Effective Test Suites for Mixed Discrete-Continuous Stateflow Controllers

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

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

ABSTRACT

Modeling mixed discrete-continuous controllers using Stateflow is common practice and has a long tradition in the embedded software system industry. Testing Stateflow models is complicated by expensive and manual test oracles that are not amenable to full automation due to the complex continuous behaviors of such models. In this paper, we reduce the cost of manual test oracles by providing test case selection algorithms that help engineers develop small test suites with high fault revealing power for Stateflow models. We present six test selection algorithms for discrete-continuous Stateflows: An adaptive random test selection algorithm that diversifies test inputs, two white-box coverage-based algorithms, a black-box algorithm that diversifies test outputs, and two search-based black-box algorithms that aim to maximize the likelihood of presence of continuous output failure patterns. We evaluate and compare our test selection algorithms, and find that our three output-based algorithms consistently outperform the coverage- and input-based algorithms in revealing faults in discrete-continuous Stateflow models. Further, we show that our output-based algorithms are complementary as the two search-based algorithms perform best in revealing specific failures with small test suites, while the output diversity algorithm is able to identify different failure types better than other algorithms when test suites are above a certain size.

Categories and Subject Descriptors [Software Engineering]: Software/Program Verification

Keywords: Stateflow testing; mixed discrete-continuous behaviors; structural coverage; failure-based testing; output diversity.

1. INTRODUCTION

Automated software testing approaches are often hampered by the *test oracle problem*, i.e., devising a procedure that distinguishes between the correct and incorrect behaviors of the system under test [5, 38]. Despite new advances in test automation, test oracles most often rely on human knowledge and expertise, and thus, are the most difficult testing ingredient to automate [5, 26]. In this situation, in order to reduce the cost of human test oracles, test case selection criteria have been proposed as a way to obtain minimal test suites with high fault revealing power [15, 17].

ESEC/FSE '15, August 30-September 04, 2015, Bergamo, Italy.

http://dx.doi.org/10.1145/2786805.2786818.

Many embedded software systems in various industry sectors are developed using Stateflow [43, 13, 46], which is integrated into the Matlab/Simulink language. Stateflow is a hierarchical state machine language that is most commonly used in practice to specify mixed discrete-continuous behaviors [18, 39, 19]. Such behavior evolves in continuous time with discrete jumps at particular time instances [45], and is typically captured using a Stateflow model consisting of some discrete states such that during each state, the system may behave based on a continuous-time differential or a difference equation. The state equations may change when the system transitions to its subsequent states due to some discrete-time event.

Stateflow models, being detailed enough to enable code generation and simulation, are subject to extensive testing. Testing Stateflow models allows engineers to verify the behavior of software functions, and is more likely to help with fault finding compared to testing code as Stateflow models are more abstract and more informative for engineers. Testing Stateflow models, however, is complicated mostly due to their continuous behaviors [51].

Existing work on formalizing and automating test oracles for Stateflow, and in general for other state machine dialects, has primarily focused on implicit test oracles [40] such as runtime errors, e.g., division by zero and data type overflow, or on oracles based on discrete behavior properties [35] such as temporal reachability, termination, or state invariants which can be specified using logical assertions or temporal logic [4, 14]. Compared to test oracle automation for discrete system behaviors, the problem of developing and automating test oracles for continuous behaviors has received significantly less attention, and is left unexplored.

In our earlier work, we proposed an approach to automate test oracles for a class of embedded controllers known as *closed-loop controllers* and for three types of continuous output failures: *stability*, *smoothness* and *responsiveness* [24, 25]. Our approach used metaheuristic 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). This approach, however, fails to automate test oracles for Stateflows because for closed-loop controllers, the environment (plant) feedback and the desired controller output (setpoint) [16] are both available. Hence, test oracles could be formalized and automated in terms of feedback and setpoint. For Stateflow models, which typically implement *open-loop controllers* [48], the plant feedback is not available.

Given that test oracles for Stateflow models are not amenable to full automation mostly due to their complex continuous-time behaviors [51], in this paper, we focus on providing test case selection algorithms for Stateflow models. Our algorithms help engineers develop small test suites with high fault revealing power for continuous behaviors, effectively reducing the cost of human test oracles [5, 26]. In this paper, we present and evaluate six test selection algorithms for mixed discrete-continuous Stateflow models: *A black-box adaptive random input-based algorithm, two white-box adaptive random coverage-based algorithms, a black-box adaptive random output-based algorithm, and two black-box search-based output-based algorithms*. Our adaptive random input-based algorithm simply attempts to generate a test suite by diversifying test inputs. Our two white-box adaptive random coverage-based algorithms aim to achieve high structural coverage. Specifically, we consider the well-known state and transition coverage criteria [6] for Stateflow models. Our black-box adaptive random output-based algorithm aims to maximize *output diversity*, i.e., diversity in continuous outputs of Stateflow models. Output diversity is an adaptation of the recently proposed output uniqueness criterion [1, 2] to Stateflow. Output uniqueness has been studied for web applications and has shown to be a useful surrogate to white-box selection techniques. We consider this criterion in our work because Stateflows have rich time-continuous outputs, providing a useful source of information for fault detection.

Our black-box search-based output-based algorithms rely on metaheuristic search [21] and aim to maximize objective functions capturing the degree of presence of continuous output failure patterns. Inspired by discussions with control engineers, we propose and formalize two continuous output failure patterns, referred to as *instability* and *discontinuity*. The instability pattern is characterized by quick and frequent oscillations of the controller output over a time interval, and the discontinuity pattern captures fast, short-duration and upward or downward pulses (i.e., spikes [49]) in the controller output. Presence of either of these failure patterns in Stateflow outputs may have undesirable impact on physical processes or objects that are controlled by or interact with a Stateflow model.

Our contributions: (1) We focus on the problem of testing mixed discrete-continuous Stateflow models. We propose two new failure patterns capturing undesirable behaviors in Stateflow outputs that may potentially harm physical processes and objects. We develop black-box search-based test selection algorithms generating test inputs that are likely to produce continuous outputs exhibiting these failure patterns. In addition, we define a black-box output diversity test selection criterion for Stateflow, and present an adaptive random test selection algorithm based on this criterion.

(2) We evaluate our six algorithms, comparing the output-based selection algorithms with the coverage-based and input-based selection algorithms. Our evaluation uses three Stateflow models, including two industrial ones. Our results show that the output-based algorithms consistently outperform the coverage-based and input-based algorithms in revealing Stateflow model faults. Further, for relatively larger test suites, the coverage-based algorithms are subsumed by the output diversity algorithm, i.e., any fault found by the coverage-based algorithms are also found with the same or higher probability by the output diversity algorithm. Finally, we show that our adaptive random and search-based output-based algorithms are complementary as the search-based algorithms perform best in revealing instability and discontinuity failures even when test suites are small, while the adaptive random output diversity algorithm is able to identify different failure types better than the search-based algorithms when test suites are above a certain size.

2. BACKGROUND AND MOTIVATION

Motivating example. We motivate our work using a simplified Stateflow from the automotive domain which controls a supercharger clutch and is referred to as the Supercharger Clutch Controller (SCC). Figure 1(a) represents the discrete behavior of SCC specifying that the supercharger clutch can be in two *quiescent* states: engaged

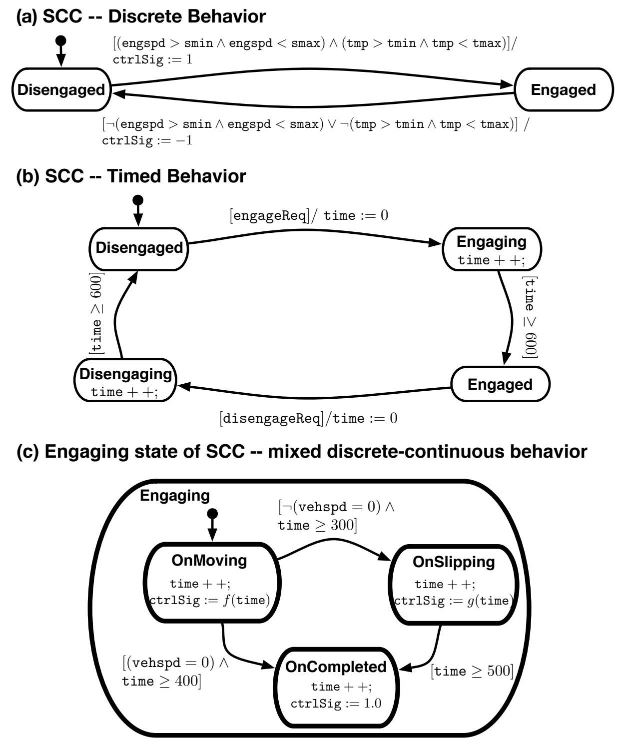

Figure 1: Supercharge Clutch Controller (SCC) Stateflow.

or disengaged. Further, the clutch moves from the disengaged to the engaged state whenever both the engine speed `engspd` and the engine coolant temperature `tmp` respectively fall inside the specified ranges of [`smin`..`smax`] and [`tmin`..`tmax`]. The clutch moves back from the engaged to the disengaged state whenever either the speed or the temperature falls outside their respective ranges. The variable `ctrlSig` in Figure 1(a) indicates the sign and magnitude of the voltage applied to the DC motor of the clutch to physically move the clutch between engaged and disengaged positions. Assigning 1.0 to `ctrlSig` moves the clutch to the engaged position, and assigning -1.0 to `ctrlSig` moves it back to the disengaged position. To avoid clutter in our figures, we use `engageReq` to refer to the condition on the **Disengaged** $\rightarrow$ **Engaged** transition, and `disengageReq` to refer to the condition on the **Engaged** $\rightarrow$ **Disengaged** transition.

