securityandtrust.lu

Software Verification and Validation Laboratory: Automated Test Suite Generation for Time-Continuous Simulink Models

Reza Matinnejad, Shiva Nejati, and Lionel Briand
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
Thomas Bruckmann
Delphi Automotive Systems, Luxembourg

TR-SnT-2015-7
ISBN: 978-2-87971-145-4

September 15, 2015
Version 1.0

UNIVERSITÉ DU LUXEMBOURG

Automated Test Suite Generation for Time-continuous Simulink Models

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

All engineering disciplines are founded and rely on models, although they may differ on purposes and usages of modeling. Interdisciplinary domains such as Cyber Physical Systems (CPSs) seek approaches that incorporate different modeling needs and usages. Specifically, the Simulink modeling platform greatly appeals to CPS engineers due to its seamless support for simulation and code generation. In this paper, we propose a test generation approach that is applicable to Simulink models built for both purposes of simulation and code generation. We define test inputs and outputs as signals that capture evolution of values over time. Our test generation approach is implemented as a meta-heuristic search algorithm and is guided to produce test outputs with diverse shapes according to our proposed notion of diversity. Our evaluation, performed on industrial and public domain models, demonstrates that: (1) In contrast to the existing tools for testing Simulink models that are only applicable to a subset of code generation models, our approach is applicable to both code generation and simulation Simulink models. (2) Our new notion of diversity for output signals outperforms random baseline testing and an existing notion of signal diversity in revealing faults in Simulink models. (3) The fault revealing ability of our test generation approach outperforms that of the Simulink Design Verifier, the only testing toolbox for Simulink.

1. INTRODUCTION

Modeling has a long tradition in software engineering. Software models are particularly used to create abstract descriptions of software systems from which concrete implementations are produced [17]. Software development using models, also referred to as Model Driven Engineering (MDE), is largely focused around the idea of *models for code generation* [16] or *models for test generation* [38]. Code or test generation, although important, is not the primary reason for software modeling when software development occurs in tandem with control engineering. In domains where software closely interacts with physical processes and objects such as Cyber Physical Systems (CPSs), one main driving force of modeling is *simulation*, i.e., design time testing of system models. Simulation aims to identify defects by testing models in early stages and before the system has been implemented and deployed.

In the CPS domain, models built for simulation have major differences from those from which code can be generated. Simulation models are heterogeneous, encompassing software, network and physical parts, and are meant to represent as accurately as possible the real world and its continuous dynamics. These models have *time-continuous* behaviors (described using differential equations) since they are expected to capture and continuously interact with the physical world [26, 22]. Code generation models, on the other hand, capture software parts only, and have *discrete time* behavior (described using some form of discrete logic or discrete state machines) [46, 25]. This is because the generated code will run on platforms that support discrete computations, and further, the code will receive input data as discrete sequences of events.

When simulation models are available, testing starts very early typically by running those models for a number of selected scenarios. Early testing of simulation models pursues, among others, two main objectives: (1) Ensuring correctness of simulation models so that these models can act as oracle. This can significantly reduce engineers' reliance on expensive and late testing and measurements on the final hardware. (2) Obtaining an initial test suite with high fault revealing power to be used for testing software code, or at later stages for testing the system on its final hardware platform.

Our goal is to provide automated techniques to generate effective test suites for Simulink models [48]. Simulink is an advanced platform for developing both simulation and code generation models in the CPS domain. The existing approaches to testing and verifying Simulink models almost entirely focus on models with time-discrete behavior, i.e., code generation models. These approaches generate discrete test inputs for Simulink models with the goal of reaching runtime errors to reveal faults [50, 23], violating assertions inserted into Simulink models based on some formal specification [41, 14], and achieving high structural coverage [36, 42]. Discrete test inputs, however, are seldom sufficient for testing Simulink models, in particular, for those models with time-continuous behaviors. Many faults may not lead to runtime crashes. Formal specifications are rather rare in practice, and further, are not amenable to capturing continuous dynamics of CPSs. Finally, effectiveness of structural coverage criteria has yet to be ascertained and empirically evaluated for Simulink model testing.

In this paper, we provide test generation techniques for Simulink models with time-continuous behaviors. We generate test inputs as *continuous signals* and do not rely on the existence of implicit oracles (e.g., runtime failures) or formal specifications. Assuming that automated oracles are not available, we focus on providing test generation algorithms that develop small test suites with high fault revealing power, effectively reducing the cost of human test oracles [6, 32]. Instead of focusing on increasing structural coverage, we propose and evaluate a test data generation approach that aims

to maximize diversity in output signals of Simulink models. 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. By inspecting output signals, one can determine whether the model output reaches appropriate values at the right times, whether the time period that the model takes to change its values is within acceptable limits, and whether the signal shape is free of erratic and unexpected changes that violate continuous dynamics of physical processes or objects.

Our intuition is that test cases that yield diverse output signals may likely reveal different types of faults in Simulink models. The key here is the definition of output diversity. In our earlier work, we proposed a notion of output diversity based on the Euclidean distance between signal vector outputs of mixed discrete-continuous Stateflow models [29]. Stateflow is a subset of Simulink for capturing state-based behaviors. In this work, we introduce a new notion of diversity for signal outputs that is defined based on a set of representative and discriminating signal feature shapes. We refer to the former as *vector-based* and to the latter as *feature-based* diversity objectives. We develop a meta-heuristic search algorithm that generates test suites with diversified output signals where the diversity objective can be either vector-based or feature-based. Our algorithm uses the whole test suite generation approach [19, 18]. Further, our algorithm adapts a single-state search optimizer [28] to generate continuous test input signals, and proposes a novel way to dynamically increase variations in test input signals based on the amount of structural coverage achieved by the generated test suites. We evaluate our algorithm using four industrial and public-domain Simulink models. Our contributions are as follows:

(1) We identify the problem of testing simulation models and argue that, though simulation models are essential in the CPS domain, few systematic testing/verification techniques exist for them.

(2) We propose a new notion of diversity for output signals and develop a novel algorithm based on this notion for generating test suites for both simulation and code generation models in Simulink. We show that our new notion of diversity for output signals outperforms random baseline testing and an existing notion of signal output diversity in revealing faults in Simulink models.

(3) We compare our test generation approach that diversifies test outputs with the Simulink Design Verifier (SLDV), the only testing toolbox of Simulink. SLDV automatically generates test suites for a subset of Simulink models with the goal of achieving high structural coverage. We argue that while our approach is applicable to the entire Simulink, SLDV supports a subset. We show that, when considering the SLDV-compatible subset, our output diversity approach is able to reveal significantly more faults compared to SLDV, and further, it subsumes SLDV in revealing faults: Any fault identified by SLDV is also identified by our approach.

2. MOTIVATION AND BACKGROUND

In this section, we provide examples of simulation and code generation models. We then introduce SimuLink Design Verifier (SLDV) and motivate our output diversity test generation approach.

Example. We motivate our work using a simplified Fuel Level Controller (FLC) which is an automotive software component used in cars' fuel level management systems. FLC computes the fuel volume in a tank using the *continuous* resistance signal that it receives from a fuel level sensor mounted on the fuel tank. The sensor data, however, cannot be easily converted into an accurate estimation of the available fuel volume in a tank. This is because the relationship between the sensor data and the actual fuel volume is impacted by the irregular shape of the fuel tank, dynamic conditions of the vehicle (e.g., accelerations and braking), and the oscillations of the indication provided by the sensors. Hence, FLC has to rely on complex filtering algorithms involving algebraic and differential equations to accurately compute the actual fuel volume [47].

Simulation models. In the automotive industry, engineers build simulation models prior to any software coding and often at the time of system-level design and engineering. Simulation models most often contain time-continuous mathematical operations [12]. For example, Figure 1(a) shows a very simplified FLC Simulink model which is adopted from [59] and includes a time-continuous integral operator ($\int$). We refer to the model in Figure 1(a) as a simulation model of FLC. The input of this model is the resistance signal from the sensor, and its output shows the fuel volume. The Simulink model in Figure 1(a) is executable. Engineers can run the model for any desired input signal and inspect the output. Automotive engineers often rely on their knowledge of mechanics and control theory to design simulation models. These models, however, need to be verified or systematically tested as they are complex and may include several hundreds of blocks.

Code generation models. Figure 1(b) shows an example FLC code generation model, (i.e., the model from which software code can be automatically generated). The code generation model is discrete: The time-continuous integrator block ($\int$) in the simulation model is replaced by a time-discrete integrator (`sum`) in the code generation model. The behavior of code generation models may deviate from that of simulation models since the latter often has time-continuous operations while the former is purely discrete. Typically, some degree of deviations between simulation and code generation model outputs are acceptable. The level of acceptable deviations, however, have to be determined by domain experts.

