Testing and Validation of Simulink Models with Reactis

Build better embedded software faster. Generate tests from Simulink models. Detect runtime errors. Execute and debug Simulink models. Track coverage. Automate functional testing of requirements. Check conformance of code to model.
About Reactive Systems, Inc.

Reactive Systems, founded in 1999, is a privately held company based in Cary, NC. The company’s Reactis product line provides automated testing and validation tools to support the development of embedded control software. Reactis, Reactis for C Plugin, Reactis for EML Plugin, and Reactis for C support model-based design with Simulink, Stateflow, Embedded MATLAB, and C code. Reactis Tester automatically generates comprehensive yet compact test suites from a Simulink model or C code. Reactis is used at companies worldwide in the automotive, aerospace, and heavy-equipment industries.

Reactive Systems, Inc.
341 Kilmayne Dr.
Suite 101
Cary, NC 27511
USA

Tel.: +1 919-324-3507
Fax: +1 919-324-3508

Web: www.reactive-systems.com
E-mail: info@reactive-systems.com
Abstract

This white paper discusses how the Reactis® a automatic test generation tool may be used to validate Simulink® b models of embedded control software and to test for conformance of code to Simulink models. Reactis Tester automatically generates test cases that stress the model. The test generation often uncovers runtime errors in Simulink models. The generated tests aim to maximize coverage with respect to a number of test coverage metrics including Modified Condition/Decision Coverage (MC/DC). Reactis Simulator is a simulation environment for Simulink models that enables the user to execute and debug models and to track coverage during test execution. Reactis Validator enables an engineer to formalize model requirements as assertions and perform an automatic search for requirement violations. Validator performs these checks by thoroughly simulating the model with the goal of violating assertions. When an assertion fails, Validator returns a test that highlights the problem. Test suites generated by Reactis serve as a testing oracle to determine if source code conforms to the behavior of a Simulink model. The Reactis for C Plugin integrates seamlessly with Reactis to offer white-box testing for the C code portions of models (C Caller blocks, S-Functions, and Stateflow® custom code). Similarly, the Reactis for EML Plugin offers white-box testing of Embedded MATLAB within a model.

aReactis is a registered trademark of Reactive Systems, Inc.
bSimulink and Stateflow are registered trademarks of The MathWorks, Inc.

1. Introduction

Reactis helps engineers build better software faster by automating many verification and validation tasks in a model-based design process.

Over the past decades, many engineering organizations have deployed model-based design to address the exploding complexity of embedded control software. In model-based design, executable visual models of embedded control software are developed in advance of system implementation. The models may be used to drive the development of control software, and may also serve as a basis for software and system testing. One benefit of model-based design is that it allows engineers to begin debugging and validation activities at design time, when the cost of detecting and dealing with design defects is much smaller than at the software and system implementation level. Another is that models may be used as a baseline for assessing implementation behavior during system testing and validation. For these reasons, judicious use of modeling can lead to quite dramatic over-all reductions in the cost of control-system development, especially when robust tool support is available.

The Reactis tool suite of Reactive Systems, Inc., substantially enhances the gains organizations realize from model-based design by automating many testing and validation activities.
Reactis works with models implemented in the Simulink/Stateflow notation offered by MathWorks. Using Reactis, engineers may:

- generate tests from a model that thoroughly exercise the model (structural testing);
- find runtime errors (e.g. overflow errors, divide-by-zero errors) in a model;
- execute the model and track coverage (e.g. MC/DC);
- perform functional tests to check whether or not a model can violate its requirements;
- use a Reactis test suite as an oracle to check whether code conforms to a model.

Reactis also includes an array of sophisticated model debug features (e.g. breakpoints, scopes, reverse execution). In this paper, we discuss how Reactis and model-based design may be used to automate different verification and validation activities in your software quality assurance process. In particular, we show how the tool may be used to develop more robust models, how it can streamline software and system testing, and how it may be used to support the reverse-engineering of models from legacy code.