The discrete transition system in Figure 1(a) assumes that the clutch movement takes no time, and further, does not provide any insight on the quality of movement of the clutch. Figure 1(b) extends the discrete transition system in Figure 1(a) by adding a timer variable, i.e., `time`, to explicate the passage of time in the SCC behavior. The new transition system in Figure 1(b) includes two *transient* states, engaging and disengaging, specifying that moving from the engaged to the disengaged state and vice versa takes 600 *ms*. Since this model is simplified, it does not show handling of alterations of the clutch state during the transient states. In addition to adding the `time` variable, we note that the variable `ctrlSig`, which controls physical movement of the clutch, cannot abruptly jump from 1.0 to -1.0, or vice versa. In order to ensure safe and smooth movement of the clutch, the variable `ctrlSig` has to gradually move between 1.0 and -1.0 and be described as a function over time, i.e., a signal. To express the evolution of the `ctrlSig` signal over time, we decompose the transient states engaging and disengaging into sub-state machines. Figure 1(c) shows the sub-state machine related to the engaging state. The one related to the disengaging state is similar. At beginning (state **OnMoving**), the func-

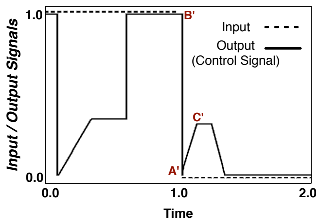

Figure 2: An example input (dashed line) and output (solid line) signals for SCC in Figure 1. The input signal represents `engageReq`, and the output signal represents `ctrlSig`.

tion `ctrlSig` has a steep grade (i.e., function f) to move the stationary clutch from the disengaged state and accelerate it to reach a certain speed in 300 *ms*. Afterwards (state **OnSlipping**), `ctrlSig` decreases the speed of clutch based on the gradual function g for 200 *ms*. This is to ensure that the clutch slows down as it gets closer to the crankshaft. Finally, at state **OnCompleted**, `ctrlSig` reaches value 1.0 and remains constant, causing the clutch to get engaged in about 100 *ms*. When the car is stationary, i.e., `vehspd` is 0, the clutch moves based on the steep grade function f for 400 *ms*, and does not have to go to the **OnSlipping** phase to slow down before it reaches the crankshaft at state **OnCompleted**.

Input and Output. The Stateflow inputs and outputs are signals (functions over time). Each input/output signal has a data type, e.g. boolean, enum or float, specifying the range of the signal. For example, Figure 2 shows an example input (dashed line) and output (solid line) signals for SCC. The input signal is related to `engageReq` and is boolean, while the output signal is related to `ctrlSig` and is a float signal. The simulation length, i.e., the time interval during which the signals are observed, is two sec for both signals. In theory, the input signals to Stateflow models can have complex shapes. In practice, however, engineers mostly test Stateflow models using *constant or step* input signals over a fixed time interval. This is because developing manual test oracles for arbitrary and complex input signals is difficult and time consuming.

Stateflow outputs might be either discrete or continuous. A discrete output is represented by a boolean or an enum signal that takes a constant value at each state. A continuous output is represented by a float signal that changes over time based on a difference or differential equation (e.g., `ctrlSig` in Figure 1(c)). Our focus in this paper is on Stateflows with some continuous outputs.

Stateflow requirements. The specification of Stateflow controllers typically includes the following kinds of requirements: (1) Requirements that can be specified as assertions or temporal logic properties over pure discrete behavior (e.g., the state machine in Figure 1(a)). For example, *If engine speed* `engspd` *and temperature* `tmp` *fall inside the ranges* [`smin..smax`] *and* [`tmin..tmax`], *respectively, the clutch should eventually be engaged.* (2) Requirements that focus on timeliness of the clutch behavior and rely on the `time` variable (see Figure 1(b)). For example, *moving the clutch from disengaged to engaged or vice versa should take* 600 ms. Note that SCC is an open-loop controller [48] and it does not receive any information from the clutch to know its whereabouts. Hence, engineers need to estimate the position (state) of the clutch using timing constraints. (3) Requirements characterizing continuous dynamics of controlled physical objects. For example, *the clutch should move smoothly without any oscillations, and it should not bump into the crankshaft or other physical components close to it.* Engineers need to evaluate the continuous `ctrlSig` signal to ensure that it does not exhibit any erratic or unexpected change with any undesirable impact on physical processes or objects.

Existing literature such as model checking and formal verification [10] largely focuses on properties that fall in groups one and two above [29]. The third group of requirements above, although of paramount importance for correct dynamic behavior of controllers, are lesser studied in the software testing literature compared to the requirements in the first and second groups. To evaluate outputs with respect to the requirements in the third group, engineers have to evaluate *the changes in the output over a time period.* In contrast, model checkers focus on discrete-time behaviours only, and evaluate outputs at a few discrete time instances (states), ignoring the pattern of output changes over time.

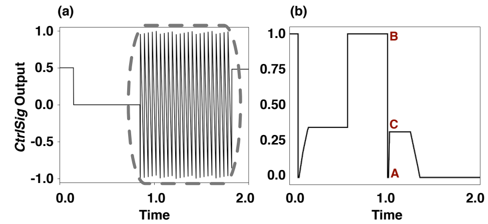

Figure 3: The output signals (`ctrlSig`) for two faulty versions of SCC: (a) an unstable output, and (b) a discontinuous output.

Failure patterns. Figure 3 shows two specific patterns of failures in continuous output signals, violating requirements on desired physical behaviors of controllers (group three). The failure in Figure 3(a) shows *instability*, and the one in Figure 3(b) refers to *discontinuity*. Specifically, the former signal shows quick and frequent oscillations of the controller output in the area marked by a grey dashed rounded box, and the latter shows a very short-duration pulse in the controller output at point A. In Section 3, we provide a number of test selection algorithms to generate test cases that reveal failures in mixed discrete-continuous Stateflow outputs including the two failure patterns in Figure 3.

3. TEST SELECTION ALGORITHMS

In our work, we propose the following test case selection algorithms to develop test suites that can reveal erroneous continuous outputs of Stateflow models:

Black-box input diversity (ID). Our input diversity selection algorithm is adaptive random and attempts to maximize diversity of test inputs selected from the input search space. Adaptive random test selection [3, 9] is a simple strategy that is commonly used as a baseline for comparison. A selection algorithm has to perform at least better than adaptive random to be considered worthwhile.

White-box coverage-based. Structural coverage criteria have been extensively studied in software testing as a method for measuring test suite effectiveness [28, 20]. We consider the two well-known *state coverage (SC)* and *transition coverage (TC)* criteria for Stateflows [6] mostly as another baseline of comparison.

Black-box output diversity (OD). In the context of web applications, recent studies have shown that selecting test cases based on outputs uniqueness, i.e., selecting test cases that produce highly diverse or distinct outputs, enhance fault finding effectiveness of test suites [1, 2]. Stateflow outputs provide a primary source of data for engineers to find faults. Hence, in our work, we adapt the output uniqueness proposed by [1, 2] and define a notion of output diversity over continuous control signals.

Black-box failure-based. The goal of failure-based test selection algorithms is to select test inputs that are able to reveal common failures specific to a particular domain [32]. We identify two failure patterns related to continuous dynamics of controllers: instability and discontinuity. Based on these two patterns, we define two failure-based and output-based selection algorithms, *output stability (OS)* and *output continuity (OC)*. Output stability aims to select test inputs that are likely to produce outputs exhibiting a period of instability, particularly in response to a sudden change in input.

An example of an output with instability failure is shown in Figure 3(a). A period of instability in this signal, which is applied to a physical device, may result in hardware damage and must be investigated by engineers.

