WHITE PAPER

Model-Based Design for Aerospace Systems: Developing Requirements and Tests with MATLAB and Simulink

MathWorks®

In Model-Based Design, a system model is at the center of the development process, from requirements development, through to design, implementation, and testing.

Aerospace engineering organizations adopt Model-Based Design for high-integrity systems as a method to manage requirements and ensure they are met, automatically generate code, and rigorously test models and systems.

This paper discusses how to manage requirements with ***MATLAB***® and ***Simulink***® and ensure test harnesses are robust, and provides a comprehensive test automation example using Simulink Test™.

Complete and Robust Requirements

Managing requirements is a complex task that can be streamlined with Model-Based Design. Text requirements come with a few complications. It can be difficult to translate high-level requirements from one team into low-level requirements for a different team without revisions. One reason for this is that a single aerospace system can include hundreds of components; predicting how a small issue on one component might affect another makes writing requirements an iterative process.

One way to handle this is to use Simulink and Stateflow® to model the system's control logic. A model designed in a Stateflow chart incorporates the logic for how the system should work and can link requirements using drag and drop. You can author, manage, and trace requirements within the model using within Simulink Requirements™. These requirements can be created within Simulink or imported into Simulink Requirements as read-only references that can be linked to your models. Simulink Requirements will indicate when a change has occurred to linked requirements, designs, or tests so you have a history of changes and can assess how that change impacts your designs.

See ***Requirements Definition for a Cruise Control Model*** for a practical example of how to use requirements-driven Model-Based Design.

When you compile requirements through a model and then generate a report that lists them, it eliminates complications accidentally introduced when requirements are begun as a written document.

Example

Airbus A380 Fuel Management System

The Airbus A380, the largest commercial aircraft currently in operation, has a range of more than 8000 miles.

The A380's fuel management system must be able to safely handle any failure in the system's 21 pumps, 43 valves, and other mechanical components. In a complex system, it's challenging for engineers at the requirements stage to predict problems that can result from combinations of relatively minor failures.

The fuel system specification document for the A380's predecessor, the A340, had over 1000 written requirements. "Text requirements can leave room for ambiguity and misinterpretation. When you have that many requirements, it's difficult for anyone to understand all the possible interactions between them and identify, for example, that a requirement on page 20 conflicts with one on page 340," says Christopher Slack, computational analysis expert in fuel systems at Airbus.

Airbus used Model-Based Design to model the A380's fuel management system, validate requirements through simulation, and clearly communicate the functional specification.

"The Simulink and Stateflow models enabled us to validate requirements early and communicate the functional specification to our suppliers, complementing the written requirements in conformance with ARP 4754," says Slack. "These models were reused to create desktop simulators, commission our HIL test rig, run on our virtual integration bench, and demonstrate system functionality to customers."

» *Read the full user story*

Test Cases That Address Requirements

When designing a robust test harness, engineering teams need to employ the right test cases that rigorously address requirements and only the stated requirements.

Simulink Check™ is a helpful tool to verify compliance with DO-178 and other guidelines and modeling standards. See ***Check for Standards Compliance in Your Model*** for an example on using the Model Advisor to check for model conditions that will cause problems in high-integrity applications.

Model and Code Coverage

Model and code coverage is a common measure of completion for testing within Model-Based Design. To adhere to DO-178C you need to be able to verify that object code has the same functionality as the model (low-level requirements), and that the object code meets the original high-level requirements. Say your requirements-based tests demonstrate 100% model coverage; if you run those requirements-based tests against the object code and get the same results as the model, that shows functional equivalence between the object code and the model. You can measure the code coverage and compare it with the model coverage to confirm that the test cases provide the required code coverage.

MathWorks®

Choosing the Right Tests and Automating Testing

Tests need to ensure compliance against modeling standards and requirements. Simulink Test and Simulink Coverage provide an environment where you can create your own automated tests and check your model and code coverage. Using Simulink Test and Simulink Requirements, you can link your requirements to test cases; see the example ***Link to Test Cases from Requirements***.

Example

Korean Air UAV Flight Control Software

In the past, Korean Air engineers hand-coded UAV flight control software. The company identified several drawbacks to this approach, particularly when a single flight control system targeted multiple UAV platforms. First, the algorithms the team was developing were too complex to program in C by hand. Second, they needed to be able to deploy hardware specification changes and the latest control algorithms quickly. Third, manual code reviews and unit tests required too much time and effort to complete.

Using Simulink Check and Simulink Coverage, the engineers performed regular checks to ensure the model complied with the company's modeling standards (based on MAAB guidelines) and to measure 100% MC/DC model coverage for their test suite.

"The software we developed with MATLAB and Simulink had more functionality and verification coverage than projects that we hand-coded," says Jungho Moon, senior flight control system engineer at Korean Air. "Model reuse, code generation, and reduced testing times with Model-Based Design cut development engineer-hours by 60%."

"A single flight test can cost more than $10,000," says Moon. "With Model-Based Design, we know that if we simulate correctly, the UAV will fly correctly. For example, the autopilot performance and functions can be verified with just one-third the number of flight tests that were previously required."

» *Read the full user story*

As system designs grow more complex, so does the risk of relying on prototypes to determine whether the system performance meets the design requirement: The prototype can be costly, risky to operate, or even unavailable or incomplete by the time testing needs to start. As a result, more and more engineering teams are using simulation and other testing techniques early in the design process when errors are easier and less costly to fix.

MathWorks®

Example

Embraer Legacy 500 Flight Control System

Embraer worked with customers to develop high-level requirements for the Legacy 500. They used this customer input to produce the aircraft's clean-sheet design. A principal challenge for the engineering team was translating the high-level requirements into well-written low-level requirements for the supplier who would develop the FCS software.

Test automation was key to development success. The group set up workstations to run more than 1500 test cases continuously and wrote MATLAB scripts to automate tasks in the testing process. Using Simulink Coverage the engineers analyzed model coverage and identified untested elements of the model, refining and extending their test cases until they achieved 100% coverage.

» *Read the full user story*

Simulink Test provides an integrated framework that lets you perform automated, repeatable tests throughout the design process, from desktop simulation to testing with hardware (Figure 1).

