Test Generation and Test Prioritization for Simulink Models with Dynamic Behavior

Reza Matinnejad, Shiva Nejati, Lionel C. Briand, *Fellow, IEEE,* and Thomas Bruckmann,

Abstract—All engineering disciplines are founded and rely on models, although they may differ on purposes and usages of modeling. Among the different disciplines, the engineering of Cyber Physical Systems (CPSs) particularly relies on models with dynamic behaviors (i.e., models that exhibit time-varying changes). The Simulink modeling platform greatly appeals to CPS engineers since it captures dynamic behavior models. It further provides seamless support for two indispensable engineering activities: (1) automated verification of abstract system models via *model simulation*, and (2) automated generation of system implementation via *code generation.*

We identify three main challenges in the verification and testing of Simulink models with dynamic behavior, namely *incompatibility, oracle* and *scalability* challenges. We propose a Simulink testing approach that attempts to address these challenges. Specifically, we propose a black-box test generation approach, implemented based on meta-heuristic search, that aims to maximize diversity in test output signals generated by Simulink models. We argue that in the CPS domain test oracles are likely to be manual and therefore the main cost driver of testing. In order to lower the cost of manual test oracles, we propose a test prioritization algorithm to automatically rank test cases generated by our test generation algorithm according to their likelihood to reveal a fault. Engineers can then select, according to their test budget, a subset of the most highly ranked test cases. To demonstrate scalability, we evaluate our testing approach using industrial Simulink models. Our evaluation shows that our test generation and test prioritization approaches outperform baseline techniques that rely on random testing and structural coverage.

Index Terms—Simulink models, search-based software testing, test generation, test prioritization, test oracle, output diversity, signal features, structural coverage.

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 [27]. Software development using models, also referred to as Model Driven Engineering (MDE) [27], is largely focused around the idea of *models for code generation* [26] or *models for test generation* [75], [101]. 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 such as the Cyber Physical System (CPS) domain where software closely interacts with physical processes and objects, 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, we are interested in models that have *dynamic* behavior (i.e., models that exhibit time-varying changes) [47], [39], [104]. These models can be classified based on their time-base (i.e., time-discrete versus time-continuous) and based on the values of their output variables (i.e., magnitude-discrete versus magnitude-continuous). Specifically, these models might be time-continuous magnitude-continuous, time-discrete magnitude-continuous, time-continuous magnitude-discrete, and time-discrete magnitude-discrete [104], [20] (see Figure 1).

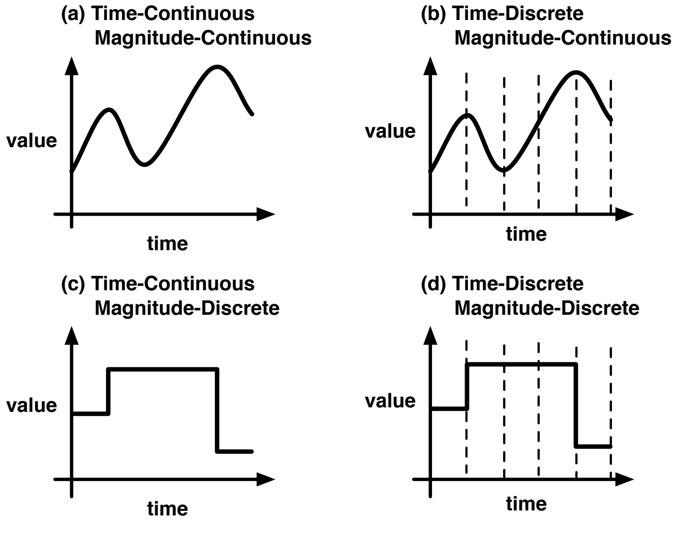


Fig. 1. Four different modeling paradigms for Cyber Physical Systems.

Models built for the purpose of simulation 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 may build on one or a combination of the four different modeling paradigms shown in Figure 1. But, most often, Simulation models include time-continuous or magnitude-continuous abstractions to be able to capture plant

- *R. Matinnejad, S. Nejati, and L. Briand are with the SnT Centre for Security, Reliability, and Trust, University of Luxembourg, Luxembourg L-2721.*
  E-mail: {reza.matinnejad,shiva.nejati,lionel.briand}@svv.lu.
- *T. Bruckmann is with Delphi Automotive Systems, Luxembourg. E-mail: thomas.bruckmann@delphi.com.*

models (i.e., environment) and the interactions between software systems and plant models [104], [20]. On the other hand, models built for the purpose of code generation capture software parts only and are described using time-discrete magnitude-discrete models [93], [44]. This is because the generated software code from these models receives sampled input data in terms of discrete sequences of events and has to run on platforms that support discrete computations only.

CPS development often starts with building simulation models capturing both continuous and discrete behaviors of a system [104], [20]. These models enable engineers to explore and understand the system behavior and to start system testing very early. Simulation models are then discretized by replacing continuous calculations with their corresponding discrete approximation calculations. This results in models from which software code can be automatically generated. Simulation models may, in addition, serve as test oracles (formal specifications) for testing and verification of software code.

It is important to develop effective verification and testing techniques to ensure correctness of both simulation and code generation models in the CPS domain. In our work, we focus on models developed in Matlab/Simulink/Stateflow (or Simulink for short) [96]. Simulink is an advanced platform for developing both simulation and code generation models and is prevalently used by the CPS industry. In the past years, we have been studying existing verification and testing techniques developed for Simulink models within the context of a research collaboration with Delphi Automotive. Drawing on our combined experiences and knowledge from research and practice, we have identified three key challenges concerning existing testing and verification techniques for Simulink models. We discuss these challenges below.

The Incompatibility Challenge. The existing approaches to testing and verification of Simulink models entirely focus on magnitude-discrete time-discrete models, i.e., code generation models [117], [73], [76], and are not compatible, and hence not applicable, to Simulink models with continuous behaviors (i.e., simulation models). This is because these techniques often require to translate Simulink models into an intermediate discrete behavior model to be analyzed by model checkers (e.g., DiVine [11], KLEE [18] and JavaPathFinder [42]) or by SAT/Constraint/SMT solvers (e.g., PVS [69], Prover [77]). The incompatibility challenge sometimes extends to some features that are commonly used in the Simulink code generation models [82], [117]. Specifically, existing techniques have difficulties to handle library code or system functions (implemented as Matlab S-Functions). For example, Simulink Design Verifier (SLDV) [97], a commercial Simulink testing tool that is a product of Mathworks and a Simulink toolbox, can handle only some restricted forms of S-Functions. Finally, due to limitations of existing constraint/SAT/SMT solvers [46], techniques that rely on these solvers to verify or test Simulink [97], [35], [8], [40], [24] often fall short when the underlying model contains floating point and non-linear math operators (e.g., square root or trigonometry functions).

The Oracle Challenge. The second challenge mostly has to do with unrealistic assumptions about test oracles for Simulink models (both simulation and code generation ones) in practical settings. Several existing techniques rely on automatable test oracles described as assertions (specified test oracles [12], [60]) or runtime errors (implicit test oracles [12], [60]) to identify faults in Simulink models [66], [67]. However, formal specifications from which assertions can be derived are expensive and are often not available in practice. Runtime errors such as integer over-/underflows are not sufficient as many faults may not lead to runtime crashes. Even in the presence of formal requirements and runtime errors, engineers tend to inspect system outputs manually to identify unforeseen failures. As a result, test oracles for Simulink models are to a great extent manual.

In the absence of automatable test oracles, existing approaches seek to reduce the manual oracle cost by generating *small* test suites that achieve high structural coverage [98], [99]. Such test suites are able to execute most of the source code or the model under test, suggesting that the code or the model is unlikely to contain undetected bugs. Further, when test suites are small, their outputs can be inspected manually without requiring a lot of effort. However, several studies demonstrate that structural coverage criteria alone may not be effective at finding faults in software models and programs [41], [91], [65], [32].

A further limitation is that test oracles in the literature are largely focused on verifying discrete system properties (e.g., invariants or reachability). Several important CPS requirements concern continuous dynamic aspects [16], [37], [73]. For example, these requirements may constrain the time it takes for a controlled variable to stabilize sufficiently close to a reference value (set-point), or they may constrain the frequency and the amount of changes of a controlled variable over a continuous period of time. Note that these requirements concern both simulation and code generation models. There is little work on verifying or testing Simulink models against CPS continuous dynamics requirements [16], [37], [73], [76].

The Scalability Challenge. There is almost no study that demonstrates scalability of existing testing and verification Simulink tools to large industrial models. Even commercial tools such as SLDV do not scale well to large and complex models, an issue explicitly recognized by Mathworks [35]. Further, as models grow larger and become more complicated, they are more likely to contain features or mathematical operations not supported by existing tools (the incompatibility challenge). In addition, existing tools may fail to effectively identify faults in practical settings due to their unrealistic assumptions about test oracles (the oracle challenge). Hence, scalability remains an open problem for Simulink testing and verification.

In this article, we provide automated techniques to generate effective test suites for Simulink models. Our goal is to alleviate the above three challenges. *First,* in order to deal with the incompatibility challenge, we address both continuous and discrete behaviors in Simulink models by generating test inputs as *signals*, i.e., functions over time, in an entirely black-box manner. Our strategy attempts to maximize chances to find unacceptable worst-case behavior by building on a combination of a single-state search optimizer [52] and the whole test suite generation approach [29], [28].

Second, instead of focusing on structural coverage alone as done in most existing approaches, we propose a test generation approach that aims to maximize diversity in output signals of Simulink models. Our intuition is that by diversifying test output signals we are more likely to find cases where there are large discrepancies between expected and actual signals, thus making it more likely for engineers to detect failures. We introduce a new notion of diversity for output signals that is defined based on a set of representative and discriminating signal feature shapes. We show how this notion guides our heuristic search-based test generation algorithm to generate test suites with diversified output signals.

We propose a test prioritization algorithm to automatically rank test cases generated by our test generation algorithm according to their likelihood to reveal a fault. Engineers can then select, based on their time constraints, a subset of the most highly ranked test cases. This is expected to lead to more failure detections within time and resource constraints. Existing test prioritization techniques mainly rely on dynamic test coverage information to prioritize test cases [122], [115]. As a result, test cases that achieve higher structural coverage are likely to be prioritized higher. In our work, to rank test cases, we use a combination of test coverage and fault-revealing probabilities of test cases. Specifically, we use the degree of output diversity of a test suite as a proxy for the fault-revealing probabilities of test cases in that test suite. We note that a number of recent studies performed in different contexts have shown that test suites generating diverse outputs are more effective in fault finding [38], [2], [3].

Third, we evaluate our test generation and our test prioritization algorithms using two industrial Simulink models. We assess the effectiveness of these algorithms and systematically compare them with baseline techniques that rely on random testing and the decision coverage criterion.

Contributions. This article extends a conference paper [58] and a tool paper [59] both published at the 38th International Conference on Software Engineering (ICSE'16). In this article, we present a consolidated Simulink model testing approach by putting all our existing findings together in a coherent form. Further, as specified below, we refine and extend ideas from our previous work and provide a number of new contributions in this article:

(1) We propose a test generation algorithm for both simulation and code generation Simulink models. Our approach does not rely on automatable test oracles and is guided by heuristics that build on a new notion of diversity for output signals. We demonstrate that our approach outperforms random baseline testing, coverage-based testing and an earlier notion of signal output diversity proposed in our previous work [54].

Contribution (1) extends our earlier work [58] as follows: (1) We provide new experimental results comparing our test generation algorithm with coverage-based testing based on the decision coverage criterion. (2) Our earlier test generation approach was applied to single-output Simulink models [58]. This can be seen as a limitation since Simulink models often contain several outputs, each of which can be tested and evaluated independently. To eliminate this limitation, we adapted and refined the formal notations and concepts to deal with multiple outputs in Simulink models. This extension significantly increased the amount of data we had to gather in our experiments and the time it took to carry out those experiments.

(2) We propose a test prioritization algorithm that combines test coverage and test suite output diversity to rank test cases. Our algorithm generalizes the existing coverage-based test prioritization based on *total* and *additional* structural coverage [115], [122]. We show that our test prioritization algorithm outperforms random test prioritization and a state-of-the-art coverage-based test prioritization [122].

Contribution (2) is completely new.

(3) We describe our Simulink testing tool (SimCoTest) and report on three *real* faults that we were able to identify in industrial Simulink models.

Contribution (3) extends the earlier work [59] as follows: The new version of SimCoTest presented here supports test case prioritization. The discussion on the real faults identified in industrial Simulink models is new.

We have made the SimCoTest tool available online [80]. The results of our experiments are also available online [81]. We are not able to make the industrial models available due to a non-disclosure agreement.

Organization. This article is structured as follows. Section 2 presents examples of simulation and code generation models and motivates our output diversity approach by comparing it with test generation driven by structural coverage. Section 3 provides background on Simulink models and Simulink test inputs, and defines our formal notation. Sections 4 and 5 describe our test generation and our test case prioritization algorithms, respectively. Section 6 explicates test oracle assumptions in our approach. Our test generation and prioritization tool, called SimCoTest, is presented in Section 7. Sections 8 and 9 present our experiments setup and experiments results, respectively. Section 10 reports on the three real faults we identified in industrial Simulink models, and further discusses limitations of some existing Simulink testing tools when they are used to reveal these faults. Section 11 compares our work with related work. Section 12 concludes the article.

2 Motivation

In this section, we provide examples of simulation and code generation models. We then motivate our output diversity test generation approach by contrasting it with the test generation approach based on structural coverage using an illustrative example.

2.1 Simulation and code generation models

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 [95].

Simulation models. Figure 2(a) shows a very simplified simulation model for FLC adopted from the book of Zander et. al. [117] and implemented in Simulink. This model captures the behavior of a software component that receives continuous resistance signals from a fuel level sensor and computes the level of fuel in the tank. The model in Figure 2(a) exhibits time-discrete magnitude-continuous behavior. More specifically, this model receives continuous signals from sensors. However, since the model represents a piece of software, signal values should be sampled at discrete time steps and the sampled values are passed to the model in Figure 2(a). As shown in the figure, this model contains a (continuous) integral operator ($\int$) to accurately compute the fuel level. The Simulink model in Figure 2(a) is executable. Engineers can run the model for any desired input signal and inspect the

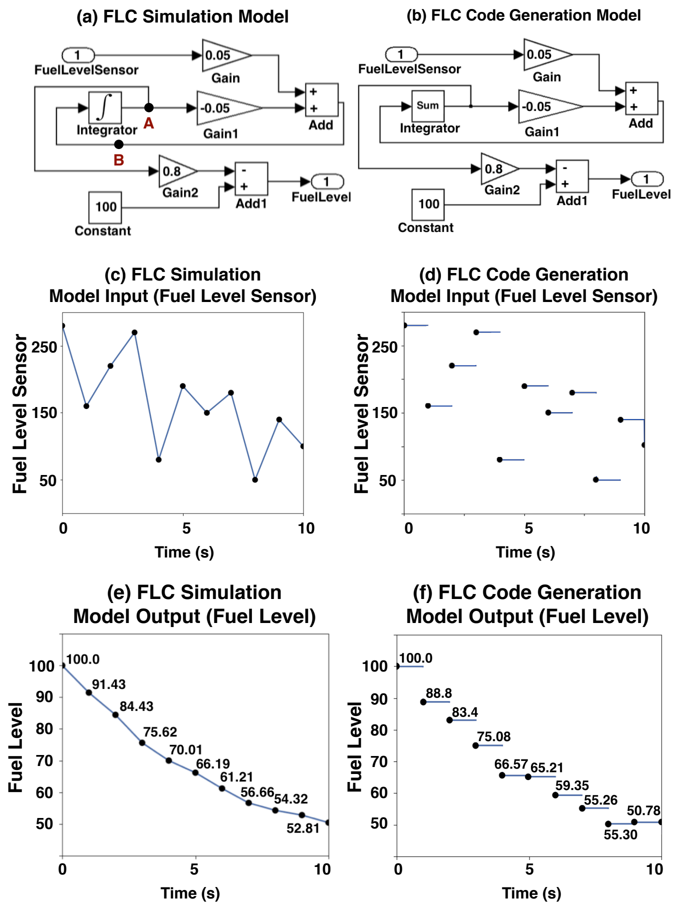


Fig. 2. A Fuel Level Controller (FLC) example: (a) A simulation model of FLC; (b) a code generation model of FLC; (c) an input to FLC simulation model; (d) an input to FLC code generation model; (e) output of (a) when given (c) as input; (f) output of (b) when given (d) as input.

