighted code lines follow in the report. The following graphic shows a portion of these coverage results from the preceding code example.

#2: function y = externalmfile1(u)

Decisions analyzed

function y = externalmfile1(u)	100%
executed	102/102

#6: if u>1 && u<5

Decisions analyzed

if u>1 && u<5	50%
false	102/102
true	0/102

Subsystem Coverage Report

In the **Coverage** pane of the Configuration Parameters dialog box, when you select **Enable coverage analysis**, you can click **Select Subsystem** to request coverage for only the selected subsystem in the model. The software creates a model coverage report for the top-level model, but includes coverage results only for the subsystem.

However, if the top-level model calls any external files and you select **MATLAB files** in the **Coverage** pane in the Configuration Parameters dialog box, the results include coverage for all external files called from:

- The subsystem for which you are recording coverage
- The top-level model that includes the subsystem

If the subsystem parameter **Read/Write Permissions** is set to NoReadOrWrite, the software does not record coverage for that subsystem.

For example, in the fuelsys model, you click **Select Subsystem**, and select coverage for the feedforward_fuel_rate subsystem.
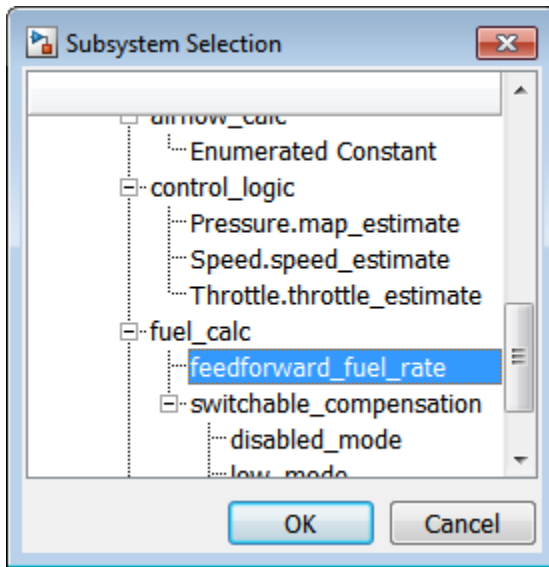
The report is similar to the model coverage report, except that it includes only results for the `feedforward_fuel_rate` subsystem and its contents.

Summary

Model Hierarchy/Complexity: **Test 1**

                                                                          D1

1. feedforward_fuel_rate   **3 33%** ▬▬

Details:

1. SubSystem block "feedforward_fuel_rate"

Parent: sldemo_fuelsys/fuel_rate_control/fuel_calc

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	3
Decision (D1)	NA	33% (1/3) decision outcomes

Code Coverage Report

For each S-Function block, the model coverage report links to a detailed code coverage report for the C/C++ code in the block. For more information on how to navigate the report, see "View Coverage Results for Custom C/C++ Code in S-Function Blocks" on page 5-61.

If you have Embedded Coder installed, you can also generate code coverage reports from models in SIL or PIL mode. For more information on how to generate code coverage reports for models in SIL or PIL mode, see "Code Coverage for Models in Software-in-the-Loop (SIL) Mode and Processor-in-the-Loop (PIL) Mode" on page 4-6.

Top-Level Model Coverage Report

If you simulate your model using the **Run** button, Simulink Coverage creates a model coverage report for the specified model named *model_name*_cov.html. The model coverage report is also opened automatically in the **Coverage Details** pane. The model coverage report contains several sections:

To access the sldemo_fuelsys model, execute the following commands in the MATLAB command window:

```
addpath([matlabroot,'\examples\simulink_automotive\main']);
open_system('sldemo_fuelsys');
```

Analysis Information

The analysis information section contains basic information about the model being analyzed:

- **Coverage Data Information**
- **Model Information**
- **Harness Information** (appears if you record coverage from a Simulink Test harness)
- **Simulation Optimization Options**
- **Coverage Options**

Coverage Report for sldemo_fuelsys

Table of Contents

Analysis Information

Coverage Data Information

Collected in version	(R2020b)

Model Information

Model version	5.0
Author	The MathWorks, Inc.
Last saved	Wed May 20 04:59:45 2020

Simulation Optimization Options

Default parameter behavior	inlined
Block reduction	forced off
Conditional branch optimization	on

Coverage Options

Analyzed model	sldemo_fuelsys
Logic block short circuiting	off

Aggregated Tests

The aggregated tests section appears if you:

- Record aggregated coverage results for at least two test cases through the Simulink Test Manager and produce a coverage report for the aggregated results, or
- Produce a coverage report for cumulative coverage results in the Results Explorer.

If you run test cases through the Simulink Test Manager, the aggregated tests section links to the associated test cases in the Simulink Test Manager.

If you aggregate test case results through the Results Explorer, the aggregated tests section links to the corresponding cvdata node in the Results Explorer.

For each run in the aggregated tests section, there is a link to the corresponding results in the Simulink Test Manager or the Results Explorer.

Aggregated Unit Tests

If you record coverage for one or more subsystem harnesses, the Aggregated Tests section lists each unit test run.

Each unit under test receives an ordinal number *n*, and each test for a unit under test receives an ordinal number *m* in the style U*n.m*.

Aggregated Tests

Run	Test Name	Date
Subsystem: "/SwitchUnit2"		
U1.1	Switch2 Unit Test - In Range	12-Jul-2019 13:54:28
U1.2	Switch2 Unit Test - Out of Range	12-Jul-2019 13:54:29
Model: "slcovSerialSwitchUnits"		
T1	Switches Integration Test - In Range	12-Jul-2019 13:54:27
T2	Switches Integration Test - Out of Range	12-Jul-2019 13:54:27

Coverage Summary

The coverage summary has two subsections:

- *Tests* — The simulation start and stop time of each test case and any setup commands that preceded the simulation. The heading for each test case includes any test case label specified using the `cvtest` command. This section only shows when the report does not contain an "Aggregated Tests" on page 6-11 section.
- *Summary* — Summaries of the subsystem results. To see detailed results for a specific subsystem, in the Summary subsection, click the subsystem name.

Tests

Test#	Started execution	Ended execution	Description
Test 1	07-Oct-2016 09:06:06	07-Oct-2016 09:08:25	This is a model of a fuel control system where Stateflow(R) is used to handle the fault management of the system. The system contains four separate sensors: a throttle sensor, a speed sensor, an oxygen sensor, and a pressure sensor. Each of these sensors is represented by a parallel state in Stateflow. Each parallel state contains two substates, a normal state and a failed state (the exception being the oxygen sensor, which also contains a warmup state). If any of the sensor readings is outside an acceptable range, then a fault is registered in Stateflow, and the substate of the corresponding subsystem transitions to the failed state. If a subsystem recovers, it can transition back to the normal state. The number of failures in the system at any given time is represented in the Fail parallel state. The last parallel state in the Stateflow chart is called Fueling_Mode. This state regulates the oxygen to fuel mixture ratio. If a failure is detected, then the oxygen to fuel ratio is increased. If multiple failures are detected, then the fuel system is disabled until there are no longer multiple failures in the system.

Summary

Model Hierarchy/Complexity		Test 1					
		Decision	Condition	MCDC	Execution	Relational Boundary	Saturation on integer overflow
1. sldemo_fuelsys	80	34%	34%	7%	90%	10%	50%
2.... Engine Gas Dynamics	13	71%	NA	NA	100%	50%	50%
3....... Mixing & Combustion	3	67%	NA	NA	100%	NA	50%
4.......... EGO Sensor	2	100%	NA	NA	NA	NA	NA
5.......... System Lag		NA	NA	NA	100%	NA	NA
6....... Throttle & Manifold	10	73%	NA	NA	100%	50%	50%
7.......... Intake Manifold	2	100%	NA	NA	100%	NA	50%
8............. MATLAB Function	2	100%	NA	NA	NA	NA	NA
9.......... Throttle	6	83%	NA	NA	100%	100%	50%

Details

The Details section reports the detailed model coverage results. Each section of the detailed report summarizes the results for the metrics that test each object in the model:

- "Filtered Objects" on page 6-14
- "Model Details" on page 6-14
- "Subsystem Details" on page 6-14
- "Block Details" on page 6-15
- "Chart Details" on page 6-16
- "Coverage Details for MATLAB Functions and Simulink Design Verifier Functions" on page 6-17
- "Requirement Testing Details" on page 6-20

You can also access a model element Details subsection as follows:

1 Right-click a Simulink element.

2 In the context menu, select **Coverage** > **Report**.

Filtered Objects

The Filtered Objects section lists all the objects in the model that were filtered from coverage recording, and the rationale you specified for filtering those objects. If the filter rule specifies that all blocks of a certain type be filtered, all those blocks are listed here.

In the following graphic, several blocks, subsystems, and transitions were filtered. Two library-linked blocks, protected division and protected division1, were filtered because their block library was filtered.

Blocks Eliminated from Coverage Analysis

Model Object	Rationale
slvnvdemo_covfilt/Saturation	It might not be executed because of Conditional input branch optimization
slvnvdemo_covfilt/protected division/Compare To Zero/Compare	It might not be executed because of Conditional input branch optimization
slvnvdemo_covfilt/protected division/Switch	It might not be executed because of Conditional input branch optimization
slvnvdemo_covfilt/protected division/Switch1	It might not be executed because of Conditional input branch optimization
slvnvdemo_covfilt/protected division1/Switch	It might not be executed because of Conditional input branch optimization

Model Details

The Details section contains a results summary for the model as a whole, followed by a list of elements. Click the model element name to see its coverage results.

The following graphic shows the Details section for the `sldemo_fuelsys` example model.

Details

1. Model "sldemo_fuelsys"

Child Systems: Engine Gas Dynamics, Throttle Command, To Controller, To Plant, fuel_rate_control

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	80
Condition	NA	34% (11/32) condition outcomes
Decision	NA	34% (41/122) decision outcomes
MCDC	NA	7% (1/14) conditions reversed the outcome
Lookup Table	NA	1% (13/1511) interpolation/extrapolation intervals
Execution	NA	90% (64/71) objective outcomes
Relational Boundary	NA	10% (5/50) objective outcomes
Saturation on integer overflow	NA	50% (10/20) objective outcomes

Subsystem Details

Each subsystem Details section contains a summary of the test coverage results for the subsystem and a list of the subsystems it contains. The overview is followed by sections for blocks, charts, and MATLAB functions, one for each object that contains a decision point in the subsystem.

The following graphic shows the coverage results for the Engine Gas Dynamics subsystem in the `sldemo_fuelsys` example model.

2. SubSystem block "Engine Gas Dynamics"

Justify or Exclude

Parent:	/sldemo_fuelsys
Child Systems:	Mixing & Combustion, Throttle & Manifold

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	13
Decision	NA	71% (10/14) decision outcomes
Execution	NA	100% (17/17) objective outcomes
Relational Boundary	NA	50% (3/6) objective outcomes
Saturation on integer overflow	NA	50% (10/20) objective outcomes

Block Details

The following graphic shows decision coverage results for the MinMax block in the Mixing & Combustion subsystem of the Engine Gas Dynamics subsystem in the `sldemo_fuelsys` example model.

MinMax block "MinMax"

Justify or Exclude

Parent:	sldemo_fuelsys/Engine Gas Dynamics/Mixing & Combustion
Uncovered Links:	⬅ ➡

Metric	Coverage
Cyclomatic Complexity	1
Decision	50% (1/2) decision outcomes
Execution	100% (1/1) objective outcomes

Decisions analyzed

Logic to determine output	50%
input 1 is the maximum	204508/204508
input 2 is the maximum	0/204508

The *Uncovered Links* element first appears in the Block Details section of the first block in the model hierarchy that does not achieve 100% coverage. The first *Uncovered Links* element has an arrow that links to the Block Details section in the report of the *next* block that does not achieve 100% coverage.

Subsequent blocks that do not achieve 100% coverage have links to the Block Details sections in the report of the previous and next blocks that do not achieve 100% coverage.

Saturate block "Limit to Positive"

Parent: sldemo_fuelsys/Engine Gas Dynamics/Throttle & Manifold

Uncovered Links: ← →

Chart Details

The following graphic shows the coverage results for the Stateflow chart `control_logic` in the `sldemo_fuelsys` example model.

17. SubSystem block "control_logic"

Justify or Exclude

Parent:	sldemo_fuelsys/fuel_rate_control
Child Systems:	fuel_rate_control/control_logic

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	1	56
Condition	NA	21% (5/24) condition outcomes
Decision	NA	25% (23/92) decision outcomes
MCDC	NA	0% (0/12) conditions reversed the outcome
Lookup Table	NA	0% (0/1082)interpolation/extrapolation intervals
Execution	NA	0% (0/4) objective outcomes
Relational Boundary	NA	0% (0/34) objective outcomes

18. Chart "fuel_rate_control/control_logic"

Justify or Exclude

Parent:	sldemo_fuelsys/fuel_rate_control/control_logic
Child Systems:	Fail, Fueling_Mode, O2, Pressure, Speed, Throttle

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	55
Condition	NA	21% (5/24) condition outcomes
Decision	NA	25% (23/92) decision outcomes
MCDC	NA	0% (0/12) conditions reversed the outcome
Lookup Table	NA	0% (0/1082)interpolation/extrapolation intervals
Execution	NA	0% (0/4) objective outcomes
Relational Boundary	NA	0% (0/34) objective outcomes

For more information about model coverage reports for Stateflow charts and their objects, see "Model Coverage for Stateflow Charts" on page 5-68.

Coverage Details for MATLAB Functions and Simulink Design Verifier Functions

By default, Simulink Coverage records coverage for all MATLAB functions in a model. MATLAB functions are in MATLAB Function blocks, Stateflow charts, or external MATLAB files.

Note For a detailed example of coverage reports for external MATLAB files, see "External MATLAB File Coverage Report" on page 6-3.

To record Simulink Design Verifier coverage for `sldv.*` functions called by MATLAB functions, and any Simulink Design Verifier blocks, select **Objectives and Constraints** on the **Coverage** pane of the Configuration Parameters dialog box.

The following example shows coverage details for a MATLAB function, `hFcnsInExternalEML`, that calls four Simulink Design Verifier functions. In this example, the code for `hFcnsInExternalEML` resides in an external file.

This example also shows Simulink Design Verifier coverage details for the following functions:

- `sldv.assume`
- `sldv.condition`
- `sldv.prove`
- `sldv.test`

In the coverage results, code that achieves 100% coverage is green. Code that achieves less than 100% coverage is red.

Embedded MATLAB function "hfcnsinexternaleml"

    Parent: hfcnsinexternaleml
    Uncovered Links:

Metric	Coverage
Cyclomatic Complexity	4
Decision (D1)	40% (2/5) decision outcomes
Test Objective	50% (1/2) objective outcomes
Proof Objective	0% (0/1) objective outcomes
Test Condition	100% (1/1) objective outcomes
Proof Assumption	0% (0/1) objective outcomes

```
 1   function y = hFcnsInExternalEML(u1, u2)
 2   % use all four functions.
 3   %#eml
 4   sldv.assume(u1 > u2);
 5   sldv.condition(u1 == 0);
 6   switch u1
 7       case 0
 8           y = u2;
 9       case 1
10           y = 3;
11       case 2
12           y = 0;
13       otherwise
14           y = 0;
15           sldv.prove(u2 < u1);
16   end
17   sldv.test(y > u1); sldv.test(y == 4);
18
```

Coverage for the `hFcnsInExternalEML` function and the `sldv.*` calls is:

- Line 1, the function declaration for `hFcnsInExternalEML`is green because the simulation executes that function at least once. `fcn` calls `hFcnsInExternalEML` 11 times during simulation.

```
#1: function y = hFcnsInExternalEML(u1, u2)

   Decisions analyzed:
   function y = hFcnsInExternalEML(u1, u2)              100%
      executed                                          11/11
```

Line 4, `sldv.assume(u1 > u2)`, achieves 0% coverage because `u1 > u2` never evaluates to true.

```
#4: sldv.assume(u1 > u2);


   Proof Assumption analyzed:
      sldv.assume(u1 > u2)                              0/11
```

- Line 5, `sldv.condition(u1 == 0)`, achieves 100% coverage because `u1 == 0` evaluates to true for at least one time step.

```
#5: sldv.condition(u1 == 0);


   Test Condition analyzed:
      sldv.condition(u1 == 0)                           11/11
```

- Line 6, `switch u1`, achieves 25% coverage because only one of the four outcomes in the `switch` statement (`case 0`) occurs during simulation.

```
#6: switch u1

   Decisions analyzed:
   switch u1                                            25%
      otherwise                                         0/11
      case 0                                            11/11
      case 1                                            0/11
      case 2                                            0/11
```

- Line 17, `sldv.test(y > u1); sldv.test (y == 4)` achieves 50% coverage. The first `sldv.test` call achieves 100% coverage, but the second `sldv.test` call achieves 0% coverage.

For more information about coverage for MATLAB functions, see "Model Coverage for MATLAB Functions" on page 5-45.

For more information about coverage for Simulink Design Verifier functions, see "Objectives and Constraints Coverage" on page 1-6.

Requirement Testing Details

If you run at least two test cases in Simulink Test that are linked to requirements in Simulink Requirements, the aggregated coverage report details the links between model elements, test cases, and linked requirements.

The **Requirement Testing Details** section includes:

- **Implemented Requirements** — Which requirements are linked to the model element.
- **Verified by Tests** — Which tests verify the requirement.
- **Associated Runs** — Which runs are associated with each verification test.

Switch block "Switch1"

Justify or Exclude

Requirement Testing Details

Implemented Requirements	Verified by Tests	Associated Runs
Enable Switch Detection	Enable button	U1.1

Parent: crs_controller/DriverSwRequest

Metric	Coverage
Cyclomatic Complexity	1
Decision	100% (2/2) decision outcomes
Execution	100% (1/1) objective outcomes

Decisions analyzed

logical trigger input	100%
false (output is from 3rd input port)	1607/1608 U1.1
true (output is from 1st input port)	1/1608 U1.1

For an example of how to trace coverage results to requirements in a coverage report, see "Trace Coverage Results to Requirements" on page 5-36.

Cyclomatic Complexity

You can specify that the model coverage report include cyclomatic complexity numbers in two locations in the report:

- The Summary section contains the cyclomatic complexity numbers for each object in the model hierarchy. For a subsystem or Stateflow chart, that number includes the cyclomatic complexity numbers for all their descendants.

Summary

Model Hierarchy/Complexity:

1. fuelsys	78
2. . . . engine gas dynamics	5
3. . . . . . . Mixing & Combustion	1
4. . . . . . . Throttle & Manifold	4
5. . . . . Throttle	2
6. . . . fuel rate controller	72
7. . . . . . . Airflow calculation	1
8. . . . . . . Fuel Calculation	11
9. . . . . Switchable Compensation	7
10. . . . . LOW Mode	2
11. . . . . RICH Mode	2
12. . . . . . . Sensor correction and Fault Redundancy	9
13. . . . . MAP Estimate	2
14. . . . . Speed Estimate	2
15. . . . . Throttle Estimate	2
16. . . . . . . control logic	51
17. . . . . SF: control logic	50
18. . . . . SF: Fail	12
19. . . . . SF: Multi	6
20. . . . . SF: Fueling_Mode	19
21. . . . . SF: Fuel_Disabled	4
22. . . . . SF: Running	10
23. . . . . SF: Low_Emissions	4
24. . . . . SF: O2	5
25. . . . . SF: Pressure	5
26. . . . . SF: Speed	4
27. . . . . SF: Throttle	5

- The Details sections for each object list the cyclomatic complexity numbers for all individual objects.

6. SubSystem block "Throttle & Manifold"

Justify or Exclude

Parent: sldemo_fuelsys/Engine Gas Dynamics

Child Systems: Intake Manifold, Throttle

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	0	10
Decision	NA	73% (8/11) decision outcomes
Execution	NA	100% (13/13) objective outcomes
Relational Boundary	NA	50% (3/6) objective outcomes
Saturation on integer overflow	NA	50% (8/16) objective outcomes

Decisions Analyzed

The Decisions analyzed table lists possible outcomes for a decision and the number of times that an outcome occurred in each test simulation. Outcomes that did not occur are in red highlighted table rows.

The following graphic shows the Decisions analyzed table for the Saturate block in the Throttle & Manifold subsystem of the Engine Gas Dynamics subsystem in the `sldemo_fuelsys` example model.

Saturate block "Limit to Positive"

Justify or Exclude

Parent:	sldemo_fuelsys/Engine Gas Dynamics/Throttle & Manifold
Uncovered Links:	⬅ ➡

Metric	Coverage
Cyclomatic Complexity	2
Decision	50% (2/4) decision outcomes
Execution	100% (1/1) objective outcomes
Relational Boundary	25% (1/4) objective outcomes

Decisions analyzed

input >= lower limit	50%
false	0/204508
true	204508/204508
input > upper limit	50%
false	204508/204508
true	0/204508

To display and highlight the block in question, click the block name at the top of the section containing the block's Decisions analyzed table.
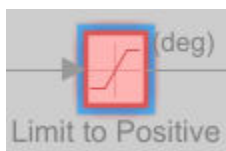
Conditions Analyzed

The Conditions analyzed table lists the number of occurrences of true and false conditions on each input port of the corresponding block.

Conditions analyzed

Description	True	False
input port 1	199521	480
input port 2	200001	0

MCDC Analysis

The MCDC analysis table lists the MCDC input condition cases represented by the corresponding block and the extent to which the reported test cases cover the condition cases.

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition:	True Out	False Out
expression for output		
input port 1	TT	FT
input port 2	TT	(TF)

Each row of the MCDC analysis table represents a condition case for a particular input to the block. A condition case for input n of a block is a combination of input values. Input n is called the *deciding input* of the condition case. Changing the value of input n alone changes the value of the block's output.

The MCDC analysis table shows a condition case expression to represent a condition case. A condition case expression is a character string where:

- The position of a character in the string corresponds to the input port number.
- The character at the position represents the value of the input. (T means `true`; F means `false`).
- A boldface character corresponds to the value of the deciding input.

For example, F**T**F represents a condition case for a three-input block where the second input is the deciding input.

The *Decision/Condition* column specifies the deciding input for an input condition case. The *True Out* column specifies the deciding input value that causes the block to output a `true` value for a condition case. The *True Out* entry uses a condition case expression, for example, F**F**, to express the values of all the inputs to the block, with the value of the deciding variable in bold.

Parentheses around the expression indicate that the specified combination of inputs did not occur during the first (or only) test case included in this report. In other words, the test case did not cover the corresponding condition case. The *False Out* column specifies the deciding input value that causes the block to output a false value and whether the value actually occurred during the first (or only) test case included in the report.

Some model elements achieve less MCDC coverage depending on the MCDC definition used during analysis. For more information on how the MCDC definition used during analysis affects the coverage results, see "Modified Condition and Decision Coverage (MCDC) Definitions in Simulink Coverage" on page 5-3.

If you select **Treat Simulink Logic blocks as short-circuited** in the **Coverage** pane in the Configuration Parameters dialog box, MCDC coverage analysis does not verify whether short-circuited inputs actually occur. The MCDC analysis table uses an x in a condition expression (for example, TFxxx) to indicate short-circuited inputs that were not analyzed by the tool.

If you disable this feature and Logic blocks are not short-circuited while collecting model coverage, you might not be able to achieve 100% coverage for that block.

Select the **Treat Simulink Logic blocks as short-circuited** option for where you want the MCDC coverage analysis to approximate the degree of coverage that your test cases achieve for the generated code (most high-level languages short-circuit logic expressions).

Cumulative Coverage

After you record successive coverage results, you can "Access, Manage, and Aggregate Coverage Results" on page 3-7 from within the Coverage Results Explorer. By default, the results of each simulation are saved and recorded cumulatively in the report.

If you select **Show cumulative progress report** in the "Results" on page 3-6 section of the configuration parameters, the results located in the right-most area in all tables of the cumulative coverage report reflect the running total value. The report is organized so that you can easily compare the additional coverage from the most recent run with the coverage from all prior runs in the session.

A cumulative coverage report contains information about:

- Current Run — The coverage results of the simulation just completed.
- Delta — Percentage of coverage added to the cumulative coverage achieved with the simulation just completed. If the previous simulation's cumulative coverage and the current coverage are nonzero, the delta may be 0 if the new coverage does not add to the cumulative coverage.
- Cumulative — The total coverage collected for the model up to, and including, the simulation just completed.

After running three test cases, the Summary report shows how much additional coverage the third test case achieved and the cumulative coverage achieved for the first two test cases.

Summary

Model Hierarchy/Complexity:		Current Run			Delta			Cumulative		
		Decision	Condition	MCDC	Decision	Condition	MCDC	Decision	Condition	MCDC
1. slvnvdemo_autopilot_test_harness	31	38%	41%	17%	8%	6%	0%	51%	41%	17%
2. .... Logic	25	34%	38%	17%	9%	8%	0%	47%	38%	17%
3. ...... SF: Logic	24	34%	38%	17%	9%	8%	0%	47%	38%	17%
4. .... SF: Altitude	11	64%	67%	33%	21%	17%	0%	93%	67%	33%
5. .... SF: Active	4	38%	NA	NA	13%	NA	NA	88%	NA	NA
6. .... SF: GS	13	11%	8%	0%	0%	0%	0%	11%	8%	0%
7. .... SF: Active	6	0%	NA	NA	0%	NA	NA	0%	NA	NA
8. .... SF: Coupled	3	0%	NA	NA	0%	NA	NA	0%	NA	NA
9. .... Verify Outputs	5	60%	50%	NA	0%	0%	NA	80%	50%	NA
10. ...... Subsystem1	1	0%	NA	NA	0%	NA	NA	100%	NA	NA
11. .... Capture time	1	0%	NA	NA	0%	NA	NA	100%	NA	NA
12. ...... Subsystem2	1	100%	NA	NA	0%	NA	NA	100%	NA	NA
13. .... Capture time	1	100%	NA	NA	0%	NA	NA	100%	NA	NA
14. ...... Subsystem3	1	0%	NA	NA	0%	NA	NA	0%	NA	NA
15. .... Capture time	1	0%	NA	NA	0%	NA	NA	0%	NA	NA
16. ...... Verification	2	100%	50%	NA	0%	0%	NA	100%	50%	NA

The *Decisions analyzed* table for cumulative coverage contains three columns of data about decision outcomes that represent the current run, the delta since the last run, and the cumulative data, respectively.

Decisions analyzed:

Transition trigger expression	100%	50%	100%
false	1097/1098	1097/1097	1097/1100
true	1/1098	0/1097	3/1100

The Conditions analyzed table uses column headers *#n T* and *#n F* to indicate results for individual test cases. The table uses *Tot T* and *Tot F* for the cumulative results. You can identify the true and false conditions on each input port of the corresponding block for each test case.

Conditions analyzed:

Description:	#1 T	#1 F	#2 T	#2 F	Tot T	Tot F
Condition 1, "alt_ctrl"	1	1097	0	1097	3	1097
Condition 2, "wow"	0	1	0	0	0	3
Condition 3, "in(GS.Active.Coupled)"	0	1	0	0	0	3

The MCDC analysis *#n True Out* and *#n False Out* columns show the condition cases for each test case. The *Total Out T* and *Total Out F* column show the cumulative results.

MC/DC analysis (combinations in parentheses did not occur)

Decision/Condition	#1 True Out	#1 False Out	#2 True Out	#2 False Out	Total Out T	Total Out F
Transition trigger expression						
Condition 1, "alt_ctrl"	TFF	Fxx	(TFF)	Fxx	TFF	Fxx
Condition 2, "wow"	TFF	(TTx)	(TFF)	(TTx)	TFF	(TTx)
Condition 3, "in(GS.Active.Coupled)"	TFF	(TFT)	(TFF)	(TFT)	TFF	(TFT)

Note You can calculate cumulative coverage for reusable subsystems and Stateflow constructs at the command line. For more information, see "Obtain Cumulative Coverage for Reusable Subsystems and Stateflow® Constructs" on page 5-34.

N-Dimensional Lookup Table

The following interactive chart summarizes the extent to which elements of a lookup table are accessed. In this example, two Sine Wave blocks generate *x* and *y* indices that access a 2-D Lookup Table block of 10-by-10 elements filled with random values.
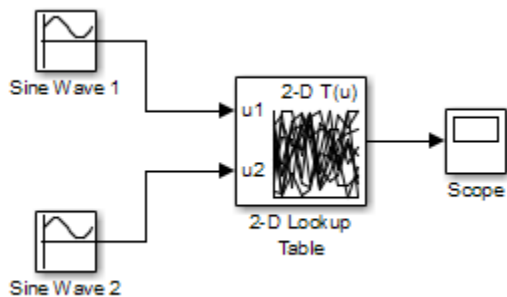
In this model, the lookup table indices are 1, 2,…, 10 in each direction. The Sine Wave 2 block is out of phase with the Sine Wave 1 block by *pi*/2 radians. This generates *x* and *y* numbers for the edge of a circle, which you see when you examine the resulting Lookup Table coverage.
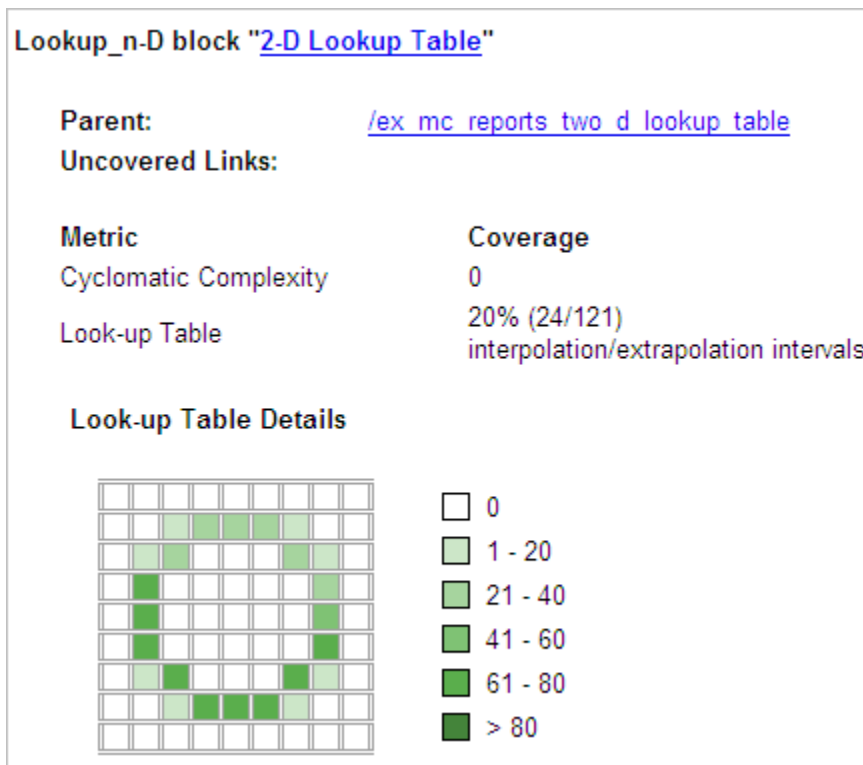
The report contains a two-dimensional table representing the elements of the lookup table. The element indices are represented by the cell border grid lines, which number 10 in each dimension. Areas where the lookup table interpolates between table values are represented by the cell areas. Areas of extrapolation left of element 1 and right of element 10 are represented by cells at the edge of the table, which have no outside border.

The number of values interpolated (or extrapolated) for each cell (*execution counts*) during testing is represented by a shade of green assigned to the cell. Each of six levels of green shading and the range of execution counts represented are displayed on one side of the table.