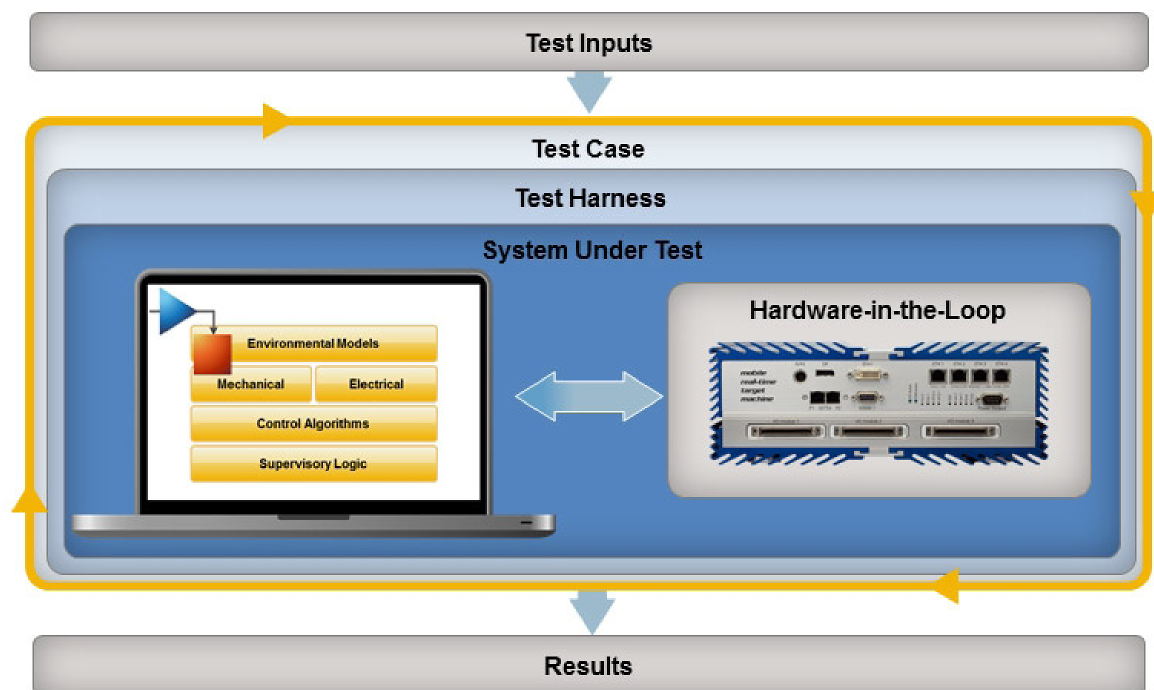

Figure 1. Framework for repeatable system testing through desktop and hardware-in-the-loop simulation.

To demonstrate this framework, we will use a flutter suppression example.

MathWorks

Example of Flutter Suppression: Test Automation from Desktop Simulation to Real-Time Testing

Flutter Suppression System Testing Goals and Setup

Flutter is the vibration caused by aerodynamic forces acting on an airplane wing. This phenomenon can have effects ranging from moderate turbulence to complete destruction of the wing. One way to control flutter is to use a control surface and attempt to dampen the vibration.

Our flutter suppression system takes three inputs: desired angle (in degrees), speed (in Mach), and altitude (in feet). Its single output is the measured deflection of the wing (in radians).

We want to test the system against two design requirements:

- The system suppresses flutter to within 0.005 radians within two seconds of a disturbance being applied.
- Flutter decays exponentially over time—specifically, the system has a positive damping ratio.

The system will need to meet these requirements under a variety of operating conditions to minimize unexpected behavior when it's deployed in the field or put into production.

Figure 2 shows the Simulink model of our flutter suppression system.

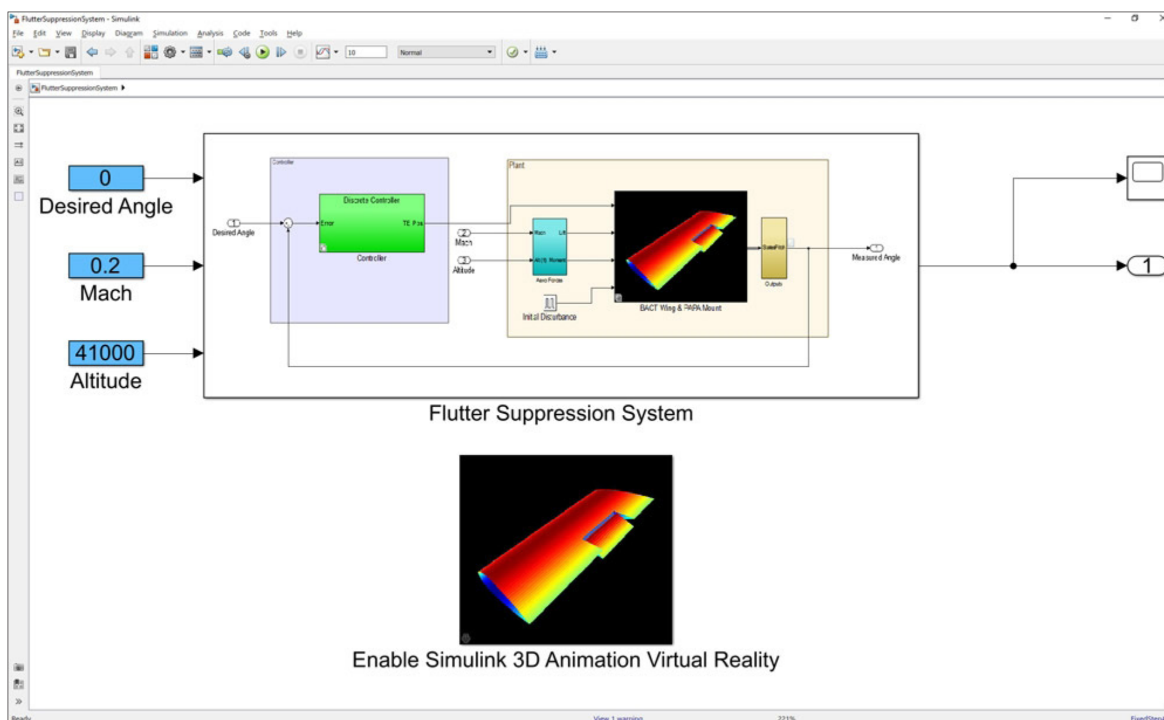

Figure 2. Aircraft flutter suppression system model.