In contrast, output continuity attempts to select test inputs that are likely to produce discontinuous outputs. The control output of a Stateflow is a continuous function with some discrete jumps at state transitions. For example, for both the control signals in Figures 2 and 3(b), there is a discrete jump at around time 1.0 sec (i.e., point A′ in Figure 2, and point A in Figure 3(b)). At discrete jumps, and in general at every simulation step, the control signals are expected to be either left-continuous or right-continuous, or both. For example, the signal in Figure 2 is right-continuous at point A′ due to the slope from A′ to C′, and hence, this signal does not exhibit any discontinuity failure at point A′. However, the signal in Figure 3(b) is neither right-continuous nor left-continuous at point A. This signal, which is obtained from a faulty version of SCC, shows a very short duration pulse (i.e., a spike) at point A. This behavior is unacceptable because it may damage the clutch by imposing an abrupt change in the voltage applied to the clutch [49]. Specifically, the failure shown in Figure 3(b) is due to a fault in a transition condition in the SCC model. Due to this faulty condition, the controller leaves a state immediately after it enters that state and modifies the control signal value from B to A.

In the remainder of this section, we first provide a formal definition of the test selection problem, and we then present our test selection algorithms.

Test Selection Problem. Let $SF = (\Sigma, \Theta, \Gamma, o)$ be a Stateflow model where $\Sigma = \{s_1, \ldots, s_n\}$ is the set of states, $\Theta = \{r_1, \ldots, r_m\}$ is the set of transitions, $\Gamma = \{i_1, \ldots, i_d\}$ is the set of input variables, and o is the controller output of the Stateflow model based on which we want to select test cases. Typically, embedded software controllers have one main output, i.e., the control signal, applied to the device under control. If a Stateflow model has more than one output, we can apply our approach to select test cases for each individual output separately.

Note that Stateflow models can be hierarchical or may have parallel states. Among our selection algorithms, only state and transition coverage algorithms, SC and TC, are impacted by the Stateflow structure. In our work, we assume that Σ and Θ, respectively, contain the states and transitions in flattened Stateflow models [37]. However, our SC and TC algorithms do not require to statically flatten Stateflow models as these algorithms dynamically identify the (flattened) states and (flattened) transitions that are actually executed during simulation of Stateflow models.

Each input/output variable of SF is a signal, i.e., a function of time. When SF is simulated, its input/output signals are discretized and represented as vectors whose elements are indexed by time. Assuming that the simulation time is T, the simulation interval $[0..T]$ is divided into small equal time steps denoted by Δt. For example for SCC, we set $T = 2s$ and $\Delta t = 1ms$. We define a signal sg as a function $sg : \{0, \Delta t, 2\cdot\Delta t, \ldots, k\cdot\Delta t\} \rightarrow \mathcal{R}_{sg}$, where Δt is the simulation time step, k is the number of observed simulation steps, and $\mathcal{R}_{sg}$ is the signal range. In our example, we have $k = 2000$. We further denote by $min_{\mathcal{R}_{sg}}$ and $max_{\mathcal{R}_{sg}}$ the min and the max of $\mathcal{R}_{sg}$. For example, when sg is a boolean, $\mathcal{R}_{sg}$ is $\{0, 1\}$, and when sg is a float signal, $\mathcal{R}_{sg}$ is the set of float values between $min_{\mathcal{R}_{sg}}$ and $max_{\mathcal{R}_{sg}}$. As discussed in Section 2, to ensure the feasibility of the generated input signals, in this paper, we only consider constant or step input signals.

Our goal is to select a test suite $TS = \{I_1, \ldots, I_q\}$ of q test inputs where q is determined by the human test oracle budget. Each

Algorithm. ID-SELECTION

Input: Stateflow model SF.
Output: Test suite $TS = \{J_1, \ldots, J_q\}$.

1. Let $TS = \{I\}$ where I is a random test input of SF
2. **for** $q - 1$ times **do:**
3. $\quad MaxDist = 0$
4. $\quad$ Let $C = \{I_1, \ldots, I_c\}$ be a candidate set of random test inputs of SF
5. $\quad$ **for each** $I_i \in C$ **do:**
6. $\quad\quad Dist = \text{MIN}_{\forall I' \in TS} dist(I_i, I')$
7. $\quad\quad$ **if** $Dist > MaxDist$ **:**
8. $\quad\quad\quad MaxDist = Dist, J = I_i$
9. $\quad TS = TS \cup J$
10. return TS

Figure 4: The input diversity test selection algorithm (ID).

test input I_j is a vector $(sg_1, \ldots, sg_d)$ of signals for the SF input variables i_1 to i_d. By simulating SF using each test input I_j, we obtain an output signal sg_o for the continuous output o of SF.

3.1 Input Diversity Test Selection

The input diversity selection algorithm (ID) generates a test suite with diverse test inputs. Given two test inputs $I = (sg_1, \ldots, sg_d)$ and $I' = (sg'_1, \ldots, sg'_d)$, we define the *normalized* Euclidean distance between each pair sg_j and sg'_j of signals as follows:

$$\hat{dist}(sg_j, sg'_j) = \frac{\sqrt{\sum_{i=0}^{k} (sg_j(i\cdot\Delta t) - sg'_j(i\cdot\Delta t))^2}}{\sqrt{k+1} \times (max_{\mathcal{R}_{sg}} - min_{\mathcal{R}_{sg}})} \quad (1)$$

Note that sg_j and sg'_j are alternative assignments to the same SF input i_j, and hence, they have the same range. Further, we assume that the values of k and Δt are the same for sg_j and sg'_j. It is easy to see that $\hat{dist}(sg, sg')$ is always between 0 and 1.

We define the distance between two test inputs $I = (sg_1, \ldots, sg_d)$ and $I' = (sg'_1, \ldots, sg'_d)$ as the sum of the normalized distances between each signal pair:

$$dist(I, I') = \sum_{j=1}^{d} \hat{dist}(sg_j, sg'_j) \quad (2)$$

Figure 4 shows the ID-SELECTION algorithm which, given a Stateflow model SF, generates a test suite TS with size q and with diverse test inputs. The algorithm first randomly selects a single test input and stores it in TS (line 1). Then, at each iteration, it randomly generates c candidate test inputs $I_1, \ldots, I_c$. It computes the distance of each test input I_i from the existing test suite TS as the minimum of the distances between I_i and the test inputs in TS (line 6). Finally, the algorithm identifies and stores in TS the test input among the c candidates with the maximum distance from the test inputs in TS (lines $7-9$).

3.2 Coverage-based Test Selection

In order to generate a test suite TS based on the state/transition coverage criterion, we need to simulate SF using each one of the candidate test inputs and compute the state and the transition coverage reports for each test input simulation. The state coverage report S is a subset of $\Sigma = \{s_1, \ldots, s_n\}$ containing the states covered by the test input I, and the transition coverage report R is a subset of $\Theta = \{r_1, \ldots, r_m\}$ containing the transitions covered by I.

The state coverage test selection algorithm, SC-SELECTION, is shown in Figure 5. The algorithm for transition coverage, TC-SELECTION, is obtained by replacing S (state coverage report) with T (transition coverage report). At line 1, the algorithm selects a random test input I and adds it to TS. At line 2, it simulates SF using I and adds the corresponding state coverage report to a set TSC. At each iteration the algorithm generates c candidate test inputs and keeps their corresponding state coverage reports in a set CC. It then computes the additional coverage that each one of the test inputs among the c candidates brings about compared to the coverage obtained by the existing test suite TS (line 8). At the end of the iteration, the test input that leads to the maximum

Algorithm. SC-SELECTION
Input: Stateflow model SF.
Output: Test suite $TS = \{J_1, \ldots, J_q\}$.

1. Let $TS = \{I\}$ where I is a random test input of SF
2. Let $TSC = \{S\}$ where S is the state coverage reports of executing SF with I
3. **for** $q-1$ times **do:**
4. $\quad MaxAddCov = 0$
5. $\quad$ Let $C = \{I_1, \ldots, I_c\}$ be a candidate set of random test inputs of SF
6. $\quad$ Let $CC = \{S_1, \ldots, S_c\}$ be the state coverage reports of executing SF with I_1 to I_c
7. $\quad$ **for each** $S_i \in CC$ **do:**
8. $\qquad AddCov = |S_i - \cup_{S' \in TSC} S'|$
9. $\qquad$ **if** $AddCov > MaxAddCov$ **:**
10. $\qquad\quad MaxAddCov = AddCov$
11. $\qquad\quad P = S_i, J = I_i$
12. $\quad$ **if** $MaxAddCov = 0$ **:**
13. $\qquad$ Let $P = S_j$, $J = I_j$ where $S_j \in CC$ and $|S_j| = \text{MAX}_{S' \in CC} |S'|$
14. $\quad TSC = TSC \cup P$, $TS = TS \cup J$
15. return TS

Figure 5: The state coverage (SC) selection algorithm. The algorithm for transition coverage, TC, is obtained by replacing S (state coverage report) with T (transition coverage report).

Algorithm. OD-SELECTION
Input: Stateflow model SF.
Output: Test suite $TS = \{J_1, \ldots, J_q\}$.

