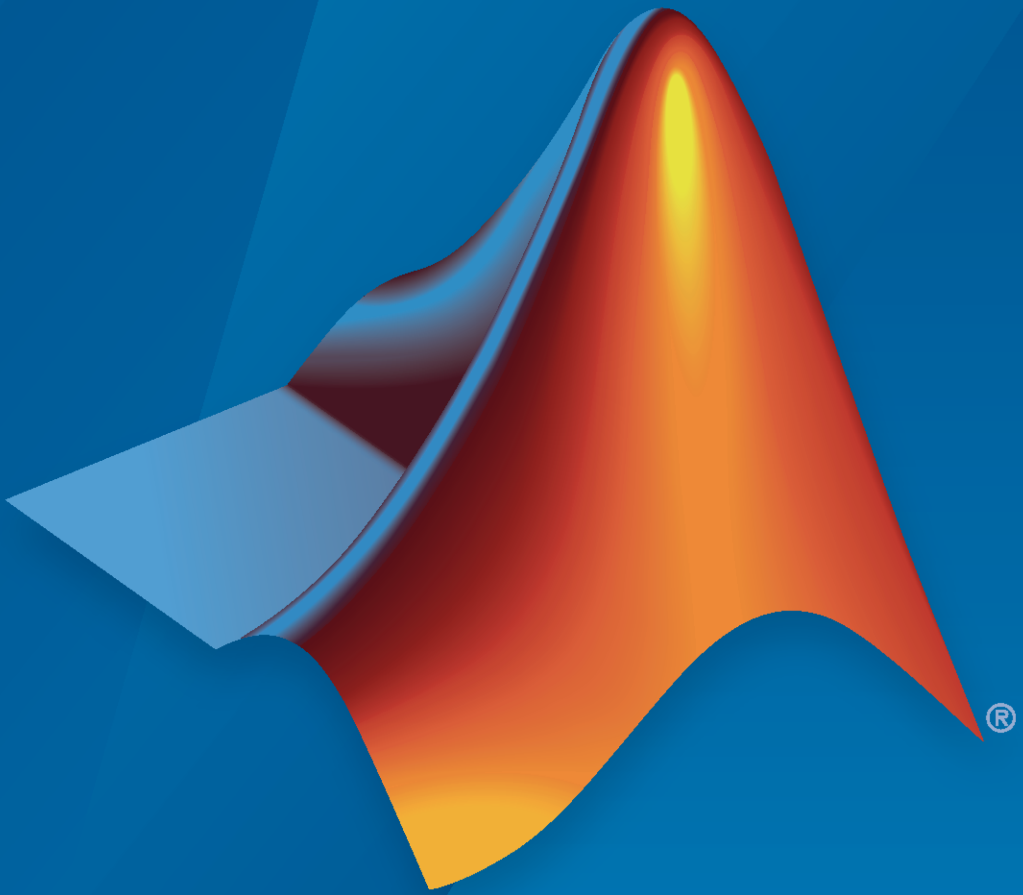
Simulink® Test™ Release Notes



MATLAB®&SIMULINK®

MathWorks®

Contents

R2023a

R2022b

R2023a

Control when to log signal data by using start and stop triggers

You can now control when signal data is logged by using start and stop triggers in the Test Manager and the Simulink Test API. You can use a trigger for a test case to start logging based on a specified condition being met or a time delay after simulation starts. The stop trigger stops logging when a specified condition is met or when the specified amount of time after the start trigger has occurred. For test cases, the triggering options are in the **Simulation Output** section of the Test Manager.

The API has these new classes for triggered logging:

- `sltest.testmanager.TriggerMode` — Whether the trigger is the same as simulation, a conditional, or a duration trigger
- `sltest.testmanager.OutputTrigger` — Trigger conditions or durations, and definitions of symbols to use in the triggers
- `sltest.testmanager.OutputTriggerResult` — Read-only summary of trigger mode, start and stop logging conditions or durations, and the simulation start and stop times when logging occurs

Copy and paste logged signal sets between test cases

You can now copy and paste one or more logged signal sets from one test case to another test case. In the Test Manager, to copy a signal set, select the signal set from the **Simulation Outputs** section, right-click on the signal set, and select **Copy**. Then display the destination test case, right-click inside the **Simulation Outputs** section, and select **Paste**.

You can also use the `addLoggedSignalSet` function to copy and paste a single logged signal set to the test case. The optional logged signal set input is an `sltest.testmanager.LoggedSignalSet` object from another test case.

Select component-under-test from within Model blocks or test harnesses

As the component to test, you can now select a Model block that is inside either another Model block or a test harness. The `setProperty` and `getProperty` methods include a new `ComponentUnderTestName` name-value argument with a `Simulink.BlockPath` object as its input. Previously, you could test a whole Model block only by extracting it into a separate test harness.

Copy, paste, and delete logical and temporal assessments

You can now copy, paste, and delete logical and temporal assessments in the Test Manager by using a context menu or keyboard shortcuts. See “Logical and Temporal Assessments”.

Reuse test cases in different simulation modes

You can now run a test in a different simulation mode without having to change the test definitions by using the new `SimulationMode` name-value argument in these methods:

- `sltest.testmanager.run`
- `sltest.testmanager.TestFile.run`

- `sltest.testmanager.TestSuite.run`
- `sltest.testmanager.TestCase.run`

Valid `SimulationMode` values are:

- `Normal`
- `Accelerator`
- `Rapid Accelerator`
- `Software-in-the-loop`
- `Processor-in-the-loop`

Exclude inactive variants from coverage

The Test Manager **Coverage Settings** section and the **Aggregated Coverage Results** section of the **Results and Artifacts** pane now include a new **Exclude inactive variants** option for test files. Selecting this option excludes variants that are not active at any time while the test runs from coverage results and reporting. If you select or clear this option when displaying the test results, the coverage results update automatically.

Coverage for MATLAB-based Simulink tests in Test Manager

You can now enable coverage collection and set coverage metrics for MATLAB®-based Simulink tests in the Test Manager. Test suites and test cases inherit the coverage settings from the test file. However for MATLAB-based Simulink tests, you can set and change coverage and coverage metrics only at the test file level.

Specify custom file names for MATLAB Unit Test framework coverage reports

You can use the new `ReportName` name-value argument of the `sltest.plugins.coverage.ModelCoverageReport` class to specify a file name for HTML and their associated CVT model coverage report files. If a test suite has multiple models, each HTML and CVT file name has the model name appended to the specified file name. The syntaxes for inputs to `sltest.plugins.coverage.ModelCoverageReport` are:

- `sltest.plugins.coverage.ModelCoverageReport` — Uses the current working folder as the file path of the folder in which to place the files and uses the default report name
- `sltest.plugins.coverage.ModelCoverageReport(path)` — Uses the specified path of the folder in which to place the files and uses the default report name
- `sltest.plugins.coverage.ModelCoverageReport(path,'ReportName',name)` — Uses the specified path and report file name
- `sltest.plugins.coverage.ModelCoverageReport('ReportName',name)` — Uses the current working folder for the file location and the specified report file name

Observe bus signals configured as internal variables of FMU

In Observer models, you can now observe bus signals that are configured as internal variables of a functional mockup unit (FMU). The Manager Observer dialog box displays the internal variables of the FMU.

Updated execution order for test case callbacks

When multiple test cases are run at the same time for different models, the order in which the callbacks for these test cases are executed is changed back to the order used previous to R2020a. The only difference between the new execution order and the order previous to R2020a is that models are not closed in between tests and instead are closed only after all tests are complete.

Note The updated execution order does not affect multirelease tests or Simulink Real-Time™ tests. The execution order for both of these types of tests is the same as the R2023a execution order and has remained unchanged since R2019b.

The following table shows the old and new execution order for two test cases run on separate models. The models are not open before the test runs.

Old Execution Order (R2020a - R2022b)	New Execution Order
Run test case 1 `Pre-Load` callback. Then run model 1 `PreLoadFcn` callback.	Run test case 1 `Pre-Load` callback. Then, run model 1 `PreLoadFcn` callback.
Load model 1.	Load model 1.
Run model 1 `PostLoadFcn` callback.	Run model 1 `PostLoadFcn` callback. Then, run test case 1 `Post-Load` callback.
Run test case 2 `Pre-Load` callback. Then run model 2 `PreLoadFcn` callback.	Simulate model 1 for test case 1.
Load model 2.	Run test case 1 `Cleanup` callback.
Run model 2 `PostLoadFcn` callback. Then run test case 1 `Post-Load` callback.	
Simulate model 1 for test case 1.	Run test case 2 `Pre-Load` callback. Then run model 2 `PreLoadFcn` callback.
Run test case 1 `Cleanup` callback.	Load model 2.
Run test case 2 `Post-Load` callback.	Run model 2 `PostLoadFcn` callback. Then, run test case 2 `Post-Load` callback.
Simulate model 2 for test case 2.	Simulate model 2 for test case 2.
Run test case 2 `Cleanup` callback.	Run test case 2 `Cleanup` callback.
Run model 1 `CloseFcn` callback.	Run model 1 `CloseFcn` callback.*
Run model 2 `CloseFcn` callback.	Run model 2 `CloseFcn` callback.*

* The order in which models are closed using the `CloseFcn` might be different than the order in which they were opened or run.

The following table shows the old and new execution orders for test cases run on a single model for a test suite with two test cases. The model is not open before the test suite runs.

Old Execution Order (R2020a - R2022b)	New Execution Order
Run test case 1 `Pre-Load` callback. Then run model `PreLoadFcn` callback.	Run test case 1 `Pre-Load` callback. Then run model `PreLoadFcn` callback.
Load model.	Load model.
Run model `PostLoadFcn` callback.	Run model `PostLoadFcn` callback. Then run test case 1 `Post-Load` callback.
Run test case 2 `Pre-Load` callback.	Simulate model.
Run test case 1 `Post-Load` callback	Run test case 1 `Cleanup` callback.
Simulate model.	Run test case 2 `Pre-Load` callback.
Run test case 1 `Cleanup` callback.	Run test case 2 `Post-Load` callback.
Run test case 2 `Post-Load` callback.	Simulate model.
Simulate model.	Run test case 2 `Cleanup` callback.
Run test case 2 `Cleanup` callback.	Run model `CloseFcn` callback.
Run model `CloseFcn` callback.	

The following table shows the old and new execution orders for test cases run on a single model for a test suite with two test cases. The model is open before the test suite runs.

Notice that the model `PreLoadFcn` and `PostLoadFcn` callbacks do not execute because the model is already loaded before the test runs. The model `CloseFcn` callback does not execute either because the model is left open after test completion.

Old Execution Order (R2020a - R2022b)	New Execution Order
Run test case 1 `Pre-Load` callback.	Run test case 1 `Pre-Load` callback.
Run test case 2 `Pre-Load` callback.	Run test case 1 `Post-Load` callback.
Run test case 1 `Post-Load` callback	Simulate model.
Simulate model.	Run test case 1 `Cleanup` callback.
Run test case 1 `Cleanup` callback.	Run test case 2 `Pre-Load` callback.
Run test case 2 `Post-Load` callback.	Run test case 2 `Post-Load` callback.
Simulate model.	Simulate model.
Run test case 2 `Cleanup` callback.	Run test case 2 `Cleanup` callback.

Automatic requirements linking when importing Simulink Design Verifier test cases

When you import tests from Simulink Design Verifier™ for a model that contains a Requirements Table block, the requirements are linked automatically to the imported test cases. Previously, you had to manually create the links from the tests to the requirements.

R2022b

Enhanced Test Manager display results

The Test Manager results have been updated to group bus and multidimensional signals according to their data hierarchy. These changes make it straightforward to associate a composite signal with its underlying signal elements. You can also associate a Simulation Data Inspector (SDI) view file with a test case. Use the view file to reuse the configuration of results plots and signals when you run test cases.

Create batch tests with Create Test for Model Component wizard

You can now select multiple test components and create multiple test cases and test harnesses at the same time in the Create Test for Model Component wizard. See Generate Tests and Test Harnesses for a Model or Components.

Create test harnesses for multiple components using sltest.harness.create

You can now use the `sltest.harness.create` function in batch mode, which lets you create test harnesses for multiple components at the same time. These name-value arguments have been updated for use in batch mode:

- `harnessOwners` — Accepts an array of components
- `TopModel` — Identifies the top model for the components in `harnessOwners`
- `Name` — Specifies the name to use as a prefix for the harness names
- `HarnessPath` — Specifies the folder in which to save the harnesses

Atomic subsystem equivalence test support for AUTOSAR

You can now perform equivalence tests for atomic subsystems on the AUTOSAR target. Equivalence tests are also known as back-to-back tests.

Linux host support for real-time testing

You can now use Simulink Test on a Linux® machine to run real-time test cases using Simulink Real-Time.

Observe conditional subsystem data

You can now use observers with conditional subsystems. See Observe Conditional Subsystem Signals. When a subsystem is active, the observer reports data for that subsystem. The supported conditional subsystems are:

- Enabled subsystems
- Triggered subsystems
- Enabled and triggered subsystems
- If and Switch Case action subsystems

See Conditionally Executed Subsystems Overview for information on conditional subsystems. See

Test file results returned in separate results sets

You can now run all tests as separate test results in the Simulink Test Manager by selecting **Run > Run All in Separate Results Sets**. If you have more than one test file open in the Test Manager, selecting this option returns the results for each file in separate result sets. Previously, results were returned in a single test set, regardless of how many test files were open.

Excel file adaptor registration

You can use the new `sltest.testmanager.registerTestAdapter` function to register a MATLAB function that reads and converts Excel® data to a format readable by Simulink Test. After you register the adaptor, it appears in the **External File** section of the test case in the Test Manager.

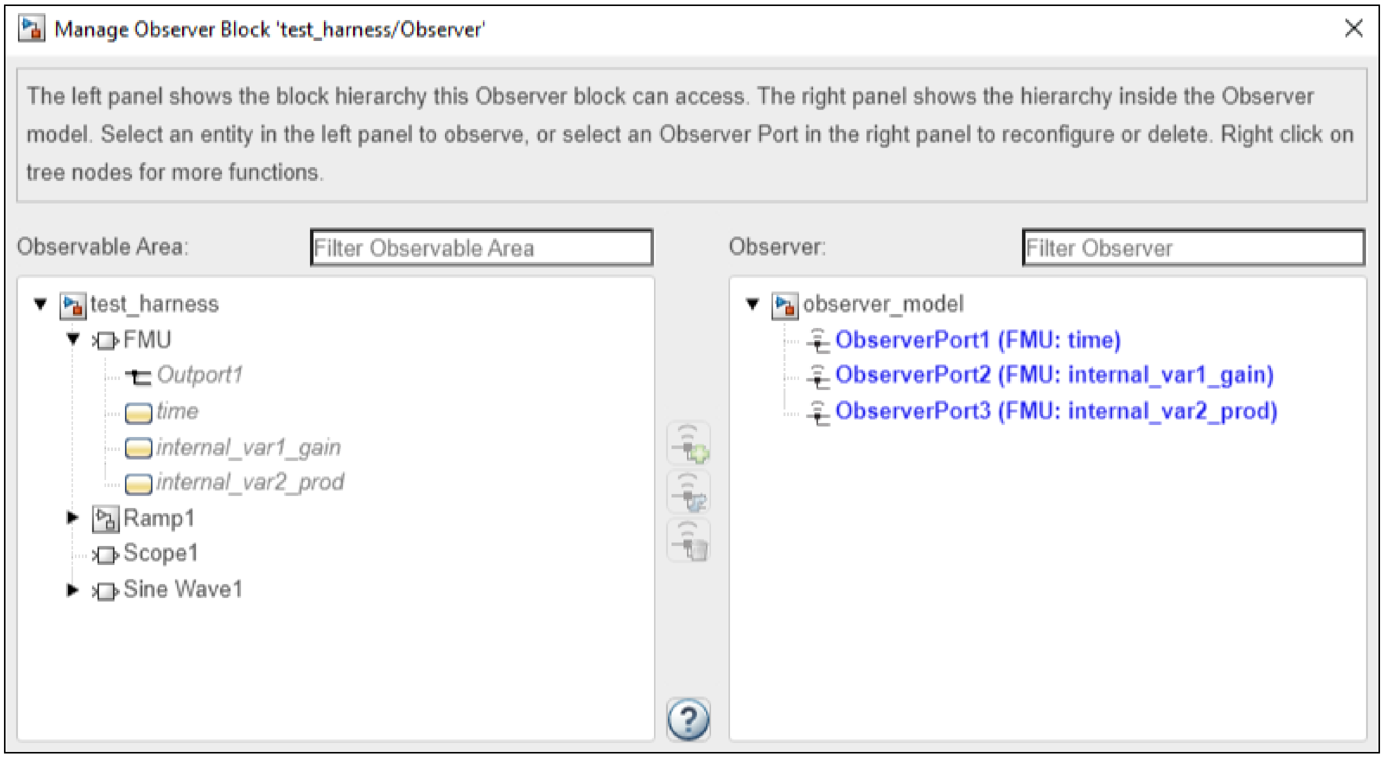
New sltest.testmanager.TestResultContainer results class for tests run using MATLAB Unit Test framework

When you use the MATLAB Unit Test framework `run` or `runInParallel` function to run test cases saved in a Simulink Test MLDATX test file, the `sltest.testmanager.TestCaseResult` or `sltest.testmanager.TestIterationResult` object is saved in an automatically created `sltest.testmanager.TestResultContainer` object. To access the results, use `results.SimulinkTestManagerResults.TestResult`. The new `sltest.testmanager.TestResultContainer` class provides improved porting of results from the Test Manager to the MATLAB Unit Test framework.