In this model, we will introduce a disturbance after three seconds and then test whether the controller can dampen flutter at a variety of Mach and altitude points. To perform these tests, we will need the following:

- A testing environment that can monitor the flutter at each time step of the simulation
- The ability to log data to determine whether the flutter has decayed exponentially over time
- The ability to iterate over various values for Mach and altitude

MathWorks®

Setting Up a Test Harness Test Sequence Block

Control design engineers sometimes create two separate models, one for testing and another for implementation. Making sure that the base model and the implemented model remain equivalent can be challenging. Additionally, depending on the testing task, it might be necessary to customize inputs or log additional data, which will change the base model.

Simulink provides two tools that enable us to avoid this version control issue: a test harness and a Test Sequence block. The test harness is a model block diagram that is associated with a component of the system under test. It contains a separate model workspace and configuration set, yet it persists within the main model. It effectively gives us a sandbox to test our design without changing or corrupting the base model.

To create a test harness from scratch in Simulink, we simply right-click on a subsystem or select **Analysis** from the toolstrip and select **Test Harness** followed by **Create Test Harness**. We can then configure the new test harness interactively (Figure 3).

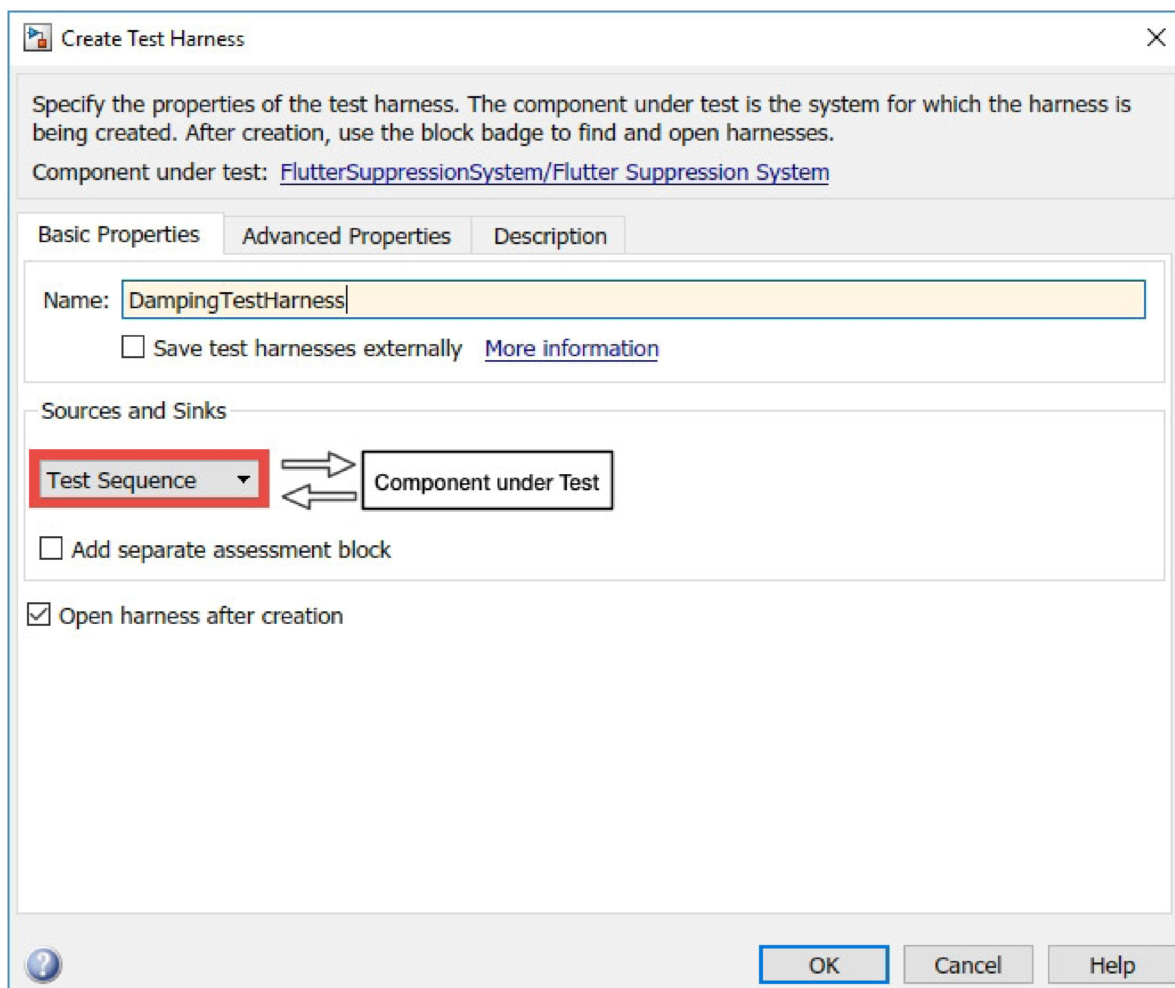

Figure 3. Test harness dialog box in Simulink.

The Test Sequence block (shown in red in Figure 4) uses MATLAB as an action language (Figure 5). It allows you to transition between test steps conditionally while evaluating a component under test. You can use conditional logic, temporal operators (such as before and after), and event operators, such as *hasChanged and hasChangedFrom*.

