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 (\(\text{sum}\)) 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
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.

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. 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.

**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 $\Delta t$. We define a signal $sg$ as a function $sg : (0,\Delta t \cdot 2, \ldots, k \cdot \Delta t) \rightarrow \mathbb{R}$, where $\Delta t$ is the simulation time step, $k$ is the number of observed simulation steps, and $\mathbb{R}$ is the signal range. The signal range $\mathbb{R}$ is bounded by its min and max values denoted $min_{\mathbb{R}}$ and $max_{\mathbb{R}}$, 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, $\Delta t$ 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 $(sg_{i_1}, \ldots, sg_{i_m})$ of signals for the input variables $i_1$ to $i_m$ 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 $sg_i$, and the output signal $sg_o$ share the same simulation time interval and simulation time steps, i.e., the values of $\Delta t$, $T$, 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 $(sg_{i_1}, \ldots, sg_{i_m})$ of signals. Each signal $sg_{i_p}$ 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. $\forall t: k_j \leq t < k_{j+1} \Rightarrow sg(t \cdot \Delta t) = v_j \wedge sg(k_{j+1} \cdot \Delta t) = v_{j+1}$. 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 $k^\prime$ is specified by $\{(0, v_0), (k^\prime, v_1)\}$ (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 $dist(sg_o, sg'_o)$.

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 $sg'_o$, $dist(sg_o, sg'_o) =$ 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 $\langle n, v \rangle$. 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 $\langle n, v \rangle$ feature can be instantiated as the one-step constant-value $\langle v \rangle$ and always constant-value $\langle v \rangle$ 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., $max_{\mathbb{R}}$ and $min_{\mathbb{R}}$).

The derivative feature is extended into sign-derivative and extreme-derivative features. The sign-derivative feature is parameterized by $\langle s, n \rangle$ 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
point E which is a strict local optimum point as well.

We define a function $F_j$ for each (non-abstract) feature $f$ in Figure 2(a). We refer to $F_j$ as feature function. The output of function $F_j$ when given signal $sg$ as input is a value that quantifies the similarity between shapes of $sg$ and $f$. More specifically, $F_j$ determines whether any part of $sg$ is similar to feature $f$. For example, suppose functions $\text{ld}(sg, i)$ and $\text{rs}(sg, i)$ 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 $F_j$ for the feature in Figure 2(d) as follows:

$$F_j(sg) = \max_{k=1}^{\text{dis}+1} (|\text{sign-derivative}(sg, i)| \times |\text{local\ Opt}(sg, i)|)$$

and

$$\text{local\ Opt}(sg, i) = \begin{cases} 
1 & |\text{sign-derivative}(sg, i)| \neq |\text{const-derivative}(sg, i)| \\
0 & \text{Otherwise}
\end{cases}$$

Function $F_j(sg)$ 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 $F_j(sg)$, 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 $sg_0$ and $sg_i$. Let $f_1, \ldots, f_m$ be $m$ features that we choose to include in our diversity measure. We compute feature vectors $F^1(sg_0) = (F_{f_1}(sg_0), \ldots, F_{f_m}(sg_0))$ and $F^1(sg_i) = (F_{f_1}(sg_i), \ldots, F_{f_m}(sg_i))$ corresponding to signals $sg_0$ and $sg_i$, 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 $F^v(sg_0)$ and $F^v(sg_i)$ corresponding to signals $sg_0$ and $sg_i$, we compute the Euclidean distance between these two vectors, i.e., $\text{dist}(F^v(sg_0), F^v(sg_i))$, as the measure of feature-based diversity between signals $sg_0$ and $sg_i$. 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 $TS = \{t_1, \ldots, t_n\}$ for a given model $M = (\Sigma, \alpha)$ to diversify the set of output signals generated by $TS$. We denote by $TSD = \{sg_1, \ldots, sg_q\}$ the set of output signals generated by $TS$. We capture the degree of diversity among output signals in $TSD$ using objective functions $O_o$ and $O_f$ that correspond to vector-based and feature-based notions of diversity, respectively:

$$O_o(TSD) = \sum_{i=1}^{2q} \text{MIN}_{\{sg_j \in TSD \}} \text{dist}(sg_i, sg_j)$$

$$O_f(TSD) = \sum_{i=1}^{2q} \text{MIN}_{\{sg_j \in TSD \}} \text{dist}(F^v(sg_i), F^v(sg_j))$$

