SimCoTest: A Test Suite Generation Tool for Simulink/Stateflow Controllers

Reza Matinnejad, Shiva Nejati, Lionel C. Briand
SnT Centre, University of Luxembourg, Luxembourg
{reza.matinnejad,shiva.nejati,lionel.briand}@uni.lu

Thomas Bruckmann
Delphi Automotive Systems, Luxembourg
{thomas.bruckmann}@delphi.com

ABSTRACT

We present SimCoTest, a tool to generate small test suites with high fault revealing ability for Simulink/Stateflow controllers. SimCoTest uses meta-heuristic search to (1) maximize the likelihood of presence of specific failure patterns in output signals (failure-based test generation), and to (2) maximize diversity of output signal shapes (output diversity test generation). SimCoTest has been evaluated on industrial Simulink models and has been systematically compared with Simuilnk Design Verifier (SLDV), an alternative commercial Simulink testing tool. Our results show that the fault revealing ability of SimCoTest outperforms that of SLDV. Further, in contrast to SLDV, SimCoTest is applicable to Simulink/Stateflow models in their entirety. A video describing the main features of SimCoTest is available at: https://youtu.be/YnXgveiGXEA

1. INTRODUCTION

The Simulink/Stateflow (SL/SF) environment is widely used for model-based design and development of control software systems in the Cyber Physical Systems (CPSs) domain [13]. In this domain, system models are largely described in terms of mathematical models [3] and are typically developed using SL/SF. These models capture software controllers as well as the system under control which consists of physical objects and processes [5].

The primary goal of control system modeling is *simulation*, i.e., design time testing of system models. Through simulation, SL/SF models are executed, i.e., the mathematical formulas are numerically solved for some given inputs to generate outputs. SL/SF models, after being sufficiently tested, are converted into code. To be able to generate code from these models, two major adaptations typically take place: First, continuous-time operators are replaced with their discrete-time counterparts [14]. Second, the floating-point computations, such as square roots and trig functions, are replaced with their corresponding fixed-point approximations. The former is needed because, in contrast to the simulation environment, code runs in real time and receives input data as discrete sequences of events. The latter is necessary since the target hardware platforms, e.g., embedded processors in cars, mostly support fixed-point computations only [14].

Testing SL/SF models is complicated by a number of factors that distinguish testing these models from the mainstream software testing applied to code: First, the inputs and outputs of SL/SF models are *signals*, i.e., variables capturing evolution of values over time. Second, SL/SF models have continuous-time behaviors (described using differential equations) since they are expected to capture and continuously interact with the physical world. Third, *automatable* test oracles for SL/SF models are not always available in practice. More precisely, specifications and formal properties, from which assertions (explicit oracles [1]) are derived, are expensive and not generally amenable to capturing continuous dynamics of CPSs [5]. Further, run-time errors (implicit oracles [1]) such as buffer overflows are not sufficient as many faults may not lead to run-time crashes. Hence, engineers often need to manually inspect the outputs of the models under test to decide if test cases pass or fail.

Existing approaches and tools for testing and verifying SL/SF models [14, 12] generate test inputs in terms of single values instead of signals. They are not applicable to testing models with continuous-time behavior as they often require models to be first converted into code (or an intermediate discrete model). When automated oracles are not available, test generation using existing tools and techniques is solely driven by some notion of structural coverage. Effectiveness of structural coverage criteria, however, has yet to be ascertained and empirically evaluated for Simulink model testing. Specifically, our recent empirical study showed that fault revealing ability of coverage-adequate test suites generated by the Simulink Design Verifier (SLDV) tool [12], the only testing toolbox of Simulink, is rather low [9, 8]. Our study, particularly showed that although the test suites generated by SLDV are able to cover the structure, including the faulty parts of SL/SF models, they fail to produce fault-revealing outputs, i.e., outputs that can be identified as failures based on a manual oracle [8].