MathWorks®

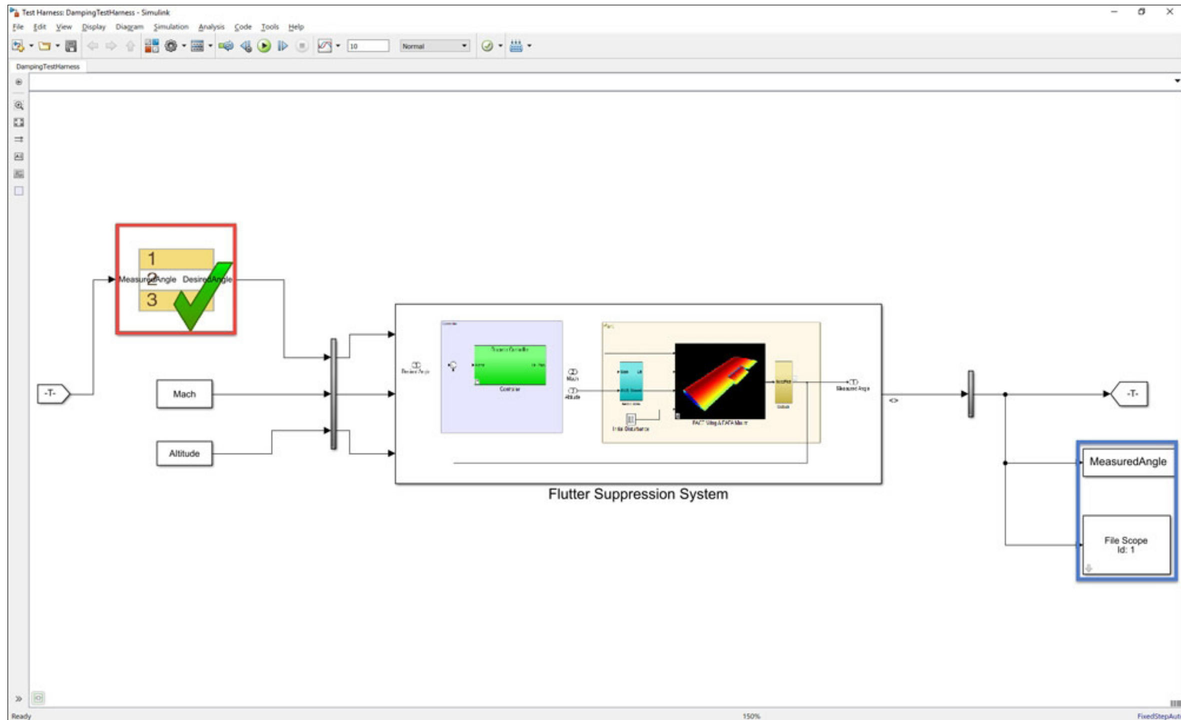

Figure 4. Test harness model with Test Sequence block (red) and data logging (blue).

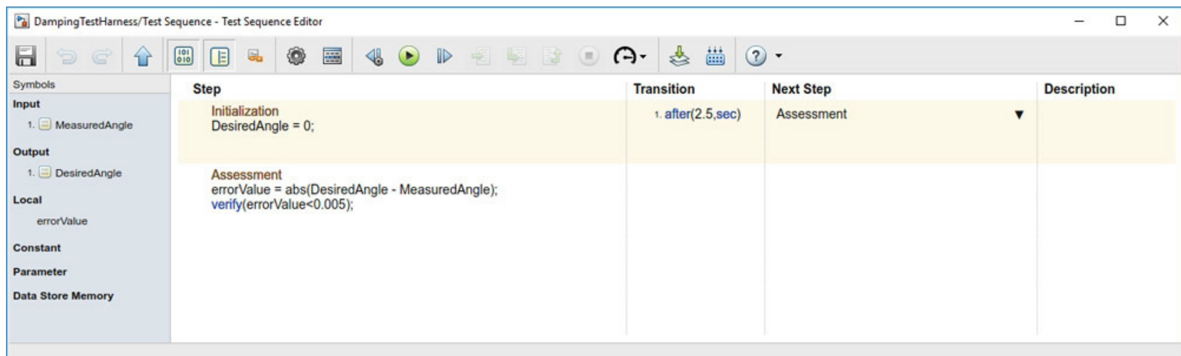

Figure 5. Test sequence with MATLAB action language.

The first requirement dictates that the flutter shall be bounded within two seconds of the initial disturbance being applied. We incorporate a Test Sequence block to implement this test case. We set the desired position to 0 radians, and after five seconds, calculate the error at each time step. The **`verify`** function allows us to log whether this condition was met at each time step.

To calculate the damping ratio, we will log data both in simulation and in real time through the To Workspace and File Scope blocks indicated in blue in Figure 4.

Creating and Running Desktop Simulations

The system under test has two inputs: Mach and altitude. The requirements specify that the system should be tested under a variety of operating conditions. We can vary the conditions by using the Simulink Test Manager to create an automated test suite. The test suite will enable us to automatically test both requirements over varying input conditions and generate reports on whether they pass or fail. This test suite can be rerun as the design changes.

MathWorks

Using the Test Manager, we create a new simulation test for the flutter suppression system, add a description to identify the purpose of the test, and link it to the requirements. Finally, we specify some operating conditions for Mach and altitude using scripted iterations (Figure 6).