Simulation model outputs vs. code generation model outputs. Figure 1(c) shows an example continuous input signal for FLC over a 10 sec time period. The signal represents the resistance values received from the tank sensor. This input signal can be applied to both simulation and code generation models of FLC. Note that a time-continuous model has to be provided with a continuous input signal (i.e., a continuous function over time). A time-discrete model, on the other hand, only requires single values at discrete time steps which can be extracted from continuous signals as well. Models in Figures 1(a) and (b) respectively produce the outputs in Figures 1(d) and (e) once they are provided with the input in Figure 1(c). As shown in Figures 1(d) and (e), the percentages of fuel volume in the continuous output signal differ from those in the discrete output signal. For example, after one second, the output of the simulation model is 91.43, while that of the code generation model is 88.8. These deviations are due to the differences between the continuous and discrete integrals. For example, the grey area in the left part of Figure 1(f) shows the value computed by $\int$ after three seconds, while the value computed by the discretized integral operator after three seconds is the grey area in the right part of Figure 1(f). Clearly, these two operators, and hence the two models in Figures 1(a) and (b), generate different values for the same input.

Simulation models as oracles for code. In the CPS domain, time-continuous simulation models play a crucial role. Not only that they are the blueprints for code generation models, and later, the software code, but also they serve as oracle generators for testing software code on various platforms. While oracles obtained based on formal specifications or runtime errors are often precise and deterministic [34, 35], those obtained based on simulation outputs are *inexact* as some deviations from oracles are acceptable. These oracles provide engineers with a reasonable and yet valuable assessment of the code behavior. Further, the oracle information

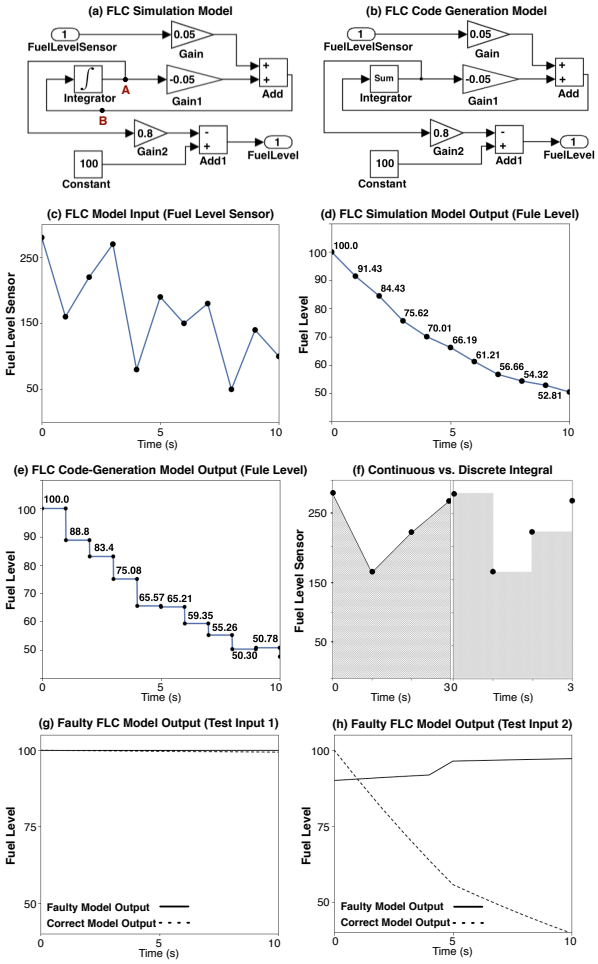

Figure 1: A Fuel Level Controller (FLC) example: (a) A simulation model of FLC; (b) a code generation model of FLC; (c) an input to FLC; (d) output of (a) when given (c) as input; (e) output of (b) when given (c) as input; (f) comparing outputs of the blocks $\int$ and `sum` from models (a) and (b), respectively; (g) A test output of a faulty version of (a); and (h) another test output a faulty version of (a).

that is obtained from simulation models is not readily available in other artifacts, e.g., requirements and specifications. Therefore, it is important to develop accurate and high quality simulation models. Hence, our goal in this paper is to provide systematic algorithms to help with testing of time-continuous simulation models.

Simulink Design Verifier. SimuLink Design Verifier (SLDV) is a product of Mathworks and a Simulink toolbox. It is the only Simulink toolbox that is dedicated to test generation. It automatically generates test input signals for Simulink models using constraint solving and model-checking techniques [50]. SLDV provides two usage modes: (1) generating test suites to achieve some form of structural coverage, and (2) generating test scenarios (counter-examples) indicating violation of some given properties (assertions). In the first usage mode, SLDV creates a test suite satisfying a given structural coverage criterion [51]. In the second usage mode, SLDV tries to prove that assertions inserted into Simulink models cannot be reached, or otherwise, it generates inputs triggering the assertions, hence, disproving the desired properties.

In this paper, we compare the fault revealing ability of our algorithm with that of the first usage mode of SLDV, i.e., test generation guided by structural coverage. Note that the second usage mode of SLDV requires exact oracles which is out of the scope of this paper. We chose to compare our work with SLDV as it is distributed by the Mathworks and is among the most well-known tools for testing and verification of Simulink models. Other existing tools in that category, e.g., Reactis [40], rely on formal techniques as well. Based on our experience working with SLDV and according to the Mathworks white papers [21], SLDV has the following practical limitations: (1) *Model Compatibility.* SLDV supports a subset of the Simulink language (i.e., discrete fragment of Simulink) [12], and is not applicable to time-continuous blocks of Simulink such as the continuous integrator in Figure 1(a). Specifically, SLDV is applicable to the model in Figure 1(b), but not to that in Figure 1(a). Further, there are a number of blocks that are acceptable by the Simulink code generator but are not yet compatible with SLDV in particular, the *S-Function* block which provides access to system functions and custom C code from Simulink models. (2) *Scalability.* The lack of scalability of SLDV is recognized as an issue by Mathworks [21]. Further, as the models get larger and more complicated, it is more likely that they contain blocks that are not supported by SLDV. So, for SLDV, the problem of compatibility often precedes and overrides the problem of lack of scale [21].

Coverage vs. Output Diversity. In contrast to SLDV which aims to maximize structural coverage, we propose a test generation algorithm that tries to diversify output signals. We illustrate the differences between test generation based on structural coverage and based on output diversity using a faulty version of the simulation model in Figure 1(a). Suppose the line connected to point A in this model is mistakenly connected to point B. Figures 1(g) and (h) show two different output signals obtained from this faulty model along with the expected outputs. The faulty output is shown by a solid line and the correct one (oracle) is shown by a dashed line. The faulty output in Figure 1(g) almost matches the oracle, while the one in Figure 1(h) drastically deviates from the oracle. Given that small deviations from oracle are acceptable, engineers are unlikely to identify any fault when provided with the output in Figure 1(g). When the goal is high structural coverage, the test inputs yielding the two outputs in Figures 1(g) and (h) are equally desirable. Indeed, for the FLC model, one test input is sufficient to achieve full structural coverage. If this test input happens to produce the output in Figure 1(g), the fault goes unnoticed. In contrast, our approach attempts to generate test cases that yield diverse output signals to increase the probability of generating outputs that noticeably diverge from the expected result. In Section 3, we introduce our feature-based notion of signal output diversity and our test generation algorithm.

3. TEST GENERATION ALGORITHMS

We propose a search-based whole test suite generation algorithm for Simulink models. Our test generation algorithm aims to maximize diversity among output signals generated by a test suite. We define two notions of diversity among output signals: *vector-based* and *feature-based*. We first fix a notation, and will then describe our notions of output diversity and our test generation algorithm.

Notation. Let $M = (\mathcal{I}, o)$ be a Simulink model where $\mathcal{I} = \{i_1, \ldots, i_n\}$ is the set of input variables and o is the output variable of M. Each input/output variable of M is a signal, i.e., a function of time. Irrespective of M being a simulation or a code generation model, each input or output of M is stored as a vector whose elements are indexed by time. Assuming that the simulation time is T, the simulation interval $[0..T]$ is divided into small

equal time steps denoted by Δt. We define a signal sg as a function $sg : \{0, \Delta t, 2 \cdot \Delta t, \ldots, k \cdot \Delta t\} \rightarrow \mathcal{R}$, where Δt is the simulation time step, k is the number of observed simulation steps, and $\mathcal{R}$ is the signal range. The signal range $\mathcal{R}$ is bounded by its min and max values denoted $min_{\mathcal{R}}$ and $max_{\mathcal{R}}$, respectively. For the example in Figure 1, we have $T = 10s$, $\Delta t = 1s$, and $k = 10$. Note that in that example, to better illustrate the input and output signals, Δt is chosen to be larger than normal. In one of our experiments, for example, we have $\Delta t = 0.001s$, $T = 2s$, and $k = 2000$.

Our goal is to generate a test suite $TS = \{I_1, \ldots, I_q\}$. Each test input I_j is a vector $(sg_{i_1}, \ldots, sg_{i_n})$ of signals for the input variables i_1 to i_n of M. By simulating M using each test input I_j, we obtain an output signal sg_o for the output variable o of M. All the input signals sg_{i_j} and the output signal sg_o share the same simulation time interval and simulation time steps, i.e., the values of Δt, T, and k are the same for all of the signals.