Observer support for FMU with internal variables

Starting in R2022b, Observer models can observe internal variables of a Functional Mockup Unit (FMU).

Internal variables of the FMU are visible in the Manage Observer dialog box.



Manage Observer Block 'test_harness/Observer'

The left panel shows the block hierarchy this Observer block can access. The right panel shows the hierarchy inside the Observer model. Select an entity in the left panel to observe, or select an Observer Port in the right panel to reconfigure or delete. Right click on tree nodes for more functions.

Observable Area: Filter Observable Area

- test_harness
  - FMU
    - *Outport1*
    - *time*
    - *internal_var1_gain*
    - *internal_var2_prod*
  - Ramp1
  - Scope1
  - Sine Wave1

Observer: Filter Observer

- observer_model
  - ObserverPort1 (FMU: time)
  - ObserverPort2 (FMU: internal_var1_gain)
  - ObserverPort3 (FMU: internal_var2_prod)

R2022a

Support for ASAM® XIL Standard and third-party test benches

The Simulink Test Support Package for ASAM® XIL Standard implements the ASAM® XIL API, which is a standard that defines communication between test automation tools, such as Simulink Test, and test benches, such as Simulink Real-Time and third-party test benches. The ASAM® XIL API enables running real-time hardware-in-the-loop (HIL), software-in-the-loop (SIL), and model-in-the-loop (MIL) test cases created in Simulink Test using its ASAM® XIL framework. For installation instructions, see Install and Set Up the Simulink Test Support Package for ASAM XIL Standard. For information about using the API see Real-Time Testing with the Simulink Test Support Package for ASAM XIL Standard.

New logical and temporal assessments functions and classes

In R2022a, you can use the new `sltest.testmanager.Assessment` and `sltest.testmanager.AssessmentSymbol` classes and `sltest.testmanager.TestCase` methods to work with logical and temporal assessments.

Using `sltest.testmanager.Assessment`, you can specify the assessment name, obtain the name of the parent test case, control whether the assessment runs when the parent test case runs, and remove the assessment. In addition, you can use these new `sltest.testmanager.TestCase` methods:

- `getAssessments` — Returns an array of `sltest.testmanager.Assessment` objects associated with a test case
- `addAssessment` — Adds an array of `sltest.testmanager.Assessment` objects to a test case
- `getAssessmentSymbols` — Returns an array of `sltest.testmanager.AssessmentSymbol` objects associated with the test case. Each assessment symbol object has these read-only properties: `Name`, `Scope`, and `Value`.
- `addAssessmentSymbol` — Adds an array of `sltest.testmanager.AssessmentSymbol` objects to a test case.

Temporal assessment extension for untested data

You can now extend the results of temporal assessments to reduce previously untested signal data. When you use an `at least`, `at most`, `between`, or `until` temporal assessment, some untested results might occur at the end of the signal. To extend the results of temporal assessments, in the Test Manager, in the **Logical and Temporal Assessments** section, select **Extend Result**. You can extend the assessment results in the **Results and Artifacts** pane by selecting **Extend Assessment Result**. When you extend the results in the Results panel, the test case is not rerun. The Test Manager analyzes and extends the obtained results to potentially reduce the untested results.

Note If you have tests with untested results from previous releases, you can use the new options to potentially reduce the untested results. Extending the result tests more of the signal, so test failures might occur in previously passing tests.

Observer support for messages

In R2022a, Observer models can now observe messages between model components, including SimEvents® components with different sample times. By observing messages in an Observer model, you can avoid including additional ports and blocks in your design model to test the logic. In addition to the signal data, you can obtain this information for a message:

- `time` — Current simulation time
- `id` — Message ID
- `eventType` — Message type, which is one of these values: `MessageDeparture`, `MessageArrival`, `Dropped`, or `Invalid`
- `order` — Order in which the message occurred in the simulation

For more information, see Observe Messages.

Support for batch unit testing using sltest.testmanager.createTestForComponent

`sltest.testmanager.createTestForComponent` now includes support for setting up unit test cases and harnesses for multiple components at the same time. Previously, you could create only one test case and harness at a time.

Add test cases for missing coverage

You can now use the `sltest.testmanager.TestOptions` class and the `sltest.testmanager.addTestsForMissingCoverage` method to add test cases for missing coverage. Use `sltest.testmanager.TestOptions` to define the options for a single test case or iteration, or a test case with multiple iterations. The output is a `TestOptions` object. You then use the `sltest.testmanager.addTestsForMissingCoverage` method to generate or update baseline, equivalence, and simulation tests using the options from the `TestOptions` object. The output is a `TestCase` object, which you can add to a test file or test suite. You can add tests to new or existing test cases for a top-level model. You can also add tests to new or existing test cases for an existing test harness, or create a new test harness for the new test cases. In addition, you can add new test cases to a newly created or existing test file. You must have Simulink Design Verifier to use this feature.

Changed model locking when test harness is open

When a test harness for a model component is open, the main model is no longer locked for editing, saving, or simulation. The locking of the component under test (CUT) in the model and harness differs depending on the **Synchronization Mode**:

- `Synchronize on harness open and close` — The main model and harness block diagram are unlocked. The CUT in the main model is locked. The CUT in the harness is unlocked.
- `Synchronize on harness open` — The main model and harness block diagram are unlocked. The CUT in the main model and in the harness are both locked.
- `Synchronize only during push and rebuild` — The main model and harness block diagram are unlocked. The CUT in the main model and in the harness are both unlocked.
- `Synchronize only during rebuild` — The main model and harness block diagram are unlocked. The CUT in the main model and in the harness are both unlocked.

For more information and limitations, see Synchronize Changes Between Test Harness and Model.

Debug equivalence tests using Model Slicer

You can now use the Model Slicer in the Test Manager to debug equivalence tests. You can also debug tests that compare two simulation modes if one of the modes is set to Normal. For information on using the Model Slicer in the Test Manager, see Debugging Equivalence Test Failures Using Model Slicer.

R2021b

Test assessments in multiple releases

You can now run test cases that contain logical and temporal assessments in multiple releases. You can evaluate assessment results in older releases or compare results across multiple releases.

You can run baseline, simulation, and equivalence tests in multiple releases in R2016b and later releases. For more information, see Assess Temporal Logic in Multiple Releases.

Logical and temporal test assessments example

A new example that shows how to implement and use the test assessment language has been added. See Test Traffic Light Control by Using Logical and Temporal Assessments.

Customize additional default harness creation properties

You can now change most default harness creation property settings. The properties are listed in `sltest.harness.create`. To change the settings, you can create and edit an `sl_customization.m` file and add that file to the MATLAB path. Alternatively, you can use `sltest.harness.setHarnessCreateDefaults` to change the default settings, and `sltest.harness.getHarnessCreateDefaults` to display the updated default settings. For more information, see Customize Properties When Creating Test Harnesses.

Updates to Test Sequence block and test sequence scenarios

The Test Sequence block now supports:

- Strings in test sequences — You can now create and manipulate strings in test sequence steps and transitions. You can also use strings as symbols in the Test Sequence editor. For information, see the "Computation with Strings" section in Manage Textual Information by Using Strings (Stateflow). The "Limitations in Charts that Use MATLAB as the Action Language" section also applies to Test Sequences.
- Active step output as strings — In previous releases, active step output supported only enumeration types and did not support duplicate step names across different scenarios or at different levels in the same scenario. Starting in R2021b, you can use the string data type for active step output, which allows you to have duplicate step names because the step name output now includes its scenario name and location in the scenario hierarchy. To output the active step as a string, set the **Create data to monitor the active step** option in the Model Explorer or Property Inspector.
- Test Sequence scenario reordering — You can now drag and drop test sequence scenarios to change their order, which also updates their indices. Reordering scenarios is useful when you want to run tests in a specific order in a loop that increments by indices.

Remove test results for comparison signals and verify statements

You can now delete individual test results for comparison signals and verify statements using the Test Manager **Results** pane. Removing results that are not of interest reduces the size of the results data set, which affects importing, exporting, and report generation. To prevent changing the overall testing outcome, you can delete tests that have passed or are untested. You cannot delete tests that fail or have unaligned data. Deleting a result also deletes its data display from the Simulation Data Inspector.

Create test harnesses for System Composer components

In R2021b, you can create test harnesses for System Composer™ components in addition to Reference Component blocks. For information, see Verify and Validate Requirements Using Test Harnesses on Components (System Composer).

Functionality being removed or changed

LogHarnessOutputs argument changed to LogOutputs
Warns

The `LogHarnessOutputs` argument of the `sltest.harness.create` function has changed to `LogOutputs`. Currently, the function returns a warning if you use `LogHarnessOutputs`, but in the future, `LogHarnessOutputs` will be removed.

External inputs and initial states no longer copied to test harness
Behavior change

External inputs and initial states are no longer copied from the model configuration parameters of the main model to the test harness created for that model. This change was made to handle two situations:

- If your model uses function calls, the Inport blocks schedule the function calls. In the test harness, the Test Sequence blocks control the scheduling. Because the harness does not have Inport blocks, the number of Inport blocks to the main model does not match the number of inputs to the harness.
- If your main model has bus interfaces with multiple bus element ports, each port is configured in the test harness as a separate input element that corresponds to the bus leaf elements.

The only exception to this change is when the main model uses a configuration set reference. The created test harness refers to the same configuration set reference and creates inputs for each main model Inport. Some of the inputs in the harness might not be needed because the harness uses a Test Sequence block.

If the main model inports and the test harness inputs do not match, you must manually add inputs as needed to the test harness.

Input and output conversion subsystems no longer grouped for variants by default
Behavior change

The **Treat As Grouped when propagating variant conditions** parameter is now set to `off` by default for input and output conversion subsystems in test harnesses. This change allows the test harness to compile even if the subsystem contains an inactive bus signal that is the output from a variant sink. For information, see Propagate Variant Conditions from Variant Source Blocks to Subsystem Blocks.

R2021a

External C/C++ code testing in the Test Manager

A new C/C++ Code Importer wizard for testing C and C++ code functions has been added to the Test Manager. The wizard maps each function to a Simulink C Caller block, saves the blocks in a new library, and creates the test file and test harness. The test file contains test cases for each imported function. To access the wizard, click **New > Test for C/C++ code** in the Test Manager. Corresponding classes and methods are also available for programmatic testing C and C++ code. See `sltest.CodeImporter`.

Collect coverage across multiple releases

You can now compare coverage across two releases in the Test Manager. Coverage results are returned in result sets and you can view both individual and aggregated coverage for the releases. You can export coverage to a MATLAB workspace variable and include it in a generated report. For releases that support coverage filtering, you can automatically add missing coverage to tests generated in the newer release. The earliest release for which multiple release coverage is supported is three years (six releases) prior to the current release. See Collect Coverage in Multiple-Release Tests.

Atomic subsystem unit testing

For atomic subsystems, you can now use a single test harness for equivalence tests that compare normal and software-in-the-loop (SIL) simulation modes when using the Test for Model Component wizard. The code for the top model must be generated using Embedded Coder® before model simulation when running the test. The Test For Model Component wizard can automatically generate these equivalence tests using Simulink Design Verifier. The tests also support collecting model and code coverage. For more information and limitations on atomic subsystem testing, see “Generate Tests and Test Harnesses for a Model or Components”.

Debug verify signal test failures using Model Slicer

You can now use Model Slicer in the Simulink Test Manager to debug failed verify signals. When you simulate the model in debug mode, Model Slicer highlights the model components that impact the failed signals at each time step. You can move between failure regions to debug the cause of the verification failures. See Debugging Test Failures Using Model Slicer.

Additional support for observers

Observers now support:

- Simulink `string` data type.
- Simulation Stepper, which enables stepping back and forward in a running simulation. The Simulation Stepper is useful for debugging unexpected simulation results. The Simulation Stepper options are on the design model toolstrip. When you select the **Step Back** check box, the observer model steps back as much as is needed to stay synchronized with the step back activity of the design model.
- `OperatingPoint` objects (see Operating Point Behavior).
- Coverage collection on observer models and models referenced by observer models. When you set the coverage scope to observer models, the model reference selection dialog now lists observer

models in addition to model references. Observer models are distinguished from model references by an **(Observer)** tag.

- Simulink Design Verifier analysis by using Observer Reference blocks, which now replace verification subsystems. For more information, see Support observer reference block in Simulink Design Verifier analysis (Simulink Design Verifier).

Customize properties when creating test harnesses

You can now change some default property settings to use customized defaults when creating a test harness. To change the settings, you use an `sl_customization.m` file that contains a structure, and add that file to the MATLAB path. The properties you can set using the `sl_customization` file are:

- `PostCreateCallback` — Specifies the callback script that is executed after the test harness is created.
- `SaveExternally` — Specifies whether the test harness is saved as an internal or external harness.
- `LogOutputs` — Specifies whether to log all outputs for the component under test of a test harness.

For information, see Customize Properties When Creating Test Harnesses.

Test sequence scenarios in iterations

In the Test Manager, you can now use test sequence scenarios, created in Test Sequence blocks, in test case iterations. Using iterations lets you run multiple scenarios in a single test case. In the Test Manager **Inputs** section, you can select the Test Sequence block that contains the scenarios and the default scenario for the iterations. Then, in the **Iterations** section, you can change the default scenario for each iteration. See Use Test Sequence Scenarios in the Test Sequence Editor and Test Manager

Updates to Create Test for Model Component wizard

The following changes have been made to the Create Test for Model Component wizard:

- Atomic subsystem equivalence (back-to-back) testing support — Create tests to compare normal and software-in-the-loop (SIL) simulations. See “Atomic subsystem unit testing” on page 5-2.
- Enhanced MCDC — Use enhanced MCDC to improve test case generation for equivalence (back-to-back) tests. A new check box is available only when software-in-the-loop (SIL) or processor-in-the-loop (PIL) is the second simulation and when you select Simulink Design Verifier as the test input. This option is on the **Verification Strategy** tab of the wizard. For more information on Enhanced MCDC, see Enhanced MCDC Coverage in Simulink Design Verifier (Simulink Design Verifier).
- Signal Editor input source — Use Signal Editor as an input source for test harnesses for simulation tests. Previously, the Signal Editor was available as an option only when selecting Simulink Design Verifier in the wizard.