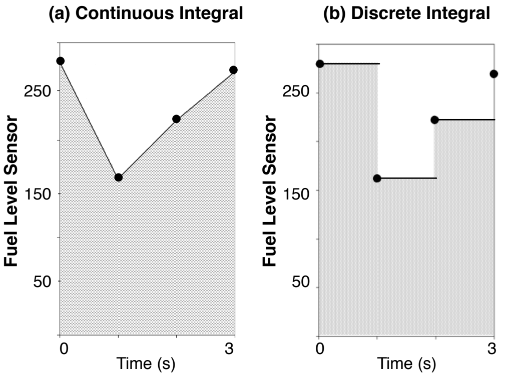


Fig. 3. Comparing outputs of (a) continuous integral $\int$ and (b) discrete integral `sum` from models in Figures 2 (a) and (b), respectively.

output. Examples of input and output signals for this model are shown in Figures 2(c) and (e), respectively. Note that both signals represent continuous functions sampled at discrete time steps. 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 2(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 time-discrete and magnitude-discrete. Further, note that the continuous integrator block ($\int$) in the simulation model is replaced by a discrete integrator (`sum`) in the code generation model. Examples of input and output signals for the code generation model are shown in Figures 2(d) and (f), respectively. Both signals represent discrete functions sampled at discrete time steps. Due to the conversion of magnitude-continuous signals to magnitude-discrete signals, the behavior of code generation models may deviate from that of simulation models. 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 and code generation model behaviors. Figure 2(c) shows a continuous input signal for the simulation model in Figure 2(a) over a 10 sec time period. Figure 2(d) shows the discrete version of the signal in Figure 2(c) that is used as input for the code generation model in Figure 2(b). Models in Figures 2(a) and (b) produce the outputs in Figures 2(e) and (f) once they are provided with the inputs in Figures 2(c) and (d), respectively. As shown in the figures, the percentages of fuel level in the continuous output signal (Figure 2(e)) differ from those in the discrete output signal (Figure 2(f)). For example, after one second, the output of the simulation model is 91.43, while that of the code generation model is 88.8. As is clear from this example, we lose precision as we move from simulation models (with continuous behavior) to code generation models (with discrete behavior). For our specific FLC example, we explain the loss of precision using the diagrams in Figure 3. The grey area in Figure 3(a) shows the value computed by the continuous integral ($\int$) used in the FLC simulation model after three seconds, while the value computed by the discretized sum operator used in the FLC code generation model corresponds to the grey area in Figure 3(b).

Conclusion. As the FLC example shows, due to discretization, simulation and code generation models of the same component are likely to exhibit different behaviors. It is important to have verification and testing techniques that are applicable to both kinds of models because (1) verifying one kind does not necessarily imply correctness of the other kind, and (2) for non-software components (e.g., physical components), only simulation models are available. In this article, we provide a testing technique that is applicable to both simulation and code generation models.

2.2 Limitations of Existing Simulink Testing Tools

A number of commercial tools are available to verify or test Simulink models. The most notable ones are SLDV and Reactis [97], [84]. These tools typically have two usage modes corresponding to two different assumptions about test oracles: (1) The first usage mode is essentially a verification activity. To verify a given Simulink model, formal properties (i.e., automatable test oracles) must be provided (e.g., in the form of assertions or runtime errors). The tools then attempt to generate test cases that can reveal violations of assertions or formal properties. Some tools such as SLDV can further generate a proof of correctness, e.g., through SMT-based model checking [35], demonstrating that given assertions or formal properties can never be violated. (2) The second usage mode assumes that automatable test oracles are not available. In this case, these tools generate test suites that achieve

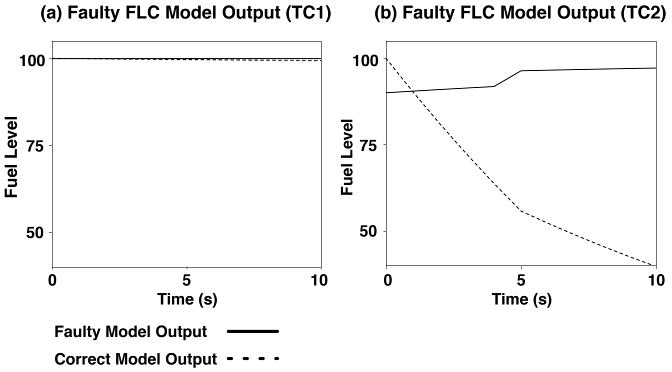


Fig. 4. (a) A test output of a faulty version of model in Figure 2(a); and (b) another test output of the same faulty model.

some notion of structural coverage (i.e., Decision, Condition, and MC/DC) [98].

In order for our approach to be widely applicable, our goal in this paper is to provide a Simulink model testing technique that does not rely on automatable test oracles. Hence, our work is comparable in objective to approaches that are guided by structural coverage i.e., the second usage mode described above. As discussed in Section 1, one main limitation of existing Simulink testing tools is that they typically have incompatibility issues with continuous blocks of Simulink, floating point and non-linear arithmetic operations and S-Functions. Focusing on the subset that is supported by existing Simulink testing tools, the main difference between our approach and existing tools lies in their underlying test generation algorithms. Typically a test generation algorithm has two main dimensions: (1) The test objective, and (2) a mechanism for test input generation. Below, we contrast our work with test generation algorithms used in Simulink testing tools along these two dimensions[1]:

- *Low effectiveness of structural coverage criteria for testing Simulink models.* Many existing Simulink testing tools (e.g. Reactis and SLDV) attempt to generate test cases that achieve high structural coverage. Recent studies show that, for Simulink models, test cases generated based on structural coverage criteria exhibit low fault finding abilities [58], [54], [32]. This is because, in Simulink models, structural coverage criteria such as MC/DC are defined on a "block level", and hence, the test cases focus on covering individual intermediary conditional blocks. However, covering conditional blocks individually may not impact the observable model outputs in a visible manner [32], [107]. In addition, effectiveness of test cases driven by structural coverage is likely to worsen further for Simulink models containing a large number of numerical computations such as lookup tables, integrator blocks, unit convertors and trigonometry and logarithmic functions. This is because faulty or wrong outputs of intermediary blocks may be masked or their magnitude may be modified by subsequent numeric computations. As a result, observable model outputs are unlikely to exhibit visible and sufficiently large deviations from their expected behaviors.
- *Lack of diversity in test inputs generated by model checking.* Many Simulink testing tools (e.g. SLDV) rely on SMT/SAT/constraint solvers to generate test inputs. As observed in recent studies and based on our experience, SMT-based model checkers tend to generate test inputs by leaving all *non-essential* inputs at some default values and only changing what is absolutely necessary [32]. In particular, in our earlier experience, we noticed that model checkers mostly change the values of the generated test input signals during the very first simulation steps, and then, the input signals remain constant for the most part and until the end of the simulation time [58]. In other words, test inputs generated by model checkers lack diversity, and many of them look almost identical. The outputs generated by similar test inputs are likely to be similar as well and may not help engineers detect faults.

To alleviate the above two limitations, in this paper, we propose a test generation approach for Simulink that (1) aims to maximize diversity among test output signals, and (2) generates test input signals in a randomized way using search algorithms.

In the remainder of this section, we use an example to contrast test generation based on structural coverage and output diversity for Simulink models. Consider a faulty version of the simulation model in Figure 2(a) where the line starting from point A is mistakenly connected to point B. We generate a test case (TC1) that achieves full structural coverage for this faulty model. Since the model in Figure 2(a) does not have any conditional behavior, a single test case can execute all the model. Figure 4(a) shows the output of TC1 along with the expected behavior where the actual output is shown by a solid line and the correct one by a dashed line. As shown in the figure, the output of TC1 is very close to the expected behavior, making it very difficult for engineers to notice any failure since, in practice, they only have a rough idea about what to expect. Further, given that in this domain small deviations from oracles are expected, engineers are unlikely to identify any fault when they use TC1.

Now suppose we use our proposed output diversity approach to generate test cases. In our work, the test suite size is not determined by structural coverage and is an input set by the engineer. Suppose we choose to generate three test cases for the given faulty model. Figure 4(b) shows the output of one of the generated test cases (TC2). As shown in the figure, the output of TC2 drastically deviates from the expected behavior, making the presence of a fault in the model quite visible to the engineer. When the goal is to achieve maximum structural coverage, TC1 and TC2 are equally desirable as they both achieve full structural coverage. But TC2 is more fault-revealing than TC1. Our approach attempts to generate a set of test cases that yield diverse output signals to increase the probability of divergence from the expected result, and hence, the chance of revealing latent faults.

3 Background and Notation

This section provides background on our test generation approach for Simulink models. We further define our formal notation in this section.

3.1 Models and Signals

Let $M = (\mathcal{I}, \mathcal{N}, \mathcal{O})$ be a Simulink/Stateflow model where $\mathcal{I} = \{i_1, \ldots, i_n\}$ is a set of input variables, $\mathcal{N} = \{n_1, \ldots, n_b\}$

[1] We note that the MathWorks license prevents publication of empirical results comparing our test generation approach with the test generation approach of SLDV. Further, we were not able to automate large experiments as our version of Reactis lacks APIs allowing such automation, hence preventing us comparing our test generation approach with that of Reactis.

is a set of nodes (i.e., Simulink blocks or Stateflow states), and $\mathcal{O} = \{o_1, \ldots, o_l\}$ is a set of output variables.

Each input/output variable of M, irrespective of M being a simulation or a code generation model, is a signal, i.e., a function of time. Assuming that the simulation time is T, we define a signal sg as a function $sg : [0..T] \rightarrow \mathcal{R}$ where $\mathcal{R}$ is the signal range. The signal range $\mathcal{R}$ is bounded by its min and max values denoted by $min_{\mathcal{R}}$ and $max_{\mathcal{R}}$, respectively.

In our test generation approach, in order to be able to generate input signals and to analyze output signals, we assume that signals are discretized based on a sampling rate (or time step) Δt. This allows us to convert a signal with a continuous domain and a continuous range into a vector of values. Note that in order to analyze signals, it is common to discretize them based on a sampling rate. At the end of this subsection, we discuss how we choose the sampling rate in our experiments. Let k be the number of time steps in the simulation time interval $[0..T]$. A discretized signal sg can be specified using the values of sg at time points 0, Δt, $2 \times \Delta t$,..., $k \times \Delta t$. We denote these values by sg^0, sg^1, sg^2, ..., sg^k, respectively.

For simulation models, every signal segment between sg^i to sg^{i+1} is a linear function, while for code-generation models, every signal segment between sg^i to sg^{i+1} is a constant function. For example, Figure 2(e) represents a signal for a simulation model, while Figure 2(f) represents a signal for a code generation model. For signals in Figures 2(e) and (f), we have $\Delta t = 1$s. The signals for code-generation models take their values from a finite (discrete) set (i.e., the signal range is finite), while the signals for simulation models take their values from an infinite (continuous) set (i.e., the signal range is infinite). For example, the range for the signal in Figure 2(e) is an interval $[50..100]$ of real numbers, while the range for the signal in Figure 2(f) is the set of fixed point values specified in the figure.

For the models used in our evaluation in Section 8, based on the guidelines provided by engineers, we set $\Delta t = 1$ms and the simulation time $T = 2$s. That is, each (discretized) signal is a vector of 2000 points. According to the Nyquist-Shannon sampling theorem [33], with a sampling rate of $1ms$, we can discretize continuous signals with a frequency of up to 500 HZ without any information loss. If signals appear to have very high frequencies ($\gg$ 500HZ), then the sampling rate may have to be much smaller to not lose any data. However, we note that, in the automotive domain, we mostly deal with input signals that are aperiodic, e.g., driver's commands, and do not have high frequencies. Further, in this domain, in contrast to the telecommunication domain for example, engineers are not typically interested in sampling rates lower than 1ms, and they consider any potential loss of data due to the 1ms sampling rate negligible.

3.2 Test Inputs and Outputs

Simulink models typically have multiple outputs. For a given test case, engineers may inspect signal values for some or all of the outputs to assess the model behavior. Our goal is to generate test cases that diversify output signals as much as possible. In our work, we focus on diversifying signal values for each output individually and independently from other model outputs. Specifically, we generate one test suite TS for each Simulink model output o such that the test cases in TS generate diverse output signals for o. In total, for a Simulink model with l outputs, we generate l test suites TS_1 to TS_l such that each test suite TS_i focuses on diversifying output signals for o_i. In our work, we consider the size of test suites TS_1 to TS_l to be the same and be equal to q.

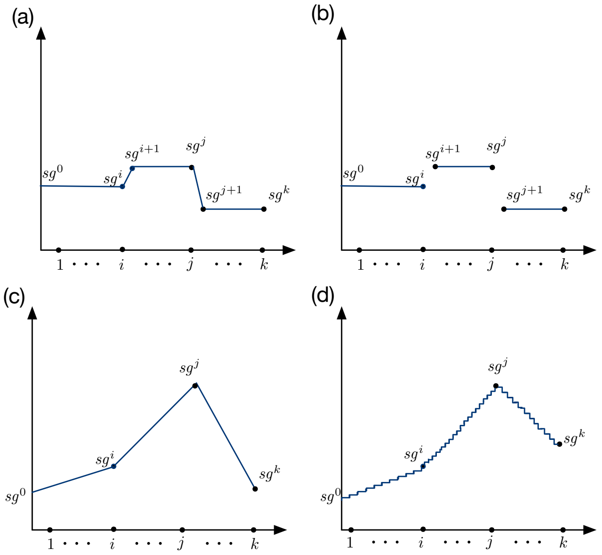


Fig. 5. Different patterns for input signals: (a) a piece-wise constant signal for simulation models; (b) a piece-wise constant signal for code-generation models; (c) a piece-wise linear signal for simulation models; and (d) a piece-wise linear signal for code-generation models. The number of pieces for all the four signal examples is three.

Each test suite TS_i contains q test inputs I_1 to I_q such that 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. To test the model behavior with respect to output o_i, engineers simulate M using each test input $I_j \in TS_i$ and inspect the signals generated for output o_i. Typically, all input and output signals generated during testing a model M 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.

To generate test inputs for Simulink models, we need to generate signals sg_{i_1} to sg_{i_n}. As discussed in Section 3.1, each signal sg_{i_j} is characterized by a set of values for $sg^0_{i_j}$, $sg^1_{i_j}$, $sg^2_{i_j}$, ..., $sg^k_{i_j}$ specifying the values of signal sg_{i_j} at time steps 0, Δt, $2 \times \Delta t$, ..., $k \times \Delta t$, respectively. Therefore, we can generate arbitrary complex input signals by generating random values for $sg^0_{i_j}$, $sg^1_{i_j}$, $sg^2_{i_j}$, ..., $sg^k_{i_j}$. However, automotive engineers typically test Simulink models using input signals with specific shapes. Further and as we will discuss in Section 6, checking the correctness of test outputs for signals with arbitrary shapes is difficult.

In our work, we consider two types of input signals: *piece-wise constant signals* and *piece-wise linear signals*. A signal specified by a sequence $sg^0, sg^1, \ldots, sg^k$ is piece-wise constant (linear respectively) if it can be partitioned into a sequence of constant (linear respectively) signals. Figure 5 illustrates *piece-wise constant signals* and *piece-wise linear signals* for simulation and code generation models. The four signals shown in Figure 5 consist of three pieces each.

Generally speaking, input signals with fewer pieces are easier to generate but they may fail to cover a large part of the underlying Simulink model. By increasing the number of pieces in input signals, structural coverage may increase, but the outputs generated by such test inputs become more complex, and engineers may find it difficult to predict expected outputs (test oracles). In our test generation algorithm discussed in Section 4.3, we ensure that, for each input variable, the generated input signals achieve high structural coverage while the number of pieces in each signal remains lower than a limit provided by domain experts.

Abbas et. al. [1] provide a detailed and formal characterization for most commonly used input signals for control systems. Their characterization includes the piece-wise constant and piece-wise linear signals exemplified in Figure 5 as well as spline and sine-shaped input signals. Our approach can be easily extended to spline and sine-shaped input signals using the characterization provided by Abbas et. al. [1].

Finally, we note that as we will discuss in Section 6, for our case study models, we generate piece-wise constant input signals for code generation models (i.e., signals similar to the one in Figure 5(b)). This is because our case study models are all code-generation models. Further, according to our domain experts, due to difficulties of predicting expected output signals (test oracles), engineers typically use piece-wise constants signals to test their models. We intend to consider simulation models and more complex input signals such as piece-wise linear signals in our future experiments.

4 Test Generation Algorithms

We propose a search-based test generation algorithm, following the whole test suite strategy [29], for Simulink models. We define two notions of diversity among output signals: *vector-based* and *feature-based*. We first introduce our two notions of output diversity and will then describe our test generation algorithm. In this section, we focus on generating a test suite for a single output of M. For a model with multiple outputs, we apply our test generation algorithm to each output of the model separately to generate a test suite for each model output.

4.1 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 [54], we defined the *vector-based* diversity measure between sg_o and sg'_o as the normalized Euclidean distance between these two signals. We define the vector-based diversity between sg_o and sg'_o as follows:

$$\widehat{dist}(sg_o, sg'_o) = \frac{\sqrt{\sum\limits_{i=0}^{k} (sg_o(i \cdot \Delta t) - sg'_o(i \cdot \Delta t))^2}}{\sqrt{k+1} \times (max_{\mathcal{R}} - min_{\mathcal{R}})} \qquad (1)$$

where $min_{\mathcal{R}}$ and $max_{\mathcal{R}}$ are the min and max values of the range of signals sg_o and sg'_o. Note that sg_o and sg'_o are both generated for output o, and hence, they have the same range. It is easy to see that $\widehat{dist}(sg_o, sg'_o)$ is always between 0 and 1.