2. An Overview of Reactis

A model-based design environment involving Reactis, Simulink and Stateflow is depicted in Figure 1. Reactis contains three core components: Tester, which provides automated test generation from models; Simulator, which enables you to visualize model execution to debug models and track coverage; and Validator, which offers automated checks of Simulink models for violations of user-specified requirements. The remainder of this section describes these components in more detail.

Figure 1: Reactis is used in a model-based design process using Simulink/Stateflow models. Reactis is a standalone application that reads the .slx files produced by the MathWorks environment.

2.1. Reactis Tester

Figure 2 shows that Reactis Tester offers automatic test generation from Simulink models. The generated test suites provide comprehensive coverage of different test coverage metrics -
including the Modified Condition/Decision Coverage (MC/DC) test coverage measure mandated by the US Federal Aviation Administration (FAA) in its DO-178/B guidelines - while at the same time minimizing redundancy in tests. Each test case in a test suite consists of a sequence of inputs fed into the model as well as the responses to those inputs generated by the model. The automatically generated test data may then be used for a variety of purposes, including the following:

**Implementation conformance.** The tests may be applied to source-code implementations of models to ensure conformance with model behavior.

**Model testing and debugging.** The tests may be run on the models themselves to detect run-time errors and to study and revise model behavior.

**Regression testing of models.** The tests may be run on new versions of models to flag differing behaviors in new versions.

**Reverse engineering of models from source.** Tests may be generated from models derived from legacy code in order to check conformance between model and legacy code.

Reactis Tester enables engineers to maximize the effectiveness of testing while reducing the time actually spent on testing.

![Diagram](https://via.placeholder.com/150)

Figure 2: Reactis Tester automatically generates comprehensive yet compact test suites.

The structure of a Tester-generated test is shown in Figure 3. A test is constructed by simulating the model and capturing the top-level inputs and outputs. A test may be viewed as a matrix in which each row corresponds to either an inport or outport and each column represents a simulation step. A test suite consists of a set of tests. When running a test suite, the model is reset to its initial state after one test completes and before the next test begins.

Test suites are constructed by simulating a model and recording the input and output values at each step. The model computes the outputs at each step, but several approaches are possible for selecting the input values to drive simulation. The input data could be captured during field testing or constructed manually by an engineer. If such data is available, it can
be imported into Reactis from a comma separated value (CSV) file. Alternatively, the inputs could be generated randomly; however, this approach yields tests with poor coverage.

Reactis Tester employs a patented technique called guided simulation to generate quality input data automatically. The idea behind this approach is to use algorithms and heuristics to automatically generate inputs that cause coverage targets (i.e. model elements that the user wants to ensure are executed at least once) that have not yet been covered to be executed. Reactis currently allows you to track several different classes of coverage targets (also called coverage criteria or coverage metrics). Some of the test coverage metrics supported by Reactis involve only Simulink, some are specific to Stateflow, and the remaining are generic in the sense that they include targets within the Simulink, Stateflow, C code, or Embedded MATLAB portions of a model.

**Simulink-specific:** Conditional subsystems. Branches of the following blocks: Dead Zone, Logical Operator, MinMax, Multiport Switch, Relational Operator, Saturation, Switch. Lookup tables.

**Stateflow-specific:** States. Condition actions. Transition actions. Child State Exit via Parent Transition (CSEPT).

**Generic:** Decisions from logic blocks in Simulink, transition segments in Stateflow, or in C or Embedded MATLAB code a boolean-valued expression used to determine which execution path to follow. Conditions (the atomic predicates that are the building blocks of decisions). Modified Condition/Decision Coverage (MC/DC) targets. Multiple Condition Coverage (MCC) targets. Boundary Values of top-level inputs.
2.2. Reactis Simulator

Reactis Simulator enables you to visualize model execution and debug Simulink models. Simulator’s user interface is similar to those of traditional debuggers for programming languages: it allows you to step through the execution of models by hand as well as set break points. Simulator also supports reverse execution, the replay of tests generated by Reactis Tester, the graphical display of different coverage metrics, the display of data item values, and the capability to fine-tune automatically generated test suites. Stepping between the Simulink, Stateflow, C code, and Embedded MATLAB portions of a model is seamless.