1. Let $TS = \{I\}$ where I is a random test input of SF
2. Let $TSO = \{sg_o\}$ where sg_o is the output signal of executing SF with I
3. **for** $q-1$ times **do:**
4. $\quad MaxDist = 0$
5. $\quad$ Let $C = \{I_1, \ldots, I_c\}$ be a candidate set of random test inputs of SF
6. $\quad$ Let $CO = \{sg_1, \ldots, sg_c\}$ be the output signals of executing SF with I_1 to I_c
7. $\quad$ **for each** $sg_i \in CO$ **do:**
8. $\qquad Dist = \text{MIN}_{\forall sg' \in TSO} dist_o(sg_i, sg')$
9. $\qquad$ **if** $Dist > MaxDist$ **:**
10. $\qquad\quad MaxDist = Dist$
11. $\qquad\quad p = sg_i, J = I_i$
12 $\quad TSO = TSO \cup p$, $TS = TS \cup J$
13. return TS

Figure 6: The output diversity test selection algorithm (OD).

additional coverage is selected and added to TS (line 14). More precisely, a test input I brings about maximum additional coverage, if, compared to other c test input candidates, it covers the most number of states that are not already covered by the test suite TS. Note that if none of the c candidates yields an additional coverage, i.e., $MaxAddCov$ is 0 at line 12, we pick a test input with the maximum coverage among the c candidates (line 13).

3.3 Output Diversity Test Selection

The output diversity (OD) algorithm aims to generate a test suite TS such that the diversity among continuous output signals produced by different test inputs in TS is maximized [2]. In order to formalize this algorithm, we define a measure of diversity ($dist_o$) between pairs of control output signals (sg_o, sg'_o). Specifically, we define the diversity between sg_o and sg'_o based on normalized Euclidean distance and as defined by Equation 1 (i.e., $dist_o(sg_o, sg'_o) = \hat{dist}(sg_o, sg'_o)$).

Figure 6 shows the OD algorithm, i.e., OD-SELECTION. The algorithm first selects a random test input I and simulates SF using I. It adds I to TS (line 1) and the output corresponding to I to another set TSO (line 2). Then, at each iteration, the algorithm first randomly generates c candidate test inputs (line 5) together with their corresponding test outputs and store the outputs in set CO (line 6). Then, in line 8, it uses $dist_o$ to compute the distance between each test output sg_i in CO and the test outputs corresponding to the existing test inputs in TS. Among the test outputs in CO, the algorithm keeps the one with the highest distance from the test outputs in TSO (line 11), and adds such a test output to TSO and its corresponding test input to TS (line 12).

3.4 Failure-based Test Selection

The goal of failure-based test selection algorithms is to generate test inputs that are likely to produce output signals exhibiting specific failure patterns. We develop these algorithms using meta-heuristic search algorithms [21] that generate test inputs maximizing the likelihood of presence of failures in outputs.

We propose two failure-based test selection algorithms, output stability and output continuity that respectively correspond to instability and discontinuity failure patterns introduced in Section 2. We first provide two heuristic (quantitative) objective functions that estimate the likelihood for each of these failure patterns to be present in control signals. We then provide selection algorithms that guide the search to identify test inputs that maximize these objective functions, and hence, are more likely to reveal faults.

Output stability. Given an output signal sg_o, we define the function $stability(sg_o)$ as the sum of the differences of signal values for consecutive simulation steps:

$$stability(sg_o) = \sum_{i=1}^{k} |sg_o(i \cdot \Delta t) - sg_o((i-1) \cdot \Delta t)|$$

Specifically, function $stability(sg_o)$ provides a quantitative approximation of the degree of instability of sg_o. The higher the value of the $stability$ function for a signal sg_o, the more certain we can be that sg_o exhibits some instability failure. For example, the value of the $stability$ function applied to the signal in Figure 3(a) is higher than that of the $stability$ function applied to the signal in Figure 3(b) since, due to oscillations in the former signal, the values of $|sg_o(i \cdot \Delta t) - sg_o((i-1) \cdot \Delta t)|$ are larger than those values for the latter signal.

Output continuity. As discussed earlier, control signals, at each simulation step, are expected to be either left-continuous or right-continuous, or both. We define a heuristic objective function to identify signals that are neither left-continuous nor right-continuous at some simulation step. Since in our work simulation time steps (Δt) are not infinitesimal, we cannot compute derivatives for signals, and instead, we rely on discrete change rates that approximate derivatives when time differences of observable changes cannot be arbitrarily small. Given an output signal sg_o, let $lc_i = \frac{|sg_o(i \cdot \Delta t) - sg_o((i-dt) \cdot \Delta t)|}{\Delta t}$ be the left change rate at step i, and let $rc_i = \frac{|sg_o((i+dt) \cdot \Delta t) - sg_o(i \cdot \Delta t)|}{\Delta t}$ be the right change rate at step i. We define the function $continuity(sg_o)$ as the maximum of the minimum of the left and the right change rates at each simulation step over all the observed simulation steps:

$$continuity(sg_o) = \max_{dt=1}^{3}(\max_{i=dt}^{k-dt}(\min(lc_i, rc_i))))$$

Specifically, we first choose a value for dt indicating the maximum expected time duration of a spike. Then for a fixed dt, for every step i such that $dt \leq i \leq k - dt$, we take the minimum of the left change rate and the right change rate at step i. Since we expect the signal to be either left-continuous or right-continuous, at least one of the right or left change rates should be a small value. We then compute the maximum of all the minimum right or left change rates for all the simulation steps to find a simulation step with the highest discontinuity from both left and right sides. Finally, we obtain the maximum value across the time intervals up to length dt. For our work, we pick dt to be between 1 and 3. For example, the signal in Figure 3(b) yields high right and left change rates at point A. As a result, function $continuity$ produces a high value for this signal, indicating that this signal is likely to be discontinuous. In contrast, the value of function $continuity$ for the signal in Figures 2 is lower than that in Figure 3(b) because at every simulation step, either the right change rate or the left change rate yields a relatively low value.

As discussed earlier, we provide a meta-heuristic search algorithm to generate test suites based on our failure patterns. Specifically, we use the *Hill-Climbing with Random Restarts (HCRR)*

Algorithm. OS-SELECTION

Input: Stateflow model SF.
Output: Test suite $TS = \{J_1, \ldots, J_q\}$.

```
1.  Let I be a random test input of SF and sg_o the output of executing SF with I
2.  Let All = {I}
3.  highestFound = stability(sg_o)
4.  for (q − 1) ∗ c iterations do:
5.     newI = Tweak(I)
6.     Let sg_o be the output of executing SF with newI
7.     All = All ∪ {newI}
8.     if stability(sg_o) > highestFound :
9.        highestFound = stability(sg_o)
10.       I = newI
11.    if TimeToRestart() :
12.       Let I be a random test input of SF and sg_o the output of executing SF with I
13.       highestFound = stability(sg_o)
14.       All = All ∪ {I}
15. Let TS be the test inputs in All with the q-highest values of stability function
16. return TS
```

Figure 7: The test selection algorithm based on output stability. The algorithm for output continuity, OC-SELECTION, is obtained by replacing $stability(sg_o)$ with $continuity(sg_o)$.

algorithm [21]. In our earlier work on computing test cases violating stability, smoothness, and responsiveness requirements for closed-loop controllers [25], HCRR performed best among a number of alternative single-state search heuristics. Figure 7 shows our output stability test selection algorithm, OS-SELECTION, based on HCRR. The algorithm for output continuity, OC-SELECTION, is obtained by replacing $stability(sg_o)$ with $continuity(sg_o)$ in OS-SELECTION. At each iteration, the algorithm *tweaks* the current solution, i.e., the test input, to generate a new solution, i.e., a new test input, and *replaces* the current solution with the new solution if the latter has higher value for the objective function. Similar to standard Hill-Climbing, the HCRR algorithm includes a $Tweak$ operator that shifts a test input I in the input space by adding values selected from a normal distribution with mean $\mu = 0$ and variance σ^2 to the values characterizing the input signals (line 5), and a replace mechanism (lines 8-10) that replaces I with $newI$, if $newI$ has a higher objective function value. In addition, HCRR restarts the search from time to time by replacing I with a randomly selected test input (lines 11-13). We run the algorithm for $(q-1) * c$ iterations where q is the size of the test suites, and c is the size of candidate sets in the greedy selection algorithms in Figures 4 to 6. This is to ensure that OC-SELECTION spends the same test execution budget as the other selection algorithms. The OC-SELECTION algorithm keeps all the test inputs generated during the execution in a set All (lines 2, 7 and 14). At the end of the algorithm, from the set All, we pick q test inputs that have the highest objective function values (line 15) and return them as the selected test suite.

4. EXPERIMENT SETUP

In this section, we present the research questions, and describe our study subjects, our metric to measure fault revealing ability of different selection algorithms, and our experiment design.