Our vector-based notion, however, may have a drawback. 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, $\widehat{dist}(sg_o, sg'_o)$ 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.

4.2 Feature-based Output Diversity

In machine learning, a feature is an individual measurable and non-redundant property of a phenomenon being observed [113]. 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 6(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 [72]. The classification in Figure 6(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 sign-second-derivative parameterized by (s, n) where s is the sign of the second derivative, and n is the number of consecutive steps during which the sign of the second derivative remains s. The sign s can be zero, positive or negative, resulting in “derivative-constant(n)”, “derivative-increasing(n)”, and “derivative-decreasing(n)” features, respectively. We set n to k to instantiate these features to “always derivative-constant”, “always derivative-increasing”, and “always derivative-decreasing” features, respectively. Note that the second derivative is undefined over a signal with one time-step length and, hence, $n = 1$ does not yield a signal feature.

Figures 6(b) to (f) respectively illustrate the “instant-value(v)”, the “increasing(n)”, the “one-sided discontinuity with local optimum”, the “discontinuity with strict local optimum”, and the “derivative-decreasing(n)” features. Specifically, the signal in Figure 6(b) takes value v at point A. The signal in Figure 6(c) is

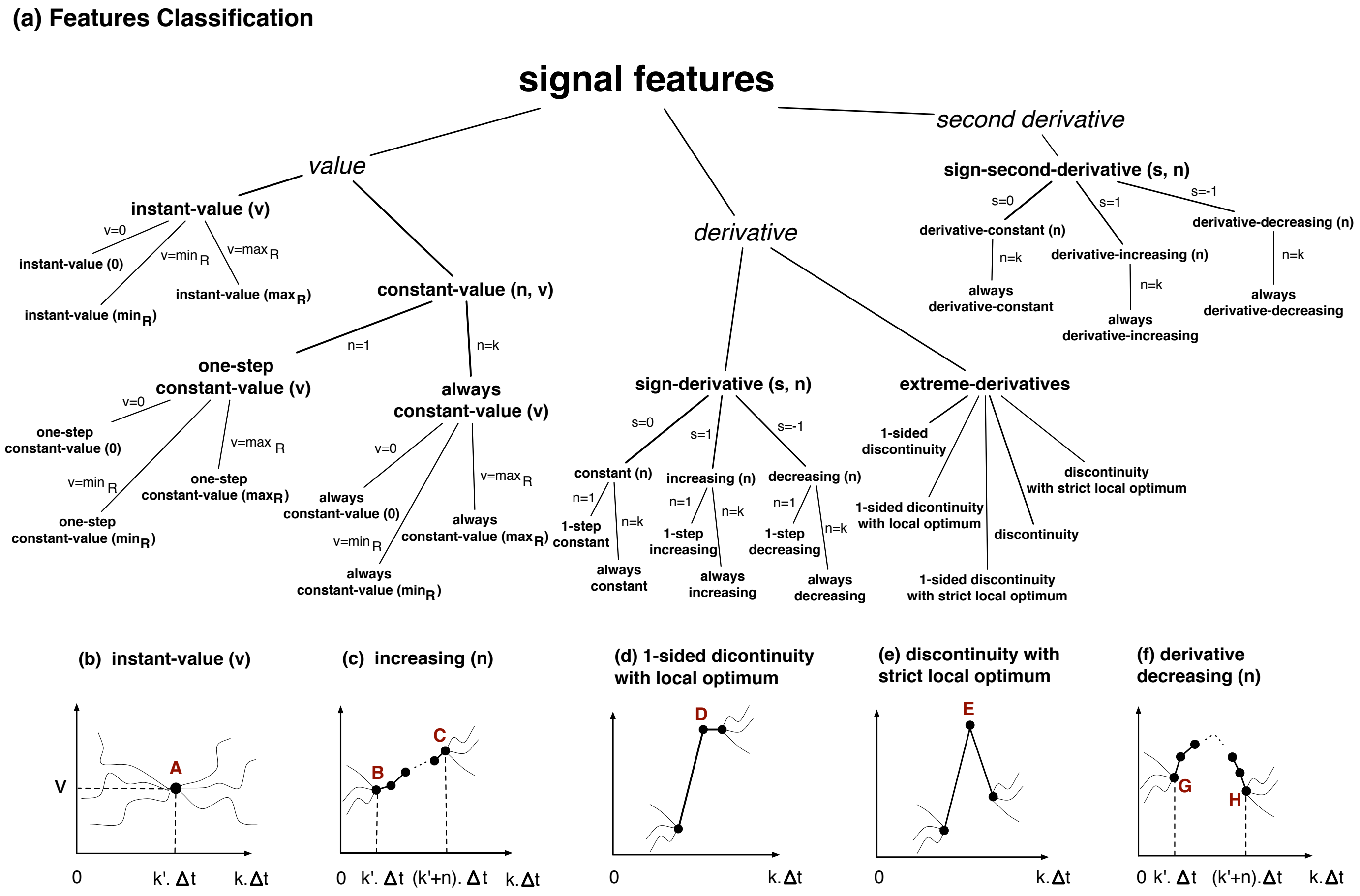

(a) Features Classification
signal features
value
derivative
second derivative
instant-value (v)
v=0
instant-value (0)
v=min R
instant-value (min R)
v=max R
instant-value (max R)
constant-value (n, v)
n=1
one-step constant-value (v)
v=0
one-step constant-value (0)
v=min R
one-step constant-value (min R)
v=max R
one-step constant-value (max R)
n=k
always constant-value (v)
v=0
always constant-value (0)
v=min R
always constant-value (min R)
v=max R
always constant-value (max R)
sign-derivative (s, n)
s=0
constant (n)
n=1
1-step constant
n=k
always constant
s=1
increasing (n)
n=1
1-step increasing
n=k
always increasing
s=-1
decreasing (n)
n=1
1-step decreasing
n=k
always decreasing
extreme-derivatives
1-sided discontinuity
1-sided dicontinuity with local optimum
1-sided discontinuity with strict local optimum
discontinuity
discontinuity with strict local optimum
sign-second-derivative (s, n)
s=0
derivative-constant (n)
n=k
always derivative-constant
s=1
derivative-increasing (n)
n=k
always derivative-increasing
s=-1
derivative-decreasing (n)
n=k
always derivative-decreasing
(b) instant-value (v)
V
A
0
k'. Δt
k.Δt
(c) increasing (n)
B
C
0
k'. Δt
(k'+n). Δt
k.Δt
(d) 1-sided dicontinuity with local optimum
D
0
k.Δt
(e) discontinuity with strict local optimum
E
0
k.Δt
(f) derivative decreasing (n)
G
H
0
k'. Δt
(k'+n).Δt
k.Δt

Fig. 6. Signal Features: (a) Our signal feature classification, and (b)–(f) Examples of signal features from the classification in (a).

increasing for n steps from B to C. The signal in Figure 6(d) is right-continuous but discontinuous from left at point D. Further, the signal value at D is more than the values at its adjacent point, hence making D a local optimum. The signal in Figure 6(e) is discontinuous from both left and right at point E. It is also decreasing on one side of E and increasing on the other side, making E a strict local optimum. Finally, the derivative of the signal in Figure 6(f) is decreasing, i.e., the second derivative is negative, for n steps from G to H.

We define a function F_f for each (non-abstract) feature f in Figure 6(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.

We provide two feature function examples related to the signal features in Figures 6(b) and (c). Specifically, the feature function F_{f_b} related to the signal feature "instant-value(v)" in Figure 6(b) is defined as follows:

$$F_{f_b}(sg, v) = \min_{i=0}^{k} |sg(i \cdot \Delta t) - v|$$

This function computes the minimum difference between a given value v and the values of signal sg at every simulation step. The lower F_{f_b}, the closer the shape of sg to the feature in Figure 6(b). Particularly, if F_{f_b} becomes zero for some v, it implies that signal sg exhibits the feature instant-value(v).

As another example, the feature function F_{f_c} related to the signal feature "increasing(n)" in Figure 6(c) is defined as follows:

$$F_{f_c}(sg, n) = \max_{i=n}^{k}\Big(\sum_{j=i-n+1}^{i} \mathtt{lds}(sg, i)\Big)$$

where $\mathtt{lds}(sg, i)$ denotes the sign of the left derivative of sg at step i. Specifically, $\mathtt{lds}(sg, i)$ is zero when sg is constant at step i when compared with its left point at step $i-1$, one when its value at i is more than its value at $i-1$, and -1 when its value at i is less than its value at $i-1$. Function F_{f_c} computes the largest sum of the left derivative signs of sg over any segment of sg consisting of n consecutive simulation steps. The higher the value of F_{f_c}, the more likely that sg exhibits the increasing(n) feature (i.e., the more likely that sg contains a segment of size n during which its values are increasing). The formal definitions for all the features in Figure 6 are available online [81].

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 [113]. 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 normalized 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. In the next section, we discuss how our diversity notions are used to generate test suites for Simulink models.

4.3 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 to diversify the set of output signals generated by TS for a specific output of M. As discussed in Section 3.2, we generate a separate test suite containing q test inputs for each output of M. We will then apply our test prioritization algorithm (see Section 5) to generate a ranking of all the generated test inputs to help engineers identify faults by inspecting a small number of test outputs.

We denote by $TSO = \{sg_1, \ldots, sg_q\}$ the set of output signals generated by TS for an output o of M. 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:

$$O_v(TSO) = \frac{\sum_{i=1}^{q} \text{MIN}_{\forall sg \in TSO \setminus \{sg_i\}} \hat{dist}(sg_i, sg)}{q} \tag{2}$$

$$O_f(TSO) = \frac{\sum_{i=1}^{q} \text{MIN}_{\forall sg \in TSO \setminus \{sg_i\}} \hat{dist}(F^v(sg_i), F^v(sg))}{q} \tag{3}$$

Function O_v computes the average of the minimum distances of each output signal vector sg_i from the other output signal vectors in TSO. Similarly, O_f computes the average 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 [29] 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 [52]. 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 compute objective functions for several new or modified members of the population at each iteration.

Figure 7 shows our output diversity test generation algorithm for Simulink models. We refer to it as OD. The core of OD is based on an adaptation of the *Simulated Annealing* search algorithm [52]. Specifically, the algorithm generates an initial solution (lines 2-3), iteratively tweaks this solution (line 11), and selects a new solution whenever its objective function is higher than the current best solution (lines 15-17). 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.

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

```
1.  $P \leftarrow 1$
2.  $TS \leftarrow$ GenerateInitialTestSuite$(q, P)$  /*Test suite size $q$ */
3.  $TSO \leftarrow$ signals obtained for output $o$ by simulating $M$ for every test input in $TS$
4.  $BestFound \leftarrow O(TSO)$
5.  $P_{max} \leftarrow$ maximum number of signal pieces permitted in test inputs
6.  whole-test-suite-coverage $\leftarrow$ coverage achieved by test cases in $TS$ over $M$
7.  initial-coverage $\leftarrow$ whole-test-suite-coverage
8.  accumulative-coverage $\leftarrow$ initial-coverage
9.  $\sigma \leftarrow \sigma$-exploration  /*Tweak parameter $\sigma \in [\sigma\text{-exploitation} \ldots \sigma\text{-exploration}]$ */
10. repeat
11.   newTS = Tweak(TS, $\sigma$, P)   /* generating new candidate solution */
12.   TSO $\leftarrow$ signals obtained for output $o$ by simulating $M$ for every test input in newTS
13.   whole-test-suite-coverage $\leftarrow$ coverage achieved by test cases in newTS over $M$
14.   accumulative-coverage $\leftarrow$ accumulative-coverage $+$ whole-test-suite-coverage
15.   if $O(TSO) > highestFound$ :
16.     $highestFound = O(TSO)$
17.     $TS = newTS$
18.   if accumulative-coverage has reached a plateau at a value less than $\%100$ :
19.     if $P < P_{max}$ :
20.       $P = P + 1$
21.   Reduce $\sigma$ proportionally from $\sigma$-exploration to $\sigma$-exploitation as
        accumulative-coverage increases over initial-coverage
22. until maximum resources spent
23. return $TS$
```

Fig. 7. Our output diversity (OD) test generation algorithm for Simulink models.

Like the simulated annealing search algorithm, our OD algorithm in Figure 7 is more explorative at the beginning and becomes more exploitative as the search progresses. In the simulated annealing search, the degree of exploration/exploitation is adjusted using a parameter called temperature. Typically, the temperature is set to a high value at the beginning of the search, making the search behaves similarly to a random explorative search. As time passes, the temperature is lowered, eventually to zero, turning the search into an exploitative search algorithm such as Hill Climbing [52]. We take a similar approach in our OD algorithm where the parameter σ acts like the temperature parameter in simulated annealing. The difference is that the value of σ in our algorithm is adjusted based on the accumulative structural coverage achieved by all the generated test suites.

The reason that we opt for such search solution is that, based on our existing experience of applying search algorithms to continuous controllers [56], a purely explorative or a purely exploitative search strategy is unlikely to lead to desirable optimal solutions. Given that the search space of input signals is very large, if we start by a purely exploitative search (e.g., $\sigma = 0.01$), our result will be biased by the initial randomly selected solution. To reduce this bias, we start by performing a more explorative search (e.g., $\sigma = 0.5$). However, if we let the search remain explorative, it may not converge fast enough to desired solutions. 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).

While being a Simulating Annealing search in essence, OD proposes two novel adaptations: (1) *Our input signal generation mechanism.* Our algorithm initially generates input signals that contain a small initial number of signal pieces P (e.g., one piece). It then increases P as needed while ensuring that P always

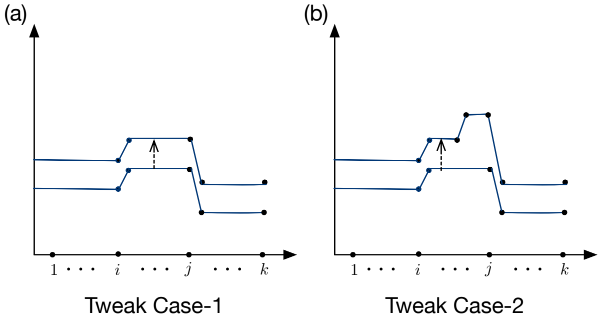


Fig. 8. Illustrating our tweak operator (line 11 of the algorithm in Figure 7) on an example constant piecewise signal for simulation models from Figure 5(a): (a) shifting the signal based on a randomly selected value (Case-1), and (b) shifting the signal and increasing the number of signal pieces (Case-2).

remains less than the limit provided by the domain expert P_{max}. Recall that, on one hand, increasing input signal pieces makes the output more difficult to analyze, but on the other hand, input signals with few pieces may not reach high model coverage. In OD, we initially generate test inputs with one piece (lines 1-2). We increase P only when the accumulative structural coverage achieved by the existing generated test suites reaches a plateau at a value less than $\%100$. In other words, we increase P only when we are not able to improve structural coverage using the current test input signals that have P pieces (lines 19-20). After increasing P on line 20, the tweak operator on line 11 increases the number of pieces in the newly generated signals. 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) *Our tweak operator for input signals.* In Figure 8, we illustrate our tweak operator (line 11 of the algorithm in Figure 7). We distinguish two cases.

Case-1: When the number of pieces in signals generated by the tweak operator does not need to be increased: In this case, the tweak operator is similar to that used in (1+1) EA [52]. The operator simply shifts input signals by a small value selected from a normal distribution with mean $\mu = 0$ and variance $\sigma \times (max_{\mathcal{R}} - min_{\mathcal{R}})$ where $\mathcal{R}$ is the range of the signal being tweaked. Our tweak operator for Case-1 is shown in Figure 8(a).

Case-2: When the number of pieces in signals generated by the tweak operator should be increased: This means that the structural coverage achieved by the current set of signals has not increased over the past few iterations (see Lines 18–20). In this case, the operator first increases the number of pieces in signals, and then similar to Case-1, the operator shifts the signals. Our tweak operator for Case-2 is shown in Figure 8(b).

To conclude this section, we discuss the asymptotic time complexity of individual iterations of the OD algorithm when we use O_v and O_f functions, respectively. Let q be the size of the generated test suites, k be the number of simulation steps, and T_M be the time it takes to simulate the underlying Simulink model for k steps. In general, T_M depends on the size of the model, the number of model inputs and outputs, and the number of simulation steps. The time complexity of one iteration of OD with O_v is $O(q \times T_M) + O(q^2 \times k)$[2].