To deal with the above-mentioned challenges, in our earlier work we proposed a number of test generation algorithms for SL/SF models. Our algorithms generate test cases in terms of signals, are applicable to both continuous-time and discrete-time behaviors, and generate small test suites with high fault revealing power, hence reducing the cost of manual test oracles [9, 8]. Specifically, we used meta-heuristic search algorithms [6] to generate test suites based on our notions of failure-based and output diversity [9, 8]. Our failure-based algorithm aims to maximize the likelihood of presence of two specific failure patterns, namely *instability* and *discontinuity*, in the continuous output signals of SL/SF models [9]. Our output diversity algorithm aims to maximize diversity in output signals and produce output signals with diverse shapes [9, 8]. In this paper, we present **SimCoTest**[1] (**Sim**ulink **Co**ntroller **Test**er) tool, that supports test suite generation for Simulink/Stateflow models based on

[1]https://sites.google.com/site/simcotesttool/

our failure-based and output diversity algorithms [9, 8]. In addition, SimCoTest includes a mechanism to prioritize test suites based on their estimated fault revealing ability. SimCoTest automatically extracts all the information required for test generation from the underlying models. It, further, provides a full-fledged user interface to help engineers perform several sanity checks on the models prior to test generation and execute the generated test cases directly from the tool. Using SimCoTest, we were able to find two real faults in Delphi models, which had not been previously found by manual testing based on domain expertise. Further, our experiments show that SimCoTest is effective and generates test suites with significantly higher fault revealing ability, compared to those generated based on structural coverage, e.g., by SLDV [8].

2. OUTPUT-BASED TEST GENERATION

As mentioned in Section 1, test generation algorithms in SimCoTest operate on model outputs in contrast to existing test generation algorithms that are white-box and guided by structural coverage. Further, we note that most existing Simulink testing tools [14], including SLDV [12] and Reactis [11], are not compatible, and hence not applicable, to Simulink models with continuous-time and floating-point computations as they require the models to be first converted into an intermediate representation with discrete-time and fixed-point operators. Our output-based approach, however, is black-box and applicable to models with continuous-time and floating-point computations.

Output signals provide a useful source of information for detecting faults in Simulink models as they not only show the values of model outputs at particular time instants, but also they show how these values change over time. In particular, SimCoTest implements two output-based algorithms that are discussed below: failure-based [9] and output diversity [9, 8]. Both of these algorithms implement a whole test suite generation approach [4] and rely on meta-heuristic search techniques [6].

Our failure-based algorithm aims to maximize objective functions capturing the degree of presence of continuous output failure patterns [9]. Inspired by discussions with control engineers, in our earlier work, we proposed and formalized two continuous output failure patterns, referred to as *instability* and *discontinuity* [9]. The instability pattern is characterized by quick and frequent oscillations of the controller output over a time interval (see Figure 1(a)), and the discontinuity pattern captures fast, short-duration and upward or downward pulses in the controller output (see Figure 1(b)). Presence of either of these patterns in SL/SF model outputs may have undesirable impact on physical processes or objects that are controlled by or interact with the model. Given an output signal o, we defined objective functions approximating the degree of instability and discontinuity in o, respectively [9]. The higher the value of these functions for a signal o, the more certain we can be that o exhibits some instability or discontinuity failure. Our failure-based algorithm iteratively searches the input space, and at each iteration, produces a whole test suite containing test inputs that yield outputs with high values of these functions [9]. Figures 1(a) and (b) show two output signals of test cases generated by SimCoTest for two faulty Simulink model examples. Figure 1(a) shows an instability failure (the area marked by red line), and Figure 1(b) shows a discontinuity failure at point A.