4.1 Research Questions

RQ1 (Fault Revealing Ability). *How does the fault revealing ability of our proposed test selection algorithms compare with one another?* We start by comparing the ability of the test suites generated using the different test selection algorithms discussed in Section 3 in revealing faults in Stateflow models. In particular, we are interested to know (1) if our selection algorithms outperform input diversity (baseline)? and (2) if there is any selection algorithm that consistently reveals the most faults across different study subjects and different fault types?

RQ2 (Fault Revealing Subsumption) *Is any of our selection algorithms subsumed by other algorithms? or for each selection algorithm, are there some faults that can be found by that algorithm, but not by others?* This question investigates if any of the selection algorithms discussed in Section 3 is subsumed by other algorithms, i.e., if any selection algorithm does not find any additional faults missed by other algorithms.

RQ3 (Fault Revealing Complementarity). *What is the impact of different failure types on fault revealing ability of our test selection algorithms?* This question investigates whether any of our selection algorithms has a tendency to reveal a certain type of failures better than others. This shows whether our selection algorithms are complementary to each other. That is, they reveal different types of failures, thus suggesting they may be combined.

RQ4 (Test Suite Size). *What is the impact of the size of test suites generated by our selection algorithms on their fault revealing ability?* With this question, we study the impact of size on fault revealing ability of test suites, and investigate whether some selection algorithms already perform well with small test suite sizes, while some may require to enlarge test suites to better reveal faults.

Table 1: Characteristics of our study subject Stateflow models.

Name	Publicly Available	No. of Inputs	No. of States	No. of Transitions	Hierarchical States	Parallel States
SCC	No	23	13	25	2	No
ASS	No	42	16	53	1	No
GCS	Yes	8	10	27	0	Yes

4.2 Study Subjects

We use three Stateflow models in our experiments: Two industrial models from Delphi, namely, SCC (discussed in Section 2) and Auto Start-Stop Control (ASS); and one public domain model from Mathworks website [41], (i.e., Guidance Control System (GCS)). Table 1 shows key characteristics of these models. All of these three models have a continuous control output signal. Specifically, the continuous control signal in SCC controls the clutch position, in ASS, it controls the engine torque, and in GCS, it controls the position of a missile. These models have a large number of input variables. SCC and ASS have hierarchical states (OR states) and GCS is a parallel state machine. The number of states and transitions reported in Table 1 are those obtained after model flattenning.

We note that our industrial subject models are representative in terms of the size and complexity among Stateflow models developed at Delphi. The number of input variables, transitions and states of our industrial models is notably more than that of the public domain models from Mathworks [44]. Further, most public domain Stateflows are small exemplars created for the purpose of training and are not representative of the models developed in industry. Specifically, while discrete-continuous controllers are very common in many embedded industry sectors, among the models available at [44], only GCS was a discrete-continuous Stateflow controller and had a continuous control signal, and hence, we chose it for our experiment. But since GCS continuous behavior was too trivial, we modified it before using it in our experiments by adding some configuration parameters and some difference equations in some states. We have made the modified version available at [23].

4.3 Measuring Fault Revealing Ability

In our study, we measure the fault revealing ability of test suites generated by different selection algorithms. To automate our experiments, we use fault-free versions of our subject models to generate test oracles (i.e, the ground truth oracle [5]). Let TS be a test suite generated by one of our selection algorithms and for a given (faulty) model SF. For the purpose of this experiment, we assume that SF contains a single fault only. We measure the ability of TS

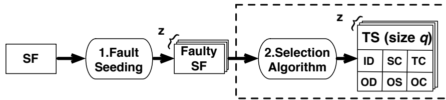

Figure 8: Our experiment design: Step 2 was repeated for 100 times due to the randomness in our selection algorithms.

in revealing the fault in SF using a boolean measure. Our measure returns true if there exists at least one test input in TS for which the output of SF sufficiently deviates from the grand truth oracle such that a manual tester conclusively detects a failure. Otherwise, our measure returns false. Formally, let $O = \{sg_1, \ldots, sg_q\}$ be the set of output signals obtained by running SF for the test inputs in $TS = \{I_1, \ldots, I_q\}$, and let $G = \{g_1, \ldots, g_q\}$ be the corresponding test oracle signals. The fault revealing rate, denoted by FRR, is computed as follows:

$$(1)\ FRR(SF, TS) = \begin{cases} 1 & \exists_{1 \leq i \leq q}\ \hat{dist}(sg_i, g_i) > THR \\ 0 & \forall_{1 \leq i \leq q}\ \hat{dist}(sg_i, g_i) <= THR \end{cases}$$

where $\hat{dist}(sg_i, g_i)$ is defined by Equation 1, and THR is a given threshold. If we set THR to zero, then a test suite detects a given fault (i.e., $FRR = 1$), if it is able to generate at least one output that deviates from the oracle irrespective of the amount of deviation. For continuous dynamic systems, however, the system output is acceptable when the deviation is small and not necessarily zero. Furthermore, for such systems, it is more likely that manual testers recognize a faulty output signal when the signal shape drastically differs from the oracle. In our work, we set THR to 0.2. As a result, a test suite detects a given fault (i.e., $FRR = 1$), if it is able to generate at least one output that diverges from the oracle such that the distance between the oracle and the faulty output is more than 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 $THR = 0.25$ and $THS = 0.15$ and showed that our results were not sensitive to such small changes in THR.

4.4 Experiment Design

Figure 8 shows the overall structure of our experiments consisting of the following two steps:

Step1: Fault Seeding. We asked a Delphi engineer to seed 30 faults in each one of our two industry subject models (z = 30), generating 30 faulty versions of SCC and ASS, i.e, one fault per each faulty version. The faults were seeded before our experiments took place. The engineer was asked to choose the faults based on his experience in Stateflow model development and debugging. In addition, we required the faults to be seeded in different parts of the Stateflow models and to be of different types. We categorize the seeded faults into two groups: (1) *Wrong Output Computation* which indicates a mistake in the equations computing the continuous control output, e.g., replacing a min function with a max function or a $-$ operator with a $+$ operator in the equations. (2) *Wrong Stateflow Structure* which indicates a mistake in the Stateflow structure, such as wrong transition conditions or wrong priorities of the transitions from the same source. As for the publicly available model (GCS), since it was smaller and less complex than the Delphi models, we seeded 15 faults into the model to create 15 faulty versions (z = 15). Among all the faulty models for each case study, around 40% and 60% of the faults belong to the wrong output computation and wrong Stateflow structure categories, respectively.

Step2: Test Case Selection. As shown in Figure 8, after seeding faults, for each faulty model, we ran our six selection algorithms, namely Input Diversity (ID), State Coverage (SC), Transition Coverage (TC), Output Diversity (OD), Output Stability (OS),

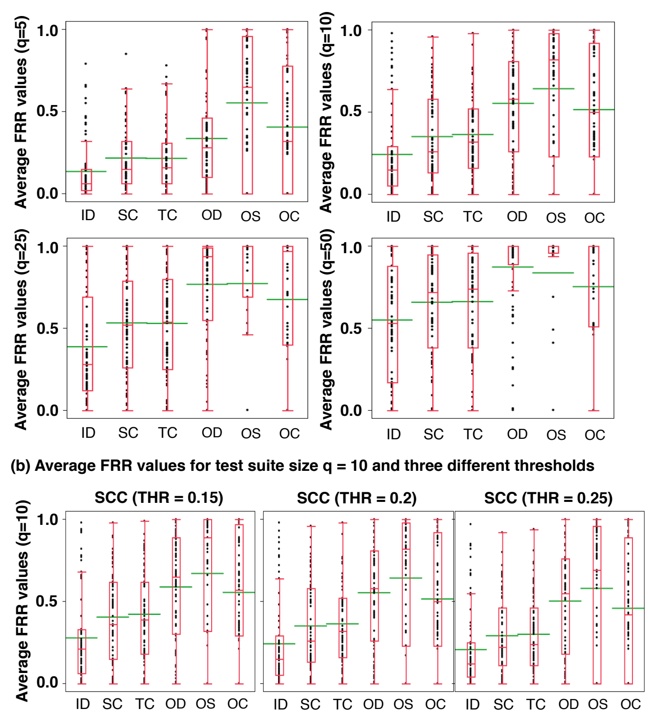

Figure 9: Boxplots comparing fault revealing abilities of our test selection algorithms for different test suite sizes and different thresholds.

and Output Continuity (OC) test selection algorithms. For each faulty model and each selection algorithm, we created a test suite of size q where q took the following values: 3, 5, 10, 25, and 50. We repeated the test selection step of our algorithm for 100 times to account for the randomness in the selection algorithms. In summary, we created 75 faulty models (30 versions for SCC and ASS, and 15 versions of GCS). For each faulty model and for each selection algorithm, we created five different test suites with sizes 3, 5, 10, 25 and 50. That is, we sampled 2250 different test suites and repeated each sampling for a 100 times (i.e., in total, 225,000 different test suites were generated for our experiment). Overall, our experiment took about 1600 hours time on a notebook with a 2.4GHz i7 CPU, 8 GB RAM, and 128 GB SSD.