If you click an individual table cell, you see a dialog box that displays the index location of the cell and the exact number of execution counts generated for it during testing. The following example shows the contents of a color-shaded cell on the right edge of the circle.
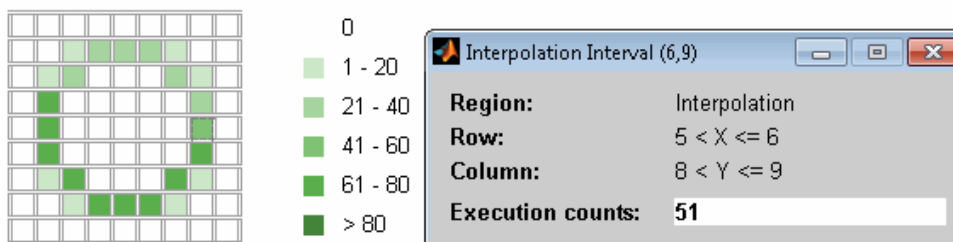
The selected cell is outlined in red. You can also click the extrapolation cells on the edge of the table.
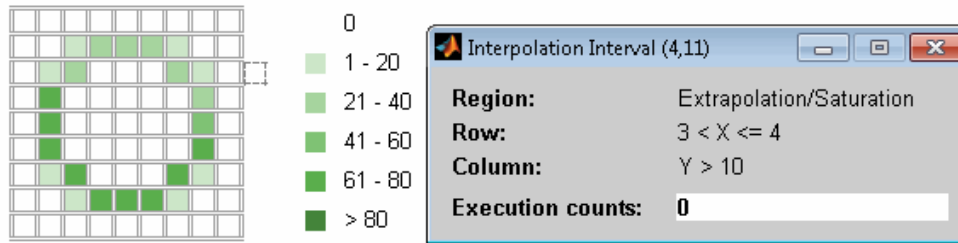
A bold grid line indicates that at least one block input equal to its exact index value occurred during the simulation. Click the border to display the exact number of hits for that index value.
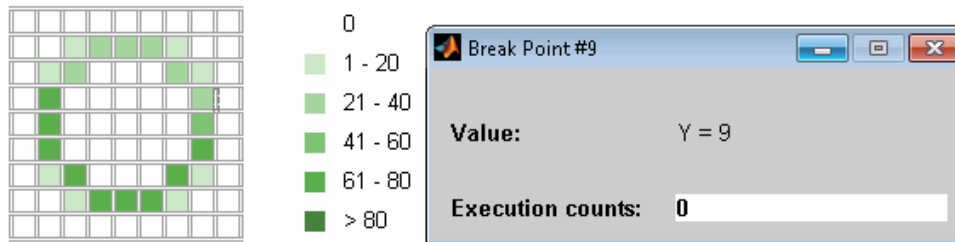
The following example model uses an n-D Lookup Table block of 10-by-10-by-5 elements filled with random values.
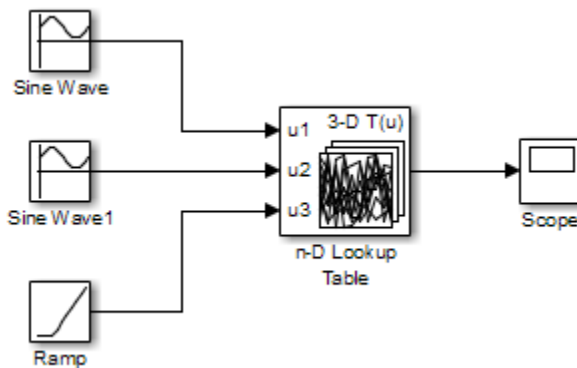
Both the x and y table axes have the indices 1, 2,..., 10. The z axis has the indices 10, 20,..., 50. Lookup table values are accessed with x and y indices that the two Sine Wave blocks generated, in the preceding example, and a z index that a Ramp block generates.

After simulation, you see the following lookup table report.

Lookup_n-D block "n-D Lookup Table"

Parent: /ex_mc_reports_three_d_lookup_table
Uncovered Links:

Metric	Coverage
Cyclomatic Complexity	0
Look-up Table	6% (42/726) interpolation/extrapolation intervals

Table map was not generated due to the table size.
Force Map Generation.

Instead of a two-dimensional table, the link `Force Map Generation` displays the following tables:

Look-up Table Details

$I_3 < 10$ I_3 $10 <= I_3 <= 20$ I_3 $20 < I_3 <= 30$ I_3

$30 < I_3 <= 40$ I_3 $40 < I_3 <= 50$ I_3 $I_3 > 50$
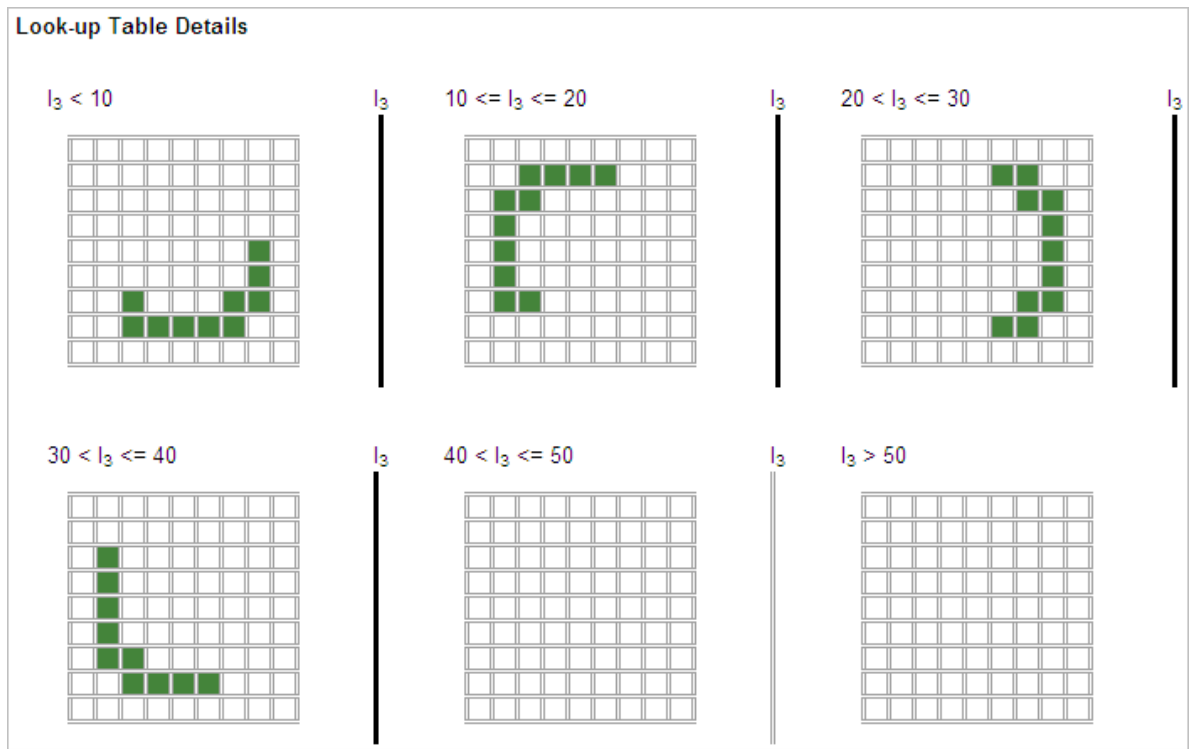
Lookup table coverage for a three-dimensional lookup table block is reported as a set of two-dimensional tables.

The vertical bars represent the exact z index values: 10, 20, 30, 40, 50. If a vertical bar is bold, this indicates that at least one block input was equal to the exact index value it represents during the simulation. Click a bar to get a coverage report for the exact index value that bar represents.

You can report lookup table coverage for lookup tables of any dimension. Coverage for four-dimensional tables is reported as sets of three-dimensional sets, like those in the preceding example. Five-dimensional tables are reported as sets of sets of three-dimensional sets, and so on.

Block Reduction

All model coverage reports indicate the status of the Simulink **Block reduction** parameter at the beginning of the report. In the following example, you set **Force block reduction off**.

Simulation Optimization Options

Default parameter behavior	tunable
Block reduction	forced off
Conditional branch optimization	on

In the next example, you enabled the Simulink **Block reduction** parameter, and you did not set **Force block reduction off**.

Simulation Optimization Options

Default parameter behavior	tunable
Block reduction	on
Conditional branch optimization	on

Consider the following model where the simulation does not execute the MinMax1 block because there is only one input — In3.
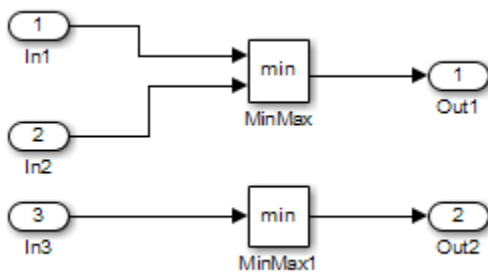
If you set **Force block reduction off**, the report contains no coverage data for this block because the minimum input to the MinMax1 block is always 1.

If you do not set **Force block reduction off**, the report contains no coverage data for reduced blocks.

Reduced Blocks

Blocks eliminated from coverage analysis by block reduction
model simulation setting:

. . . ex_minmax_coverage/MinMax1

Relational Boundary

On the "Coverage Pane" on page 3-2 of the Configuration Parameters dialog box, if you select the
Relational Boundary coverage metric, the software creates a Relational Boundary table in the
model coverage report for each model object that is supported for this coverage. The table applies to
the explicit or implicit relational operation involved in the model object. For more information, see:

- "Relational Boundary Coverage" on page 1-7.
- The **Relational Boundary** column in "Model Objects That Receive Coverage" on page 2-2.

The tables below show the relational boundary coverage report for the relation `input1 <= input2`.
The appearance of the tables depend on the operand data type.

- "Integers" on page 6-32
- "Fixed point" on page 6-33
- "Floating point" on page 6-33

Integers

If both operands are integers (or if one operand is an integer and the other a Boolean), the table
appears as follows.

Relational Boundary

input1 - input2	33%
-1	0/51
0	51/51
+1	0/51

For a relational operation such as *operand_1 <= operand_2*:

- The first row states the two operands in the form *operand_1 - operand_2*.
- The second row states the number of times during the simulation that *operand_1 - operand_2*
  is equal to -1.
- The third row states the number of times during the simulation that *operand_1* is equal to
  operand_2.

- The fourth row states the number of times during the simulation that *operand_1 - operand_2* is equal to 1.

Fixed point

If one of the operands has fixed-point type and the other operand is either a fixed point or an integer, the table appears as follows. LSB represents the value of the least significant bit. For more information, see "Precision" (Fixed-Point Designer). If the two operands have different precision, the smaller value of precision is used.

Relational Boundary

input1 - input2	33%
-LSB	51/51
0	0/51
+LSB	0/51

For a relational operation such as *operand_1 <= operand_2*:

- The first row states the two operands in the form *operand_1 - operand_2*.
- The second row states the number of times during the simulation that *operand_1 - operand_2* is equal to -LSB.
- The third row states the number of times during the simulation that *operand_1* is equal to *operand_2*.
- The fourth row states the number of times during the simulation that *operand_1 - operand_2* is equal to LSB.

Floating point

If one of the operands has floating-point type, the table appears as follows. tol represents a value computed using the input values and a tolerance that you specify. If you do not specify a tolerance, the default values are used. For more information, see "Relational Boundary Coverage" on page 1-7.

Relational Boundary

input1 - input2	50%
[-tol..0]	51/51
(0..tol]	0/51

For a relational operation such as *operand_1 <= operand_2*:

- The first row states the two operands in the form *operand_1 - operand_2*.

6-33

- The second row states the number of times during the simulation that *operand_1 - operand_2* has values in the range [-tol..0].

- The third row states the number of times during the simulation that *operand_1 - operand_2* has values in the range (0..tol] during the simulation.

The appearance of this table changes according to the relational operator in the block. Depending on the relational operator, the value of *operand_1 - operand_2* equal to 0 is either:

- Excluded from relational boundary coverage.

- Included in the region above the relational boundary.

- Included in the region below the relational boundary.

Relational Operator	Report Format	Explanation
==	[-tol..0)	0 is excluded.
	(0..tol]	
!=	[-tol..0)	0 is excluded.
	(0..tol]	
<=	[-tol..0]	0 is included in the region below the relational boundary.
	(0..tol]	
<	[-tol..0)	0 is included in the region above the relational boundary.
	[0..tol]	
>=	[-tol..0)	0 is included in the region above the relational boundary.
	[0..tol]	
>	[-tol..0]	0 is included in the region below the relational boundary.
	(0..tol]	

0 is included below the relational boundary for <= but above the relational boundary for <. This rule is consistent with decision coverage. For instance:

- For the relation input1 <= input2, the decision is true if input1 is less than or equal to input2. < and = are grouped together. Therefore, 0 lies in the region below the relational boundary.

- For the relation input1 < input2, the decision is true only if input1 is less than input2. > and = are grouped together. Therefore, 0 lies in the region above the relational boundary.

Saturate on Integer Overflow Analysis

On the "Coverage Pane" on page 3-2 of the Configuration Parameters dialog box, if you select the **Saturate on integer overflow** coverage metric, the software creates a Saturation on Overflow analyzed table in the model coverage report. The software creates the table for each block with the **Saturate on integer overflow** parameter selected.

The Saturation on Overflow analyzed table lists the number of times a block saturates on integer overflow, indicating a true decision. If the block does not saturate on integer overflow, the table indicates a false decision. Outcomes that do not occur are in red highlighted table rows.

The following graphic shows the Saturation on Overflow analyzed table for the MinMax block in the Mixing & Combustion subsystem of the Engine Gas Dynamics subsystem in the `sldemo_fuelsys` example model.

MinMax block "MinMax"

Parent: sldemo_fuelsys/Engine Gas Dynamics/Mixing & Combustion
Uncovered Links: ➡

Metric	Coverage
Cyclomatic Complexity	0
Saturation on Overflow	50% (1/2) objective outcomes

Saturation on Overflow analyzed:

Saturate on integer overflow	50%
false	204508/204508
true	0/204508

To display and highlight the block in question, click the block name at the top of the section containing the block's Saturation on Overflow analyzed table.

Signal Range Analysis

If you select the **Signal Range** coverage metric, the software creates a Signal Range Analysis section at the bottom of the model coverage report. This section lists the maximum and minimum signal values for each output signal in the model measured during simulation.

Access the Signal Range Analysis report quickly with the *Signal Ranges* link in the nonscrolling region at the top of the model coverage report, as shown below in the `sldemo_fuelsys` example model report.

Signal Ranges

Hierarchy	Min	Max
sldemo_fuelsys		
...Engine_Speed_Selector	300	300
...MAP_Selector	0.405559	0.889674
...O2_Voltage_Selector	0.456832	1
...Throttle_Angle_Selector	10	20
...Constant2	0	0
...Constant3	12	12
...Constant4	0	0
...Constant5	0	0
...EGO Fault Switch	1	1
...Engine Speed	300	300
...Engine Speed Fault Switch	1	1
...MAP Fault Switch	1	1
...Throttle Angle Fault Switch	1	1
...Engine Gas Dynamics		
......Mixing & Combustion		

Each block is reported in hierarchical fashion; child blocks appear directly under parent blocks. Each block name in the *Signal Ranges* report is a link. For example, select the `EGO sensor` link to display this block highlighted in its native diagram.
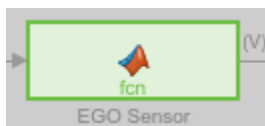
Signal Size Coverage for Variable-Dimension Signals

If you select **Signal Size**, the software creates a Variable Signal Widths section after the Signal Ranges data in the model coverage report. This section lists the maximum and minimum signal sizes for all output ports in the model that have variable-size signals. It also lists the memory that Simulink allocated for that signal, as measured during simulation. This list does *not* include signals whose size does not vary during simulation.

The following example shows the Variable Signal Widths section in a coverage report. In this example, the Abs block signal size varied from 2 to 5, with an allocation of 5.

Variable Signal Widths:

Hierarchy	Min	Max	Allocated
. . . Abs	2	5	5
. . . Abs1	4	4	5
. . . MinMax1	2	5	5
. . . Switch	2	5	5
. . . Switch1	2	5	5
. . . Selector	4	4	5
. . . c2ri			
. . . . . . out1	4	4	5
. . . . . . out2	4	4	5
. . . Subsystem			
. . . . . . LogicalOperator	1	2	2
. . . . . . Switch1	1	2	2
. . . . . . Switch2	1	2	2

Each block is reported in hierarchical fashion; child blocks appear directly under parent blocks. Each block name in the Variable Signal Widths list is a link. Clicking on the link highlights the corresponding block in the Simulink Editor. After the analysis, the variable-size signals have a wider line design.

Simulink Design Verifier Coverage

If you select **Objectives and Constraints**, the analysis collects coverage data for all Simulink Design Verifier blocks in your model.

For an example of how this works, open the `sldvdemo_debounce_testobjblks` model.

This model contains two Test Objective blocks:

- The True block defines a property that the signal have a value of 2.
- The Edge block, inside the Masked Objective subsystem, describes the property where the output of the AND block in the Masked Objective subsystem changes from 2 to 1.

The Simulink Design Verifier software analyzes this model and produces a harness model that contains test cases that achieve certain test objectives. To see if the original model achieves those objectives, simulate the harness model and collect model coverage data. The model coverage tool analyzes any decision points or values within an interval that you specify in the Test Objective block.

In this example, the coverage report shows that you achieved 100% coverage of the True block because the signal value was 2 at least once. The signal value was 2 in 6 out of 14 time steps.

Design Verifier Test Objective block "True"

Justify or Exclude
Parent: /sldvdemo_debounce_testobjblks
Uncovered Links: ◀ ▶

Metric   **Coverage**
Test Objective   0% (0/1) objective outcomes

Test Objective analyzed

2	0/1001

The input signal to the Edge block achieved a value of `True` once out of 14 time steps.

Design Verifier Test Objective block "Edge"

Justify or Exclude
Parent: sldvdemo_debounce_testobjblks/Masked Objective
Uncovered Links: ◀

Metric   **Coverage**
Test Objective   0% (0/1) objective outcomes

Test Objective analyzed

T	0/1001

Export Model Coverage Web View

You can export a Model Coverage Web View for your model. A Web View is an interactive rendition of a model that you can view in a Web browser. A Model Coverage Web View includes model coverage highlighting and analysis information from the Coverage Display Window, as described in "View Coverage Results in Simulink Canvas" on page 5-22.

Use the Results Explorer to generate a Model Coverage Web View. After you record coverage, you access the Results Explorer from the **Coverage** app. In the Results Explorer, open the **Settings**, select **Generate Web View Report**, and click **Apply**.
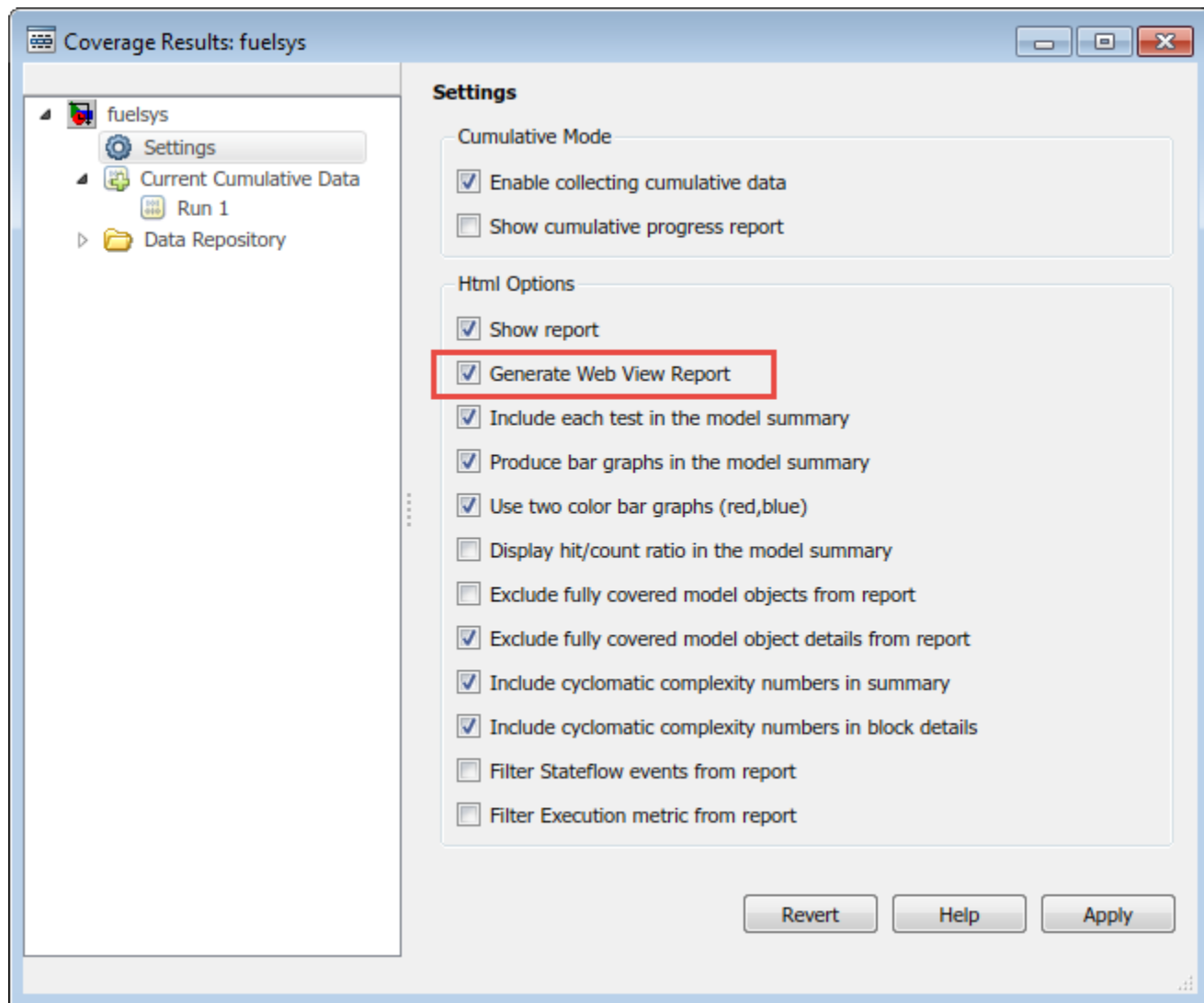
Next, select the Current Cumulative Data click **Generate report**.

When you generate a coverage report for your model with these settings enabled, the software generates a Model Coverage Web View that you can open in a browser. To see model coverage information for a block in a Model Coverage Web View, click that block. The model coverage data appears in the **Informer** pane, below the model.

For more information, see "Web Views" (Simulink Report Generator).

Filtering in Simulink Coverage

Coverage Filtering

In this section...
"When to Use Coverage Filtering" on page 7-2
"What Is Coverage Filtering?" on page 7-2

When to Use Coverage Filtering

Use coverage filtering to facilitate a bottom-up approach to recording model coverage. If you have a large model, there can be design elements that intentionally do not record 100% coverage. You can also have several design elements that you require to record 100% coverage but that do not achieve 100% coverage. You can temporarily or permanently eliminate these elements from coverage recording to focus on a subset of objects for testing and modification.

You can then iterate more efficiently—focus on a small issue, fix it, and then move on to resolve the next small issue. Before recording coverage for the entire model, you can resolve missing coverage issues within individual parts of the model.

What Is Coverage Filtering?

Coverage filtering enables you to exclude certain model objects from model coverage reporting after you simulate your Simulink model. You specify which objects you want to filter from coverage recording. There are two modes of filtering, Excluded and Justified.

Excluded objects do not contribute to coverage reports. After you specify the objects to exclude when you simulate your model, the coverage report does not record coverage for those objects.

Justified objects do contribute to coverage reports. After you specify the objects to justify when you simulate your model, the coverage report considers these blocks as achieving 100% coverage, and they appear light blue in the "Coverage Summary" on page 6-12.

Summary

Model Hierarchy/Complexity		Test 1			
		D1	C1	MCDC	Execution
1. slvnvdemo_covfilt	29	52%	40%	50%	13%
2. .... Mode Logic	13	86%	75%	50%	NA
3. ...... SF: Mode Logic	12	86%	75%	50%	NA
4. .... SF: Clipped	6	100%	NA	NA	NA
5. .... SF: Full	2	25%	NA	NA	NA

In the "Details" on page 6-13 section of the coverage report, justified objects show their coverage outcomes as ((covered outcomes + justified outcomes)/possible decisions).

4. State "Clipped"

Justified (Remove this rule)
Justification rationale: Justification rationale
Parent: slvnvdemo_covfilt/Mode Logic
Uncovered Links: ◀ ▶

Metric	Coverage (this object)	Coverage (inc. descendants)
Cyclomatic Complexity	2	6
Decision (D1)	100% ((2+2)/4) decision outcomes	100% ((5+7)/12) decision outcomes

To filter objects, see "Create, Edit, and View Coverage Filter Rules" on page 7-6 and "Creating and Using Coverage Filters" on page 7-11.

Coverage Filter Rules and Files

In this section...
"What Is a Coverage Filter Rule?" on page 7-4
"What Is a Coverage Filter File?" on page 7-4

What Is a Coverage Filter Rule?

A coverage filter rule specifies a model object, a set of objects, or lines of code that you want to exclude from coverage recording or that you want to justify for coverage.

Each coverage filter rule includes the following fields:

- **Name**—Name or path of the object to filter from coverage
- **Type**—Whether a specific object is filtered or all objects of a given type are filtered
- **Mode**—Whether the object to be filtered is `Excluded` or `Justified`

  Coverage reports do not include `Excluded` blocks. The coverage reports assume that `Justified` blocks receive full coverage, but show that they are distinct from other covered blocks in the coverage report.

- **Rationale**—An optional description that describes why this object is filtered from coverage

What Is a Coverage Filter File?

A coverage filter file is a set of coverage filter rules. Each rule specifies one or more objects or lines of code to exclude from coverage recording.

After you create and apply coverage filter rules, the specified objects or lines of code are excluded from coverage when you generate a report. You can reuse a coverage filter file for several Simulink models.

When you make changes to the coverage filter rules after you record coverage, you can update the coverage report without needing to resimulate your model. After you make changes, click **Apply**, then click **Generate Report** in the **Applied filters** section of the coverage Results Explorer to update the report.

If you use the default file name for the active model, and the coverage filter file exists on the MATLAB path, you see the coverage filter rules each time that you open the model. To save your current coverage filter rules to a file, click **Save filter**. To load an existing coverage filter file, click **Load filter**.

For more information on filtering objects, see "Create, Edit, and View Coverage Filter Rules" on page 7-6 and "Creating and Using Coverage Filters" on page 7-11.

Model Objects to Filter from Coverage

In your model, the objects that you can filter from coverage recording are:

- Simulink blocks that receive coverage, including MATLAB Function blocks
- Subsystems and their contents. When you exclude a subsystem from coverage recording, none of the objects inside the subsystem record coverage.
- Individual library-linked blocks or charts
- All reference blocks linked to a library
- Stateflow charts, subcharts, states, transitions, and events

For a complete list of model objects that receive coverage, see "Model Objects That Receive Coverage" on page 2-2.

Create, Edit, and View Coverage Filter Rules

In this section...
"Create and Edit Coverage Filter Rules" on page 7-6
"Save Coverage Filter to File" on page 7-8
"Create New Coverage Filter File" on page 7-8
"Load Coverage Filter File" on page 7-8
"Remove Applied Coverage Filter" on page 7-9
"Manage Applied filters by Using the Simulink Test Manager" on page 7-9
"Update the Report with the Current Filter Settings" on page 7-9
"View Coverage Filter Rules in Your Model" on page 7-9

Create and Edit Coverage Filter Rules

- "Create a Coverage Filter Rule" on page 7-6
- "Select the Filtering Mode" on page 7-7
- "Add Rationale to a Coverage Filter Rule" on page 7-7
- "Justify Dead Logic from Simulink Design Verifier Dead Logic Analysis" on page 7-7
- "Justify Dead Logic from Polyspace Code Prover Results" on page 7-8

Create a Coverage Filter Rule

To create a coverage filter rule:

1 Enable model coverage.

2 To record coverage results, simulate the model.

3 Create a new filter rule in one of these ways:

- In the model window, right-click a model object and select **Coverage > Exclude**.
- In the Details section of the Coverage Report, click **Justify or Exclude** for a model object.
- Create a new coverage filter file directly from the coverage Results Explorer:

    a Click **Applied filters**.
    b Click **New filter**.
    c Enter a **Name** and **Description** for the filter.
    d Click **Save as**.
    e Specify a file name and folder for the filter file and click **Save**.

    Alternatively, you can right-click the **Applied filters** label and select **New filter**

Depending on which option you select, the **Type** field in the "View Applied Filters in the Coverage Results Explorer" on page 7-10 is set for the coverage filter rule you selected. You cannot override the value in the **Type** field.

Select the Filtering Mode

When you create a filtering rule, the default filtering mode is `Excluded`. Excluded objects do not appear in the coverage reports. You can also set the filtering mode to `Justified`. Justified blocks appear as achieving 100% coverage.

For more information, see "Coverage Filtering" on page 7-2.

Add Rationale to a Coverage Filter Rule

Optionally, you can add text that describes why you exclude that object or objects from coverage recording. This information can be useful to others who review the coverage for your model. When you add a coverage filter rule, the **Applied filters** section of the coverage Results Explorer opens. To add the rationale:

1 Double-click the Rationale field for the rule.

2 Delete the existing text.

3 Add the rationale for excluding this object.

Note The **Rationale** field and **Mode** field are the only coverage filter rule fields that you can edit in the **Applied filters** section of the coverage Results Explorer.

After you add a new coverage filter rule or edit an existing coverage filter rule, click **Apply** to enable the **Generate report** and **Highlight model with coverage results** links.

Justify Dead Logic from Simulink Design Verifier Dead Logic Analysis

You can create justification rules in the coverage Results Explorer using the dead logic detected during a Simulink Design Verifier Dead Logic Analysis.

1 Open the Results Explorer from the **Coverage** app.
2 Click **Applied filters** to access the coverage filters.
3 Click **Make justification filter rules for dead logic (using Simulink Design Verifier)**.

Simulink Design Verifier runs the Dead Logic Analysis and populates the list of filters.
4 Click **Generate report**.

The justified rules from the previous step are shown in the **Objects Filtered from Coverage Analysis** section at the beginning of the report. To navigate to the rules' corresponding items in the **Details** section of the report, use the hyperlinks in the rule descriptions. Clicking the hyperlinks in the **Rationale** column navigates to the coverage Results Explorer.

Objects Filtered from Coverage Analysis

# Model Object	Rationale
j1. input port 1 T in Logic block "Or"	dead logic
j2. input > lower limit F in Saturate block "Saturation"	dead logic

You can add justification rules for elements that do not receive coverage to the filter by clicking in the **Details** section of the report.

Justify Dead Logic from Polyspace Code Prover Results

You can create justification rules for code coverage in the coverage Results Explorer using Polyspace Code Prover results.

1   Open the Results Explorer from the **Coverage** app.
2   Click **Applied filters** to access the coverage filters.
3   Click **Make justification filter rules for dead logic (using Polyspace Code Prover results)**.

    Opens a file selection window so you can provide a Polyspace Code Prover results file.
4   Click **Generate report**.

Save Coverage Filter to File

After you define the coverage filter rules, save the rules to a file so that you can reuse them with this model or other models. By default, coverage filter files are named *<model_name>*_covfilter.cvf.

1   In the **Apps** tab, click **Coverage Analyzer**. In the **Coverage** tab, open the coverage Results Explorer.
2   Click **Applied filters**, then select your filter.
3   Enter a **Name** and **Description** for the filter, if none already exist.
4   Click **Apply**. A save dialog box opens.
5   Specify a file name and folder for the filter file and click **Save**.

If you make multiple changes to the coverage filter rules, apply the changes to the coverage filter file each time.