2.3. Reactis Validator

Reactis Validator performs automated searches of models for violations of user-specified requirements. If Validator finds a violation, it returns a test that leads to the problem. This test may then be executed in Reactis Simulator to gain an understanding of the sequence of events that leads to the problem. Validator enables the early detection of design errors and inconsistencies and reduces the effort required for design reviews. Some checks that may be performed with Validator include the following.

- Will a car’s brake pedal always deactivate the cruise-control?
- Will a plane’s thrust reversers ever engage when the aircraft is airborne?
- Will a medical device ever deliver an unsafe dose of radiation?

3. Advanced Model Validation

The model-validation capabilities of Reactis help engineers detect bugs earlier, when they are less costly to fix.

A primary benefit of model-based design is that it allows the detection and correction of system-design defects at design (i.e. modeling) time, when they are much less expensive and time consuming to correct, rather than at system-implementation and testing time. Moreover, with proper tool support, the probability of detecting defects at the model level can be significantly increased. In this section, we elaborate on the advanced model-validation capabilities of Reactis that help engineers build better models.

3.1. Debugging with Tester and Simulator

Reactis Tester and Simulator support model debugging through the automatic generation of test suites that thoroughly exercise the model under investigation (Reactis Tester), and through the visualization of tests as they are executed on the model (Reactis Simulator). One such usage scenario of Tester and Simulator is shown in Figure 4. Since Tester’s guided-simulation test-generation algorithm thoroughly simulates a model during test generation, it often uncovers runtime errors. For example, overflows, missing cases, and bad array indexes can be discovered. Note that this type of error is also detected when running simulations
in Simulink; however, since Tester’s guided-simulation engine systematically exercises the model much more thoroughly than random simulation can, the probability of finding such modeling problems is much higher using Reactis.

Tester-generated tests may be executed in Simulator, which offers a number of useful model debugging features; some of these are illustrated in Figure 5. The figure includes a screenshot of Reactis invoked on a Simulink/Stateflow model of an automotive cruise control system. This example is one of several example applications included with the Reactis distribution. The main window in the figure depicts the model hierarchy on the left and an execution snapshot of a Stateflow diagram from the model on the right. Reactis allows you to choose between three distinct sources of input values when visualizing model execution:

1. Input values may be read from a Tester-generated test.
2. They may be generated randomly.
3. They may be supplied interactively by the user.

As depicted, input values come from Test 6 of a Tester-generated test suite. The other model-debugging facilities illustrated in the figure are as follows.

- You may take forward or reverse execution steps when simulating model behavior.
• You may dynamically open scopes to view the values of Stateflow variables or Simulink blocks and signals. An example scope, depicting how the value of Stateflow variable mode varies over time, is shown. This scope was opened by right-clicking on the mode variable in the diagram panel and selecting “Open Scope.”

• You may query the current value of any Simulink block or signal, Stateflow variable, or C variable by hovering over it with the mouse.

• You may set execution breakpoints. In the example, a breakpoint has been set in state Active. Therefore, model execution will be suspended when control reaches this state during simulation, allowing the user to carefully examine the model before continuing simulation. Simulation may be resumed in any input mode, i.e. reading inputs from the test, generating them randomly, or querying the user for them.

• As shown in the execution snapshot, the current simulation state of the model is highlighted in green and portions of the model that have not yet been exercised during simulation are highlighted in red for easy recognition.

Figure 5: Reactis Simulator offers an advanced debug environment for Simulink models.

3.2. Validating Models with Validator

The advanced model-validation capabilities of Reactis are implemented in Reactis Validator. Validator searches for defects and inconsistencies in models. The tool lets you formulate a
requirement as an assertion, attach the assertion to a model, and perform an automated search for a simulation of the model that leads to a violation of the assertion. If Validator finds an assertion violation, it returns a test that leads to the problem. This test may then be executed in Reactis Simulator to gain an understanding of the sequence of events that leads to the problem. Validator also offers an alternative usage under which the tool searches for tests that exercise user-defined coverage targets. The tool enables the early detection of design errors and inconsistencies and reduces the effort required for design reviews.