Test inputs. In Simulink, every variable even those representing discrete events are described using signals. In this context, test input generation is essentially signal generation. Each test input is a vector $(sg_{i_1}, \ldots, sg_{i_n})$ of signals. Each signal sg_{i_j} is also a vector with a few thousands of elements, and each element can take an arbitrary value from the signal range. To specify an input signal, however, engineers never define a few thousands of values individually. Instead, they specify a signal by defining a sequence of *signal segments*. To formalize input signals generated in practice, we characterize each input signal sg with a set $\{(k_1, v_1), \ldots, (k_p, v_p)\}$ of points where k_1 to k_p are the simulation steps s.t. $k_1 = 0$, $k_1 \leq k_2 \leq \ldots \leq k_p$, and v_1 to v_p are the values that sg takes at simulation steps k_1 to k_p, respectively. The set $\{(k_1, v_1), \ldots, (k_p, v_p)\}$ specifies a signal sg with p segments. The first $p-1$ segments of sg are defined as follows: For $1 \leq j < p$, each pair (k_j, v_j) and (k_{j+1}, v_{j+1}) specifies a signal segment s.t. $\forall l \cdot k_j \leq l < k_{j+1} \Rightarrow sg(l \cdot \Delta t) = v_j \wedge sg(k_{j+1} \cdot \Delta t) = v_{j+1}$. The last segment of sg is a constant signal that starts at (k_p, v_p) and ends at (k, v_p) where k is the maximum number of simulation steps. For example, $\{(0, v)\}$ specifies a constant signal at v (i.e., one segment $p = 1$). A step signal going from v_0 to v_1 and stepped at k' is specified by $\{(0, v_0), (k', v_1)\}$ (i.e., two segments $p = 2$).

Input signals with fewer segments are easier to generate but they may fail to cover a large part of the underlying Simulink model. By increasing the number of segments in input signals, structural coverage increases, but the output generated by such test inputs becomes more complex, and engineers may not be able to determine expected outputs (oracle). Furthermore, highly segmented input signals may not be reproducible on hardware platforms as they may violate physical constraints of embedded devices. For each input variable, engineers often have a good knowledge on the maximum number of segments that a signal value for that variable may possibly contain and still remains feasible. In our test generation algorithm discussed at the end of this section, we ensure that, for each input variable, the generated input signals achieve high structural coverage while their segment numbers remain lower than the limits provided by domain experts.

Vector-based output diversity. This diversity notion is defined directly over output signal vectors. Let sg_o and sg'_o be two signals generated for output variable o by two different test inputs of M. In our earlier work [29], we defined the *vector-based* diversity measure between sg_o and sg'_o as the normalized Euclidean distance between these two signals. We denote the vector-based diversity between sg_o and sg'_o by $\hat{dist}(sg_o, sg'_o)$.

Our vector-based notion, however, has two drawbacks: (1) It is computationally expensive since it is defined over signal vectors with a few thousands of elements. Using it in a search algorithm amounts to computing the Euclidean distance between many pairs of output signals at every iteration of the search. (2) A search driven by vector-based distance may generate several signals with similar shapes whose vectors happen to yield a high Euclidean distance value. For example, for two constant signals sg_o and sg'_o, $\hat{dist}(sg_o, sg'_o)$ is relatively large when sg_o is constant at the maximum of the signal range while sg'_o is constant at the minimum of the signal range. A test suite that generates several output signals with similar shapes may not help with fault finding.

Feature-based output diversity. In machine learning, a feature is an individual measurable and non-redundant property of a phenomenon being observed [57]. Features serve as a proxy for large input data that is too expensive to be directly processed, and further, is suspected to be highly redundant. In our work, we define a set of basic features characterizing distinguishable signal shapes. We then describe output signals in terms of our proposed signal features, effectively replacing signal vectors by *feature vectors*. Feature vectors are expected to contain relevant information from signals so that the desired analysis can be performed on them instead of the original signal vectors. To generate a diversified set of output signals, instead of processing the actual signal vectors with thousands of elements, we maximize the distance between their corresponding feature vectors with tens of elements.

Figure 2(a) shows our proposed signal feature classification. Our classification captures the typical, basic and common signal patterns described in the signal processing literature, e.g., constant, decrease, increase, local optimum, and step [37]. The classification in Figure 2(a) identifies three abstract signal features: value, derivative and second derivative. The abstract features are italicized. The value feature is extended into: instant-value and constant-value features that are respectively parameterized by (v) and (n, v). The former indicates signals that cross a specific value v at some point, and the latter indicates signals that remain constant at v for n consecutive time steps. These features can be instantiated by assigning concrete values to n or v. Specifically, the constant-value (n, v) feature can be instantiated as the one-step constant-value(v) and always constant-value(v) features by assigning n to one and k (i.e., the simulation length), respectively. Similarly, specific values for v are zero, and max and min of signal ranges (i.e., $max_{\mathcal{R}}$ and $min_{\mathcal{R}}$).

The derivative feature is extended into sign-derivative and extreme-derivative features. The sign-derivative feature is parameterized by (s, n) where s is the sign of the signal derivative and n is the number of consecutive time steps during which the sign of the signal derivative is s. The sign s can be zero, positive or negative, resulting in constant(n), increasing(n), and decreasing(n) features, respectively. As before, specific values of n are one and k. The extreme-derivatives feature is non parameterized and is extended into one-sided discontinuity, one-sided discontinuity with local optimum, one-sided discontinuity with strict local optimum, discontinuity, and discontinuity with strict local optimum features. The second derivative feature is extended into more specific features similar to the derivative feature. Due to space limit, we have not shown those extensions in Figure 2(a).

Figures 2(b) to (e) respectively illustrate the instant-value(v), the increasing(n), the one-sided continuity with local optimum, and the discontinuity with strict local optimum features. Specifically, the signal in Figure 2(b) takes value v at point A. The signal in Figure 2(c) is increasing for n steps from B to C. The signal in Figure 2(d) is right-continuous but discontinuous from left at point D, and further, the signal value at D is not less than the values at its adjacent point, hence making D a local optimum. Finally, the signal in Figure 2(e) is discontinuous from both left and right at

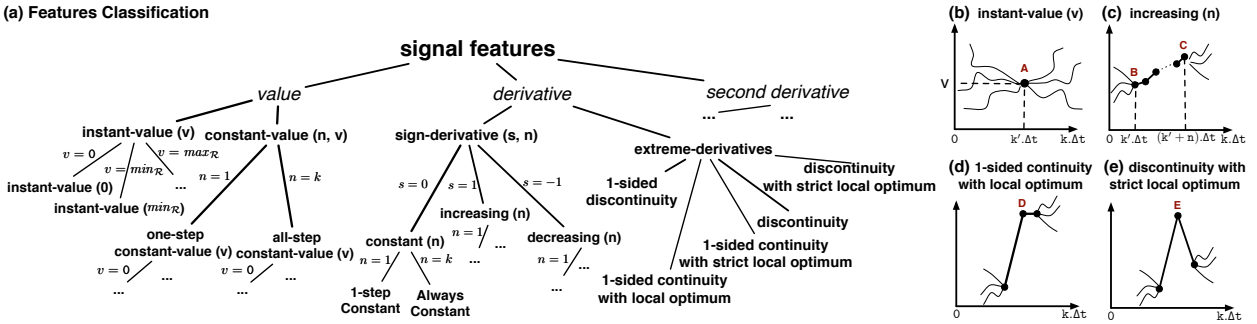

Figure 2: Signal Features: (a) Our signal feature classification, and (b)–(e) Examples of signal features from the classification in (a).

point E which is a strict local optimum point as well.

We define a function F_f for each (non-abstract) feature f in Figure 2(a). We refer to F_f as *feature function*. The output of function F_f when given signal sg as input is a value that quantifies the similarity between shapes of sg and f. More specifically, F_f determines whether any part of sg is similar to feature f. For example, suppose functions $\texttt{lds}(sg, i)$ and $\texttt{rds}(sg, i)$ respectively compute the left and right derivative signs of sg at simulation step i. Specifically, they generate 1, -1, and 0 if the derivative value is positive, negative, and zero, respectively. We define F_f for the feature in Figure 2(d) as follows:

$$F_f(sg) = \max_{i=1}^{k}(|\texttt{derivative}(sg, i)| \times \texttt{localOpt}(sg, i)) \quad \text{such that}$$

$$\texttt{derivative}(sg, i) = \frac{sg((i)\cdot\Delta t) - sg((i-1)\cdot\Delta t)}{\Delta t} \quad \text{and}$$

$$\texttt{localOpt}(sg, i) = \begin{cases} 1, & \texttt{lds}(sg, i) \neq \texttt{rds}(sg, i) \\ 0, & Otherwise \end{cases}$$

Function $F_f(sg)$ computes the largest left or right derivative of sg that occurs at a local optimum point, i.e., the largest one-sided derivative that occurs at a point i such that the derivative of sg changes its sign at i. The higher $F_f(sg)$, the more similar sg is to the feature in Figure 2(d). Our complete signal feature classification and the corresponding feature functions are available at [30].

Having defined features and feature functions, we now describe how we employ these functions to provide a measure of diversity between output signals sg_o and sg'_o. Let $f_1, \ldots, f_m$ be m features that we choose to include in our diversity measure. We compute feature vectors $F^v(sg_o) = (F_{f_1}(sg_o), \ldots, F_{f_m}(sg_o))$ and $F^v(sg'_o) = (F_{f_1}(sg'_o), \ldots, F_{f_m}(sg'_o))$ corresponding to signals sg_o and sg'_o, respectively. Since the ranges of the feature function values may vary widely, we standardize these vectors before comparing them. Specifically, we use feature scaling which is a common standardization method for data processing [57]. Having obtained standardized feature vectors $\hat{F}^v(sg_o)$ and $\hat{F}^v(sg'_o)$ corresponding to signals sg_o and sg'_o, we compute the Euclidean distance between these two vectors, (i.e., $\hat{dist}(\hat{F}^v(sg_o), \hat{F}^v(sg'_o))$), as the measure of feature-based diversity between signals sg_o and sg'_o. Below, we discuss how our diversity notions are used to generate test suites for Simulink models.