Create New Coverage Filter File

You can create a new coverage filter file directly from the coverage Results Explorer.

1   Click **Applied filters**.
2   Click **New filter**. Alternatively, you can right-click **Applied filters** and select **New filter**.
3   Enter a **Name** and **Description** for the filter.
4   Click **Apply**. A save dialog box opens.
5   Specify a file name and folder for the filter file and click **Save**.

Load Coverage Filter File

After you save a coverage filter file, you can load the coverage filter file for use in other models. In the coverage Results Explorer:

1   Click **Applied filters**.
2   Click **Load filter**. Alternatively, you can right-click **Applied filters** and select **Load filter**.
3   Navigate to the filter file and click **Open**.

You can load multiple coverage filter files for any model. Loaded filter files show in the **Applied filters** section of the coverage Results Explorer.

Two or more models can have the same coverage filter file attached. If a model has an attached filter file that contains coverage filter rules for specific objects in a different model, those rules are ignored during coverage recording.

Remove Applied Coverage Filter

To remove an applied coverage filter, from the coverage Results Explorer:

1   Expand the **Applied filters**.
2   Right-click the coverage filter you want to remove and select **Remove**.

Manage Applied filters by Using the Simulink Test Manager

You can also add and remove coverage filter files from the Simulink Test Manager. For more information, see "Coverage Filtering Using the Test Manager" (Simulink Test).

Update the Report with the Current Filter Settings

If you change the filtering settings or add filters after you simulate the model, you can update the coverage report and model highlighting without resimulating the model. After you have simulated the model, in the Current Cumulative Data section of the **Applied filters** section of the coverage Results Explorer:

1   **Apply** or **Revert** any changes you have made.
2   Click **Generate Report**.

View Coverage Filter Rules in Your Model

Whenever you define a coverage filter rule or remove an existing coverage filter rule, the **Applied filters** section of the coverage Results Explorer opens. This pane lists the coverage filter rules for your applied filters. For more information, see "View Applied Filters in the Coverage Results Explorer" on page 7-10.

The list of currently applied filters for a model is available in the **Applied filters** section of the coverage Results Explorer. Alternatively, you can right-click anywhere in the model window and select **Coverage > Open Filter Viewer**.

If you are inside a subsystem, you can view any coverage filter rule attached to the subsystem. To open the **Applied filters** section of the coverage Results Explorer, right-click any object inside the subsystem and select **Coverage > Show filter parent**.

View Applied Filters in the Coverage Results Explorer

In the Applied filters section of the coverage Results Explorer, you can:

- Review and manage the coverage filter rules for your Simulink model.
- Create, load, or save coverage filter files for your model.
- Navigate to the model to create additional coverage filter rules.

To access the Applied filters section of the coverage Results Explorer:

- "Create a Coverage Filter Rule" on page 7-6
- From the Coverage Analyzer app, open the coverage Results Explorer. Currently applied filters are listed under the **Applied filters** label, or you can create a new coverage filter file.

To	Action	
Navigate to a model object associated with a rule.	**1**	Select the rule.
	2	Click **View in model**.
Delete a rule.	**1**	Select the rule.
	2	Click **Remove rule**.
Save the current rules to a file.	**1**	Click **Save filter**.
	2	Specify a file name and folder for the filter file and click **Save**.
Load an existing coverage filter file.	**1**	Click **Load filter**.
	2	Navigate to the filter file and click **Open**.
Create a new coverage filter file.	**1**	Right-click **Applied filters** and select **New filter**.
Update the current coverage report with the current filtering rules.	**1**	**Apply** or **Revert** any changes you have made.
	2	Click **Generate Report**.

Creating and Using Coverage Filters

This example shows how to use Simulink® Coverage™ model coverage filters to exclude model items from coverage results and justify missing coverage in reports.

Coverage Filters

During the verification process, a model can contain several constructs that prevent full model coverage, such as a subsystem that contains a driver for a controller that is not tested and is not relevant to the validation process. You can exclude this subsystem from the coverage results.

Alternatively, you may have testing criteria that requires exercising certain aspects of a block, such as hitting particular decision points. If it is not feasible to satisfy all objective outcomes for this block, and you did not intend for your tests to exercise these unsatisfied outcomes, then you could justify this missing coverage.

Filtering these constructs in coverage results by excluding or justifying them allows you to focus on other aspects of missing coverage that can and should be tested.

Coverage filters are stored in `.cvf` files. Each filter consists of rules that exclude or justify certain model objects, or individual coverage objective outcomes. Multiple filter files can be applied to coverage results for a model. Furthermore, multiple models can also make use of the same filter file.

Coverage filters can be created and applied either before or after simulating a model. Both workflows are described in this example.
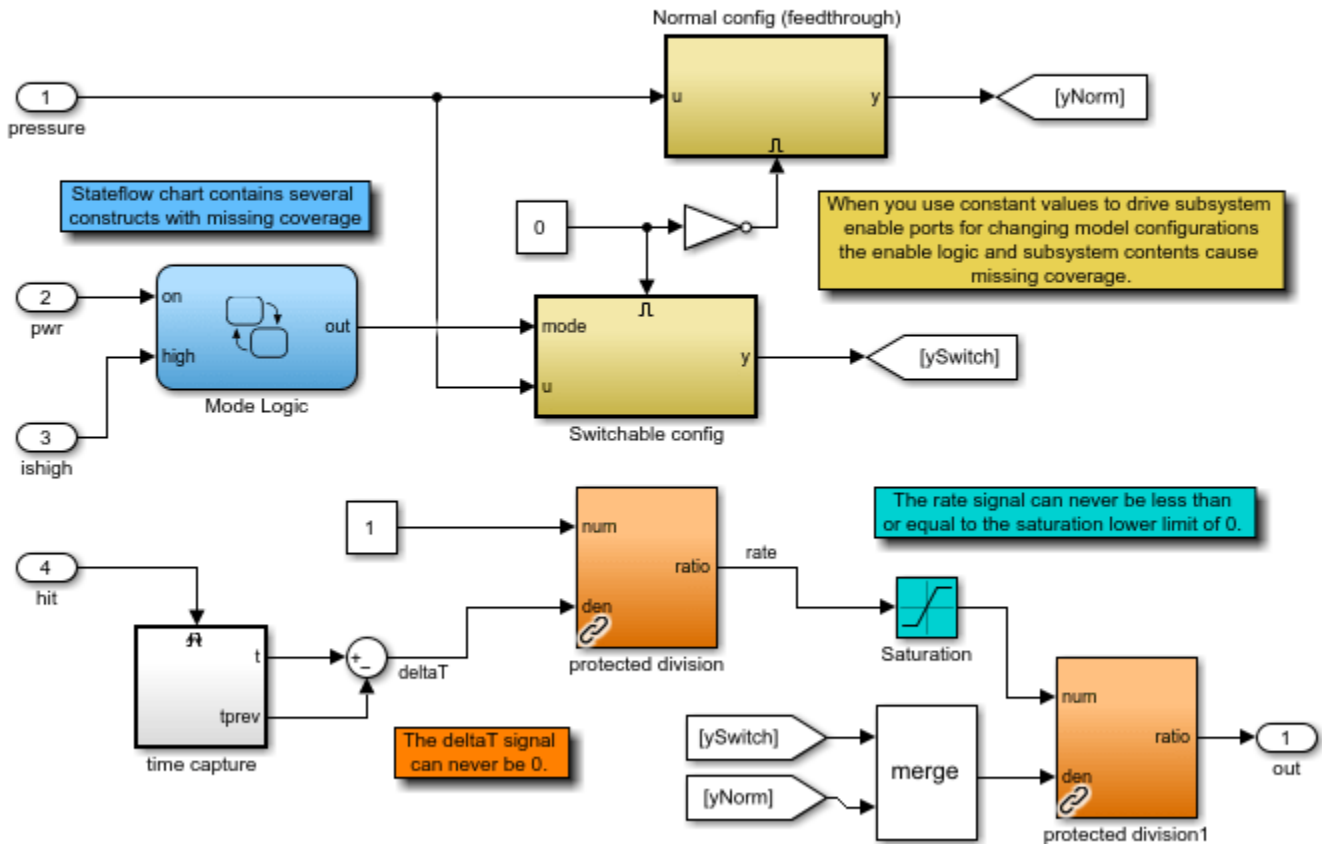
Open Model

This example makes use of the `slvnvdemo_covfilt` model. The model shows some common patterns that might need to be filtered from coverage results. Open the model.

```
open_system('slvnvdemo_covfilt');
```

Coverage Filtering Example

This model contains several constructs that prevent complete model coverage. Filtering these constructs from coverage results allows you to focus on other aspects of missing coverage that can and should be tested.

Normal config (feedthrough)

1 pressure

Stateflow chart contains several constructs with missing coverage

When you use constant values to drive subsystem enable ports for changing model configurations the enable logic and subsystem contents cause missing coverage.

2 pwr

3 ishigh

Mode Logic

Switchable config

[yNorm]

[ySwitch]

The rate signal can never be less than or equal to the saturation lower limit of 0.

1

4 hit

time capture

deltaT

protected division

The deltaT signal can never be 0.

Saturation

[ySwitch]

[yNorm]

merge

protected division1

1 out

Copyright 2010-2020 The MathWorks, Inc.

Specify Items to Exclude from Coverage Results before Simulation

The library block `slvnvdemo_covfilt_lib/protected division` protects against division by zero. If you determine that your testing is not expected or intended to fully cover every instance of this block in this context, the block can be excluded from coverage results.

In the Simulink Editor, right-click an instance of the `protected division` library block and navigate to the **Coverage** options. The options for this block allow you to filter the specific instance of the library or all references to this library. Select **Exclude referenced library: slvnvdemo_covfilt_lib/protected division** to filter all references.
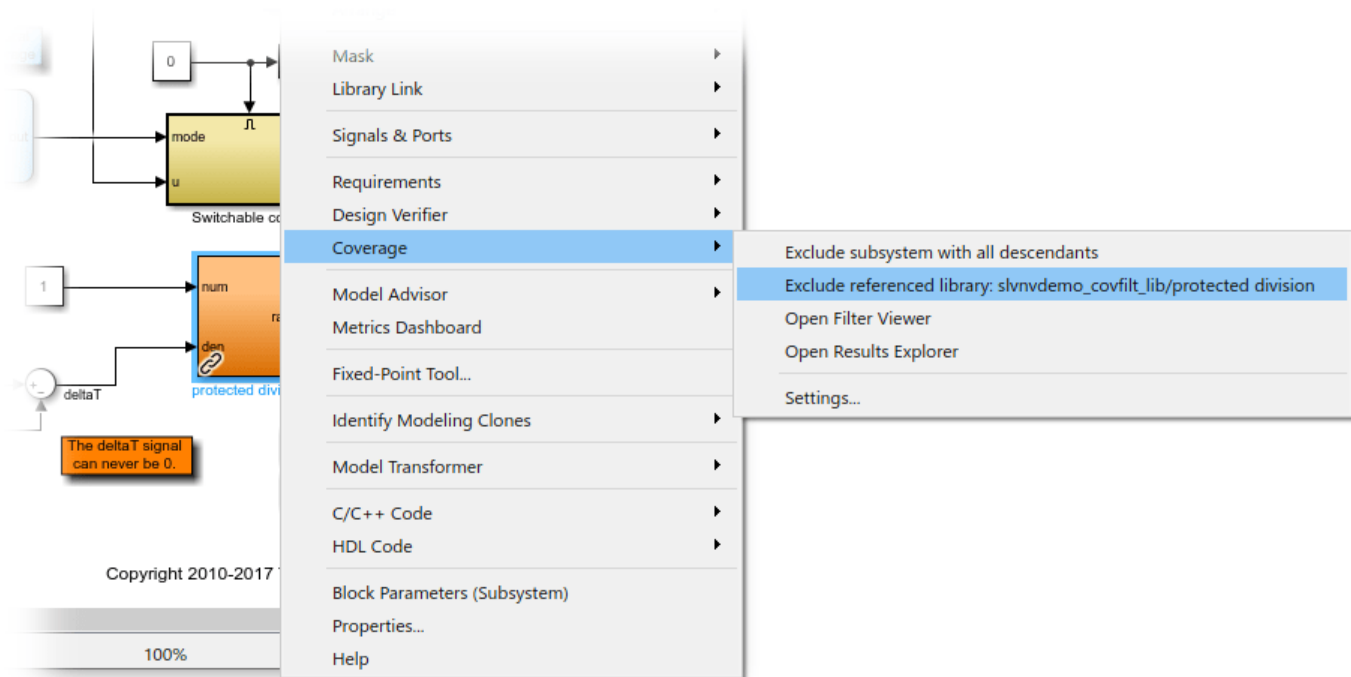
This opens the **Filter Editor** section of the **Coverage Results Explorer**. Note that a new filter file, initially named `Untitled`, has been created, and the filter rule for excluding all references to the library block has been added.
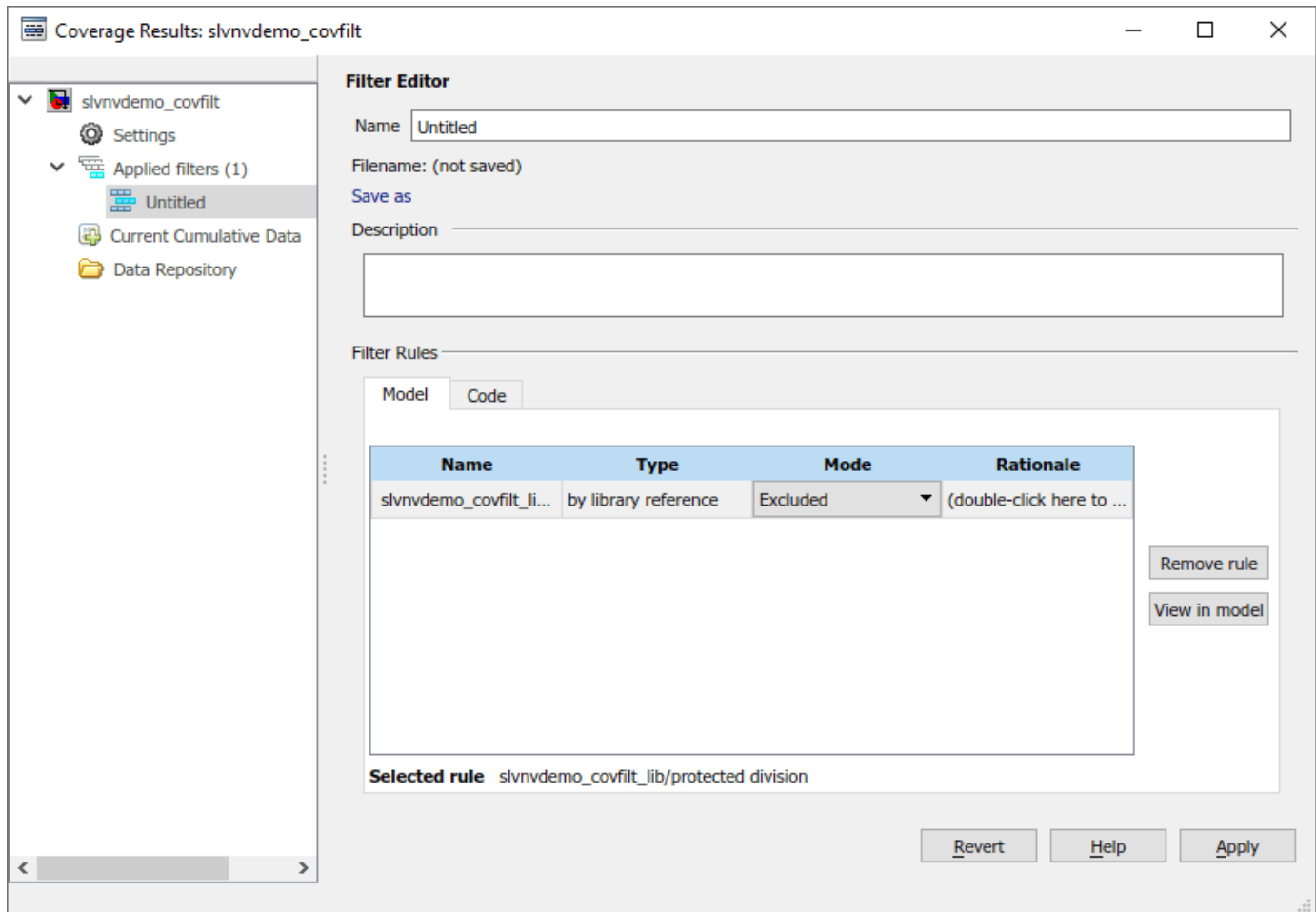
Specify a **Name** and **Description** for the new filter file. In the table, open the **Rationale** field for the new rule and enter text describing why this block is excluded, such as `"division by zero protection"`. Click **Apply** when finished to save the filter file. A file dialog will prompt you to specify where to save this file.
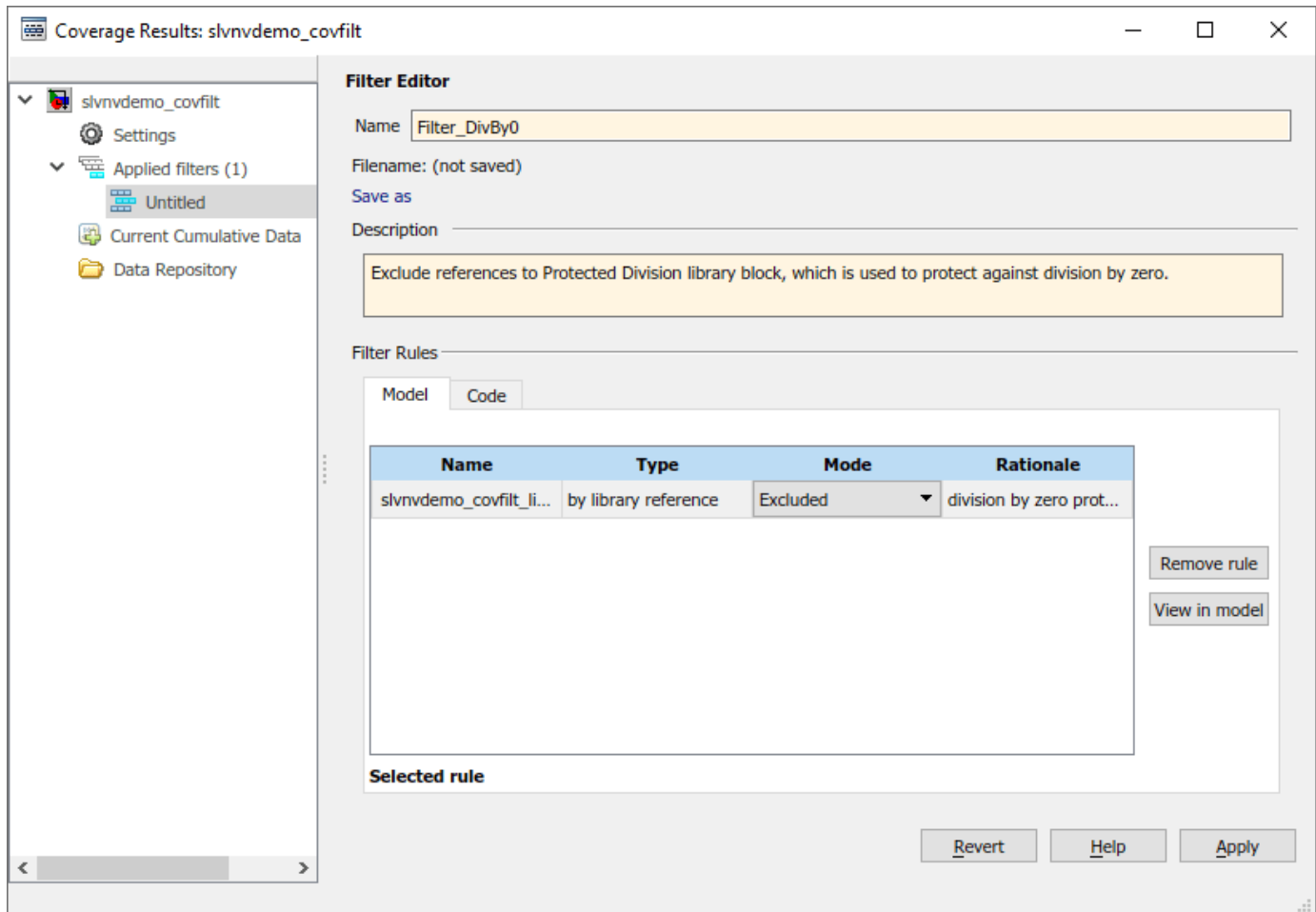
Reuse Existing Filter File

You can share and reapply filter files with generalized rules to filter coverage results for various different models that contain similar constructs.

For example, the existing filter file `Filter_Tick.cvf` captures a rule to exclude the Stateflow temporal event `tick` from coverage results. This event can never be false and, therefore, could prevent full Condition and MCDC coverage in any model using `tick` in event-based temporal logic in Stateflow.

Because `slvnvdemo_covfilt/Mode Logic` contains such a construct, you can apply the filter file `Filter_Tick.cvf` to the model.

To apply this existing filter file, right-click on the **Applied filters** node in the **Coverage Results Explorer** and select **Load filter**. In the file dialog that opens, select `Filter_Tick.cvf` and click **Open**.
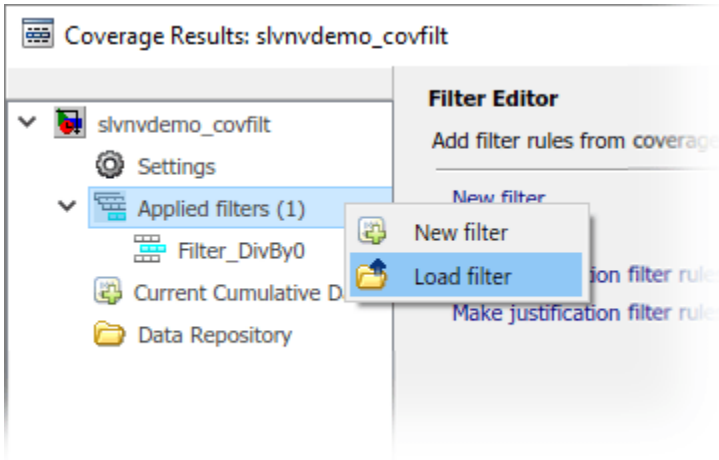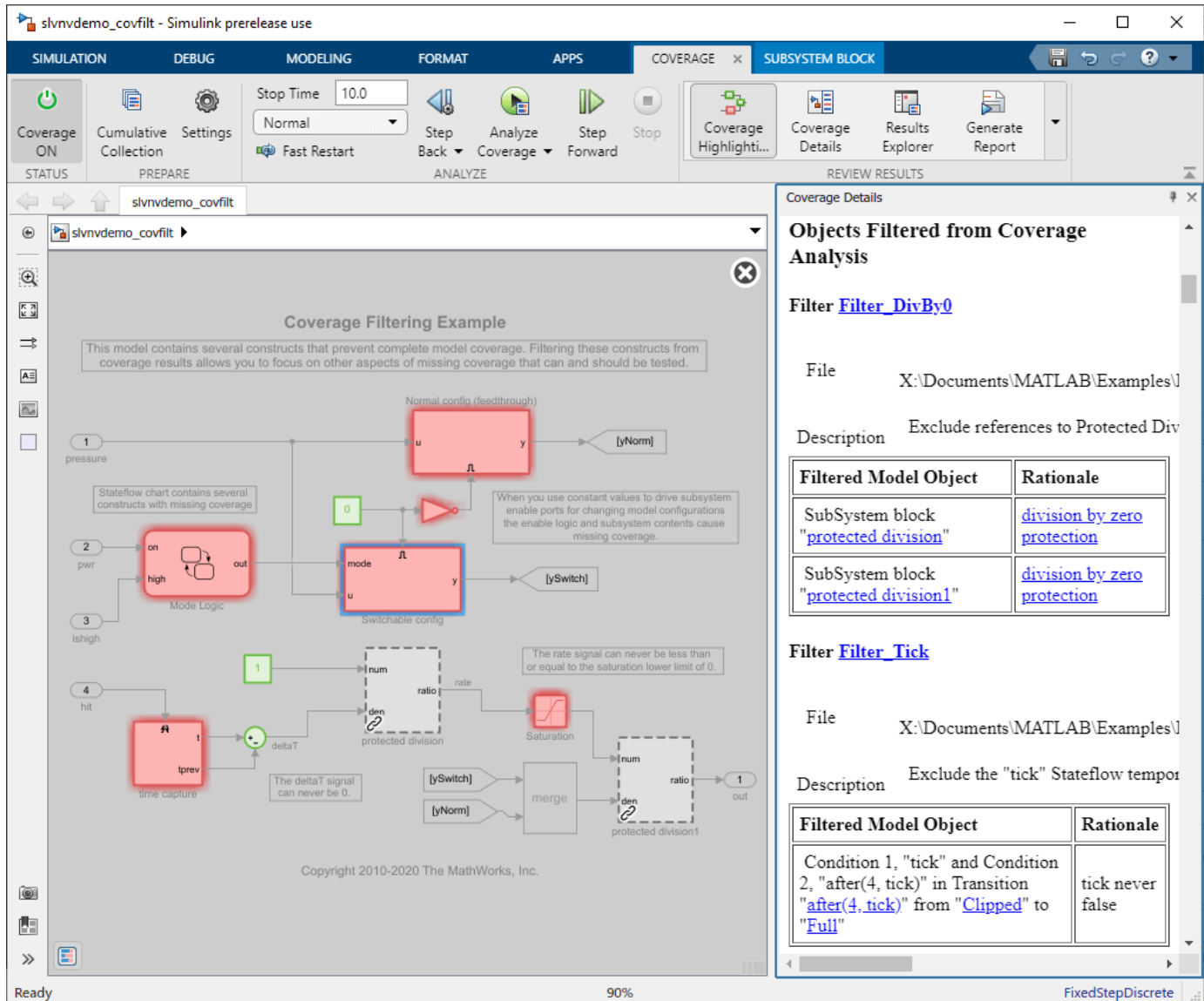
Note that **Applied filters** now lists both `Filter_DivBy0` and `Filter_Tick`.
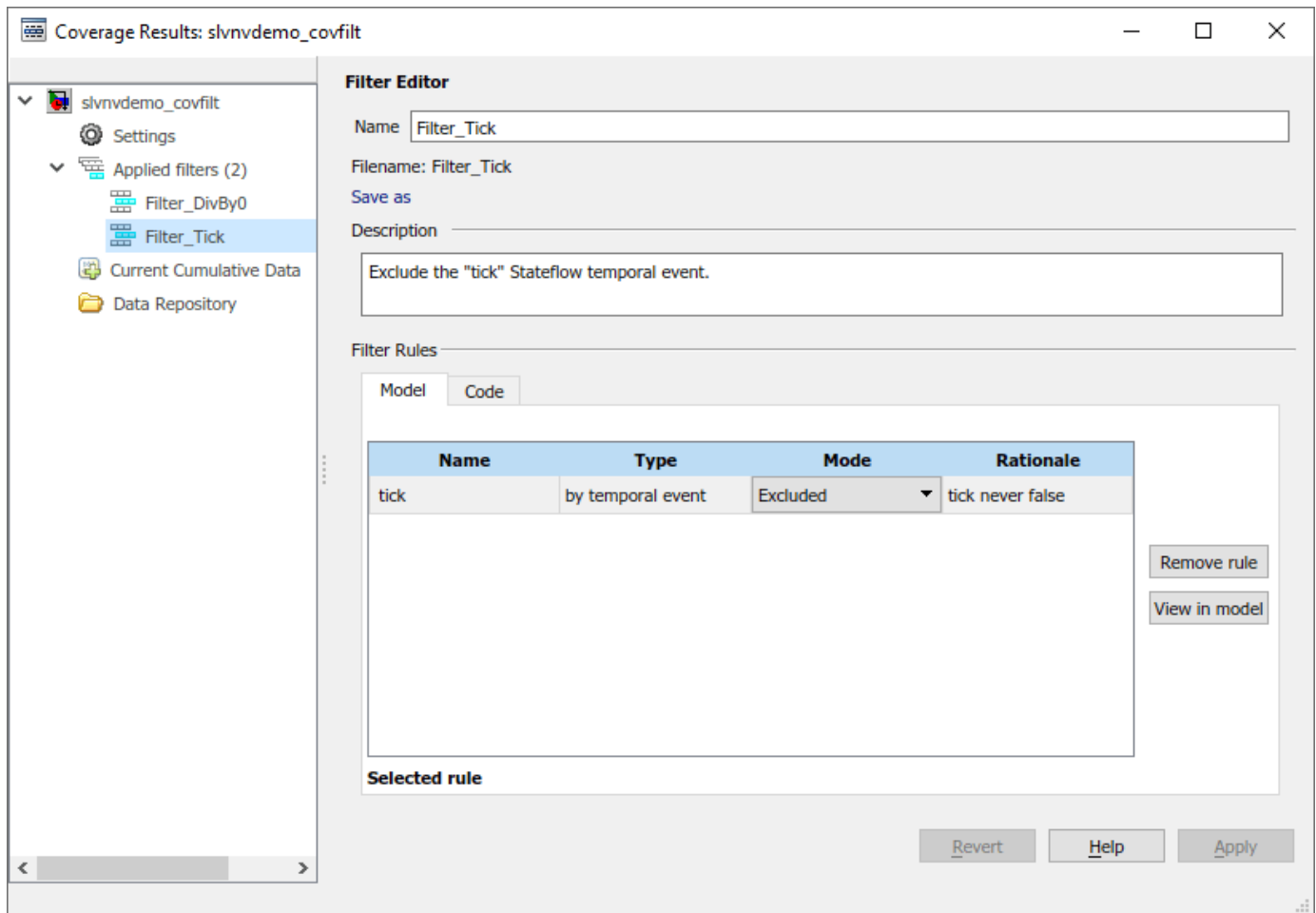
Simulate and Review Filtered Coverage Results

Click the **Run (Coverage)** button to simulate the model and record coverage. When the simulation completes, coverage results are highlighted on the model and the **Coverage Details** window is opened.

Notice that both references to the `protected division` library block are colored gray in the Simulink canvas, indicating that they have been excluded from the coverage results.

Scroll through the content in the **Coverage Details** window, and note the section titled **Objects Filtered from Coverage Analysis**. This section lists each of the excluded elements and the corresponding rationales for each and is organized by filter file. Both `Filter_DivBy0` and `Filter_Tick` have been applied.

Create a New Filter File

The filter files used above capture generalized filter rules that could be broadly applicable to many different models.

Create another filter file to separately capture filter rules exclusively relevant to this model.