Simulation mode iterations in a single test case

You can now use the Test Manager to iterate simulation modes (such as, normal, SIL, accelerator, rapid accelerator) for all types of test cases. A new **Simulation Modes** check box has been added to the Auto Generate Iterations dialog box.

Output signal logging option for test harness creation

You can now turn on signal logging for all outputs from the component under test when you create its test harness. Previously, you had to enable logging manually for each output signal of interest after you created the test harness. Now, when you right-click a block to add a test harness, the **Advanced Properties** tab of the Create Test Harness dialog box includes a new **Log Output Signals** check box. This check box is on the **Basic Properties** tab of the Create Test Harness dialog box for library and subsystem harnesses. For programmatic harness creation, a new `LogHarnessOutputs` name-value pair has been added to `sltest.harness.create`.

Include test harnesses when converting subsystem to model reference

Now, if you convert a subsystem or a subsystem reference to a model reference, test harnesses in the subsystem are saved in the model reference. Previously, if a subsystem had a test harness defined for the whole subsystem, you could not convert that subsystem to a model reference. For subsystems with test harnesses for blocks, those test harnesses were deleted when the subsystem was converted to a model reference.

Simulation progress indicator updates

The number of simulation progress messages shown in real-time during test execution has been increased to provide more information about the current simulation phase. The progress messages include Autosave, Compiling, and the Simulation Progress percentage. For tests with iterations, progress messages are shown for each iteration.

Simulation log messages at the command line

You can now print simulation log messages to the MATLAB command line. To turn on this option, use `sltest.testmanager.setpref('ShowSimulationLogs','IncludeOnCommandPrompt',true)`. To view the current preference setting, use `sltest.testmanager.getpref('ShowSimulationLogs','IncludeOnCommandPrompt')`. For more information, see `sltest.testmanager.setpref` and `sltest.testmanager.getpref`, respectively.

Enhanced test harness support for MATLAB-based Simulink tests

Test harness support has been enhanced for MATLAB-based Simulink tests. Use `sltest.testCase.createSimulationInput(<model>,'WithHarness',<harness>)` to create a new `sltest.harness.SimulationInput` object that integrates input to test harnesses used in the Test Manager.

Additionally, support for adding coverage and pushing test results to the Test Manager have been added to the MATLAB test runner. See `addModelCoverage` and `addSimulinkTestResults`.

Run with Stepper support at test suite and test file levels

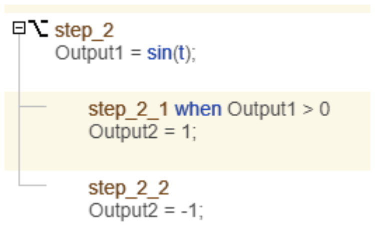
You can now use **Run with Stepper** in the Test Manager to run steps interactively for test suites and test files, in addition to using the stepper for individual test cases.

Functionality being removed or changed

Old semantics for Test Sequence blocks will be removed
Warns

The legacy semantics that Test Sequence blocks currently use will be replaced by semantics that are compatible with Stateflow®. As a result of this change, you might see different Simulink Coverage™ and Simulink Design Verifier results. In most cases, this change will not affect the behavior of Test Sequence blocks. However, if a Test Sequence block uses a symbol that is read in a transition or `when` condition, and is written in the parent step (or parent of the parent step) of that transition or `when` condition, the behavior might be different. For legacy semantics, the transition condition of the child step is evaluated before the parent step. For semantics compatible with Stateflow, the transition condition of the child step is evaluated after executing the parent step.

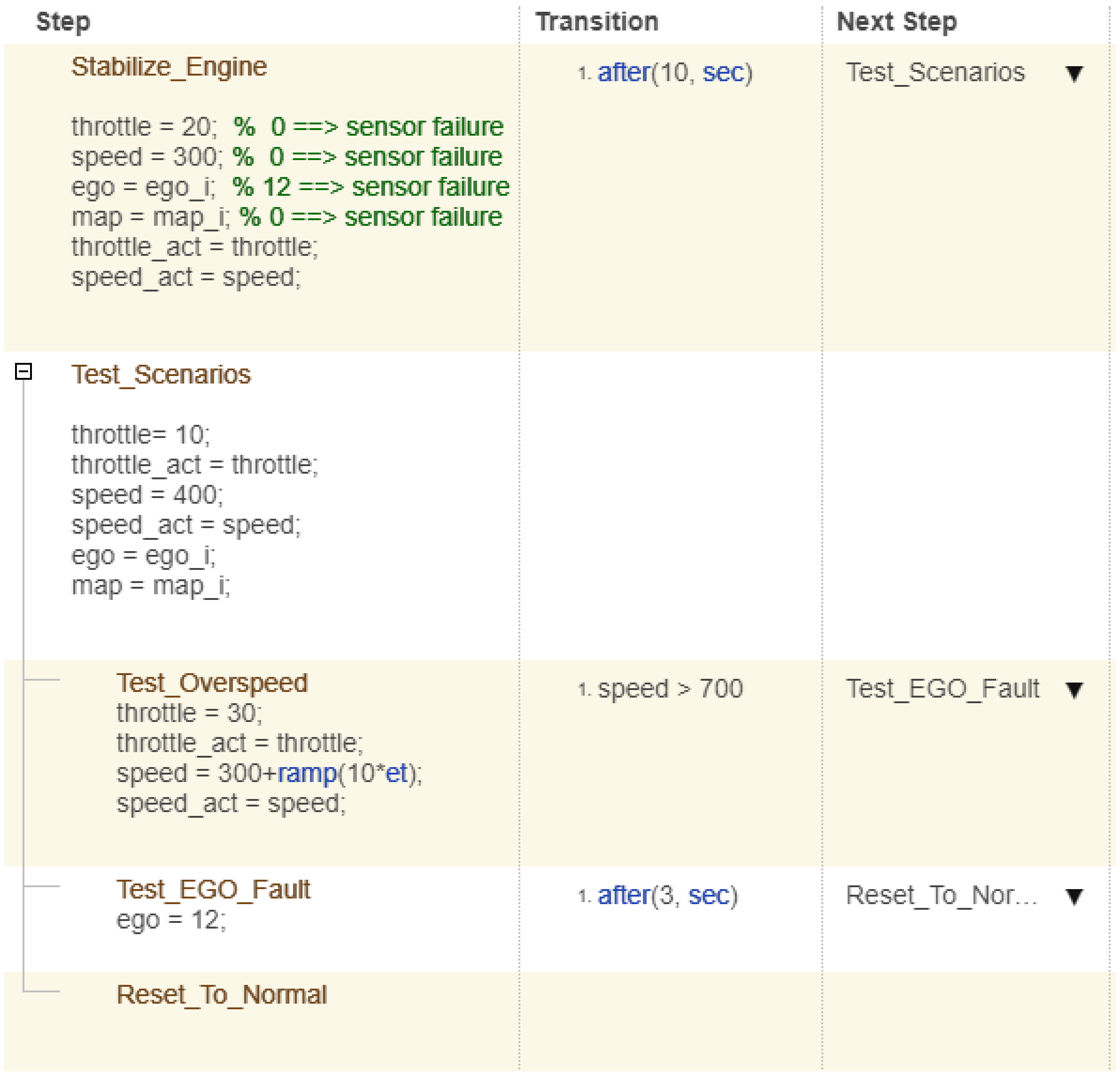
This step from a Test Sequence shows the legacy case where the `Output1` symbol is written in parent **step_2** and read in the `when` condition of child **step_2_1**. When you load or compile a model with blocks that use legacy semantics, if this pattern is not detected, then those blocks are updated automatically to use semantics compatible with Stateflow. If the pattern is found, a warning is displayed and those blocks with the detected pattern are not updated.


```
step_2
Output1 = sin(t);

    step_2_1 when Output1 > 0
    Output2 = 1;

    step_2_2
    Output2 = -1;
```

The examples show how to update Test Sequence blocks that use legacy semantics so they behave the same and produce the same simulation results as semantics compatible with Stateflow blocks. In most cases, you can update the blocks by moving code from the parent step to the child step.

This example shows how to update the Test Sequence block by moving the `throttle`, `throttle_act`, `speed`, and `speed_act` writes from the **Test_Scenarios** parent step to the **Test_Overspeed** and **Test_EGO_Fault** child steps.

This image shows the legacy semantics.



Step	Transition	Next Step

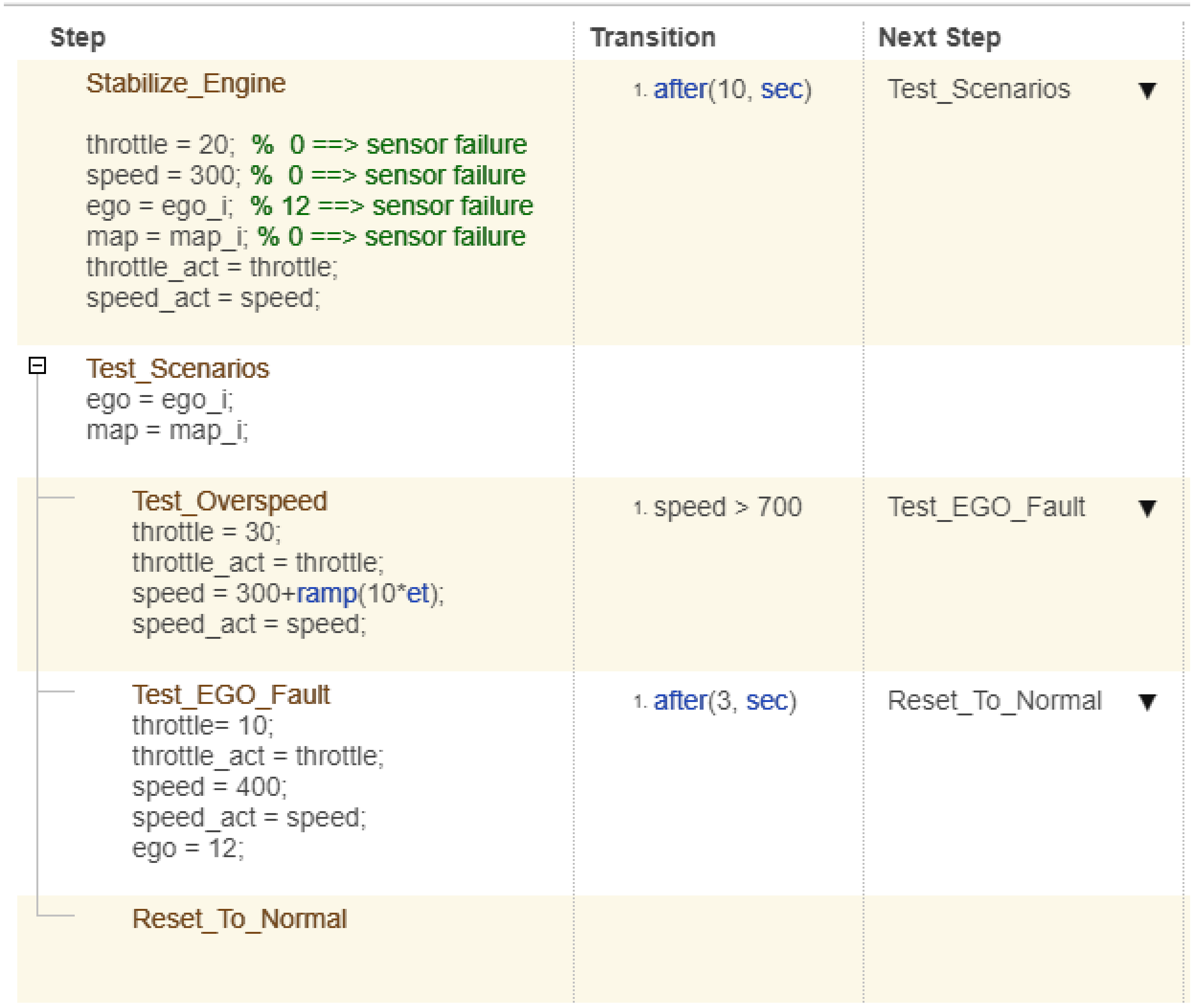
Stabilize_Engine
 	
throttle = 20; % 0 ==> sensor failure
speed = 300; % 0 ==> sensor failure
ego = ego_i; % 12 ==> sensor failure
map = map_i; % 0 ==> sensor failure
throttle_act = throttle;
speed_act = speed;
	1. after(10, sec)	Test_Scenarios ▼

⊟ Test_Scenarios
 	
throttle= 10;
throttle_act = throttle;
speed = 400;
speed_act = speed;
ego = ego_i;
map = map_i;		

Test_Overspeed
throttle = 30;
throttle_act = throttle;
speed = 300+ramp(10*et);
speed_act = speed;
	1. speed > 700	Test_EGO_Fault ▼

Test_EGO_Fault ego = 12;	1. after(3, sec)	Reset_To_Nor… ▼
Reset_To_Normal		

This image shows the updated semantics that are compatible with Stateflow.



Step	Transition	Next Step

Stabilize_Engine
 	
throttle = 20; % 0 ==> sensor failure
speed = 300; % 0 ==> sensor failure
ego = ego_i; % 12 ==> sensor failure
map = map_i; % 0 ==> sensor failure
throttle_act = throttle;
speed_act = speed;
	1. after(10, sec)	Test_Scenarios ▼

⊟ Test_Scenarios
ego = ego_i;
map = map_i;		

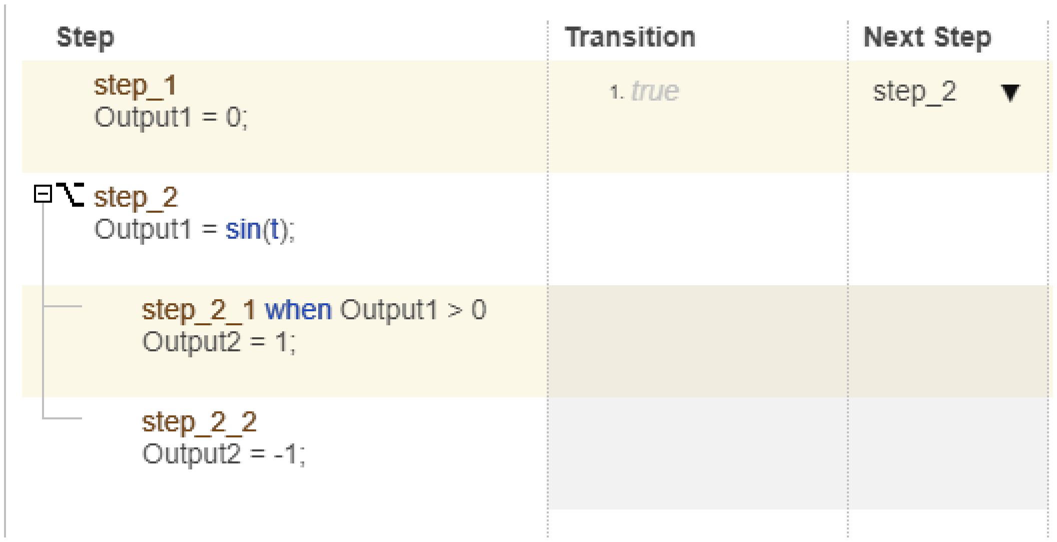
Test_Overspeed
throttle = 30;
throttle_act = throttle;
speed = 300+ramp(10*et);
speed_act = speed;
	1. speed > 700	Test_EGO_Fault ▼

Test_EGO_Fault
throttle= 10;
throttle_act = throttle;
speed = 400;
speed_act = speed;
ego = 12;
	1. after(3, sec)	Reset_To_Normal ▼

Reset_To_Normal		

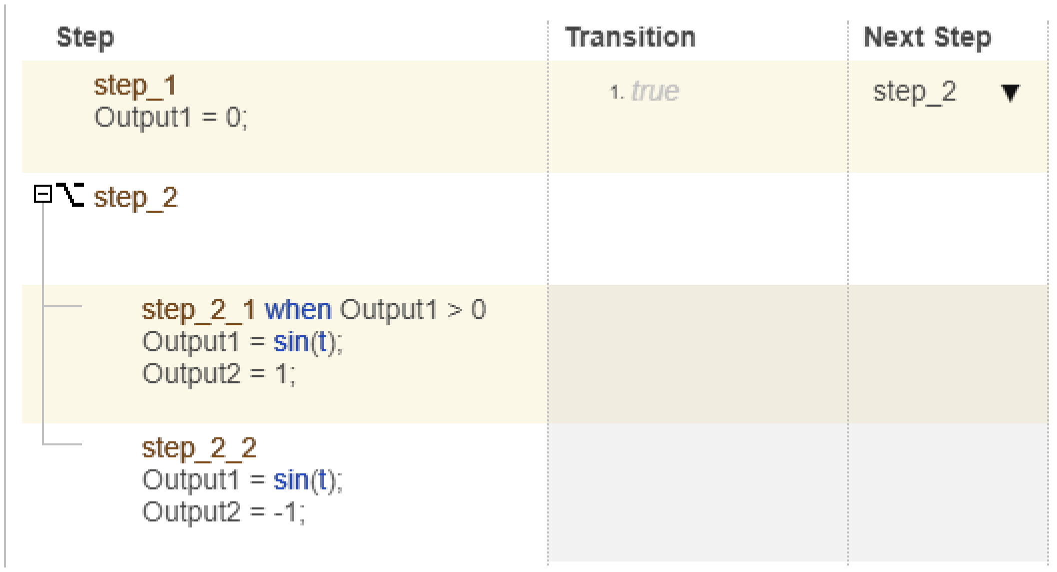
This example shows how to update the Test Sequence block by moving the `Output1` write from the **step_2** parent step to the **step_2_1** and **step_2_2** child steps.

