PhD-FSTC-2016-34
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defense held on 31/08/2016 in Luxembourg
to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG EN INFORMATIQUE

by

REZA MATINNEJAD
Born on 22 September 1983 in Esfahan (Esfahan, Iran)

AUTOMATED TESTING OF SIMULINK/STATEFLOW MODELS IN THE AUTOMOTIVE DOMAIN

DISSERTATION DEFENSE COMMITTEE

PROF. DR. ING. LIONEL BRIAND, Dissertation Supervisor
University of Luxembourg (Luxembourg)

PROF. DR. ING. HOLGER VOOS, Chairman
University of Luxembourg (Luxembourg)

DR. MEHRDAD SABETZADEH, Deputy Chairman
University of Luxembourg (Luxembourg)

PROF. DR. ANTONIO FILIERI, Member
Imperial College London (UK)

DR. ANDREA ARCURI, Member
Scienta, Oslo (Norway)

DR. SHIVA NEJATI, Expert in an advisory capacity (Co-supervisor)
University of Luxembourg (Luxembourg)

Abstract

Context. Simulink/Stateflow is an advanced system modeling platform which is prevalently used in the Cyber Physical Systems domain, e.g., automotive industry, to implement software controllers and *plant*, i.e., environment, models. Testing Simulink models is complex and poses several challenges to research and practice. Simulink models often have mixed discrete-continuous behaviors and their correct behavior crucially depends on time. Inputs and outputs of Simulink models are *signals*, i.e., values evolving over time, rather than discrete values. Further, Simulink models are required to operate satisfactorily for a large variety of hardware configurations. Finally, developing test oracles for Simulink models is challenging, particularly for requirements capturing their continuous aspects. In this dissertation, we focus on testing mixed discrete-continuous aspects of Simulink models, an important, yet not well-studied, problem. Existing Simulink testing techniques are more amenable to testing and verification of logical and state-based properties. Further, they are mostly incompatible with Simulink models containing time-continuous blocks and floating point and non-linear computations. In addition, they often rely on the presence of formal specifications that are expensive and rare in practice, to automate test oracles.

Approach. In this dissertation, we propose a set of approaches based on meta-heuristic search and machine learning techniques to automate testing of software controllers implemented in Simulink. The work presented in this dissertation is motivated by Simulink testing needs at Delphi Automotive Systems, a world leading part supplier to the automotive industry. To address the above-mentioned challenges, we rely on discrete-continuous output signals of Simulink models and provide output-based, black-box test generation techniques to produce test cases with high fault-revealing ability. Our algorithms are black-box, hence compatible with Simulink/Stateflow models in their entirety. Further, we do not rely on the presence of formal specifications to automate test oracles. Specifically, we propose two sets of test generation algorithms for *closed-loop* and *open-loop* controllers implemented in Simulink: (1) For closed-loop controllers, test oracles can be formalized and automated relying on the feedback received from the controlled system or its model. We characterize the desired behavior of closed-loop controllers in a set of common requirements, and then use search to identify the worst-case test scenarios of controllers with respect to each requirement. (2) For open-loop controllers, we cannot automate test oracles since the feedback is not available, and test oracles are manual. Hence, we focus on providing test generation algorithms that develop small test suites with high fault revealing ability. We further provide a test case prioritization algorithm to rank the generated test cases based on their fault revealing ability and lower the manual oracle cost.

Our test generation and prioritization algorithms are evaluated with several industrial and publicly available Simulink models. Specifically, we showed that fault revealing ability of our approach outperforms that of Simulink Design Verifier (SLDV), the only test generation toolbox of Simulink and a well-known commercial Simulink testing tool. In addition, using our approach, we were able to detect several real faults in Simulink models from our industry partner, Delphi, which had not been previously found by manual testing based on domain expertise and existing Simulink testing tools.

Contributions. The main research contributions in this dissertation are:

1. An automated approach for testing closed-loop controllers that characterizes the desired behavior of such controllers in terms of a set of common requirements and combines random

exploration and search to effectively identify the worst-case test scenarios of controllers with respect to each requirement.
2. An automated approach for testing highly configurable closed-loop controllers by accounting for all their feasible configurations and providing strategies to scale the search to large multi-dimensional spaces relying on dimensionality reduction and surrogate modeling
3. A black-box output-based test generation algorithm for open-loop Simulink models which uses search to maximize the likelihood of presence of specific failure patterns (i.e., anti-patterns) in Simulink output signals.
4. A black-box output-based test generation algorithm for open-loop Simulink models that maximizes output diversity to develop small test suites with diverse output signal shapes and, high fault revealing ability.
5. A test case prioritization algorithm that relies on output diversity of the generated test suites as well as the dynamic structural coverage achieved by individual tests to rank test cases and help engineers identify faults faster by inspecting a few test cases.
6. Two test generation tools, namely *CoCoTest* and *SimCoTest*, that respectively implement our test generation approaches for closed-loop and open-loop controllers.

Acknowledgements

Over the last years, a great many people contributed to the success of my PhD project. I would like to express my gratitude to all those who helped me succeed with my work.

I would like to thank my supervisor, Prof. Lionel Briand, for his significant effort to make my research work match the highest academic standards. I am grateful to have had the opportunity to work with and learn from one of the best researchers in the field.

I would like to thank my co-supervisor, Dr. Shiva Nejati, for all her encouragement throughout my PhD project and her advice on how to conduct good research. Her support was undoubtedly crucial for the success of my PhD project.

I would like to thank Delphi Automotive Systesm for the numerous meetings, in which the engineers detailed Delphi systems, and for providing the case study system that was the subject of my empirical studies. In particular, I would like to thank Thomas Bruckmann, Claude Poull and Camile Feyder for providing me with technical assistance and guidance related to the case study systems.

I would like to express my gratitude for all the friendships I have formed with my colleagues in the Software Verification and Validation Lab. My colleagues not only provided me with helpful advice along the way but also contributed to a warm work environment.

Last but not the least, I would like to thank my wife for her patience, support and encouragement that motivated me to keep up the hard work during my PhD. I would also like to thank my parents for their lifelong support and for enabling me to pursue my dreams.

Supported by Delphi Automotive Systems and the Fonds National de la Recherche, Luxembourg (FNR/P10/03 and FNR 4878364).

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface.

CoCoTest Continous Controller Tester.
CPS Cyber-Physical Systems.
CPU Central Processing Unit.

DC Motor Direct Current Motor.

ECUs Electrical Control Units.

FNR Fond National de la Recherche.

HC Hill-Climbing.
HCRR Hill-Climbing with Random Restarts.
HiL Hardware-in-the-Loop.
HPC High Performance Cluster.

MBT Model-Based Testing.
MDE Model-Driven Engineering.
MiL Model-in-the-Loop.

PID Proportional-Integral-Derivative.

SA Simulated-Annealing.
SiL Software-in-the-Loop.
SimCoTest Simulink Controller Tester.
SLDV Simulink Design Verifier.
SUT System Under Test.

Chapter 1

Introduction

1.1 Context

This dissertation presents a set of approaches based on meta-heuristic search and machine learning techniques to automate testing of software controllers and their plant/environment models implemented in Simulink/Stateflow [The MathWorks Inc., 2003b]. The work presented in this dissertation has been done in collaboration with Delphi Automotive Systems [Delphi, 2016], a world leading part supplier company to the automotive industry, based in Luxembourg. Simulink/Stateflow is a system modeling notation which is prevalently used by engineers to develop software controllers in the Cyber-Physical Systems (CPS) domain, e.g., in automotive and avionics industries. In this dissertation, we mainly focus on testing mixed discrete-continuous aspects of Simulink models, an important–yet not well-studied–problem [Briand et al., 2016, Heimdahl et al., 2013, Zander et al., 2012, Pretschner et al., 2007a].

Software development and testing in the CPS domain typically comprises three stages shown in Figure 1.1: (1) Model-in-the-Loop (MiL): At MiL level, the control functions and environment models are developed and tested in a system modeling notation. In many sectors and in particular in the automotive domain, these models are created in Matlab/Simulink/Stateflow (or Simulink for short) from Mathworks [The MathWorks Inc., 2003b]. The focus of MiL testing is to verify the control behavior or logic and explore and compare alternative control design options, (2) Software-in-the-Loop (SiL): At SiL level, the controller model is converted to code (partly auto coded and partly manually). The focus of SiL testing is on controller code which can run on the target platform, and (3) Hardware-in-the-Loop (HiL): At HiL level, the controller software is fully integrated into the final control system (e.g., in automotive domain, the controller software is installed on Electrical Control Units (ECUs)). The main objective of HiL testing is to verify the integration of hardware and software in a more realistic environment.