5. RESULTS AND DISCUSSIONS

This section provides responses, based on our experiment design, for research questions **RQ1** to **RQ4** described in Section 4.

RQ1 (Fault Revealing Ability). To answer **RQ1**, we ran the experiment in Figure 8 with test suite sizes q = 5, 10, 25, and 50, and for all the 75 faulty models (i.e., $z = 30$ for SCC, $z = 30$ for ASS, and $z = 15$ for GCS). We computed the fault revealing rates FRR with three thresholds THR=0.2, 0.15 and 0.25. Figure 9(a) shows four plots comparing the fault revealing ability of the test selection algorithms discussed in Section 3 with THR=0.2. Each plot in Figure 9(a) compares six distributions corresponding to our six test selection algorithms. Each distribution consists of 75 points. Each point relates to one faulty model, and represents the average fault revealing ability of the 100 different test suites with a fixed size and obtained by applying one of our test selection algorithms to that faulty model. For example, a point with (x = SC) and (y = 0.32) in the (q = 5) plot of Figure 9(a) indicates that among the 100 different test suites with size 5 generated by applying SC to one faulty model, 32 test suites were able to reveal the fault (i.e.,

$FRR = 1$) and 68 could not reveal that fault (i.e., $FRR = 0$).

To statistically compare the fault revealing ability of different selection algorithms, we performed the non-parametric pairwise Wilcoxon Pairs Signed Ranks test [8], and calculated the effect size using Cohen's d [12]. The level of significance (α) was set to 0.05, and, following standard practice, d was labeled "small" for $0.2 \leq d < 0.5$, "medium" for $0.5 \leq d < 0.8$, and "high" for $d \geq 0.8$ [12].

Comparison with Input Diversity. Testing differences in FRR distributions with THR=0.2 shows that, for all the test suite sizes, all the test selection algorithms perform significantly better than ID. In addition, for all the test suite sizes, the effect size is "high" for OD, OS and OC, and "medium" for SC and TC.

Coverage achieved by coverage-based algorithms. In our experiments, on average for the 100 different test suites obtained by SC/TC selection algorithms and for our three subject models, we achieved 81/65%, 88/71%, 93/76% and 97/81% state/transition coverage for the test suites with size 5, 10, 25 and 50, respectively. Further, we noticed that the largest test suites generated by our coverage-based selection algorithms (i.e., $q = 50$) were able to execute the faulty states or transitions of 73 out of the 75 faulty models.

Comparing output-based and coverage-based algorithms. For all the test suite sizes, statistical test results indicate that OD, OS, and OC perform significantly better than SC and TC. For OS and for all the test suite sizes, the effect size is "high". For OD with all the test suite sizes except for $q = 50$, the effect size is "medium", and for $q = 50$, the effect size is "high". For OC with all the test suite sizes except for $q = 50$, the effect size is "medium", and for $q = 50$, the effect size is "low".

Comparing output-based algorithms. For $q = 5$ and 10, OS is significantly better than OD and OC with effect sizes of "medium" (for q =5) and "low" (for q =10). However, neither of OC and OD is better than the other for q =5 and 10. For q =25, OS is better than OD with a "low" effect size, with no significant difference between OS and OC or OC and OD. Finally, for $q = 50$, there is no significant difference between OS, OC and OD.

Modifying THR. The above statistical test results were consistent with those obtained based on FRR values computed with THR=0.25 and 0.15. As an example, Figure 9(b) shows average FRR values for q=10, for THR=0.15, 0.2 and 0.15. Increasing the threshold from 0.15 to 0.25 decreases the FRR values but, however, does not change the relative differences in FRR values across different selection algorithms.

In summary, the answer to **RQ1** is that the test suites generated by OD, OS, OC, SC, and TC, have significantly higher fault revealing ability than those generated by ID. Further, even though coverage-based algorithms (SC and TC) were able to achieve a high coverage and execute the faulty states or transitions of 73 faulty models, the failure-based and output diversity algorithms (OS, OC, and OD) generate test suites with significantly higher fault revealing ability compared to those generated by SC and TC. For smaller test suites ($q < 25$), OS performs better than OC and OD, while for $q = 50$, we did not observe any significant differences among the failure-based and output diversity algorithms (OS, OC, and OD). Finally, our results are not impacted by small modifications in the threshold values used to compute the fault revealing measure FRR.

RQ2 (Fault Revealing Subsumption). To answer **RQ2**, we consider the results of the experiment in Figure 9(a). We applied the Wilcoxon test to identify, for each of the 75 faulty models, which selection algorithm yielded the highest fault revealing rate (i.e., the highest average FRR over 100 runs). Figure 10 and Table 2 show the results. Figure 10 shows which algorithms are best in finding each of the 75 faults (30 for SCC, 30 for ASS, and 15 for GCS) for each test suite size (q = 5, 10, 25 and 50). In this figure, an algorithm A is marked as best for a fault F (denoted by •), if, based on the Wilcoxon test results for F, there is no other algorithm that is significantly better than A in revealing F. Table 2 shows two numbers I/E for each algorithm and for each test suite size. Specifically, given a pair I/E for an algorithm A, I indicates the number of faults that are best found by A and possibly by some other algorithms (i.e., inclusively found by A), while E indicates the number of faults that are best found by A only (i.e., exclusively found by A). For example, when the test suite size is 5, OD is among the best algorithms in finding 20 faults, and among these 20 faults, OD is the only best algorithm for 8 faults.

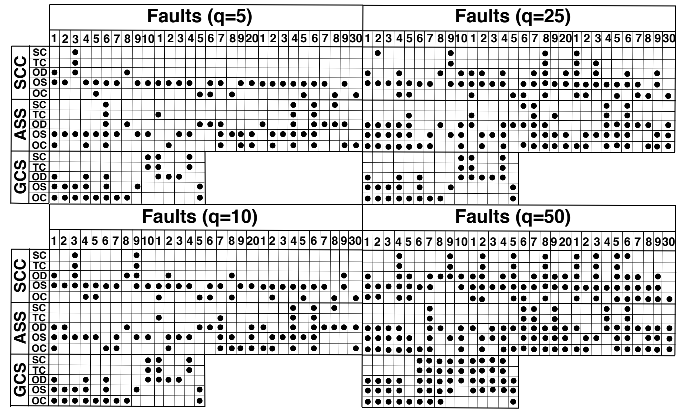

Figure 10: The best selection algorithm(s) for each of the 75 faulty models.

Table 2: The number of faults (out of 75) found inclusively (I) and exclusively (E) by each algorithm and for each test suite size.

	SC (I / E)	TC (I / E)	OD (I / E)	OS (I / E)	OC (I / E)
q=5	8 / 0	8/1	20 / 8	51 / 32	28 / 8
q=10	8 / 0	9/1	28 / 8	51 / 24	32 / 8
q=25	11 / 0	13/0	41 / 6	55 / 10	44 / 7
q=50	24 / 0	23 / 0	59 / 3	63 / 7	50 / 2

Coverage algorithms. As shown in Table 2, SC is subsumed by the other algorithms for every test suite size (E = 0). That is, SC does not find any fault exclusively, and any fault found by SC is also found with the same or higher probability by some other algorithm. TC is able to find one fault exclusively for $q < 25$, but is subsumed by other algorithms for $q \geq 25$. Further, based on Figure 10, SC and TC together are able to find three faults exclusively for $q = 5$ (11 of ASS, and 10 and 14 of GCS), and two faults exclusively for $q = 10$ (11 of ASS, and 14 of GCS). However, for $q \geq 25$, they are subsumed by OD.

Output-based algorithms. As shown in Table 2, OS fares best as it finds the most number of faults both inclusively and exclusively for different values of q. In contrast, OD shows the highest growth in the number of inclusively and exclusively found faults as q increases compared to OS and OC.

In summary, the answer to **RQ2** is that coverage algorithms find the least number of faults both exclusively and inclusively, and as test suite size increases, these algorithms are subsumed by the output diversity (OD) algorithm. The output-based algorithms are complementary (i.e., are not subsumed by one another) and while output stability (OS) finds the highest number of faults both inclusively and exclusively, output diversity (OD) shows the highest

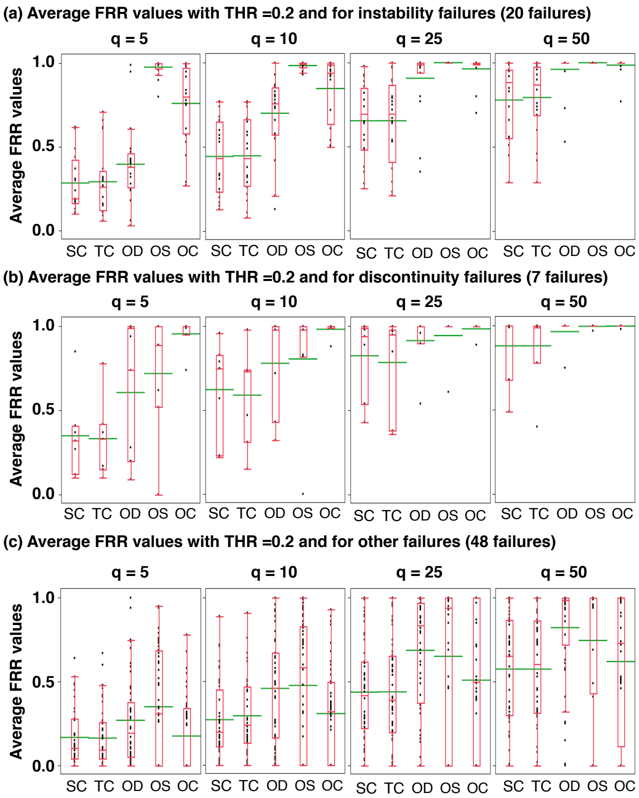

Figure 11: The average FRR values for different types of failures and for different test suite sizes.