Failure-based test generation is useful for revealing faults that lead to specific and known failure patterns in model outputs. For other faults, SimCoTest implements an output diversity test generation algorithm [8]. Similar to above, this algorithm is a meta-heuristic search algorithm [6], and is guided to produce test outputs with diverse signal shapes. Our past experiences [9, 8] show

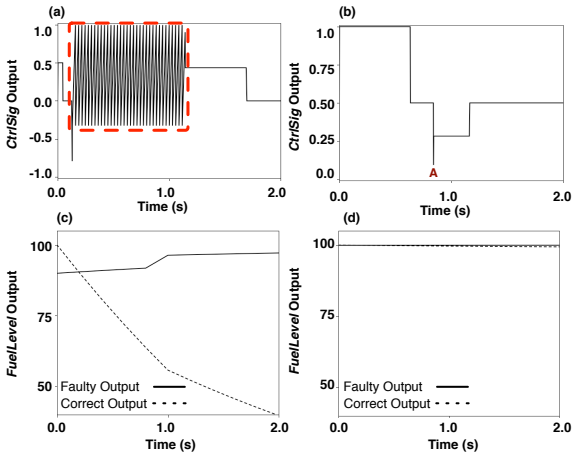

Figure 1: Examples of Simulink/Stateflow output signals.

that test cases that yield diverse output signals are likely to reveal different types of faults in Simulink models. The key in our work is the definition and characterization of output diversity. We provide two alternative definitions for output diversity: *vector-based* and *feature-based* [9, 8]. The former maximizes Euclidean distance over output signal vectors. The latter maximizes the distance between the *feature vectors* developed based on output signals. In our previous work, we defined a set of basic features characterizing distinguishable signal shapes. Each signal feature is characterized by a quantitative feature function [8]. Given any signal, the elements of a feature vector corresponding to that signal represent the values of feature functions applied to that signal. SimCoTest implements test generation algorithms related to both *vector-based* and *feature-based* notions of output diversity.

We illustrate the differences between test generation based on structural coverage and based on output diversity using signals shown in Figures 1(c) and (d). Figures 1(c) and (d) show two different output signals obtained by applying two test cases to a faulty Simulink model along with the expected outputs (oracles). The faulty outputs are shown by solid lines and the oracles are shown by dashed lines. We note that the test input yielding the output in Figure 1(d) covers exactly the same parts of the model as the test input yielding the output in Figure 1(c). That is, the two test inputs achieve exactly the same level of structural coverage. However, the faulty output in Figure 1(d) almost matches the correct output (the oracle), while the one in Figure 1(c) drastically deviates from the oracle. In this domain, small output deviations from oracle are typically considered to be due to discretization conversions (e.g., replacing continuous-time or floating-point operators with discrete-time or fixed-point operators), and, hence, are not considered as failures. Therefore, engineers are unlikely to identify any fault when provided with the output in Figure 1(d). When the goal is high structural coverage, the test inputs yielding the output in Figure 1(d) and the one in Figure 1(c) are equally desirable. However, by relying on the test input that generates the former output, the fault most likely goes unnoticed. In contrast, our output-based algorithms attempt to generate test cases that yield either outputs exhibiting some failure patterns or diverse output signals with the aim of increasing the probability of generating outputs that noticeably diverge from the expected result (e.g., the output in Figure 1(c)).

3. TOOL OVERVIEW

Figure 2 shows an overview of SimCoTest that consists of four steps: (1) checking SL/SF models to identify basic structural or syntactic errors (sanity checks), (2) automatically extracting the in-

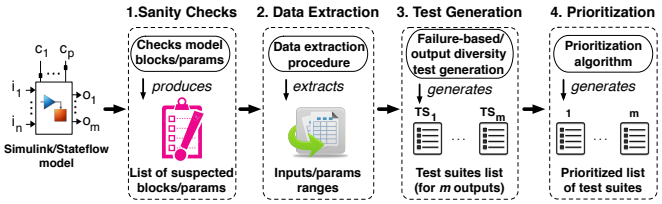

Figure 2: An overview of test suite generation for SL/SF models in SimCoTest.

Table 1: SimCoTest sanity checks.