In this thesis, among the above three levels, we focus on MiL testing. That is, testing Simulink models which implement control functions used in embedded systems. Development and testing at MiL level is relatively fast compared to SiL and HiL as the engineers can quickly modify the controller model and immediately test the system. Furthermore, Simulink model testing is entirely performed in a virtual environment, enabling execution of a large number of test cases where cost and time budget are driven by available CPU time and there is no risk of hardware damage. Finally, test cases obtained

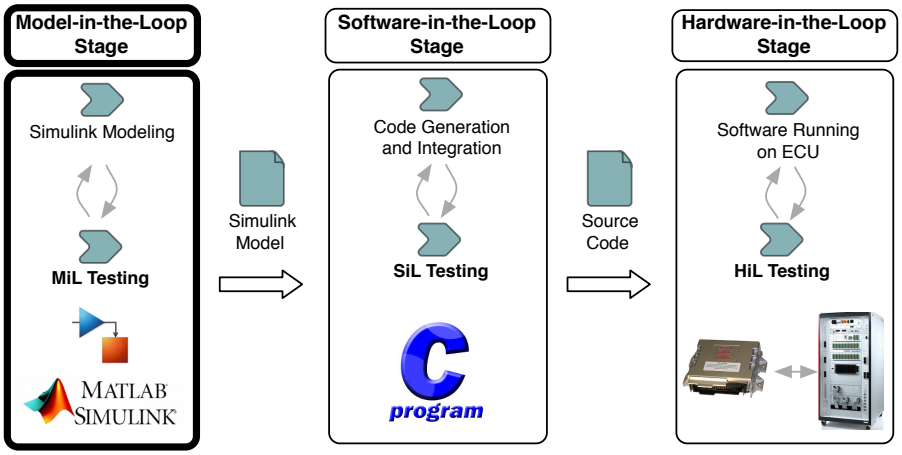


Figure 1.1. Three stages of software development and testing in Cyber Physical Systems domain: Model-in-the-Loop (MiL), Software-in-the-Loop (SiL), and Hardware-in-the-Loop (HiL). In this thesis, we focus on MiL testing.

for Simulink models can later be used at SiL and HiL levels either directly or after some adaptations.

Currently, in most companies, testing Simulink models is limited to executing the models for a small number of simulations, and manually inspecting the results of individual simulations. The simulations are often selected based on the engineers' domain knowledge and experience, but in a rather unsystematic way. Our primary goal in this thesis is to develop automated test generation techniques for testing Simulink/Stateflow models. Such automation should address the following challenges:

1. Simulink models have mixed discrete-continuous behaviours. That is, they often process magnitude-continuous and time-continuous input *signals* (i.e., real-valued functions over time) and produce magnitude-continuous and time-continuous output signals.
2. Simulink models implement *timed* systems. The correct behavior of such systems crucially depends on the time that is required for completion of a particular computation or for a particular physical phenomenon.
3. The embedded software functions implemented by Simulink models are highly configurable and are required to operate satisfactorily on a large variety of hardware/physical devices.
4. Developing *test oracles*, i.e., a procedure that distinguishes between the correct and incorrect behaviors of the software, is challenging. Formalization and automation of test oracles are specifically less studied for requirements capturing continuous aspects of embedded software systems [Briand et al., 2016, Heimdahl et al., 2013, Pretschner et al., 2007a].

1.2 Approach

To address the above challenges, in our work we rely on discrete-continuous output signals of Simulink models to generate test cases with high fault revealing ability. Simulink models have rich discrete-continuous outputs, providing a useful source of information for fault detection. By inspecting output signals, one can determine whether the model output reaches appropriate values at the right times, whether the time period that the model takes to change its values is within acceptable limits, and

whether the signal shape is free of erratic and unexpected changes that violate continuous dynamics of physical processes or objects. In our work, we identified a set of heuristics based on output signals of Simulink models that mimic the engineers' intuition when they look for failures in output signals. We then used search algorithms to generate fault revealing test cases for Simulink models relying on the identified heuristics. Specifically, in this thesis, we propose two separate sets of test generation techniques for *closed-loop* and *open-loop* controllers implemented in Simulink, as follows:

1. Closed-loop controllers are a class of embedded controllers where the goal is to control a system, often called the *plant*, such that its output follows a reference control signal, called the desired output. Closed-loop controllers are typically used when the control behavior of the plant is not precisely known and any disturbance can significantly impact the result of the control process. Figure 1.2(a) shows an abstract view of a closed-loop controller in a feedback loop with a plant model.

   For closed-loop controllers, the plant feedback and the desired controller output are both available. Hence, from the perspective of test generation, test oracles can be formalized and automated in terms of desired output and actual output (feedback). In our approach, we characterize the desired behavior of such controllers in a set of common requirements. We then use search to identify the worst-case test scenarios of the controller with respect to each controller requirement. Figure 1.2(b) shows an example of the controller requirements, called *smoothness*. The smoothness requirement states that the actual output shall not change abruptly when it is close to the desired one. That is, the value of the controller *overshoot/undershoot*, indicated with a solid arrow in Figure 1.2(b), must always remain lower than a threshold value. To find the worst-case test scenario with respect to smoothness requirement, we apply search to find the test scenario with the maximum value of controller overshoot/undershoot. Our approach for testing closed-loop controllers is described in Chapter 3. We further extend this approach to include all the feasible configurations of closed-loop controllers by providing strategies to scale the search to large multi-dimensional spaces in Chapter 4.
2. Open-loop controllers are another class of embedded controllers that control the plant in the absence of environment feedback. Open-loop controllers are typically used when the possible disturbances do not largely impact the control behavior or when it is too costly to implement the feedback mechanism. Figure 1.2(c) shows an abstract view of an open-loop controller.

   Our approach for testing closed-loop controllers fails to automate test oracles for open-loop controllers, because the plant feedback is not available in open-loop controllers. As a result, we assume that test oracles are manual for open-loop controllers and focus on providing test case generation algorithms that develop small but effective test suites with high fault revealing ability. Our approach is hence able to effectively reduce the cost of manual test oracles. To develop such test suites, we identified two heuristics based on output signals of Simulink models. The first heuristic attempts to maximize the likelihood of failure patterns (i.e., anti-patterns) in the Simulink output signals. Figure 1.2(d) shows two failure patterns in mixed discrete-continuous outputs of Simulink models, namely *instability* and *discontinuity*. Presence of either of these failure patterns in Simulink outputs may have undesirable impact on physical processes or objects that are controlled by or interact with the controller. Our failure-based test generation algorithm uses search to maximize the likelihood of presence of these failure patterns in the output signals of Simulink models. Our second heuristic attempts to maximize the diversity between output signals of test cases within a test suite. Figure 1.2(e) provides an insight as to how our output diversity

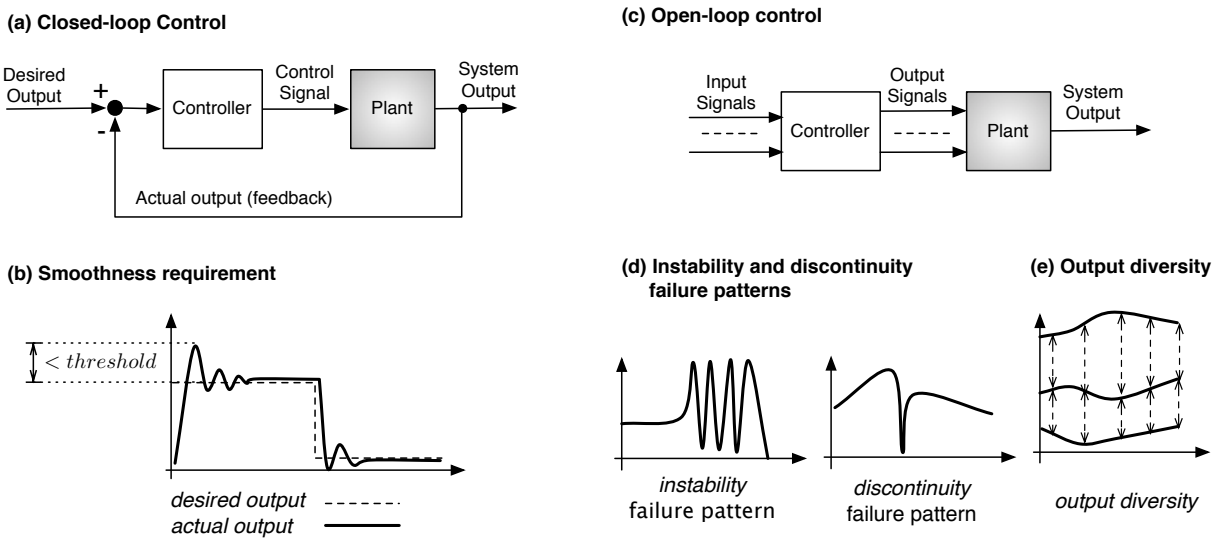


Figure 1.2. An overview of our output-based test generation approaches: (a) a closed-loop controller, (b) an smoothness requirement for closed-loop controller,(c) an open-loop controller, (d) two failure patterns for open-loop controllers, and (e) output diversity.

algorithm works. For output signals of test cases within a test suite (e.g., three output signals in Figure 1.2(e)), we attempt to maximize the distances between output signals, aiming at generating output signals with highly diverse shapes. Our intuition is that test cases that yield diverse output signals are likely to reveal different types of faults in Simulink models. Our failure-based and output diversity test generation algorithms for open-loop controllers are described in Chapters 5 and 6.