The time complexity of one iteration of OD with O_f is $O(q \times T_M) + O(q \times m \times k) + O(q^2 \times m)$ where m is the number of signal features that we use to compute feature vectors. Note that the time complexity of computing features in Figure 6 is $O(k)$. This is mainly because in those features we consider the parameter n to be either one or k. In our problem, k is considerably larger than m and larger than q. For example, in our experiment, we have $k = 2000$, while we use 23 features ($m = 23$), and we typically choose q to be less than 10. In Section 9, we will provide the average time for model simulations (T_M) and for executing one iteration of the OD algorithm using O_v and O_f functions based on our empirical evaluation.

5 Test Prioritization Algorithm

Our OD test generation algorithm discussed in Section 4 generates a test suite (with q test cases) for each model output. To help engineers effectively inspect model behavior with respect to all the generated test cases, we provide a test prioritization technique. The goal of our prioritization algorithm is to generate a ranked list of test cases such that the most fault-revealing test cases are ranked higher in the list, helping engineers identify faults faster by inspecting a few test cases.

We take a dynamic test prioritization approach based on greedy algorithms to rank test cases. This choice is driven based on the following two main considerations: First, in our work, test prioritization occurs after the test generation step where all the test cases are already executed. Hence, test coverage information is already available. Therefore, to prioritize test cases, we do not need to resort to static techniques that, due to unavailability of test coverage information, are restricted to static analysis of code or other artifacts [78], [100]. Second, based on our experience, typical industrial Simulink models have less than 50 outputs, and in our work, we consider to generate less than 10 test cases for each output. Hence, the total number of test cases that we need to rank is relatively small (less than 500). Therefore, we chose to consider greedy-based prioritization algorithms. These algorithms iteratively compare all the test cases with one another to identify the best locally optimal choice at each iteration. Other implementation alternatives include adaptive random test prioritization and search-based test prioritization [78]. These are mainly proposed to improve efficiency by comparing only a subset (not all) of test cases or test case rankings at each iteration. Neither of these approaches, however, outperform the greedy approach in terms of the ability to find faults faster [49], [78].

Our test case prioritization algorithm is shown in Figure 9. The algorithm generates an ordered list $Rank$ of test cases in $\mathcal{TC}$ where $\mathcal{TC}$ is the union of all the generated test suites for a given Simulink model $M = (\mathcal{I}, \mathcal{N}, \mathcal{O})$. In addition to the aggregated test suite $\mathcal{TC}$ and the model M, the algorithm receives the following three functions as input and uses them to compute the test case ranking: (1) The test coverage information for each individual test case $tc \in \mathcal{TC}$, denoted by function $covers : \mathcal{TC} \rightarrow 2^{\mathcal{N}}$. (2) The fault-revealing probability of test cases in $\mathcal{TC}$, denoted by $FRP : \mathcal{TC} \rightarrow [0..1]$. (3) The faultiness probability of individual Simulink nodes of M, denoted by $faultiness : \mathcal{N} \rightarrow [0..1]$.

Our algorithm aims to reward and prioritize test cases that are likely to find more faults in models. To achieve this, it relies

[2] Note that the O here refers to the bigO time complexity and should not be mistaken by objective function O used in the OD algorithm.

on functions FRP (fault-revealing probability of test cases) and $faultiness$ (faultiness probability of individual Simulink nodes). In reality, however, we do not have any a priori knowledge about the fault-revealing ability of a test case (FRP), and we do not know the likelihood of a node being faulty ($faultiness$) at the time of test prioritization. Therefore, similar to existing approaches on test case prioritization, our reward functions can only be based on *surrogate criteria* [115]. Most test prioritization techniques primarily use test coverage as the surrogate for fault-revealing ability of test cases. Given that test coverage alone may not be a good indicator for fault-revealing ability, in our algorithm (Figure 9), we define functions FRP and $faultiness$ based on a combination of test coverage and other criteria described below.

For $faultiness$, initially we assume that the nodes are all equally probable of containing a fault. So, we initialize the faultiness probability of each node with one. This is just to ensure that all the nodes have the same *relative* faultiness probability at the beginning. The $faultiness$ probabilities are then iteratively reduced depending on the selected test cases and their FRP values. We note that our decision to initialize the $faultiness$ values by one is consistent with the test prioritization algorithm presented by Zhang et. al. [122].

We use the output diversity functions defined in Section 4 as a proxy for test case fault-revealing ability (FRP). We note that output diversity (i.e., output uniqueness) has been shown to correlate to fault finding [2], [3], [54], [58] and to act as an effective complement to test coverage [2], [3]. Recall that we defined output diversity functions over test suites generated by our test generation algorithm in Figure 7, and that $\mathcal{TC}$ in Figure 9 is the union of all these test suites. Indeed output diversity is a property of individual test suites, and not a property of test cases inside test suites. However, based on our previous results [54], [58], we know that if a test suite TS has a high output diversity, it likely contains some test cases that are effective in fault finding. Of course, we have no way of telling apart the more effective test cases in TS from the less effective ones. But since TS is typically small (less than 10 elements), by giving a prioritization boost to all test cases in TS including both effective and ineffective test cases, we are still likely to have some effective test cases to be ranked high. Hence, we assume all the test cases in TS have the same fault-revealing ability equal to the output diversity of TS. More specifically, given a test case tc such that $tc \in \mathcal{TC} \cap TS$, we set $FRP(tc)$ to be equal to $O(TS)$ where O can be either the vector-based O_v or the feature-based O_f output diversity functions described in Equations 2 and 3, respectively.

In the remainder of this section, we first describe how the test coverage function, $covers$, used in our algorithm is computed for Simulink models. We then describe how our proposed prioritization algorithm works. Recall from Section 4 that each test suite TS generated by the OD algorithm is related to a specific output o of the underlying Simulink model. Let $tc \in TS$ be a test case generated for an output o. We write $test(tc, o)$ to denote that test case tc is related to output o. Note that each test case is related to exactly one output, but an output is related to a number of test cases (i.e., q test cases). For Simulink models, test coverage is the set of Simulink nodes (i.e., Simulink blocks or Stateflow states) executed by a given test case tc to generate results for the output o related to tc. Given a Simulink model $M = (\mathcal{I}, \mathcal{N}, \mathcal{O})$ and a test case $tc \in \mathcal{TC}$, we denote the test coverage of tc by $covers(tc)$ and define it as follows:

$$covers(tc) = \{n \mid n \in static\_slice(o) \wedge test(tc, o) \wedge tc \text{ executes } n\}$$

Algorithm. Test case prioritization algorithm

```
Input:  – $M = (\mathcal{I}, \mathcal{N}, \mathcal{O})$: Simulink Model
        – $\mathcal{TC}$: A test suite for $M$
        – $covers : \mathcal{TC} \to 2^{\mathcal{N}}$: Test coverage of test cases
        – $FRP : \mathcal{TC} \to [0..1]$: Fault-revealing probability of test cases
        – $faultiness : \mathcal{N} \to [0..1]$: Simulink node faultiness probabilities
Output: – $Rank$: A ranked list of the test cases in $\mathcal{TC}$

1.  $Rank = []$
2.  $Ranked = 0$
3.  while ($\mathcal{TC} \neq \emptyset$) do
        /* Lines 4-5: For each test case $tc$, compute the summation of the probabilities that $tc$
         can find a fault in a Simulink node that it covers.*/
4.      for ($tc \in \mathcal{TC}$) do
5.          $P(tc) = FRP(tc) \times \sum_{n \in covers(tc)} faultiness(n)$
        /* Select the test case $\mathbf{tc}$ that yields the highest aggregated fault-revealing probabilities*/
        and add it to $Rank$*/
6.      Let $\mathbf{tc} \in \mathcal{TC}$ yield the largest $P(tc)$
7.      $Rank[Ranked] = \mathbf{tc}$
        /* Lines 8-10: Update the faultiness probability of Simulink nodes covered by $\mathbf{tc}$
        for the remaining unranked test cases*/
8.      for ($n \in covers(\mathbf{tc})$) do
9.          $old = faultiness(n)$
10.         $faultiness(n) = old \times (1 - FRP(\mathbf{tc}))$
11.     $\mathcal{TC} = \mathcal{TC} \setminus \{\mathbf{tc}\}$
12.     $Ranked + +$
13. return $Rank$
```

Fig. 9. Our test prioritization algorithm for Simulink models.

where $o \in \mathcal{O}$ and $static\_slice(o)$ is the *static backward slice* of output o and is equal to the set of all nodes in $\mathcal{N}$ that can reach output o via data or control dependencies.

Note that our notion of test coverage is specific to a model output. The set $covers(tc)$ includes only those nodes that are executed by tc, and further, appear in the static backward slice of the output related to tc. The nodes that cannot reach that output (via Simulink control or data dependency links) are not included in $covers(tc)$ even if they happen to be executed by tc. Our notion of test coverage is the same as the notion of test execution slices defined in our previous work on fault localization of Simulink models [51]. There, we provided a detailed discussion on how the sets $static\_slice(o)$ and $covers(tc)$ can be computed for Simulink models. Therefore, we do not discuss the implementation details of these concepts for Simulink models in this article.

As discussed earlier, the algorithm in Figure 9 takes a greedy approach to rank test cases. At each iteration, it identifies the test case that yields the highest aggregated fault-revealing ability among the unprioritized test cases and adds it to the top of the ranked list $Rank$ (lines 4–7). In particular, the algorithm first computes the aggregated fault-revealing probabilities for every unprioritized test case tc by multiplying the fault-revealing probability of tc and the summation of faultiness probabilities of the nodes that are covered by tc. Note that the fault-revealing probability of a test case and the faultiness probability of a node are independent, and their cross product indicates the probability that a test case reveals a fault in a node. The test case that yields the highest aggregated fault-revealing probability is added to the ranked list $Rank$ as the best locally optimal choice (line 7). After that, the algorithm updates the faultiness probabilities of the nodes covered by the test case that was just added to $Rank$ (lines 8–10). Specifically, the faultiness probabilities of each of the nodes covered by that test case is multiplied by $(1 - FRP)$, i.e., the probability that the test case fails to reveal a fault. The algorithm terminates when all the test cases in $\mathcal{TC}$ are ranked.

Our proposed test prioritization algorithm (Figure 9) generalizes and extends the existing dynamic test prioritization techniques [122], [115], [78]. These techniques rank test cases using either *total* or *additional* structural coverages achieved by individual test cases. Specifically, in the case of total coverage, a test case is ranked higher if it yields higher structural coverage independently from other test cases. However, in the case of additional coverage, a test case is ranked higher if it produces larger additional structural coverage compared to the accumulative structural coverage achieved by the already ranked test cases. Our algorithm in Figure 9 turns into a test prioritization algorithm based on additional coverage if we set $FRP(tc)$ to one for every $tc \in \mathcal{TC}$. If, in addition, we remove lines 8 to 10 from our algorithm in Figure 9 (i.e., the part related to updating $faultiness$ with respect to the already ranked test cases), the result will be a test prioritization algorithm based on total coverage. In Section 9, we compare our test prioritization algorithm in Figure 9 with the test prioritization algorithms based on additional and total coverage [122].

6 Test Oracle

In our work, we make three important assumptions about test oracles: First, we assume that no automatable test oracle is available, a common situation in practice. Second, test oracles are typically inexact. In particular, during design time testing of cyber-physical systems, small deviations between test outputs and expected outputs are often tolerated and not considered failures. Third, the correctness of a test output is not only determined by evaluating discrete output values at a few discrete time instances, but the correctness also depends on the frequency and the amount of changes of output values over a period of time. Our assumptions have the following two implications on our approach that we discuss in this section.

First, since we assume that test outputs are evaluated manually, we need to provide a way to estimate the oracle cost pertaining to a test suite generated by a test generation technique. This is particularly important for comparing different test generation strategies. Specifically, test suites generated by two different strategies can be used as a basis for comparing the strategies only if the test suites have similar test oracle costs, i.e., evaluating their test outputs requires the same amount of effort. The oracle cost of a test suite depends on the following:

- The total number of outputs that are generated by that test suite *and* are required to be inspected by engineers. For example, our test generation algorithm (Figure 7) generates a test suite TS with size q to exercise a specific model output o. Let $\mathcal{TC} = \bigcup TS$ be the union of all such test suites. Assuming that the underlying model M has l outputs, the number of output signals that are generated by $\mathcal{TC}$ *and* need to be inspected is $l \times q$. Alternatively, another technique may generate a test suite TS' containing q test inputs for model M such that all the output signals generated by each test input in TS' are expected to be inspected by engineers. In this case, the number of output signals that are generated by TS' and need to be inspected is the same as that number for $\mathcal{TC}$, i.e., $l \times q$.
- The complexity of input data. Recall from Section 3 that test input signals in our approach are piece-wise. The fewer pieces the input signals have, the easier to determine whether their outputs are correct or not. In the automotive domain, constant signals are considered the least complex and the most common test inputs for Simulink models. Moving from constant input signals to linear signals or to piecewise constant signals causes the resulting output signals to become more complex, and hence, the cost of manual test oracles to increase. To ensure that test suites $TS = \{I_1, \dots I_{q_1}\}$ and $TS' = \{I'_1, \dots I'_{q_2}\}$ have the same input complexity, the input signals in TS and TS' should have the same number of pieces. That is, for every test input $I_i = (sg_1, \dots, sg_n)$ in TS (respectively TS'), there exists some test input $I_j = (sg'_1, \dots, sg'_n)$ in TS' (respectively TS) such that sg_l and sg'_l (for $1 \leq l \leq n$) have the same number of pieces.

In our experiments described in Section 8.5, we ensure that the test suites used to compare different test generation algorithms have the same test oracle costs, i.e., (1) the number of outputs generated by these test suites and are required to be inspected by engineers are the same, and (2) the signals related to their test inputs have the same number of pieces.

Second, we define a heuristic test oracle function that has these two characteristics: (1) To address the fact that test oracles are inexact, we define our test oracle function as a quantitative measure comparing test outputs and the expected results. (2) We define the test oracle function over the entire vectors of signal outputs to account for output changes over the entire simulation time interval.

Let sg_o be a test output signal. We define a (heuristic) test oracle function, denoted by $oracle$, that maps a given output signal to a value in $[0..1]$. The higher the value of $oracle(sg_o)$, the more likely the signal sg_o is to reveal a fault in the underlying Simulink model. In our work, we compute $oracle(sg_o)$ as the normalized Euclidean distance between sg_o and the *ground truth oracle signal* denoted by g. That is, $oracle(sg_o) = \hat{dist}(sg_o, g)$ (see Equation 1 for definition of $\hat{dist}$). The ground truth oracle is a conceptual oracle that always gives the "right answer" [12]. In practice, signal g is supposed to be created manually, while in our experiments, we use fault-free models to automatically produce the ground truth oracle signals (see Section 8.3).

In Section 8.3, we will use our heuristic oracle function, $oracle$, to provide a metric to measure the fault-revealing ability of test generation techniques. Our fault-revealing measure attempts to capture impacts of faults on output signal vectors over the entire simulation time interval as opposed to focusing on violation of discrete properties over model outputs.

The alternative fault revealing metric used in existing research on testing Simulink models (e.g., [121], [32]) is a binary measure assuming that correct test outputs should exactly match the reference output and otherwise, they reveal a failure. Any slight deviation without any regard to signal shapes or the deviation degree is assumed to be sufficient enough to reveal a failure. We believe a fault revealing metric should be quantitative and not binary as engineers typically do not inspect test outputs in a binary manner and tolerate small deviations.

Finally, we note that the $oracle$ function is only used as a heuristic to assess how easily engineers will be able to identify failures while analyzing output signals. Although not studied in this paper, the $oracle$ function could also be defined as a measure comparing the shapes of test output signals and the ground truth oracle signals, for example using the signal feature taxonomy in Figure 6. We leave to future work to develop a more comprehensive fault revealing measure for Simulink testing approaches that accounts for differences between both signal distances and signal shapes.

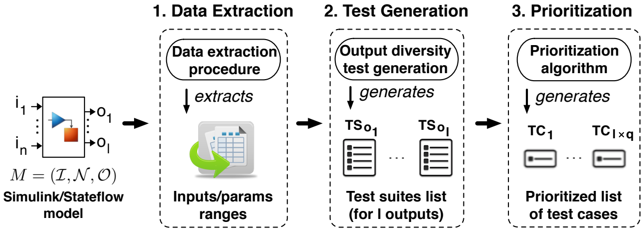

1. Data Extraction
2. Test Generation
3. Prioritization
Data extraction procedure
extracts
Inputs/params ranges
Output diversity test generation
generates
TS_{o_1} ... TS_{o_l}
Test suites list (for l outputs)
Prioritization algorithm
generates
TC_1 ... $TC_{l \times q}$
Prioritized list of test cases
i_1
i_n
o_1
o_l
$M = (\mathcal{I}, \mathcal{N}, \mathcal{O})$
Simulink/Stateflow model

Fig. 10. An overview of SimCoTest.