This image shows the legacy semantics.



Step	Transition	Next Step
step_1 Output1 = 0;	1. true	step_2 ▼
step_2 Output1 = sin(t);		
step_2_1 when Output1 > 0 Output2 = 1;		
step_2_2 Output2 = -1;		

This image shows the updated semantics that are compatible with Stateflow.



Step	Transition	Next Step
step_1 Output1 = 0;	1. true	step_2 ▼
step_2		

step_2_1 when Output1 > 0
Output1 = sin(t);
Output2 = 1;		

step_2_2
Output1 = sin(t);
Output2 = -1;		

After you update a block, use `sltest.testsequence.setProperty(<blockpath>,'Semantics','StateflowCompatible')` to save the semantics setting. Changing the setting prevents future warnings about the semantics for that block.

R2020b

Multiple scenarios in Test Sequence block

You can now define more than one test sequence scenario in a single Test Sequence block by clicking **Use Scenarios**. Each scenario has a unique name and is in a separate tab in the editor. You can add, edit, delete, and copy scenarios. Before running the model, select the scenario to enable for that run. Previously, the Test Sequence block defined only a single scenario. This limitation required using multiple Test Sequence blocks to define multiple scenarios for testing a single model or harness.

Programmatically, you can use these functions for test scenarios:

- `sltest.testsequence.useScenario`
- `sltest.testsequence.isUsingScenarios`
- `sltest.testsequence.addScenario`
- `sltest.testsequence.deleteScenario`
- `sltest.testsequence.editScenario`
- `sltest.testsequence.activateScenario`
- `sltest.testsequence.getActiveScenario`
- `sltest.testsequence.getAllScenarios`
- `sltest.testsequence.setScenarioControlSource`
- `sltest.testsequence.getScenarioControlSource`
- `sltest.testsequence.getActiveScenario`

Parallel test execution on a remote cluster

You can now run tests in parallel on a remote cluster, such as in the cloud. MATLAB Parallel Server™ is required to use this feature. See Run Tests Using Parallel Execution and Running Code on Clusters and Clouds (MATLAB Parallel Server).

Test failure debugging using Model Slicer

You can now use the Model Slicer in the Test Manager to debug a failed signal in a baseline test failure. When you simulate the model in debug mode, the model components that impact the failed signal are highlighted for each time step. You can move between failure regions to debug the cause of the baseline differences. To use this feature, you must have a Simulink Check™ license. See Debugging Baseline Test Failures Using Model Slicer.

Multi-release equivalence tests

You can now run each simulation of an equivalence (back-to-back) test case on a different MATLAB release. The results for both releases are combined into a single results set. This feature lets you run, for example, the normal mode simulation in an earlier MATLAB release and the SIL mode simulation in a later release. See Run Tests in Multiple Releases of MATLAB for more information.

Test case reordering

In the Test Browser pane of the Test Manager, you can drag and drop test cases to change the order in which the cases run during a test and appear in a test report. Alternatively, you can right-click on a test case and select **Move Up** or **Move Down**.

MATLAB-based Simulink tests

You can now write baseline and equivalence tests for Simulink models in MATLAB test files (`.m`), and then open and run the files in the Test Manager. You can also create a new MATLAB-based Simulink test from the Test Manager. MATLAB test files give you a readable view of the test and let you compare or merge tests from different MATLAB program files. They also allow you to use the Test Manager to create a single results set for the whole test file, view and link requirements to MATLAB test files and test cases, view aggregated coverage results, and generate test reports.

For more information, see Test Models Using MATLAB-Based Simulink Tests. For examples, see Using MATLAB-Based Simulink Tests in the Test Manager and Collect Coverage Using MATLAB-Based Simulink Tests.

Generate tests for models and reuse test files using Test for Model Component wizard

You can now use the Test for Model Component wizard to generate tests for top-level models. If you specify only a top model, you can optionally create a test harness for that model. If you choose to create a test harness, you can specify the inputs and define the verification logic for the test harness. You can also choose whether to create the tests in an existing test file or create a new test file.

Alternatively, you can use the API to create tests for top-level models. The new `sltest.testmanager.createTestForComponent` function is recommended instead of the `sltest.testmanager.createTestForSubsystem` function.

sltest.testmanager.TestFile.createTestForSubsystem and sltest.testmanager.TestSuite.createTestForSubsystem are not recommended
Still runs

`sltest.testmanager.TestFile.createTestForSubsystem` and `sltest.testmanager.TestSuite.createTestForSubsystem` are not recommended. Use `sltest.testmanager.createTestForComponent` instead. This replacement function allows you to create test cases for models, in addition to subsystems. In addition to the name-value pairs in the `createTestForSubsytem` functions, four additional name-value pairs are included in `createTestForComponent`:

- `TestFile` — Required name-value pair to specify the test file for the created test case
- `Component` — Required name-value pair to specify the component or model to test
- `CreateTestFile` — Optional name-value pair for whether to create a new test file
- `CreateHarness` — Optional name-value pair for whether to create a test harness in addition to the test case

Model reference parameter overrides

You can override parameters within model reference workspaces. All model element parameters that can be overridden, including those in Model blocks, appear in the **Parameter Overrides** section of the Test Manager. You can also use `addParameterOverride` to override a parameter in a specified workspace. If you import your parameters from or export them to an Excel spreadsheet, the BlockPath column in the spreadsheet now supports model reference workspace names.

Reusable library subsystem SIL testing

You can now conduct equivalence (back-to-back) testing on a reusable library subsystem by using software-in-the-loop (SIL) or processor-in-the-loop (PIL) simulation. This feature allows you to:

- Unit test the generated code for a library subsystem
- Test an instance of reusable library subsystem in a model
- Obtain code coverage results for the library subsystem

To enable this testing, the component under test must be or must contain a reusable library subsystem, and the subsystem must have function interfaces. If both of these conditions are met, when you use the Test Manager or the Test for Model Component wizard to create a test harness, you can now specify the function interface. Alternatively, when a test harness for a function interface is open, you can use the SIL/PIL app to create an equivalence test case and export it to the Test Manager. For programmatic use, both `sltest.harness.create` and `sltest.harness.set` have a `FunctionInterfaceName` property.

For an example of how to perform a SIL test on a reusable subsystem library, see Test Library Blocks.

Conditional breakpoints in Test Sequence block

In addition to breakpoints that always pause a test, you can now add breakpoints that pause a Test Sequence step or transition only when a specified condition is met. You can also convert normal breakpoints to conditional ones and conditional breakpoints to normal ones. For information on breakpoints, see Debug a Test Sequence.

Assessment callback and t (time) symbol

The **Logical and Temporal Assessments** section of the Test Manager has a new **Assessment Callback** section. In this section, you can write a MATLAB script to define variables and map symbols to those variables. The script is saved as part of the test file. In addition, `t` is now a symbol that is automatically mapped to simulation time and can be used in assessment conditions and expressions. For example, to limit an assessment to a time between 5 and 7, the expression is `(t > 5 & t < 7)`.

Test harness locking and model synchronization

The `EnableComponentEditing` name-value pair and `lockMode` property have been removed from `sltest.harness.create`. These parameters controlled the locking behavior of the test harness and component under test. Instead, use the `SynchronizatonMode` property to control harness and component locking and synchronization. See Synchronize Changes Between Test Harness and Model for more information.

Compatibility Considerations

Use the `SynchronizationMode` parameter instead of `EnableComponentEditing` and `lockMode` in `sltest.harness.create` to control the locking and syncing behavior of the test harness and model.

Test harness metadata storage anywhere on MATLAB path

You can move the `harnessInfo` metadata file for an external test harness to anywhere on the MATLAB path. The file links the model and its test harnesses and is created at the same time the harness is created. Previously, this file had to be stored in the same folder as the model. Being able to move the harness metadata file allows you to separate your model and test artifacts.

Baseline data file updates for Excel files

You can now update that baseline test data stored in an Excel file without having to rerun the model to recapture the baseline. You can update all signals, a specific signal, or a specific region of a signal. You update Excel baseline files the same way you update MAT-files. See Examine Test Failures and Modify Baselines for more information.

Bus element logging

The signal logging override interface has been extended to allow you to specify which leaf elements of a bus signal to include in the results. Including specific elements, rather than the entire bus, reduces memory usage when executing tests. It also improves the performance of generating reports and exporting results from the Test Manager. The specified leaf elements appear as individual rows in the signal set table of the Test Manager **Logged Signals** section. You can also specify a plot index to plot a leaf signal on a specific subplot automatically upon execution.

See the Logging Leaf Signals of a Bus section in Capture Simulation Data in a Test Case for more information.

Model Testing Dashboard

The Model Testing Dashboard collects and displays metric data on the status and quality of your requirements-based testing. Assess the testing status of a model by using the dashboard to view:

- Summary data on requirements, tests, and traceability between requirements and tests
- Status and results for the latest test runs
- Model coverage measurements achieved by tests and justifications
- A list of the latest artifacts in the project, organized by the associated models

For more information, see “Model Testing Dashboard: Track completeness of requirements-based testing for compliance to standards such as ISO 26262” (Simulink Check). To use this feature, you must have a Simulink Check license.

Functionality being removed or changed

sltest.testmanager.TestFile.createTestForSubsystem and sltest.testmanager.TestSuite.createTestForSubsystem are not recommended

Still runs

`sltest.testmanager.TestFile.createTestForSubsystem` and `sltest.testmanager.TestSuite.createTestForSubsystem` are not recommended. Use `sltest.testmanager.createTestForComponent` instead. This replacement function allows you to create test cases for models, in addition to subsystems. In addition to the name-value pairs in the `createTestForSubsytem` functions, four additional name-value pairs are included in `createTestForComponent`:

- `TestFile` — Required name-value pair to specify the test file for the created test case
- `Component` — Required name-value pair to specify the component or model to test
- `CreateTestFile` — Optional name-value pair for whether to create a new test file
- `CreateHarness` — Optional name-value pair for whether to create a test harness in addition to the test case

Removed real-time test case support for parameter overrides in model references

Errors

Support for overriding parameters in model references for real-time test cases has been removed.

To prevent failures in real-time test cases that override parameters in model references, make the model reference parameters accessible as model arguments (see Parameterize Instances of a Reusable Referenced Model (Simulink)). After changing the accessibility, remove the parameters from the Test Manager **Parameter Overrides** section and replace them with the corresponding model arguments. If you do not update the parameters, an error occurs when you run the test. You can then click **Refresh Parameters** in the Test Manager **Parameter Overrides** section to see which parameters are not found in the model.

Removed real-time test case support for logging output ports and states

Errors

Support for logging output ports and states for real-time test cases has been removed. The **Override model settings** section in the Test Manager **Simulation Outputs** section has also been removed for real-time tests.

To prevent failures in real-time test cases that log output ports, mark the output signal for logging either in the model (see Configure a Signal for Logging (Simulink)) or by using the **Logged Signals** table in the Test Manager **Simulation Outputs** section (see Capture Simulation Data in a Test Case).

States set for logging in real-time tests are ignored, so you do not need to update your tests.

R2020a

Generated code reuse for SIL/PIL test harness generation

When creating a test harness, you can now reuse existing code to generate a SIL/PIL block. This code reuse allows you to use the Test Manager to verify generated code from a previous release without having to regenerate that code. This feature speeds up SIL/PIL test harness creation. See Use SIL/PIL to Verify Generated Code from an Earlier Release.

Test coverage enhancements in the Test Manager

In the Test Manager, the following improvements have been made for test coverage. You must have a Simulink Coverage license to use these new features.

- Multiple test coverage filters can be applied simultaneously
- Models with updated test coverage filters do not require you to simulate the model again to see updated coverage results
- Coverage results show the filters that were applied
- Colors used for coverage reporting match the colors used in Simulink Coverage

See Collect Coverage in Tests and Test Coverage for Requirements-Based Testing

Use Stateflow chart as test harness source and scheduler

You can now use Stateflow charts as test harness sources and schedulers. Using a chart as a test source or scheduler allows support for MATLAB functions, supertransitions, and super steps. See Use Stateflow Chart for Test Harness Inputs and Scheduling.

verify keyword support in Stateflow charts

In addition to supporting the verify keyword in Test Sequence blocks, Simulink Test now supports `verify` in Stateflow charts. The `verify` keyword assesses a logical expression and is useful for state-based testing.

Observer support for variable solvers and Stateflow states and variables

Observers allow you to view dynamic information during test simulations. You can now:

- Observe Stateflow states and local variables.
- Add observers to models that use variable-step solvers
- Add and manage observers using the Wireless Observers section of the Tests tab

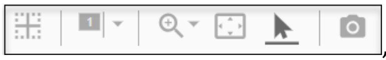
See Access Model Data Wirelessly by Using Observers.

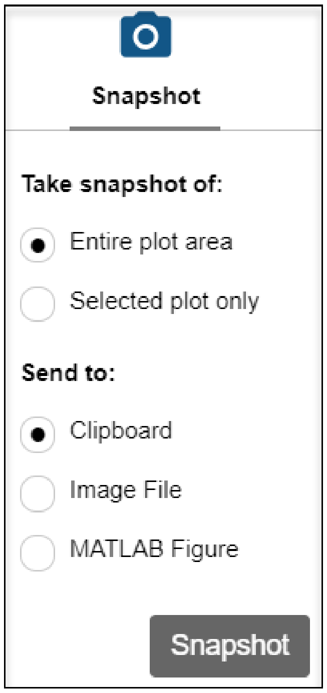
Test harness support for Subsystem Reference blocks

Simulink Test now supports creating test harnesses for Subsystem models and Subsystem Reference blocks in a model. You create these harnesses in the same way that you create other harnesses. For information about Subsystem Reference blocks and models, see “Subsystem Reference: Save subsystem contents in a file that you can reference in a model” in the Simulink Release Notes.