1.3 State of the art

The existing techniques that have been previously applied for testing and verification of Simulink models can be broadly categorized into two groups: *model-based testing techniques* and *model testing or verification*:

- Model-based testing relies on 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. For example, search-based approaches [Windisch, 2009, Windisch, 2010], reachability analysis [Mohalik et al., 2014, Hamon, 2008], guided random testing [Satpathy et al., 2008, Sims and DuVarney, 2007], and a combination of these techniques [The MathWorks Inc., 2016g, Reactive Systems Inc., 2016a, Satpathy et al., 2012, Peranandam et al., 2012, Gadkari et al., 2008, Bohr and Eschbach, 2011] have been previously applied to Simulink models to generate coverage-adequate test suites. Alternatively, some search-based [Zhan and Clark, 2005, Zhan and Clark, 2008] and bounded reachability analysis [Brillout et al., 2009] techniques have been used to generate mutant-killing test suites from Simulink models.
  The model-based testing techniques aim to generate test inputs as well as test oracles from Simulink models. That is, the Simulink models are considered to be *correct*. In reality, however, Simulink

models might contain faults. Hence, in our work, we propose techniques to provide test inputs with high fault-revealing ability to detect faults in complex discrete-continuous Simulink models for which precise test oracles are not available.

- In contrast to model-based testing, model testing and model checking techniques aim to evaluate the correctness of models. Specifically, they attempt (1) to exhaustively verify the correctness of models against a given set of properties using *model checking*, (2) to provide probabilistic guarantees for the given properties using *statistical model checking*, or (3) to detect faults in models by executing test inputs aiming at *falsifying* the given properties. Model checking has been previously used to detect faults in Simulink models [Cleaveland et al., 2008, Hamon, 2008, Barnat et al., 2012, Mazzolini et al., 2010] by showing that a path leading to an error is reachable. To alleviate the scalability problem of model checking, statistical model checking approaches check some randomly sampled simulations from the space of all possible model simulations to estimate the probability of the satisfaction or violation of the properties of interest [Zuliani et al., 2013, Legay et al., 2010, Younes and Simmons, 2006]. Simulation-based testing techniques run a set of test cases attempting to *falsify* assertions and properties instrumented into Simulink models [Reactive Systems Inc., 2016b, Cleaveland et al., 2008, S-Taliro, 2016, Zutshi et al., 2015].
  Similar to our approach, these techniques aim at detecting faults and verifying the correctness of Simulink models. However, they all rely on the presence of formal specifications that characterize the expected behaviors of Simulink models. Formal specifications are expensive and may not be available in practice. Furthermore, existing formal specifications typically describe only discrete system properties. As a result, even in the presence of some formal properties, engineers still need to check output signals to detect more failures. In fact, in practice it is likely and common that engineers assess output signals manually to detect failures. Hence, in our work, we do not rely on the presence of formal specifications to automate test oracles. Instead, for closed-loop controllers, we automate test oracle based on controller's feedback, and for open-loop controllers we assume that test oracles are manual. In the latter case, we generate small test suites with high fault revealing ability that effectively reduce the manual oracle cost. We further provide a test prioritization algorithm to lower the manual oracle cost.

In the beginning of Chapter 6, we discuss the challenges concerning the existing test generation techniques for Simulink models, in more details. We further discuss the related work on Simulink testing and other related topics in Chapter 7.

1.4 Research Contributions

In this dissertation, we addressed the challenges of testing Simulink/Stateflow models with mixed discrete-continuous behaviors. Specifically, the main research contributions in this thesis are:

1. An automated approach for testing closed-loop controllers that characterize the desired behavior of such controllers in a set of common requirements, and combines random exploration and meta-heuristic search (1) to generate shaded diagrams, called HeatMaps [Grinstein et al., 2001], visualizing the overall controller behavior with respect to the controller's requirements, and (2) to identify the worst-case test scenarios of the controller with respect to each requirement. This contribution has been published in a conference paper [Matinnejad et al., 2013] and a journal paper [Matinnejad et al., 2015a] and is discussed in Chapter 3.

2. An automated approach for testing highly configurable closed-loop controllers by accounting for all their feasible configurations in addition to input signals of the controller. Our approach includes strategies to scale the search to large multi-dimensional spaces relying on dimensionality reduction and surrogate modeling. Specifically, we use regression trees [Witten et al., 2011] instead of HeatMaps to visualize the overall controller behavior in a multi-dimensional space, composed of input variables and configuration parameters. In addition, we use sensitivity analysis [Campolongo et al., 2007] to identify configuration parameters with the most significant impact on search fitness functions and focus on them. Finally, machine learning [Witten et al., 2011] is applied to compute a surrogate function that can predict the output of controller model with a reasonably high-degree of confidence without executing the model. This contribution has been published in a conference paper [Matinnejad et al., 2014b] and is discussed in Chapter 4.
3. Two black-box output-based (failure-based and output diversity) test generation algorithms for Stateflow models with mixed discrete-continuous behaviors. Our failure-based algorithm attempts to maximize the likelihood of presence of specific failure patterns, i.e., instability and discontinuity, in discrete-continuous output signals of Stateflow models. Our output diversity algorithm maximizes the distance between output signals of test cases within a test suite to develop small test suites with high fault revealing ability. The notion of signal distance in this algorithm is defined based on the Euclidean distance between signal vectors, and hence we refer to it as vector-based output diversity. This contribution has been published as a conference paper [Matinnejad et al., 2015a] and is presented in Chapter 5.
4. An output diversity test generation algorithm based on a new notion of signal distance and a test prioritization algorithm for Simulink/Stateflow models in their entirety. We distinguished between the problem of testing simulation Simulink models with time-continuous behaviors and code-generation Simulink models with time-discrete behaviors, and proposed test generation algorithms applicable to both. Our output diversity algorithm uses a new notion of feature-based output diversity defined based on a set of representative and discriminating signal feature shapes, to enhance our vector-based output diversity algorithm. Our test prioritization algorithm generates 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 further compared our test generation approach with Simulink Design Verifier (SLDV) [The MathWorks Inc., 2016g], the only testing toolbox of Simulink and showed that our approach outperforms SLDV in revealing faults. This contribution has been published published as a conference paper at the 38th International Conference on Software Engineering (ICSE 2016) [Matinnejad et al., 2016a] and is discussed in Chapter 6. We have also submitted an extension of our ICSE paper to Transactions on Software Engineering (TSE) journal.
5. Continous Controller Tester (CoCoTest) [Matinnejad, 2016] and Simulink Controller Tester (SimCoTest) [Matinnejad, 2016] are tools that implement our test generation approaches for closed-loop and open-loop controllers, respectively. CoCoTest and SimCoTest have been published as two tool papers [Matinnejad et al., 2014a] and [Matinnejad et al., 2016a], and are presented in Chapters 3 and 6.
6. Empirical evaluation of our proposed approaches by applying them to real industrial closed-loop and open-loop Simulink/Stateflow controllers from our industry partner as well as publicly available models.

1.5 Organization of the Dissertation

Chapter 2 provides some foundational background on system modeling of dynamic systems, MATLAB/Simulink/Stateflow models, and meta-heuristic search algorithms.

Chapter 3 describes our search-based test generation approach for closed-loop controllers implemented in Simulink.

Chapter 4 describes our approach for testing highly configurable closed-loop controllers.

Chapter 5 describes our test generation algorithms for developing small and effective test suites for mixed discrete-continuous Stateflow models that implement open-loop controllers.

Chapter 6 describes our test suite generation and test prioritization algorithms for testing Simulink/Stateflow models in their entirety.

Chapter 7 discusses related work.

Chapter 8 summarizes the thesis contributions and discusses perspectives on future work.

Chapter 2

Background

In this chapter, we present some foundational background related to system modeling of dynamic systems described as MATLAB/Simulink/Stateflow models and single-state meta-heuristic search algorithms. We will present the background related to supervised learning and surrogate modeling techniques in Chapter 4 as they are used only in that chapter.

2.1 System modeling of dynamical systems

System modeling is the practice of creating mathematical descriptions of system properties and interactions in the system, specified as a set of functions that describe the relations between those properties [Wainer, 2009, Chaturvedi, 2009, Ledin, 2001]. When engineers are supposed to control physical objects and processes, they create mathematical models of the controller software as well as the system under control. These models capture the system behaviors and describe how the system is monitored, controlled and regulated. In domains such as the CPS domain, the primary goal of system modeling is *simulation*, i.e., design time testing of system models [Briand et al., 2016]. Simulation aims to identify defects by testing models in early stages and before the system has been implemented and deployed. The models usually go through several rounds of simulation and refinement, before engineers gain enough confidence in their behaviors and prepare them for *code generation*.

In the CPS domain, we are interested in models that have *dynamic* behavior (i.e., models that exhibit time-varying changes) [Lee and Seshia, 2011]. These models can be classified based on their time-base (i.e., time-discrete versus time-continuous) and based on the values of their state 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 [Wainer, 2009, Chaturvedi, 2009] (see Figures 2.1 (a) to (d)).