In the **Coverage Results Explorer**, right-click on the **Applied filters** node and select **New filter**.
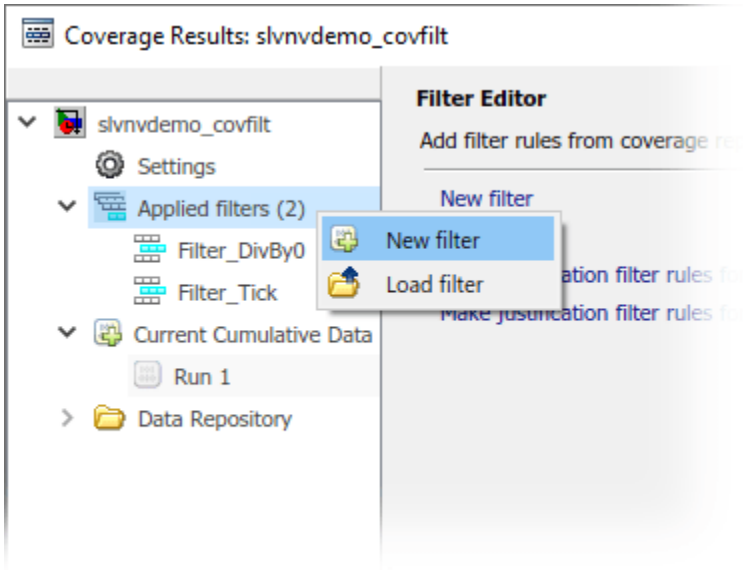
Enter a name and description for this filter file. Click **Apply** and specify where to save the file.
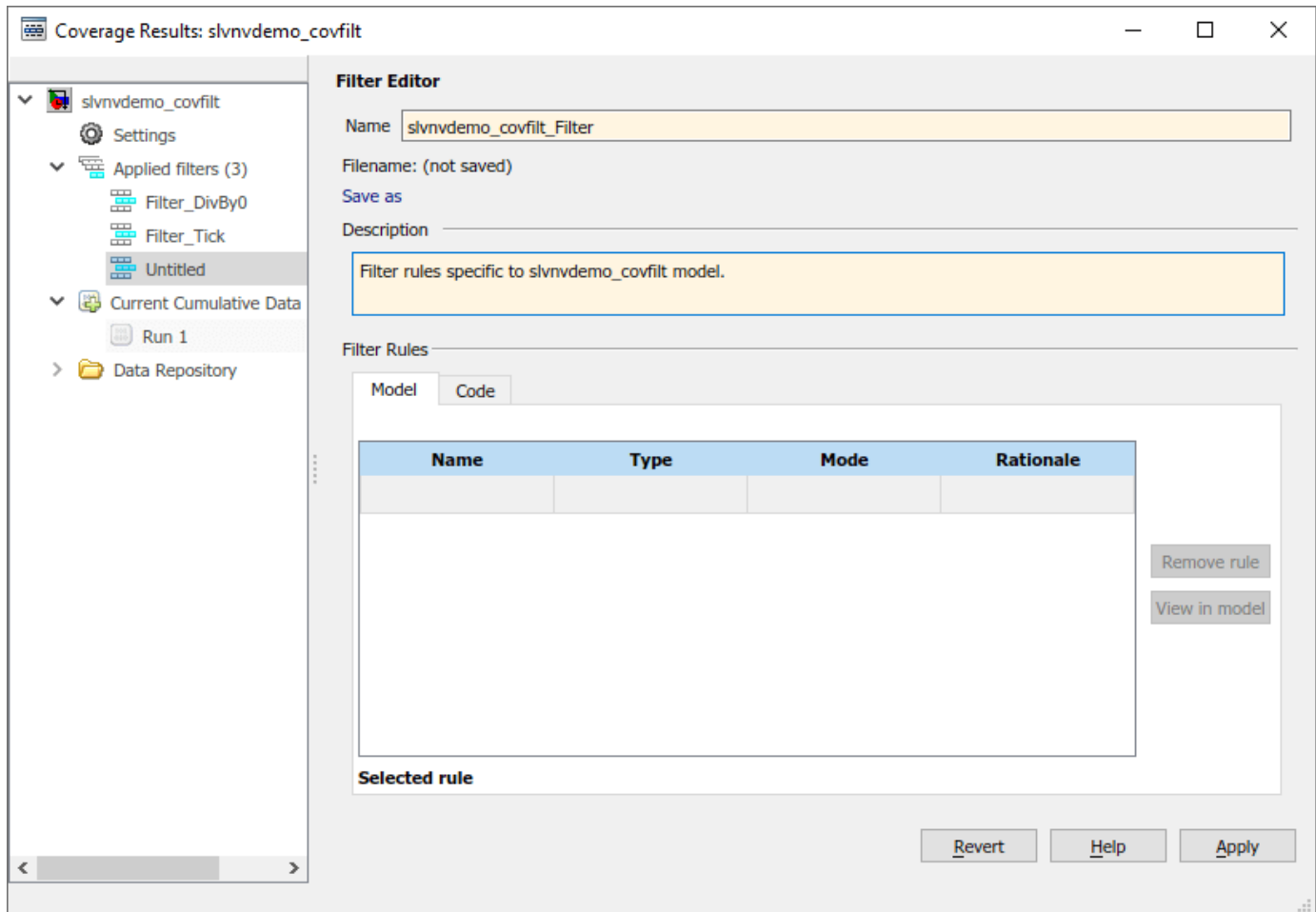
Exclude Items from Coverage Results after Simulation

The previous sections of this example show how to specify filter rules to apply before running a simulation. You can also create and apply filter rules to coverage results after simulation. This allows you to review coverage results, create or adjust filters, and generate a new coverage report without having to rerun the simulation.

Consider, for example, the Switchable config subsystem. It is a common design pattern to use constant values to drive subsystem enable ports for changing model configurations; however, the enable logic and subsystem contents might lead to missing coverage. This configuration is not used in this model, and therefore can be excluded from coverage results.

In the Simulink Editor, click on the Switchable config subsystem. The **Coverage Details** window will navigate to the details for this subsystem. Under these details, click the **Justify or Exclude** link.

A new filter rule is added to the currently selected filter file in the **Filter Editor**. By default, the mode for this rule is set to `"Excluded"`. Enter the rationale for this rule, such as `"unused config"`.
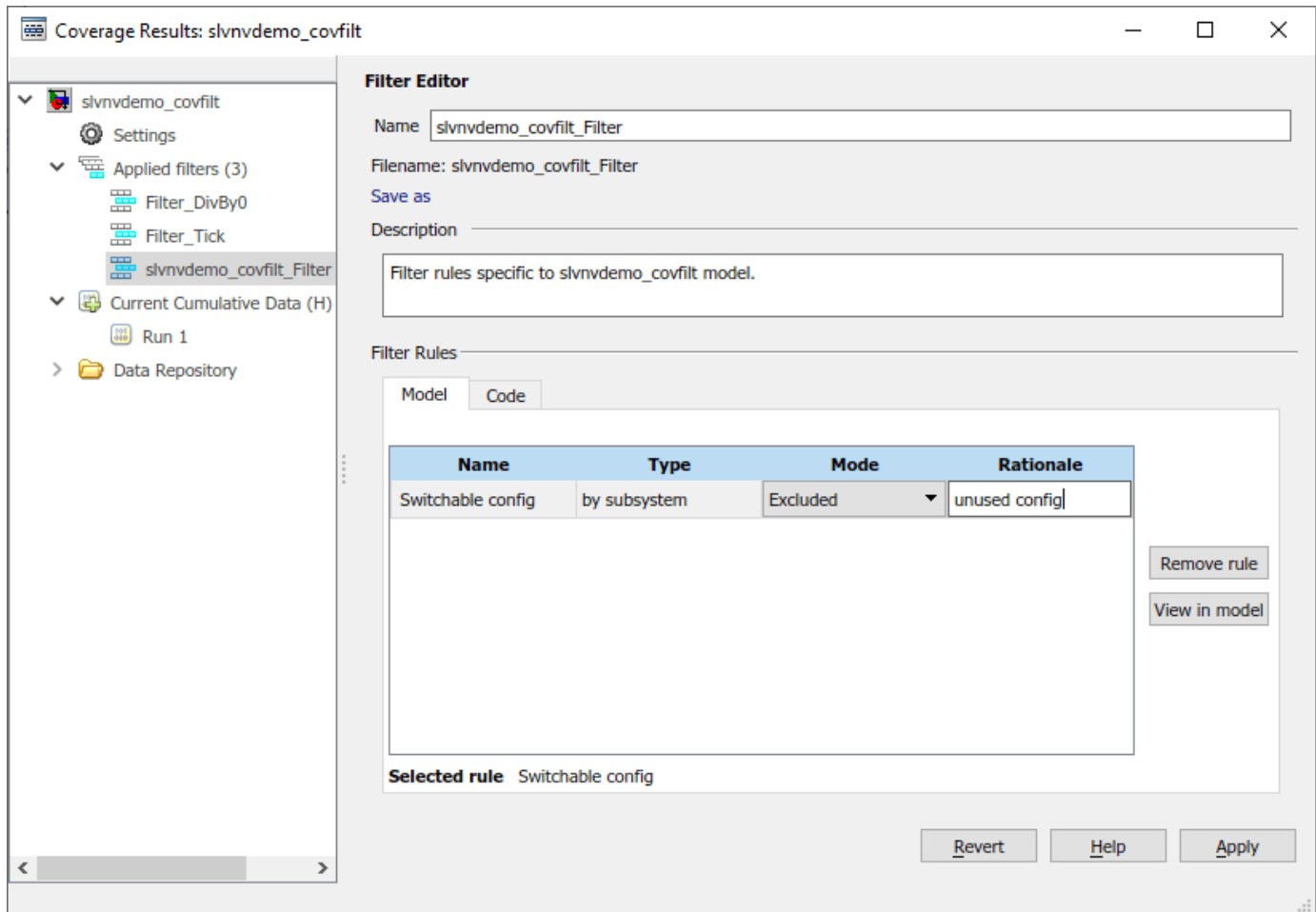
Click **Apply** when finished. This saves the changes to the filter file and automatically updates the coverage results displayed on the model. Notice that the Switchable config subsystem is now shown as gray, indicating that it has been excluded from the coverage results.

Justify Individual Objective Outcome from the Coverage Results

In the Simulink Editor, click on the `Saturation` block and review the coverage results in the **Coverage Details** window. Notice that two Decision outcomes are unsatisfied. This is because the `Saturation` block has a lower limit of `0` and an upper limit of `200`. However, the input to this block is the `rate` signal, which can never be less than or equal to `0`. As such, the lower limit of the `Saturation` block is not expected to be fully exercised, so the corresponding Decision outcome can be justified.

Click on the **Add justification rule** icon next to the *false* outcome for Decision `"input > lower limit"`.

A new filter rule is added to the currently selected filter file in the **Filter Editor**. Specify a justification rationale, such as `"rate > 0"`.

Click **Apply** when finished to save the filter file and update the coverage results shown on the model.
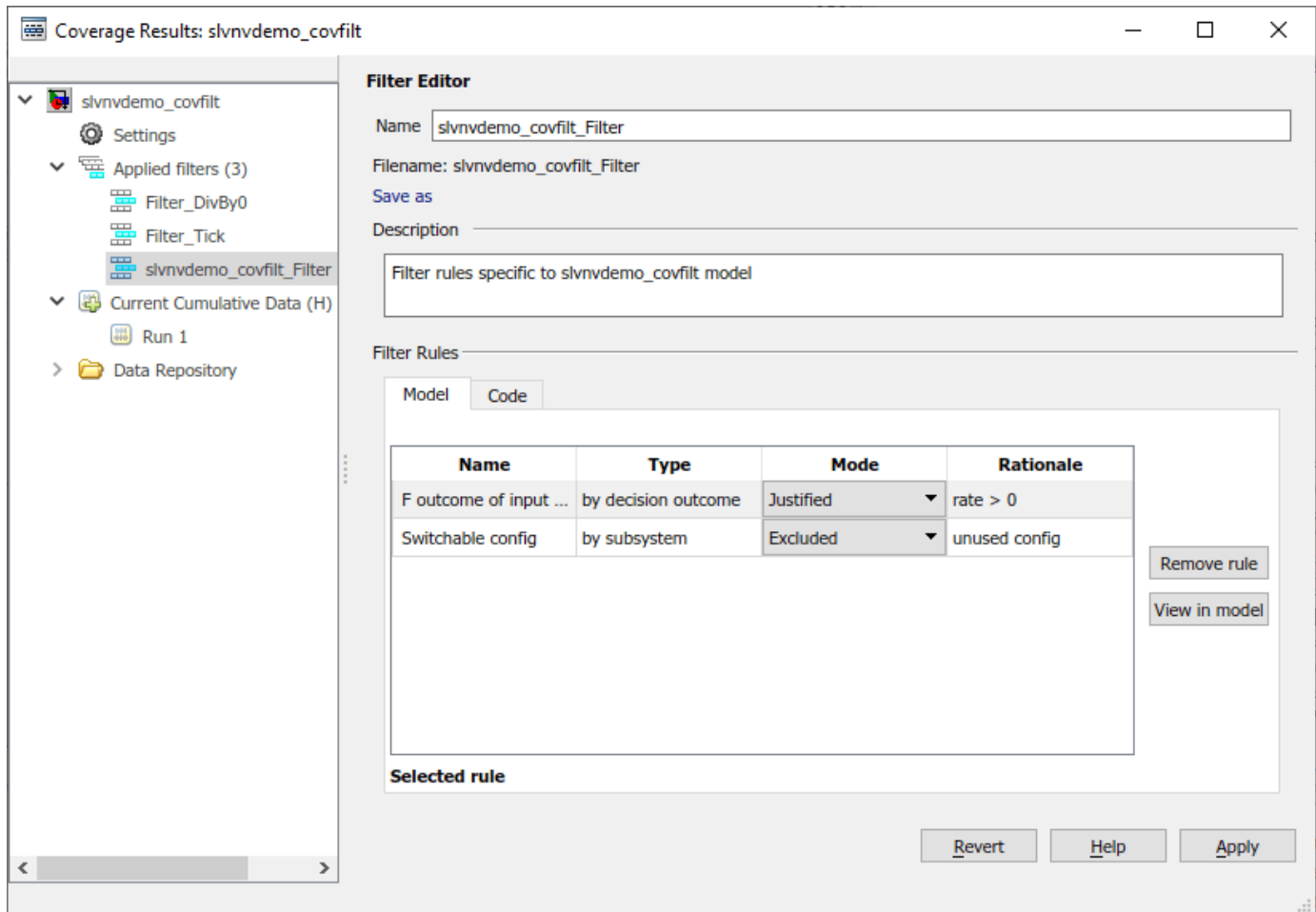
Notice that in the **Coverage Details** window, the `Saturation` block's justified outcome is highlighted in light blue and has a link to the rationale. The *true* outcome of the decision `"input >= upper limit"` is not filtered and is reported as unsatisfied. As such, the `Saturation` block is still highlighted red in the model to indicate missing coverage.

Automatically Generate Filter Rules for Dead Logic

In some cases, missing coverage is due to dead logic. The associated coverage objectives are unsatisfiable. If this logic is meant for constructs that you do not wish to remove from your model, then those coverage objectives can be justified.

If you have a Simulink Design Verifier™ license, then you can automatically create justification filter rules for dead logic.

In the **Coverage Results Explorer**, select the **Applied filters** node. In the **Filter Editor** pane, select the option **Make justification filter rules for dead logic (using Simulink Design Verifier)**.
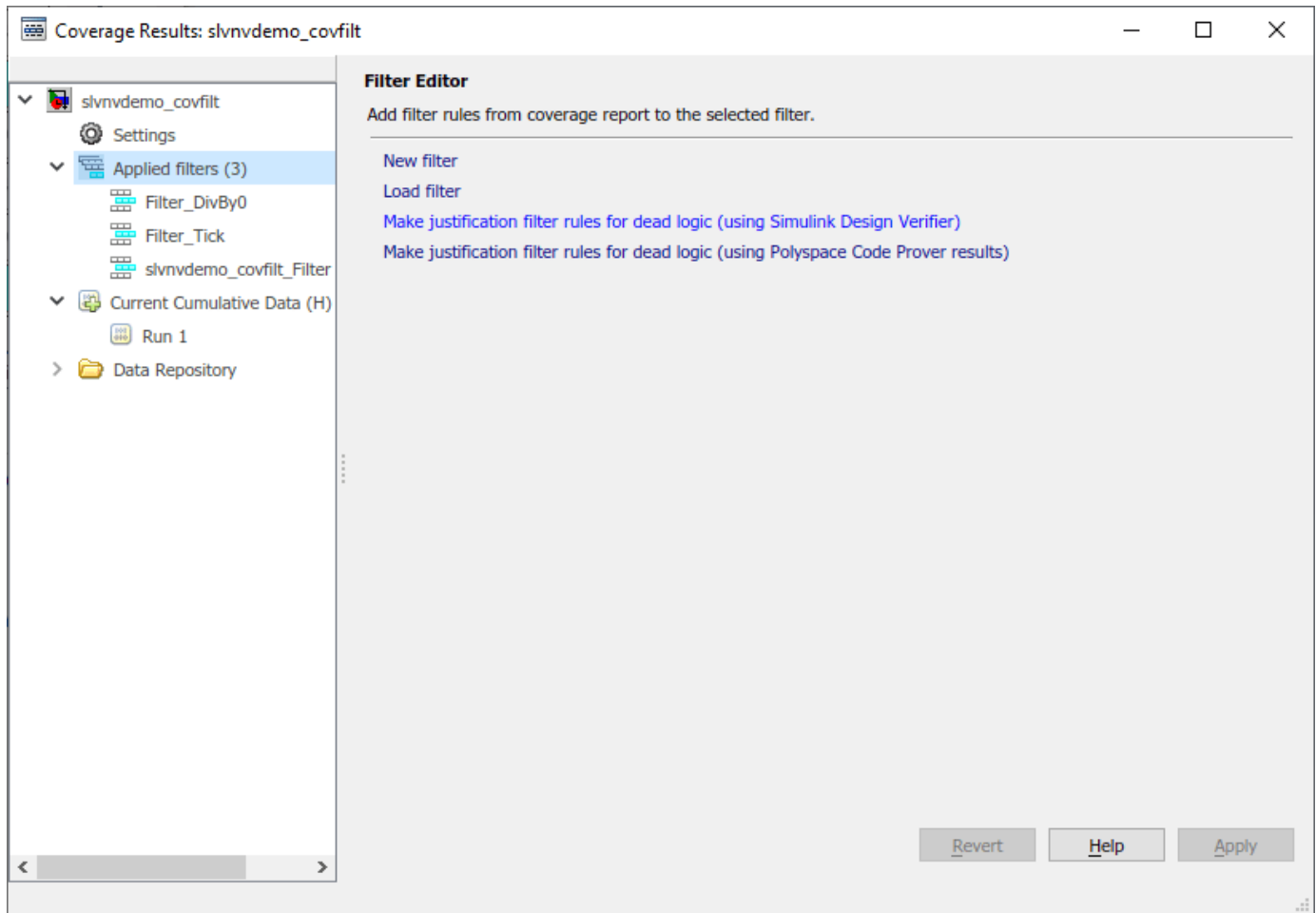
This option uses Simulink Design Verifier™ to analyze the model for dead logic. Upon completion, a new filter file is created, and justification rules are added for each of the corresponding coverage outcomes.
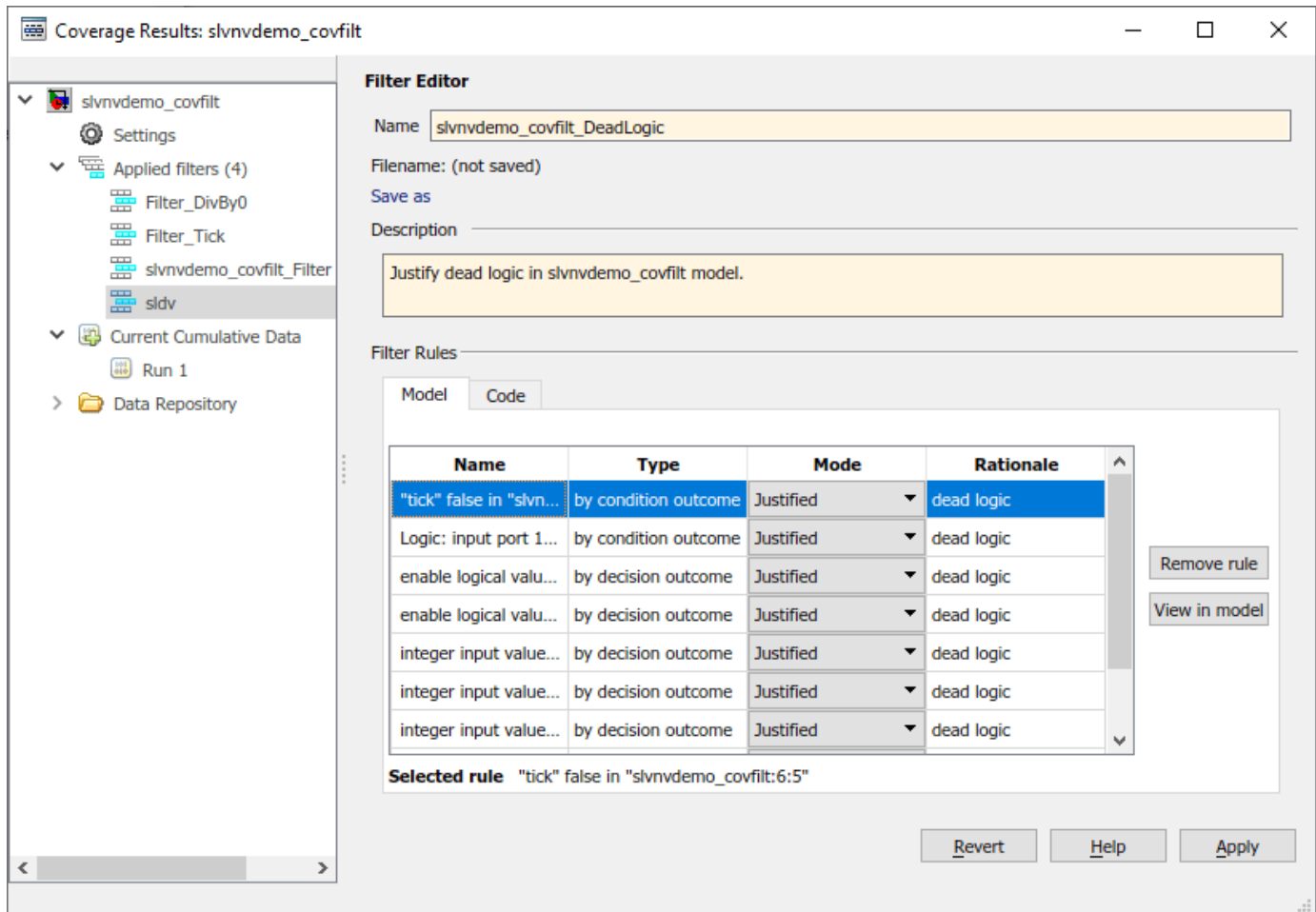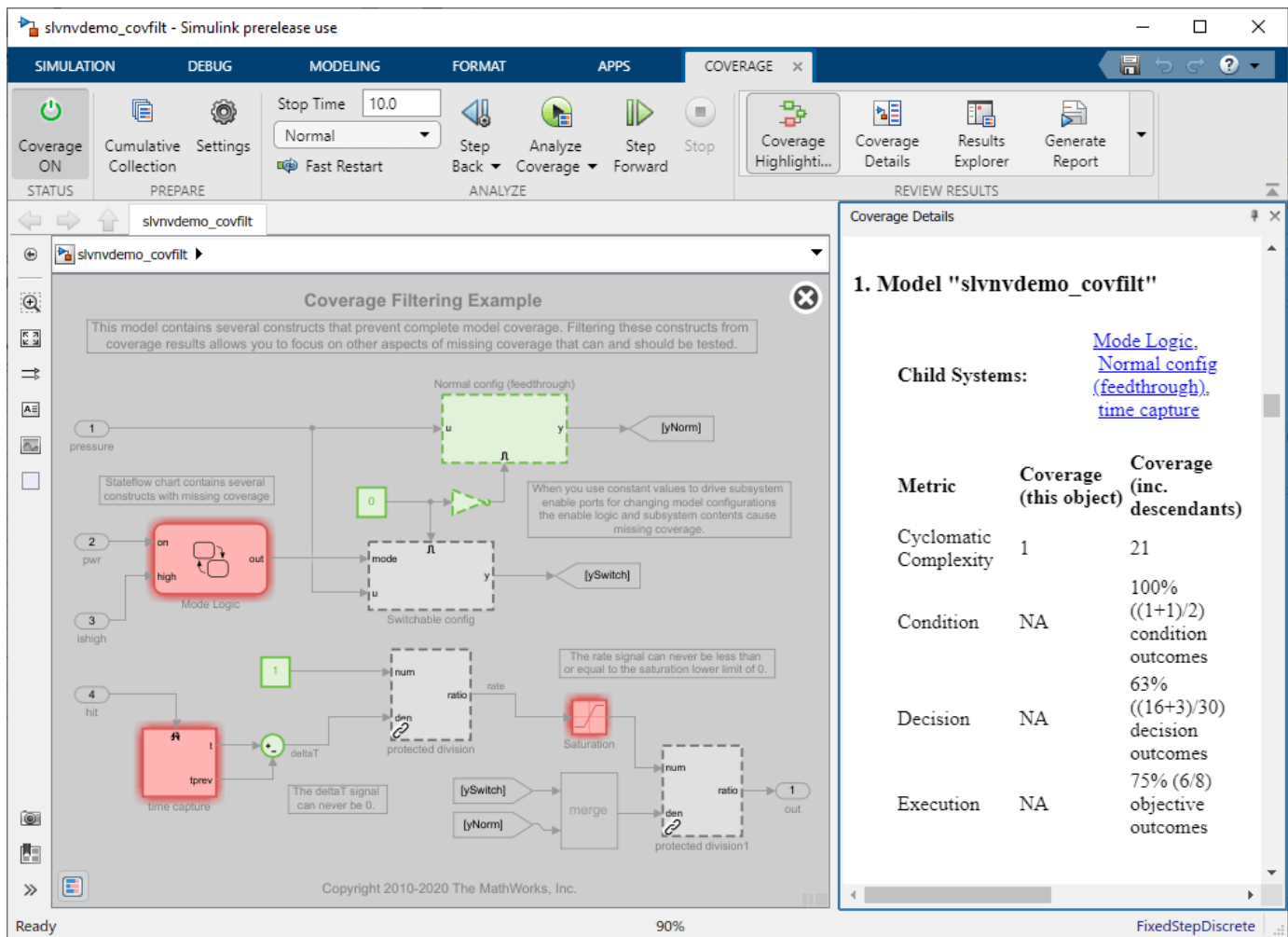
Specify a name and description for this filter file. Click **Apply** when finished. In the resulting file dialog, specify where to save this filter file.

Close the **Simulink Design Verifier Results** windows that opened.

Review Filtered Coverage Results

In the Simulink Editor, navigate to the **Review Results** section of the **Coverage** app, and click **Coverage Highlighting**.

After applying all four filters used in this example, the simulation now achieves 100% Condition, 63% Decision, and 75% Execution coverage for this model. Note that the coverage results no longer contain any model objects that receive MCDC, so this metric is not listed.

The remaining missing coverage in the `Mode Logic` chart, `time capture` subsystem, and `Saturation` block indicates inadequate testing and should be addressed by using further simulation with input values that more thoroughly exercise these elements.

Conclusion

Coverage filtering provides various methods for indicating aspects of your model that are not expected or intended to be fully exercised in the current testing context.

Filtering model objects and coverage objective outcomes that do not need to be exercised allows you to focus on the missing coverage for constructs that can and should be tested.

Automating Model Coverage Tasks

Automating Model Coverage Tasks

You can automate coverage analysis in a script by using the Simulink Coverage functions and classes. For example, you might want to collect coverage data by simulating the same model with different model parameters. Instead of changing parameters manually, you can run the simulations and collect the coverage data in a loop.

Collect Coverage Data Using a Script

This example shows how to collect coverage data using `sim`.

Load the Model

First, load the model and the system you want to analyze into memory.

```
load_system('slvnvdemo_ratelim_harness');
```

Set Coverage Settings

Set up the coverage parameters using one of the methods described in `sim`, such as a simulation input, parameter structure, or name-value pairs. For example, in order to use a structure of parameters, set up a structure whose fields are names of configuration parameters, and whose values are the corresponding values of those parameters.

```
paramStruct.CovEnable = 'on';
paramStruct.CovMetricStructuralLevel = 'Decision';
paramStruct.CovSaveSingleToWorkspaceVar = 'on';
paramStruct.CovSaveName = 'covData';
paramStruct.CovScope = 'Subsystem';
paramStruct.CovPath = '/Adjustable Rate Limiter';
paramStruct.StartTime = '0.0';
paramStruct.StopTime = '2.0';
```

For an example that uses the `Simulink.SimulationInput` object, see "Record Coverage in Parallel Simulations by Using Parsim" on page 8-30.

Set up a Test and Simulate the Model

The example model uses input values that are defined in the MATLAB® workspace. The values used in this example are defined in a data file called `within_lim.mat`. You can use `load` to load the file into the workspace.

```
load within_lim.mat;
```

Simulate the model using `sim` with `paramStruct` as an additional input to collect coverage data using the specified parameters.

```
simOut = sim('slvnvdemo_ratelim_harness',paramStruct);
```

For a complete list of Simulink Coverage configuration parameters, see "Coverage Settings".

Generate a Coverage Report

You can generate an HTML report to view the coverage data that your simulation generates with `cvhtml`. The first input is the name of the coverage report that will be saved in the current directory.

The second input is the `cvdata` object that was saved to the workspace based on the model parameters `CovSaveSingleToWorkspaceVar` and `CovSaveName`.

You can generate the report without automatically opening it by using the flag `'-sRT=0'` as the third input to `cvhtml`.

```
cvhtml('covReport',covData,'-sRT=0');
```

Save Coverage Data

Use `cvsave` to save the coverage results. The first input is the name of the coverage data file, and the second input is the `cvdata` object.

```
cvsave('covdata',covData);
```

Close the Model

Exit the coverage environment by using `cvexit` and close the model by using `close_system`. A second input of `0` indicates that you do not want to save model before closing.

```
cvexit
close_system('slvnvdemo_ratelim_harness',0);
```

Differences between sim and the Run Button

When you run a simulation with coverage enabled by using the **Run** button, the coverage report opens automatically and **Coverage Highlighting** is enabled by default. When you run a simulation programmatically by using `sim`, the coverage report does not open and **Coverage Highlighting** is not enabled.

- To see coverage results displayed using model highlighting, use `cvmodelview`.
- To see a coverage report, use `cvhtml`.
- To open the Results Explorer, open the model in Simulink. In the **Apps** tab, click **Coverage Analyzer**. Then click **Results Explorer**.

For another detailed example, see "Command Line Verification Tutorial" on page 8-7.

Collecting Coverage with Simulink Test

If you have a Simulink Test license, you can use the Test Manager to collect coverage data. For more information, "Run a Test Case and Collect Coverage" (Simulink Test).

See Also
`sim` | `Simulink.SimulationInput` | `cvsim` | `cvtest` | `cvhtml`

More About
- "Programmatically Retrieve Coverage Details from Results" on page 8-4
- "Coverage Settings"
- "Record Coverage in Parallel Simulations by Using Parsim" on page 8-30

Programmatically Retrieve Coverage Details from Results

Analyze Coverage Data Using A Script

This example shows how to load, parse, and query coverage data using a script.

Load Coverage Data

Load the model, then restore saved coverage data from the file `covdata.cvt` using `cvload`. The data and test settings are retrieved in a cell array. The test settings are stored in a `cvtest` object that contains the parameters from the simulation that created the coverage data.

```
load_system('slvnvdemo_ratelim_harness');
[savedSettings,savedData] = cvload('covdata');
savedData = savedData{1};
```

Extract Information from Coverage Data Objects

Retrieve coverage information from a block path or block handle by using `decisioninfo`. The output is a vector with the achieved and total outcomes for a single model object.

```
subsysCov = decisioninfo(savedData,...
                   'slvnvdemo_ratelim_harness/Adjustable Rate Limiter')


subsysCov =

     5     6
```

Determine the percentage coverage achieved by using `decisioninfo`.

```
percentCov = 100 * (subsysCov(1)/subsysCov(2))


percentCov =

   83.3333
```

Specify that you want to extract the decision coverage data for the switch block called Apply Limited Gain by using `decisioninfo`. This returns a structure which contains the decisions and outcomes.

```
[blockCov,desc] = decisioninfo(savedData, ...
        'slvnvdemo_ratelim_harness/Adjustable Rate Limiter/Apply limited gain');
descDecision = desc.decision;
outcome1 = desc.decision.outcome(1)
outcome2 = desc.decision.outcome(2)


outcome1 =

  struct with fields:

             text: 'false (out = in3)'
    executionCount: 0
        executedIn: []
```

```
        isFiltered: 0
        isJustified: 0
    filterRationale: ''


outcome2 =

  struct with fields:

             text: 'true (out = in1)'
   executionCount: 101
       executedIn: []
       isFiltered: 0
       isJustified: 0
    filterRationale: ''
```

From the `decisioninfo` output, you can see that the switch block called Apply Limited Gain was never false because the false case `executionCount` field has a value of `0`. If this behavior is expected, and you did not intend to execute this case with your tests, you can add a filter rule to justify this missing coverage using the `slcoverage.Filter` class.