SimCoTest Sanity Checks
Simulink Blocks
1. Simulink blocks for which *'Saturate on Integer Overflow'* is not enabled 2. From blocks without a corresponding Goto block 3. Input/output signals for which *'Resolve to Signal Object'* is not enabled
Configuration Parameters
4. Single parameters with multiple entries 5. Tabular parameters assigned to a single value 6. Tabular parameters with all entries assigned to the same value 7. A configuration parameter with a name pattern x_high has a value smaller than its corresponding parameter with a name pattern x_low.

formation required for test generation from SL/SF models (data extraction), (3) generating test suites using our failure-based and output diversity test generation algorithms (test suite generation), and (4) prioritizing the generated test suites for different model outputs according to their fault revealing ability (prioritization). Below, we discuss the four steps of the process in Figure 2.

3.1 Sanity Checks

Prior to test generation, SimCoTest performs a number of sanity checks on SL/SF models to identify basic errors that can be identified by evaluating Simulink models' structure and syntax. Note that Model Advisor, which is a Simulink toolbox, also performs a number of sanity checks on Simulink models, e.g., identifying disconnected model blocks [13]. However, our sanity checks implemented in SimCoTest were identified in the course of our discussions with control engineers, and are not covered by Simulink Model Advisor. Figure 1 shows a list of sanity checks implemented in SimCoTest. The sanity check number one helps engineers to bypass overflow/underflow runtime crashes. The check number two is focused on well-formedness of go-to and from blocks. The check number three ensures that model input and output signals resolve to a Simulink signal object, explicitly identifying their data types and data ranges. The checks number four to seven particularly focus on well-definedness of configuration parameters in Simulink models. Configuration parameters are common in automotive applications, and come in the form of single or tabular values. One common error is that the default values assigned to configuration parameters contradict their types (e.g., checks number four and five). In addition, SimCoTest checks if all elements of a tabular configuration parameter are assigned to the same value (check number six). Finally, the check number seven looks for cases where the max value of a configuration parameter is less than its min value.

3.2 Data Extraction

To be able to perform test generation, SimCoTest requires to get the simulation time, and further, obtain the names, data types and data ranges of the input and output variables, and configuration parameters of the model under test. SimCoTest uses Matlab APIs to get the model simulation time and iterates over the model blocks to identify the input and output ports, and the configuration parameters. It then automatically extracts data types and data ranges, as

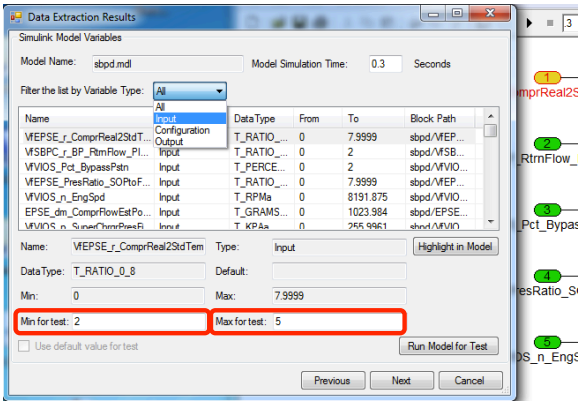

Figure 3: Data extraction results form in SimCoTest

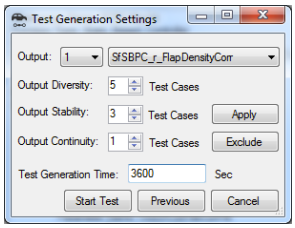

Figure 4: Test generation settings form.

well as default values for the configuration parameters. Figure 3 shows the *Data Extraction Results* form representing the model simulation time (0.3 sec) and a list of inputs, configuration parameters and outputs. SimCoTest also allows users to change value ranges for input variables prior to testing. In Figure 3 for example, the user has decided to use the range $[2, 5]$ for the first input of the model (highlighted by red boxes in the figure) instead of the extracted range $[0, 7.9999]$. For configuration parameters, the user may choose to treat them as input variables and let our test generation algorithms compute their values, or she may decide to fix their values to their default values.