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 2.1. But most often, Simulation models include time-continuous or magnitude-continuous abstractions to be able to capture plant models and the interactions between software systems and plant models [Wainer, 2009, Chaturvedi, 2009]. Figure 2.2(a) shows an abstract

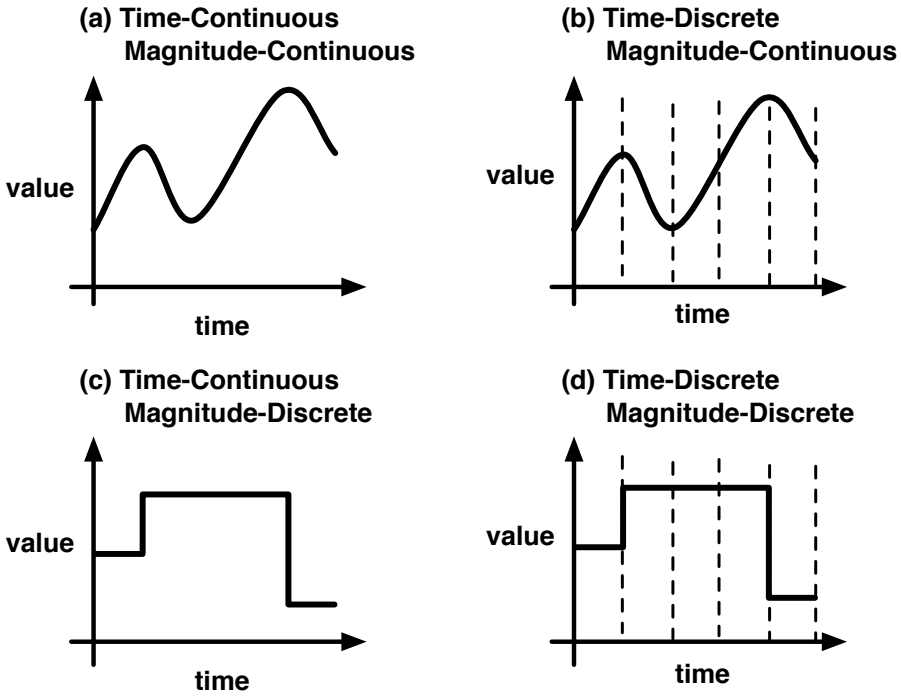


Figure 2.1. Four different modeling paradigms for cyber physical systems.

overview of a simulation model. 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 [The MathWorks Inc., 2016b, Krizan et al., 2014]. 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. Figure 2.2(b) shows an abstract overview of a code generation model. CPS development often starts with building simulation models capturing both continuous and discrete behaviors of a system [Wainer, 2009, Chaturvedi, 2009]. These models enable engineers to explore and understand the system behavior, to compare alternative design options and to start system testing very early. Simulation models are then discretized by replacing continuous calculations with their corresponding discrete approximation calculations to develop 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. In this thesis we provide effective verification and testing techniques to help engineers ensure correctness of simulation and code generation models in the CPS domain.

2.2 MATLAB/Simulink/Stateflow

MATLAB/Simulink/Stateflow is an advanced platform for modeling, simulation and code-generation of dynamic systems [The MathWorks Inc., 2003b], which is prevalently used in the CPS domain, e.g., in automotive industry. Simulink is a toolbox of MATLAB and provides a data-flow driven block diagram notation for modeling control software. Figure 2.3 shows the Simulink model of a cruise controller [The MathWorks Inc., 2016f]. Simulink models consist of blocks executing mathematical functions and lines connecting the blocks to implement the desired flow of data between the functions. Simulink models are structured into a hierarchy of subsystems, each one having a specific number of

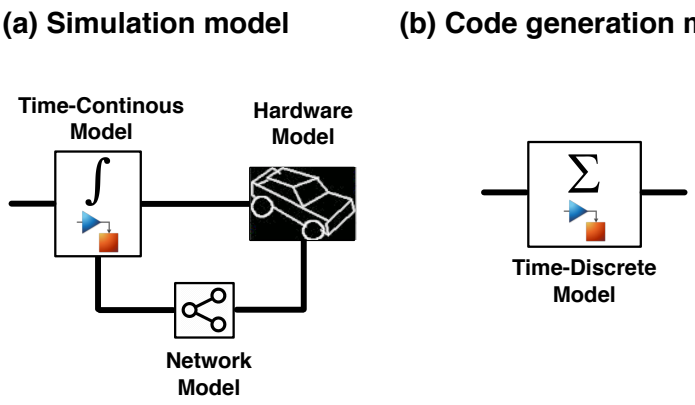


Figure 2.2. Simulation and Code generation models.

input and output signals. For example, the Controller subsystem in Figure 2.3(a) has six input and two output signals. Each subsystem contains some Simulink blocks and may contain some other subsystems. For example, the Controller subsystem contains another subsystem called PI Controller, indicated by a red circle in Figure 2.3(b). Figure 2.3(c) shows inside the PI Controller subsystem. The value of each model input goes through a sequence of computations performed by Simulink blocks in different subsystems, and eventually contributes to the value of one or several outputs of the model. In addition, Simulink models mostly incorporate a large number of configuration parameters, which are used to adjust the controller behavior for the final hardware platform. For example, the two configuration parameters KInc and KDec, indicated by dashed red circles in Figure 2.3(b), identify the step values by which the cruise controller increases and decreases the car speed, respectively.

Simulink blocks can be broadly classified into stateless and stateful blocks. The output of stateless blocks only depends on the current values of the block inputs. Examples of stateless blocks are logical and mathematical operations and user-defined functions. On the other hand, output of the stateful blocks depends on both the current input values as well as the state of the block. Unit Delay is an example of stateful blocks which holds and delays its input by one simulation step. Discrete and continuous integrator blocks are other examples of stateful blocks. The stateful blocks can further be categorized into discrete and continuous blocks. In the discrete stateful blocks, the state does not change between time steps. Hence, their output signals are magnitude-discrete, like the signal in Figure 2.1(d). The PI Controller in Figure 2.3(c) contains a time-discrete integrator. On the contrary, the continuous stateful blocks update the state continuously according to specified differential equations. Hence, their output signals are magnitude-continuous, like the signal in Figure 2.1(b). For example, a continuous integrator block represents the differential equation $\frac{dy}{dt} = x$, where x and y are the block input and output, respectively.

Stateflow [The MathWorks Inc., 2016h] is a hierarchical state machine language integrated into Simulink to model and simulate event-driven behavior of reactive systems. Such systems transition from one operating mode to another in response to input events and system conditions. Stateflow charts model the mode changes as a transition system. Stateflow allows input and output data, events for triggering other Stateflow charts,and actions and conditions that can be attached to states and transitions. Figure 2.4 shows an example of a Stateflow model. The ShiftLogic Stateflow subsystem, shown in Figure 2.4(a), implements part of an Auto Transmission Simulink controller. Figure 2.4(b)

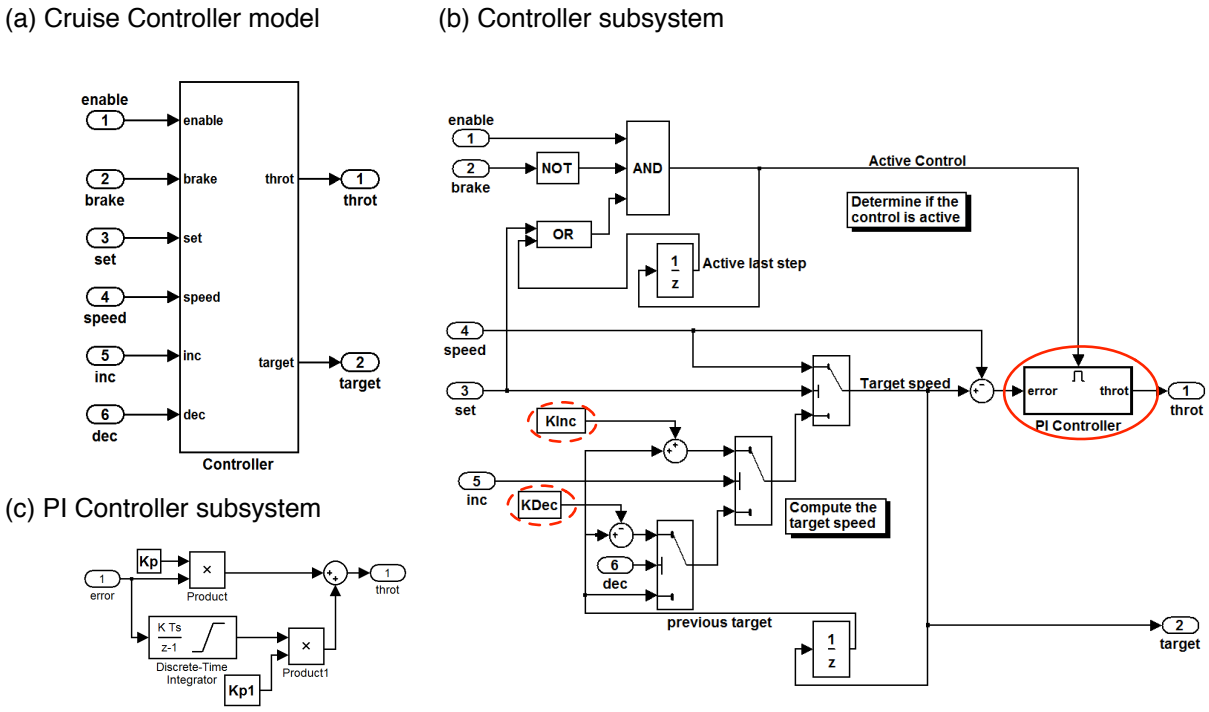


Figure 2.3. A Simulink model example: (a) Cruise Controller model, (b) Controller subsystem, and (c) PI Controller subsystem.

shows the ShiftLogic Stateflow model consisting of four states, each one representing one gear state. Up and Down represent the events of shifting the gear up and down by the driver, respectively.