Results viewer enhancements in the Test Manager

These Simulation Data Inspector (SDI) features have been incorporated into the Test Manager:

- The SDI toolbar , which is located in the upper right of the Test Manager results viewer. The toolbar includes subplots, data cursors, zoom, pan, select, fit to view, and snapshot.
- Take a snapshot of the entire plot area or of a selected plot only.
- Save a snapshot to the clipboard, as an image file or as a MATLAB figure.



Snapshot

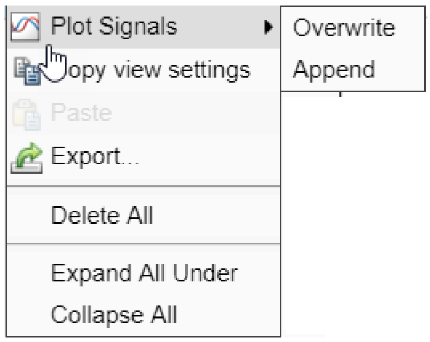
Take snapshot of:

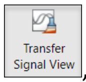
- [✓ Entire plot area]
- Selected plot only

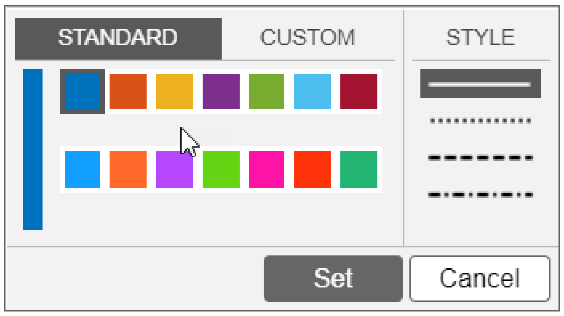
Send to:

- [✓ Clipboard]
- Image File
- MATLAB Figure

- Preserve signal selection and plot view when you rerun a test. If you rerun a test for the same model, the signals plotted from the previous run are cleared and the results from the latest run are plotted using the same plot layout and signal selection. Right-click another Sim Output to and select `Overwrite` to replace the latest output plots with the selected outputs. To append the selected Sim Output results to the currently displayed results, click `Append`.



- Copy a plot view layout and the signal selection from one Sim Output to another Sim Output in the Results and Artifacts pane. Select the Sim Output to copy from. Then, click Transfer Signal View , and select the Sim Output to copy to.
- Change signal colors and line styles. Click on a signal line representation under a Sim Output to open the dialog.



Updated execution order for test case callbacks

When multiple test cases are run at the same time for different models, the order in which the callbacks for these test cases are executed is different than in previous releases. In previous releases, all actions for the first test case executed, and then all actions for subsequent test cases. The new execution order preloads each test case and loads its model. Then the first test case post-loads, its model is simulated, and clean up is done. The following test cases then post-load, simulate, and clean up. All the models are closed at the end of the entire test execution.

In previous releases, when a test suite ran multiple test cases on the same model, the model was closed after each test case and then reopened for subsequent test cases. In the new execution order, the model is kept open between test executions. As a result, code in the `PreLoadFcn` callback for the model is executed only when the model is opened for the first test case. The `Pre-Load` callback is not executed for subsequent test cases because the model remains open.

The following table shows the old and new execution order for two test cases run on separate models. The models are not open before the test suite runs.

Old Execution Order	New Execution Order
Run test case 1 `Pre-Load` callback. Then, run model `PreLoadFcn` callback.	Run test case 1 `Pre-Load` callback. Then run model 1 `PreLoadFcn` callback.

Old Execution Order	New Execution Order
Load model 1.	Load model 1.
Run model 1 `PostLoadFcn` callback. Then, run test case 1 `Post-Load` callback.	Run model 1 `PostLoadFcn` callback.
Simulate model 1 for test case 1.	Run test case 2 `Pre-Load` callback. Then run model 2 `PreLoadFcn` callback.
Run test case 1 `Cleanup` callback.	Load model 2.
Run model 1 `CloseFcn` callback.	Run model 2 `PostLoadFcn` callback. Then run test case 1 `Post-Load` callback.
Run test case 2 `Pre-Load` callback. Then run model `PreLoadFcn` callback.	Simulate model 1 for test case 1.
Load model 2.	Run test case 1 `Cleanup` callback.
Run model 2 `PostLoadFcn` callback. Then, run test case 2 `Post-Load` callback.	Run test case 2 `Post-Load` callback.
Simulate model 2 for test case 2.	Simulate model 2 for test case 2.
Run test case 2 `Cleanup` callback.	Run test case 2 `Cleanup` callback.
Run model 2 `CloseFcn` callback.	Run model 2 `CloseFcn` callback.
	Run model 1 `CloseFcn` callback.

The following table shows the old and new execution orders for test cases run on a single model for a test suite with two test cases. The model is not open before the test suite runs.

Old Execution Order	New Execution Order
Run test case 1 `Pre-Load` callback. Then run model `PreLoadFcn` callback.	Run test case 1 `Pre-Load` callback. Then run model `PreLoadFcn` callback.
Load model.	Load model.
Run model `PostLoadFcn` callback. Then run test case 1 `Post-Load` callback.	Run model `PostLoadFcn` callback.
Simulate model.	Run test case 2 `Pre-Load` callback.
Run test case 1 `Cleanup` callback.	Run test case 1 `Post-Load` callback
Run model `CloseFcn`.	Simulate model.
Run test case 2 `Pre-Load` callback. Then run model `PreLoadFcn` callback.	Run test case 1 `Cleanup` callback.
Load model.	Run test case 2 `Post-Load` callback.
Run model `PostLoadFcn` callback. Then run test case 2 `Post-Load` callback.	Simulate model.
Simulate model.	Run test case 2 `Cleanup` callback.
Run test case 2 `Cleanup` callback.	Run model `CloseFcn` callback.

Old Execution Order	New Execution Order
Run model `CloseFcn` callback.	

The following table shows the old and new execution orders for test cases run on a single model for a test suite with two test cases. The model is open before the test suite runs.

Notice that the model `PreLoadFcn` and `PostLoadFcn` callbacks do not execute because the model is already loaded before the test runs. The model `CloseFcn` callback does not execute either because the model is left open after test completion.

Old Execution Order	New Execution Order
Run test case 1 `Pre-Load` callback.	Run test case 1 `Pre-Load` callback.
Run test case 1 `Post-Load` callback.	Run test case 2 `Pre-Load` callback.
Simulate model.	Run test case 1 `Post-Load` callback
Run test case 1 `Cleanup` callback.	Simulate model.
Run test case 2 `Pre-Load` callback.	Run test case 1 `Cleanup` callback.
Run test case 2 `Post-Load` callback.	Run test case 2 `Post-Load` callback.
Simulate model.	Simulate model.
Run test case 2 `Cleanup` callback.	Run test case 2 `Cleanup` callback.

Compatibility Considerations

If you have code in a model `PreLoadFcn`, `PostLoadFcn`, or `CloseFcn` callback, you might see different behavior or unexpected errors for your tests due to the changes of when models open and close. You might need to change the code in the callbacks. For example, for a script defined in the model `PreLoadFcn` callback, if you see an `Undefined function or variable` error, the issue might be resolved by moving the script from the `PreLoadFcn` callback to the `PostLoadFcn` callback.

If you have code in multiple `Pre-Load` callbacks for a test case, those functions might overwrite or conflict with each other because all of the test case `Pre-Load` callbacks run before any of the models simulate. When setting up test cases, do not use `Pre-Load` callbacks. Instead, use `Post-Load` callbacks.

R2019b

Guided Back-to-Back Testing Wizard: Use guided workflow to set up back-to-back equivalence and baseline testing

Back-to-back testing compares the simulation outputs from different simulation modes, such as Simulink normal simulation mode and SIL mode. The Test Manager now includes guided test authoring for a model component.

The new Test for Model Component wizard helps you set up back-to-back equivalence and baseline testing. It replaces the Test for Subsystem option. The wizard automatically creates the required test harness or harnesses based on your selections. In the wizard dialog boxes, you select the:

- Simulink model component to test
- Source of the test inputs (from the model simulation or from Simulink Design Verifier)
- Type of testing to perform
- Format for saving the test data

See Generate Tests for a Component and Create and Run a Back-to-Back Test.

SIL/PIL Equivalence Testing: Create equivalence test cases from the SIL/PIL Manager in Embedded Coder

The SIL/PIL Manager in Embedded Coder runs a model in normal mode and SIL/PIL simulation mode so you can compare the results. To export a SIL/PIL automated verification test case to the Test Manager, configure the test case, and then use the new **Export to Test Manager** button. The Test Manager creates an equivalence test case that is populated with settings from the SIL/PIL Manager. See Import Test Cases for Equivalence Testing.

You must have both Simulink Test and Embedded Coder to use this feature.

Customized Test Reports: Create test specification reports from a test file, test suite, or test case

The new `sltest.testmanager.TestSpecReport` class allows you to create a test specification report from a test file, test suite, or test case.

If you have MATLAB Report Generator™ and Simulink Report Generator, you can create custom report templates and reporters for a test case or test suite reporter. To rearrange items or change the fonts, indentation, or other features of a report, create custom report templates by using the `createTemplate` method of `sltest.testmanager.TestCaseReporter` or `sltest.testmanager.TestSuiteReporter`. To add new types of data to a report, create a custom reporter by using the `customizeReporter` method.

Test Result Reports: Multiple plots per page

The test result report now supports more than one plot per page. You can specify up to four rows and four columns of plots on a single page.

To specify the number of plots using the API, set the `NumPlotRowsPerPage` and `NumPlotColumnsPerPage` properties in `sltest.testmanager.Options` or `sltest.testmanager.report`.

In the Test Manager under Test Files Options, use the Create Test Result Report dialog box. To specify the number of rows and columns, first select **Plots for simulation output and baseline** in the dialog box.

Test Assessments: Display results in the Simulation Data Inspector

You can plot `sltest.Assessment` result data programmatically by using its new `plot` function. This function automatically opens and displays the results in the Simulation Data Inspector.

Simulink Toolstrip: Contextual tabs for Simulink Test

The Simulink Toolstrip now includes contextual tabs that appear only when you need them. When you click **Simulink Test** in the **Apps** tab, the **Tests** tab appears in the toolbar. The items displayed in the **Tests** tab depend on the context. For more information, see Simulink Editor - Simulink Toolstrip in the Simulink Release Notes.

R2019a

Temporal Requirements Verification: Assess model timing and signal behavior using a form-based editor

Hybrid systems with discrete- and continuous-time behavior can require complex timing-dependent signal logic. To verify this logic, you can create temporal assessments using the new **Logical and Temporal Assessments** editor in the Test Manager.

The editor helps you translate your textual requirements into unambiguous assessments with clear, well-defined semantics. To create a temporal assessment, you use a form-based editor that prompts you for particular conditions, events, signal values, delays, and responses that make up the assessment. For increased readability, the editor summarizes the assessment in a language-like statement.

To help with debugging, results include graphical representations of both the desired behavior and the actual output. You can expand statements to view plots of each signal that contributes to the overall result. For more information, see Assess Temporal Logic Using Temporal Assessments.

Observers: Wirelessly access signals and add verification logic without side effects

Observers allow you to monitor the dynamic response of your system model without signal lines. You can wirelessly debug and test Simulink signals from multiple areas and hierarchies of your system.

To use Observers, you add an Observer Reference block to your system model. The Observer Reference block houses a separate Simulink model: the Observer model. The Observer model contains Observer Ports, which are mapped to signals from your system model.

Accessing data by using Observers keeps your model block diagram simple. You do not need to add ports, blocks, connections, or interfaces to your model. Observer blocks preserve your model simulation semantics, which can help increase confidence that your verification results apply to the implemented system. For more information, see Access Model Data Wirelessly by Using Observers.

Simulation Output: Select signals and log data using a revised interface

R2019a introduces a revised interface and additional capabilities for capturing signal data:

- When you select signals in your model, you can now navigate through the model hierarchy and add signals as you go.
- The signal selection dialog displays new signals to add and also displays signals you have already selected.
- You can now capture data from local and global data stores, and select signals inside referenced models.

To capture signal data in a test case:

1. Expand the **Simulation Outputs** section of the test case.
2. Highlight blocks or signals in the system under test.
3. Select signals of interest to add to your test case.

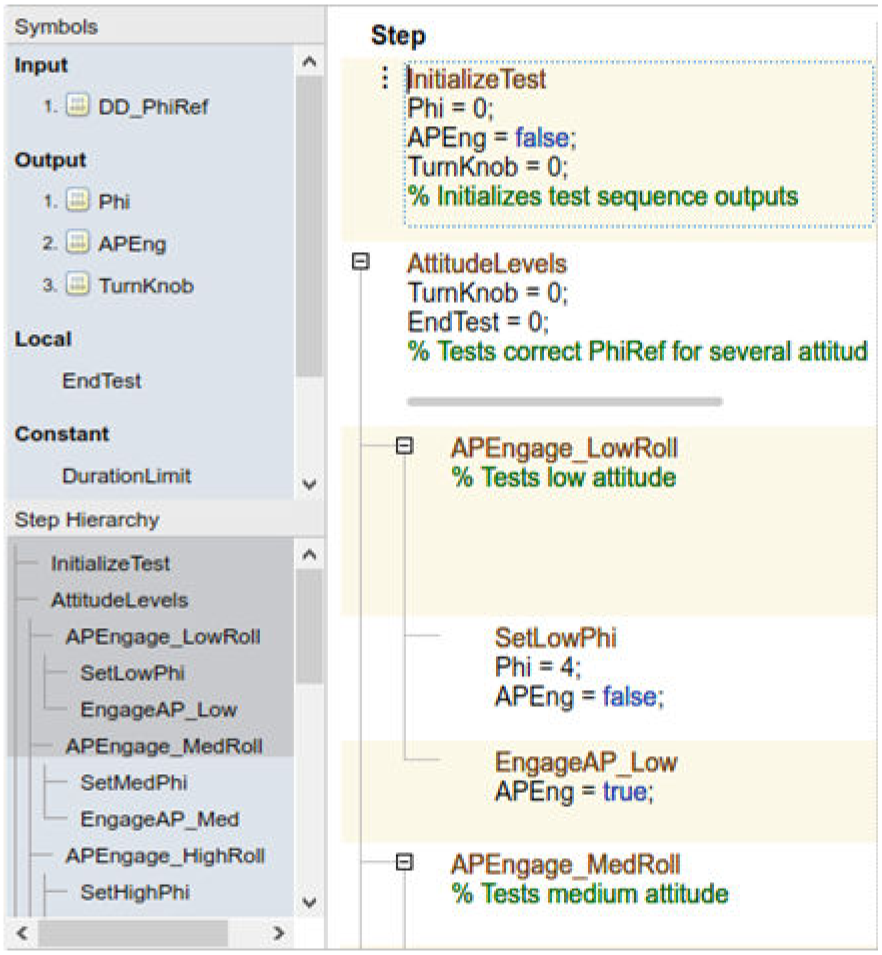
For more information, see Capture Simulation Data in a Test Case.

Input Data and Iterations: Test and iterate using Signal Editor scenarios

The Signal Editor block allows you to author multiple input data scenarios in a model or test harness. You can now select a Signal Editor scenario as a test case input. You can also run test case iterations that use different scenarios from the Signal Editor block. For more information on selecting inputs and creating iterations, see Inputs and Test Iterations.

Test Sequence Summary: Navigate test sequence steps using a summary view

You can scroll through test steps by using the **Step Hierarchy** pane in the Test Sequence Editor. For long and complex test sequences, the hierarchy pane gives you a high-level summary of the sequence using the test step names only. The pane highlights the test steps that are visible in the editor. You can scroll through the hierarchy and click in the pane to navigate to a new group of test steps in the editor.



```
InitializeTest
Phi = 0;
APEng = false;
TurnKnob = 0;
% Initializes test sequence outputs

AttitudeLevels
TurnKnob = 0;
EndTest = 0;
% Tests correct PhiRef for several attitud

    APEngage_LowRoll
    % Tests low attitude

        SetLowPhi
        Phi = 4;
        APEng = false;

        EngageAP_Low
        APEng = true;

    APEngage_MedRoll
    % Tests medium attitude
```

Test Sequence Search: Find text in a test sequence

You can find and replace text in Test Sequence actions, transitions, and descriptions by using the **Find & Replace** tool in the Test Sequence Editor toolbar. To open the **Find & Replace** tool, click the 🔍 icon in the toolbar.