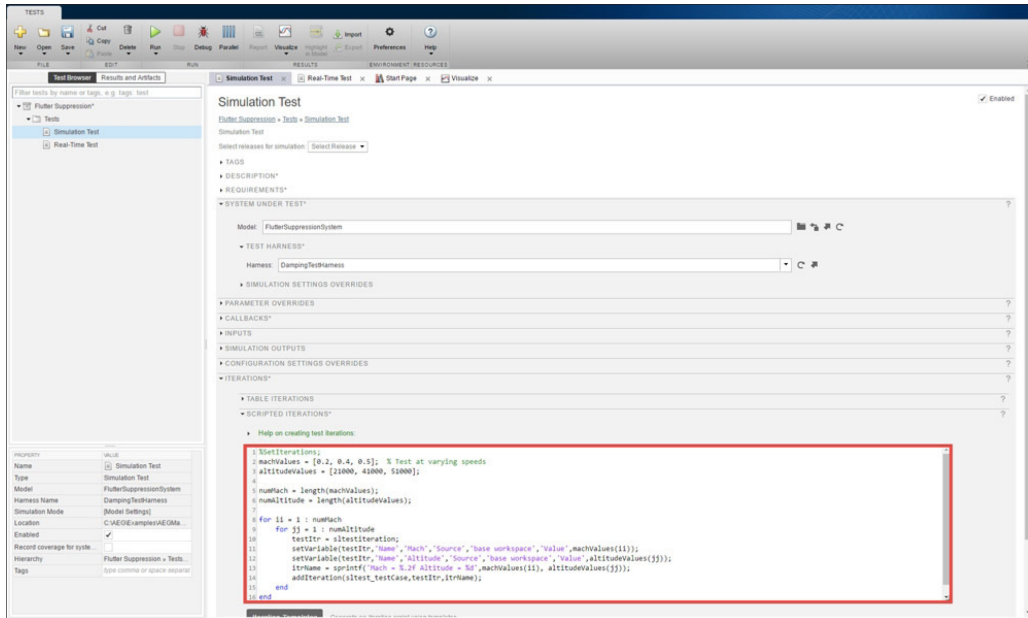

Figure 6. Setting iterations for different Mach and altitude values using a MATLAB script in the Test Manager interface.

We will now use this test automation workflow to test our two requirements. The first requirement was handled with the Test Sequence block. Recall the use of the **`verify`** function. If at any point the verify criteria fail, the overall test will fail.

For the second requirement, we added blocks to log simulation data. We need to do some data analysis on the measured angle to determine whether the test has passed or failed. This analysis can be done through the cleanup callback that executes after each simulation run (Figure 7). We can leverage previous data analysis work to do an exponential fit and declare pass or fail based on the fit parameters.

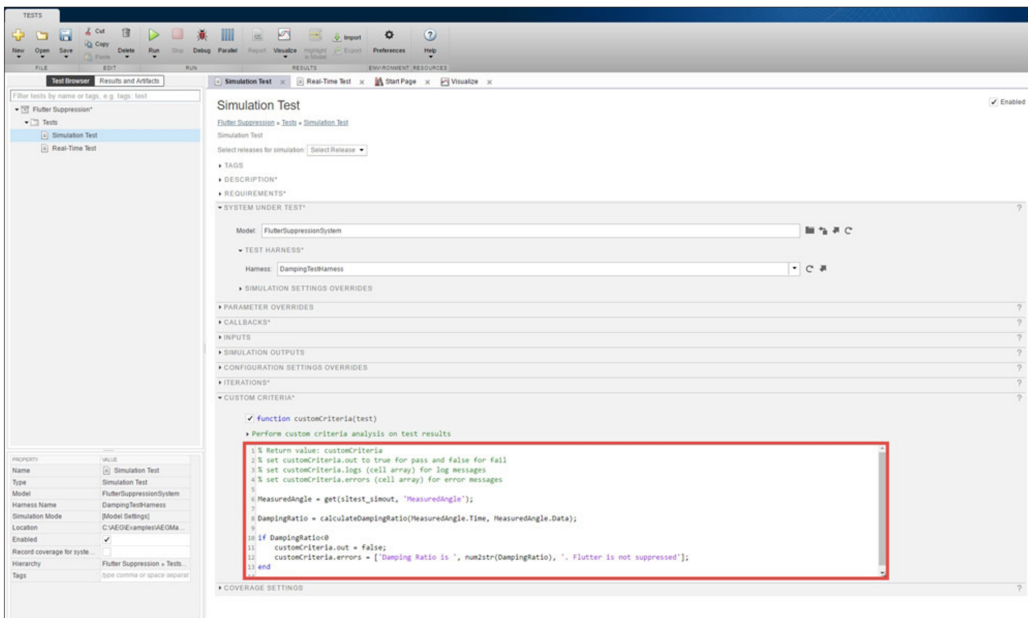

Figure 7. Setting up the cleanup callback for custom criteria in the Test Manager interface.

MathWorks®

From here, this test will automatically exercise our system with the operating conditions we have specified. We can see the test results in the Results and Artifacts pane (Figure 8). We can check the output of the **`verify`** statement and identify times when the test criteria were not evaluated, when it passed, and when it failed. Additionally, we can visualize the logged Commanded and Measured Angle data.

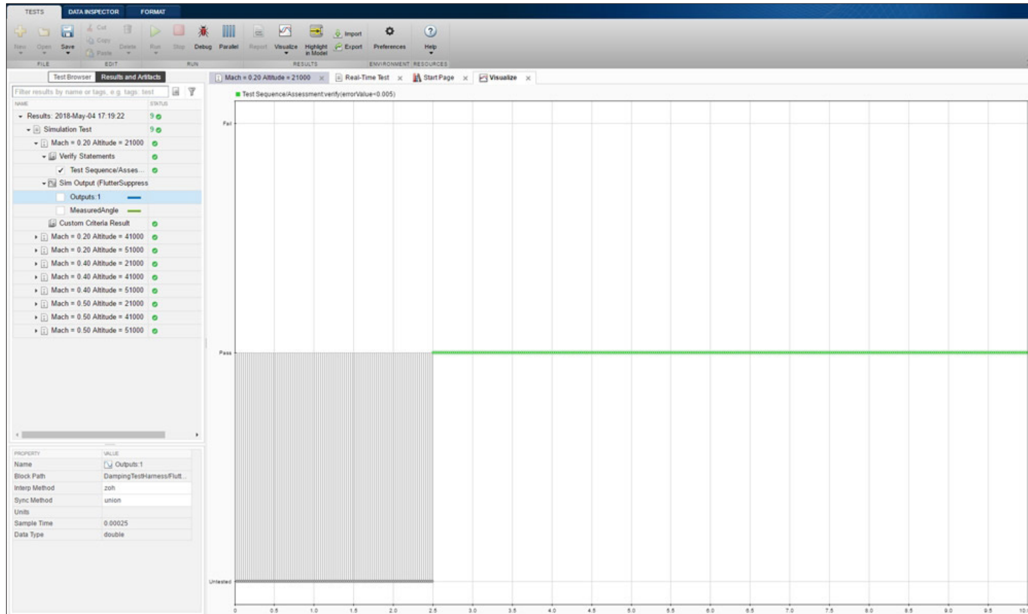

Figure 8. Output of **`verify`** *statement (requirement 1).*

The system passed all the tests in simulation, but let's take a closer look to make sure the requirements have been met. Recall the requirement that the flutter shall be bounded within two seconds of a disturbance being applied. Given that the disturbance is applied three seconds into the simulation, it is expected that the **`verify`** statement is untested for the first five seconds of simulation. From there on we can see that the test passed.