3.3 Test Generation

As discussed earlier, we assume that test oracles are manual and are determined by engineers. Hence, the test suite size cannot be arbitrarily large, and has to be determined by users based on their manual test oracle budget. SimCoTest receives the test suite size prior to testing. Since our test generation algorithms generate one test suite per each model output, users determine the size for each of these test suites. Figure 4 shows the *Test Generation Settings* form where the user can specify the size of the test suites for each algorithm (i.e., failure-based or output diversity) and per each model output. Users have the option of excluding less important outputs, i.e., outputs that are unlikely to reveal any fault as their values are not related to the main system function. Finally, users may determine the test generation time and start the test generation process. SimCoTest proceeds with the test suite generation step for all the algorithms and for all the model outputs, concurrently. Users may stop the test generation process at any time and before the end of the allotted time, and access the generated test suites up to that point.

3.4 Prioritization

After test generation, users need to go through the generated test suites for different outputs and inspect the output signals to decide if test cases pass or fail. Note that for a test suite ts generated for an output o, users need to evaluate only the values of o and not the values generated by ts for outputs other than o. To help users effec-

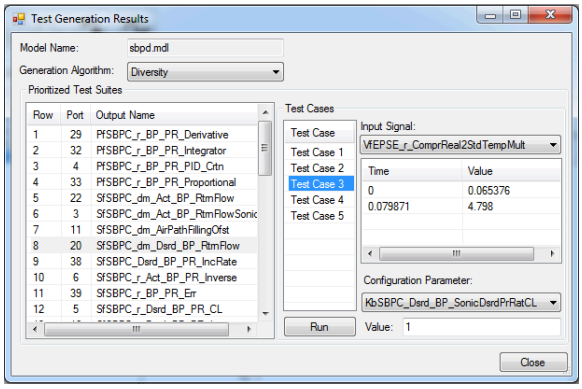

Figure 5: Test generation results form.

tively inspect test outputs, SimCoTest prioritizes the generated test suites based on their likelihood of revealing faults. This helps users prioritize their manual test oracle budget and focus on test suites and model outputs that are more likely to reveal faults. Our prioritization algorithm assigns a priority to each test suite based on the stability/continuity/diversity of the test cases within the test suite as well as their accumulative structural coverage. Figure 5 shows the *Test Generation Results* form where the outputs are sorted in the left-hand side list (i.e., the *Prioritized Test Suites* list), based on their fault revealing ability as estimated by our prioritization algorithm. Users can click on each output to obtain the corresponding test suite in the right side of the form (i.e., the *Test Cases* list). Further, for each test case within a test suite, users can see the information about signals assigned to input variables and values assigned to configuration parameters. Finally, SimCoTest enables users to run each test case in Simulink and generate the simulation results. In particular, SimCoTest autonomically plots output signals for each corresponding output variable.

4. EVALUATION

We have evaluated the practical utility of SimCoTest by applying it to three automotive industrial SL/SF models from Delphi Automotive [9, 8]. These models are representative in terms of the size and complexity of models developed at Delphi. Specifically, they contain a total number of 1469 Simulink blocks and include 53 input variables, 126 configuration parameters, and 68 output variables. Using the test suites generated by SimCoTest, we were able to identify two faults in these models which had not been previously found by manual testing: (1) We identified a model output whose value remained constant, despite its expected behaviour, for test cases generated by our output diversity algorithm, and (2) SimCoTest was able to help identify a fault in a delay buffer. The test case that revealed this fault was generated by our output diversity test generation algorithm, and the fault was identified through manual inspection of the output for this test case. Note that none of these two faults could be revealed using implicit test oracles since they do not lead to runtime crashes. Further, engineers did not know about these faults a priori and did not have formal assertions capturing them. Finally, the warnings generated by Simulink did not help reveal these faults.