7 Tool Support

We have implemented our approach in a tool called Simulink Controller Tester (SimCoTest) (https://sites.google.com/site/simcotesttool/) [59]. Figure 10 shows an overview of SimCoTest. Specifically, SimCoTest takes a Simulink/Stateflow model M as input. It, then, (1) automatically extracts the information required for test generation from the model including the names, data types and data ranges of the input and output variables of the model (data extraction), (2) generates one test suite for each output of model M using our output diversity test generation algorithm in Figure 7 (test generation), and (3) prioritizes the generated test cases obtained for different model outputs based on our prioritization algorithm in Figure 9 (prioritization).

SimCoTest is implemented in Microsoft Visual Studio 2010 and Microsoft .NET 4.0. It is an object-oriented program in C# with 92 classes and roughly 25K lines of C# code. In addition, the key functions of SimCoTest, including the data extraction, test generation and test prioritization, are partly implemented using MATLAB script functions, which are called from SimCoTest using the MLApp COM interface [94]. Specifically, 64 MATLAB functions are implemented in roughly 7K lines of MATLAB script and are called from SimCoTest. SimCoTest source code is available online [81]. The main functionalities of SimCoTest have been tested with a test suite containing more than 100 test cases [81]. SimCoTest requires Matlab/Simulink to be installed and operational on the same machine to be able to execute Simulink/Stateflow models and generate test suites. We have tested SimCoTest on Windows XP and Windows 7, and with Matlab 2011b and Matlab 2015b. Matlab 2011b was selected to ensure backward compatibility of our tool with (legacy) industry models. We have made SimCoTest available to Delphi, and have presented it in a hands-on tutorial to Delphi function engineers. Finally, we note that using SimCoTest, we were able to find three real faults in Simulink models from Delphi, which had not been previously found by manual testing based on domain expertise. We discuss these faults in Section 10.

8 Experiment Setup

In this section, we present the research questions and our study subjects. We further describe metrics to measure fault-revealing ability and effectiveness of our test generation and test prioritization algorithms. Finally, we provide our experiment design.

8.1 Research Questions

RQ1 (Comparing Test Generation with State-of-the-art). *How does the fault-revealing ability of the OD test generation algorithm compare with that of a random test generation strategy or a coverage-based test generation strategy? How does the fault-revealing ability of these test generation techniques compare with their degree of structural coverage?* We investigate whether OD test generation is able to perform better than random testing, which is a baseline of comparison, and a coverage-based test generation strategy. For coverage-based test generation, we replace the objective function O in our OD algorithm in Figure 7 with an objective function that computes the accumulative dynamic test coverages of all the test cases in TS. In both comparisons, we consider the fault-revealing ability of the test suites generated by OD when used with each of the O_v and O_f objective functions. We further compare the degree of structural coverage (more specifically decision coverage) achieved by OD, coverage-based testing and random testing to investigate any relationship between the fault-revealing ability and structural coverage for these techniques.

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 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 (Comparing Test Prioritization with state-of-the-art). *How does the effectiveness of our test prioritization algorithm compare with that of a random test prioritization strategy? How does the effectiveness of our test prioritization algorithm compare with that of coverage-based test prioritization strategies?* We compare the effectiveness of our test prioritization technique with a random test prioritization algorithm (baseline) and with the state-of-the-art coverage-based test prioritization. Specifically, we investigate whether engineers can identify faults faster by inspecting the test case rankings generated by our algorithm compared to inspecting test case rankings generated randomly or by coverage-based techniques. As for the coverage-based test prioritization, we compare with both the *additional* and *total* coverage-based test prioritization alternatives [122].

8.2 Study Subjects

We use two industrial Simulink models in our experiments: a Clutch Position Controller (CPC) and a Flap Position Controller (FPC) developed by Delphi Automotive Systems. Table 1 shows the key characteristics of these models. CPC and FPC are representative models from the automotive domain with many input variables and blocks. In Table 1, we report the total number of Simulink blocks and Stateflow states as well as input/output variables and configuration parameters for each model.

We further report in Table 1 the total number of decision goals in our study subjects. This is because in **RQ1** and **RQ3** we compare our approach with (baseline) coverage-based test generation and test prioritization algorithms that work based on decision coverage [98]. Specifically, the baseline algorithms aim to cover each one of the decision goals in a model under analysis at least once and thereby ensuring that all reachable blocks are executed. Decision goals in Simulink models are data inputs to switch blocks and conditional transitions emanating from the same state in a Stateflow model.

TABLE 1
Characteristics of our study subject Simulink models.

Name	No. Inputs	No. Configs	No. Outputs	No. Blocks/ States	No. Decision Goals
CPC	10	41	15	590	126
FPC	21	65	37	810	120

As discussed earlier, Simulink models have multiple output variables. These outputs can be categorized based on their function into *control*, *status*, or *diagnostic* outputs. Control outputs are commands applied to physical objects. These, for example, include physical signals representing a voltage applied to a DC motor to rotate a drive shaft in a car. Control outputs can be of type float (e.g., representing an analogue voltage signal), integer (e.g., representing a digital voltage signal) or enum/boolean (e.g., enabling or disabling a device). Status outputs report the system state variables, e.g., if a gate is open or close. They can be of type float (e.g., measurements such as estimated gas emission), integer (e.g., timer) or enum/boolean (e.g., gate open or close). Diagnostic outputs provide access to intermediary signal values and are used solely for debugging purposes. They can be of type float, integer or enum/boolean.

The CPC and FPC models are organized into five and six levels of subsystems, respectively. Both models contain various types of Simulink blocks including numerical and logical operations, from and goto blocks, lookup tables and S-Functions. Most of the computations are done by S-Functions which receive as input, configurable parameters, outputs of lookup tables or results of other computations. Both FPC and CPC are controller models and do not include a plant model. CPC controls the status of a clutch using a relatively large StateFlow including 13 states and 17 transitions. FPC implements five PIDs to control movements of a flap.

In our earlier work [58], our experiments focused on one main control output of CPC and FPC models. In this article, we account for all outputs of the CPC and FPC models except for those of type enum and boolean. The number of CPC and FPC outputs (excluding enum/boolean outputs) are 15 and 37, respectively (Table 1). We did not consider enum/boolean outputs because our notion of oracle is not meaningful for them. For ordinal values (i.e., enum values), the actual numerical quantities are meant to define some relative ranking over data points. Euclidean distances between vectors of ordinal values as prescribed by our oracle function would be meaningless. We note that while CPC and FPC have only four boolean and one enum outputs in total, they have 52 float and integer outputs. Based on our experience [57], Simulink models developed in the automotive industry tend to have several float and integer outputs, but few enum and boolean outputs.

8.3 Measuring Fault-Revealing Ability

We use our heuristic test oracle function, $oracle$, defined in Section 6 to automatically assess and compare the fault-revealing ability of test suites in our experimental setting. For the purpose of experimentation, we use fault-free versions of our subject models to produce the ground truth oracle signals (i.e., signal g in Section 6). Let $\mathcal{TC}$ be the set of all generated test cases for a given Simulink model M by a particular test generation technique, and let SG be the set of all signals sg_o^{tc} that are generated by a test case $tc \in \mathcal{TC}$ for an output o of M and are required to be inspected by engineers. We define an aggregated oracle function $Oracle$ over the set $\mathcal{TC}$ as follows:

$$Oracle(\mathcal{TC}) = MAX_{sg \in SG}\, oracle(sg)$$

That is, the aggregated oracle function, $Oracle$, returns the largest deviation between the ground truth oracle signal and all the output signals that are generated by $\mathcal{TC}$ and are expected to be checked by engineers. In order to reveal a fault, it is sufficient to have one fault-revealing test case among the test cases in $\mathcal{TC}$. Hence, we define $Oracle$ as the maximum of the deviations from the ground truth oracle generated by the test cases in $\mathcal{TC}$. We use a threshold value THR to translate the aggregated oracle $Oracle$ into a boolean fault-revealing measure denoted by FR. Specifically, FR returns true (i.e., $Oracle(\mathcal{TC}) > THR$) if some output signal in SG sufficiently deviates from the ground truth oracle such that a manual tester can conclusively detect 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 $THR = 0.25$ and showed that our results were not sensitive to such small changes in THR.

8.4 Measuring Test Prioritization Effectiveness

To compare the effectiveness of different prioritization algorithms, we measure how early faults can be detected when engineers inspect the test case rankings generated by alternative test prioritization algorithms. We use a metric, referred to as the Number of Tests to be Evaluated (NTE), that computes the number of test cases that need to be evaluated by engineers so that they can identify a fault. Lower NTE values denote faster fault detection, hence, more effective test prioritization. NTE directly counts the number of tests that need to be evaluated to find a fault, and provides a more intuitive measure to compare different test case rankings than existing evaluation metrics for test prioritization, such as APFD measure [115]. Finally we note that NTE values are impacted by the threshold THR used to compute the fault-revealing measure FR (see Section 8.3). Hence, in our experiments we report NTE values corresponding to the three different thresholds of 0.2, 0.15 and 0.25 used to compute FR.

8.5 Experiment Design

We developed a comprehensive list of Simulink fault patterns. We identified these patterns through our discussions with senior engineers from Delphi Automotive and by reviewing the existing literature on mutation operators for Simulink models [120], [17], [14], [114]. Tables 2 and 3 report these fault patterns. We note that these fault patterns represent the most common faults observed in practice.

To seed faults into the CPC and FPC models, we used an automated fault seeding program to generate the *mutant candidates* for the CPC and FPC models. We also developed a set of mutation operators corresponding to the fault patterns in Tables 2 and 3. Our fault seeding program enumerated each model element in each of these models, and mutated that element using mutation operators that were applicable to that element. Our fault seeding program generated 141 mutant candidates for CPC and 136 mutant candidates for FPC such that each mutant candidate has one fault. We then generated 10,000 test inputs for each of the CPC and FPC models using the adaptive random testing algorithm. We executed each mutant candidate of CPC and FPC using the

TABLE 2
Simulink Fault Patterns Identified at Delphi

Fault Pattern.	Corresponding Mutation Operator
Incorrect signal data types in math operations	Replacing a signal data type with a different data type, e.g., the MTALB "double" data type with MATLAB "single" data type, or MATLAB "fixdt(0,8,3)" data type with MATLAB "fixdt(0,8,2)" data type
Missing a "GoTo" block of a "From" block	Removing the "GoTo" block corresponding to a "From" block
Missing "Saturate on integer overflow" in math operations blocks	Unchecking the "Saturate on integer overflow" property for the blocks with this property checked
Missing "Signal name must resolve to Simulink signal object" in properties of a signal	Unchecking the "Signal name must resolve to Simulink signal object" property for the signals with this property checked
Improper "Merge" block utilization	Adding a Merge block for two signals that should not be merged

TABLE 3
Simulink Fault Patterns Identified in the literature [120], [17], [14], [114]

Fault Pattern.	Corresponding Mutation Operator
Incorrect signal data types	Replacing the MTALB "double" data type with MATLAB "single" data type, or MATLAB "fixdt(0,8,3)" data type with MATLAB "fixdt(0,8,2)" data type.
Incorrect Constant Values	Replacing constant c with constant $c-1$ or $c+1$; Negating boolean constants.
Incorrect Simulink blocks	Modifying arithmetic operators, e.g., replacing + with - or replacing + with $\times$. Modifying relation operators, e.g., replacing $\leq$ with $\geq$ or $=$ with $\neq$. Modifying logical operator, e.g., replacing $\wedge$ with $\vee$. Introducing boolean negation operators.
Incorrect connections	Switching the input lines of the "Switch" block
Incorrect Transition Conditions in Stateflow models	Modifying relation and logical operators.
Incorrect Actions in Stateflow models	Modifying arithmetic operators, modifying constants
Wrong Initial Conditions and Delay Values	Changing the initial value in "Integration" and "Unit Delay" blocks

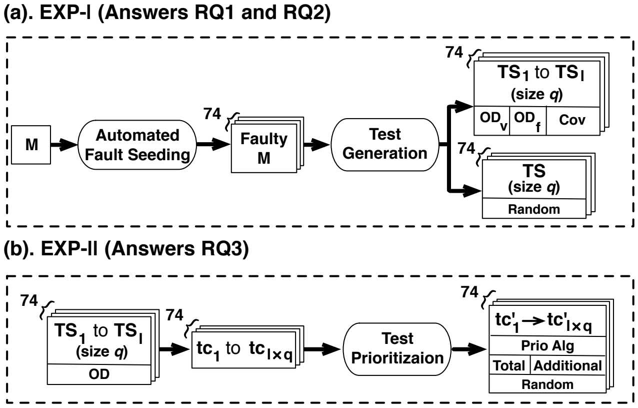


Fig. 11. Our experiment design: (a) **EXP-I** to answer **RQ1** and **RQ2**: test generation algorithms are repeated for 20 times to account for their randomness. (b) **EXP-II** to answer **RQ3**: **EXP-II** is repeated for all the fault-revealing OD test suites from **EXP-I**. Further, random prioritization is repeated for 20 times.

10,000 test cases. We discarded those mutant candidates whose output signals for all the 10,000 test cases exactly matched the corresponding reference model output signals. From the remaining mutant candidates, we randomly selected 44 mutants for CPC and 30 mutants for FPC as *chosen mutants* to be used in our experiments. We did so in such a way that among the chosen mutants we have a balanced and sufficient number of mutants for different fault pattern categories in Tables 2 and 3. We note that our experiments based on the 74 mutants were expensive and took 20 days to execute, excluding the process of removing equivalent mutants. So we had to limit the number of mutants in our experiments.

We then performed two experiments, **EXP-I** and **EXP-II**, to answer **RQ1** to **RQ3**, described below.

EXP-I focuses on answering **RQ1** and **RQ2**. Figure 11(a) shows the overall structure of **EXP-I**. We ran the OD algorithm in Figure 7 with vector-based (O_v) and feature-based (O_f) objective functions. We also ran our random and coverage-based (Cov) test generation algorithms. As mentioned in Section 8.1, for the Cov algorithm, we use an objective function that computes the set of Simulink blocks covered by test cases. Specifically, we use the *covers* function described in Section 5 for this purppose.

As shown in Figure 11(a), OD and Cov generate l separate test suites for l outputs of the model under test, while the random test generation algorithm generates one test suite for all the model outputs. For each faulty model and each objective function, we ran OD, Cov and Random for 600 sec and created test suites with the following sizes: 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 the test generation algorithms in **EXP-I** for 20 times to account for their randomness. Specifically, for 44 faulty versions of CPC model with 15 outputs and 30 faulty versions of FPC model with 37 outputs, we sampled 16152 (i.e., $44 \times 3 \times 3 \times 15 + 30 \times 3 \times 3 \times 37 + 74 \times 3$) different test suites and repeated each sampling 20 times (i.e., in total, 323040 different test suites were generated for **EXP-I**). Overall, **EXP-I** took around 20 days to run on our High Performance Clusters (HPC) [102]. Thanks to our HPC, we were able to parallelize **EXP-I** execution. Otherwise, it would have taken more than four years to complete **EXP-I** on a single core CPU system.

EXP-II answers the research question **RQ3** and evaluates our test prioritization algorithm. Figure 11(c) shows the overall structure of **EXP-II**. We used our prioritization algorithm in Figure 9 to rank the test cases generated by OD for the 74 faulty versions of the CPC and FPC models. We also used a random prioritization algorithm as well as the total and additional coverage-based test prioritization strategies [122] to rank the same test cases. We repeated **EXP-II** for all the fault-revealing test suites obtained by the 20 different runs of OD in EXP-I. We ignored those test suites obtained in **EXP-I** that were not able to detect any fault since test prioritization is irrelevant for them.

Recall from Section 5 that our prioritization algorithm in Figure 9 turns into an additional coverage-based prioritization algorithm by setting the fault-revealing probability function to one for all the test cases. If, in addition, we remove the part updating the faultiness probabilities of the covered nodes, the algorithm turns into a total coverage-based prioritization algorithm. When multiple test cases are equally desirable with respect to coverage, we select a test case randomly. Further, additional coverage strategy usually reaches a point where nodes are covered by at least one of the prioritized test cases and none of the remaining test cases can add any additional coverage. At this point, we reset the accumulative coverage and reapply the additional coverage strategy to order the remaining test cases. Overall, **EXP-II** took around half an hour to run on our HPC clusters. It would have taken more than a month on a single node. Note that, all the test cases were already executed during **EXP-I** and their dynamic test execution information, including coverage and output signals, were available before running **EXP-II**.

9 Results

This section provides responses, based on our experiment results, for research questions **RQ1** to **RQ3** described in Section 8. We have made the result data files available online [81].

RQ1 (Comparing OD with State-of-the-art). To answer **RQ1**, we ran **EXP-I** to compare our OD algorithm with Random and Cov. We ensured that the test suites generated by different algorithms have the same oracle cost (see Section 6). Figures 12(a) to (c) compare the fault-revealing ability of Random (R), Cov, and OD with the objective functions O_v and O_f. Each distribution in Figures 12(a) to (c) contains 74 points. Each point relates to one faulty model and represents, for the 20 test suites with size q obtained for that faulty model, the average aggregated oracle (i.e., $Oracle$) in the diagrams on the leftmost column, and the average fault revealing measure (i.e., FR) in the other diagrams. 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 = 0.149) in the $Oracle$ plot of Figure 12(a) indicates that the 20 different random test suites with size three generated for a faulty model achieved an average aggregated oracle of 0.149. Similarly, a point with (x = OD(O_f)) and (y = 0.85) in any of the FR plots of Figure 12(b) indicates that, among the 20 test suites with size five obtained for each output of a faulty model using OD with objective function O_f, 17 test suites had some fault-revealing test case (i.e., FR = 1), while three test suites had no fault-revealing test case (i.e., FR = 0).