First, query for the block instance to be filtered, because we only need to filter the one block instance that received incomplete coverage, and not all instances of that block type. Then use the `slcoverage.BlockSelector` class with the `BlockInstance` selector type to designate one block instance for filtering.

```
id = getSimulinkBlockHandle('slvnvdemo_ratelim_harness/Adjustable Rate Limiter/Apply limited gain
sel = slcoverage.BlockSelector(slcoverage.BlockSelectorType.BlockInstance,id);
```

Create a filter object and a filter rule using the `slcoverage.Filter` and `slcoverage.FilterRule` classes.

```
filt = slcoverage.Filter;
rule = slcoverage.FilterRule(sel,'Edge case',slcoverage.FilterMode.Justify);
```

Add the rule to the filter using the `addRule` method. Then save the new filter file with the `save` method.

```
filt.addRule(rule);
filt.save('blfilter');
```

Create a new `cvdata` object from the original object, and apply the filter file to it. Use `decisioninfo` on the filtered coverage data to see that there is now 100% decision coverage because the justified objectives are counted as satisfied.

```
FilteredData = savedData;
FilteredData.filter = 'blfilter';
newCov = decisioninfo(FilteredData,...
                  'slvnvdemo_ratelim_harness/Adjustable Rate Limiter')
percentNewCov = 100 * (newCov(1)/newCov(2))


newCov =

     6     6
```

```
percentNewCov =

   100
```

Coverage Information Functions

After you collect coverage data, you can extract specific coverage information from the `cvdata` object by using the following functions. Use these functions to retrieve the specified coverage information for a block, subsystem, or Stateflow chart in your model, or for the model itself.

You can turn on coverage highlighting on your Simulink model by using `cvmodelview`. You can also view the coverage report using `cvhtml`.

- `complexityinfo` — Cyclomatic complexity coverage
- `executioninfo` — Execution coverage
- `conditioninfo` — Condition coverage
- `decisioninfo` — Decision coverage
- `mcdcinfo` — Modified condition decision coverage (MCDC)
- `overflowsaturationinfo` — Saturate on integer overflow coverage
- `relationalboundaryinfo` — Relational boundary coverage
- `sigrangeinfo` — Signal range coverage
- `sigsizeinfo` — Signal size coverage
- `tableinfo` — Lookup table block coverage
- `getCoverageInfo` — Coverage for Simulink Design Verifier blocks

For an example that uses these functions, see "Extracting Detailed Information from Coverage Data" on page 8-16.

See Also
`slcoverage.Filter` | `slcoverage.FilterRule` | `slcoverage.BlockSelector` | `slcoverage.MetricSelector`

More About
- "Automating Model Coverage Tasks" on page 8-2
- "Operations on Coverage Data" on page 8-24

Command Line Verification Tutorial

This example creates three test cases for an adjustable rate limiter and analyzes the resulting model coverage using the command-line API of the Model Coverage tool.

Simulink® Model for the Adjustable Rate Limiter

The Simulink® subsystem Adjustable Rate Limiter is a rate limiter in the model 'slvnvdemo_ratelim_harness'. It uses three switch blocks to control when the output should be limited and the type of limit to apply.

Inputs are produced with the From Workspace blocks 'gain', 'rising limit', and 'falling limit', which generate piecewise linear signals. The values of the inputs are specified with six variables defined in the MATLAB® workspace: t_gain, u_gain, t_pos, u_pos, t_neg, and u_neg.

Open the model and the Adjustable Rate Limiter subsystem.

```
modelName = 'slvnvdemo_ratelim_harness';
open_system(modelName);
open_system([modelName,'/Adjustable Rate Limiter']);
```
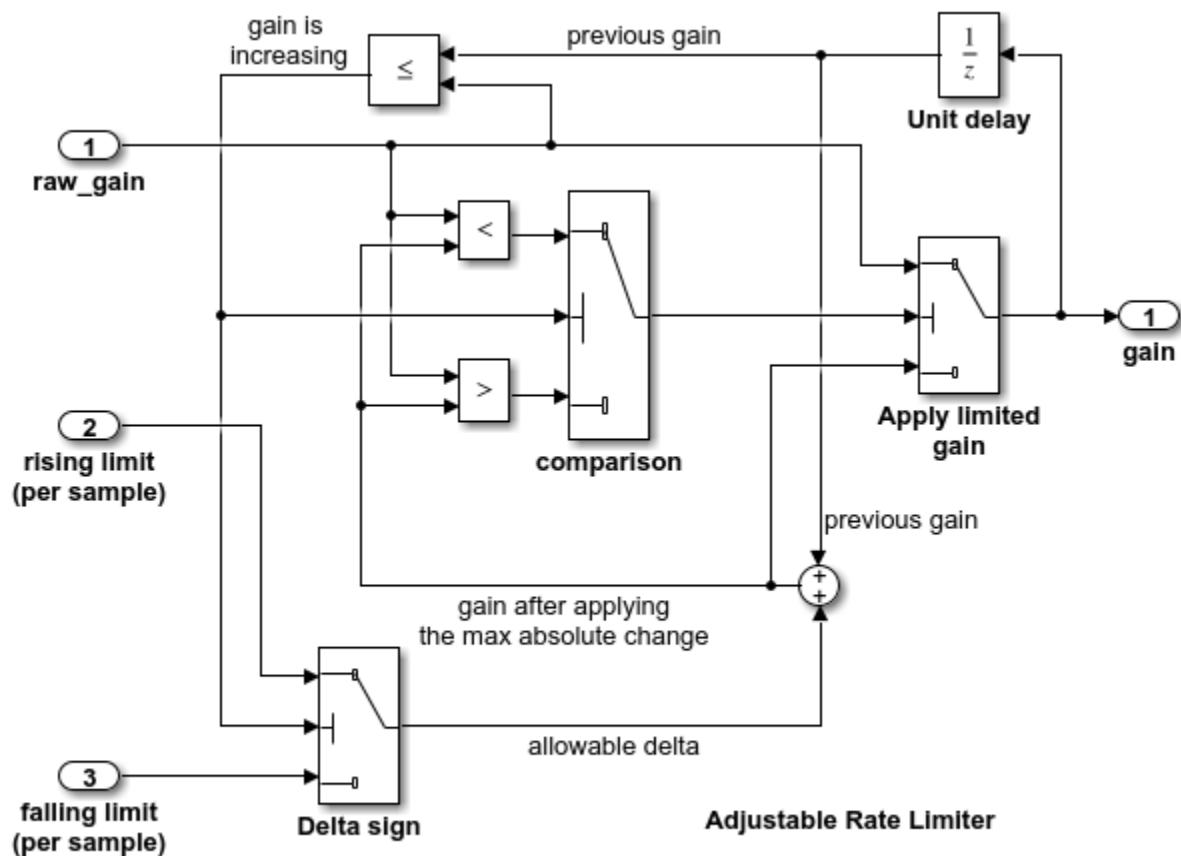
Creating the First Test Case

The first test case verifies that the output matches the input when the input values do not change rapidly. It uses a sine wave as the time varying signal and constants for rising and falling limits.

```
t_gain = (0:0.02:2.0)';
u_gain = sin(2*pi*t_gain);
```

Calculate the minimum and maximum change of the time varying input using the MATLAB diff function.

```
max_change = max(diff(u_gain))
min_change = min(diff(u_gain))


max_change =

    0.1253


min_change =

    -0.1253

```

Because the signal changes are much less than 1 and much greater than -1, set the rate limits to 1 and -1. The variables are all stored in the MAT file 'within_lim.mat', which is loaded before simulation.

```
t_pos = [0;2];
u_pos = [1;1];
t_neg = [0;2];
u_neg = [-1;-1];

save('within_lim.mat','t_gain','u_gain','t_pos','u_pos','t_neg','u_neg');
```

Additional Test Cases

The second test case complements the first case with a rising gain that exceeds the rate limit. After a second it increases the rate limit so that the gain changes are below that limit.

```
t_gain = [0;2];
u_gain = [0;4];
t_pos = [0;1;1;2];
u_pos = [1;1;5;5]*0.02;
t_neg = [0;2];
u_neg = [0;0];

save('rising_gain.mat','t_gain','u_gain','t_pos','u_pos','t_neg','u_neg');
```

The third test case is a mirror image of the second, with the rising gain replaced by a falling gain.

```
t_gain = [0;2];
u_gain = [-0.02;-4.02];
t_pos = [0;2];
u_pos = [0;0];
t_neg = [0;1;1;2];
u_neg = [-1;-1;-5;-5]*0.02;

save('falling_gain.mat','t_gain','u_gain','t_pos','u_pos','t_neg','u_neg');
```

Defining Coverage Tests

The test cases are organized and executed using `sim`.

In this example, a simulation input object is used to set the coverage configuration.

```
covSet = Simulink.SimulationInput(modelName);
covSet = setModelParameter(covSet,'CovEnable','on');
covSet = setModelParameter(covSet,'CovMetricStructuralLevel','Decision');
covSet = setModelParameter(covSet,'CovSaveSingleToWorkspaceVar','on');
covSet = setModelParameter(covSet,'CovScope','Subsystem');
covSet = setModelParameter(covSet,'CovPath','/Adjustable Rate Limiter');
covSet = setModelParameter(covSet,'StartTime','0.0');
covSet = setModelParameter(covSet,'StopTime','2.0');
```

Executing Coverage Tests

Load the data for the first test case, set the coverage variable name, and execute the model using `sim`.

```
load within_lim.mat
covSet = setModelParameter(covSet,'CovSaveName','dataObj1');
simOut1 = sim(covSet);
dataObj1


dataObj1 = ... cvdata
              version: (R2021b)
                   id: 1533
                 type: TEST_DATA
                 test: cvtest object
               rootID: 1535
             checksum: [1x1 struct]
            modelinfo: [1x1 struct]
            startTime: 01-Sep-2021 12:03:48
             stopTime: 01-Sep-2021 12:03:48
    intervalStartTime: 0
     intervalStopTime: 0
  simulationStartTime: 0
   simulationStopTime: 2
               filter:
              simMode: Normal
```

Verify the first test case by checking that the output matches the input.

```
subplot(211)
plot(simOut1.tout,simOut1.yout(:,1),simOut1.tout,simOut1.yout(:,4))
xlabel('Time (sec)'), ylabel('Value'),
title('Gain input and output');
subplot(212)
plot(simOut1.tout,simOut1.yout(:,1)-simOut1.yout(:,4))
xlabel('Time (sec)'),ylabel('Difference'),
title('Difference between the gain input and output');
```
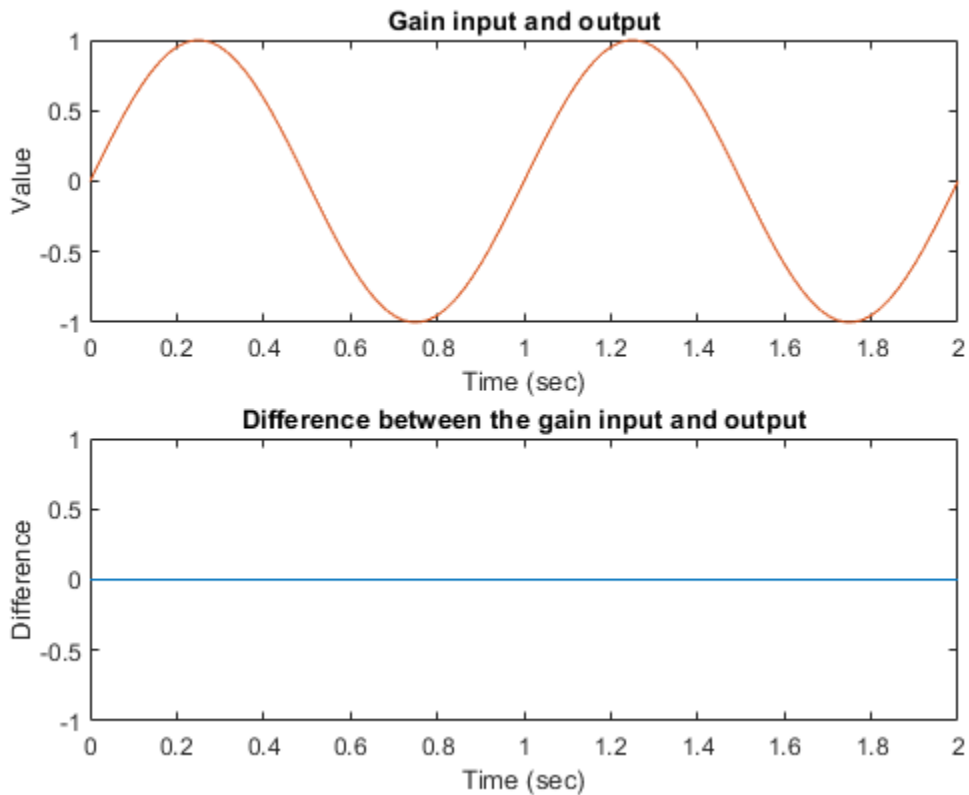
Execute and plot results for the second test case in the same way.

Notice that once the limited output has diverged from the input it can only recover at the maximum slew rate. This is why the plot has an unusual kink. Once the input and output match, the two change together.

```
load rising_gain.mat
covSet = setModelParameter(covSet,'CovSaveName','dataObj2');
simOut2 = sim(covSet);
dataObj2

subplot(211)
plot(simOut2.tout,simOut2.yout(:,1),simOut2.tout,simOut2.yout(:,4))
xlabel('Time (sec)'), ylabel('Value'),
title('Gain input and output');
subplot(212)
plot(simOut2.tout,simOut2.yout(:,1)-simOut2.yout(:,4))
xlabel('Time (sec)'), ylabel('Difference'),
title('Difference between the gain input and output');


dataObj2 = ... cvdata
           version: (R2021b)
                id: 1649
              type: TEST_DATA
              test: cvtest object
            rootID: 1535
```

```
         checksum: [1x1 struct]
        modelinfo: [1x1 struct]
        startTime: 01-Sep-2021 12:03:51
         stopTime: 01-Sep-2021 12:03:51
intervalStartTime: 0
 intervalStopTime: 0
simulationStartTime: 0
 simulationStopTime: 2
           filter:
          simMode: Normal
```
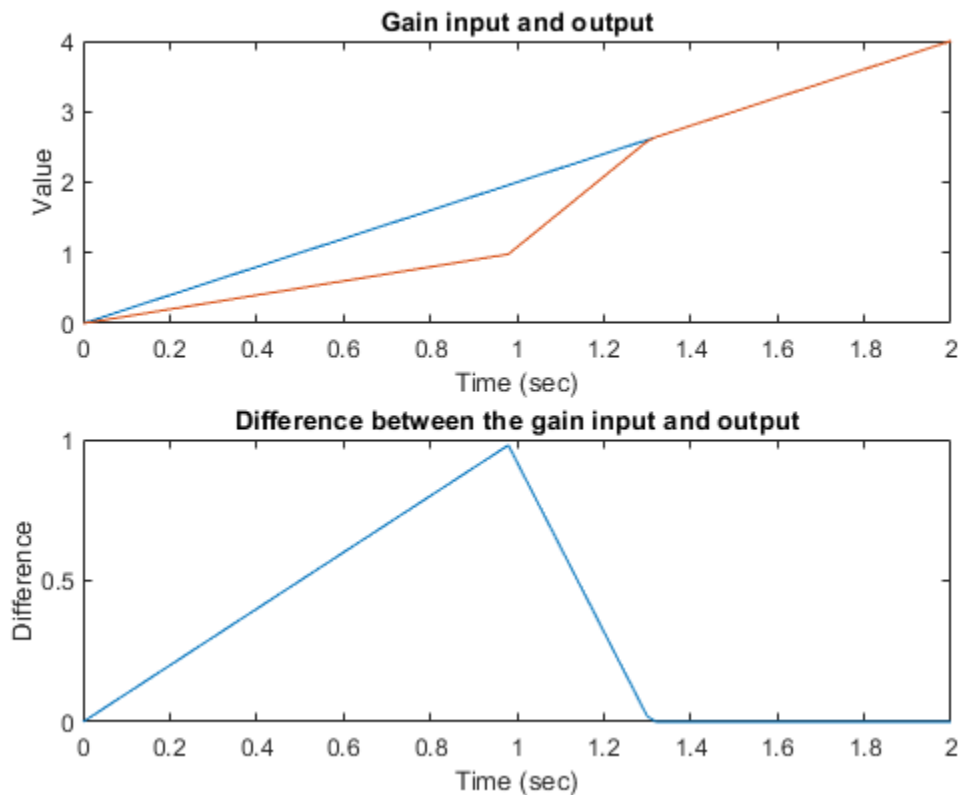

Execute and plot results for the third test case.

```
load falling_gain.mat
covSet = setModelParameter(covSet,'CovSaveName','dataObj3');
simOut3 = sim(covSet);
dataObj3

subplot(211)
plot(simOut3.tout,simOut3.yout(:,1),simOut3.tout,simOut3.yout(:,4))
xlabel('Time (sec)'), ylabel('Value'),
title('Gain input and output');
subplot(212)
plot(simOut3.tout,simOut3.yout(:,1)-simOut3.yout(:,4))
xlabel('Time (sec)'), ylabel('Difference'),
title('Difference between the gain input and output');
```

```
dataObj3 = ... cvdata
              version: (R2021b)
                   id: 1764
                 type: TEST_DATA
                 test: cvtest object
               rootID: 1535
             checksum: [1x1 struct]
            modelinfo: [1x1 struct]
            startTime: 01-Sep-2021 12:03:53
             stopTime: 01-Sep-2021 12:03:53
    intervalStartTime: 0
     intervalStopTime: 0
  simulationStartTime: 0
   simulationStopTime: 2
               filter:
              simMode: Normal
```
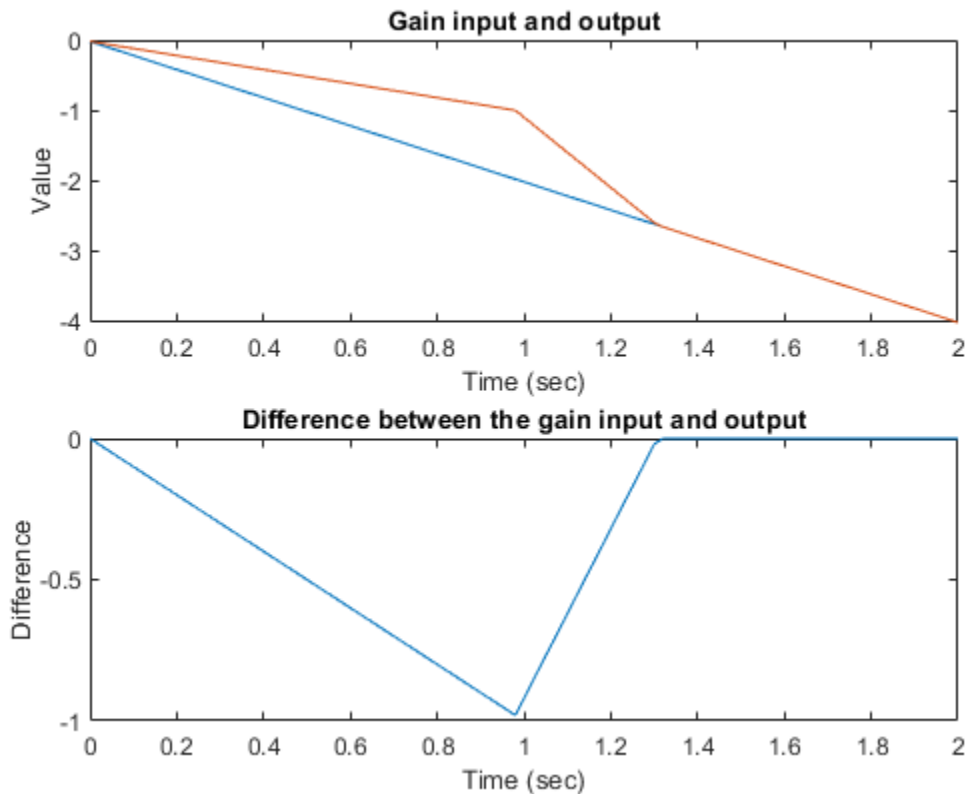

Generating a Coverage Report

Assuming that all the tests have passed, produce a combined report from all test cases to verify the achievement of 100% coverage. Coverage percentages for each test are displayed under the heading "Model Hierarchy." Although none of the tests individually achieved 100% coverage, in aggregate, they achieve complete coverage.

```
cvhtml('combined_ratelim',dataObj1,dataObj2,dataObj3);
```

Saving Coverage Data

Save the collected coverage data in the file "ratelim_testdata.cvt" by using `cvsave`.

```
cvsave('ratelim_testdata',dataObj1,dataObj2,dataObj3);
```

Close the model and exit the coverage environment

```
close_system('slvnvdemo_ratelim_harness',0);
clear dataObj*
```

Loading Coverage Data

Restore saved coverage tests from the file "ratelim_testdata.cvt" **after** opening the model by using `cvload`. The data and tests are retrieved in a cell array.

```
open_system('slvnvdemo_ratelim_harness');
[SavedTests,SavedData] = cvload('ratelim_testdata')

SavedTests =

  1x3 cell array

    {1x1 cvtest}    {1x1 cvtest}    {1x1 cvtest}

SavedData =

  1x3 cell array

    {1x1 cvdata}    {1x1 cvdata}    {1x1 cvdata}
```
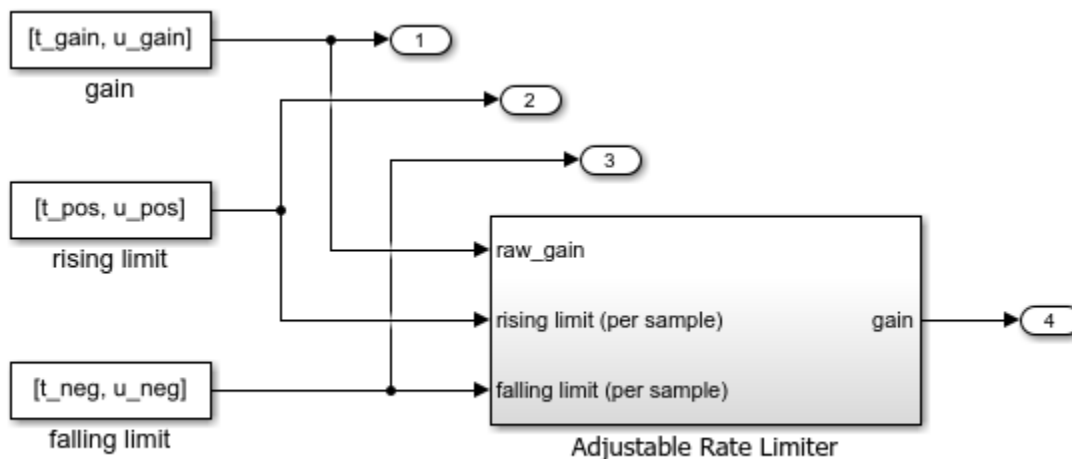

Copyright 1990-2006 The MathWorks Inc.

Manipulating Coverage Data Objects

Manipulate cvdata objects using the overloaded operators: +, -, and *. The * operator is used to find the intersection of two coverage data objects, which results in another cvdata object. For example, the following command produces an HTML report of the common coverage from all three tests.

```
common = SavedData{1} * SavedData{2} * SavedData{3}
cvhtml('intersection',common)


common = ... cvdata
              version: (R2021b)
                   id: 0
                 type: DERIVED_DATA
                 test: []
               rootID: 1882
             checksum: [1x1 struct]
            modelinfo: [1x1 struct]
            startTime: 01-Sep-2021 12:03:48
             stopTime: 01-Sep-2021 12:03:53
    intervalStartTime: 0
     intervalStopTime: 0
               filter:
              simMode: Normal
```

Extracting Information from Coverage Data Objects

Retrieve decision coverage information from a block path or block handle by using `decisioninfo`. The output is a vector with the achieved and total outcomes for a single model object, respectively.

```
cov = decisioninfo(SavedData{1} + SavedData{2} + SavedData{3}, ...
                   'slvnvdemo_ratelim_harness/Adjustable Rate Limiter')


cov =

     6     6
```

Use the retrieved coverage information to access the percentage coverage.

```
percentCov = 100 * (cov(1)/cov(2))


percentCov =

   100
```

When two output arguments are used, `decisioninfo` returns a structure that captures the decisions and outcomes within the Simulink block or Stateflow® object.

```
[blockCov,desc] = decisioninfo(common, ...
        'slvnvdemo_ratelim_harness/Adjustable Rate Limiter/Delta sign')
descDecision = desc.decision
outcome1 = desc.decision.outcome(1)
outcome2 = desc.decision.outcome(2)


blockCov =

     0     2
```

```
desc =

  struct with fields:

           isFiltered: 0
    justifiedCoverage: 0
          isJustified: 0
       filterRationale: ''
             decision: [1x1 struct]


descDecision =

  struct with fields:

                 text: 'Switch trigger'
      filterRationale: ''
            isFiltered: 0
           isJustified: 0
              outcome: [1x2 struct]


outcome1 =

  struct with fields:

                 text: 'false (out = in3)'
      executionCount: 0
           executedIn: []
            isFiltered: 0
           isJustified: 0
       filterRationale: ''


outcome2 =

  struct with fields:

                 text: 'true (out = in1)'
      executionCount: 0
           executedIn: []
            isFiltered: 0
           isJustified: 0
       filterRationale: ''
```

Extracting Detailed Information from Coverage Data

This example shows how coverage utility commands can be used to extract information for an individual subsystem, block, or Stateflow® object from cvdata objects.

Example Model

This example illustrates command line access of coverage data for a small model that contains aspects of various supported coverage metrics.

Use the following commands to open the model 'slvnvdemo_cv_small_controller' and its subsystem 'Gain.'

```
open_system('slvnvdemo_cv_small_controller');
open_system('slvnvdemo_cv_small_controller/Gain');
```
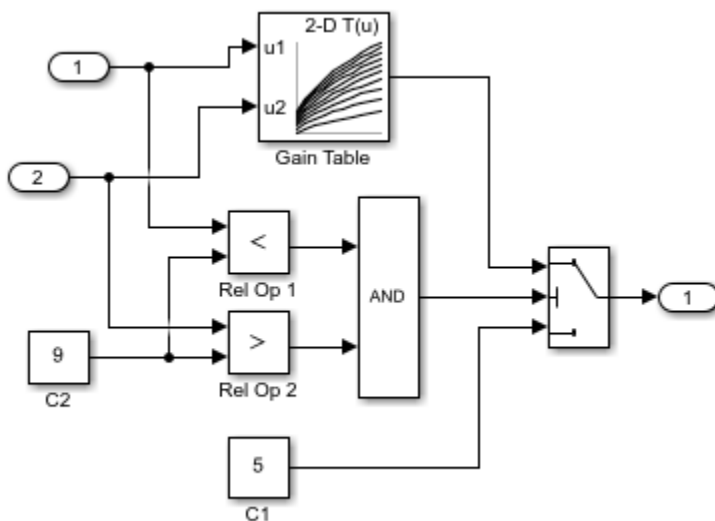

Generate Coverage Data and an HTML Report

Simulate the model using sim. Use a Simulink.SimulationInput object to capture coverage settings and use it as an input to sim. After the simulation, coverage data will be stored in a cvdata object.

```
simIn = Simulink.SimulationInput('slvnvdemo_cv_small_controller');
simIn = simIn.setModelParameter('CovEnable','on');
simIn = simIn.setModelParameter('CovMetricStructuralLevel','MCDC');
simIn = simIn.setModelParameter('CovSaveSingleToWorkspaceVar','on');
simIn = simIn.setModelParameter('CovSaveName','covData');
simIn = simIn.setModelParameter('CovScope','EntireSystem');
simIn = simIn.setModelParameter('CovMetricLookupTable','on');
simIn = simIn.setModelParameter('CovMetricSignalRange','on');
simOut = sim(simIn);
```

Process the coverage data returned from a cvsim command with the report generation command cvhtml. The resulting report is a convenient representation of model coverage for the entire model.

```
cvhtml('tempfile.html',covData);
```

The coverage data is also available in the simulation output object.

```
simOut
```

```
simOut =

  Simulink.SimulationOutput:
               covData: [1x1 cvdata]
                  tout: [59x1 double]
                  yout: [59x1 double]

     SimulationMetadata: [1x1 Simulink.SimulationMetadata]
          ErrorMessage: [0x0 char]
```

Extract Decision Coverage Information

Use the `decisioninfo` command to extract decision coverage information for individual Simulink blocks or Stateflow objects.

The following command extracts a coverage array for the entire model. The first element is the number of coverage objective outcomes satisfied for the model; the second element is the total number of coverage objective outcomes for the model.

```
cov = decisioninfo(covData,'slvnvdemo_cv_small_controller')
percent = 100*cov(1)/cov(2)
```

```
cov =

     4     6
```

```
percent =

   66.6667
```

Retrieve coverage information for the 'Saturation' block using the full path to that block. Provide a second return argument for textual descriptions of the coverage objective outcomes within that block.

```
[blkCov, description] = decisioninfo(covData,'slvnvdemo_cv_small_controller/Saturation')

decision1 = description.decision(1).text
out_1a = description.decision(1).outcome(1).text
count_1a = description.decision(1).outcome(1).executionCount
out_1b = description.decision(1).outcome(2).text
count_1b = description.decision(1).outcome(2).executionCount
```

```
blkCov =

     3     4
```

```
description =
```

```
    struct with fields:

              isFiltered: 0
       justifiedCoverage: 0
             isJustified: 0
          filterRationale: ''
                 decision: [1x2 struct]


decision1 =

    'U >= LL'


out_1a =

    'false'


count_1a =

     0


out_1b =

    'true'


count_1b =

     7
```

Quantitative coverage information is available for every outcome in the hierarchy that contains or has coverage objective outcomes. Textual descriptions are generated only for objects that have coverage objective outcomes themselves. For example, invoke `decisioninfo` for the virtual subsystem Gain, and the description return value is empty.

```
[blkCov, description] = decisioninfo(covData,'slvnvdemo_cv_small_controller/Gain')


blkCov =

     1     2


description =

  struct with fields:

            isFiltered: 0
     justifiedCoverage: 0
           isJustified: 0
        filterRationale: ''
```

In some cases an object has internal coverage objectives but also contains descendants with additional coverage objectives. Coverage information normally includes all the descendants unless a third argument for ignoring descendants is set to 1.

```
subsysOnlycov = decisioninfo(covData,'slvnvdemo_cv_small_controller/Gain',1)
```

```
subsysOnlycov =

     []
```

The `decisioninfo` command also works with block handles, Stateflow IDs, and Stateflow API objects.

```
blkHandle = get_param('slvnvdemo_cv_small_controller/Saturation','Handle')
blkCov = decisioninfo(covData,blkHandle)
```

```
blkHandle =

   31.0085
```

```
blkCov =

     3      4
```

If an object has no decision coverage, the command returns empty outputs.

```
missingBlkCov = decisioninfo(covData,'slvnvdemo_cv_small_controller/Sine1')
```

```
missingBlkCov =

     []
```

Extract Condition Coverage Information

Condition coverage indicates if the logical inputs to Boolean expressions have been evaluated to both true and false. In Simulink, conditions are the inputs to logical operations.