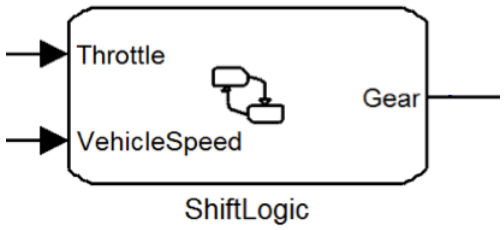
2.3 Single-State Meta-Heuristic Search

Meta-heuristic search is a randomized optimization method [Luke, 2013]. It has been applied to a wide range of problems for finding the optimal solution, where no analytic or numerical approximation technique can be used and brute force search is inapplicable due to the large size of the input space. To be able to apply meta-heuristic search to a given problem, all one needs is: (1) a way for creating and modifying arbitrary solutions to the problem, and (2) a method to assess and compare the quality of alternative solutions with respect to the optimization goal. The latter is usually called the *fitness* or *objective* function. In general, all meta-heuristic search algorithms can be described in terms of five steps [Luke, 2013]:

1. *initialize* step creates an initial set of solutions
2. *tweak* step creates a new set of solutions by randomly modifying the existing ones
3. *assess* step evaluates and compares the quality of the new and existing solutions
4. *select* step defines a new set of solutions by choosing a mix of old and new solutions
5. *iterate* step repeats steps 2-4 until maximum resources are spent or optimal solution is found

Single-state meta-heuristic search algorithms keep only one single candidate solution in the solu-

(a) ShiftLogic Stateflow subsystem



(b) ShiftLogic Stateflow model

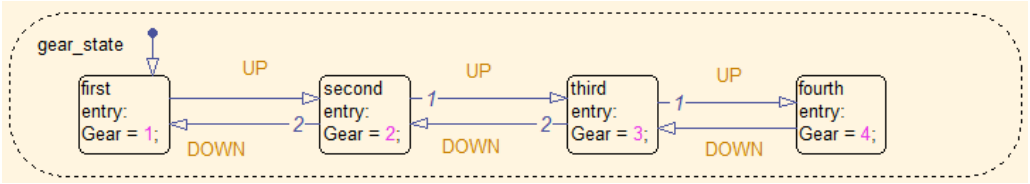


Figure 2.4. A Stateflow model example: (a) ShitfLogic Stateflow subsystem, and (b) ShitfLogic Stateflow model.

tion set. *Population-based search algorithms*, on the other hand, keep multiple candidate solutions in the solution set, called the *population*. In our work, evaluating fitness functions takes a relatively long time. Each fitness computation requires us to execute the simulation of the given Simulink model. Depending on the simulation time, the number of simulation steps, and the presence of feedback-loop in the model, this can take up to several minutes for one single simulation. For this reason, in our work we opt for single-state search methods in contrast to population-based search methods such as Genetic Algorithms (GA) [Luke, 2013]. Note that in population-based algorithms, we have to compute the fitness functions for the new and modified members of the solution set at each iteration.

Below, we describe three single-state meta-heuristic search algorithms that we use in our work: Hill-Climbing (HC), Hill-Climbing with Random Restarts (HCRR), and Simulated-Annealing (SA). We present each algorithm and highlight their differences between in terms of the five general steps of meta-heuristic search described above.

2.3.1 Hill-Climbing

Figure 2.5(a) shows the Hill-Climbing (HC) algorithm. HC is the simplest form of single-state meta-heuristic search that continuously loops in an uphill direction. Initially, a candidate solution S is generated at line 1 (*initialize* step). Then, at each iteration, a new solution R is generated by slightly modifying the current solution S at line 3 (*tweak* step). The *tweak* operation slightly modifies point S by adding some random *noise* to S. For example, if the candidate solution S is encoded by a single real value x, the *Tweak* operator shifts S in the input space by adding value x' to x. For HC algorithm, the x' is typically selected from a normal distribution with mean $\mu = 0$ and variance σ^2. Then, the new solution R replaces the current solution S at line 5, only if $Fitness(R)$ is larger than $Fitness(S)$, i.e., R fits better to the optimization goal (*assess* and *select* steps). Figures 2.5(b) and (c) illustrate the two different situations which may occur when comparing the fitness of R and S at line 4. In Figure 2.5(b), we have moved in an uphill direction after tweaking S, so the algorithm keeps R as the

(a)

Algorithm. Hill-Climbing

1. Let S be an initial candidate solution
2. **repeat**:
3. $R \leftarrow Tweak(S)$
4. **if** $Fitness(R) > Fitness(S)$ **:**
5. $S \leftarrow R$
6. **until** maximum resources spent
7. **return** S

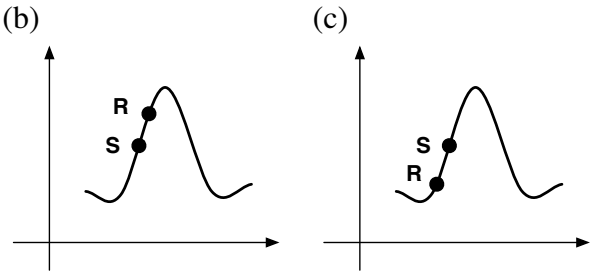


Figure 2.5. (a) Hill-Climbing procedure and (b,c) two different situations which may occur when comparing new (R) and current (S) solutions.

current solution and throws away S. In Figure 2.5(c), on the contrary, we have moved in a downhill direction, so the algorithm keeps S as the current solution and ignores R. The algorithm terminates either after a certain number of iterations or when the fitness function reaches a *plateau*, meaning no neighbor with a higher value can be found.

2.3.2 Hill-Climbing With Random Restarts

The Hill-Climbing algorithm in Figure 2.5(a) is essentially a greedy algorithm [Cormen et al., 2001] that always makes a locally optimal choice by picking a good neighbor as the current solution. As a result, it often gets stuck at a local optimum since it may be unable to find its way off a plateau in the fitness function. An attempt to solve this problem is provided by Hill-Climbing with Random Restarts algorithm. HCRR is essentially a series of Hill-Climbing searches from different random initial points. Specifically, it differs from HC only in the *replace* step. Like HC, HCRR replaces the current solution with a new solution with a higher fitness function. In addition, HCRR replaces the current solution with a random candidate solution from time to time. Figure 2.6(b) illustrates how replacing S with a random candidate solution may help getting off the plateau in fitness function and finding the global optimum. While *Best* has reached a local maximum in Figure 2.6(b), replacing S with a random candidate solution moves the search to a region where climbing the hill yields the global optimum.

2.3.3 Simulated Annealing

The success of Hill-Climbing with Random Restarts largely depends on the shape of the landscape. Specifically, when the landscape contains a few local optima, as illustrated in Figure 2.7(a), HCRR is likely to find a global optimum pretty quickly. However, when the landscape contains many needle-like optima, as illustrated in Figure 2.7(b), HCRR becomes slow and inefficient and is likely to get stuck at a local optimum. That is because for such a landscape, HCRR may need to make a large number of random restarts before the initial random candidate eventually moves to the region containing the global optimum.

Simulated annealing algorithm attempts to provide a more efficient solution for noisy landscapes with many needle-like optima. Figure 2.8 presents the Simulated Annealing (SA) algorithm. It differs from HC only in the replacement strategy (*replace* step). Like HC, SA always replaces S with R if R has a higher fitness function (lines $5-6$). However, SA may replace S with R even if S has a better fitness function. This latter situation occurs only if another condition based on a random

(a)

Algorithm. Hill-Climbing with Random Restarts

1. Let S be an initial candidate solution
2. $Best \leftarrow S$
3. **repeat**:
4. $\quad R \leftarrow Tweak(S)$
5. $\quad$ **if** $Fitness(R) > Fitness(S)$ **:**
6. $\quad\quad S \leftarrow R$
7. $\quad$ **if** $Fitness(S) > Fitness(Best)$ **:**
8. $\quad\quad Best \leftarrow S$
9. $\quad$ Replace S with a random candidate solution from time to time
10. **until** maximum resources spent
11. **return** *Best*

(b)

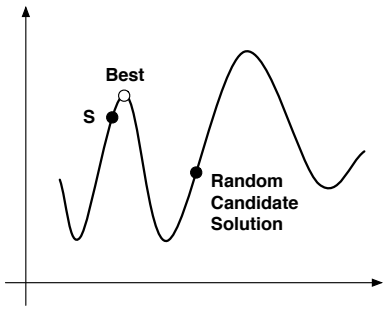


Figure 2.6. (a) Hill-Climbing with Random Restarts procedure and (b) the situation where random restart helps getting off the plateau.