To statistically compare the $Oracle$ and FR values, we performed the non-parametric pairwise Wilcoxon signed-rank test [19], and calculated the effect size using the Cohen's d [25]. The level of significance (α) was set to 0.05, and, following standard practice, d was labeled "small" for $0.2 \leq d \leq 0.5$, "medium" for $0.5 \leq d \leq 0.8$ and "high" for $d \geq 0.8$ [25].

Comparing fault-revealing ability of OD, R and Cov. The average $Oracle$ and FR values obtained by OD, with both objective functions O_f and O_v, for all the three thresholds and with all the three test suite sizes, are significantly better than those obtained by Random and Cov. Further, for all the comparisons between OD and Random, the effect size is consistently "high" for OD with both O_f and O_v. As for comparing OD with Cov, the effect size is "high" for all the comparisons except for the comparisons of FR distributions for OD(O_v) with test suite sizes five and ten, where the effect size is "medium".

Comparing decision coverage achieved by OD, R and Cov. Figure 13 compares the average percentages of decision coverage achieved by the 20 different runs of R, Cov, OD(O_f) and OD(O_v) over the faulty CPC and FPC models. As discussed in Section 5, in our work, the test coverage for a test case is a subset of the static backward slice of the output related to that test case. Therefore, we computed the values reported in Figure 13 by taking the average percentage of decision coverage for each test case in the fault-revealing test suite over the static backward slice of the output related to that test suite. As shown in Figure 13, Cov is able to achieve higher structural coverage than the two other algorithms across all the test suite sizes. Specifically, it achieves, on average, 89%, 91% and 93% decision coverage for the test suite sizes 3, 5 and 10, respectively. As shown in the figure, this is at least 3% points higher than the structural coverages achieved by the other algorithms across all the test suite sizes. Nevertheless, as shown in Figure 12, achieving higher structural coverage with Cov does not result in higher fault-revealing ability.

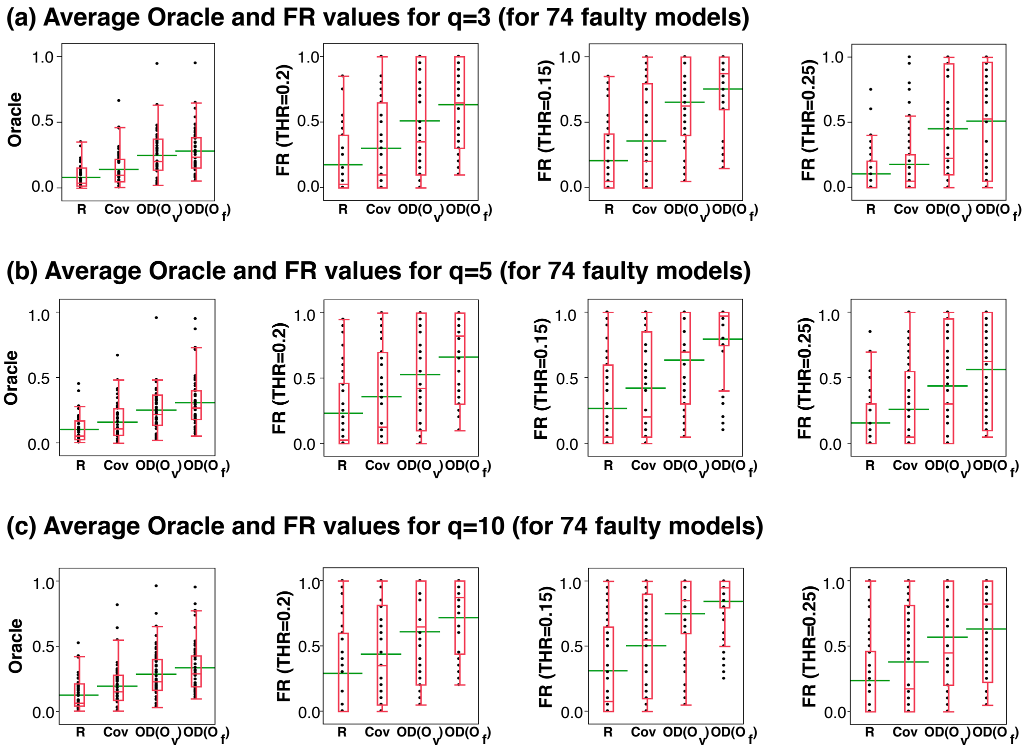
In summary, the answer to **RQ1** is that while OD's decision coverage is on average 4% points lower than the decision coverage achieved by Cov, the fault-revealing ability of OD significantly outperforms that of both Cov and Random.

RQ2 (Comparing O_f with O_v). The results in Figure 12 enable us to compare the average $Oracle$ and FR values for the feature-based, OD(O_f), and the vector-based, OD(O_v), output diversity algorithms.

Comparing fault-revealing abilities of OD(O_f) and OD(O_v). As for the average $Oracle$ distributions, the statistical test results indicate that OD(O_f) performs significantly better than OD(O_v) for the test suite sizes 5 and 10 with a "small" effect size. For the test suite size 3, there is no statistically significant difference, but OD(O_f) achieves higher mean and median $Oracle$ 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 .63, .66 and .71 for OD(O_f), and .51, .52 and .61 for OD(O_v) for the test suite sizes 3, 5, and 10, respectively. That is, across all the faults and with all the test suite sizes, the average probability of detecting a fault is at least %10 points higher when we use OD(O_f) instead of OD(O_v).

Why does OD(O_f) perform better than OD(O_v)? Here, we provide more insight as to why OD(O_f) achieves higher fault-revealing ability than OD(O_v). Specifically, our investigation of OD execution in our experiments indicated that OD ran for the same number of iterations with both O_v and O_f within the given test execution time budget. Recall that in Section 4.3, we discussed the asymptotic time complexity of individual iterations of OD(O_f) and OD(O_v). According to our experimental results, the time required to run the underlying model for q test cases (i.e., $T_M \times q$ in Section 4.3) significantly dominates the time required to compute O_f and O_v. Specifically, a single model execution T_M takes on average 1.1 second, while computing O_f or O_v takes on average 0.012 and 0.005 second, respectively. Since OD(O_f) and OD(O_v) are given the same test execution time budget in **EXP-I**, on average, they ran for the same number of iterations in our experiments. As a result, we conjecture that the reason for better fault-revealing ability of OD(O_f) lies in providing a better landscape for the search. That is, the feature-based diversity objective function provides a better surrogate for fault-revealing ability of the generated test suites compared to the vector-based output diversity objective function.

In summary, the answer to **RQ2** is that the fault-revealing ability of OD with the feature-based diversity objective is higher



(a) Average Oracle and FR values for q=3 (for 74 faulty models)

(b) Average Oracle and FR values for q=5 (for 74 faulty models)

(c) Average Oracle and FR values for q=10 (for 74 faulty models)

Fig. 12. Boxplots comparing average aggregated oracle values ($Oracle$) and fault revealing measures (FR) of OD (with both diversity objectives), coverage-based (Cov) and random test suites (R) for different thresholds and different test suite sizes.

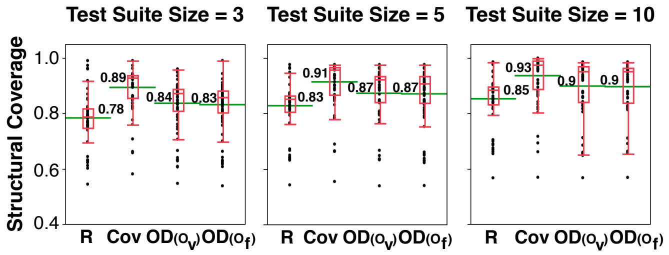


Fig. 13. The percentages of dynamic test coverage achieved by different test generation algorithms over the faulty versions of CPC and FPC subject models for different test suite sizes.

than that of OD with the vector-based diversity objective.

RQ3 (Comparing Test Prioritization with state-of-the-art). To answer **RQ3**, we performed **EXP-II** using the fault-revealing samples of the test suites generated by OD(O_f) (i.e., the best performing algorithm) in **EXP-I**. Figures 14 and 15 compare the average NTE distributions obtained by the random prioritization (R), total (Tot) and additional (Add) coverage-based prioritization, and our test prioritization (PrioAlg) algorithms for the CPC and FPC models, respectively. Note that in contrast to the *Oracle* and *FR* measures used in **EXP-I**, *NTE* measure is not normalized (e.g., it can go up to 45 for Figure 14(a), and up to 75 for Figure 14(b)). Hence, we present the results of **EXP-II** in separate plots for CPC and FPC case studies. Each distribution in Figures 14(a) to (c) (resp. in Figures 15(a) to (c)) contains 44 (resp. 30) points. Each point relates to one faulty model and represents the average *NTE* values obtained by applying a test prioritization algorithm to the combined set of test cases generated by OD(O_f) for that faulty model. Further, the results for random prioritization represent the average *NTE* values obtained over 20 different runs of the random prioritization algorithm. For example, a point with (x = Tot) and (y = 12.35) in any of the plots in Figure 14(c) indicates that among the 150 (i.e., 15 × 10) test cases generated for 15 outputs of CPC model, on average, when test cases are prioritized using the total coverage algorithm, 12.35 test cases need to be evaluated to find a fault. Similarly, a point with (x = PrioAlg) and (y = 9.8) in any of the plots in Figure 15(b) indicates that among all the 185 (i.e., 37 × 5) test cases generated for the 37 outputs of FPC, on average, when test cases are prioritized using our test prioritization algorithm, 9.8 test cases need to be evaluated to find a fault.

To statistically compare the *NTE* values, we used the same setting as in **EXP-I**. Recall that lower *NTE* values denote faster fault detection and hence more effective test prioritization. Testing differences in the average *NTE* distributions for both CPC and FPC models, for all the three thresholds, and with all the three test suite sizes, shows that PrioAlg performs significantly better than the other three algorithms. In addition, for all the comparisons between PrioAlg and both R and Tot, the effect size is consistently “high’. For the comparisons between PrioAlg and Add, the effect size is “high’ for the test suite sizes 5 and 10, and “medium’ for the test suite size 3.

The NTE results shown in Figures 14 and 15, in addition to demonstrating statistical significance, are practically significant as well. Specifically, across all the faults and with all the test suite sizes, on average, engineers inspect 12.1 fewer test cases (i.e., 48% fewer test cases) to find a fault when they use our prioritization (PrioAlg) algorithm instead of the additional coverage-based (Add) algorithm, the second best prioritization algorithm. That is,

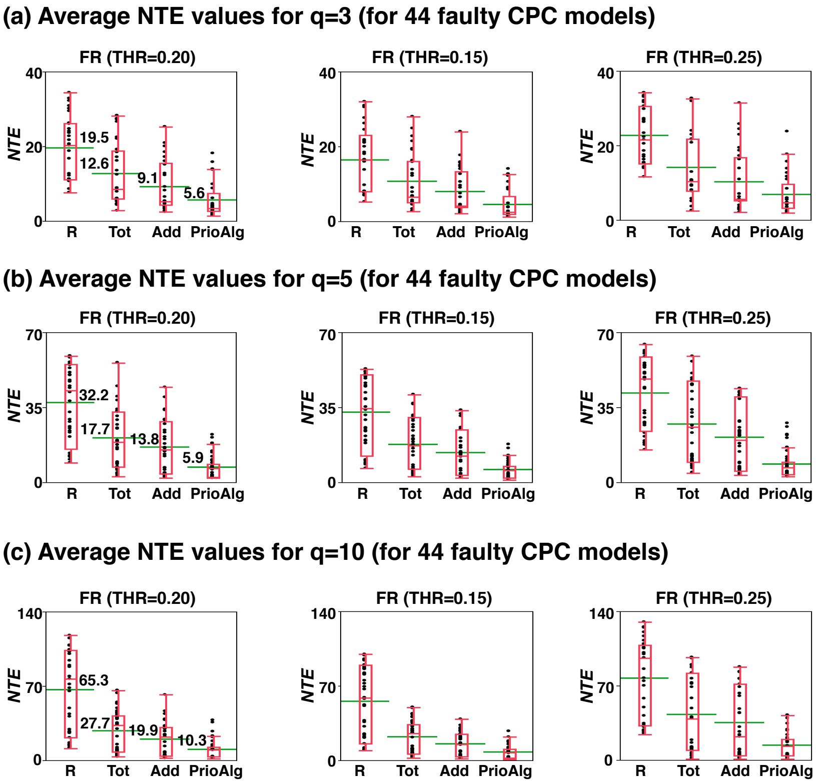


Fig. 14. Boxplots comparing average *NTE* values obtained by our prioritization algorithm (PrioAlg), coverage-based prioritization algorithm (Tot and Add) and random prioritization algorithm (R) with different thresholds and different test suite sizes for CPC case study.

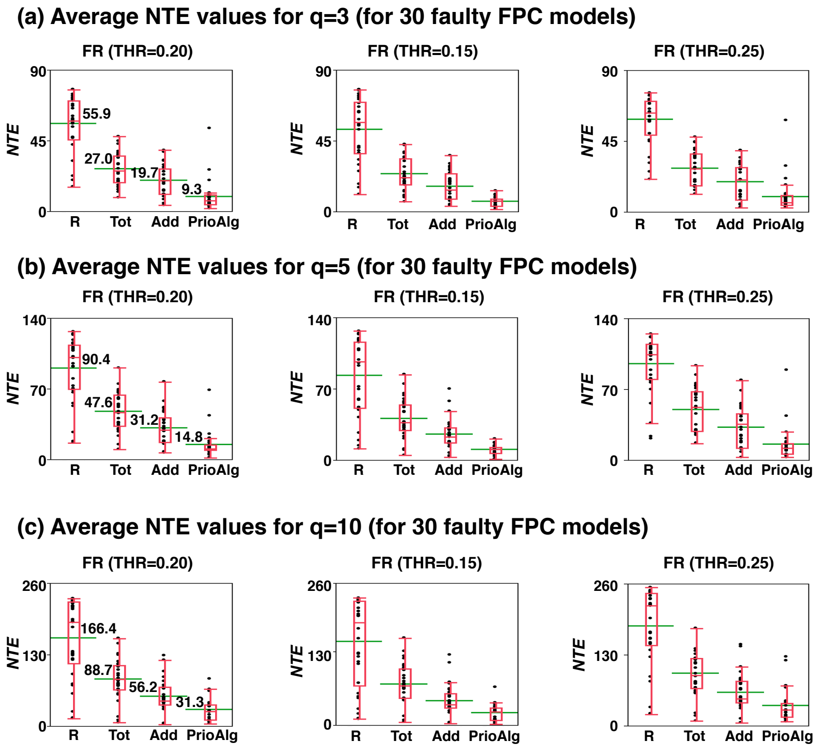


Fig. 15. Boxplots comparing average NTE values obtained by our prioritization algorithm (PrioAlg), coverage-based prioritization algorithm (Tot and Add) and random prioritization algorithm (R) with different thresholds and different test suite sizes for FPC case study.

PrioAlg reduces the time required to inspect test cases to almost half when compared with the existing state-of-the-art prioritization algorithms.

In summary, our prioritization algorithm significantly outperforms the random, and the total and additional coverage-based prioritization algorithms. Further, it reduces the inspection time by almost half compared to the second best performing prioritization algorithm.

Validity considerations and threats. Internal and external validity threats are the most relevant validity aspects in our experiments.

Internal validity: We mitigated the factors that could potentially cause confounding effects in our experiments. We repeated all of our experiments for three different test suite sizes of three, five and ten. In addition, our results are not impacted by small changes made to the fault revealing threshold *THR* and are consistent with the results obtained based on the quantitative fault revealing measure, i.e., *Oracle*, that does not rely on a threshold. We also note that in our experiments, we have reported the quantitative fault revealing measures obtained for the OD, Random and Cov algorithms without considering any threshold.

For our experiments, we obtained a comprehensive list of fault patterns for Simulink models based on our discussions with Delphi engineers as well as by surveying the literature. To discard mutants that are semantically equivalent to the reference model (i.e., the non-faulty model), we relied on an adaptive random testing algorithm. The issue that arises here is that we may have spuriously discarded some stubborn mutants, i.e., the mutants that are unlikely to be found by random (or adaptive random) testing. First, to mitigate this issue, in our work, we generated a large number of test inputs (i.e., 10,000 test inputs) and used *adaptive* random testing which attempts to maximize diversity among test inputs. Second, we note that the CPC and FPC models used in our evaluation contained complex S-Functions. The Simulink toolboxes that can perform Simulink model equivalence checking based on formal methods, e.g., SLDV, could not run on neither CPC nor FPC. Third, we note that our experiments compared our OD approach with two other *randomized* baseline algorithms: a random testing (Random) algorithm and a search-based coverage-based test generation (Cov) algorithm. The (potential) removal of stubborn mutants may impact the absolute fault finding of all three approaches reported in the experiment. Nevertheless, it is very unlikely this (potential) removal would have significantly impacted our comparison results with the Cov and Random algorithms and biased the results in favor of our OD approach.