The `conditioninfo` command for extracting condition coverage information is very similar to the `decisioninfo` command. It normally returns information about an object and all its descendants, but can take a third argument that indicates if descendants should be ignored. It can also return a second output containing descriptions of each condition.

```
cov = conditioninfo(covData,'slvnvdemo_cv_small_controller/Gain/Logic')
[cov, desc] = conditioninfo(covData,'slvnvdemo_cv_small_controller/Gain/Logic');
desc.condition(1)
desc.condition(2)
```

```
cov =

     2      4
```

```
ans =

  struct with fields:

          isFiltered: 0
          isJustified: 0
       filterRationale: ''
                  text: 'port1'
              trueCnts: 59
             falseCnts: 0
     trueOutcomeFilter: [1x1 struct]
    falseOutcomeFilter: [1x1 struct]
         trueExecutedIn: []
        falseExecutedIn: []


ans =

  struct with fields:

          isFiltered: 0
          isJustified: 0
       filterRationale: ''
                  text: 'port2'
              trueCnts: 0
             falseCnts: 59
     trueOutcomeFilter: [1x1 struct]
    falseOutcomeFilter: [1x1 struct]
         trueExecutedIn: []
        falseExecutedIn: []
```

Extract Modified Condition/Decision Coverage Information

Modified Condition/Decision Coverage (MCDC) is satisfied for a condition within a Boolean expression if there are two evaluations of the expression, representing an *independence pair*, which illustrate that the value of the condition independently affects the outcome of the entire expression. That is to say, for these evaluations, toggling the value of the condition would cause the expression outcome to toggle as well.

In this example, the logical AND block is analyzed for MCDC and this information can be extracted using the `mcdcinfo` command. This command uses the same syntax as `conditioninfo` and `decisioninfo` commands.

```
[cov, desc] = mcdcinfo(covData,'slvnvdemo_cv_small_controller/Gain/Logic')
desc.condition(1)
desc.condition(2)


cov =

     0     2


desc =
```

```
  struct with fields:

              text: 'Output'
         condition: [1x2 struct]
        isFiltered: 0
    filterRationale: ''
  justifiedCoverage: 0


ans =

  struct with fields:

              text: 'port1'
          achieved: 0
          trueRslt: '(TT)'
         falseRslt: '(FT)'
        isFiltered: 0
       isJustified: 0
    filterRationale: ''
     trueExecutedIn: []
    falseExecutedIn: []


ans =

  struct with fields:

              text: 'port2'
          achieved: 0
          trueRslt: '(TT)'
         falseRslt: 'TF'
        isFiltered: 0
       isJustified: 0
    filterRationale: ''
     trueExecutedIn: []
    falseExecutedIn: []
```

Extract Lookup Table Coverage Information

Lookup table coverage records the frequency that lookup occurs for each interpolation interval. Valid intervals for coverage purposes also include values less than the smallest breakpoint and values greater than the largest breakpoint. For consistency with the other commands, this information is returned as a pair of counts with the number of intervals that executed and the total number of intervals.

A second output argument causes `tableinfo` to return the execution counts for all interpolation intervals. If the table has M-by-N output values, execution counts are returned in an M+1-by-N+1 matrix.

A third output argument causes `tableinfo` to return the counts where the input was exactly equal to the breakpoint. This is returned in a cell array of vectors, one for each dimension in the table.

```
[cov,execCnts,brkEq] = tableinfo(covData, 'slvnvdemo_cv_small_controller/Gain/Gain Table')


cov =
```

```
    23    121


execCnts =

     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     2    12    14    10     2     0     0     0
     0     0     4    12     0     0     0    12     0     0     0
     0     0    22     0     0     0     0     0    12     0     0
     0     0    21     0     0     0     0     0    59     0     0
     0     0    21     0     0     0     0     0    29     0     0
     0     0     7    28     0     0     0    28     6     0     0
     0     0     0     4    22    18    23     5     0     0     0
     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0


brkEq =

  1x2 cell array

    {10x1 double}    {10x1 double}
```

Extract Signal Range Information

The signal range metric records the smallest and largest value of Simulink block outputs and Stateflow data objects. The `sigrangeinfo` command returns two return arguments for the minimum and maximum values, respectively.

The `sigrangeinfo` command works only for leaf blocks that perform a computation; otherwise the command returns empty arguments.

```
[sigMin, sigMax] = sigrangeinfo(covData,'slvnvdemo_cv_small_controller/Gain/Gain Table')   % Leaf
[sigMin, sigMax] = sigrangeinfo(covData,'slvnvdemo_cv_small_controller/Gain')              % Nonle


sigMin =

    3.3656


sigMax =

    7.6120


sigMin =

    []


sigMax =
```
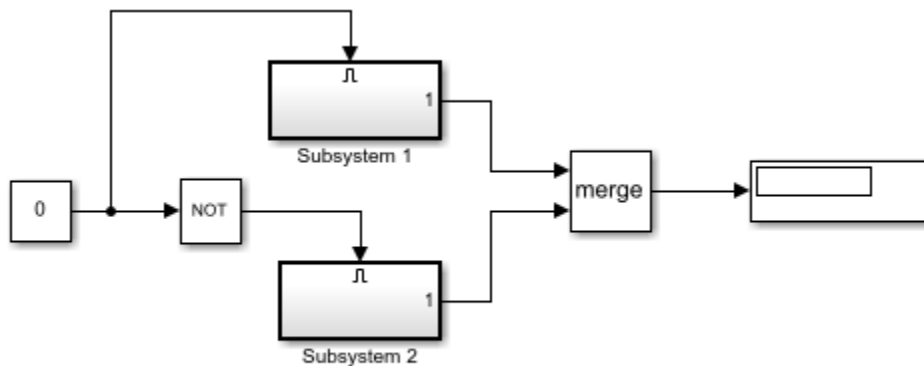
[ ]

Operations on Coverage Data

This example shows how to use the overloaded operators +, *, and - to combine coverage results into a union, intersection, or set difference of results.

Example Model

Open a simple model with two mutually-exclusive enabled subsystems.

```
open_system('slvnvdemo_cv_mutual_exclusion')
```


Copyright 1990-2019 The MathWorks Inc.

Use the commands `cvtest` and `cvsim` to start simulation. Initially, the value of the Constant block is 0, which forces Subsystem 2 to execute.

```
test1 = cvtest('slvnvdemo_cv_mutual_exclusion');
data1 = cvsim(test1)
```

```
data1 = ... cvdata
              version: (R2021b)
                   id: 698
                 type: TEST_DATA
                 test: cvtest object
               rootID: 700
             checksum: [1x1 struct]
            modelinfo: [1x1 struct]
            startTime: 01-Sep-2021 12:09:41
             stopTime: 01-Sep-2021 12:09:41
    intervalStartTime: 0
     intervalStopTime: 0
  simulationStartTime: 0
   simulationStopTime: 10
               filter:
              simMode: Normal
```

The following commands change the value of the Constant block to 1 before running the second simulation. This forces Subsystem 1 to execute.

```
set_param('slvnvdemo_cv_mutual_exclusion/Constant','Value','1');
test2 = cvtest('slvnvdemo_cv_mutual_exclusion');
data2 = cvsim(test2)


data2 = ... cvdata
              version: (R2021b)
                   id: 753
                 type: TEST_DATA
                 test: cvtest object
               rootID: 700
             checksum: [1x1 struct]
            modelinfo: [1x1 struct]
            startTime: 01-Sep-2021 12:09:42
             stopTime: 01-Sep-2021 12:09:42
    intervalStartTime: 0
     intervalStopTime: 0
  simulationStartTime: 0
   simulationStopTime: 10
               filter:
              simMode: Normal
```

We use the `decisioninfo` command to extract the decision coverage from each test and list it as a percentage.

Note: While both tests have 50% decision coverage, whether or not they cover the same 50% is unknown.

```
cov1 = decisioninfo(data1,'slvnvdemo_cv_mutual_exclusion');
percent1 = 100*(cov1(1)/cov1(2))

cov2 = decisioninfo(data2,'slvnvdemo_cv_mutual_exclusion');
percent2 = 100*(cov2(1)/cov2(2))


percent1 =

    50


percent2 =

    50
```

Finding the Union of Coverage

Use the + operator to derive a third cvdata object representing the union of data1 and data2 cvdata objects.

Note: New cvdata objects created from combinations of other simulation results are marked with the type property set as DERIVED_DATA.

```
dataUnion = data1 + data2


dataUnion = ... cvdata
```

```
            version: (R2021b)
                 id: 0
               type: DERIVED_DATA
               test: []
             rootID: 700
           checksum: [1x1 struct]
          modelinfo: [1x1 struct]
          startTime: 01-Sep-2021 12:09:41
           stopTime: 01-Sep-2021 12:09:42
  intervalStartTime: 0
   intervalStopTime: 0
             filter:
            simMode: Normal
```

Notice that the union of the coverage is 100% because there is no overlap in the coverage between the two sets.

```
covU = decisioninfo(dataUnion,'slvnvdemo_cv_mutual_exclusion');
percentU = 100*(covU(1)/covU(2))
```

```
percentU =

   100
```

Finding the Intersection of Coverage

Confirm that the coverage does not overlap between the two tests by intersecting data1 and data2 with the * operator. As expected, there is 0% decision coverage in the intersection.

```
dataIntersection = data1 * data2
```

```
covI = decisioninfo(dataIntersection,'slvnvdemo_cv_mutual_exclusion');
percentI = 100*(covI(1)/covI(2))
```

```
dataIntersection = ... cvdata
            version: (R2021b)
                 id: 0
               type: DERIVED_DATA
               test: []
             rootID: 700
           checksum: [1x1 struct]
          modelinfo: [1x1 struct]
          startTime: 01-Sep-2021 12:09:41
           stopTime: 01-Sep-2021 12:09:42
  intervalStartTime: 0
   intervalStopTime: 0
             filter:
            simMode: Normal
```

```
percentI =

     0
```

Using Derived Coverage Data Objects

Derived cvdata objects can be used in all reporting and analysis commands, and as inputs to subsequent operations. As an example, generate a coverage report from the derived dataIntersection object.

```
cvhtml('intersect_cov', dataIntersection);

% Input to another operation
newUnion = dataUnion + dataIntersection


newUnion = ... cvdata
            version: (R2021b)
                 id: 0
               type: DERIVED_DATA
               test: []
             rootID: 700
           checksum: [1x1 struct]
          modelinfo: [1x1 struct]
          startTime: 01-Sep-2021 12:09:41
           stopTime: 01-Sep-2021 12:09:42
  intervalStartTime: 0
   intervalStopTime: 0
             filter:
            simMode: Normal
```

Coverage Report for slvnvdemo_cv_mutual_exclusion

Table of Contents

Analysis Information

Model Information

Model version	1.14
Author	The MathWorks, Inc.
Last saved	Mon Nov 21 06:43:01 2016

Simulation Optimization Options

Default parameter behavior	tunable
Block reduction	forced off
Conditional branch optimization	on

Coverage Options

Analyzed model	slvnvdemo_cv_mutual_exclusion
Logic block short circuiting	off

Tests

Test#	Started execution	Ended execution
Test 1	23-Dec-2016 17:15:32	23-Dec-2016 17:15:33

Summary

Model Hierarchy/Complexity		Test 1	
		Decision	Execution
1. slvnvdemo_cv_mutual_exclusion	5	0%	50%
2. . . . Subsystem 1	2	0%	0%
3. . . . Subsystem 2	2	0%	0%

Computing the Coverage (set) Difference

The - operator is used to form a cvdata object that represents the set difference between left and right operands. The result of the operation contains the coverage points that are satisfied in the left operand but not satisfied in the right operand. This operation is useful for determining how much additional coverage is attributed to a particular test.

In the following example, the difference between the union of the first and second test coverage and the first test coverage should indicate how much additional coverage the second test provided. As already shown, since none of the decision coverage points overlapped, the new decision coverage from test 2 is 50%.

```
newCov2 = dataUnion - data1

covN = decisioninfo(newCov2,'slvnvdemo_cv_mutual_exclusion');
percentN = 100*(covN(1)/covN(2))


newCov2 = ... cvdata
            version: (R2021b)
                 id: 0
               type: DERIVED_DATA
               test: []
             rootID: 700
           checksum: [1x1 struct]
          modelinfo: [1x1 struct]
          startTime: 01-Sep-2021 12:09:41
           stopTime: 01-Sep-2021 12:09:42
  intervalStartTime: 0
   intervalStopTime: 0
             filter:
            simMode: Normal


percentN =

    50
```

# Record Coverage in Parallel Simulations by Using Parsim

This example shows how to record coverage in multiple parallel Simulink® simulations corresponding to different test cases by using SimulationInput objects and the `parsim` command. If Parallel Computing Toolbox is installed on your system, the `parsim` command runs simulations in parallel. Otherwise, the simulations are run in serial.

Model Overview

The `slvnvdemo_powerwindow_parsim` model contains a power window controller and a low-order plant model. The component `slvnvdemo_powerwindow_parsim/power_window_control_system/control` is a Model block that references the model `slvnvdemo_powerwindow_controller`, which implements the controller with a Stateflow® chart.

```
mdl                = 'slvnvdemo_powerwindow_parsim';
isModelOpen        = bdIsLoaded(mdl);
open_system(mdl);
```


**Simulink Coverage
Power Window Controller Hybrid System Model**
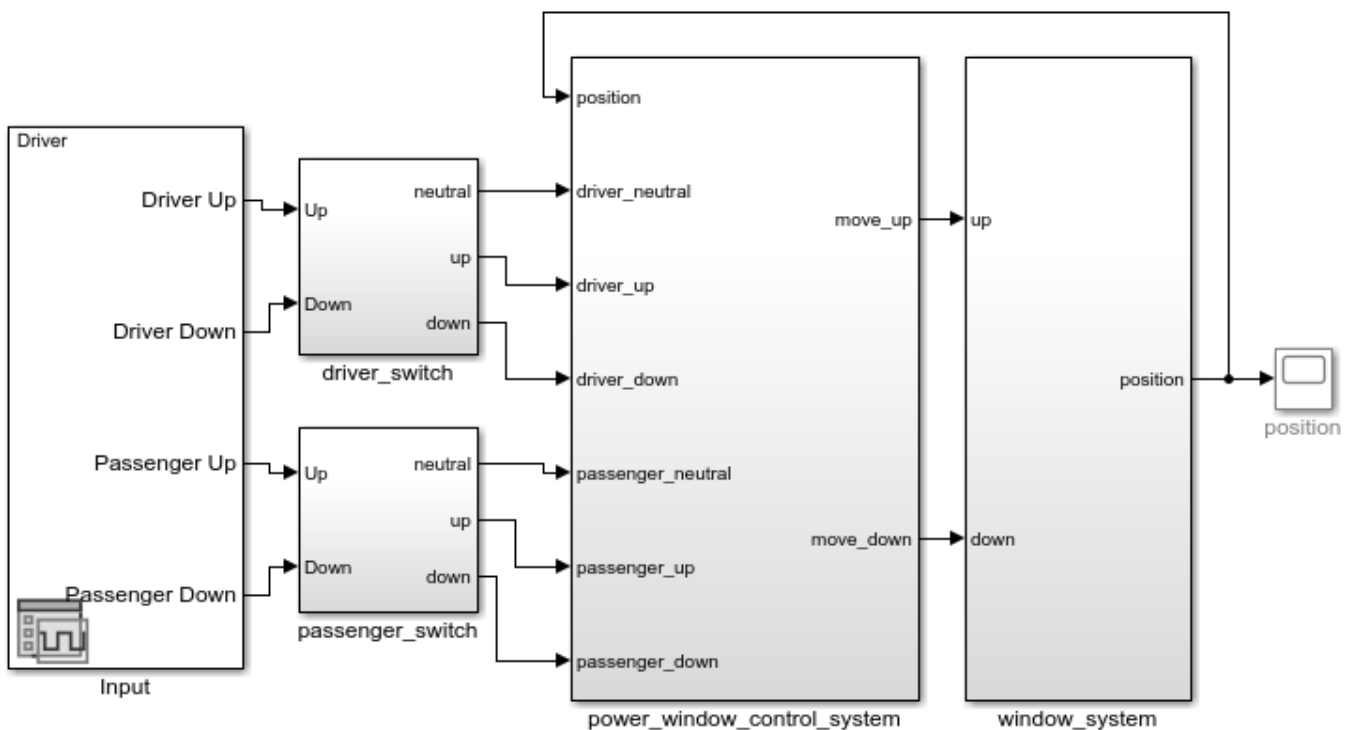
Copyright 1990-2018 The MathWorks, Inc.

Set Up Data for Multiple Simulations

Determine the number of test cases in the Signal Editor block by using the `NumberOfScenarios` parameter. The number of test cases determines the number of iterations to run.

```
sigEditBlk = [mdl '/Input'];
numCases   = str2double(get_param(sigEditBlk,'NumberOfScenarios'));
```

Create an array of `Simulink.SimulationInput` objects to define the set of simulations to run. Each SimulationInput object corresponds to one simulation and is stored in array `simIn`. For each simulation, set these parameters:

- `ActiveScenario` to indicate which scenario of the Signal Editor block to execute
- `CovEnable` to turn on coverage analysis
- `CovSaveSingleToWokspaceVar` to save the coverage results to a workspace variable
- `CovSaveName` to specify the name of the variable.

```
for idx = numCases:-1:1
    simIn(idx) = Simulink.SimulationInput(mdl);
    simIn(idx) = setBlockParameter(simIn(idx), sigEditBlk, 'ActiveScenario', idx);
    simIn(idx) = setModelParameter(simIn(idx), 'CovEnable', 'on');
    simIn(idx) = setModelParameter(simIn(idx), 'CovSaveSingleToWorkspaceVar', 'on');
    simIn(idx) = setModelParameter(simIn(idx), 'CovSaveName', 'covdata');
end
```

Run Simulations in Parallel by Using Parsim

Use the `parsim` function to execute the simulations in parallel. Pass the array of SimulationInput objects, `simIn`, into the `parsim` function as the first argument. Set the `ShowProgress` option to `on` to display the progress of the simulations in the MATLAB Command Window. The output from the `parsim` command is `simOut`, an array of `Simulink.SimulationOutput` objects.

```
simOut = parsim(simIn, 'ShowProgress', 'on');
```

```
[01-Sep-2021 12:09:48] Checking for availability of parallel pool...
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 12).
[01-Sep-2021 12:13:47] Starting Simulink on parallel workers...
[01-Sep-2021 12:14:33] Configuring simulation cache folder on parallel workers...
[01-Sep-2021 12:14:35] Loading model on parallel workers...
[01-Sep-2021 12:14:45] Running simulations...
[01-Sep-2021 12:16:22] Completed 1 of 2 simulation runs
[01-Sep-2021 12:16:22] Completed 2 of 2 simulation runs
[01-Sep-2021 12:16:22] Cleaning up parallel workers...
```

Each `Simulink.SimulationInput` object contains logged coverage results stored as `cv.cvdatagroup objects`. These coverage results are stored in a field named `covdata`, as previously specified by the `CovSaveName` parameter. Using `parsim` to run multiple simulations means that errors are captured so that subsequent simulations can continue to run. Any errors are recorded in the `ErrorMessage` property of the SimulationOutput object.

`covdata` references a file containing the coverage results. The coverage data from the referenced file is automatically loaded into memory when `covdata` is used by a coverage function.

```
simOut(1).covdata
```

```
ans = ... cvdata
            file: C:\TEMP\Bdoc21b_1757077_3096\ib2EDA31\9\tpf9c986ce\ex16619798\slcov_output\s
            date: 01-Sep-2021 12:16:20
```
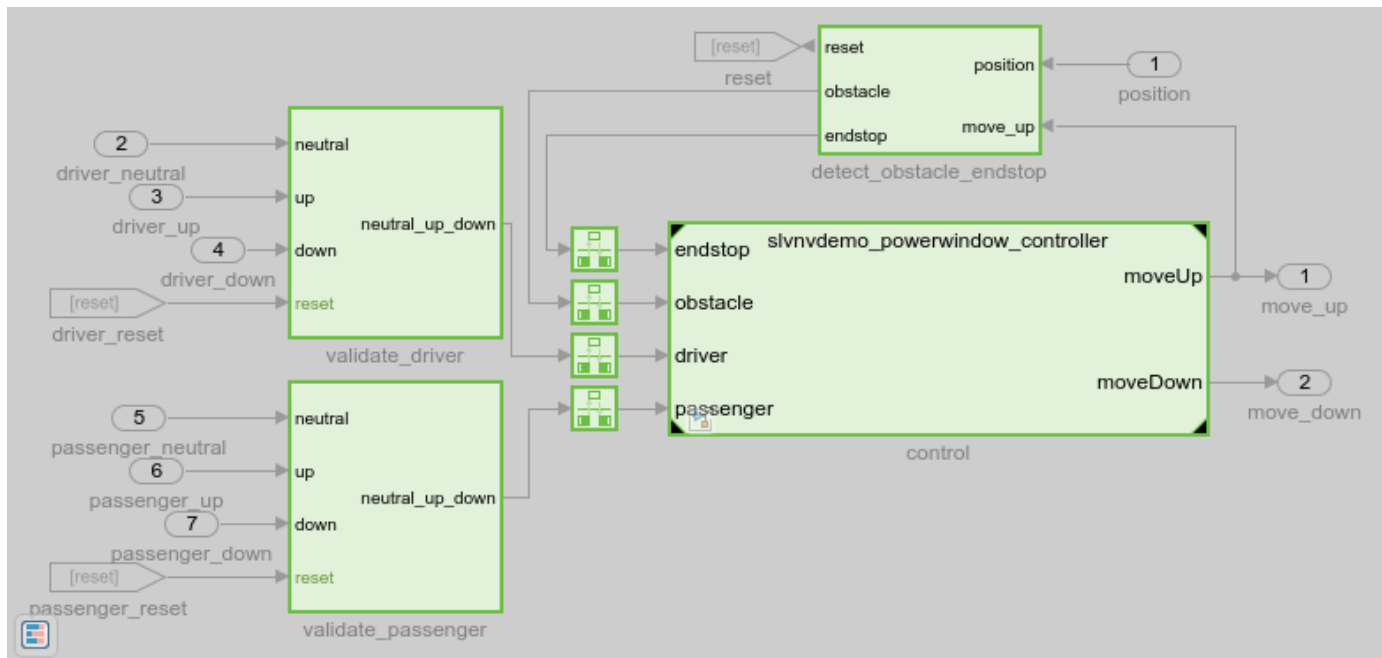
Compute Cumulative Coverage

Obtain the coverage data from each element of `simOut` and cumulate the results.

```
coverageData = simOut(1).covdata;
for i = 2 : numCases
    coverageData = coverageData + simOut(i).covdata;
end
```

View the cumulative coverage results on the model by using coverage highlighting.

```
cvmodelview(coverageData);
open_system('slvnvdemo_powerwindow_parsim/power_window_control_system');
```


Generate a cumulative coverage report.

```
cvhtml('cummulative_cov_report.html', coverageData);
```

Filter Coverage Results Using a Script

This example shows how to programmatically filter objects and outcomes from coverage results.

Open the Model and Enable Coverage Analysis

First, load the model into memory.

```
modelName = 'slvnvdemo_covfilt';
load_system(modelName);
```

Configure the coverage settings for the model by using a `Simulink.SimulationInput` object.

```
simIn = Simulink.SimulationInput(modelName);
simIn = simIn.setModelParameter('CovEnable','on');
simIn = simIn.setModelParameter('CovMetricStructuralLevel','MCDC');
simIn = simIn.setModelParameter('StopTime','20');
simIn = simIn.setModelParameter('CovSaveSingleToWorkspaceVar','on');
simIn = simIn.setModelParameter('CovSaveName','covData');
```

For a list of coverage parameters, see "Coverage Settings".

Simulate the model using the `SimulationInput` object as the input.

```
simOut = sim(simIn);
```

View Decision Coverage Results

View the coverage results before applying a filter. You can access the decision coverage results using `decisioninfo`.

```
saturationInitial = decisioninfo(covData,'slvnvdemo_covfilt/Saturation');
percentSaturationCov = 100 * saturationInitial(1)/saturationInitial(2)


percentSaturationCov =

    50

```

The Saturation block has 50% decision coverage. If you do not intend for this block to be satisfied, you can filter a missing objective outcome so that it is no longer reported as missing coverage. First, you need a selector for the unsatisfied objective outcome that you want to filter.

Create a Selector

You can directly create a selector using the appropriate constructor. In this case, you would use `slcoverage.MetricSelector`.

Because the objective being justified is a decision outcome, the first input to the metric selector constructor is `slcoverage.MetricSelectorType.DecisionOutcome`. The second input is the block handle. The last two are the index of the objective to justify and the index of the outcome of that objective, respectively. Because the `input > lower limit` decision objective is the first objective for the Saturation block, its objective index is 1. Because the `false` outcome of this objective is the first outcome, its outcome index is also 1.

```
metricSel = slcoverage.MetricSelector(slcoverage.MetricSelectorType.DecisionOutcome,...
                        'slvnvdemo_covfilt/Saturation',1,1)
```

```
metricSel =

  MetricSelector with properties:

      ObjectiveIndex: 1
        OutcomeIndex: 1
         Description: 'N/A'
                Type: DecisionOutcome
                  Id: 'slvnvdemo_covfilt:5'
     ConstructorCode: 'slcoverage.MetricSelector(slcoverage.MetricSelectorType.DecisionOutcome, 's
```

You can also use `slcoverage.Selector.allSelectors` to see the available selectors for the Saturation block.

`saturationAllSels = slcoverage.Selector.allSelectors('slvnvdemo_covfilt/Saturation')`

```
saturationAllSels =

  1x6 heterogeneous Selector (BlockSelector, MetricSelector) array with properties:

    Description
    Type
    Id
    ConstructorCode
```

You can also see the objective and outcome indices by using the `allSelectors` method. Use the `Description` name-value pair to search for `F`.

`falseSelectors = slcoverage.Selector.allSelectors('slvnvdemo_covfilt/Saturation',...`
`              'Description','F')`

```
falseSelectors =

  1x2 MetricSelector array with properties:

    ObjectiveIndex
    OutcomeIndex
    Description
    Type
    Id
    ConstructorCode
```

There are two false case selectors in the Saturation block. The first selector is `F outcome of input > lower limit`.

`falseSel = falseSelectors(1)`

```
falseSel =

  MetricSelector with properties:

      ObjectiveIndex: 1
```

```
     OutcomeIndex: 1
      Description: 'F outcome of input >= lower limit in Saturate block "Saturation"'
             Type: DecisionOutcome
               Id: 'slvnvdemo_covfilt:5'
  ConstructorCode: 'slcoverage.MetricSelector(slcoverage.MetricSelectorType.DecisionOutcome, '
```

The `falseSel` selector is the same one we constructed manually using `slcoverage.MetricSelector`. The objective and outcome indices are properties of the resulting selector object.

Create a Justification Rule

Create a filter object by using `slcoverage.Filter`. You can set the filter file name and filter description by using the methods `setFilterName` and `setFilterDescription`, respectively.

```
filt = slcoverage.Filter;
setFilterName(filt,'slcoverage_filter');
setFilterDescription(filt,'Example Filter');
```

Create a filter rule by using `slcoverage.FilterRule`. The first input to `FilterRule` is the selector for the block or outcome you want to filter. This can be a selector you create, or one you retrieve from `allSelectors`.

The second input is the rationale for filtering the outcome or block. This is specified as a character array.

The third input is the filter mode you want to use. The two coverage filter modes are justify and exclude. Use justify mode to filter individual coverage objective outcomes such as `F outcome of input > lower limit`. Use exclude mode to filter entire model elements or blocks, which means that the block and its descendants, if applicable, are ignored. In this example, use justify mode to specify that you want to filter a specific outcome.

```
rule = slcoverage.FilterRule(metricSel,'rate > 0',slcoverage.FilterMode.Justify);
```

Add the rule to the filter using `addRule`.

```
filt.addRule(rule);
```

Save the filter to a filter file using the `save` method. Then apply the filter file to the `cvdata` object by assigning the `filter` property to the new filter file.

```
filt.save('covfilter');
covData.filter = 'covfilter';
```

Re-generate the coverage results for the Saturation block using the filtered `cvdata` object.

```
filteredSaturation = decisioninfo(covData,'slvnvdemo_covfilt/Saturation');
percentSaturationFilt = 100 * filteredSaturation(1)/filteredSaturation(2)
```

```
percentSaturationFilt =

    75
```

Decision coverage for the Saturation block is now 75%.

Justify an MCDC Objective in a Stateflow® Chart

You can apply the same workflow to justify a specific Stateflow action. In this example, we want to justify the `tick` MCDC objective that is part of the `after(4, tick)` transition.

First, get the Stateflow root object by using `sfroot` (Stateflow).

```
chartID = sfroot;
```

Get the `'after(4, tick)'` transition ID by using the `find` (Stateflow) method. You can use `find` to search for transitions by using the `'-isa'` flag with `'Stateflow.Transition'`. You can further specify the exact transition by using searching for the label string using additional inputs.

```
transID = chartID.find('-isa','Stateflow.Transition','LabelString','after(4, tick)');
```

Get the Simulink ID of the chart by using `Simulink.ID.getSID`.

```
transSID = Simulink.ID.getSID(transID);
```

Get the selector for the MCDC objective outcome that we want to filter by using `allSelectors`. Pass the Simulink ID of the Stateflow transition as the first input. Because we want to justify a `tick` outcome, search for `"tick"` in the description.

```
sfSelectors = slcoverage.Selector.allSelectors(transSID,'Description','"tick"')


sfSelectors =

  1x3 MetricSelector array with properties:

    ObjectiveIndex
    OutcomeIndex
    Description
    Type
    Id
    ConstructorCode
```

`allSelectors` returns three possible selectors. The transition we want to filter is the third selector returned.

```
sfSel = sfSelectors(3)


sfSel =

  MetricSelector with properties:

      ObjectiveIndex: 1
        OutcomeIndex: 1
         Description: 'Condition 1, "tick" outcome of Transition trigger expression in Transition
                Type: MCDCOutcome
                  Id: 'slvnvdemo_covfilt:6:5'
     ConstructorCode: 'slcoverage.MetricSelector(slcoverage.MetricSelectorType.MCDCOutcome, 'slvn
```

Create the rule, add it to the filter, and save it. The filter file is already applied to the `cvdata` object.

```
rule2 = slcoverage.FilterRule(sfSel,'tick never false');
filt.addRule(rule2);
filt.save('covfilter');
```

For more information about the stateflow programmatic API, see "Overview of the Stateflow API" (Stateflow).