Function $O_o$ computes the sum of the minimum distances of each output signal vector $sg_i$ from the other output signal vectors in $TSD$. 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 $TSD$. Our test generation algorithm aims to maximize functions $O_o$ 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 $O_o$ 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 $TSD$ that are obtained from test suites. The objective function can be either $O_o$ or $O_f$, 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 $P_{\text{max}}$. 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. $P ←$ initial number of signal segments for test inputs
2. $TS ← \text{GENERATEINITIALTESTSUITE}(P)$
3. BestFound $← O(TSO)$
4. $P_{max} ←$ maximum number of signal segments permitted in test inputs
5. $TSO ←$ signal outputs obtained by simulating $M$ for every test input in $TS$
6. $\text{whole-test-suite-coverage} ←$ coverage achieved by $TS$ over $M$
7. $\text{initial-coverage} ←$ whole-test-suite-coverage
8. $\text{accumulative-coverage} ←$ initial-coverage
9. Let $\sigma ← \alpha$-exploration and $\tau ← \tau$-exploitation be the max and min tweak parameters, respectively.
10. $\sigma ← \sigma$-exploration $\tau$-tweak parameter $\tau$
11. repeat
12. $\text{newTS} ← \text{TWEAK}(TS, \sigma, P)$ /* generating new candidate solution */
13. $TSO ←$ signal outputs obtained by simulating $M$ for every test input in $\text{newTS}$
14. $\text{whole-test-suite-coverage} ←$ coverage achieved by $\text{newTS}$ over $M$
15. $\text{accumulative-coverage} ← \text{accumulative-coverage} + \text{whole-test-suite-coverage}$
16. if $O(TSO) > \text{highestFound}$
17. $\text{highestFound} ← O(TSO)$
18. $\text{TS} ← \text{newTS}$
19. if accumulative-coverage has reached a plateau at a value less than $\%100$ and $P < \text{P}_{max}$
20. $P ← P + 1$
21. Reduce $\sigma$ proportionally as accumulative-coverage increases over initial-coverage
22. until maximum resources spent
23. return $\text{TS}$

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 $\%100$. $k$ 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 $s_g \in TS$ denoted by $\{k_1, \nu_1), \ldots, (k_p, \nu_p)\}$ as follows: We add values $x_i$ (respectively $y_i$) to every $\nu_i$ (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 $\sigma \times (\max_R - \min_R)$ (respectively $\sigma \times k$), where $R$ is the signal range and $k$ is the number of simulation steps.

We control the degree of exploration and exploitation of our search using $\sigma$. 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 exploitative search (i.e., $\sigma = 0.5$). However, if we let the search remain exploitative, it may reduce to a random search. Hence, we reduce $\sigma$ iteratively in OD such that the amount of reduction in $\sigma$ 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 $s_g$.

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 a purely exploitative search (i.e., $\sigma$-exploration) or a random search. Hence, we reduce $\sigma$-exploration $\tau$-tweak parameter $\tau$.

RQ2 (Comparing $O_1$ and $O_f$). How does the $O_f$ diversity objective perform compared to the $O_1$ diversity objective? We compare the ability of the test suites generated by OD with $O_1$ 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.

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
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 \( O = \{ s_1, \ldots, s_q \} \) be the set of output signals obtained by running \( M \) for the test inputs in TS, and let \( G = \{ g_1, \ldots, g_k \} \) be the corresponding ground truth oracle signals. We define our quantitative oracle (QO) as follows: \( QO(M, TS) = \max_{1 \leq i \leq q} \text{dist}(s_i, g_i) \).

We use a threshold value THR to translate the quantitative oracle QO into a boolean fault revealing measure denoted by FR. Specifically, FR returns true (i.e., \( QO(M, TS) > \text{THR} \)) 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 \( \text{THR} = 0.15 \) and \( \text{THS} = 0.25 \) 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 suite TS = \( \{ I_1, \ldots, I_{g_1} \} \) and TS’ = \( \{ I_1', \ldots, I_{g_2}' \} \) 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 \( I_i = (s_{g_1}, \ldots, s_{g_n}) \) in TS (respectively TS’), there exists some test input \( I_i' = (s'_{g_1}, \ldots, s'_{g_n}) \) in TS’ (respectively TS) such that \( s _{g_k} \) and \( s'_{g_k} \) (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 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 \( O_v \) and \( O_f \). Each distribution in Figures 4(a) to (c) contains 74 points. Each point relates to one faulty model and represents either the average quantitative 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 \( \text{THR} \) 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 \( x = \text{OD}(O_v) \) 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
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 ($\alpha$) 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 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 $P$ 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 90% 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 $\blacksquare$. 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.

<table>
<thead>
<tr>
<th>Faults</th>
<th>OD</th>
<th>SLDV</th>
<th>OD</th>
<th>SLDV</th>
<th>OD</th>
<th>SLDV</th>
<th>OD</th>
<th>SLDV</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.007</td>
<td>0.007</td>
<td>0.008</td>
<td>0.008</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
</tr>
<tr>
<td>2</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
</tr>
<tr>
<td>3</td>
<td>0.007</td>
<td>0.007</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
<td>0.006</td>
</tr>
<tr>
<td>4</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
</tr>
<tr>
<td>5</td>
<td>0.007</td>
<td>0.007</td>
<td>0.007</td>
<td>0.007</td>
<td>0.007</td>
<td>0.007</td>
<td>0.007</td>
<td>0.007</td>
</tr>
<tr>
<td>6</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
</tr>
<tr>
<td>7</td>
<td>0.007</td>
<td>0.007</td>
<td>0.007</td>
<td>0.007</td>
<td>0.007</td>
<td>0.007</td>
<td>0.007</td>
<td>0.007</td>
</tr>
<tr>
<td>8</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
<td>0.009</td>
</tr>
</tbody>
</table>

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 (P = 1). For all the faults, the average of QO obtained by OD (denoted by ¯Q) is higher than the value of QO obtained by SLDV. Finally, SLDV totally fails to detect faults 3, 4, 7, 8, 16, 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 timeout 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 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 cases, 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


https://sites.google.com/site/myiscresources/.