External validity: To account for the cost of manual test oracles in practice, we considered small test suites that do not contain more than ten test cases. The test input signals used in our experiments were piecewise constant signals. According to our domain experts, such test inputs were sufficient for testing our study subjects. While we considered two industrial case studies in our experiments and we anticipate them to be representative of Simulink models in the automotive domain, additional case studies, in particular from other domains, will be essential in the future.

10 Real Faults in Industry Simulink models

In this section, we discuss three *real* faults that we were able to identify using our black-box output-based Simulink testing tool. We found these faults during a pilot study conducted in collaboration with Delphi engineers where we applied SimCoTest to a number of Simulink models that were under development at Delphi. We further argue why existing tools are unlikely to reveal these faults.

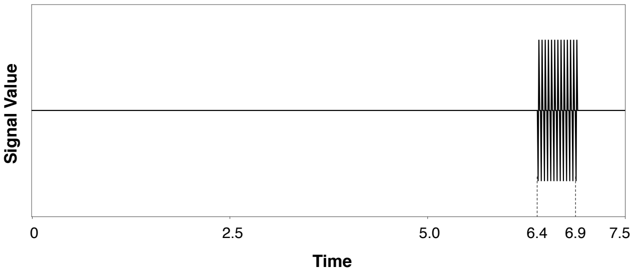


Fig. 16. An output signal containing instability failure caused by a real fault in an industrial Simulink model.

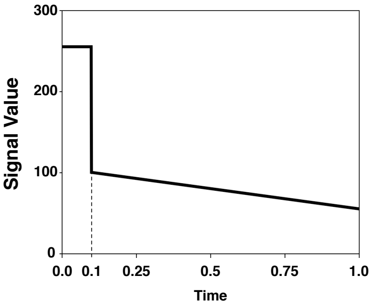


Fig. 17. A faulty output signal generated by output saturation on overflow.

The three outputs that revealed these faults are shown in Figures 16, 17 and 18(c). Specifically, (1) Figure 16 shows quick and frequent oscillations of a controller output over a time interval between 6.4 and 6.9 sec. These oscillations violate the controller stability requirement. (2) Figure 17 shows an output with a discrete jump at 0.1 sec. According to engineers, this jump is undesirable and indicates a fault. This fault was generated due to an output saturation of a Simulink block. (3) The third fault is related to a faulty delay buffer (Figure 18). The fault was due to an integer overflow inside the buffer. The impact was that *some* output signals (e.g., the output signal in Figure 18(c)) were not correctly-shifted copies of their corresponding input signals (e.g., the input signal in Figure 18(b)). These three faults were identified when engineers inspected test outputs generated by our black-box output diversity algorithm. These faults had *not* been previously found via manual expertise-based testing nor by commercial tools.

One important question is whether existing Simulink testing tools, given their underlying technology, can possibly find the above faults. In the remainder of this section, we try to answer this question considering the first usage mode of these tools that we discussed in Section 2.2 (i.e., checking Simulink models against formal properties). We note that the second usage mode of these tools was already discussed in Section 2.2. In this comparison, we consider the Reactis tool since it can test Simulink models against formal properties/assertions, and further, the Reactis license, in contrast to the license of Mathworks toolboxes, permits such comparisons.

Assertions capturing dynamic properties. Since Reactis was

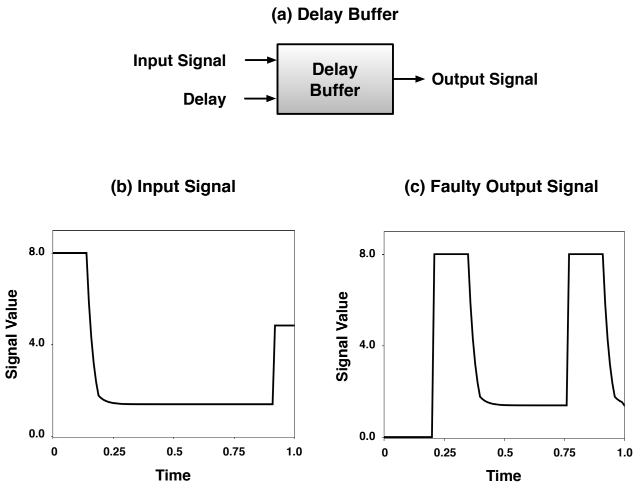


Fig. 18. A delay buffer that uses overflow as the underlying implementation mechanism, and an input signal and a faulty output signal of the delay buffer.

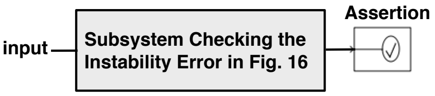


Fig. 19. A Simulink model to check if the fault in Figure 16 could be identified by Reactis.

not applicable to the model in which the fault in Figure 16 was originally observed, we created the Simulink model in Figure 19 to check if the fault in Figure 16 could be identified by Reactis. The model in Figure 19 includes a subsystem that returns zero if it identifies the behavior in Figure 16. Our implementation for the subsystem in Figure 19 is available at [79]. The output of this subsystem is connected to a Simulink assertion block. We used Reactis to generate an input signal such as the one in Figure 16 to trigger the assertion block. We let the tool execute for 24 hours but it did not generate any results. We conjecture that due to its underlying technology, Reactis is not able to find faults that manifest late during the simulation time (e.g., after 6000 steps in our example in Figure 16).

Runtime errors. Relying on runtime crashes as test oracles may not help with fault finding in practice due to some particular engineering practices in Simulink model development. Specifically, in Simulink models, to prevent runtime crashes, engineers often enable *output saturation on under/overflow* for all the blocks that may potentially lead to an under/overflow. Enabling this property generates a safety check for the respective block that sets the block's output to the maximum (resp. minimum) of its value range if the block produces an output larger (resp. less) than the maximum (resp. minimum) of the output range. This eliminates runtime crashes due to under/overflows in Simulink models. Nevertheless, engineers still need to ensure that saturating outputs do not lead to incorrect behaviors such as the erroneous signal jump in Figure 17 or the delay buffer error in Figure 18. Tools such as Reactis, however, do not address the identification of such erroneous behaviors since they focus on triggering runtime errors and assertions.

11 Related Work

As explained in Section 1, this article considerably extends and improves our previous papers [54], [58] and further provides a comprehensive exposition of our approach. In this section, we focus on comparing our approach with the most related research threads to our work on software testing, test case prioritization, controller testing and signal processing

11.1 Software Testing

A large part of existing test automation techniques rely on program analysis and focus on testing software implementation (source code). Our work, in contrast, aims to test models capturing both software and its environment. Having said that, we have used the following two specific ideas from the research focused on testing software code: (1) *Whole test suite generation*: Our algorithm uses whole test suite generation [29] that was proposed for unit 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 6) at the same time. (2) *Output uniqueness/diversity*: The notion of output diversity in our work is inspired by the output uniqueness criterion [2], [3]. As noted by Alshahwan and Harman [3], effectiveness of this criterion depends on the definition of *output difference* and differs from one context to another. While Alshahwan and Harman [2], [3] describe output differences in terms of the textual, visual or structural aspects of HTML code, in our work, output differences are characterized by signal shape features.

In the remainder of this section, we compare our work with testing approaches that rely on or relate to software models. In particular, we consider *model-based testing*, and *model checking and testing* techniques.

11.1.1 Model-based testing

Model-based testing relies on software models to generate both test scenarios and test oracles for testing 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. Below, we discuss these approaches in detail.

Coverage-based techniques. Various model-based testing tools have been developed to generate coverage-adequate test suites for Simulink/Stateflow models [70], [31], [15]. Search-based techniques have been applied to minimize a fitness function that approximates how far a given test input is from covering a specific Simulink block or Stateflow state [109], [110], [119]. Such fitness functions are typically defined in terms of metrics measuring the distance between input values and conditions characterizing the targeted blocks/states.

Reachability analysis is used to generate coverage-adequate test inputs or to provide proofs of correctness by showing unreachability of the faulty model parts [35], [64], [87]. For each coverage goal, a boolean assertion is instrumented into the model in such a way that violation of the assertion ensures coverage of the desired coverage goal and vice versa. The reachability analysis (e.g., using model checkers) either yields a counterexample (test scenario) demonstrating that the assertion under analysis is violated or it proves that the assertion is never violated, hence the underlying model is correct.

Reactis tester [83], [24], a well-known commercial tool for model-based testing of Simulink models, adapts a guided random test generation strategy consisting of two steps [88], [90]. First, test inputs are generated randomly. Second, the coverage goals that are not covered by the randomly generated inputs are attempted to be covered either using constraint solvers and static analysis or heuristic-based strategies.

Mutant-killing techniques. Another group of model-based testing techniques focus on generating mutant-killing test suites from Simulink models. These techniques assess the adequacy of test inputs by measuring the number of mutants that are detected by a given test suite. A mutant is detected by a test input if the test input yields different values for some output when applied to both the mutant model and the original model. Mutant-based test generation is done either using search techniques or behavioral analysis techniques (e.g., bounded reachability). Search techniques can be used to produce different outputs between the mutant model and the original model by generating different values at the fault seeded points and propagating those values to outputs [120], [121]. Alternatively, bounded reachability analysis techniques [17], [36] can be used to detect mutants by checking k-step (bi)similarity [45] between the original and the mutant models. The k-step (bi)similarity either asserts that the original and the mutant models are equivalent for the first k simulation steps or provides a test input showing that the models differ in some outputs.

Almost all existing model-based test generation approaches applied to Simulink/Stateflow consider only models with discrete behaviors. The work of Philipps et. al. [71] is one of the few exceptions and proposes a model-based testing approach for mixed discrete-continuous Simulink models. That work, however, focuses on generating test inputs from the discrete fragments of Simulink models. These test inputs are then applied to the original model to obtain test oracles in terms of continuous signals.

All the model-based testing techniques described above assume models are correct and aim to generate test suites and oracles from models. 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.

11.1.2 Model testing or verification

In contrast to model-based testing that focuses on deriving test cases from models to test implementation-level artifacts, model testing and model checking techniques aim to evaluate the correctness of models. We consider three categories of such techniques: (1) Model checking techniques that exhaustively verify correctness of models against some given formal properties, (2) Statistical model checking techniques that aim to provide probabilistic guarantees indicating that a model satisfies some given formal properties, and (3) Model testing techniques that attempt to identify faults in models by simulating models.

Model checking. Model checking is an exhaustive verification technique that explores the reachable states of a model in order to determine whether some given formal properties are satisfied or not [23]. It has a long history of application in software and hardware verification. It has been previously used to detect faults in Simulink models [35], [10] by showing that a path leading to an error (e.g., an assertion or a runtime error) is reachable. To solve the reachability problem, these techniques often need to translate Simulink models as well as the given properties into the input languages of some existing model checkers [62], [61], [89], [6]. For example, Barnat et al. [10] transform Simulink models into the DiVinE model checker's input language [11] to verify Simulink models against some linear temporal logic properties. Whalen et al. [106], [62] first translate Simulink models into the LUSTRE formal specification language [34] and then transform the LUSTRE specifications into the input languages of several well-known model checkers such as NuSMV [21] and the SAL tool suite [13]. Finally, Simulink Design Verifier [35] translates and feeds Simulink models into a commercial SMT-based model checker, called Prover [77]. Some alternative techniques [40], [8], [103] translate Simulink models into code and use existing code analysis tools to detect faults in the models. 8Cage [40] *marks* the Simulink models in places where specific fault models [74] are detected. It then converts the models into c-code and directs KLEE [18] toward those markers to generate test inputs that raise failures corresponding to the fault models. Polyglot [8] transforms Stateflow models into java code and uses JavaPathFinder [42] to analyze and check properties on the generated java code.

Statistical model checking. Model checking approaches, being exhaustive, suffer form the *state explosion problem* [23]. To alleviate the scalability problems of exhaustive model checking, statistical model checking approaches have been proposed. These approaches try to achieve scalability by checking some randomly sampled simulations from the space of all possible model simulations [116], [48]. They use statistical inference methods to answer whether the sampled simulations provide a statistical evidence for the satisfaction or violation of the properties of interest [116], [123]. Statistical model checking has been previously applied to Simulink models to estimate the probability that properties specified in temporal logic hold over models [123], [22]. Note that in contrast to model checking, statistical model checking does not guarantee to produce exact results (i.e., true/false results) and only estimates the probability of property satisfaction/violation.

Simulation-based testing. Simulation-based testing techniques run a set of test cases attempting to *falsify* assertions and properties instrumented into Simulink models [84], [86]. Reactis validator [84], [24] adapts such an approach by running the coverage-adequate test suites generated by Reactis tester [83] and tracking whether any assertions are violated by the test cases. S-Taliro toolbox [86], [5], [124] has usage modes that rely on Monte-Carlo to falsify Metric Temporal Logic properties [43] instrumented into Simulink models. Note that though these techniques look for possibility of assertion violations, they provide no guarantee to uncover all assertion violations.

The main limitation of model checking techniques when applied to Simulink models is the incompatibility challenge discussed in Section 1. Specifically, model checking is not applicable to dynamical systems, i.e., systems described solely in terms of time-continuous differential or difference equations [4]. Examples of dynamical systems include PID controllers [68] or mathematical models of physical plants. Model checking has been applied to *linear* hybrid systems i.e., linear systems whose dynamics consists of both continuous evolution of time and discrete instantaneous updates to states [4], [30]. Further, there has been techniques to extend model checking to *fragments* of non-linear hybrid automata by approximating nonlinear systems using piecewiseaffine models or using abstraction techniques [30]. More recently, translation techniques are proposed to convert Simulink models into hybrid automata models that can be verified using model checking [63]. However, industrial Simulink models are likely to include look-up tables, S-Functions containing legacy C code or executables, and switching conditions that depend on inputs. Presence of these features is likely to prohibit the translation into hybrid automata [1], [92], and hence, prevents analysis using the state of the art model checking tools, e.g., SpaceEx [30].

Statistical model checking [116] and simulation-based testing techniques such as S-Taliro [1] attempt to address the limitations of applying model checking to complex systems. Like our work, these approaches are black-box and analyze systems by sampling and simulating scenarios selected from their test input spaces. However, statistical model checking uses a randomized sampling to develop statistical guarantees that a given temporal property holds on a model. In contrast, our work uses a guided, randomized sampling to generate test suites that maximize the likelihood of triggering failures within a limited test budget.

The closest work to ours is the S-Taliro tool [1], [86] that uses random search techniques such as Monte Carlo simulation to test Simulink models by identifying scenarios violating a given formal temporal property. Our approach, however, does not rely on the presence of formal properties or any form of automatable test oracles. We focus on generating small test suites with high fault-revealing ability to effectively reduce the manual oracle cost. Further, in contrast to the S-Taliro tool, our approach is based on a dedicated search algorithm, tailored to the problem at hand.

11.2 Test Case Prioritization

Test case prioritization algorithms have been mostly studied in the context of regression testing where the goal is to identify an optimal ranking of test cases to help detect faults that might be introduced after a change as quickly as possible [78], [115], [85]. These techniques are broadly categorized into *dynamic* techniques that use test execution information, and *static* techniques that rely on static analysis of source code or other artifacts such as test code [78]. As discussed in Section 5, in this article, we take a greedy dynamic test prioritization algorithm to rank the generated test cases. We made this choice based on the following two contextual factors: First the number of test cases is relatively small in our work. Hence, a greedy algorithm will not be too expensive. Second, we have access to test execution information.

Existing dynamic test prioritization techniques typically rank test cases by relying on *total* or *additional* structural coverages achieved by individual test cases [78], [115]. To unify the total and additional coverage-based strategies, Zhang et al. [122] propose an algorithm that provides a knob to control the amount of feedback from previously prioritized test cases incorporated in prioritization of the remaining tests. No feedback from the previous iteration is equivalent to prioritization based on total coverage, and maximum feedback yields an additional coverage algorithm. Our test prioritization algorithm generalizes and extends this algorithm by explicitly considering the fault-revealing probability of individual test cases in test prioritization (i.e., FRP function in Figure 9). We consider the notion of output diversity as a proxy for FRP. This is because output diversity has shown to correlate to fault finding [3], [54], [58]. As a result, individual test cases with slightly lower coverage but coming from test suites with higher output diversity are likely to be ranked higher. As shown in Section 9, our prioritization algorithm significantly outperforms total and additional coverage-based prioritization and reduces the inspection time by almost half compared to existing coverage-based test prioritization.