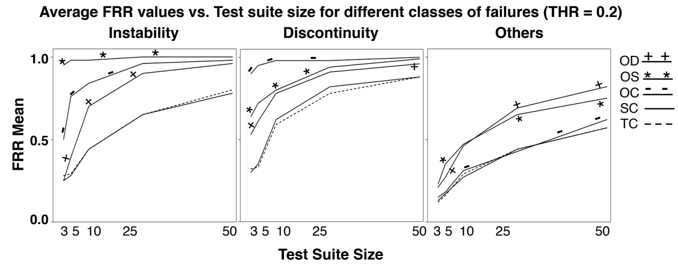

Figure 12: The impact of test suite size on the average FRR over 100 test suites of different faulty models.

improvement in fault finding as the test suite size increases.

RQ3 (Fault Revealing Complementarity). To answer **RQ3**, we first divide the 75 faulty models in our experiments based on the failure type that they exhibit. To determine the failure type exhibited by a faulty model, we inspect the output that yields the highest FRR among the outputs produced by the test suites related to that model. We identified three types of failures in these outputs and divided the 75 faulty models into the following three groups: (1) the faulty models exhibiting instability failure (20 models), (2) the faulty models exhibiting discontinuity failure (7 models), and (3) the other models that neither show instability nor discontinuity (48 models). Figures 11(a) to (c) compare the fault revealing ability of our test selection algorithms for test suite sizes q = 5, 10, 25, and 50 and for each of the above three categories of failures (i.e., instability, discontinuity, and other).

Instability and discontinuity. The statistical test results show that, for the instability failure, OS has the highest fault revealing rate for $q = 5$, 10, and 25. Similarly for the discontinuity failure, OC has the highest fault revealing rate for $q = 5$ and 10. However, for larger test suites (q = 50 for instability, and q = 25 and 50 for discontinuity), OS, OC and OD are equally good at finding the instability and discontinuity failures.

Other. As for the "other" failures, OS and OD are better able to find these failures compared to other algorithms for q = 5, 10, and 50. For q = 25, there is no significant difference between OS, OD and OC in revealing these failures. However, as shown in Figure 11(c), for q = 50, the FRR value distribution for OD has the highest average compared to other algorithms. Further, the variance of FRR values for OD in Figure 11(c) with q = 50 is the lowest, making OD the best algorithm for finding failures other than instability and discontinuity when large test suites are available.

In summary, the answer to **RQ3** is that when test suites are small, OS and OC show a clear tendency to, respectively, reveal the instability and discontinuity failures better than other types of failures and better than other algorithms. With large test suites, however, OS, OC and OD are equally good at finding the instability and discontinuity failures. Further, with small test suites, OS and OD are better than other algorithms in revealing failures other than instability and discontinuity. For large test suites, however, OD shows a tendency to perform better for the "other" types of failures since by diversifying outputs it increases the chances of finding failures not following any specific pattern.

RQ4 (Test Suite Size). To answer **RQ4**, we extended the experiment in Figure 9 to include $q = 3$, as well. Figure 12 shows how the average of FRR over 100 test suites for different faulty models and in different failure groups is impacted by increasing the test suite size. Specifically, for Figure 12, the 75 faulty models are divided based on the failure type they exhibit (20 for instability, 7 for discontinuity, and 48 for others).

According to Figure 12, OS performs best in revealing instability failures even with small test suite sizes where its average FRR is very close to one (0.97). Similarly, OC performs best in revealing discontinuity failures and its average FRR for small test suites is already very high, i.e., 0.95. For "other" kinds of failures, OS performs best for very small test suites, but for $q \geq 10$, OD performs the best. Finally, for instability and discontinuity, OS, OD and OC perform better than SC and TC for all test suite sizes, while for other failures, OS and OD perform better than OC, SC and TC for all the test suite sizes.

In summary, the answer to **RQ4** is that the fault revealing ability of OS (respectively, OC) for instability (respectively, discontinuity) failures is very high for small test suites and almost equal to the highest possible fault revealing rate value. For failures other than instability and discontinuity, the ability of OD in revealing failures rapidly improves as the test suite size increases, making OD the best algorithm for such failures for test suite sizes more than or equal to 10.

Discussion. We present our observations as to why the coverage algorithms are less effective than the output-based algorithms for generating test suites for mixed discrete-continuous Stateflows. Further, we outline our future research direction on effective combination of our output-based test selection algorithms.

Why coverage algorithms are less effective? Overall, our results show that, compared to output-based algorithms, coverage algorithms are less effective in revealing Stateflow faults, and as discussed in **RQ2**, they are subsumed by the output diversity algorithm. Based on our experiments, even though test suites generated by SC and TC cover the faulty parts of the Stateflow models, they fail to generate output signals that are *sufficiently* distinct from the oracle signal, hence yielding a low fault revealing rate. That is, a discrete notion of state or transition coverage does not help reveal continuous output failures. Note that these failures depend on the value changes of outputs over a continuous time interval. The poor performance of coverage algorithms might be due to the fact that state and transition coverage criteria do not account for the time duration spent at each state or for the time instance at which a transition is triggered. For example, an objective to cover states while trying to reduce the amount of time spent in each state may better help reveal discontinuity failures (see Figure 3(b)).

Combining output-based selection algorithms. Our results show that for large test suites and for *all* failure types, the fault revealing ability of OS, OC and OD are the same with an average FRR of 0.75 to 0.87. However, for smaller test suites and for *specific* failures, some algorithms (i.e., OS for instability and OC for discontinuity) perform remarkably well with an average FRR higher than 0.95. This essentially eliminates the need to use large test suites for those specific failure types. These findings offer the potential for engineers to combine our output-based algorithms to achieve a small test suite with a high fault revealing rate. Recall that test oracles for mixed discrete-continuous Stateflows are manual, and hence the test suite size has to be kept as low as possible (typically $q \leq 100$). For example, given our results on fault revealing ability of OS, OC, and OD, and assuming that a test suite size budget of q is provided, we may allocate a small percentage of the test suite size to OC to find discontinuity failures, and share the rest of the budget between OS and OD by giving OD a higher share. This is because OS is able to find instability failures with small test suites, but also, it performs well at finding other failures. However, only OD was able to subsume SC/TC with large test suites, and given that OD's performance increases with the test suite size, a larger test suite size might be allocated to OD. This suggests future work to investigate guidelines on dividing the test suite size budget across different output-based test selection algorithms.

6. RELATED WORK

What distinguishes our work from the existing model-based testing approaches is that in our work, Stateflow models are the artifacts under test, while in model-based testing, test cases are derived from Stateflow models or other state machine variants in order to exercise the system on its target platform. These model-based testing approaches often generate test cases from models using various automation mechanisms, e.g., search-based techniques [50], model checking [27, 42], guided random testing [37, 33, 11] or a combination of these techniques [36, 30]. In [31], a model-based testing approach for mixed discrete-continuous Stateflow models is proposed where test inputs are generated based on discrete fragments of Stateflow models, and are applied to the original models to obtain test oracles in terms of continuous signals. In all these approaches, Stateflow models, in addition to generating test cases, are used to automate test oracles. In reality, however, Stateflows might be faulty and cannot be used to automate test oracles. Hence, we focus on generating small and minimal test suites to identify faults in complex, executable Stateflow models for which automated test oracles are not available, a common situation in industry.

Formal methods and model checking techniques [42, 34, 10] have been previously applied to verify Stateflow models. These approaches largely focus on Stateflows with discrete behaviours, and attempt to maximize (discrete) state or transition coverage. In our work, we test Stateflows with mixed discrete-continuous. Further, our results show that, for discrete-continuous Stateflows, test inputs that cover faulty parts of Stateflow models may not be able to reveal faults (i.e., may not yield outputs with sufficient distance from the oracle). Hence, focusing on coverage alone may not result in test suites with high fault revealing ability.