The measured angle data shows that the flutter is not only bounded but also decaying, which fulfils the second requirement (Figure 9).

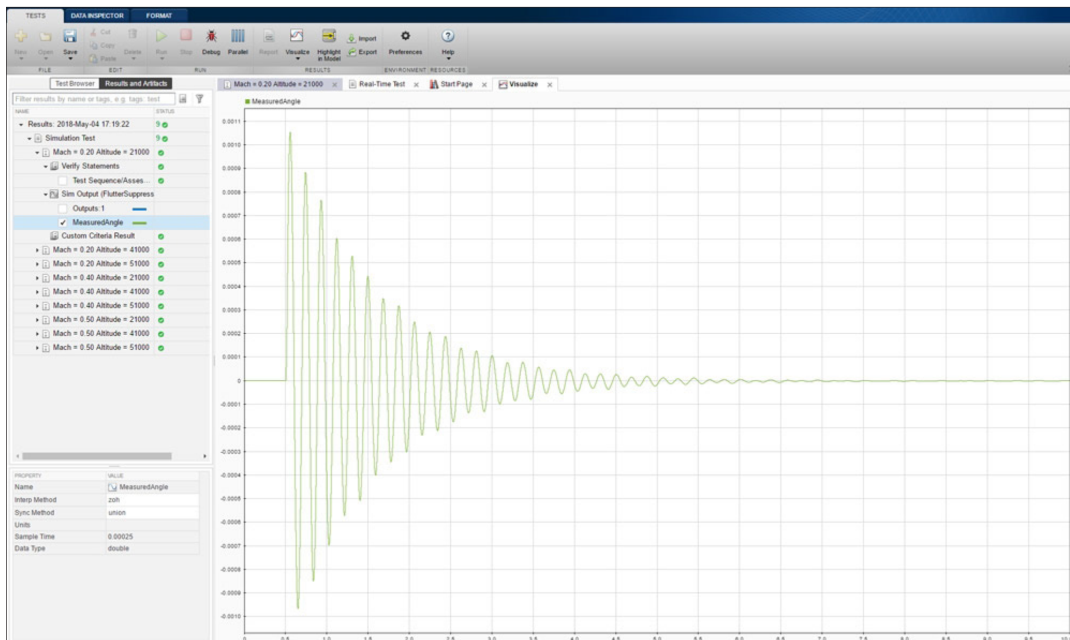

Figure 9. Measured angle output.

Real-Time Testing

We're now ready to test with hardware using hardware-in-the-loop (HIL) simulation. The goal of HIL is to simulate the plant model dynamics in real time while interfacing with the embedded controller that will be used in the field. For HIL we'll be using a laptop computer running Simulink, a Speedgoat real-time target, and an embedded controller connected through analog and discrete I/O (Figure 10).

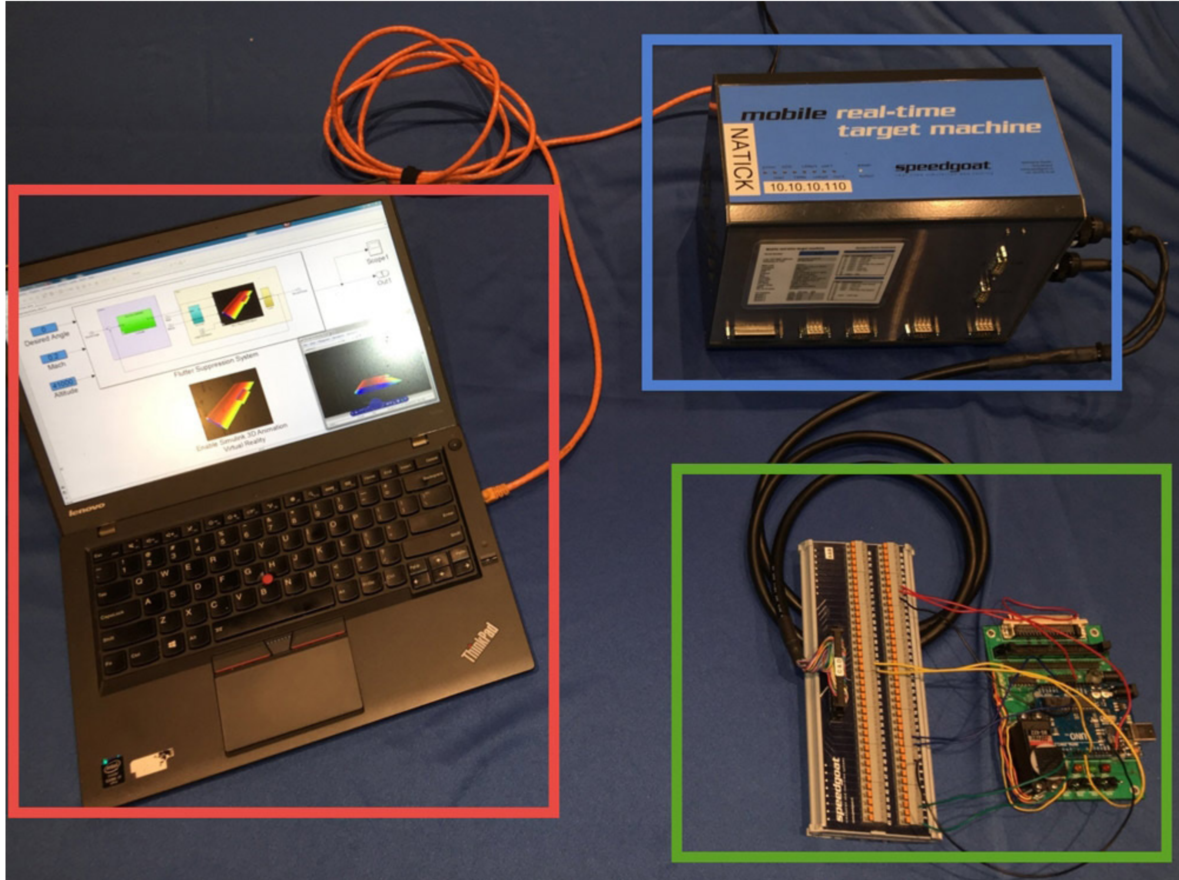

Figure 10. Test hardware: Windows PC (red), Speedgoat target (blue), and embedded controller (green).