Figure 6: Reactis Validator automates functional testing.

Figure 6 shows how engineers use Validator. First, a model is instrumented with assertions to be checked and user-defined coverage targets. In the following discussion, we will refer to such assertions and coverage targets as Validator objectives. The tool is then invoked on the instrumented model to search for assertion violations and paths leading to the specified coverage targets. The output of a Validator run is a test suite that includes tests leading to objectives found during the analysis. Validator objectives may be added to any Simulink system or Stateflow diagram in a model.

Two mechanisms for formulating objectives in Simulink models are supported: Expression objectives are C-like boolean expressions. Diagram objectives are Simulink / Stateflow observer diagrams.

Diagram objectives are attached to a model using the Reactis GUI to specify a Simulink system from a library and “wire” it into the model. The diagrams are created using Simulink and Stateflow in the same way standard models are built. After adding a diagram objective to a model, the diagram will be included in the model’s hierarchy tree, just as library links are in a model. Note that the diagram objectives are stored in a separate library and the .slx file containing the controller model remains unchanged.

Because of its sophisticated model-debugging capabilities, the Reactis tool suite provides significant added value to the MathWorks Simulink/Stateflow modeling environment. The great virtue of model-level debugging is that it enables engineers to debug a software design before any source code is generated. The earlier logic errors are detected, the less costly they are to fix.
4. Testing Code Against Model

The automatic test-generation and execution offered by Reactis enables engineers to easily check whether an implementation conforms to the behavior specified in a model.

The benefits of model debugging and validation have been discussed above. A question that immediately presents itself is: How can the effort expended on these activities be “reused” to support the testing of system implementations? This is the question addressed in this section.

4.1. Software Testing

A crucial aspect of the tests generated by Reactis Tester is that they also store model outputs. Therefore, these tests encode all the information needed to ensure that model-derived source code conforms to its model. Reactis-driven source-code testing proceeds as follows:

1. For each test in the suite, execute the software using the input values contained in the test.
2. Compare the output values produced by the software with those stored in the test.
3. Record any discrepancies.

This methodology is referred to as model-based software testing or back-to-back testing. Its key advantage is that the model serves as an “oracle” for testing purposes: the outputs produced by the model can be used as a basis for assessing those generated by the software. If the software does not agree with the model, then the developer can assume that the problem lies within the source code.

The net effect of model-based testing with Reactis is better-quality software at a lower cost. Because good test data is generated and run automatically, less engineer time is required to create and run tests. Because the tests are thorough, the probability of finding bugs is maximized. Because the test suites are compact they may be run quickly. In sum, Reactis dramatically reduces the costs of testing embedded control software.

Figure 7 illustrates how the Reactis tool suite can provide advanced model-based testing of source code. As the figure indicates, the model-based testing protocol supported by Reactis is as follows:

1. The developer provides as input to Reactis an .slx file representing the validated Simulink/Stateflow model of the system under development.
2. Reactis Tester is used to automatically generate a test suite that thoroughly exercises the given model according to the various coverage metrics supported by Reactis.
3. The developer may deploy Reactis Simulator to visualize test execution and to fine tune the tests in the test suite to further improve model coverage.
4. The test suite and the software implementing the model are fed as inputs into a test harness to automate the source-code testing process. Note, that if Reactis for C is used as the test harness it will read the native test suite format, execute the tests on the C code, flag any runtime errors, track coverage within the C code, and compare the outputs computed by the C code against those generated by the model and stored in the tests.

5. By comparing the outputs produced by the software against the model-generated outputs (stored in the test), deviations in the behavior of the source code from the model are readily detectable and help the developer ensure that the source code conforms to the model.