1. In the **Find what:** box, enter the text you want to locate.
2. In the **Replace with:** with box, enter the updated text.
3. To locate the text, click **Find Next** or **Find Previous**.
4. Click **Replace** to replace the old text with the updated text.

Test Coverage: Justify or exclude model objects using a coverage filter

When using Simulink Coverage to record test coverage, you can exclude or justify model objects that are intentionally not exercised. Specify a coverage filter file in the Test Manager, for a test file, test suite, or test case. For information on setting the coverage filter from the Test Manager, see Measure Model Coverage. For information on setting the coverage filter programmatically, see `sltest.testmanager.CoverageSettings`. For more information on how to use coverage filters, see Coverage Filtering (Simulink Coverage).

Baseline Testing: Store baseline data in MLDATX format

In R2019a, you can add signal data from MLDATX files as baseline data, and use MLDATX files to capture baseline data from a simulation. Additional model elements are supported for baseline data capture:

- Arrays of buses
- Simscape™ data
- For Each Subsystem data

Testing with MATLAB Unit Test

You can create an MLDATX results file when you run a Simulink Test test file using MATLAB Unit Test. By creating an MLDATX results file, you can view and investigate results in the Test Manager.

1. Create the `sltest.plugins.TestManagerResultsPlugin` using the `'ExportToFile'` name-value pair. Specify a name for the MLDATX file.
2. To view results in the Test Manager, click the **Import** button in the Test Manager toolbar and select the MLDATX file.

Simulink Test contextual tab in the Simulink toolstrip

In R2019a, you have the option to turn on the Simulink Toolstrip. See "Simulink Toolstrip Preview replaces menus and toolbars in the Simulink Desktop" in the Simulink Editor section of the Simulink release notes for more details.

The Simulink Toolstrip includes contextual tabs –– they appear only when you need them. The Simulink Test contextual tabs include options for completing actions that apply only to Simulink Test.

- To enable the Simulink Test toolstrip, from the Simulink toolstrip, click **Apps** then click the **Simulink Test** button. The **Test** tab appears.
- Options in the **Test** tab change depending on what you select in the Simulink Editor. For example, you can create a test harness for a subsystem by selecting the subsystem, then clicking the **Add Test Harness for Selection** button.

Test harness support for bus element ports

In Simulink, you can use bus element ports to access individual bus signals at model interfaces. In Simulink Test, you can create a test harness for a Model block or a model block diagram that uses bus element ports at its root level. Creating a test harness and selecting test harness properties follows the process described in Create Test Harnesses and Select Properties.

R2018b

Excel-Based Testing: Define test cases in Excel spreadsheets

In R2018b, you can use Excel to specify test case input data, output data, tolerances, and parameters. You can also capture simulation output in the same file.

If you have existing test data, you can generate an Excel template from your model, then copy the data to the template. For details, see Author a Test in Excel. Once your Excel data is complete, you can create test from the Excel file. For details, see Create Test with Data from Excel.

C Caller Block Support: Test models that contain handwritten code

The C Caller block allows you to call a C function directly from a model. You can test the C function by creating a test harness for the C Caller block. Highlight the block and select **Analysis** > **Test Harness** > **Create for** *[block name]*. For details, see Test Integrated Code. For an example, see C Code Verification with Simulink Test.

SystemVerilog Assertions: Map Test Assessment blocks to SystemVerilog Assertions in generated DPI components

DPI component generation for HDL Verifier™ is now supported for `verify` statements in Test Sequence blocks and Test Assessment blocks. After adding `verify` statements to the block, you can generate a SystemVerilog DPI component that includes a verification to be used in a Verilog® or SystemVerilog simulation. For more information, see Generate SystemVerilog DPI Component from `verify` Statement (HDL Verifier).

Unified Scheduling Options: Generate scheduler for modeling styles including export function models and rate-based models

You can use scheduler blocks to execute functions and function-call subsystems in your test harness. When you include a scheduler in a test harness, the scheduler interface is consistent for different modeling styles, including export function models, rate-based models, and models with asynchronous function calls (JMAAB style B). Also, you can select a Test Sequence block or MATLAB Function block as the scheduler. In previous releases, you could use a Test Sequence block only.

If your model or subsystem has function calls or root inports, the test harness includes the scheduler block.

To include a scheduler in the test harness:

1. From the Simulink menu, select **Analysis** > **Test Harness** > **Create for Model**. For a subsystem test harness, select the subsystem and select **Analysis** > **Test Harness** > **Create for** *[block name]*.
2. In the test harness create dialog box, for **Add scheduler for function-calls and rates**, select `Test Sequence` or `MATLAB Function` from the list. To include no scheduler block and connect function calls to Inport blocks, select `None`.
3. To include calls to initialize, reset, and terminate functions in your model, in the test harness create dialog box, select **Enable initialize, reset, and terminate ports**.

When creating test harnesses with `sltest.harness.create`, use the new option `'SchedulerBlock'`.

Test harness support for string data types

When you create a test harness, a string input is connected to an Inport, Ground, or String Constant block, depending on which harness source you choose. A string output is connected to an Outport or Terminator block, depending on the harness sink you choose. For a list of source and sink blocks that get connected to string inputs and outputs, see **Test Harness Construction for Specific Model Elements**.

Testing with MATLAB Unit Test

- You can specify the location to save the HTML model coverage report. Construct a `ModelCoverageReport` object that specifies the folder. Use the `ModelCoverageReport` object when you create the `ModelCoveragePlugin`. For more information, see the example on the `ModelCoverageReport` reference page.
- You can get model coverage results in a format compatible with continuous integration systems such as Jenkins®. Use the `ModelCoveragePlugin`, `ModelCoverageReport`, and `CoberturaFormat`. For more information, see Tests for Continuous Integration.
- You can include Test Manager results in the **Details** field of each `TestResult` object by using the `TestManagerResultsPlugin`. To publish Test Manager results in the MATLAB Test Report, configure your test file for reporting and add the `TestReportPlugin` and `TestManagerResultsPlugin` classes to the `TestRunner` object. For more information, see the example on the `TestManagerResultsPlugin` reference page.

Ability to overwrite simulation output results

When you run the same test multiple times, by default simulation output signals are added to existing plots. To instead overwrite the results with the newest data, in the test results, right-click **Sim Output** and select **Plot Signals** > **Overwrite**. For more information, see Simulation Outputs.

Functionality being removed or changed

DriveFcnCallWithTestSequence in sltest.harness.create is not recommended
Still runs

When you use `sltest.harness.create`, using `DriveFcnCallWithTestSequence` to include a scheduler block is not recommended. Use the new name-value pair `'SchedulerBlock','Test Sequence'`.

R2018a

Coverage Aggregation: Merge coverage results from multiple test runs

You can merge code coverage results from different test runs to a new set of results. In previous releases, only multiple test cases from the same test file showed all the results in the **Aggregated Coverage Results** section. In R2018a, you can merge results from multiple test files.

In the Test Manager **Results and Artifacts** pane, select the results sets whose coverage results you want to show together. From the context menu, select **Merge Coverage Results**. The merged results appear in the list, and the combined coverage results appear under **Aggregated Coverage Results**. For more information, see Collect Coverage in Tests.

Input Data Templates: Generate Excel and MAT-file templates for signal data from models

You can create Microsoft® Excel and MAT-file templates for input data from your Simulink model. After you create a template, fill it in with your test data. You can then use the resulting file as input to a baseline test.

To create the template, in a baseline test case for a model, under **Inputs**, click **Create**. Use the dialog box to specify whether to create a MAT-file or Excel file and the location. Then click **Create**.

For more information, see Create Data Files to Use as Test Inputs.

Excel Format for Baseline Tests: Capture baseline simulation data in Excel files

You can now capture baseline simulation data to a Microsoft Excel file. Previously, you captured baseline data only to MAT-files.

In a baseline test case, under **Baseline Criteria**, click **Capture**. Select **Excel** as the file format, and then click **Capture**.

For more information on capturing baselines, see Capture Baseline Criteria.

Excel Format Changes: Save test inputs and outputs in same sheet

You can now save Test Manager inputs and outputs in the same sheet. When you capture your baseline or create an input file, by default you save them to the same sheet and file. Also, when you create an Excel for inputs and outputs, you can enter all of the information in the same file. For details on the format, see Specify Microsoft Excel File Format for Signal Data.

Signal Editor: Create test scenario input data using a Signal Editor block source

With the Signal Editor block, you can create test input data scenarios and switch between them during simulation. Select the Signal Editor block source during test harness creation. For more information on the Signal Editor block, see Create and Edit Signal Data.

Coverage Metrics: Function and function-call metrics reported in Test Manager

Test Manager now reports on function and function-call coverage metrics. You can select these metrics in the **Coverage Settings** section of your test case to include the function and function-call coverage percentage in your test results.

For more information on selecting code coverage options in Test Manager, see Collect Coverage in Tests. For more information on coverage metrics, see Types of Model Coverage.

Changes to Test Sequence block programmatic interface

The Test Sequence Block programmatic interface has the following changes:

- The block property `EnableActiveStepOutput` is renamed to `EnableActiveStepData`. This property is used in `sltest.testsequence.getProperty` and `sltest.testsequence.setProperty`.
- The block property `OutputData` is renamed to `ActiveStepDataSymbol`. This property is used in `sltest.testsequence.getProperty` and `sltest.testsequence.setProperty`.

Compatibility Considerations

- For scripts using the `EnableActiveStepOutput` property in `sltest.testsequence.getProperty`, update scripts to use `EnableActiveStepData`:

  ```
  blockInfo = sltest.testsequence.getProperty...
     (blockPath,'EnableActiveStepData')
  ```
- For scripts using the `EnableActiveStepOutput` property in `sltest.testsequence.setProperty`, update scripts to use `EnableActiveStepData`:

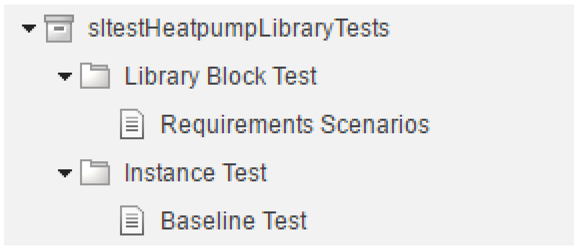
  ```
  blockInfo = sltest.testsequence.setProperty...
     (blockPath,'EnableActiveStepData',Value)
  ```
- For scripts using the `OutputData` property in `sltest.testsequence.getProperty`, update scripts to use `ActiveStepDataSymbol`:

  ```
  blockInfo = sltest.testsequence.getProperty...
     (blockPath,'ActiveStepDataSymbol')
  ```
- For scripts using the `OutputData` property in `sltest.testsequence.setProperty`, update scripts to use `ActiveStepDataSymbol`:

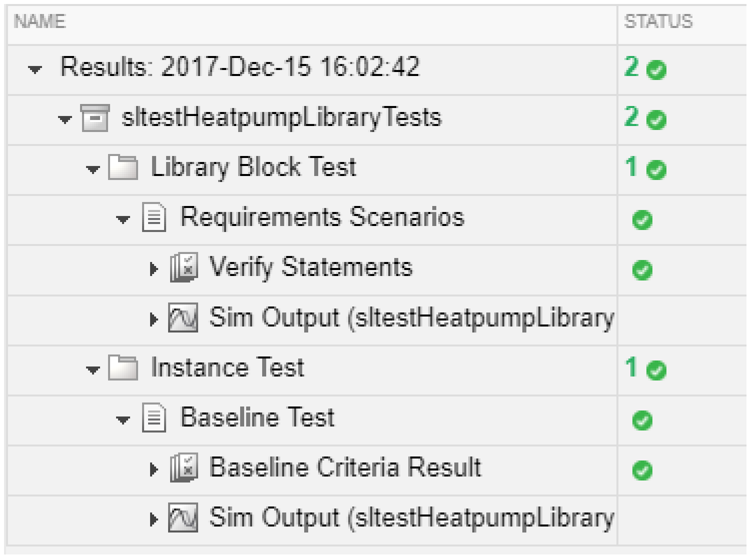
  ```
  blockInfo = sltest.testsequence.setProperty...
     (blockPath,'ActiveStepDataSymbol',Value)
  ```

Testing with MATLAB Unit Test

- Model coverage: You can collect model coverage when running test cases using MATLAB Unit Test. Create a model coverage plugin for the test runner, attach the plugin to the test runner, then run the test. For more information and an example, see `sltest.plugins.ModelCoveragePlugin`.
- Results hierarchy with MATLAB Unit Test: When you run Simulink Test test files using MATLAB Unit Test, the Test Manager results now reflect the test hierarchy. Previously, the results appeared in a flat list. For example, this test file contains two test suites:



Running the test with MATLAB Unit Test produces Test Manager results with hierarchy:



NAME	STATUS
Results: 2017-Dec-15 16:02:42	2 ✓
sltestHeatpumpLibraryTests	2 ✓
Library Block Test	1 ✓
Requirements Scenarios	✓
Verify Statements	✓
Sim Output (sltestHeatpumpLibrary	
Instance Test	1 ✓
Baseline Test	✓
Baseline Criteria Result	✓
Sim Output (sltestHeatpumpLibrary	

In the command-line results, the result `Name` field reflects the hierarchy. For example, running the test file produces:

```
heatpumpResult =

  1×2 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
   2 Passed, 0 Failed, 0 Incomplete.
   4.4774 seconds testing time.
```

The result `Name` field lists test file, test suite, and test case names:

```
>> heatpumpResult.Name

ans =

    'sltestHeatpumpLibraryTests >
        Library Block Test/Requirements Scenarios'
```

```
ans =

    'sltestHeatpumpLibraryTests >
        Instance Test/Baseline Test'
```

R2017b

Test Harness Generation Callbacks: Customize harness creation with post-create and post-rebuild callbacks

Customize test harnesses using callback functions that run after you create or rebuild the test harness. For example, you can add custom source or sink blocks, add a plant model for closed-loop testing of a controller, or enable data logging. Test harness callbacks are reusable functions, allowing you to add or change harness features and settings systematically. To create a callback:

1. Create the callback function. In the function, script the commands to customize the test harness.
2. Specify the function as the **Post-create callback method** or **Post-rebuild callback** method. For a new test harness, specify a post-create or post-rebuild callback in the **Advanced Properties** of the test harness creation dialog box. For an existing test harness, specify a post-rebuild callback in the harness properties dialog box.

For more information and an example, see Customize Test Harnesses.

Harness Component Synchronization Comparison: Identify differences in the component under test before explicitly synchronizing harnesses

Before synchronizing the component under test between the test harness and main model, you can check whether the component under test differs from the component in the main model by using sltest.harness.check. For more information on component synchronization, see Synchronize Changes Between Test Harness and Model.

Multirelease Testing: Execute test cases with older MATLAB releases

You can develop baseline, equivalence, and simulation tests in the current MATLAB release and run the test simulations in a different release. Developing tests in the current MATLAB release allows you to take advantage of recent Test Manager developments and run simulations in your production environment. The release you run the test in must support the features used in the test—for example, test harnesses—to run.

For baseline tests, you can establish the baseline data in one release and run the test case in that release. Or you can establish the baseline data in one release and run the test in another release, enabling you to compare results from different releases.