On the laptop, we generate C code from the model and compile it to a real-time application, which is downloaded to the Speedgoat real-time target computer over an Ethernet connection. The generated code includes the plant model dynamics, the drivers for the I/O required to communicate with the controller, and the test assessment block containing the **`verify`** function.

A key difference between the simulation and real-time tests is that we remove the simulated control system and use the control system implemented on the embedded processor. We can then test the implemented control system with its field inputs and outputs at its natural operating frequency.

We'll now use the Test Manager to create a real-time test (Figure 11).

MathWorks®

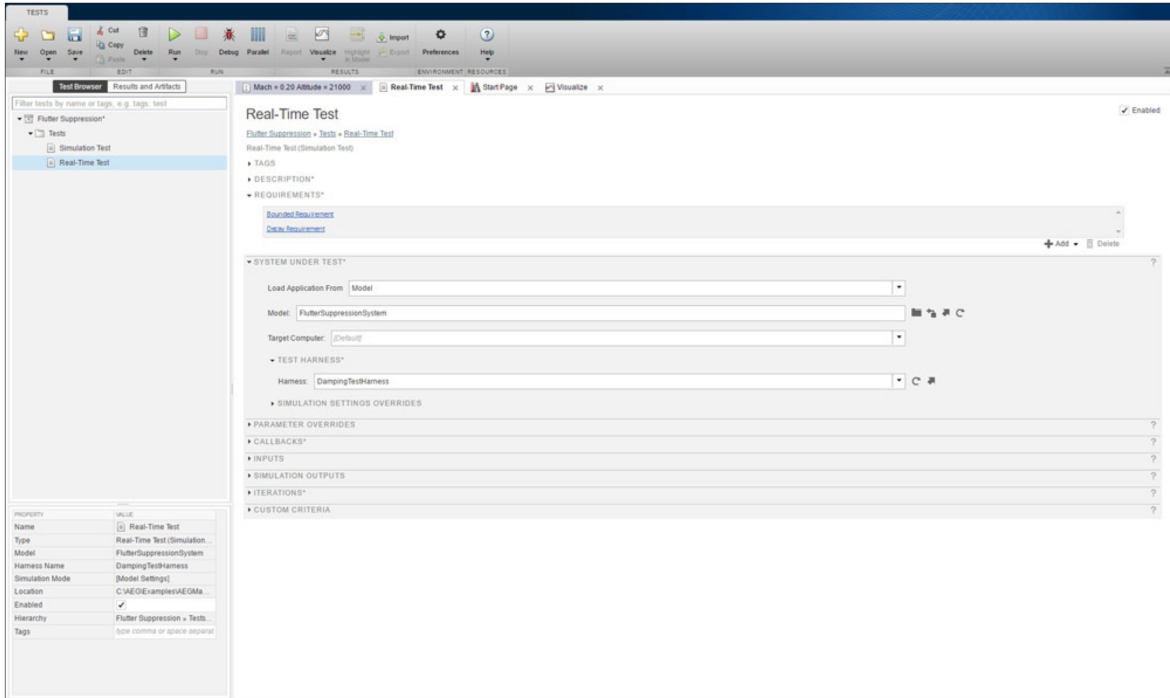

Figure 11. Test Manager interface for setting up a real-time test.

In the real-time test we populate the same fields as before, plus an additional field for the system under test, Target Computer. This field can be used to specify the real-time target computer that the test will run on.

We will test the requirements under the same varying operating conditions as before, and we'll do the same data analysis on the measured angle to determine whether the test has passed or failed. We can view the results in the Test Manager (Figure 12).

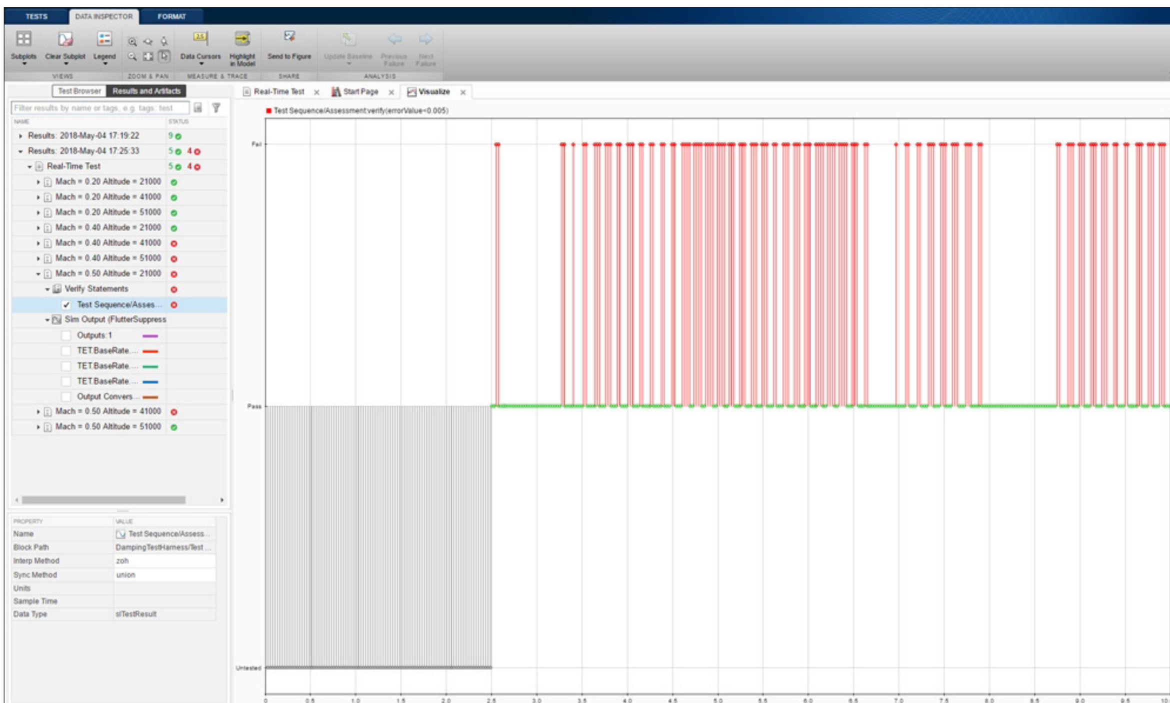

Figure 12. Real-time test results.