We empirically evaluated the fault revealing ability of SimCoTest by applying it to 149 faulty versions of the three industrial SL/SF models from Delphi as well as three publicly available SL/SF models [9, 8]. To do so, we first developed a comprehensive list of Simulink fault patterns [10] through our discussions with senior Delphi engineers and by reviewing the existing literature on mutation operators for Simulink models (e.g., [2]). Based on this list, we then implemented an automated fault seeding program to generate the faulty versions of the models [9, 8]. In our experiments, we compared the fault revealing ability of the test suites generated by SimCoTest, based on our failure-based and output diversity test generation algorithms, with randomly generated and coverage-adequate test suites. Our experiment results showed that (1) our output-based test generation algorithms outperform random and coverage-based test suite generation, (2) in the presence of failure patterns in the faulty model outputs, failure-based algorithms outperform the others, particularly with small test suite sizes, and (3) in the absence of failure patterns in the faulty model outputs, the output diversity performs the best [9], particularly with relatively larger test suite sizes. Further, we showed that our test suites generated based on output diversity yields significantly higher fault revealing rates compared to test suites generated by SLDV [8].

5. CONCLUSION

We presented a tool, SimCoTest, to generate small test suites with high fault revealing ability for Simulink/Stateflow controller models. SimCoTest generates test suites for SL/SF models using our failure-based and output diversity test generation algorithms and prioritizes them based on their estimated fault revealing ability. Using SimCoTest, we were able to generate test cases that identified two faults in representative SL/SF models from Delphi Automotive. These faults had not been previously found by manual testing based on domain expertise.

In our earlier work, we developed a tool called CoCoTest [7] to automatically test closed-loop continuous controllers (i.e., PIDs) in Simulink models. The underlying algorithms of CoCoTest were different from those used in SimCoTest. Further, CoCoTest required automated oracles and was not applicable to Simulink/Stateflow models in their entirety.

SimCoTest is implemented as part of an industry-driven research effort to devise cost-effective and scalable techniques to testing automotive software components. We are currently taking the necessary steps to integrate SimCoTest into Delphi's development process, so that it can be used by Delphi engineers on a daily basis.

6. REFERENCES

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *TSE*, 41(5):507–525, 2015.

[2] A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rümmer, and G. Weissenbacher. Mutation-based test case generation for Simulink models. In *FMCO 2010*, pages 208–227. Springer, 2010.

[3] D. K. Chaturvedi. *Modeling and simulation of systems using MATLAB and Simulink*. CRC Press, 2009.

[4] G. Fraser and A. Arcuri. Whole test suite generation. *TSE*, 39(2):276–291, 2013.

[5] E. A. Lee and S. A. Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Lee & Seshia, 2011.

[6] S. Luke. *Essentials of metaheuristics*, volume 113. Lulu Raleigh, 2009.

[7] R. Matinnejad, S. Nejati, L. Briand, and T. Bruckmann. Cocotest: a tool for model-in-the-loop testing of continuous controllers. In *ASE2014*, pages 855–858. ACM, 2014.

[8] R. Matinnejad, S. Nejati, L. Briand, and T. Bruckmann. Automated Test Suite Generation for Time-Continuous Simulink Models. Technical report, UL, 2015.

[9] R. Matinnejad, S. Nejati, L. Briand, and T. Bruckmann. Effective test suites for mixed discrete-continuous stateflow controllers. In *ESEC/FSE 2015*, 2015.

[10] Matinnejad, Reza. The paper extra resources (technical reports and the models). `https://sites.google.com/site/myicseresources/`.

[11] Reactive Systems Inc. Reactis Validator. `http://www.reactive-systems.com/simulink-testing-validation.html`, 2010.

[12] The MathWorks Inc. Simulink Design Verifier. `http://nl.mathworks.com/products/sldesignverifier/?refresh=true`.

[13] The MathWorks Inc. Simulink. `http://www.mathworks.nl/products/simulink`, 2003.

[14] J. Zander, I. Schieferdecker, and P. J. Mosterman. *Model-based testing for embedded systems*, volume 13. CRC Press, 2012.