Exclude a Block Using Block Selector

You can filter a block using `slcoverage.BlockSelector`. In this case, we want to exclude the Switchable config subsystem, so we use the `SubsystemAllContent` selector type and the `slcoverage.FilterMode.Exclude` filter mode.

```
subsysSel = slcoverage.BlockSelector(...
                    slcoverage.BlockSelectorType.SubsystemAllContent,...
                    'slvnvdemo_covfilt/Switchable config');
```

Create the filter rule by passing the selector, rationale, and the exclude filter mode as inputs.

```
rule3 = slcoverage.FilterRule(subsysSel,...
                    'Unused configuration',...
                     slcoverage.FilterMode.Exclude);
```

Add the rule to the filter and save it.

```
filt.addRule(rule3);
filt.save('covfilter');
```

Finally, you can view the coverage report by using `cvhtml`. The **Objects Filtered from Coverage Analysis** section shows a summary of the filtered model objects and the rationales. The `'sRT=0'` flag can be used to generate the coverage report but not open the report automatically.

```
cvhtml('filteredCovReport',covData,'-sRT=0');
```

Objects Filtered from Coverage Analysis

Filter slcoverage_filter

File	covfilter.cvf
Description	Example Filter

Filtered Model Object	Rationale
SubSystem block "Switchable config"	Unused configuration
J2. F outcome of input > lower limit in Saturate block "Saturation"	Expected result
J3. Condition 1, "tick" outcome of Transition trigger expression in Transition "after(4, tick)" from "Clipped" to "Full"	Not tested

See Also

slcoverage.BlockSelector | slcoverage.FilterRule | slcoverage.MetricSelector | slcoverage.Selector | allSelectors | cvhtml | decisioninfo | cvdata

More About

- "Programmatically Retrieve Coverage Details from Results" on page 8-4
- "Creating and Using Coverage Filters" on page 7-11
- "Stateflow Programmatic Interface" (Stateflow)

Component Verification

Component Verification

In this section...
"Simulink Coverage Tools for Component Verification" on page 9-2
"Workflow for Component Verification" on page 9-2
"Verify a Component Independently of the Container Model" on page 9-4
"Verify a Model Block in the Context of the Container Model" on page 9-4

Using component verification, you can test a design component in your model with one of these approaches:

- System analysis. Within the context of the model that contains the component, you use systematic simulation of closed-loop controllers to verify components within a control system model. You can then test the control algorithms with your model.

- Component analysis. As standalone components, for a high level of confidence in the component algorithm, verify the component in isolation from the rest of the system.

  Verifying standalone components provides several advantages:

  - You can use the analysis to focus on portions of the design that you cannot test because of the physical limitations of the system being controlled.

  - For open-loop simulations, you can test the plant model without feedback control.

  - You can use this approach when the model is not yet available or when you need to simulate a control system model in accelerated mode for performance reasons.

Simulink Coverage Tools for Component Verification

By isolating a component to verify and by using tools that the Simulink Coverage software provides, you create test cases to expand the scope of the testing for large models. You can:

- Achieve 100% model coverage — If certain model components do not record 100% coverage, the top-level model cannot achieve 100% coverage. By verifying these components individually, you can create test cases that fully specify the component interface, allowing the component to record 100% coverage.

- Debug the component — To verify that each model component satisfies the specified design requirements, you can create test cases that verify that specific components perform as they were designed to perform.

- Test the robustness of the component — To verify that a component handles unexpected inputs and calculations properly, you can create test cases that generate data. Then, test the error-handling capabilities in the component.

Workflow for Component Verification

This graphic illustrates two approaches for component verification.
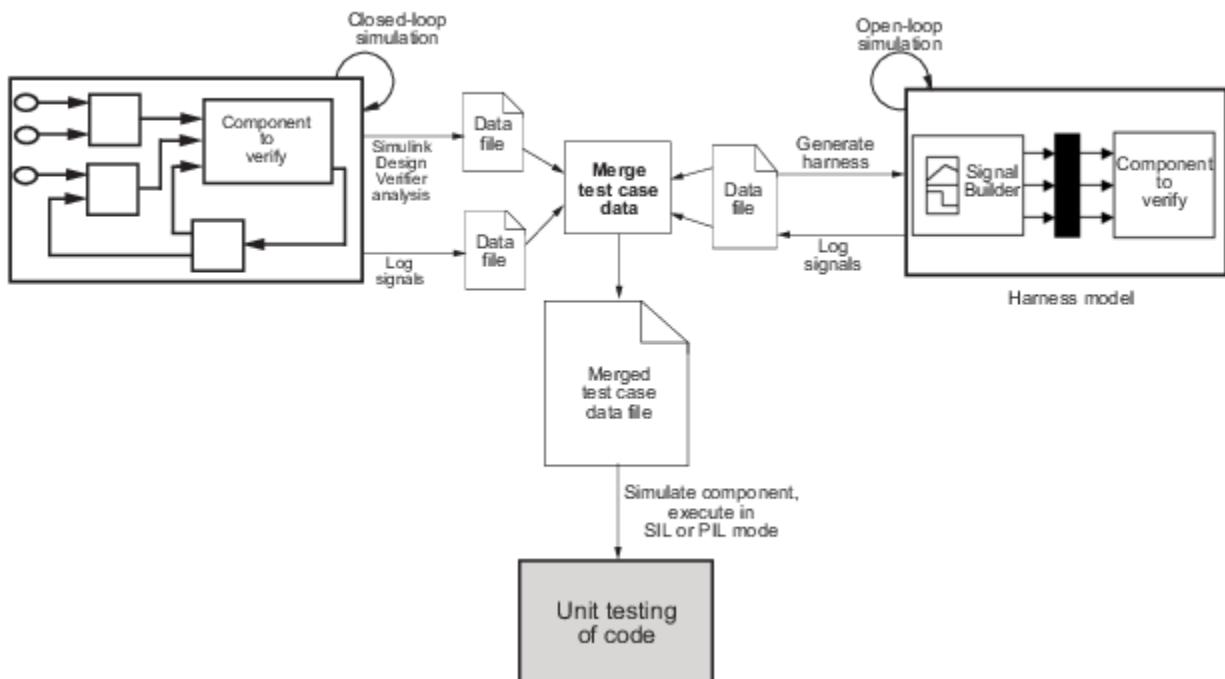
1 Choose your approach for component verification:

- For closed-loop simulations, verify a component within the context of its container model by logging the signals to that component and storing them in a data file. If those signals do not constitute a complete test suite, generate a harness model and add or modify the test cases in the Signal Builder.
- For open-loop simulations, verify a component independently of the container model by extracting the component from its container model and creating a harness model for the extracted component. Add or modify test cases in the Signal Builder and log the signals to the component in the harness model.

2 Prepare component for verification.

3 Create and log test cases. You can also merge the test case data into a single data file.

The data file contains the test case data for simulating the component. If you cannot achieve the expected results with a certain set of test cases, add new test cases or modify existing test cases in the data file. Merge the test cases into a single data file.

Continue adding or modifying test cases until you achieve a test suite that satisfies your analysis goals.

4 Execute the test cases in software-in-the-loop or processor-in-the-loop mode.

5 After you have a complete test suite, you can:

- Simulate the model and execute the test cases to:

  - Record coverage.
  - Record output values to make sure that you get the expected results.

- Invoke the Code Generation Verification (CGV) API to execute the generated code for the model that contains the component in simulation, software-in-the-loop (SIL), or processor-in-the-loop (PIL) mode.

> **Note** To execute a model in different modes of execution, you use the CGV API to verify the numerical equivalence of results. See "Programmatic Code Generation Verification" (Embedded Coder).

Verify a Component Independently of the Container Model

Use component analysis to verify:

- Model blocks
- Atomic subsystems
- Stateflow atomic subcharts

1 Depending on the type of component, take one of the following actions:

- Model blocks — Open the referenced model.
- Atomic subsystems — Extract the contents of the subsystem into its own Simulink model.
- Atomic subcharts — Extract the contents of the Stateflow atomic subchart into its own Simulink model.

2 Create a harness model for:

- The referenced model
- The extracted model that contains the contents of the atomic subsystem or atomic subchart

3 Add or modify test cases in the Signal Builder of the harness model.

4 Log the input signals from the Signal Builder to the test unit.

5 Repeat steps 3 and 4 until you are satisfied with the test suite.

6 Merge the test case data into a single file.

7 Depending on your goals, take one of these actions:

- Execute the test cases to:
  - Record coverage.
  - Record output values and make sure that they equal the expected values.
- Invoke the Code Generation Verification (CGV) API to execute the test cases in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode on the generated code for the model that contains the component.

If the test cases do not achieve the expected results, repeat steps 3 through 5.

Verify a Model Block in the Context of the Container Model

Use system analysis to:

- Verify a Model block in the context of the block's container model.
- Analyze a closed-loop controller.

1 Log the input signals to the component by simulating the container model or analyze the model by using the Simulink Design Verifier software.

2 If you want to add test cases to your test suite or modify existing test cases, create a harness model with the logged signals.

3 Add or modify test cases in the Signal Builder in the harness model.

4 Log the input signals from the Signal Builder to the test unit.

5 Repeat steps 3 and 4 until you are satisfied with the test suite.

6 Merge the test case data into a single file.

7 Depending on your goals, do one of the following:

- Execute the test cases to:

    - Record coverage.

    - Record output values and make sure that they equal the expected values.

- Invoke the Code Generation Verification (CGV) API to execute the test cases in software-in-the-loop (SIL) or processor-in-the-loop (PIL) mode on the generated code for the model.

If the test cases do not achieve the expected results, repeat steps 3 through 5.

Fix Requirements-Based Testing Issues

This example shows how to address common traceability issues in model requirements and tests by using the Model Testing Dashboard. The dashboard analyzes the testing artifacts in a project and reports metric data on quality and completeness measurements such as traceability and coverage, which reflect guidelines in industry-recognized software development standards, such as ISO 26262 and DO-178C. The dashboard widgets summarize the data so that you can track your requirements-based testing progress and fix the gaps that the dashboard highlights. You can click the widgets to open tables with detailed information, where you can find and fix the testing artifacts that do not meet the corresponding standards.

Collect Metrics for the Testing Artifacts in a Project

The dashboard displays testing data for a model and the artifacts that the model traces to within a project. For this example, open the project and collect metric data for the artifacts.

1   Open the project. At the command line, type `dashboardCCProjectStart`.

2   Open the dashboard. On the **Project** tab, click **Model Testing Dashboard**.

3   If you have not previously opened the dashboard for the project, the dashboard must identify the artifacts in the project and trace them to the models. To run the analysis and collect metric results, click **Trace and Collect All**.

4   In the **Artifacts** pane, the dashboard organizes unit models under the component models that contain them in the model hierarchy. Artifacts such as requirements, test cases, and test results appear under the unit models that they trace to. View the metric results for the model `db_DriverSwRequest`. In the **Artifacts** pane, click the name of the model. The dashboard populates the widgets with data from the most recent metric collection for the model.
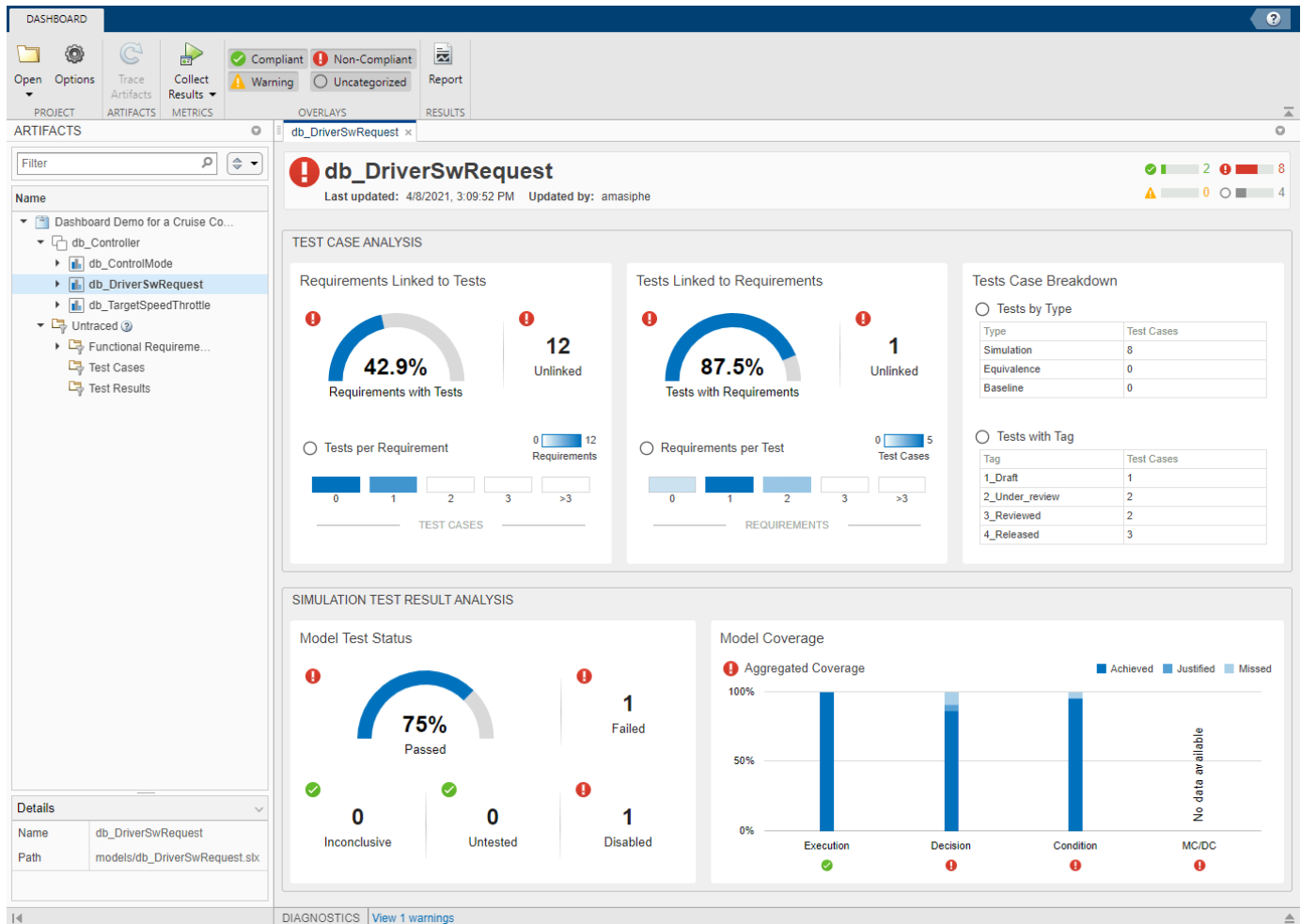
You can use the overlays in the Model Testing Dashboard to see if the metric results for a widget are compliant, non-compliant, or generate a warning that the metrics results should be reviewed. Results are compliant if they show full traceability, test completion, or model coverage. In the **Overlays** section, ensure the **Compliant** and **Non-Compliant** buttons are selected. The overlay appears on the widgets that have results in that category. You can see the total number of widgets in each compliance category in the top-right corner of the dashboard.

To see the compliance thresholds for a metric, point to the overlay icon.
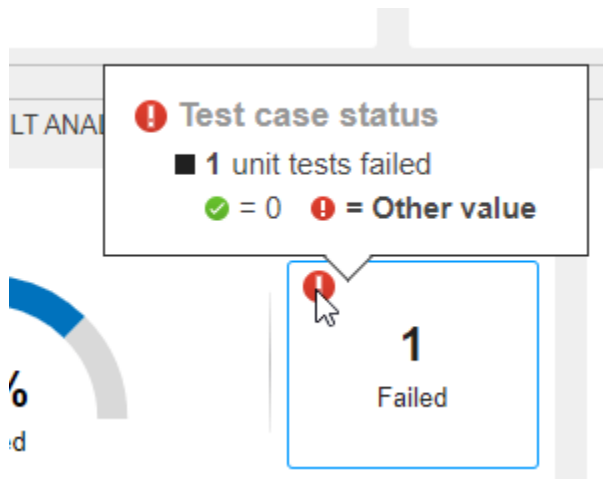
You can hide the overlay icons by clicking a selected category in the **Overlays** section.

For more information on the compliance thresholds for each metric, see "Model Testing Metrics" (Simulink Check).

Link a Requirement to its Implementation in a Model

On the **Artifacts** panel, the **Untraced** folder shows artifacts that do not trace to the unit models in the project. You can check the artifacts in this folder to see if there are any requirements that should be implemented by the models but are missing links. For this example, link one of these requirements to the model block that implements it and update the **Artifacts** panel to reflect the link.

1   In the **Artifacts** panel, navigate to the requirement **Untraced** > **Functional Requirements** > `db_req_func_spec.slreqx` > **Switch precedence**.

2   Open the requirement in the Requirements Editor. On the **Artifacts** panel, double-click **Switch precedence**. This requirement describes the order in which the cruise control system takes action if multiple switches are enabled at the same time. Keep the Requirements Editor open with the requirement selected.

3   Open the model `db_Controller`. To open the model from the Model Testing Dashboard, in the **Artifacts** panel, expand the folder **db_Controller** > **Design** and double-click `db_Controller.slx`.

4   The Model block `DriverSwRequest` references the model `db_DriverSwRequest`, which controls the order in which the cruise control system takes action when the switches are enabled. Link this model block to the requirement. Right-click the model block and select `Requirements > Link to Selection in Requirements Browser`.

5   Save the model. On the **Simulation** tab, click **Save**.

6   Save the requirements set. In the Requirements Editor, click the **Save** icon.

7   To update the artifact traceability information, in the Model Testing Dashboard, click **Trace Artifacts**.

The **Artifacts** panel shows the **Switch precedence** requirement under **db_Controller** > **Functional Requirements** > `db_req_func_spec.slreqx`. Next, find traceability issues in the artifacts by collecting metrics in the dashboard.

Address Testing Traceability Issues

Open the dashboard for the unit **db_DriverSwRequest** by clicking the name of the unit in the **Artifacts** panel. Because you changed the requirements file by adding a link, the dashboard widgets are highlighted in gray to show that the results might represent stale data. To update the results for the unit, click **Collect Results**.

The widgets in the **Test Case Analysis** section of the dashboard show data about the model requirements, test cases for the model, and links between them. The widgets indicate if there are gaps in testing and traceability for the implemented requirements.

Link Requirements and Test Cases

In the model `db_DriverSwRequest`, the **Requirements Linked to Tests** section shows that some of the requirements in the model are missing links to test cases. Examine the requirements by clicking one of the dashboard widgets. Then, use the links in the table to open the artifacts and fix the traceability issues.

To see detailed information about the unlinked requirements, in the **Requirements Linked to Tests** section, click the widget **Unlinked**. The table shows the requirements that are implemented in the model, but do not have links to a test case. The table is filtered to show only requirements that are missing links to test cases. For this example, link a test for the requirement `Set Switch Detection`.

Requirement linked to test cases		
Metric that determines if the requirement is linked to test cases.		

Artifact	Source	Test Link Status
Intermediate state	db_req_func_spec.slreqx	Missing linked tests
Resume Switch	db_req_func_spec.slreqx	Missing linked tests
Avoid repeating commands	db_req_func_spec.slreqx	Missing linked tests
Increment Switch	db_req_func_spec.slreqx	Missing linked tests
Set Switch	db_req_func_spec.slreqx	Missing linked tests
Driver Request	db_req_func_spec.slreqx	Missing linked tests
Set Switch Detection	db_req_func_spec.slreqx	Missing linked tests
Decrement Switch	db_req_func_spec.slreqx	Missing linked tests
Waiting state for Long Decrement switch detection	db_req_func_spec.slreqx	Missing linked tests
Enable Switch	db_req_func_spec.slreqx	Missing linked tests
Cancel Switch	db_req_func_spec.slreqx	Missing linked tests
Intermediate state	db_req_func_spec.slreqx	Missing linked tests

1　Open the requirement in the Requirements Editor. In the table, click `Set Switch Detection`.

2　In the Requirements Editor, examine the details of the requirement. This requirement describes the behavior of the `Set` switch when it is pressed. Keep the requirement selected in the Requirements Editor.

3　Check if there is already a test case for the switch behavior. To return to the metric results, at the top of the Model Testing Dashboard, click **db_DriverSwRequest**. The **Tests Linked to Requirements** section shows that one test case is not linked to requirements.

4   To see the unlinked test cases, in the **Tests Linked to Requirements** section, click **Unlinked**.

5   To open the test in the Test Manager, in the table, click the test case `Set button`. The test case verifies the behavior of the `Resume` switch. If there were not already a test case for the switch, you would add a test case by using the Test Manager.

6   Link the test case to the requirement. In the Test Manager, for the test case, expand the **Requirements** section. Click **Add > Link to Selected Requirement**. The traceability link indicates that the test case `Set button` verifies the requirement `Set Switch Detection`.

7   The metric results in the dashboard reflect only the saved artifact files. To save the test suite `db_DriverSwRequest_Tests.mldatx`, in the **Test Browser**, right-click `db_DriverSwRequest_Tests` and click **Save**.

8   Save the requirements file `db_req_func_spec.slreqx`. In the Requirements Editor, click the **Save** button.

Next, update the metric data in the dashboard to see the effect of adding the link.

Update Metric Results in the Dashboard

Update the metric results in the Model Testing Dashboard so that they reflect the traceability link between the requirement and the test case.

1   To analyze the artifact changes in the Model Testing Dashboard, click **Trace Artifacts**. The button is enabled when there are changes in the project artifacts that the dashboard has not analyzed.

2

At the top of the dashboard, the **Stale Metrics** icon indicates that at least one metric widget shows stale data for the model. Widgets that show stale metric data appear highlighted in grey. To refresh the widgets, re-collect the metric data for the model by clicking **Collect Results**.

The **Test Case Analysis** widgets show that there are 11 remaining unlinked requirements. The **Tests Linked to Requirements** section shows that there are no unlinked tests. Typically, before running the tests, you investigate and address these testing traceability issues by adding tests and linking them to the requirements. For this example, leave the unlinked artifacts and continue to the next step of running the tests.

Test the Model and Analyze Failures and Gaps

After you create and link unit tests that verify the requirements, run the tests to check that the functionality of the model meets the requirements. To see a summary of the test results and coverage measurements, use the widgets in the **Test Result Analysis** section of the dashboard. The widgets highlight testing failures and gaps. Use the metric results for the underlying artifacts to address the issues.

Perform Unit Testing

Run the test cases for the model by using the Test Manager. Save the results as an artifact in the project and review them in the Model Testing Dashboard.

1   Open the unit tests for the model in the Test Manager. In the Model Testing Dashboard, in the **Artifacts** pane, expand the model `db_DriverSwRequest`. Expand the **Test Cases** folder and double-click the test file `db_DriverSwRequest_Tests.mldatx`.

2   In the Test Manager, click **Run**.

3   To use the test results in the Model Testing Dashboard, export the test results and save the file in the project. On the **Tests** tab, in the **Results** section, click **Export**. Name the results file `Results1.mldatx` and save the file under the project root folder.
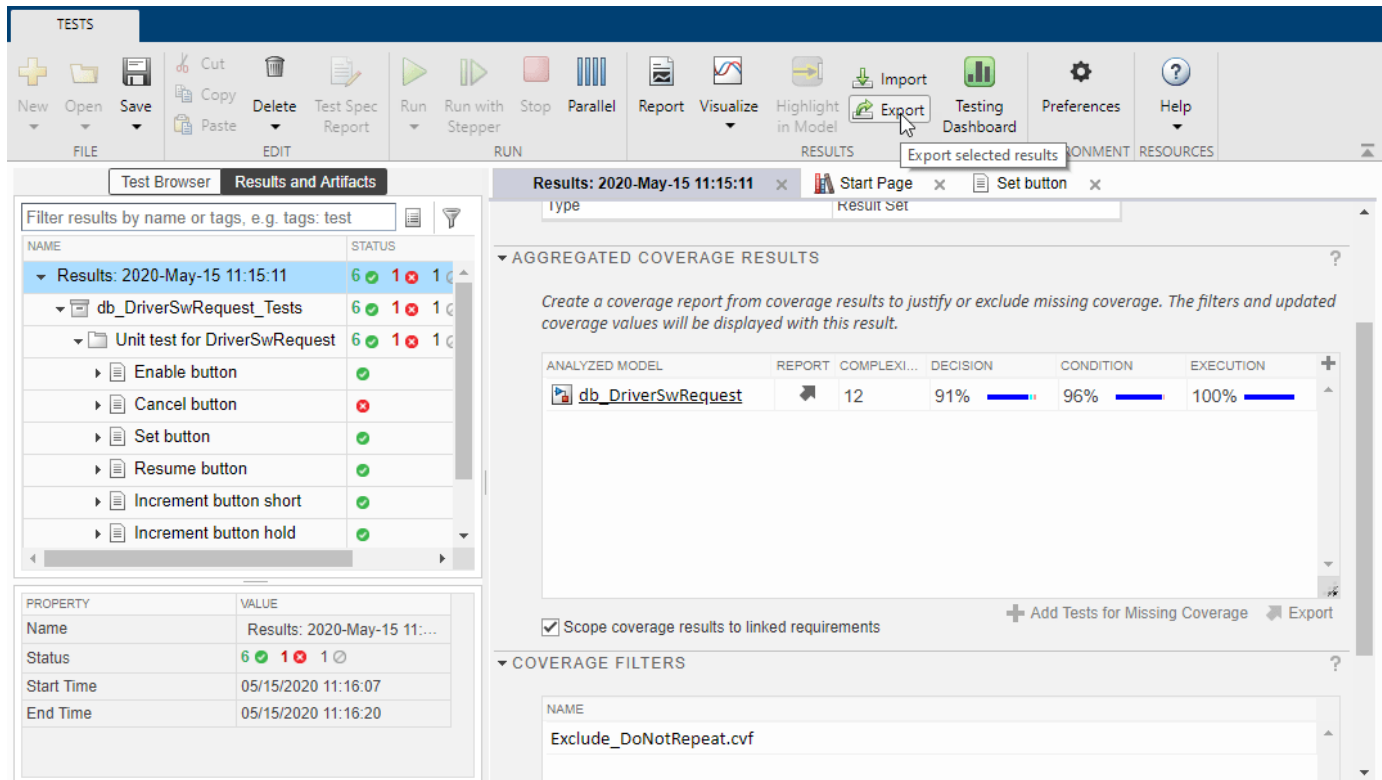
The Model Testing Dashboard detects that you exported the results and automatically updates the **Artifacts** panel to reflect the new results. The widgets in the **Test Result Analysis** section are highlighted in grey to indicate that they are showing stale data. To update the data in the dashboard widgets, click **Collect Results**.

Address Testing Failures and Gaps

In the model `db_DriverSwRequest`, the **Model Test Status** section indicates that one test failed and one test was disabled during the latest test run. Open the tests and fix these issues.

1   To view the disabled test, in the dashboard, click the **Disabled** widget. The table shows the disabled test cases for the model.

2   Open the disabled test in the Test Manager. In the table, click the test `Decrement button hold`.

3   Enable the test. In the **Test Browser**, right-click the test case and click **Enabled**. Save the test suite file.

4   To view the failed test, in the dashboard, click the **Failed** widget.

5   Open the failed test in the Test Manager. In the table, click the test `Cancel button`.

6   Examine the test failure in the Test Manager. You can determine if you need to update the test or the model by using the test results and links to the model. For this example, instead of fixing the failure, continue on to examine test coverage.

Check if the tests that you ran fully exercised the model design by using the coverage metrics. For this example, the **Model Coverage** section of the dashboard indicates that some conditions in the model were not covered. Place your cursor over the bar in the widget to see what percent of condition coverage was achieved. For this example, 86.4% of decisions were covered by the tests and 4.55% of the decisions were justified in a coverage filter.

1   View the decision coverage details. Click the **Decision** bar.

2   In the table, expand the model artifact. The table shows the test case results for the model and the results file that contains them. Open the results file `Results1.mldatx` in the Test Manager.

3   To see detailed coverage results, open the model in the Coverage perspective. In the Test Manager, in the **Aggregated Coverage Results** section, in the **Analyzed Model** column, click `db_DriverSwRequest`.

4   Coverage highlighting on the model shows the points that were not covered by the test cases. For a point that is not covered, add a test that covers it. Find the requirement that is implemented by the model element or, if there is none, add a requirement for it. Link the new test case to the requirement. If the point should not be covered, justify the missing coverage by using a filter. For this example, do not fix the missing coverage.

Once you have updated the unit tests to address failures and gaps, run the tests and save the results. Then examine the results by collecting the metrics in the dashboard.

Iterative Requirements-Based Testing with the Model Testing Dashboard

In a project with many artifacts and traceability connections, you can monitor the status of the design and testing artifacts whenever there is a change to a file in the project. After you change an artifact, check if there are downstream testing impacts by updating the tracing data and metric results in the dashboard. Use the tables to find and fix the affected artifacts. Track your progress by updating the dashboard widgets until they show that the model testing quality meets the standards for the project.

Verification and Validation

Test Model Against Requirements and Report Results

Requirements – Test Traceability Overview

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

This example demonstrates a common requirements-based testing workflow for a cruise control model. You start with a requirements set, a model, and a test case. You add traceability between the tests and the safety requirements. You run the test, summarize the verification status, and report the results.
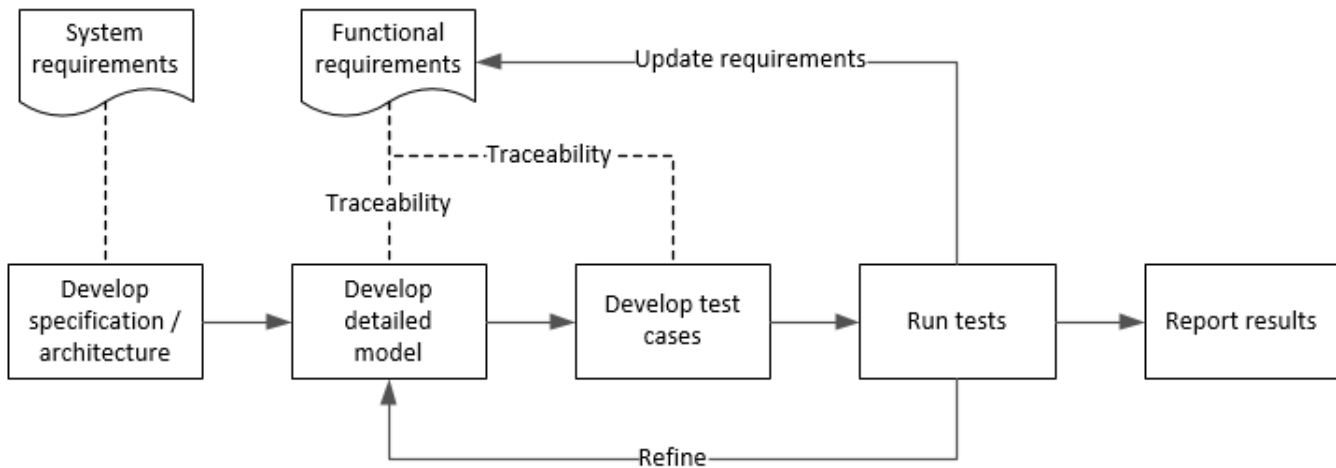
In this example, you conduct a simple test of two requirements in the set:

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

Display the Requirements

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

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

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

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

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

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

Link Requirements to Tests

Link the requirements to the test case.

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

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

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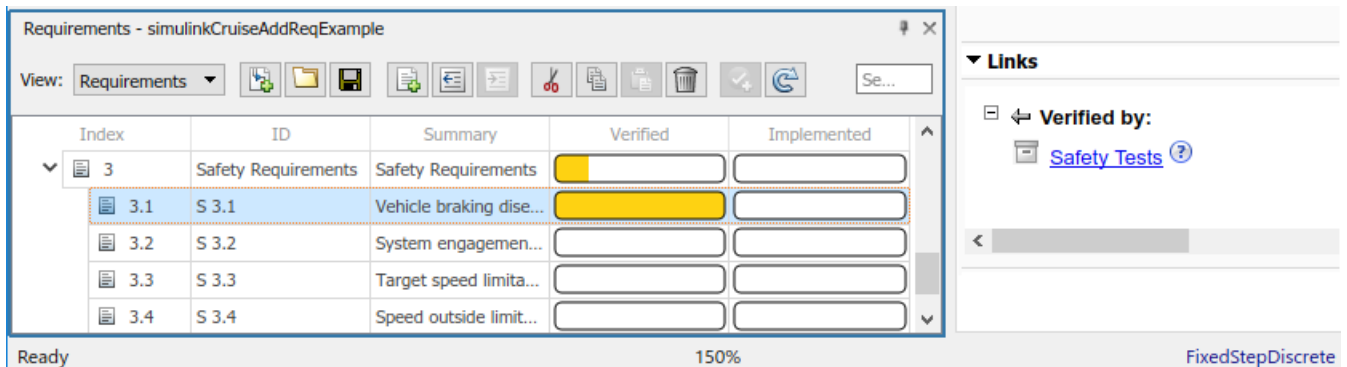
2 Also add a link for item S 3.4.