Whole test suite generation based on output diversity. We propose a meta-heuristic search algorithm to generate a test suite $TS = \{I_1, \ldots, I_q\}$ for a given model $M = (\mathcal{I}, o)$ to diversify the set of output signals generated by TS. We denote by $TSO = \{sg_1, \ldots, sg_q\}$ the set of output signals generated by TS. We capture the degree of diversity among output signals in TSO using objective functions O_v and O_f that correspond to vector-based and feature-based notions of diversity, respectively:

$$O_v(TSO) = \sum_{i=1}^{q} \text{MIN}_{\forall sg \in TSO \setminus \{sg_i\}} \hat{dist}(sg_i, sg)$$

$$O_f(TSO) = \sum_{i=1}^{q} \text{MIN}_{\forall sg \in TSO \setminus \{sg_i\}} \hat{dist}(F^v(sg_i), F^v(sg))$$

Function O_v computes the sum of the minimum distances of each output signal vector sg_i from the other output signal vectors in TSO. Similarly, O_f computes the sum of the minimum distances of each feature vector $F^v(sg_i)$ from feature vectors of the other output signals in TSO. Our test generation algorithm aims to maximize functions O_v and O_f to increase diversity among the signal vectors and feature vectors of the output signals, respectively.

Our algorithm adapts the whole test suite generation approach [19] by generating an entire test suite at each iteration and evolving, at each iteration, every test input in the test suite. The whole test suite generation approach is a recent and preferred technique for test data generation specially when, similar to O_v and O_f, objective functions are defined over the entire test suite and aggregate all testing goals. Another benefit of this approach for our work is that it allows us to optimize our test objectives while fixing the test suite size at a small value due to the cost of manual test oracles.

Our algorithm implements a single-state search optimizer that only keeps one candidate solution (i.e, one test suite) at a time, as opposed to population-based algorithms that keep a set of candidates at each iteration. This is because our objective functions are computationally expensive as they require to simulate the underlying Simulink model and compute distance functions between every test input pair. When objective functions are time-consuming, population-based search may become less scalable as it may have to re-compute objective functions for several new or modified members of the population at each iteration.

Figure 3 shows our output diversity test generation algorithm for Simulink models. We refer to it as OD. The core of OD is a Hill-Climbing search procedure [28]. Specifically, the algorithm generates an initial solution (lines 2-3), iteratively tweaks this solution (line 12), and selects a new solution whenever its objective function is higher than the current best solution (lines 16-18). The objective function O in OD is applied to the output signals in TSO that are obtained from test suites. The objective function can be either O_f or O_v, respectively generating test suites that are optimized based on feature-based and vector-based diversity notions.

While being a Hill-Climbing search in essence, OD proposes two novel adaptations: (1) It initially generates input signals that contain a small number of signal segments P. It then increases P only when it is needed while ensuring that P is never more than the limit provided by the domain expert P_{max}. Recall that, on one hand, increasing segments of input signals makes the output more difficult to analyse, but that, on the other hand, input signals with few segments may not reach high model coverage. In OD, we initially generate test inputs with P segments (lines 1-2). The tweak operator

Algorithm. The test generation algorithm applied to a Simulink model M.

1. $P \leftarrow$ initial number of signal segments for test inputs
2. $TS \leftarrow$ GENERATEINITIALTESTSUITE(P)
3. $BestFound \leftarrow O(TSO)$
4. $P_{max} \leftarrow$ maximum number of signal segments permitted in test inputs
5. $TSO \leftarrow$ signal outputs obtained by simulating M for every test input in TS
6. *whole-test-suite-coverage* $\leftarrow$ coverage achieved by TS over M
7. *initial-coverage* $\leftarrow$ *whole-test-suite-coverage*
8. *accumulative-coverage* $\leftarrow$ *initial-coverage*
9. Let σ-$exploration$ and σ-$exploitation$ be the max and min tweak parameters, respectively.
10. $\sigma \leftarrow \sigma$-$exploitation$ /*tweak parameter */
11. **repeat**
12. *newTS* = TWEAK(*TS*, σ, P) /* generating new candidate solution */
13. *TSO* $\leftarrow$ signal outputs obtained by simulating M for every test input in *newTS*
14. *whole-test-suite-coverage* $\leftarrow$ coverage achieved by *newTS* over M
15. *accumulative-coverage* $\leftarrow$ *accumulative-coverage* + *whole-test-suite-coverage*
16. **if** $O(TSO) > highestFound$ **:**
17. $highestFound = O(TSO)$
18. $TS = newTS$
19. **if** *accumulative-coverage* has reached a plateau at a value less than %100 **and** $P < P_{max}$ **:**
20. $P = P + 1$
21. Reduce σ proportionally as *accumulative-coverage* increases over *initial-coverage*
22. **until** maximum resources spent
23. **return** TS

Figure 3: Our output diversity (OD) test generation algorithm for Simulink models.

does not change the number of segments either (line 12). We increase P only when the accumulative structural coverage achieved by the existing generated test suites reaches a plateau at a value less than %100, i.e., remains constant for some consecutive iterations of the algorithm (lines 19-20). Further, although not shown in the algorithm, we do not increase P if the last increase in P has not improved the accumulative coverage.

(2) The tweak operator in OD (line 12) is explorative at the beginning and becomes more exploitative as the search progresses. Our tweak is similar to the one used in (1+1) EA algorithm [28]. At each iteration, we shift every signal $sg \in TS$ denoted by $\{(k_1, v_1), \ldots, (k_p, v_p)\}$ as follows: We add values x_i (respectively y_i) to every v_i (respectively k_i) for $1 \leq i \leq p$. The x_i (respectively y_i) values are selected from a normal distribution with mean $\mu = 0$ and variance $\sigma \times (max_{\mathcal{R}} - min_{\mathcal{R}})$ (respectively $\sigma \times k$), where $\mathcal{R}$ is the signal range and k is the number of simulation steps. We control the degree of exploration and exploitation of our search using σ. Given that the search space of input signals is very large, if we start by a purely exploitative search (i.e., $\sigma = 0.01$), our result will be biased by the initially randomly selected solution. To reduce this bias, we start by performing a more explorative search (i.e., $\sigma = 0.5$). However, if we let the search remain explorative, it may reduce to a random search. Hence, we reduce σ iteratively in OD such that the amount of reduction in σ is proportional to the increase in the accumulative structural coverage obtained by the generated test suites (line 21). Finally, we note that the tweak operator takes the signal segments P as an input (line 12) and, in case the number of signal segments has increased from the previous iteration, it ensures to increase the number of segments in signal sg.

4. EXPERIMENT SETUP

In this section, we present the research questions. We further describe our study subjects, our metrics to measure the fault revealing ability of test generation algorithms and the way we approximate their oracle cost. We finally provide our experiment design.

4.1 Research Questions

RQ1 (Sanity check). *How does the fault revealing ability of the OD algorithm compare with that of a random test generation strategy?* We investigate whether OD is able to perform better than random testing which is a baseline of comparison. We compare the fault revealing ability of the test suites generated by OD when used with each of the O_v and O_f objective functions with that of the test suites generated by a random test generation algorithm.

RQ2 (Comparing O_v and O_f). *How does the O_f diversity objective perform compared to the O_v diversity objective?* We compare the ability of the test suites generated by OD with O_v and O_f in revealing faults in time-continuous Simulink models. In particular, we are interested to know if, irrespective of the size of the generated test suites, any of these two diversity objectives is able to consistently reveal more faults across different study subjects and different fault types than the other.

RQ3 (Comparison with SLDV). *How does the fault revealing ability of the OD algorithm compare with that of SLDV?* With this question, we compare an output diversity approach (OD) with an approach based on structural coverage (SLDV) in generating effective test suites for Simulink models. This question, further, enables us to provide evidence that our approach is able to outperform the most widely used industry strength Simulink model testing tool. Finally, in contrast to **RQ1** and **RQ2** where we applied OD to time-continuous Simulink models, this question has to focus on discrete models because SLDV is only applicable to time-discrete Simulink models. Hence, this question allows us to investigate the capabilities of OD in finding faults for discrete Simulink models, as well.