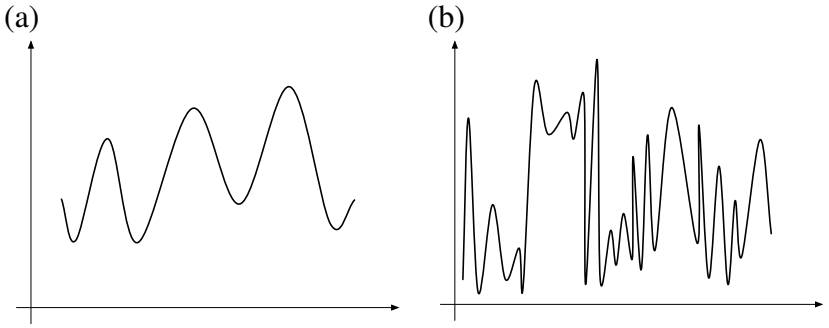


Figure 2.7. Two different shapes of fitness landscape where HCRR performs (a) good and (b) poorly.

Algorithm. Simulated Annealing

1. Let S be an initial candidate solution
2. $Best \leftarrow S$
3. **repeat**:
4. $\quad R \leftarrow Tweak(S)$
5. $\quad$ **if** $Fitness(R) > Fitness(S)$ **:**
6. $\quad\quad S \leftarrow R$
7. $\quad$ **elseif** A random value chosen between 0 and 1 is less than $e^{\frac{Fitness(R)-Fitness(S)}{t}}$ **:**
8. $\quad\quad S \leftarrow R$
9. $\quad Decrease\ t$
10. $\quad$ **if** $Fitness(S) > Fitness(Best)$ **:**
11. $\quad\quad Best \leftarrow S$
12. **until** maximum resources spent
13. **return** *Best*

Figure 2.8. The Simulated Annealing algorithm.

variable (called temperature t) holds. Specifically, S is replaced with R only if a randomly chosen

value between 0 and 1 is less than a certain replacement probability $RP(t,R,S) = e^{\frac{Fitness(R)-Fitness(S)}{t}}$ (line $7-8$). Since t is always positive and $Fitness(R)$ cannot be greater than $Fitness(S)$ at line 7, $RP(t,R,S)$ is always less than or equal to one. Further, the replacement probability is higher when (1) there exists a lager difference between $Fitness(R)$ and $Fitness(S)$, and (2) temperature t is higher. In SA algorithm, temperature t is initialized to some value at the beginning of the search and decreases over time, meaning SA replaces S with R more often at the beginning and less often towards the end of the search. That is, SA is more explorative and looks at the entire search space during the first iterations, but becomes more and more exploitative over time and focuses the search on the current solution's neighborhood.

Chapter 3

Search-Based Automated Testing of Continuous Controllers

In this chapter, we propose a search-based approach to automate generation of MiL level test cases for continuous controllers. We identify a set of common requirements characterizing the desired behavior of such controllers. We develop a search-based technique to generate stress test cases attempting to violate these requirements by combining *explorative* and *exploitative* search algorithms [Luke, 2013]. Specifically, we first apply a purely explorative random search to evaluate a number of input signals distributed across the search space. Combining the domain experts' knowledge and random search results, we select a number of regions that are more likely to lead to critical requirement violations in practice. We then start from the worst-case input signals found during exploration, and apply an exploitative *single-state* search [Luke, 2013] to the selected regions to identify test cases for the controller requirements. Our search algorithms rely on *objective* functions created by formalizing the controller requirements.

We have implemented our approach in a tool, called Continuous Controller Tester (CoCoTest). We evaluated our approach by applying it to an automotive air compressor module and to a publicly available controller model. Our experiments show that our approach automatically generates several test cases for which the MiL level simulations indicate potential errors in the controller model or in the environment model. Furthermore, the resulting test cases had not been previously found by manual testing based on domain expertise. In addition, our approach computes test cases better and faster than a random test case generation strategy. Finally, our generated test cases uncover discrepancies between environment models and the real world when they are applied at the Hardware-in-the-Loop (HiL) level.

Organization. This chapter is organized as follows. Section 3.1 precisely formulates the problem we aim to address in this chapter. Section 3.2 outlines our solution approach and describes how we cast our MiL testing approach as a search problem. The results of our evaluation of the proposed MiL testing approach, and our MiL testing tool, CoCoTest, are presented in Sections 3.3 and 3.4, respectively. Finally, Section 3.5 concludes the chapter.

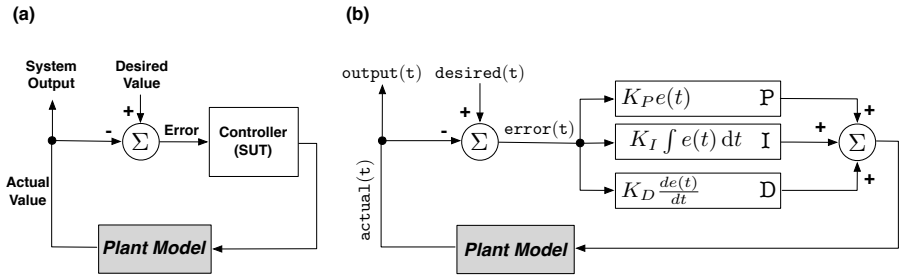


Figure 3.1. Continuous controllers: (a) A MiL level controller-plant model, and (b) a generic PID formulation of a continuous controller.

3.1 Problem Formulation

Figure 3.1(a) shows an overview of a controller-plant model at the MiL level. Both the controller and the plant are captured as models and linked via virtual connections. We refer to the input of the controller-plant system as *desired* or *reference* value. For example, the desired value may represent the location we want a robot to move to, the speed we require an engine to reach, or the position we need a valve to arrive at. The *system output* or the *actual* value represents the actual state/position/speed of the hardware components in the plant. The actual value is expected to reach or get as close as possible to the desired value over a certain time limit. The task of the controller is to minimize the *Error*, i.e., the difference between the actual and desired values.

The overall objective of the controller in Figure 3.1(a) may sound simple. In reality, however, the design of such controllers requires calculating proper corrective actions for the controller to stabilize the system within a certain time limit, and further, to guarantee that the hardware components will eventually reach the desired state without oscillating too much around it and without any damage. A controller design is typically implemented via complex differential equations known as *proportional-integral-derivative (PID)* [Nise, 2004]. Figure 3.1(b) shows the generic (most basic) formulation of a PID equation. Let $e(t)$ be the difference between $desired(t)$ and $actual(t)$ (i.e., error). A PID equation is a summation of three terms: (1) a proportional term $K_P e(t)$, (2) an integral term $K_I \int e(t)dt$, and (3) a derivative term $K_D \frac{de(t)}{dt}$. Note that the PID formulation for real world controllers are more complex than the formula shown in Figure 3.1(b). Figure 3.2 shows a typical output diagram of a PID controller. As shown in the figure, the actual value starts at an initial value (here zero), and gradually moves to reach and stabilize at a value close to the desired value.

Continuous controllers are characterized by a number of generic requirements discussed in Section 3.1.1. Having specified the requirements of continuous controllers, we show in Section 3.1.2 how we define testing objectives based on these requirements, and how we formulate MiL testing of continuous controllers as a search problem.

3.1.1 Testing Continuous Controller Requirements

To ensure that a controller design is satisfactory, engineers perform several simulations, and analyze the output simulation diagram (Figure 3.2) with respect to a number of requirements. After careful

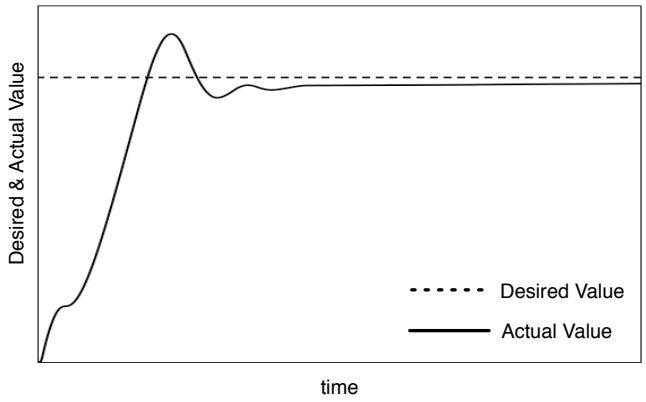


Figure 3.2. A typical example of a continuous controller output (Note that actual value does not reach desired value in this example).

investigations, we identified the following requirements for controllers:

Stability (safety, functional): The actual value shall reach and stabilize at the desired value after t_1 seconds. This is to make sure that the actual value does not divert from the desired value or does not keep oscillating around the desired value after reaching it. [1]

Smoothness (safety): The actual value shall not change abruptly when it is close to the desired one. That is, the difference between the actual and desired values shall not exceed v_2, once the difference has already reached v_1 for the first time. This is to ensure that the controller does not damage any physical devices by sharply changing the actual value when the error is small.