When running tests in multiple releases, your model or test harness must be compatible with the MATLAB version running your test. Also, the MATLAB version must support the features of your test case. Previous MATLAB versions do not support test case features unavailable in that release. Certain test case features are not supported for multiple release testing.

For more information, see Run Tests in Multiple Releases.

Signal Failure Navigation and Baseline Updates: Navigate between signal comparison failure regions, and automatically update baseline data

If a baseline test result includes failures, you can navigate between signal comparison failures in the data inspector view of the Test Manager. From this view, you can update the test baseline data in MAT-files. Updating the baseline data based on test failures is useful if your failures are due to a recent model update, such as a parameter value change.

For more information, see Examine Test Failures and Modify Baselines.

Expanded Microsoft Excel Support: Define test inputs and expected outputs including bus support and data type definition

Microsoft Excel files in Test Manager can now specify detailed information about signal data. You can use the expanded format for input signal data when you select an Excel file as baseline criteria.

In previous releases, in Excel file format only scalar doubles were supported. Now you can specify signal types (such as complex and multidimensional), data types, units, interpolation, bus element information, and function-call execution times. To learn about the format, see Specify Microsoft Excel File Format for Signal Data.

You can import multiple Microsoft Excel sheets at once to use as separate test input groups or a range of data to import. You can also specify sheets and a range of cells to use as baseline criteria.

The Test Manager API has some changes in support of the new capabilities.

- A new method, `addExcelSpecification`, for the `sltest.testmanager.BaselineCriteria` and `sltest.testmanager.TestInput` classes
- New arguments for specifying options for `sltest.testmanager.TestInput` files with `sltest.testmanager.TestCase.addBaselineCriteria` and `sltest.testmanager.TestCase.addInput`.
- A new property, `ExcelSpecifications`, for the `sltest.testmanager.BaselineCriteria` and `sltest.testmanager.TestInput` classes.

Compatibility Considerations

The `refreshIfExists` argument for `sltest.testmanager.TestCase.addBaselineCriteria` has been replaced with a Name,Value pair. Scripts that add MAT-files that use this argument will still work. See “Functionality Being Removed or Changed” on page 12-5 for details.

The `SheetName` property for `sltest.testmanager.TestInput` has been replaced with the `ExcelSpecifications` property. Scripts that use this property will still work. See “Functionality Being Removed or Changed” on page 12-5 for details.

The `sltest.testmanager.TestCase.addInput` method by default now returns one `sltest.testmanager.TestInput` object for each sheet in an Excel file added as a test case input. In previous releases, this method returned a single `sltest.testmanager.TestInput` object even if the Excel file had multiple sheets.

As a result, commands in existing scripts that operate on the result of this method might return an error. An error occurs with commands that expect a single object. Modify any such script to operate on an array instead. Or, to return a single option as in earlier releases, set the `'SeparateInputs'` option to `false`.

Test File Comparison: Identify differences by comparing two test files

You can review test changes or updates by comparing two test files side by side. Test file comparison is similar to comparisons of other MATLAB files that you compare using **Compare** on the MATLAB toolstrip.

For more information, see Compare Test Files.

Test Sequence Editor enhancements: Change step hierarchy, reorder transitions, and cut/copy/paste test steps

Create and edit test sequences with the enhanced Test Sequence Editor. For detailed information, see Test Sequence Editor.

- Change the hierarchy level of a test step by indenting or outdenting the test step. Right-click the test step, and select **Indent** to move it to a lower level, or **Outdent** to move it to a higher level.
- Reorder test steps. When hovering over a step, three dots appear left of the step name. Click and drag the dots to move the step to a new position within the same hierarchy level.
- Reorder test step transitions. Hover over the transition number, then click and drag the transition to the new order. The corresponding next step is maintained.
- Cut, copy, and paste test steps using the right-click context menu. You can paste before or after a step, or paste as a substep.

Input and Baseline Signal Editing: Edit test case baseline and input signal data

In R2017b, you can edit input signal data and expected outputs (baseline criteria) from MAT-files and Microsoft Excel in the Test Manager. In previous releases, you edited only Excel input files in the Test Manager.

You edit MAT-file data in the signal editor and Microsoft Excel data directly in Excel. For more information on editing the baseline signal data, see Manually Update Signal Data in a Baseline. For more information on editing input signals, see Edit Input Data Files in Test Manager.

Iteration Naming: Create rules to name autogenerated iterations

When auto generating iterations for a test case, you can customize iteration names by specifying a naming pattern. For more information, see Create Table Iterations.

External Inputs for Real-Time Applications: In test cases for real-time applications, map external data inputs to root Inport blocks

You can use external data mapped to root Inport blocks in real-time tests. This facilitates reusing desktop test cases in real-time testing, as both desktop and real-time tests can access the same test input data stored in an external file.

In your test harness or model, use root Inport blocks as sources. In the test case Inputs, map external data to the root Inport blocks using Microsoft Excel or MAT-files. For an example of reusing a desktop test case using input data in an Excel file, see Reuse Desktop Test Cases for Real-Time Testing.

Test Harness .MDL Support: Create test harnesses for models saved in .MDL file formats

Test harnesses support models that use the .MDL file format. Test harnesses for .MDL models are saved externally. For more information about externally vs. internally saved test harnesses, see Manage Test Harnesses.

Functionality Being Removed or Changed

The Test Manager API has these changes.

Functionality	Result	Use Instead	Compatibility Considerations
`refreshIfExists` argument in `sltest.testmanager.TestCase.addBaselineCriteria`	Still runs for MAT-files	`'RefreshIfExists',Boolean`	
`SheetName` property for `sltest.testmanager.TestInput`	Still runs	`ExcelSpecifications`	`SheetName` is now read-only
`sltest.testmanager.TestCase.addInput`	Now returns an array with Excel files if the file had multiple sheets	Use `'SeparateInputs',false` with existing scripts	Use `'SeparateInputs',false` only to match functionality from previous releases. The default of `true` matches the behavior in the interface.

R2017a

Harness Import: Simplify test harness management by converting test harness models to Simulink Test harnesses

To simplify management and synchronization of standalone harness models, such as those created manually, using Simulink Verification and Validation™, or Simulink Design Verifier, convert the standalone models to test harnesses in Simulink Test. Converting standalone models to test harnesses allows you to iterate on your design using push and rebuild functions, and manage test harnesses using the UI and programmatic interface.

For more information, see Create Test Harnesses from Standalone Models and the function `sltest.harness.import`.

Expanded Test Harness Capabilities: Create test harnesses with additional semantics, blocks, sources, and synchronization behavior

1. Control when the component under test design synchronizes between the model and harness. When creating a test harness, select the harness **Synchronization Mode** in the **Advanced Properties**. For an existing harness, click the harness properties badge to change the synchronization mode. For more information, see Synchronize Changes Between Test Harness and Model. Synchronization modes include:

   - `Synchronize on harness open and close`: The component in the main model, or the test harness, is automatically updated when the harness closes or opens.
   - `Synchronize on harness open`: The component in the test harness is updated when the harness opens.
   - `Synchronize only during push and rebuild`: Synchronization does not occur when the harness opens or closes. You manually control synchronization by selecting **Analysis** > **Test Harness** > **Push Component and Parameters to Main Model** or **Analysis** > **Test Harness** > **Rebuild Harness from Main Model**.
2. Use Signal Builder blocks as sources for test inputs generated by Simulink Design Verifier analysis. `Signal Builder` is a **Harness Source** option when you select **Export test cases to Simulink Test** in the Simulink Design Verifier results dialog box. For more information, see Test Models Using Inputs Generated by Simulink Design Verifier™ .
3. Test user-defined functions by creating test harnesses for the following additional blocks in the User-Defined Functions library:

   - S-Function block
   - S-Function Builder block
   - Level-2 MATLAB S-Function block
4. Create test harnesses for subsystems within linked library blocks.
5. Move and clone test harnesses across different blocks in the same model or in different models. See Move and Clone Test Harnesses and the functions `sltest.harness.move` and `sltest.harness.clone`.
6. Create test harnesses for models that use a mix of asynchronous and rate-based modeling.
7. Create test harnesses that honor the sorted execution order of blocks and subsystems for export function modeling.

Compatibility Considerations

For `sltest.harness.create` and `sltest.harness.set`, the `'EnableComponentEditing'` option is removed. Editing the component under test is controlled by the CUT synchronization mode. Update scripts that use the `'EnableComponentEditing'` option to specify `'SynchronizationMode'`.

- To prevent editing the component under test in the test harness (previously specified by `'EnableComponentEditing',false`), use `'SynchronizationMode','SyncOnOpen'`.
- To allow editing the component under test in the test harness (previously specified by `'EnableComponentEditing',true`), use `'SynchronizationMode','SyncOnOpenAndClose'` (default) or `'SynchronizationMode','SyncOnPushRebuildOnly'`.

Support for Debug Mode: Debug system under test simulation

From the Test Manager, you can debug the system under test simulation. Click the **Debug** button in the Test Manager toolstrip before executing the test. The simulation pauses at simulation start and presents a debug prompt at the command line. You can step through simulation using the Stepping Options buttons in the Simulink Editor toolstrip. For more information, see Use Simulation Stepper (Simulink).

Time Tolerance: Specify leading and lagging time tolerance in test cases

Account for temporal shifts in your test results using time tolerances. Some test cases, such as real-time tests, are affected by timing attributes of the execution environment and shifts in data logged from physical systems. You can account for timing differences by including time tolerance in baseline and equivalence criteria. For details, see Apply Tolerances to Test Criteria.

Simscape Component Support: Create test harnesses for subsystems connected by Simscape physical connections

You can test subsystems connected by Simscape physical connections using test harnesses. Test harnesses isolate Simscape subsystems, providing physical signal ports at the subsystem interface. To define the physical system, add blocks to the test harness. To connect Test Sequence and Test Assessment blocks to the physical system, use Simulink-PS Converter and PS-Simulink Converter blocks.

Expanded API: Edit and manage Test Sequence Blocks with an expanded API

The `sltest.testsequence` API includes additional functions to read, edit, and delete test sequence steps, transitions, and data symbols. Use these functions to create, edit, and manage Test Sequence and Test Assessment blocks. For more information, see the Test Sequence Programming section on the Logic-Based Testing page.

The `'Label'` property name has been changed to `'Action'` to more closely match the functionality of the argument in creating and editing test sequence steps. The change applies to the functions

- `sltest.testsequence.addstepafter`
- `sltest.testsequence.addstepbefore`
- `sltest.testsequence.addstep`
- `sltest.testsequence.editstep`

Compatibility Considerations

If you have a script that uses these functions with the property name `'Label'`, running the script returns a warning that `'Label'` is removed. Update the script to use the property name `'Action'` instead of `'Label'`.

Signal Logging Selection: Specify signals to log in a test case without modifying the model

Using the **Simulation Outputs** section, you can log additional signals in the Test Manager without changing the logging settings in your model. For more information, see Simulation Outputs.

Proof Objective Support for verify() Statements: verify() statements interpreted as proof objectives for Simulink Design Verifier analysis

If your model or test harness contains a `verify()` statement in a Test Assessment or Test Sequence block, Simulink Design Verifier property proving analysis interprets the `verify()` statement as a proof objective. This allows your `verify()` statements to be used for both functional testing and formal analysis, without having to add Proof Objective blocks to the model. Also, for `verify()` statements falsified, you can create counterexamples that falsify the objective during simulation. For more information about property proving, see Prove Properties in a Model (Simulink Design Verifier). For more information about `verify()` statements, see Assess Simulation Using Logical Statements.

Enhanced Reporting: Include MATLAB figures, save report options, and automatically generate reports after testing

- You can include custom MATLAB figures in your report. For details, see Create, Store, and Open MATLAB Figures.
- You can automatically create a report after executing a test file. In the test file, under **Test File Options**, select **Generate report after execution**. The Test Manager displays options for the report, which are saved with the test file. For details, see Export Test Results and Generate Reports.

Test Manager Integration with Simulink Design Verifier: Generate additional tests from within the Test Manager to increase coverage

If you have Simulink Design Verifier, you can generate tests to achieve additional coverage starting from the coverage results pane in the Test Manager. After executing tests, view the cumulative coverage results in the Test Manager results pane. Select the coverage result and click **Add Tests for Missing Coverage**. For an example, see Perform Functional Testing and Analyze Test Coverage.

Test Manager Hierarchies: Expand and collapse trees using context menu

You can expand or collapse hierarchies in the Test Manager using the context menu. For example, in the **Test Browser** pane, right-click a test file or test suite. From the context menu, select **Expand All** or **Collapse All**. A similar context menu item appears in hierarchies in the **Results and Artifacts** pane.

Stop simulation at the end of test input data

If you have timeseries test input data, you can limit the simulation data output by stopping simulation at the end of the test input timeseries. For example, if your input data stops after 10 seconds, but your model simulation time is set to 300 seconds, limit the simulation to avoid 290 seconds of unnecessary data. Select **Stop simulation at last time point** in the **Inputs** section of the test case definition.

Capture baseline data from test case iterations

If your test case is configured with table iterations, you can capture baseline data from logged signals using a one-click process. Baseline criteria is captured in a separate file for each iteration, and each baseline data file appears with its corresponding iteration in the iterations table. For an example, see Capture Baseline Data from Iterations.

Simplify test editor view by hiding unused sections

You can simplify the view for test files, test suites, and test cases by hiding unused test sections. In the Test Manager, click the **Preferences** button in the toolbar, and select sections to hide or display. Populated sections are always displayed. This is a global Test Manager setting. To customize view for multiple users, you can set the preferences programmatically using `sltest.testmanager.getpref` and `sltest.testmanager.setpref`. For more information, see Test Sections.

Test Case Conversion: Convert desktop simulation test cases to real-time test cases

You can convert test cases that run on desktop simulation into real-time test cases. In the **Test Browser**, right-click the test case name and select **Convert to** > **Real-Time Test**.

Data dimension settings for certain test harness inputs

When you create a test harness using From Workspace, From File, or Constant blocks as sources, the default value of the source reflects dimensions of the signal.

Run MATLAB Unit Tests in Fast Restart mode

The MATLAB Unit Test framework supports Fast Restart mode for running test cases authored in Simulink Test.

R2016b

Custom Criteria: Define custom criteria for test case evaluation

You can customize test case pass and fail criteria using MATLAB and the MATLAB Unit Testing framework. Use custom criteria in addition to assessments such as `verify` statements and timeseries comparisons. For example, assess the final value of a signal, set a maximum threshold, or post-process signal results using MATLAB toolboxes. With MATLAB Unit Test, qualifications return `pass` or `fail` results to the Test Manager. See Apply Custom Criteria to Test Cases.

MATLAB Unit Test Integration: Use MATLAB Unit Test to execute and integrate tests

You can run tests authored in Simulink Test using the MATLAB Unit Test framework. This allows you to combine execution of tests authored in both frameworks. You can customize test execution with a test runner, and access test results in MATLAB.

With the MATLAB Unit Test framework, you can set up systematic testing using continuous integration systems. Use plugins such as the `TAPPPlugin` to create results that are compatible with CI systems such as Jenkins. See Test Models Using MATLAB Unit Test.

Test Case Tagging: Filter and execute tests using custom tags

You can group tests by assigning custom tags to test cases and suites, and filter tests to view test case subsets. Running filtered tests can save time compared to running a full test suite. Filter tests in the test browser using the syntax `tag:<name>`. To run filtered tests, expand the drop-down menu under **Run** and select **Run filtered**. See Filter Test Execution and Results.

Graphical Output for Model Verification Blocks: Analyze results from Assertion and Model Verification blocks in Simulation Data Inspector and the Test Manager

Blocks in the Model Verification library return a `pass` or `fail` result to the Test Manager, using semantics similar to a `verify` statement. Viewing results graphically helps you to:

- Determine the time when a failure occurs.
- Debug the model by comparing the verification result with relevant signals.
- Trace failures from the graphical results to the model.

See View Graphical Results From Model Verification Library.