4.2 Study Subjects

We use four Simulink models in our experiments: Two industrial models, Clutch Position Controller (CPC) and Flap Position Controller (FPC), from Delphi, and two public domain models, Cruise Controller (CC) [49] and Clutch Lockup Controller (CLC), from the Mathworks website [45]. Table 1 shows key characteristics of these models. CPC and CC include Stateflows, and FPC and CLC are Simulink models without Stateflows. FPC and CPC are time-continuous models and incompatible with SLDV. The CC model, which is the largest model from the SLDV tutorial examples, is compatible with SLDV. Since the other tutorial examples of SLDV were small, we modified the CLC model from the Mathworks website to become compatible with SLDV by replacing the time-continuous and other SLDV-incompatible blocks with their equivalent or approximating discrete blocks. We have made the modified version of CLC available at [30]. Note that we were not able to make CPC, FPC or any other Delphi Simulink models compatible with SLDV since they contained complex S-Function blocks, and hence, they have to be almost reimplemented before SLDV can be applied to them. The coverage criterion used by both SLDV and OD in Figure 3 is *decision coverage* [51], also known as branch coverage, that aims to ensure that each one of the possible branches from each decision point is executed at least once and thereby ensuring that all reachable blocks are executed. We chose branch coverage as it is the predominant coverage criterion in the literature [19]. Table 1 reports the total number of decision points in our study subjects. In addition, we report the total number of Simulink blocks and Stateflow states as well as input variables and configuration parameters for each model. CPC and FPC are representative models from the automotive domain with many input variables and blocks. In order to compare OD with SLDV, we use CC from the Mathworks website and the modified version of CLC as both models are reasonably large and complex, and yet compatible with SLDV.

4.3 Measuring Fault Revealing Ability

We need test oracles to automatically assess the fault revealing ability of generated test suites in our experimental setting. As discussed earlier, in our work, test oracles depend on *manual* inspection of output signals and on engineers' estimates of acceptable deviations from the expected results. To measure the fault revealing

Table 1: Characteristics of our study subject Simulink models.

Name	Publicly Available	No. Inputs	No. Configs	No. Blocks/ States	No. Decision Points
CPC	No	10	41	590	126
FPC	No	21	65	810	120
CC	Yes	6	3	43	12
CLC	Yes	2	0	81	10

ability of a test suite, we use a quantitative notion of oracle defined as the largest normalized distance, among all output signals, between test results and the ground truth oracle [6]. For the purpose of experimentation, we use fault-free versions of our subject models to produce the ground truth oracle. Let TS be a test suite generated by either OD or SLDV for a given faulty model M, let $O = \{sg_1, \ldots, sg_q\}$ be the set of output signals obtained by running M for the test inputs in TS, and let $G = \{g_1, \ldots, g_q\}$ be the corresponding ground truth oracle signals. We define our *quantitative oracle* (QO) as follows: $QO(M, TS) = MAX_{1 \leq i \leq q} \hat{dist}(sg_i, g_i)$. We use a threshold value THR to translate the quantitive oracle QO into a boolean fault revealing measure denoted by FR. Specifically, FR returns true (i.e, $QO(M, TS) > THR$) if there is at least one test input in TS for which the output of M sufficiently deviates from the ground truth oracle such that a manual tester conclusively detects a failure. Otherwise, FR returns false. In our work, we set THR to 0.2. We arrived at this value for THR based on our experience and discussions with domain experts. In our experiments, in addition, we obtained and evaluated the results for $THR = 0.15$ and $THS = 0.25$ and showed that our results were not sensitive to such small changes in THR.

4.4 Test Oracle Cost Estimation

Since we assume that test oracles are evaluated manually, to compare the fault revealing ability of OD and SLDV (**RQ3**), we need to ensure that the test suites used for comparison have the same (oracle) cost. The oracle cost of a test suite depends on the size of the test suite and the complexity of input data. The latter in our work is determined by the number of signal segments (P) of each input signal. More precisely, test suites $TS = \{I_1, \ldots I_{q_1}\}$ and $TS' = \{I'_1, \ldots I'_{q_2}\}$ have roughly the same oracle cost if (1) they have the same size ($q_1 = q_2$), and (2) the input signals in TS and TS' have the same number of segments. That is, for every test input $I_i = (sg_1, \ldots, sg_n)$ in TS (respectively TS'), there exists some test input $I_j = (sg'_1, \ldots, sg'_n)$ in TS' (respectively TS) such that sg_k and sg'_k (for $1 \leq k \leq n$) have the same number of segments. In our experiments described in Section 4.5, we ensure that the test suites used for comparison of different test generation algorithms satisfy the above two conditions, and hence, can be used as a proper basis to compare algorithms.

4.5 Experiment Design

We developed a comprehensive list of Simulink fault patterns and have made it available at [30]. Examples of fault patterns include incorrect signal data type, incorrect math operation, and incorrect transition condition. We identified these patterns through our discussions with senior Delphi engineers and by reviewing the existing literature on mutation operators for Simulink models [61, 9, 7, 58]. We have developed an automated fault seeding program to automatically generate 44 faulty versions of CPC, 30 faulty versions of FPC, 17 faulty versions of CC, and 13 faulty versions of CLC (one fault per each faulty model). In order to achieve diversity in terms of the location and the types of faults, our automation seeded faults of different types and in different parts of the models. We ensured that every faulty model remains executable (i.e., no syntax error).

Having generated the fault-seeded models, we performed two sets of experiments, **EXP-I** and **EXP-II**, described below.

EXP-I focuses on answering **RQ1** and **RQ2** using the 74 faulty versions of the time-continuous models from Table 1, i.e., CPC and FPC. We ran the OD algorithm in Figure 3 with vector-based (O_v) and feature-based (O_f) objective functions. For each faulty model and each objective function, we ran OD for 400 sec and created a test suite of size q where q took the following values: 3, 5, and 10. We chose to examine the fault revealing ability of *small* test suites to emulate current practice where test suites are small so that the test results can be inspected manually. We repeated OD 20 times to account for its randomness. Specifically, we sampled 444 different test suites and repeated each sampling 20 times (i.e., in total, 8880 different test suites were generated for **EXP-I**). Overall, **EXP-I** took about 1000 hours in execution time on a notebook with a 2.4GHz i7 CPU, 8 GB RAM, and 128 GB SSD.

EXP-II answers **RQ3** and is performed on the SLDV-compatible subject models from Table 1, i.e., CC and CLC. To answer **RQ3**, we compare the fault revealing ability of the test suites generated by SLDV with that of the test suites generated by OD. We give SLDV and OD the same execution time budget (120 sec in our experiment). This time budget was sufficient for SLDV to achieve a high level of structural coverage over the subject models. Further, we ensure that the generated test suites have the same test oracle cost. Specifically, for each faulty model M, we first use SLDV to generate a test suite TS_M based on the decision coverage criterion within the time allotted (120 sec). We then apply OD to M to generate a test suite TS'_M such that TS_M and TS'_M have the same test oracle cost (see Section 4.4). We have implemented a Matlab script that enables us to extract the size of the test suites as well as the number of input signal segments for each individual test input of TS_M. Further, we have slightly modified the OD algorithm in Figure 3 so that it receives as input the desired number P of signal segments for each input signal and it does not modify P during search. Finally, we note that while SLDV is deterministic and is executed once per input model, OD is randomized, and hence, we rerun it 20 times for each faulty model.

5. RESULTS AND DISCUSSIONS

This section provides responses, based on our experiment design, for research questions **RQ1** to **RQ3** described in Section 4.

RQ1 (Sanity). To answer **RQ1**, we ran **EXP-I**, and further, in order to compare with random testing, for each faulty version, we randomly generated test suites with size 3, 5 and 10. We ensured that each test suite generated by random testing has the same oracle cost as the corresponding test suites generated by the OD algorithm. Moreover, similar to OD, we reran random testing 20 times. Figures 4(a) to (c) compare the fault revealing ability of random testing and OD with the objective functions O_f and O_v. Each distribution in Figures 4(a) to (c) contains 74 points. Each point relates to one faulty model and represents either the average quantitive oracle QO or the average fault revealing measure FR over 20 different test suites with a fixed size and obtained by applying a test generation algorithm to that faulty model. Note that the FR values are computed based on three different thresholds THR of 0.2, 0.15, and 0.25. For example, a point with (x = R) and (y = 0.104) in the QO plot of Figure 4(a) indicates that the 20 different random test suites with size 3 generated for one faulty model achieved an average QO of 0.104. Similarly, a point with (x = OD(O_f)) and (y = 0.65) in any of the FR plots of Figure 4(b) indicates that among the 20 different test suites with size 5 generated by applying OD

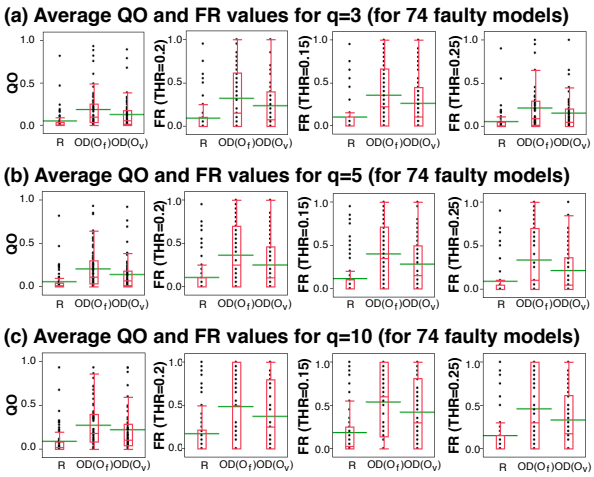

Figure 4: Boxplots comparing average quantitative oracle values (QO) and fault revealing measures (FR) of OD (with both diversity objectives) and random test suites for different thresholds and different test suite sizes.

with objective function O_f to one faulty model, 13 test suites were able to reveal the fault (i.e., $FR = 1$) and 7 could not reveal that fault (i.e., $FR = 0$).