Run the Test

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

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

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

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

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

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

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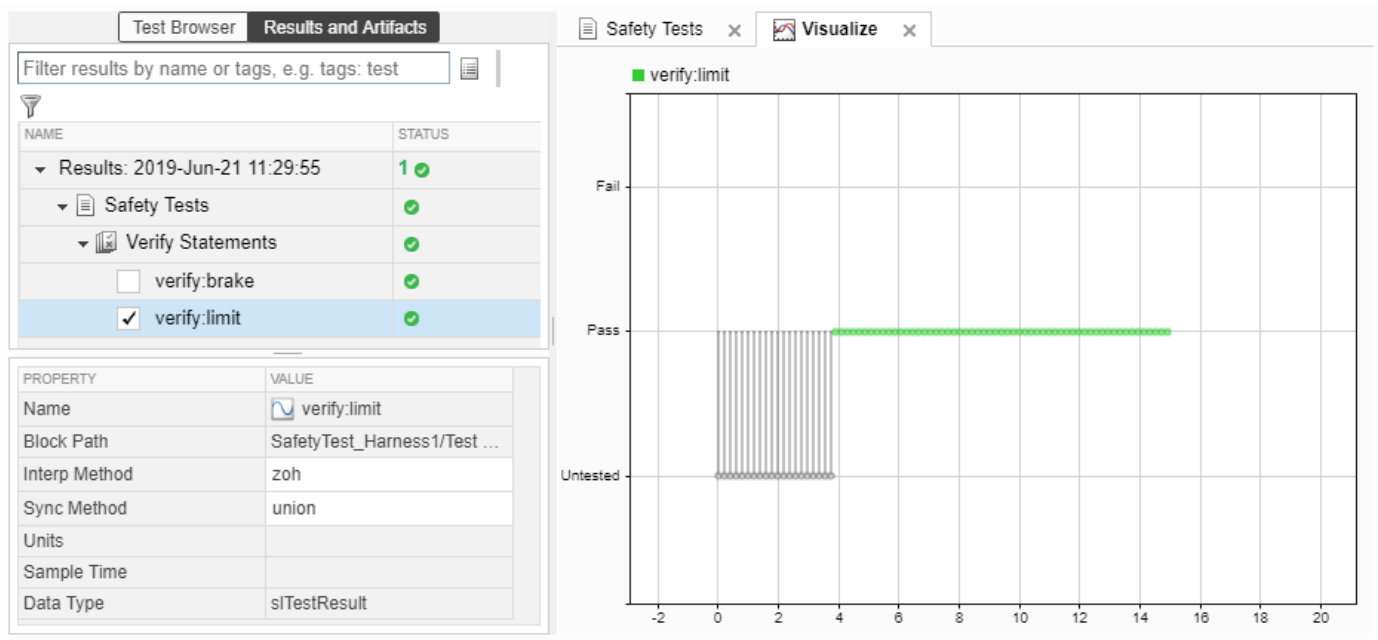
3 Return to the model and refresh the Requirements. The green bar in the **Verified** column indicates that the requirement has been successfully verified.
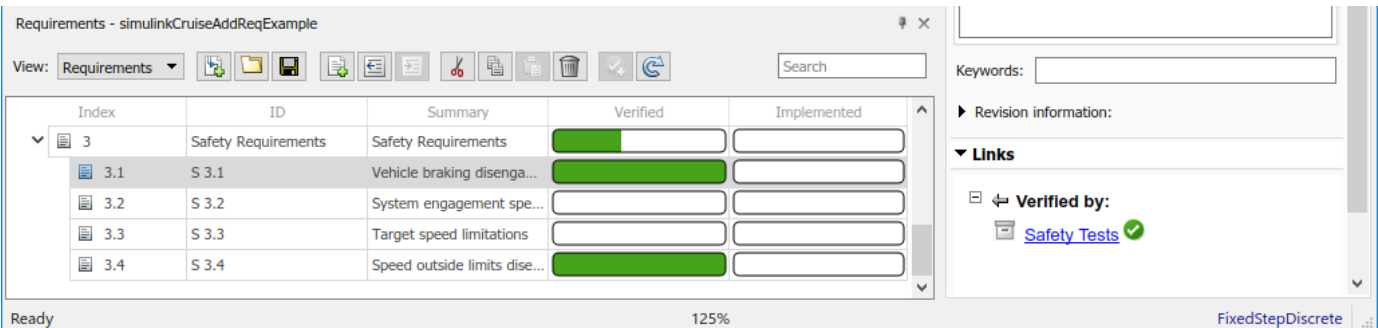
Report the Results

1 Create a report using a custom Microsoft Word template.

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

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

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

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

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

e Click **Create**.

2 Review the report.

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

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

See Also

Related Examples

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

External Websites

- Requirements-Based Testing Workflow

Analyze a Model for Standards Compliance and Design Errors

Standards and Analysis Overview

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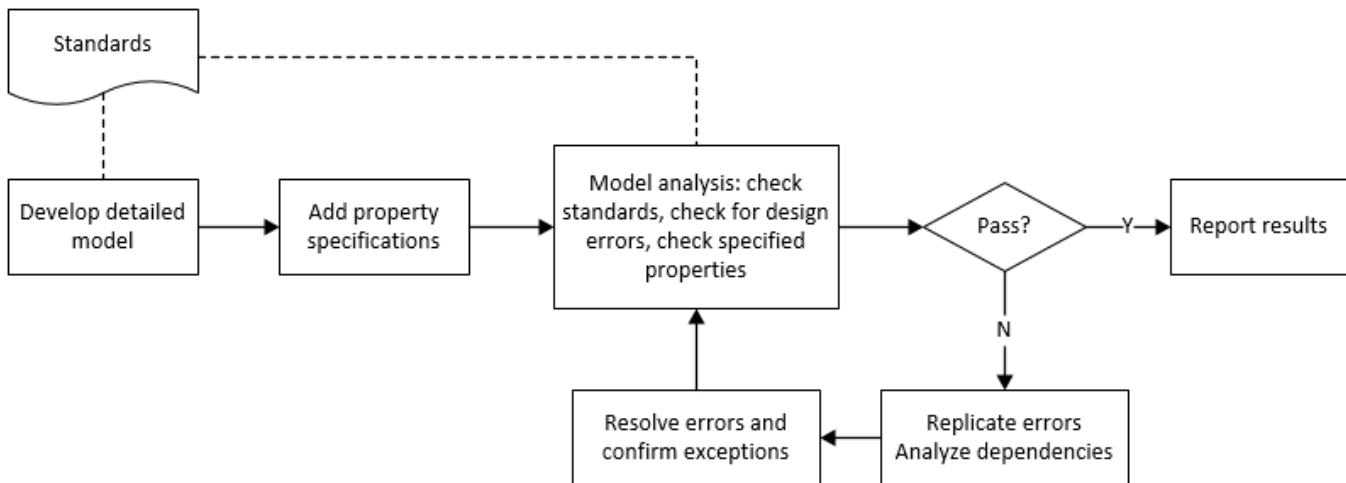
Check Model for Style Guideline Violations and Design Errors

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

Check Model for MAB Style Guideline Violations

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

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

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

2   Open the model. On the command line, enter

```
open_system simulinkCruiseErrorAndStandardsExample
```

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

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

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

> **a** In the left pane, in the **By Product > Simulink Check > Modeling Standards > MAB Checks** folder, select:
>
>   - **Check Indexing Mode**
>   - **Check model diagnostic parameters**
>
> **b** Right-click on the **MAB Checks** node and select `Run Selected Checks`.
>
> **c** To review the configuration parameter settings that violate MAB style guidelines, click on the **Check model diagnostic parameters** check. The analysis results appear in the right pane and include the recommended action.
>
> **d** Click the parameter hyperlinks, which opens the Configuration Parameters dialog box, and update the model diagnostic parameters. Save the model.
>
> **e** To verify that your model passes, rerun the check. Repeat steps `c` and `d`, if necessary, to reach compliance.
>
> **f** To generate a results report of the Simulink Check checks, select the **MAB Checks** node, and then, in the right pane click **Generate Report...**.

Check Model for Design Errors

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

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

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

3 In the right pane, select **Show report after run** and **Run Selected Checks**.

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

> **a** Review the analysis results on the model. Click **Highlight analysis results on model**. Click the `Compute target speed` subsystem, outlined in red. The Simulink Design Verifier Results Inspector window provides derived ranges that can help you understand the source of an error by identifying the possible signal values.
>
> **b** Review the harness model or create one if it does not already exist.
>
> **c** View tests and export test cases.
>
> **d** Review the analysis report. To see a detailed analysis report, click **HTML** or **PDF**.

See Also

Related Examples

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

Perform Functional Testing and Analyze Test Coverage

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

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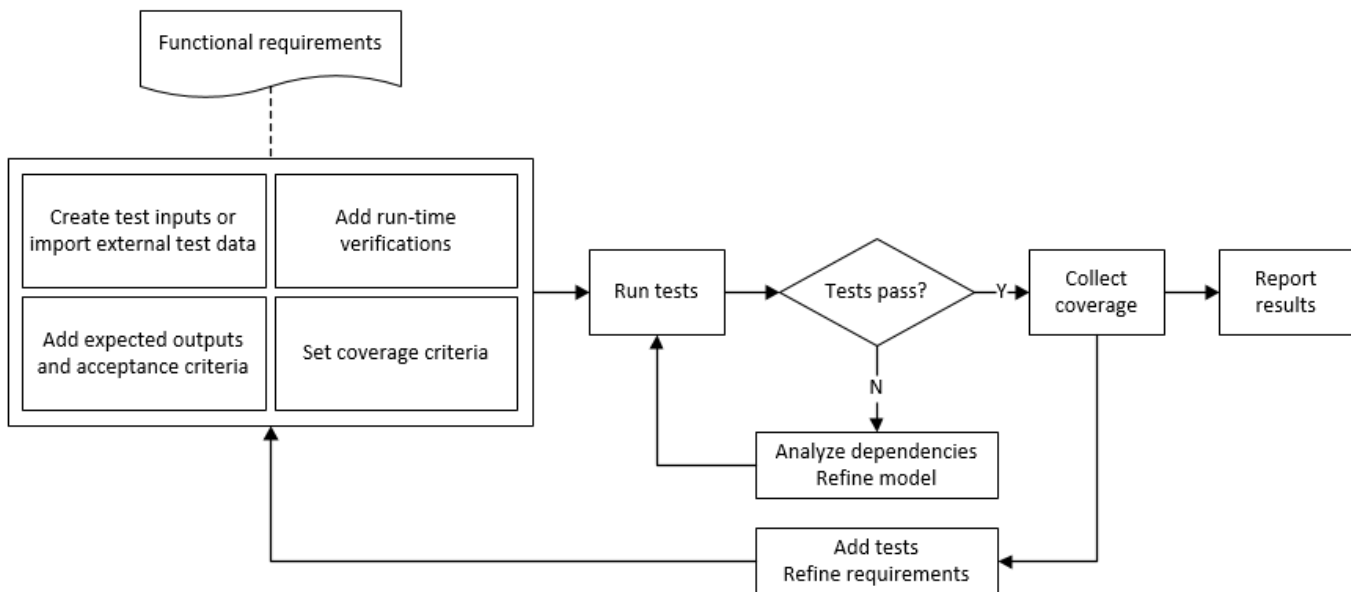
Incrementally Increase Test Coverage Using Test Case Generation

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

Explore the Test Harness and the Model

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

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

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

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

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

- Brake pedal is pressed

- Speed exceeds a limit

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

Measure Model Coverage

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

- Select **Record coverage for referenced models**

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

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

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

Generate Tests to Increase Model Coverage

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

See Also

Related Examples

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

Analyze Code and Test Software-in-the-Loop

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

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

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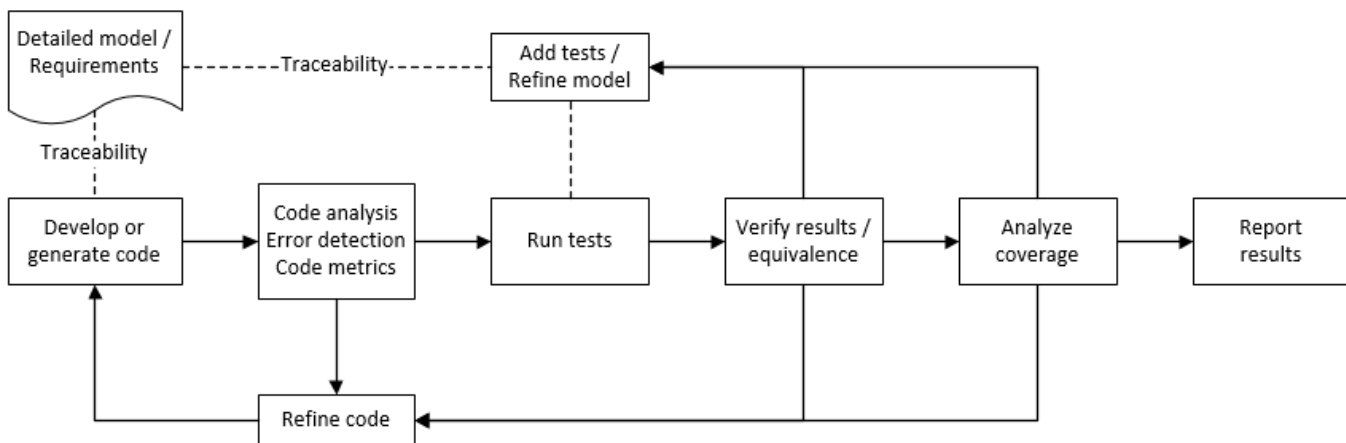
Analyze Code for Defects, Metrics, and MISRA C:2012

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

1   Open the project.

    ```
    path = fullfile(matlabroot,'toolbox','shared','examples',...
    'verification','src','cruise')
    run(fullfile(path,'slVerificationCruiseStart'))
    ```
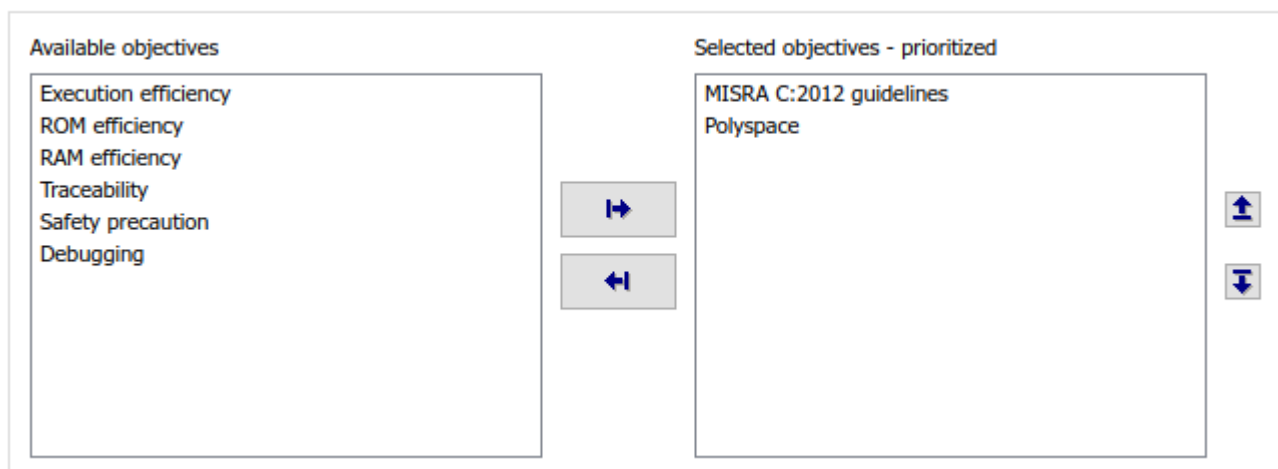2   From the project, open the model `simulinkCruiseErrorAndStandardsExample`.

Compute target speed
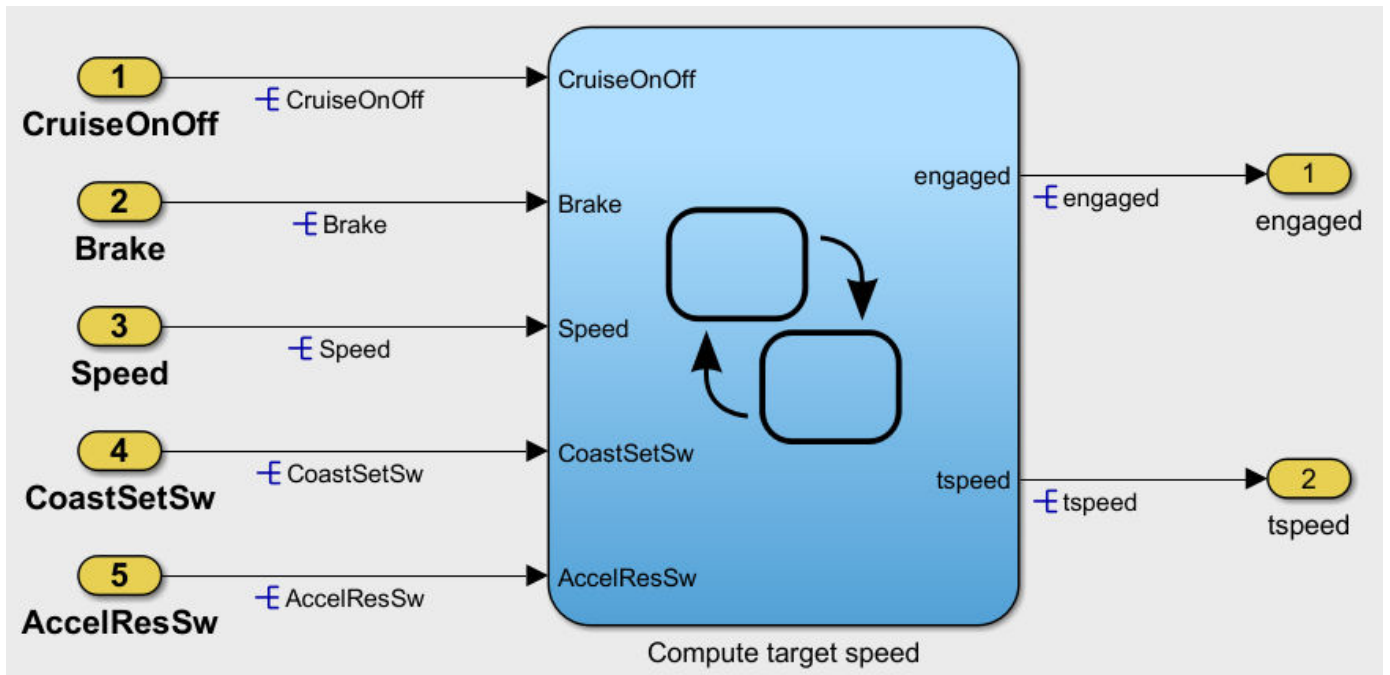
Run Code Generator Checks

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

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

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

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

Available objectives	Selected objectives - prioritized
Execution efficiency	MISRA C:2012 guidelines
ROM efficiency	Polyspace
RAM efficiency	
Traceability	
Safety precaution	
Debugging	

3   Click **Run Selected Checks**.

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

model, the check for incompatible blocks passes, but some configuration settings are incompatible with MISRA compliance and Polyspace checking.
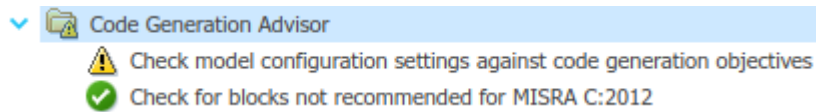
4 Click the check that did not pass. Accept the parameter changes by selecting **Modify Parameters**.

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

Run Model Advisor Checks

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

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

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

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

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

Generate and Analyze Code

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

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

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

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

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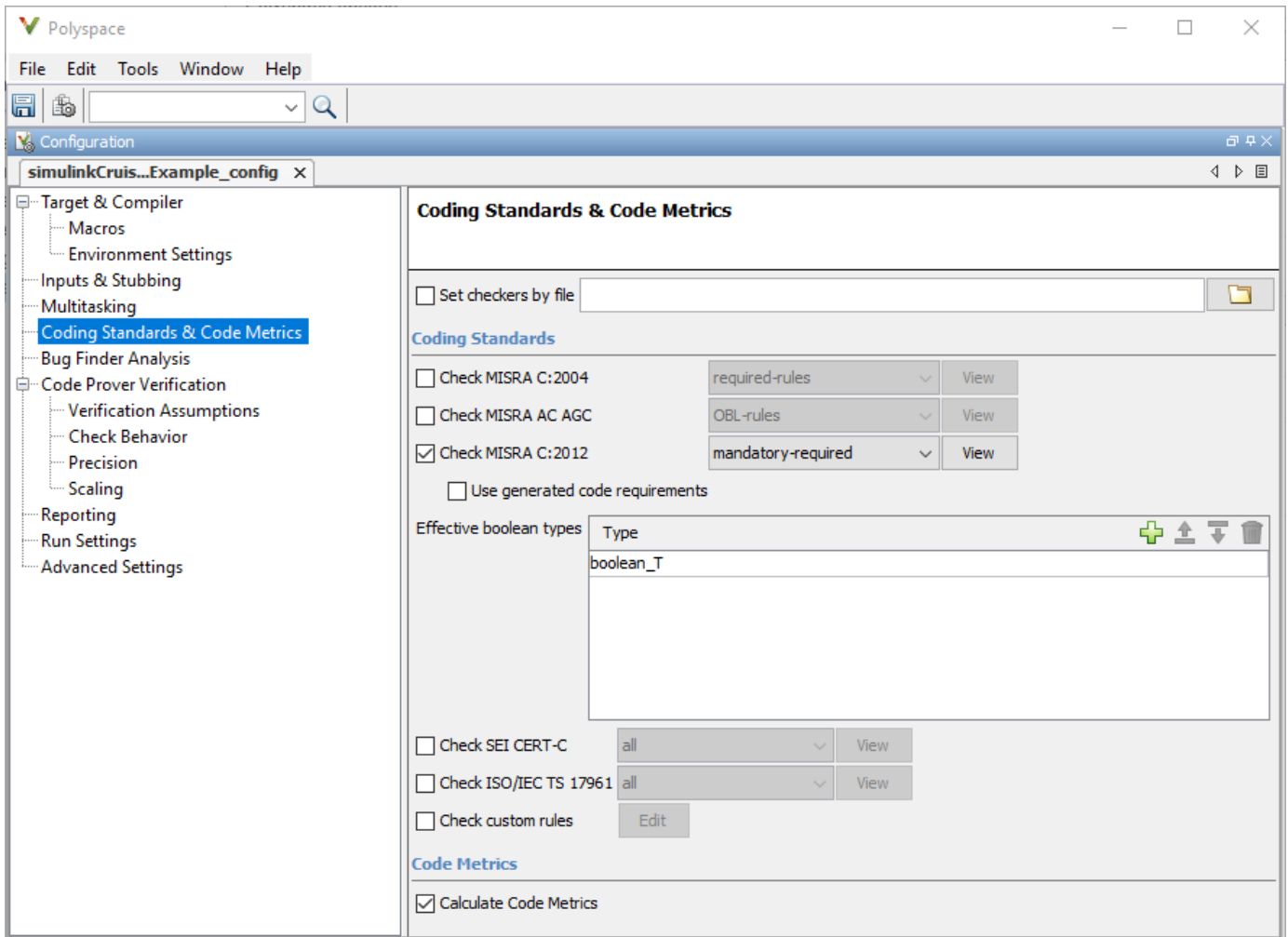
5   On the left pane, click **Coding Standards & Code Metrics**, then select **Calculate Code Metrics** to enable code metric calculations for your generated code.

6   Save and close the Polyspace configuration window.

7   From your model, right-click Compute target speed and select **Polyspace > Verify > Code Generated For Selected Subsystem**.

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

Review Results

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

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

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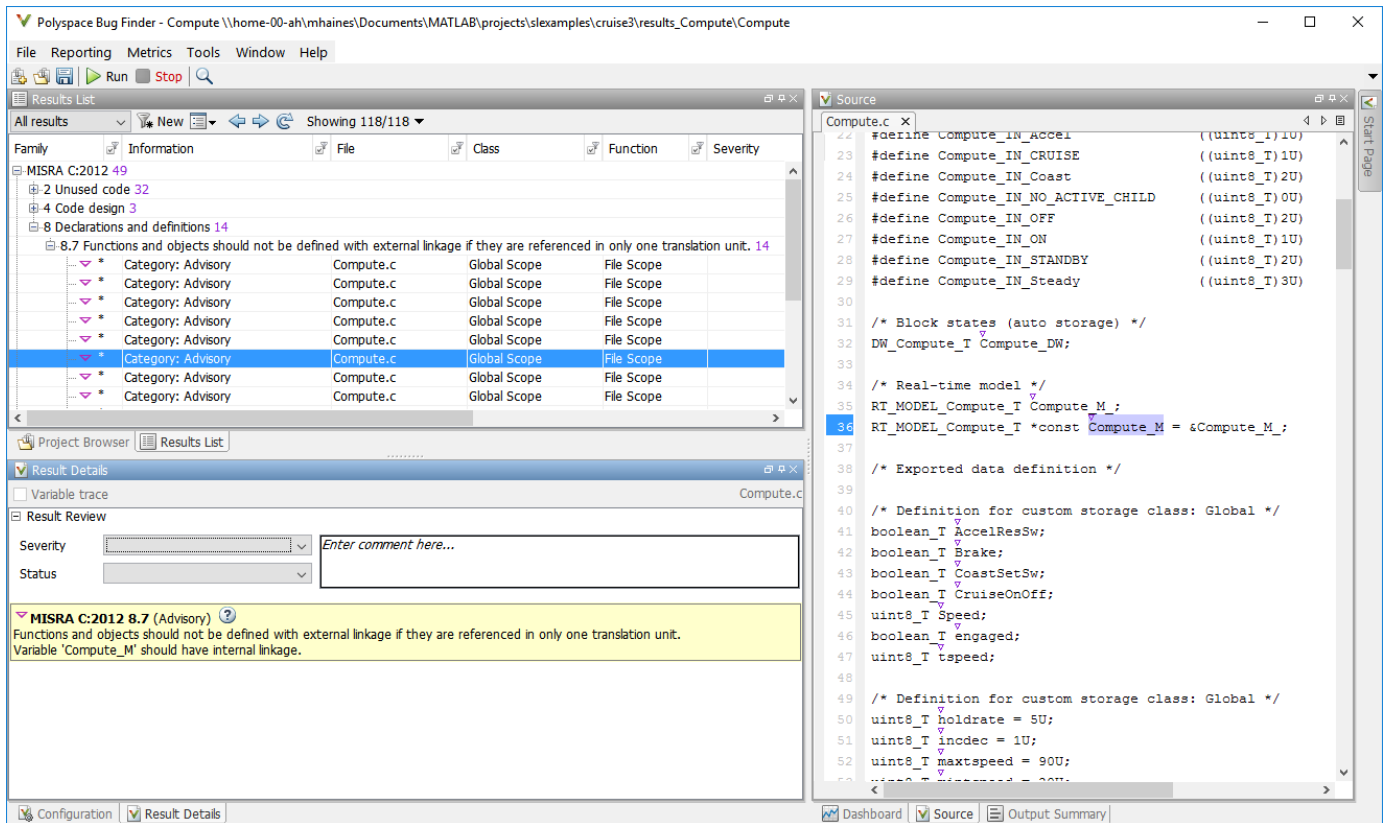
2   In your model, right-click Compute target speed and select **Polyspace > Options**.

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

4   Click **Configure**.

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

6   Save and close the Polyspace configuration window.

7   Rerun the analysis with the new configuration.

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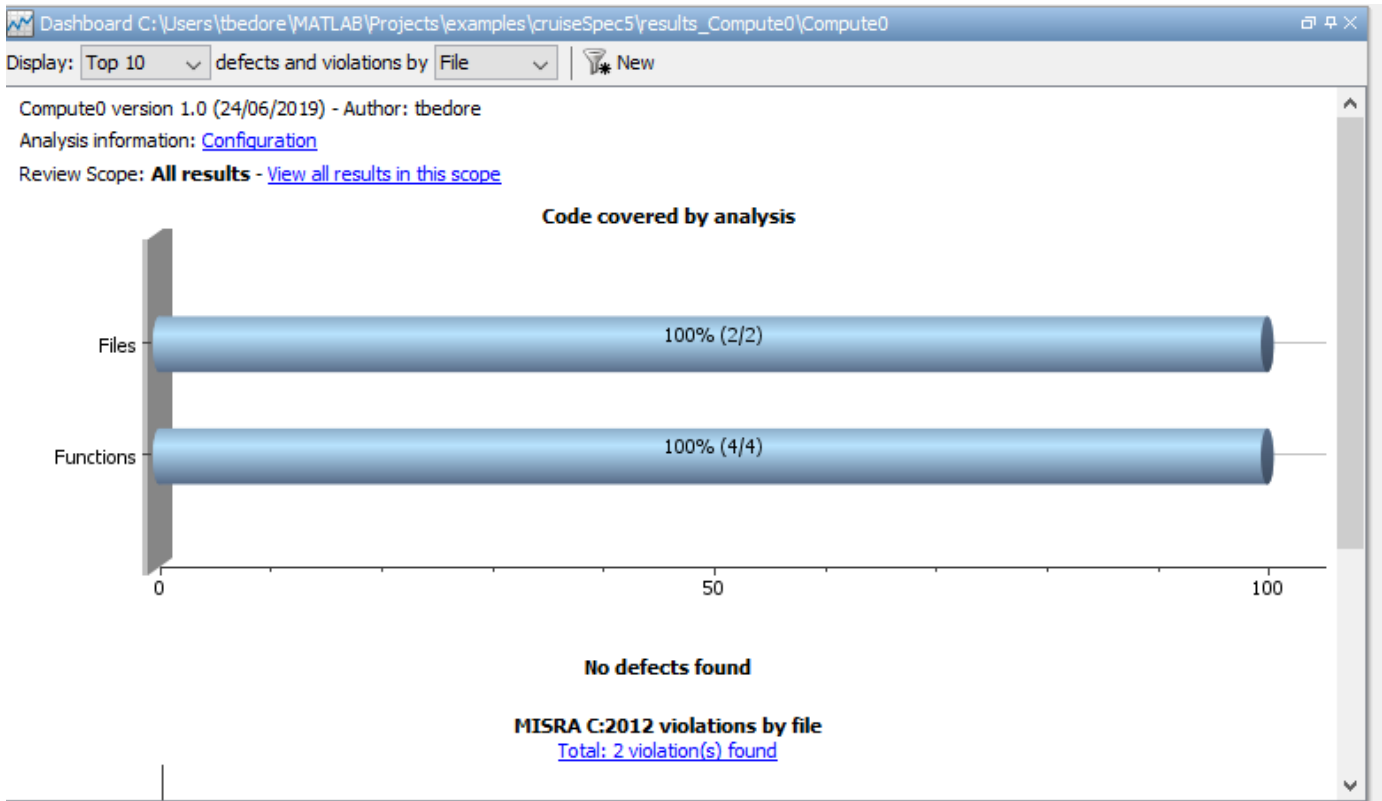
When you integrate this model with its parent model, you can add the rest of the MISRA C:2012 rules.

Generate Report

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

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

    The report is saved in the same folder as your results.
5   To open the report, select **Reporting > Open Report**.

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

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

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

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

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

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

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

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

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

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

See Also

Related Examples

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