We proposed two failure patterns capturing specific errors in continuous outputs of Stateflow models. Our failure patterns are analogous to the notion of fault models [32]. Fault models provide abstract descriptions for specific things that can go wrong in a certain domain [32]. Several examples of fault models have been proposed (e.g., for access control policies [22], or for specific concurrency or security faults [7, 32]). Our work is the first to define such notion for mixed discrete-continuous Stateflows and apply it using meta-heuristic search. Our failure patterns capture the intuition of domain experts, and are defined over continuous controller outputs. We further note that instability and discontinuity patterns are, respectively, similar to the accuracy and change rate properties that are typically used to characterize physical behaviour of cyber physical software controllers [16].

Our output diversity selection algorithm is inspired by the output uniqueness criterion that has been proposed and evaluated in the context of web application testing [1, 2], and has shown to be a useful surrogate to white-box coverage selection criteria [2]. However, while in [1, 2], output uniqueness is characterized based on the textual, visual or structural aspects of HTML code, in our work, we define output diversity as Euclidean distance between pairs of continuous output signals and apply it to Stateflow models.

Several approaches to test input generation, when test inputs are discrete, rely on techniques such as symbolic execution, constraint solvers or model checkers (e.g., [47]). In our coverage-based algorithms (SC and TC), we used adaptive random test input generation because our test inputs are signals. As discussed in Section 5, with our adaptive random strategy, SC and TC were able to achieve a high coverage despite small test suite sizes. Adapting symbolic techniques to generate test input signals is left for future work.

7. CONCLUSIONS

Embedded software controllers are largely developed using discrete-continuous Stateflows. To reduce the cost of manual test oracles associated with Stateflow models, test case selection algorithms are required. These algorithms aim at providing minimal test suites with high fault revealing power. We proposed and evaluated six test selection algorithms for discrete-continuous Stateflows: three output-based (OD, OS, OC), two coverage-based (SC, TC), and one input-based (ID). Our experiments based on two industrial and one public domain Stateflow models showed that the output-based algorithms consistently outperform the coverage-based algorithms in revealing faults in mixed discrete-continuous Stateflows. Further, for test suites larger than 25, the output-based algorithms were able to find with the same or higher probability all the faults revealed by the coverage-based algorithms, and hence subsumed them. In addition, OS and OC selection algorithms had very high fault revealing rates, even with small test suites, for instability and discontinuity failures, respectively. For the other failures, OD outperformed the other algorithms in finding faults for test suite sizes larger than 10, and further, its fault detection rate kept improving at a faster rate than the others when increasing the test suite size.

In future, we will seek to develop optimal guidelines on dividing test oracle budget across our output-based selection algorithms. Further, we intend to apply our test selection algorithms to models consisting of Simulink blocks as well as Stateflow models with discrete-continuous behaviors, as Stateflow models are most often embedded in a network of Simulink blocks.

Acknowledgments

We thank Fabrizio Pastore for his useful comments on a draft of this paper. Supported by the Fonds National de la Recherche, Luxembourg (FNR/P10/03 - Verification and Validation Laboratory, and FNR 4878364), and Delphi Automotive Systems, Luxembourg.

8. REFERENCES

[1] N. Alshahwan and M. Harman. Augmenting test suites effectiveness by increasing output diversity. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1345–1348. IEEE Press, 2012.

[2] N. Alshahwan and M. Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 181–192. ACM, 2014.

[3] A. Arcuri and L. Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 265–275. ACM, 2011.

[4] X. Bai, K. Hou, H. Lu, Y. Zhang, L. Hu, and H. Ye. Semantic-based test oracles. In *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*, pages 640–649. IEEE, 2011.

[5] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 2015.

[6] R. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[7] M. Buchler, J. Oudinet, and A. Pretschner. Semi-automatic security testing of web applications from a secure model. In *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*, pages 253–262. IEEE, 2012.

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

[9] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

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

[11] R. Cleaveland, S. A. Smolka, and S. T. Sims. An instrumentation-based approach to controller model validation. In *Model-Driven Development of Reliable Automotive Services*, pages 84–97. Springer, 2008.

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

[13] R. Colgren. *Basic MATLAB, Simulink and Stateflow*. AIAA (American Institute of Aeronautics and Astronautics), 2006.

[14] D. Coppit and J. M. Haddox-Schatz. On the use of specification-based assertions as test oracles. In *Software Engineering Workshop, 2005. 29th Annual IEEE/NASA*, pages 305–314. IEEE, 2005.

[15] M. Gligoric, S. Negara, O. Legunsen, and D. Marinov. An empirical evaluation and comparison of manual and automated test selection. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 361–372. ACM, 2014.

[16] M. P. Heimdahl, L. Duan, A. Murugesan, and S. Rayadurgam. Modeling and requirements on the physical side of cyber-physical systems. In *2nd International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks), 2013*, pages 1–7. IEEE, 2013.

[17] H. Hemmati, A. Arcuri, and L. Briand. Empirical investigation of the effects of test suite properties on similarity-based test case selection. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 327–336. IEEE, 2011.

[18] T. Henzinger. The theory of hybrid automata. In *LICS*, pages 278–292, 1996.

[19] T. Henzinger and J. Sifakis. The embedded systems design challenge. In *FM*, pages 1–15, 2006.

[20] 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.

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

[22] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 667–676. ACM, 2007.

[23] R. Matinnejad. The modified version of GCS Stateflow. `https://drive.google.com/file/d/0B104wPnuxJVhSU44WFlTVE84bmM/view?usp=sharing`. [Online; accessed 10-March-2015].

[24] 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.

[25] 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.

[26] P. McMinn, M. Stevenson, and M. Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *Proceedings of the First International Workshop on Software Test Output Validation*, pages 1–4. ACM, 2010.

[27] S. Mohalik, A. A. Gadkari, A. Yeolekar, K. Shashidhar, and S. Ramesh. Automatic test case generation from simulink/stateflow models using model checking. *Software Testing, Verification and Reliability*, 24(2):155–180, 2014.

[28] 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.

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

[30] 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 *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 308–311. IEEE, 2013.

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

[32] A. Pretschner, D. Holling, R. Eschbach, and M. Gemmar. A generic fault model for quality assurance. In *Model-Driven Engineering Languages and Systems*, pages 87–103. Springer, 2013.

[33] Reactive Systems Inc. Reactis Tester. `http://www.reactive-systems.com/simulink-testing-validation.html`, 2010. [Online; accessed 25-Nov-2013].

[34] Reactive Systems Inc. Reactis Validator. `http://www.reactive-systems.com/simulink-testing-validation.html`, 2010. [Online; accessed 25-Nov-2013].

[35] D. J. Richardson, S. L. Aha, and T. O. O'malley.

Specification-based test oracles for reactive systems. In *Proceedings of the 14th international conference on Software engineering*, pages 105–118. ACM, 1992.

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

[37] M. Satpathy, A. Yeolekar, and S. Ramesh. Randomized directed testing (redirect) for simulink/stateflow models. In *Proceedings of the 8th ACM international conference on Embedded software*, pages 217–226. ACM, 2008.

[38] M. Staats, M. W. Whalen, and M. P. E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 391–400. IEEE, 2011.

[39] T. Stauner. Properties of hybrid systems-a computer science perspective. *Formal Methods in System Design*, 24(3):223–259, 2004.

[40] The MathWorks Inc. Common Modeling Errors Stateflow Can Detect. `http://nl.mathworks.com/help/stateflow/ug/common-modeling-errors-the-debugger-can-detect.html`. [Online; accessed 23-Fev-2015].

[41] The MathWorks Inc. Designing a Guidance System in MATLAB/Simulink. `http://nl.mathworks.com/help/simulink/examples/designing-a-guidance-system-in-matlab-and-simulink.html`. [Online; accessed 23-Fev-2015].

[42] The MathWorks Inc. Simulink Design Verifier. `http://nl.mathworks.com/products/sldesignverifier/?refresh=true`. [Online; accessed 6-May-2015].

[43] The MathWorks Inc. Stateflow. `http://www.mathworks.nl/products/stateflow`. [Online; accessed 23-Fev-2015].

[44] The MathWorks Inc. Stateflow Model Examples. `http://nl.mathworks.com/products/stateflow/model-examples.html`. [Online; accessed 23-Fev-2015].

[45] A. Tiwari. Formal semantics and analysis methods for simulink stateflow models. *Unpublished report, SRI International*, 2002.

[46] A. K. Tyagi. *MATLAB and SIMULINK for Engineers*. Oxford University Press, 2012.

[47] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.

[48] Wikipedia. Open-loop controller. `http://en.wikipedia.org/wiki/Open-loop_controller`. [Online; last accessed 10-Nov-2014].

[49] Wikipedia. Voltage Spike. `http://en.wikipedia.org/wiki/Voltage_spike`. [Online; accessed 23-Fev-2015].

[50] A. Windisch. Search-based test data generation from stateflow statecharts. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1349–1356. ACM, 2010.

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