To statistically compare the QO and FR values, we performed the non-parametric pairwise Wilcoxon Pairs Signed Ranks test [11], and calculated the effect size using Cohen's d [15]. The level of significance (α) was set to 0.05, and, following standard practice, d was labeled "small" for $0.2 \leq d < 0.5$, "medium" for $0.5 \leq d < 0.8$, and "high" for $d \geq 0.8$ [15].

Testing differences in the average QO and FR distributions, for all the three thresholds and with all the three test suite sizes, shows that OD with both objective functions O_f and O_v performs significantly better than random test generation. In addition, for all the comparisons between OD and random, the effect size is consistently "high" for OD with O_f and "medium" for OD with O_v. *To summarize,* the fault revealing ability of OD outperforms that of random testing.

RQ2 (Comparing O_f with O_v). The results in Figure 4 compare the average QO and FR values for the feature-based, OD(O_f), and the vector-based, OD(O_v), output diversity algorithms. As for the QO distributions, the statistical test results indicate that OD(O_f) performs significantly better than OD(O_v) for test suite sizes 3 and 5 with a "small" effect size. For test suite size 10, there is no statistically significant difference, but OD(O_f) achieves higher mean and median QO values compared to OD(O_v). As for the FR distributions, the improvements of OD(O_f) over OD(O_v) are not statistically significant. However, for all the three thresholds and with all the test suite sizes, OD(O_f) consistently achieves higher mean and median FR values compared to OD(O_v). Specifically, with threshold 0.2, the average FR is .33, .35 and .48 for OD(O_f), and .23, .24 and .36 for OD(O_v) for test suite sizes 3, 5, and 10, respectively. That is, across all the faults and with all test suite sizes, the average probability of detecting a fault is about %10 higher when we use OD(O_f) instead of OD(O_v). *To summarize,* the fault revealing ability of the OD algorithm with the feature-based diversity objective is higher than that of the OD algorithm with the vector-based diversity objective.

RQ3 (Comparison with SLDV). To answer RQ3, we used the better diversity objective from **RQ2** (i.e., OD with the feature-based diversity objective) and performed **EXP-II** on 30 faulty models of

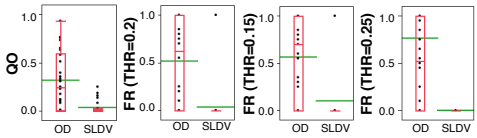

Figure 5: Boxplots comparing quantitative oracle values (QO) and fault revealing abilities of OD and SLDV for different thresholds.

CC and CLC. We evaluate the fault revealing ability of SLDV and OD by comparing the quantitative oracle QO and the fault revealing measure FR values obtained over these 30 faulty models. In addition, we investigate if any of SLDV and OD subsumes the other technique (fault revealing subsumption). That is, we determine if any of OD and SLDV does not find any additional faults missed by the other technique. Finally, we compare the structural coverage achieved by each of SLDV and OD over these 30 faulty models. We report coverage results for two reasons: (1) We confirm our earlier claim that with the timeout of 120 sec used in **EXP-II**, SLDV has been able to achieve high structural coverage. (2) We provide evidence that achieving higher structural coverage does not necessarily lead to better fault revealing ability.

Comparing fault revealing ability. We computed QO and FR with three THR values of 0.15, 0.20, and 0.25 over the test suites generated by OD and SLDV. Figure 5 compares the distributions obtained for the 30 faulty models of CC and CLC. Recall that SLDV is deterministic, and OD is randomized. So, in Figure 5, each point in distributions related to SLDV shows the value of QO or FR obtained for one and the only one test suite generated by SLDV for one faulty model. In contrast, each point in distributions related to OD shows the *average* value of QO or FR for 20 different test suites generated by OD for each faulty model. Further, for each faulty model, SLDV yields a FR value of one or zero, respectively indicating whether SLDV reveals the fault or not. However, for OD, we compute an average FR value over 20 different runs, which is between zero and one, indicating an estimated probability for OD to reveal a fault.

As shown in Figure 5, the QO and FR values obtained for SLDV are very small compared to those obtained by OD. Testing differences in QO and FR distributions with all the three thresholds shows that OD performs significantly better than SLDV. In addition, for all the four comparisons depicted in Figure 5, the effect size is "high".

Fault revealing subsumption. Tables 2 and 3 compare performance of OD and SLDV for individual faults. Specifically, Table 2 shows, for each faulty model, the distribution of QO values obtained by OD and the single value of QO obtained by SLDV. Note that $\bar{x}$ shows the mean, and Q_1, Q_2, and Q_3 refer to the three quartiles of the QO distribution obtained by 20 different runs of OD (i.e., Q_1, Q_2, and Q_3 are the 25th, 50th, and 75th percentiles, respectively). In addition, we report (as denoted by $\mathcal{P}$ in Table 2) the percentages of OD runs that achieve a QO value greater than or equal to that obtained by SLDV. For example, for faults 1, 2, and 3 in Table 2, 100%, 100% and 30% of the OD runs, respectively, yield QO values that are not worse than those generated by SLDV. Table 3 shows, for each faulty model and when considering FR with threshold 0.2, whether SLDV is able to identify the fault or not. We depict detection of a fault by SLDV with ■. Further, the table shows, out of the 20 runs, how many times OD is able to find the fault. Note that the results for the thresholds 0.15 and 0.25 were similar to those in Table 3.

Based on Table 2, six faults go undetected by both SLDV and OD irrespective of the threshold value (i.e., for six faults, we have

Table 2: Quantitative oracle QO distributions for OD and single QO values for SLDV per each faulty model.

	1		2		3		4		5		6		7		8	
	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV
$\bar{x}$	0.631		0		0.011		0.002		0.323		0.581		0.228		0.175	
Q_1	0.566		0		0		0.001		0.246		0.575		0		0.027	
Q_2	0.628	0.013	0	0	0	0	0.002	0	0.347	0.012	0.589	0.013	0.248	0	0.043	0
Q_3	0.705		0		0.002		0.003		0.462		0.596		0.425		0.186	
$\mathcal{P}$	1.0		1.0		0.3		1.0		0.9		1.0		0.65		1.0	

	9		10		11		12		13		14		15		16	
	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV
$\bar{x}$	0.913		0.356		0.651		0.238		0.512		0.082		0.119		0.306	
Q_1	0.81		0.066		0.374		0.008		0.405		0		0		0.141	
Q_2	0.925	0.023	0.345	0.012	0.762	0.011	0.232	0.012	0.579	0.011	0.006	0.011	0.062	0.012	0.032	0
Q_3	0.987		0.665		0.925		0.449		0.579		0.18		0.257		0.461	
$\mathcal{P}$	1.0		0.85		0.85		0.75		1.0		0.4		0.55		1.0	

	17		18		19		20		21		22		23		24	
	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV
$\bar{x}$	0.102		0.733		0		0		0.239		0.25		0		0.756	
Q_1	0.041		0.716		0		0		0.224		0.25		0		0.752	
Q_2	0.061	0.007	0.744	0.082	0	0	0	0	0.25	0.158	0.25	0.25	0	0	0.765	0.082
Q_3	0.199		0.763		0		0		0.25		0.25		0		0.769	
$\mathcal{P}$	1.0		1.0		1.0		1.0		1.0		1.0		1.0		1.0	

	25		26		27		28		29		30	
	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV
$\bar{x}$	0.76		0		0.764		0		0.395		0.332	
Q_1	0.753		0		0.762		0		0.381		0.11	
Q_2	0.766	0.135	0	0	0.767	0.185	0	0	0.397	0	0.442	0
Q_3	0.768		0		0.77		0		0.412		0.445	
$\mathcal{P}$	1.0		1.0		1.0		1.0		1.0		1.0	

Table 3: The number of fault revealing runs of OD (out of 20) for our 30 faulty models, and the fault(s) that SLDV is able to find with a threshold (THR) of 0.2.

Faults	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
OD	20	0	0	0	16	20	11	5	20	14	17	11	20	4	5	14	2	20	0	0	0	0	20	0	20	20	0	20	0	15	15
SLDV																							■								

$QO = 0$ for both SLDV and all OD runs). There is no fault that SLDV can possibly detect ($QO > 0$) but OD cannot. For 23 faults, all 20 runs of OD yield results that are at least as good as those of SLDV ($\mathcal{P} = 1$). For all the faults, the average of QO obtained by OD (denoted by $\bar{x}$) is higher than the value of QO obtained by SLDV. Finally, SLDV totally fails to detect faults 3, 4, 7, 8, 16, 29, and 30, while some runs of OD are able to identify these seven faults. Based on Table 3, SLDV identifies only one fault with a threshold of 0.2, and that particular fault is also detected by all the 20 runs of OD. The results for thresholds 0.15 and 0.25 are similar.

Comparing coverage. Figure 6 compares the structural coverage percentages (i.e., decision coverage) achieved by test suites generated by OD and SLDV over the faulty models of CC and CLC. As before, the distribution for SLDV shows the percentages of structural coverage achieved by individual test suites generated for individual faulty models, while the distribution for OD shows the average of structural coverage percentages obtained by 20 different runs of OD for each faulty model. As shown in the figure, SLDV was able to achieve, on average, a coverage of 94% for CC models and 85% for CLC models. In contrast, OD achieved, on average, a coverage of 84% for CC and 75% for CLC. In addition, SLDV has been able to cover 29 out of the 30 fault-seeded blocks, and OD covered 28 out of the 30 fault-seeded blocks. This shows that within 120 sec, SLDV had sufficient time to cover the structure of the 30 faulty models (only one fault-seeded block was missed). Indeed, for 22 out of 30 faults, SLDV terminated before the time-out period elapsed. Hence, by increasing the execution time, it is unlikely that SLDV's fault revealing ability would be impacted.