6. Testing concludes when the source code passes all the tests in the test suite.

4.2. System Testing

After software testing, the next step in certifying a system is to compile the code and test the resulting executable on the platform, including the target microprocessor and associated system software, on which it will eventually be deployed. Such testing is often referred to as system, or integration, testing.

System testing typically involves the use of hardware-in-the-loop (HIL) simulation tools. These HIL tools are expensive, and thus using them as efficiently as possible can promote significant cost savings.
Provided that system-level models are given in Simulink / Stateflow, Reactis can greatly facilitate system testing. As in the case of software testing, test engineers can use Reactis Tester and Reactis Simulator to generate thorough yet compact test suites from these models and feed the test data into their HIL environments in order to check system behavior against model behavior. The compactness of Reactis-generated tests means that expensive HIL hardware need not be tied up with long test runs in order to get precise insights into system behavior.

How the Reactis-generated test data may be used in HIL testing will in general depend on the HIL environment used. HIL tools typically provide a scripting facility for defining test runs. Reactis exports test data in several easy to parse formats (comma separated value, CSV, for example) to simplify the writing of scripts to read Reactis-generated test data into an HIL environment.

---

5. Reverse Engineering Models from Legacy Code

The automatic test-generation and execution offered by Reactis enables engineers to easily check whether a reverse-engineered model conforms to the behavior of legacy code.

Model-based design technology can also play an important role with regard to legacy systems. Such systems are often poorly documented and very difficult to modify to meet evolving system requirements due to the fragile nature of the underlying code. It would benefit developers to have a precise and unambiguous model of the behavior of a legacy system for which they were responsible. Such a model would serve as a formal and executable specification of the legacy system, thereby facilitating system maintenance, documentation, and evolution. The focus of this section is on how Reactis can indeed be used to derive, or “reverse engineer”, models from code.

Figure 8 illustrates the process one would follow in order to use Reactis to reverse engineer models from legacy code. Reverse engineering proceeds as follows.

1. The Simulink / Stateflow modeling environment is used to draft a model of the legacy code.

2. The resulting .slx file is fed into Reactis Tester which then automatically generates a test suite from the model. The result is an .rst file (a Reactis test-suite file). The generated test suite thoroughly exercises the draft model according to various coverage metrics. Example coverage metrics include Stateflow state and transition coverage, Simulink branch coverage, and MC/DC coverage. Tester also eliminates redundancy in the test suite it generates in order to eliminate unnecessary test steps.

3. Reactis Simulator may then be used to visualize the execution of the tests produced by Reactis Tester, and also to fine-tune the test suite to further improve model coverage.

4. The test suite is exported as a CSV file and given as input to a test harness to automate the process of applying the tests to the legacy code.
5. By comparing the outputs produced by the software and the model on the tests in the test suite, deviations in the behavior of the model from the legacy code are readily detectable and can be used to guide the user in refining the model to ensure that it faithfully captures the behavior of the legacy system.

6. Reverse engineering of the model concludes when the code passes all tests generated from the model.

The beauty of having a model to go along with a legacy system is that the model serves as a formal and executable specification of the code, thereby easing the tasks of code maintenance, documentation, and evolution.

6. Conclusions

In this paper we have described several ways that the Reactis tool suite provides significant added value to the market-leading MathWorks Simulink/Stateflow modeling environment. Through its sophisticated model-debugging capabilities, Reactis enables engineers to debug a software design before system implementation is undertaken. The earlier design errors are detected, the less costly they are to fix, so better model debugging can reduce overall software costs. We also discussed how the comprehensive yet compact test suites produced by Reactis can dramatically reduce the costs of checking for conformance between a model and system and of reverse-engineering a model from existing control software. By automating tasks that currently require significant manual effort, Reactis cuts development costs. By enabling more
thorough testing and validation to be undertaken, it also enables errors to be detected and fixed before systems are fielded and therefore cuts recall and liability costs.

Reactis is available now from Reactive Systems, Inc. Please see the Company’s web site at reactive-systems.com for ordering information and for instructions on how to download a free 30-day evaluation copy of the software.