Initialize and Terminate Functions: Test harnesses include calls for initialize and terminate systems in a model

When you create a test harness for a model block diagram, you can include calls to initialize and terminate systems. The test harness creates a Test Sequence block configured to schedule function calls to initialize and terminate systems.

Harness Requirements Linking: Establish requirements traceability for external test harnesses

Requirements linking is supported for test harnesses that are stored externally as independent SLX files.

Test Case Conversion: Convert test cases between baseline, equivalence, and simulation test types

You can convert existing test cases between baseline, equivalence, and simulation test types. This helps facilitate test case reuse. For example, to reuse an existing baseline test as an equivalence test, copy the baseline test and change the copied test case to an equivalence test. To convert a test case,

1. In the Test Browser, right-click the test case name.
2. Select **Convert to** > **Baseline Test / Equivalence Test / Simulation Test**.

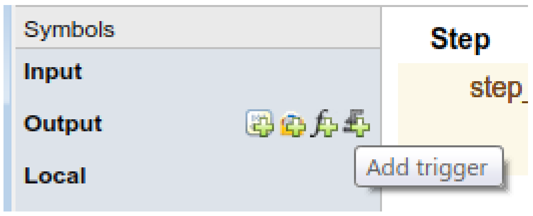
Additional Report Templates: Create PDF and HTML reports using report templates

With a MATLAB Report Generator license, you can create custom PDF and HTML reports from the Test Manager using report templates. In the Create Test Result Report dialog box, select a PDFTX or HTMTX template file. For more information, see Export Test Results and Generate Reports, and Create Report Templates.

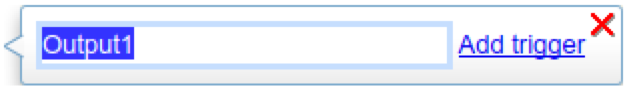
Output Event Definition: Create Test Sequences that trigger actions in other subsystems

You can use trigger outputs in a Test Sequence block or Test Assessment block to activate a triggered subsystem or signal an event in a Stateflow chart. To create a trigger output in a Test Sequence block,

1. In the test sequence editor **Symbols** pane, click the **Add trigger** icon next to the **Output** section.



2. Enter the output name, and click **Add trigger**.



3. Trigger outputs initialize to 0. In the test sequence, use the `send` command to activate the trigger output.

```
send(Output1)
```

Access verify statement output using a programmatic interface

Simulation data output from `verify` statements are available via a programmatic interface. To get assessment results, use the `sltest.getAssessments` function.

Revised visualization for verify statements and other assessments

The Test Manager displays `verify` statement results and other assessments using a stem plot, which plots the `verify` output data as stems extending from a baseline along the x-axis. The baseline is a `pass` result, with stems extending from the baseline for `untested` and `fail` results. Data color corresponds to the statement result:

Color	Result
Red	`Fail`
Green	`Pass`
Gray	`Untested`

To maintain readability, plots adjust the data displayed if the result is the same for numerous consecutive data points. Zooming in to smaller x-axis intervals displays additional discrete data points. Changes in result are always displayed, for example, from `pass` to `fail`. See Assess Simulation Using Logical Statements for an example. To get assessment results programmatically, use the `sltest.getAssessments` function.

Common test harness sources and sinks for signal and control inputs, and revised signal routing

When you create a test harness, additional inputs are connected to the specified harness source and sink type, rather than to Inport or Outport blocks. Additional inputs expand your ability to drive component inputs with a single source type for:

- Control inputs
- Function call inputs
- Data store memory
- Goto and From blocks
- Export function models
- Simulink functions

To simplify test harness block diagrams, the signal routing uses Goto and From blocks to connect component under test input and output signals to Test Assessment blocks. Also, some signal routing blocks are contained in input and output conversion subsystems.

Add baseline criteria using expected outputs captured by Simulink Design Verifier

Simulation output captured by running Simulink Design Verifier tests are now included as baseline data in test cases exported from Simulink Design Verifier to Simulink Test.

Set SIL or PIL simulation mode from the Test Manager for a model referenced by a test harness

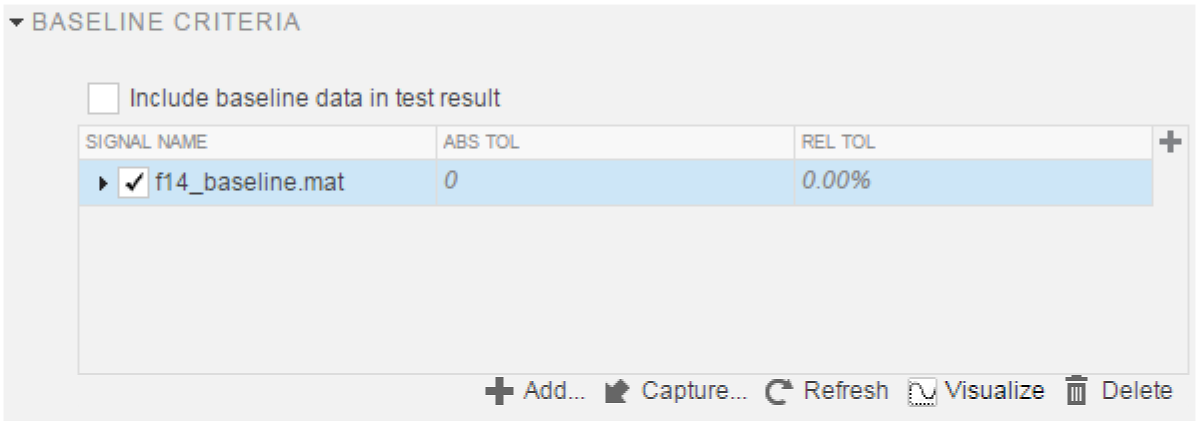
From the Test Manager, you can override the simulation mode to software-in-the-loop (SIL) or processor-in-the-loop (PIL) for a model referenced by a component under test in a test harness. Overriding the referenced model simulation model applies to test harnesses for block diagrams and test harnesses for Model blocks, since these types of test harnesses use Model blocks as the component under test. Setting the simulation mode from the Test Manager allows you to use an equivalence test and a single test harness to perform SIL or PIL output equivalence verification.

Highlight dependencies in test harnesses using Model Slicer

If you have a Simulink Design Verifier license, you can use the Model Slicer to highlight functional dependencies in test harnesses created by Simulink Test. Highlighting functional dependencies helps you analyze behavior in large or complex test harnesses. See Highlight Functional Dependencies.

View baseline data graphically

In the Test Manager, you can view a graph of test case baseline data. This facilitates reviewing the baseline data before you map it to a test case and run tests. In the **Baseline Criteria** section of the test case, highlight the signal name and click **Visualize**. The graph appears in the Simulation Data Inspector.



SIGNAL NAME	ABS TOL	REL TOL
✓ f14_baseline.mat	0	0.00%

R2016a

Real-Time Testing: Author and execute real-time tests with Simulink Real-Time

A new Real-Time Test builds a Simulink Real-Time application from your model or test harness and runs it on a target computer. You can assess the real-time execution using `verify` statements, and collect real-time data for analysis in the Test Manager. See Test Models in Real Time.

verify Statement: Author test sequence assessments to verify simulation behavior without stopping the simulation

In the Test Sequence and Test Assessment blocks, you can use `verify` statements to assess a logical condition without stopping simulation. A `verify` statement returns a fail, pass, or untested result. Results of each `verify` statement appear in the Test Manager. See Assess Simulation Using Logical Statements.

Test Report Customization: Customize test result reports using Simulink Report Generator

You can write scripts to customize the details of Test Manager result reports such as text formatting, output plots, headers and footers, layouts, and more. See Customize Generated Reports.

External Test Harnesses: Save Simulink Test harnesses as external files

You can opt to save your test harnesses externally, as independent SLX files. External test harnesses allow you to create or change test harnesses without changing the model SLX file, which is useful for models under change management. External harnesses provide the same synchronization and push/rebuild capability as internal harnesses saved with the model SLX file. See Manage Test Harnesses.

Test Iterations: Create tests with iterations such as parameters and input vectors

To test and sweep through a range of parameters, inputs, and other test case settings, you can author and organize many tests in one place using iterations. To help create iterations, templates are available for Signal Builder groups, parameter sets, inputs, configuration settings, and baseline criteria. You can run iterations using fast restart if it is supported by your model. See Run Multiple Combinations of Tests Using Iterations.

Test generated using Simulink Design Verifier now appear as iterations in a test case rather than separate test cases in a test suite.

Aggregate Coverage Results: Aggregate Simulink Verification and Validation coverage results across executed tests

If you have a Simulink Verification and Validation license, then you can collect the coverage results from your tests. The results are aggregated at test case, test suite, and test file levels. Coverage results can also be included in the Test Manager results report. See Collect Coverage in Tests.

Parallel Test Execution: Distribute test execution in parallel to decrease test run time

If you have a Parallel Computing Toolbox™ license, then you can run tests in parallel across multiple workers on the same machine to decrease test execution time. See Run Tests Using Parallel Execution.

Test Harness for Libraries: Create and manage test harnesses for library components

You can create test harnesses for library blocks and move test harnesses from linked blocks to the library source. See Test Library Blocks.

Requirement Traceability: Link to requirements in test harnesses and test sequences

If you have a Simulink Verification and Validation license, you can create requirements links for model objects in internally stored test harnesses. Requirement links for the component under test synchronize between the main model and the test harness. You can also create requirements links for test steps in Test Sequence and Test Assessment blocks. See Link Tests to Requirements.

Simulink and Export Function Support: Create test harnesses for models containing Simulink functions and export functions

If you generate a harness for a model configured to export functions, the harness will contain a new Test Sequence block that schedules the function-call signals and Simulink Functions in the export-function model. You choose the sources and sinks for other subsystem inputs and outputs. See Test Models that use Export Functions for AUTOSAR-Compliant Code.

Test Assessment block available for all harness sources

The Simulink Test library offers a separate Test Assessment block entry. You can include a Test Assessment block with any test harness source using the test harness creation dialog box. The Test Assessment block is a Test Sequence block configured with a default When decomposition sequence and a `verify` statement, which are commonly used in model assessment.

Test Sequence block support for messages

Test Sequence blocks support sending and receiving messages. Messages are objects that carry data and can be queued. You can send a message using a message output and the `send` command, and receive a message using a message input and the `receive` command. When a test step receives a message, it can use the `receive` result or the message data in a step action or transition. See Test Sequence Action and Transition Operations.

Test Sequence Editor enhancements

The Test Sequence Editor offers several enhancements for the Test Sequence and Test Assessment blocks.

- You can add a description for a test step using the **Description** field.
  - Code generated from the block includes test step descriptions as commented code. To include the commented descriptions in generated code, select **Simulink block descriptions** in the **Code Generation** > **Comments** section of the model configuration parameters.
  - Simulink Report Generator includes descriptions in the Test Sequence block reports.
- Syntax highlighting: The Test Sequence Editor includes MATLAB syntax highlighting for improved readability.
- Tab completion: The Test Sequence Editor suggests words, such as data symbols and functions, to complete test step programming syntax. A list appears based on the characters you type. Select a word from the list and press **Tab**, or press **Esc** to close the suggestions.
- Port reordering: You can reorder block inputs and outputs by dragging an **Input** or **Output** symbol name up or down in the **Symbols** sidebar.

Simulink Projects integration

You can create a Simulink project from a test file. When you create a project from a test file, it enables you to perform file dependency analysis. Projects let you easily see the impact that changes could have on tests that might use shared files. You can also run tests directly from the Simulink Projects interface. For more information about Simulink Projects integration, see Manage Test File Dependencies.

Test Generation for Subsystems

You can generate tests for a subsystem in a model from the Test Manager. The Test Manager creates a test harness for the subsystem and enables you to test the subsystem independently, thereby isolating it from the main model. See Generate Test Cases from Model Components.

R2015aSP1

Version: 1.0.1

Bug Fixes

R2015b

Expanded Simulink Test API: Automate test creation, editing, and execution using MATLAB scripts

You can use functions, classes, and methods to programmatically:

- Generate test cases from a model based on existing test harnesses and signal builder groups. See `sltest.testmanager.createTestsFromModel`.
- Edit test case simulation properties, parameter sets, and comparison criteria
- Create test files, test suites, and test cases
- Copy and move test suites and test cases
- Run individual test suites and test cases
- Copy a test harness, including its contents, configuration set, properties, and model association, using the `sltest.harness.clone` function

For more information about using the API, see Automate Tests Programmatically.

Test Case Automation: Create test cases with inputs generated by Simulink Design Verifier

Starting with the results of a Simulink Design Verifier analysis, Simulink Test creates test cases that use the inputs generated by Simulink Design Verifier. Test cases appear in the Test Manager and can use an existing or new test harness. See Test Models Using Inputs Generated by Simulink Design Verifier.

Qualification and Certification: Qualify Simulink Test for supported industry standards, including DO-178 and ISO 26262

You can use the IEC Certification Kit and DO Qualification Kit to qualify Simulink Test for supported industry standards, including DO-178, ISO 26262, and IEC 61508.

Enhanced Reporting: Use Microsoft Word templates to customize report generation

If you have a MATLAB Report Generator license, you can insert report items from the Simulink Test generated report into your own Microsoft Word templates. For more information on report generation, see Export Test Results and Generate Reports.

Additional tools for Test Sequence editing and debugging

The test sequence editor includes new tools you can use when creating, editing, and debugging a test sequence, including

- Undo and redo edits
- Cut, copy, and paste test steps using keyboard shortcuts
- Simulation rollback

Simulink Report Generator inclusion for Test Sequence block

You can include data from Test Sequence blocks in a report, using the Test Sequence component in Simulink Report Generator.

R2015a

Version: 1.0

New Features

Introduction to Simulink Test

Simulink Test provides tools for authoring, managing, and systematically executing simulation-based tests. You can create nonintrusive test harnesses to test models and subsystems. You can generate reports, archive and review test results, rerun failed tests, and debug the component or system under test.

Test harness for subsystem and model testing

Test harnesses provide a separate, nonintrusive testing environment for your models. A test harness associates with a particular model or model component and persists with the model. You define tests by adding inputs and assessments to the harness, and you can set harness-specific simulation parameters. The test harness synchronizes model changes to the main model. The Test Manager can access the test harnesses in your model. See Refine, Test, and Debug a Subsystem and Test Harness and Model Relationship.

Test Sequence block for defining tests and assessments

Test Sequence blocks concisely define a series of test steps and transitions using MATLAB action language. Each step defines the block output values and the condition that triggers the transition to another test step. You can define a test step hierarchy using different transition modes. Test Sequence blocks include concise output functions, such as square and sawtooth, and operators that return temporal information, such as the elapsed step time or the duration of a condition.

You can assess the model operation in the test sequence, or in a separate Test Sequence block. See Test Downshift Points of a Transmission Controller and the Test Sequence block.

Test Manager for test authoring and systematic test execution

The Simulink Test Manager enables you to organize and run large sets of tests for Simulink models. Using the Test Manager, you can author and execute test cases individually or as a batch. You can also link to test requirements from each test case if you have a Simulink Verification and Validation license. After you execute tests, the test outcome and any simulation output appear in the **Results and Artifacts** pane of the Test Manager.

Baseline, equivalence, and back-to-back testing with pass-fail criteria

You can test models using baseline and equivalence test case templates in the Test Manager. Baseline test cases compare simulation output to defined expected outputs. Equivalence test cases compare simulation output a second simulation. The simulation output comparison is evaluated according to absolute or relative tolerances, which you specify under **Baseline Criteria** or **Equivalence Criteria**. For more information on tolerances, see How Tolerances Are Applied to Test Criteria.

Archiving and reporting test cases and test results

After you execute tests, you can export the results in the **Results and Artifacts** pane of the Test Manager to a file or save them in a report. For more information on exporting results and generating reports, see Export Test Results and Generate Reports.