MathWorks

We find that the real-time test fails for some of the test conditions. As Figure 12 and Figure 13 show, the **`verify`** statement fails at various points and the measured angle does not decay, resulting in an unstable system.

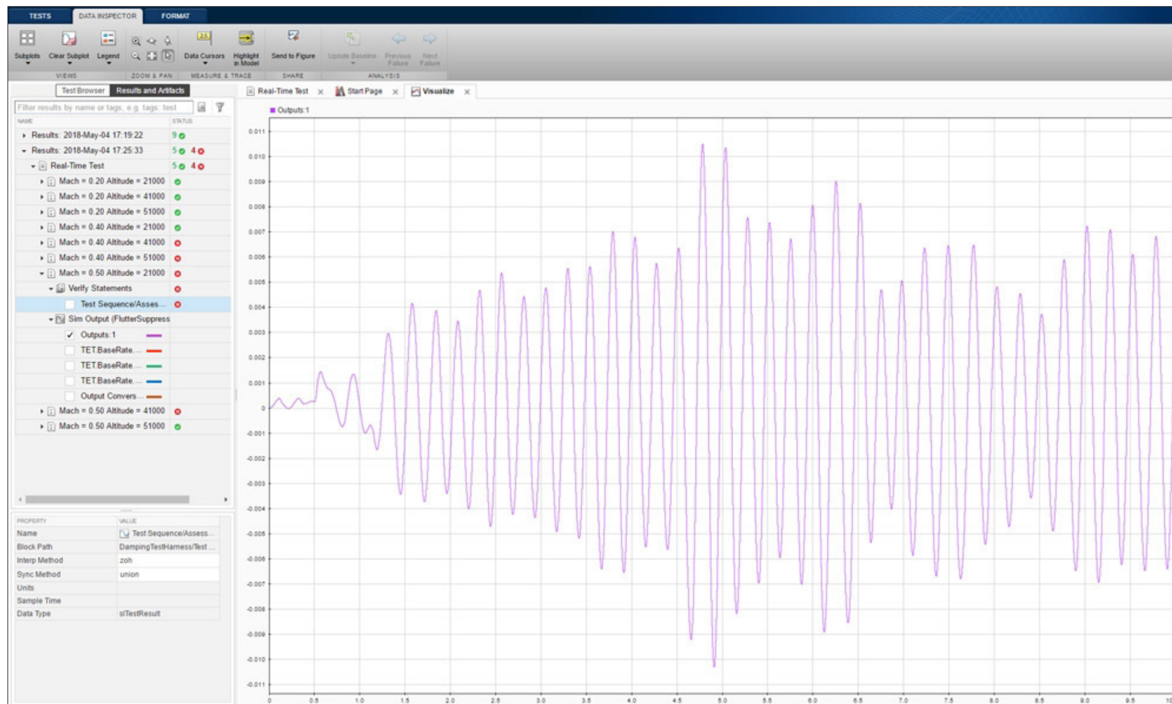

Figure 13. Measured angle output from real-time test.

What was different about this test that caused it fail when it passed in simulation?

There are several reasons why these tests failed, and these reasons highlight the importance of testing with hardware. First, in the simulation model, the controller and plant were directly connected using double-precision values as the interface. The connection between the real-time simulation and the production controller is through digital and analog signals, and so right away we lose precision, due to quantization, in the interface. Second, the simulation testing did not account for the real-world latencies that exist in actual systems. Third, it is possible that the control design that was tested in simulation was not implemented correctly in the production controller.

Even though these tests did not pass and we still have work to do, we want to create a report to send to colleagues. Using the report generator from within the Test Manager, we create a report that documents who performed the tests, the test criteria, the links to the requirements, and a summary of the results.

As the system design evolves, we can use the Test Manager and the tests we have already created to automatically perform regression tests and generate the reports for those, as well.

Benefits of Test Automation

This example demonstrated a framework for testing systems against requirements throughout the design process. Using Simulink Test, we were able to create repeatable tests that verify our requirements in two different ways: one with the test criteria in the control loop and another with more conventional postprocessing. We could link the tests to the design requirements to ensure traceability and automatically test the requirements against a variety of inputs.

MathWorks®

As the test results demonstrated, simulation and real-world behavior do not always match—hence the need to test designs early and with the same interfaces used in the field. Lastly, we can use Simulink Real-Time™ with Simulink Test to create real-time tests that verify the design without the real-world system or the risk of damaging a prototype.

Conclusion

MATLAB and Simulink provide many advantages to aerospace companies practicing Model-Based Design, from managing requirements to automating tests. Through Model-Based Design, companies reduce misinterpreted requirements, eliminate defects from hand-coding tests, and find and fix errors earlier in the development cycle.

Learn More

- *Model-Based Design for DO-178C Software Development with MathWorks Tools* - Video Series
- *Model-Based Design for Production Real-Time Embedded Systems* - Consulting Services
- *Using Qualified Tools in a DO-178C Development Process* - Video Series
- *Model-Based Design & Verification for DO-178C Using Simulink and the LDRA Tool Suite* - Video
- *Verifying Models and Code for High-Integrity Systems* - Article

© 2019 The MathWorks, Inc. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See mathworks.com/trademarks for a list of additional trademarks.
Other product or brand names may be trademarks or registered trademarks of their respective holders.