In summary, our comparison of SLDV and OD shows that: (1) For both studied models, OD is able to reveal significantly more faults

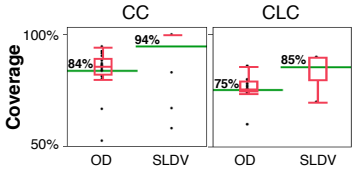

Figure 6: The percentages of branch (decision) coverage achieved by OD and SLDV over the faulty versions of CC and CLC subject models.

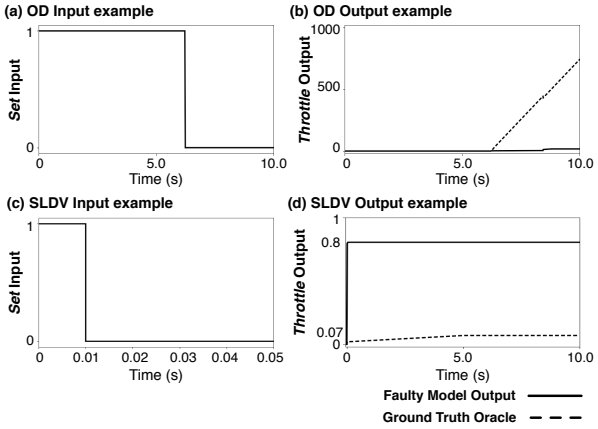

Figure 7: Examples of test inputs and output signals generated by SLDV and OD algorithm.

compared to SLDV. (2) OD subsumes SLDV in revealing faults: Any fault identified by SLDV is also identified by OD. (3) SLDV was able to cover a large part of the underlying models within the given timeout period (i.e., 29 out of the 30 fault-seeded blocks), and further, it achieved slightly higher decision coverage over study subjects compared to OD. However, covering a fault does not necessarily lead to detecting that fault. In particular, SLDV was able to reveal only one out of the 29 faults that it could cover. (4) Finally, our results on comparing SLDV and OD are not impacted by small modifications in the threshold values used to compute the fault revealing measure FR.

Discussion. *Why does SLDV perform poorly compared to OD?* Our results in **RQ3** show that, compared to the output diversity (OD) algorithm, SLDV is less effective in revealing faults in Simulink models. In our experiment, even though test suites generated by SLDV cover most faulty parts of the Simulink models, the outputs produced by these test suites either do not deviate or only slightly deviate from the ground truth oracle, hence yielding very small QO values. In contrast, OD generates test suites with output signals that are more distinct from the ground truth oracle. Note that, as discussed in Section 2, any deviation should exceed some threshold to be conclusively deemed a failure. For example, Figures 7(b) and (d) show two output signals (solid lines) of a faulty model together with the oracle signals (dashed lines) generated by OD and SLDV, respectively. Note that the range of the Y-axis in Figure 7(b) is 1000 times larger than that in Figure 7(d). Hence, the deviation from the oracle in Figure 7(b) is much larger than that in Figure 7(d). In particular, the signals in Figures 7(b) and (d) respectively produce QO values of 0.43 and 0.01. Therefore, the output in Figure 7(b) is more fault revealing that the one in Figure 7(d).

Since SLDV is commercial and its technical approach description is not publicly available, we cannot precisely determine the

reasons for its poor performance. We conjecture, however, that the reason for SLDV's poor fault finding lies in its input signal generation strategy. Specifically, all value changes in the input signals generated by SLDV typically occur during the very first simulation steps, and then, the signals remain constant for the most part and until the end of the simulation time. In contrast, changes in the input signals generated by OD can occur at any time during the entire simulation period. For example, Figures 7(a) and (c) show two examples of input signals generated by OD and SLDV, respectively. In both case, the signal value changes from one to zero. However, for the signal in Figure 7(a), the change occurs almost in the middle of the simulation (at 6 sec), while in Figure 7(c), the change occurs after the first step (at 0.01 sec). Signals in Figures 7(a) and (c) happen to cover exactly the same branches of the underlying model. However, they, respectively, yield the outputs in Figures 7(b) and (d) with drastically different fault revealing ability.

6. RELATED WORK

Modeling is by no means new to the testing and verification community and has already been the cornerstone of a number of well-studied techniques. In particular, two well known techniques, *model-based testing* and *model checking*, have been previously applied to test and verify Simulink models. Model-based testing relies on models to generate test scenarios and oracles for implementation-level artifacts. A number of model-based testing techniques have been applied to Simulink models with the aim of achieving high structural coverage or detecting a large number of mutants. For example, search-based approaches [54, 55], reachability analysis [33, 21], guided random testing [43, 44], and a combination of these techniques [50, 39, 42, 36, 20, 8] have been previously applied to Simulink models to generate coverage-adequate test suites. Alternatively, various search-based [61, 62] and bounded reachability analysis [9] techniques have been used to generate mutant-killing test suites from Simulink models. These techniques aim to generate test suites as well as oracles from models that are considered to be correct. In reality, however, Simulink models might contain faults. Hence, in our work, we propose techniques to help testing complex Simulink models for which automated and precise test oracles are not available. Further, even though in Simulink, every variable is described using signals, unlike our work, none of the above techniques generate test inputs in terms of signals.

Model checking is an exhaustive verification technique and has a long history of application in software and hardware verification [13]. It has been previously used to detect faults in Simulink models [14, 21, 5, 31] by showing that a path leading to an error (e.g., an assertion or a runtime error) is reachable, or by maximizing structural coverage (e.g., by executing as many paths as possible in a model). To solve the reachability problem or to achieve high coverage, these techniques often extract constraints from the underlying Simulink models and feed the constraints into some constraint solver or SAT solver. Some alternative techniques [52, 23, 3] translate Simulink models into code and use existing code analysis tools such as Java PathFinder [24] or KLEE [10] to detect faults. All these approaches only work for code generation models with linear behavior and fail to test or verify simulation models with time-continuous behavior. Our approach, however, is applicable to both simulation and code generation Simulink models.

Recent work in the intersection of Simulink testing and signal processing has focused on test input signal generation using evolutionary search methods [4, 56, 27, 53]. These techniques, however, assume automated oracles, e.g., assertions, are provided. Since test oracles are automated, they do not pose any restriction on the shape of test inputs. In our work, however, we restrict variations in input signal shapes as more complex inputs increase the oracle cost. Similar to our work, the work of [60] proposes a set of signal features. These features are viewed as basic constructs which can be composed to specify test oracles. In our work, since oracle descriptions do not exist, we use features to improve test suite effectiveness by diversifying feature occurrences in test outputs.

Our algorithm uses whole test suite generation [19] that was proposed for testing software code. This approach evolves an entire test suite, instead of individual test cases, with the aim of covering all structural coverage goals at the same time. Our algorithm, instead, attempts to diversify test outputs by taking into account all the signal features (see Figure 2) at the same time. The notion of output diversity in our work is inspired by the output uniqueness criterion [1, 2]. As noted in [2], effectiveness of this criterion depends on the definition of *output difference* and differs from one context to another. While in [1, 2], output differences are described in terms of the textual, visual or structural aspects of HTML code, in our work, output differences are characterized by signal shape features.

In our earlier work, we proposed a number of test generation algorithms including an algorithm based on output diversity for mixed discrete-continuous Stateflow models [29]. Our current paper provides a test generation algorithm for the entire Simulink including Stateflows. Moreover, our output diversity algorithm in this paper is defined based on a feature-based notion of output diversity, and our evaluation shows that this approach to diversity outperforms the vector-based approach that was used in our previous paper [29].

7. CONCLUSIONS

Simulink is a prevalent modeling language for Cyber Physical Systems (CPSs) and supports the two main CPS modeling goals: automated code generation and simulation, i.e., design time testing. In this paper, we distinguished Simulink simulation and code generation models and illustrated differences in their behaviors using examples. In contrast to the existing testing approaches that are only applicable to code generation Simulink models, we proposed a testing approach for both kinds of Simulink models based on our notion of feature-based output diversity for signals. Our testing approach is implemented using a meta-heuristic search algorithm that is guided to produce test outputs exhibiting a diverse set of signal features. Our evaluation is performed using two industrial and two public domain Simulink models and shows that (1) Our approach significantly outperforms random test generation. (2) The average fault finding ability of our algorithm when used with the feature-based notion of output diversity is higher than that of our approach when output diversity is measured based on the Euclidean distance between signal vectors. (3) We empirically compared our approach with Simulink Design Verifier (SLDV) which is the only Simulink toolbox provided by Mathworks and dedicated to testing. Our comparison shows that our approach is able to reveal significantly more faults compared to SLDV, and further, our approach is able to find the faults identified by SLDV with a 100% probability. Hence, our approach subsumes SLDV.

SLDV and Reactis Validator [40] are the most well-known commercial tools for testing Simulink models. According to [14], the underlying approach of Reactis is similar to that of SLDV. We leave the comparison with Reactis for future work. We further plan to improve our feature-based algorithm by optimizing and customizing the set of feature shapes for a given model. To do so, we intend to identify shapes that are more likely to occur in a given model's outputs and modify the derivative feature functions to observe changes over varying time intervals.