Responsiveness (performance): The difference between the actual and desired values shall be at most v_3 within t_2 seconds, ensuring the controller responds within a time limit.

The above three requirement templates are illustrated on a typical controller output diagram in Figure 3.3 where the parameters t_1, t_2, v_1, v_2, and v_3 are represented. The first three parameters represent time while the last three are described in terms of the controller output values. As shown in the figure, given specific controller requirements with concrete parameters and given an output diagram of a controller under test, we can determine whether that particular controller output satisfies the given requirements.

Having discussed the controller requirements and outputs, we now describe how we generate input test values for a given controller. Typically, controllers have a large number of configuration parameters that affect their behaviors. For the configuration parameters, we use a value assignment commonly used for HiL testing because it enables us to compare the results of MiL and HiL testing. In our approach, we focus on two essential controller inputs in our MiL testing approach: (1) the initial actual value, and (2) the desired value. Among these two inputs, the desired value can be easily manipulated externally. However, since the controller is a closed loop system, it is not generally pos-

[1] Note that in our work we use step input signals as the desired value of the controller. Since the desired value is not oscillating, the actual value shall not oscillate as well.

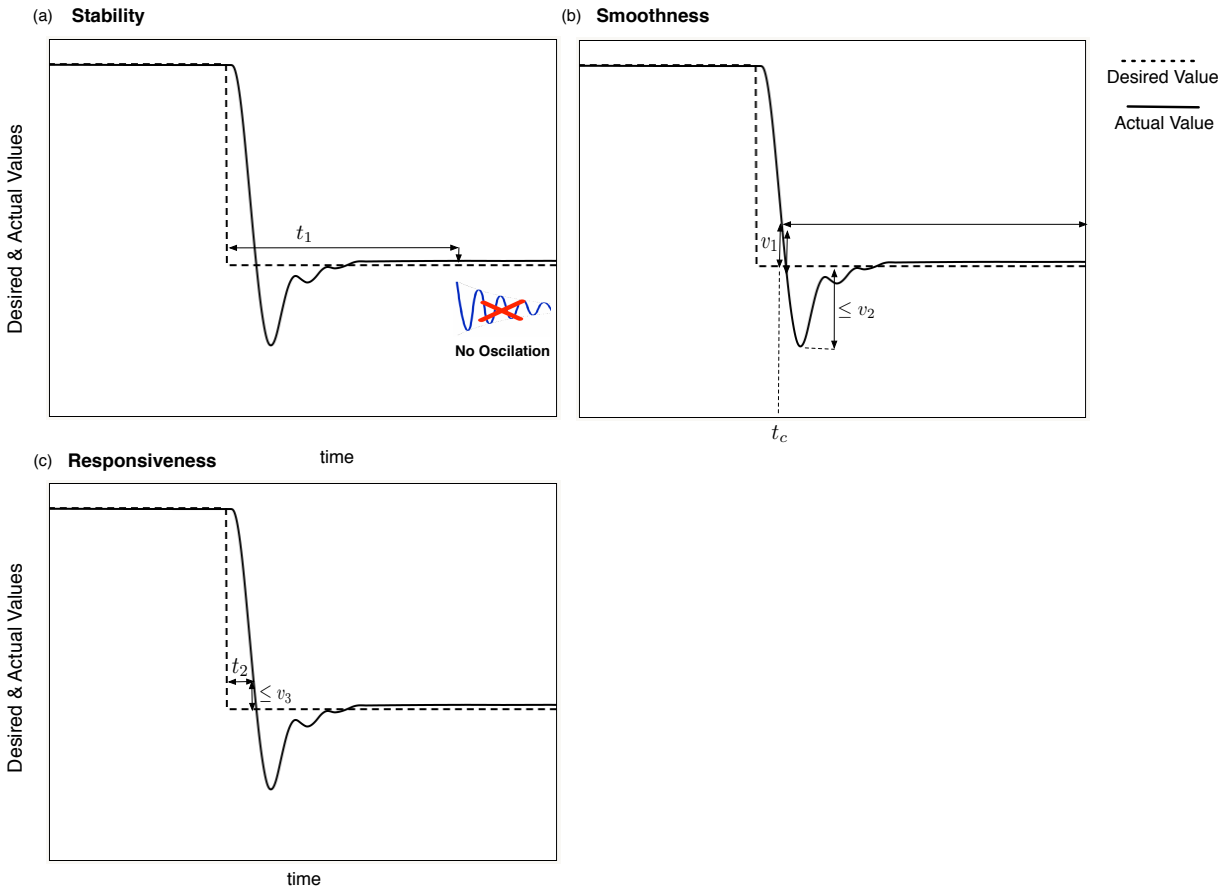


Figure 3.3. The controller requirements illustrated on the controller output: (a) Stability, (b) Smoothness, and (c) Responsiveness.

sible to modify the initial actual value and start the system from an arbitrary initial state. In general, the initial actual state, which is usually set to zero, depends on the plant model and cannot be manipulated externally. Assuming that the system always starts from zero is like testing a cruise controller only for positive car speed increases, and missing a whole range of speed decreasing scenarios.

To eliminate this restriction, we provide a step signal for the desired value of the controller (see examples of step signals in Figure 3.3 and Figure 3.4(a)). The step signal consists of two consecutive constant signals such that the first one sets the controller at the *initial* desired value, and the second one moves the controller to the *final* desired value. The lengths of the two signals in a step signal are equal (see Figure 3.4(a)), and should be sufficiently long to give the controller enough time to stabilize at each of the initial and final desired values. Figure 3.4(b) shows an example of a controller output diagram for the input step signal in Figure 3.4(a).

3.1.2 Formulating MiL Testing as a Search Problem

Given a controller-plant model and a set of controller requirements, the goal of MiL testing is to generate controller input values such that the resulting controller output values violate, or become close to violating, the given requirements. Based on this description, any MiL testing strategy has to perform the following common tasks: (1) It should generate input signals to the controller, i.e.,

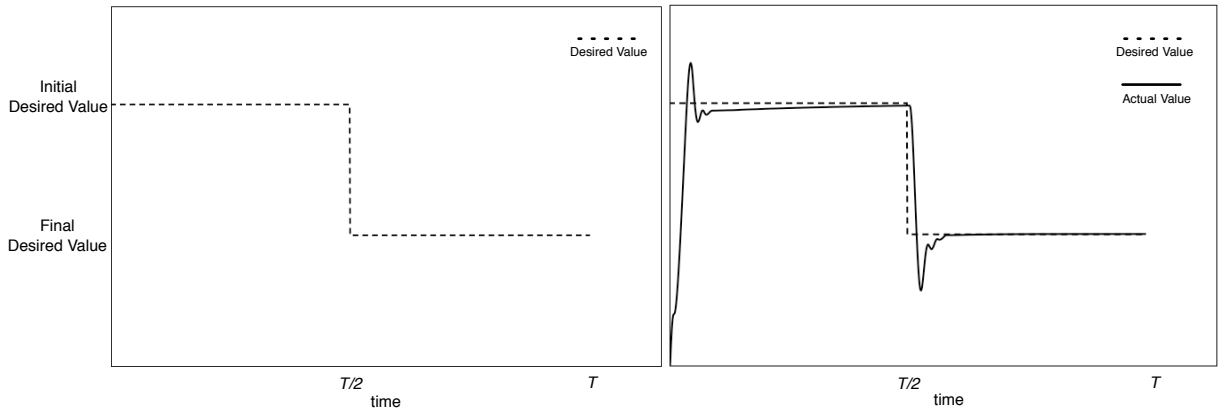


Figure 3.4. Controller input step signals: (a) Step signal. (b) Output of the controller (actual) given the input step signal (desired).

step signal in Figure 3.4(a). (2) It should receive the output, i.e., `Actual` in Figure 3.4(b), from the controller model, and evaluate the output against the controller requirements. Below, we first formalize the controller input and output, and then, we derive four objective functions from the three requirements introduced in Section 3.1.1. Specifically, we develop one objective function for each of the stability and responsiveness requirements, and two objective functions for the smoothness requirement.

Controller input and output: Let $\mathcal{T} = \{0, \ldots, T\}$ be a set of time points during which we observe the controller behavior, and let `min` and `max` be the minimum and maximum values for the `Actual` and `Desired` attributes in Figure 3.1(a). In our work, since input is assumed to be a step signal, the observation time T is chosen to be long enough so that the controller can reach and stabilize within $\frac{T}{2}$ at each of the two `Desired` position (e.g., see Figure 3.4(b)). Note that `Actual` and `Desired` are of the same type and bounded within the same range, denoted by $[\texttt{min} \ldots \texttt{max}]$. As discussed in Section 3.1, the test inputs are step signals representing the `Desired` values (e.g. Figure 3.4(a)). We define an input step signal in our approach to be a function $\texttt{Desired} : \mathcal{T} \rightarrow \{\texttt{min}, \ldots, \texttt{max}\}$ such that there exists a pair `Initial Desired` and `Final Desired` of values in $[\texttt{min} \ldots \texttt{max}]$ that satisfy the following conditions:

$$\forall t \cdot 0 \leq t < \tfrac{T}{2} \Rightarrow \texttt{Desired}(t) = \texttt{Initial Desired} \bigwedge$$
$$\forall t \cdot \tfrac{T}{2} \leq t < T \Rightarrow \texttt{Desired}(t) = \texttt{Final Desired}$$

We define the controller output, i.e., `Actual`, to be a function $\texttt{Actual} : \mathcal{T} \rightarrow \{\texttt{min}, \ldots, \texttt{max}\}$ that is produced by the given controller-plant model, e.g., in MATLAB/Simulink environment.