11.3 Controller testing and Signal Generation

Continuous controllers have been widely studied in the control theory domain [68], [7], [105] where the focus has been to optimize controllers' behaviors for a specific application by design optimization [68] or for a specific hardware setup by configuration optimization [7]. In general, existing work in control theory mainly deals with optimizing the controller design or configuration rather than testing. They normally check and optimize the controller behavior over one, or a few number of test cases. These techniques, however, cannot substitute systematic testing as addressed by our approach.

In our earlier work, we proposed an approach to testing a class of continuous controllers known as *closed-loop controllers* based on automated test oracles derived from three types of continuous controller requirements: *stability*, *smoothness* and *responsiveness* [56], [53], [55]. We used meta-heuristic search to generate test cases maximizing the likelihood of presence of failures in controller outputs (i.e., test cases that produce outputs that break or are close to breaking stability, smoothness and responsiveness requirements). Our earlier work [56], [53], [55], however, cannot be used to test Simulink models in general because for closed-loop controllers, the environment (plant) feedback and the desired controller output (setpoint) [37] are both available. Hence, test oracles could be formalized and automated in terms of feedback and setpoints. In Simulink models that do not include plant models or contain open loop controllers, the plant feedback is not generally available.

Recent work in the intersection of Simulink testing and signal processing has focused on test input signal generation using evolutionary search methods [9], [111], [112], [50], [108]. Complex continuous input signals are generated either by sequencing parameterized signals [9], [111], or by modifying parameters of Fourier series characterizing signals [112]. These techniques, however, either apply the input signals to Simulink models to obtain test oracles, as in model-based testing, or assume automated oracles, e.g., assertions, are provided. Since they assume test oracles are not manual, they do not pose any restriction on the shape of test inputs. In our work, however, we restrict the number of steps in input signals as more complex inputs increase the oracle cost. Finally, similar to our work, the work of [118] proposes a set of signal features. These features are viewed as basic constructs which can be composed to specify test specifications as well as 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.

12 Conclusions

Simulink is a prevalent modeling language for Cyber Physical Systems (CPSs). In this article, we identified three main challenges in testing Simulink models, namely the incompatibility, oracle and scalability challenges. To address these challenges, we proposed a Simulink testing approach consisting of a test generation algorithm and a test prioritization algorithm. Our test generation algorithm is implemented using meta-heuristic search and is guided to produce test suites with output signals exhibiting a diverse set of signal features. Our test prioritization algorithm combines test coverage and test suite output diversity to automatically rank test cases according to their likelihood of revealing a fault. Our evaluation is performed using two industrial Simulink models and shows that (1) Our test generation approach significantly outperforms random and coverage-based test generation. (2) Our test prioritization algorithm significantly outperforms random and coverage-based test prioritization.

In future, we plan to combine output diversity and structural coverage objectives to achieve high structural coverage while maximizing output diversity. We note that generating coverage-adequate test suites for Simulink models containing continuous operations is still an open problem. We further plan to devise testing techniques that, instead of generating one test suite for each model output, generate one test suite for several model outputs together by relying on test objectives defined over a set of outputs. Such objectives, in addition to diversity, may rely on known relationships between model outputs or between outputs and inputs.

Acknowledgments

We gratefully acknowledge funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277) and from Delphi Automotive Systems, Luxembourg.

References

[1] H. Abbas, G. E. Fainekos, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Probabilistic temporal logic falsification of cyber-physical systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):95:1–95:30, 2013.

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

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

[4] R. Alur. *Principles of Cyber-Physical Systems*. MIT Press, 2015.

[5] Y. Annpureddy, C. Liu, G. E. Fainekos, and S. Sankaranarayanan. S-taliro: A tool for temporal logic falsification for hybrid systems. In *Proceedings of 17th International Conference Tools and Algorithms for the Construction and Analysis of Systems TACAS'11, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*, pages 254–257, 2011.

[6] D. Araiza-Illan, K. Eder, and A. Richards. Verification of control systems implemented in simulink with assertion checks and theorem proving: A case study. In *Control Conference (ECC), 2015 European*, pages 2670–2675. IEEE, 2015.

[7] M. Araki. PID control. *Control systems, robotics and automation*, 2:1–23, 2002.

[8] 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.

[9] 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.

[10] 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.

[11] J. Barnat, L. Brim, I. Cerna, P. Moravec, P. Rockai, and P. Simecek. Divine – a tool for distributed verification. In *International Conference on Computer Aided Verification*, pages 278–281. Springer, 2006.

[12] 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.

[13] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saıdi, N. Shankar, et al. An overview of sal. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*. Williamsburg, VA, 2000.

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

[15] 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.

[16] L. Briand, S. Nejati, M. Sabetzadeh, and D. Bianculli. Testing the untestable: model testing of complex software-intensive systems. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 789–792. ACM, 2016.

[17] 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 2009*, pages 208–227. Springer, 2009.

[18] 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.

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

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

[21] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[22] E. M. Clarke and P. Zuliani. Statistical model checking for cyber-physical systems. In *International Symposium on Automated Technology for Verification and Analysis*, pages 1–12. Springer, 2011.

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

[24] 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.

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

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

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

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

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

[30] G. Frehse, C. L. Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In *Proceedings of 23rd International Conference Computer Aided Verification, (CAV'11)*, pages 379–395, 2011.

[31] 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.

[32] G. Gay, A. Rajan, M. Staats, M. W. Whalen, and M. P. E. Heimdahl. The effect of program and model structure on the effectiveness of MC/DC test adequacy coverage. *ACM Transaction on Software Engineering Methodology*, 25(3):25:1–25:34, 2016.

[33] B. Gold, T. G. Stockham, A. V. Oppenheim, and C. M. Rader. *Digital processing of signals*. McGraw-Hill, 1969.

[34] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[35] G. Hamon. Simulink Design Verifier - Applying Automated Formal Methods to Simulink and Stateflow. In *AFM 2008*. Citeseer, 2008.

[36] N. He, P. Rümmer, and D. Kroening. Test-case generation for embedded simulink via formal concept analysis. In *Proceedings of the 48th Design Automation Conference*, pages 224–229. ACM, 2011.

[37] 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.

[38] H. Hemmati, A. Arcuri, and L. C. Briand. Achieving scalable model-based testing through test case diversity. *ACM Transaction Software Engineering and Methodology*, 22(1):6:1–6:42, 2013.

[39] T. A. Henzinger and J. Sifakis. The embedded systems design challenge. In *International Symposium on Formal Methods*, pages 1–15. Springer, 2006.

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

[41] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pages 435–445. ACM, 2014.

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

[43] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[44] 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.

[45] A. Kuehlmann and C. A. van Eijk. Combinational and sequential equivalence checking. In *Logic Synthesis and Verification*, pages 343–372. Springer, 2002.

[46] K. Lakhotia, N. Tillmann, M. Harman, and J. De Halleux. Flopsy-search-based floating point constraint solving for symbolic execution. In *Testing Software and Systems*, pages 142–157. Springer, 2010.

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

[48] A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In *International Conference on Runtime Verification*, pages 122–135. Springer, 2010.

[49] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering,*, 33(4):225–237, 2007.

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

[51] B. Liu, S. Nejati, L. C. Briand, and T. Bruckmann. Simulink fault localization: an iterative statistical debugging approach. *Software Testing, Verification and Reliability*, 2016.

[52] S. Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[53] R. Matinnejad, S. Nejati, L. Briand, and T. Brcukmann. Mil testing of highly configurable continuous controllers: scalable search using surrogate models. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 163–174. ACM, 2014.

[54] R. Matinnejad, S. Nejati, L. Briand, and T. Bruckmann. Effective Test Suites for Mixed Discrete-Continuous Stateflow Controllers. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, pages 84–95, 2015.

[55] R. Matinnejad, S. Nejati, L. Briand, T. Bruckmann, and C. Poull. Automated model-in-the-loop testing of continuous controllers using search. In *Search Based Software Engineering*, pages 141–157. Springer, 2013.

[56] R. Matinnejad, S. Nejati, L. Briand, T. Bruckmann, and C. Poull. Search-based automated testing of continuous controllers: Framework, tool support, and case studies. *Information and Software Technology*, 57:705–722, 2015.

[57] R. Matinnejad, S. Nejati, and L. C. Briand. Automated testing of hybrid simulink/stateflow controllers: industrial case studies. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*, pages 938–943, 2017.

[58] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann. Automated test suite generation for time-continuous simulink models. In *Proceedings of the 38th International Conference on Software Engineering, (ICSE'16)*, pages 595–606, 2016.

[59] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann. SimCoTest: a test suite generation tool for simulink/stateflow controllers. In *Proceedings of the 38th International Conference on Software Engineering, (ICSE'16)*, pages 585–588, 2016.

[60] 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.

[61] B. Meenakshi, A. Bhatnagar, and S. Roy. Tool for translating simulink models into input language of a model checker. In *International Conference on Formal Engineering Methods*, pages 606–620. Springer, 2006.

[62] S. P. Miller. Bridging the gap between model-based development and model checking. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 443–453. Springer, 2009.

[63] S. Minopoli and G. Frehse. SL2SX translator: From simulink to spaceex models. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016*, pages 93–98, 2016.

[64] 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.

[65] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 57–68. ACM, 2009.

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

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

[68] N. S. Nise. *Control Systems Engineering*. John-Wiely Sons, 4th edition, 2004.

[69] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *International Conference on Computer Aided Verification*, pages 411–414. Springer, 1996.

[70] 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.

[71] J. Philipps, G. Hahn, A. Pretschner, and T. Stauner. Tests for mixed discrete-continuous reactive systems. In *Proceedings of the 14th IEEE International Workshop on Rapid Systems Prototyping*, pages 78–84. IEEE, 2003.

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

[73] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner. Software engineering for automotive systems: A roadmap. In *2007 Future of Software Engineering*, pages 55–71. IEEE Computer Society, 2007.

[74] A. Pretschner, D. Holling, R. Eschbach, and M. Gemmar. A generic fault model for quality assurance. In *MODELS 2013*, pages 87–103. Springer, 2013.

[75] 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.

[76] A. Pretschner, C. Salzmann, B. Schätz, and T. Stauner. 4$^{\text{th}}$ intl. ICSE workshop on software engineering for automotive systems. *ACM SIGSOFT Software Engineering Notes*, 32(4):40, 2007.

[77] Prover Technology. Prover Plug-In Software. http://www.prover.com. [Online; accessed 17-Aug-2015].

[78] K. M. Qi Luo and D. Poshyvanyk. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2016.

[79] R. Matinnejad. Reactis test generation report. https://github.com/shnejati/TSE-Master/blob/master/SLDV-Reactis-TestGenerationReport/SLDV-Reactis-TestGenerationReport.pdf.

[80] R. Matinnejad. SimCoTest. https://sites.google.com/site/simcotesttool/.

[81] R. Matinnejad. The paper extra resources (technical reports, experiment results, and test plan and source code of SimCoTest tool). https://github.com/shnejati/TSE-Master.

[82] A. C. Rao, A. Rajeev, and A. Yeolekar. Applying design verification tools in automotive software v&v. Technical report, SAE Technical Paper, 2011.

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

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

[85] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering,*, 27(10):929–948, 2001.

[86] S-Taliro. S-Taliro Toolbox. https://sites.google.com/a/asu.edu/s-taliro/s-taliro. [Online; accessed 17-Aug-2015].

[87] 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.

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

[89] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a safe subset of simulink/stateflow into lustre. In *Proceedings of the 4th ACM international conference on Embedded software*, pages 259–268. ACM, 2004.

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

[91] M. Staats, G. Gay, M. Whalen, and M. Heimdahl. On the danger of coverage directed test case generation. In *Fundamental Approaches to Software Engineering (FASE)*, pages 409–424. Springer, 2012.

[92] T. Strathmann and J. Oehlerking. Verifying properties of an electro-mechanical braking system. In *Proceedings of the 2nd Workshop on Applied Verification of Continuous and Hybrid Systems (ARCH 2015)*, 4 2015.

[93] 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].

[94] The MathWorks Inc. Call MATLAB Function from C# Client. http://mathworks.com/help/matlab/matlab_external/call-matlab-function-from-c-client.html.

[95] 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].

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

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

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

[99] The Reactive Systems Inc. Reactis Coverage Metrics. http://www.reactive-systems.com/reactis/doc/user/user006.html. [Online; accessed 26-Jun-2016].

[100] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein. Static test case prioritization using topic models. *Empirical Software Engineering*, 19(1):182–212, 2014.

[101] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[102] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. Management of an academic hpc cluster: The UL experience. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 959–967. IEEE, 2014.

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

[104] G. A. Wainer. *Discrete-event modeling and simulation: a practitioner's approach*. CRC Press, 2009.

[105] T. Wescott. PID without a PhD. *Embedded Systems Programming*, 13(11):1–7, 2000.

[106] M. Whalen, D. Cofer, S. Miller, B. H. Krogh, and W. Storm. Integration of formal analysis into a model-based software development process. In *International Workshop on Formal Methods for Industrial Critical Systems*, pages 68–84. Springer, 2007.

[107] M. W. Whalen, G. Gay, D. You, M. P. E. Heimdahl, and M. Staats. Observable modified condition/decision coverage. In *Proceedings of 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 102–111, 2013.

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

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

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

[111] A. Windisch and N. Al Moubayed. Signal generation for search-based testing of continuous systems. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 121–130. IEEE, 2009.

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

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

[114] 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.

[115] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[116] H. L. Younes and R. G. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. *Information and Computation*, 204(9):1368–1409, 2006.

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

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

[119] Y. Zhan and J. Clark. Search based automatic test-data generation at an architectural level. In *Genetic and Evolutionary Computation Conference*, pages 1413–1424. Springer, 2004.

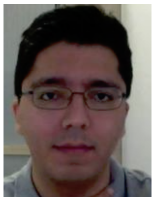
[120] Y. Zhan and J. A. Clark. Search-based mutation testing for Simulink models. In *GECCO 2005*, pages 1061–1068. ACM, 2005.

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

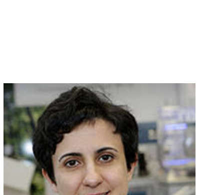
[122] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*, pages 192–201. IEEE, 2013.

[123] P. Zuliani, A. Platzer, and E. M. Clarke. Bayesian statistical model checking with application to stateflow/simulink verification. *Formal Methods in System Design*, 43(2):338–367, 2013.

[124] A. Zutshi, S. Sankaranarayanan, J. V. Deshmukh, J. Kapinski, and X. Jin. Falsification of safety properties for closed loop control systems. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 299–300. ACM, 2015.



Reza Matinnejad is an embedded software engineer at IEE company in Luxembourg. He received his PhD in September 2016 from university of Luxembourg, and his MSc and BSc in software engineering from Sharif University of Technology and Isfahan University of Technology in Iran. During his PhD, he received two ACM/SIGSOFT distinguished paper awards. His research interests include testing Cyber-Physical Systems (CPSs), search-based and model-based software testing, and empirical software engineering. Prior to his PhD, he worked for five years as a software engineer in developing software-intensive systems. More details can be found at: http://people.svv.lu/matinnejad/



Shiva Nejati is currently a Research Scientist at the SnT Centre for Security, Reliability, and Trust, University of Luxembourg. Form 2009 to 2012, she was a researcher at Simula Research Laboratory in Norway. She received her PhD degree from the University of Toronto, Canada in 2008. Nejati's research interests are in software engineering, focussing on model-based development, software testing, analysis of cyber physical systems, search-based software engineering and formal and empirical software engineering methods. Nejati has coauthored over 50 journal and conference papers, and regularly serves on the program committees of international conferences in the area of software engineering. She has for the past ten years been conducting her research in close collaboration with industry partners in telecommunication, maritime, energy, automotive and aerospace sectors. More details can be found at: http://people.svv.lu/nejati/

Lionel C. Briand is a professor in software verification and validation at the SnT centre for Security, Reliability, and Trust, University of Luxembourg, where he is also the vice-director of the centre. He is currently running multiple collaborative research projects with companies in the automotive, satellite, financial, and legal domains. Lionel has held various engineering, academic, and leading positions in five other countries before that. Lionel was elevated to the grade of IEEE Fellow in 2010 for his work on the testing of object-oriented systems. He was granted the IEEE Computer Society Harlan Mills award and the IEEE Reliability Society engineer-of-the-year award for his work on model-based verification and testing, respectively in 2012 and 2013. He received an ERC Advanced grant in 2016 – on the topic of modelling and testing cyber-physical systems – which is the most prestigious individual research grant in the European Union. His research interests include: software testing and verification, model-driven software development, search-based software engineering, and empirical software engineering. More details can be found at: http://people.svv.lu/briand/

Thomas Bruckmann received the Diploma Ing. (FH) degree in electrical engineering from the Institute of Technology in Trier (Germany). He is a Team Leader at Delphi Technologies where he heads the software test and validation group. He has been working as a senior engineer for nine years in software validation and hardware-in-the loop model development.