8. REFERENCES

[1] N. Alshahwan and M. Harman. Augmenting test suites effectiveness by increasing output diversity. In *ICSE 2012*, pages 1345–1348. IEEE Press, 2012.

[2] N. Alshahwan and M. Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *ISSTA 2014*, pages 181–192. ACM, 2014.

[3] D. Balasubramanian, C. S. Pasareanu, M. W. Whalen, G. Karsai, and M. Lowry. Polyglot: modeling and analysis for multiple statechart formalisms. In *ISSTA 2011*, pages 45–55. ACM, 2011.

[4] A. Baresel, H. Pohlheim, and S. Sadeghipour. Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In *GECCO 2003*, pages 2428–2441. Springer, 2003.

[5] J. Barnat, L. Brim, J. Beran, T. Kratochvila, and I. R. Oliveira. Executing model checking counterexamples in Simulink. In *TASE 2012*, pages 245–248. IEEE, 2012.

[6] 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.

[7] N. T. Binh et al. Mutation operators for Simulink models. In *KSE 2012*, pages 54–59. IEEE, 2012.

[8] F. Bohr and R. Eschbach. SIMOTEST: A tool for automated testing of hybrid real-time Simulink models. In *ETFA 2011*, pages 1–4. IEEE, 2011.

[9] 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.

[10] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI 2008*, volume 8, pages 209–224, 2008.

[11] J. A. Capon. *Elementary Statistics for the Social Sciences: Study Guide*. Wadsworth Publishing Company, 1991.

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

[13] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[14] R. Cleaveland, S. A. Smolka, and S. T. Sims. An instrumentation-based approach to controller model validation. In *MDRAS 2008*, pages 84–97. Springer, 2008.

[15] J. Cohen. *Statistical power analysis for the behavioral sciences (rev)*. Lawrence Erlbaum Associates, Inc, 1977.

[16] K. Czarnecki and U. W. Eisenecker. Generative programming. *Edited by G. Goos, J. Hartmanis, and J. van Leeuwen*, page 15, 2000.

[17] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE 2007*, pages 37–54. IEEE Computer Society, 2007.

[18] G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In *QSIC 2011*, pages 31–40. IEEE, 2011.

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

[20] A. A. Gadkari, A. Yeolekar, J. Suresh, S. Ramesh, S. Mohalik, and K. Shashidhar. Automotgen: Automatic model oriented test generator for embedded control systems. In *CAV 2008*, pages 204–208. Springer, 2008.

[21] G. Hamon, B. Dutertre, L. Erkok, J. Matthews, D. Sheridan, D. Cok, J. Rushby, P. Bokor, S. Shukla, A. Pataricza, et al. Simulink Design Verifier - Applying Automated Formal Methods to Simulink and Stateflow. In *AFM 2008*. Citeseer, 2008.

[22] M. P. Heimdahl, L. Duan, A. Murugesan, and S. Rayadurgam. Modeling and requirements on the physical side of cyber-physical systems. In *TwinPeaks 2013*, pages 1–7. IEEE, 2013.

[23] D. Holling, A. Pretschner, and M. Gemmar. 8Cage: lightweight fault-based test generation for Simulink. In *ASE 2014*, pages 859–862. ACM, 2014.

[24] JPF. Java pathfinder tool-set. `http://babelfish.arc.nasa.gov/trac/jpf`. [Online; accessed 17-Aug-2015].

[25] J. Krizan, L. Ertl, M. Bradac, M. Jasansky, and A. Andreev. Automatic code generation from MATLAB/Simulink for critical applications. In *CCECE 2014*, pages 1–6. IEEE, 2014.

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

[27] F. Lindlar, A. Windisch, and J. Wegener. Integrating model-based testing with evolutionary functional testing. In *ICSTW 2010*, pages 163–172. IEEE, 2010.

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

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

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

[31] M. Mazzolini, A. Brusaferri, and E. Carpanzano. Model-checking based verification approach for advanced industrial automation solutions. In *ETFA 2010*, pages 1–8. IEEE, 2010.

[32] P. McMinn, M. Stevenson, and M. Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *ISSTA 2010*, pages 1–4. ACM, 2010.

[33] S. Mohalik, A. A. Gadkari, A. Yeolekar, K. Shashidhar, and S. Ramesh. Automatic test case generation from Simulink/Stateflow models using model checking. *STVR*, 24(2):155–180, 2014.

[34] P. Nardi, M. E. Delamaro, L. Baresi, et al. Specifying automated oracles for Simulink models. In *RTCSA 2013*, pages 330–333. IEEE, 2013.

[35] P. A. Nardi. *On test oracles for Simulink-like models*. PhD thesis, Universidade de Sao Paulo, 2014.

[36] P. Peranandam, S. Raviram, M. Satpathy, A. Yeolekar, A. Gadkari, and S. Ramesh. An integrated test generation tool for enhanced coverage of Simulink/Stateflow models. In *DATE 2012*, pages 308–311. IEEE, 2012.

[37] B. Porat. *A course in digital signal processing*, volume 1. Wiley New York, 1997.

[38] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *ICSE 2005*, pages 392–401. ACM, 2005.

[39] Reactive Systems Inc. Reactis Tester. `http://www.reactive-systems.com/simulink-testing-validation.html`, 2010. [Online; accessed 17-Aug-2015].

[40] Reactive Systems Inc. Reactis Validator. http://www.reactive-systems.com/simulink-testing-validation.html, 2010. [Online; accessed 17-Aug-2015].

[41] D. J. Richardson, S. L. Aha, and T. O. O'malley. Specification-based test oracles for reactive systems. In *ICSE 1992*, pages 105–118. ACM, 1992.

[42] M. Satpathy, A. Yeolekar, P. Peranandam, and S. Ramesh. Efficient coverage of parallel and hierarchical stateflow models for test case generation. *STVR*, 22(7):457–479, 2012.

[43] M. Satpathy, A. Yeolekar, and S. Ramesh. Randomized directed testing (REDIRECT) for Simulink/Stateflow models. In *EMSOFT 2008*, pages 217–226. ACM, 2008.

[44] S. Sims and D. C. DuVarney. Experience report: the Reactis validation tool. *SIGPLAN*, 42(9), 2007.

[45] The MathWorks Inc. Building a Clutch Lock-Up Model. http://nl.mathworks.com/help/simulink/examples/building-a-clutch-lock-up-model.html?refresh=true. [Online; accessed 17-Aug-2015].

[46] The MathWorks Inc. C Code Generation from Simulink. http://nl.mathworks.com/help/dsp/ug/generate-code-from-simulink.html. [Online; accessed 17-Aug-2015].

[47] The MathWorks Inc. Modeling a Fault-Tolerant Fuel Control System. http://nl.mathworks.com/help/simulink/examples/modeling-a-fault-tolerant-fuel-control-system.html. [Online; accessed 17-Aug-2015].

[48] The MathWorks Inc. Simulink. http://www.mathworks.nl/products/simulink. [Online; accessed 17-Aug-2015].

[49] The MathWorks Inc. Simulink Deign Verifier Cruise Control Test Generation. http://nl.mathworks.com/help/sldv/examples/extending-an-existing-test-suite.html?prodcode=DV&language=en. [Online; accessed 17-Aug-2015].

[50] The MathWorks Inc. Simulink Design Verifier. http://nl.mathworks.com/products/sldesignverifier/?refresh=true. [Online; accessed 17-Aug-2015].

[51] The MathWorks Inc. Types of Model Coverage. http://nl.mathworks.com/help/slvnv/ug/types-of-model-coverage.html. [Online; accessed 17-Aug-2015].

[52] R. Venkatesh, U. Shrotri, P. Darke, and P. Bokil. Test generation for large automotive models. In *ICIT 2012*, pages 662–667. IEEE, 2012.

[53] B. Wilmes and A. Windisch. Considering signal constraints in search-based testing of continuous systems. In *ICSTW 2010*, pages 202–211. IEEE, 2010.

[54] A. Windisch. Search-based testing of complex simulink models containing stateflow diagrams. In *ICSE 2009*, pages 395–398. IEEE, 2009.

[55] A. Windisch. Search-based test data generation from stateflow statecharts. In *GECCO 2010*, pages 1349–1356. ACM, 2010.

[56] A. Windisch, F. Lindlar, S. Topuz, and S. Wappler. Evolutionary functional testing of continuous control systems. In *GECCO 2009*, pages 1943–1944. ACM, 2009.

[57] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier, 2011.

[58] Y. F. Yin, Y. B. Zhou, and Y. R. Wang. Research and improvements on mutation operators for Simulink models. In *AMM 2014*, volume 687, pages 1389–1393. Trans Tech Publ, 2014.

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

[60] J. Zander-Nowicka. *Model-based testing of real-time embedded systems in the automotive domain*. Fraunhofer-IRB-Verlag, 2008.

[61] Y. Zhan and J. A. Clark. Search-based mutation testing for Simulink models. In *GECCO 2005*, pages 1061–1068. ACM, 2005.

[62] Y. Zhan and J. A. Clark. A search-based framework for automatic testing of MATLAB/Simulink models. *JSS*, 81:262–285, 2008.