The search space in our problem is the set of all possible input functions, i.e., the `Desired` function. Each `Desired` function is characterized by the pair `Initial Desired` and `Final Desired` values. In control system development, it is common to use floating point data types at MiL level. Therefore, the search space in our work is the set of all pairs of floating point values for `Initial Desired`

and `Final Desired` within the [`min...max`] range.

Objective Functions: Our goal is to guide the search to identify input functions in the search space that are more likely to break the properties discussed in Section 3.1.1. To do so, we create the following four objective functions:

- **Stability:** Let t_1 be the stability property parameter in Section 3.1.1. We define the stability objective function F_{st} as:

  $$StdDev_{t_1+\frac{T}{2}<t\leq T}\{\texttt{Actual}(t)\}$$

  That is, F_{st} is the standard deviation of the values of `Actual` function between $t_1+\frac{T}{2}$ and T.
- **Smoothness**: Let v_1 be the smoothness property parameter in Section 3.1.1. Let $tc \in \mathcal{T}$ be such that $tc > \frac{T}{2}$ and

  $$|\texttt{Desired}(tc) - \texttt{Actual}(tc)| \leq v_1 \ \wedge$$
  $$\forall t \cdot \tfrac{T}{2} \leq t < tc \Rightarrow |\texttt{Desired}(t) - \texttt{Actual}(t)| > v_1$$

  That is, tc is the first point in time after $\frac{T}{2}$ where the difference between `Actual` and `Final Desired` values has reached v_1. We then define the smoothness objective function F_{sm} as:

  $$max_{tc<t\leq T}\{|\texttt{Desired}(t) - \texttt{Actual}(t)|\}$$

  That is, the function F_{sm} is the maximum difference between `Desired` and `Actual` after tc.
  Note that F_{sm} measures the *absolute* value of undershoot/overshoot. We noticed in our work that in addition to finding the largest undershoot/overshoot scenarios, we need to identify scenarios where the overshoot/undershoot is large compared to the step size in the input step signal. In other words, we are interested in scenarios where a small change in the position of the controller, i.e., a small difference between `Initial Desired` and `Final Desired`, yields a *relatively* large overshoot/undershoot. These latter scenarios cannot be identified if we only use the F_{sm} function for evaluating the Smoothness requirement. Therefore, we define the next objective function, normalized smoothness, to help find such test scenarios.
- **Normalized Smoothness**: We define the normalized smoothness objective function F_{ns}, by normalizing the F_{sm} function:

  $$F_{ns} = \frac{F_{sm}}{|\texttt{Final Desired} - \texttt{Initial Desired}|}$$

  F_{ns} evaluates the overshoot/undershoot values relative to the step size of the input step signal.
- **Responsiveness**: Let v_3 be the responsiveness parameter in Section 3.1.1. We define the responsiveness objective function F_r to be equal to tr such that $tr \in \mathcal{T}$ and $tr > \frac{T}{2}$ and

  $$|\texttt{Desired}(tr) - \texttt{Actual}(tr)| \leq v_3 \ \wedge$$
  $$\forall t \cdot \tfrac{T}{2} \leq t < tr \Rightarrow |\texttt{Desired}(t) - \texttt{Actual}(t)| > v_3$$

  That is, F_r is the first point in time after $\frac{T}{2}$ where the difference between `Actual` and `Final Desired` values has reached v_3.

We did not use v_2 from the smoothness and t_2 from the responsiveness properties in definitions of F_{sm} and F_r. These parameters determine pass/fail conditions for test cases, and are not required to guide the search. Further, v_2 and t_2 depend on the specific hardware characteristics and vary from customer to customer. Hence, they are not known at the MiL level. Specifically, we define F_{sm} to measure the maximum overshoot rather than to determine whether an overshoot exceeds v_2, or not. Similarly, we define F_r to measure the actual response time without comparing it with t_2.

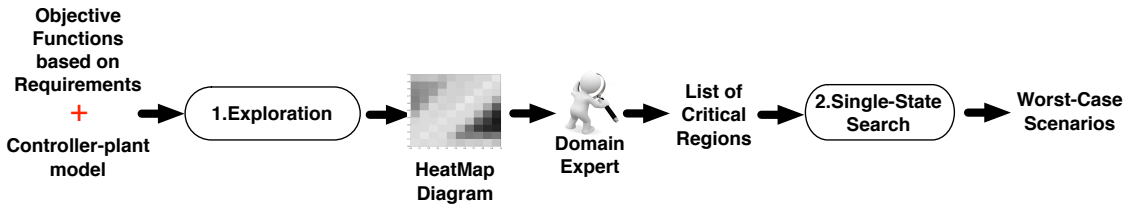


Figure 3.5. An overview of our automated approach to MiL testing of continuous controllers.

The above four objective functions are heuristics and provide quantitative estimates of the controller requirements, allowing us to compare different test inputs. The higher the objective function value, the more likely it is that the test input violates the requirement corresponding to that objective function. We use these objective functions in our search algorithms discussed in the next section.

In the next section, we describe our automated search-based approach to MiL testing of controller-plant systems. Our approach automatically generates input step signals such as the one in Figure 3.4(a), produces controller output diagrams for each input signal, and evaluates the four controller objective functions on the output diagram. Our search is guided by a number of heuristics to identify the input signals that are more likely to violate the controller requirements. Our approach relies on the fact that, during MiL, a large number of simulations can be generated quickly and without breaking any physical devices. Using this characteristic, we propose to replace the existing manual MiL testing with our search-based automated approach that enables the evaluation of a large number of output simulation diagrams and the identification of critical controller input values.

3.2 Solution Approach

In this section, we describe our search-based approach to MiL testing of controllers, and show how we employ and combine different search algorithms to guide and automate MiL testing of continuous controllers. Figure 3.5 shows an overview of our automated MiL testing approach. In the first step, we receive a controller-plant model (e.g., in MATLAB/Simulink) and a set of objective functions derived from requirements. We partition the input search space into a set of regions, and for each region, compute a value indicating the evaluation of a given objective function on that region based on random search (exploration). We refer to the result as a *HeatMap* diagram [Grinstein et al., 2001]. HeatMap diagrams are graphical 2-D or 3-D representations of data where a matrix of values are represented by colors. In this work, we use grayscale 2-D HeatMap diagrams (see Figure 3.6(b) for an example). These diagrams are intuitive and easy to understand by the users of our approach. In addition, our HeatMap diagrams are divided into equal regions (squares), making it easier for engineers to delineate critical parts of the input space in terms of these equal and regular-shape regions. Based on domain expert knowledge, we select some of the regions that are more likely to include critical and realistic errors. In the second step, we focus our search on the selected regions and employ a single-state heuristic search to identify, within those regions, the worst-case scenarios to test the controller. Single-state search optimizers only keep one candidate solution at a time, as opposed to *population-based* algorithms that maintain a set of samples at each iteration [Luke, 2013].

In the first step of our approach in Figure 3.5, we apply a random (unguided) search to the entire search space in order to identify high risk areas. The search explores diverse test inputs to provide an

unbiased estimate of the average objective function values in different regions of the search space. In the second step, we apply a heuristic single-state search to a selection of regions in order to find worst-case test scenarios that are likely to violate the controller properties. In the following two sections, we discuss these two steps.

3.2.1 Exploration

Figure 3.6(a) shows the exploration algorithm used in the first step. The algorithm takes as input a controller-plant model M and an objective function F, and produces a HeatMap diagram (e.g., see Figure 3.6(b)). Briefly, the algorithm divides the input search space S of M into a number of equal regions. It then generates a random point p in S in line 4. The dimensions of p characterize an input step function `Desired` which is given to M as input in line 6. The model M is executed in Matlab/Simulink to generate the `Actual` output. The objective function F is then computed based on the `Desired` and `Actual` functions. The tuple (p, f) where f is the value of the objective function at p is added to P. The algorithm stops when the number of generated points in each region is at least N. Finding an appropriate value for N is a trade off between accuracy and efficiency. Since executing M is relatively expensive, it is often not practical to generate many points (large N). Likewise, a small number of points in each region is unlikely to give us an accurate estimate of the average objective function for that region.

Input: The exploration algorithm in Figure 3.6(a) takes as input a controller-plant model M, an objective function F, and an observation time T. Note that the controller model is an abstraction of the software under test, and the plant model is required to simulate the controller model, and evaluate the objective functions. The length of simulation is determined by the observation time T. Finally, the objective function F is chosen among the four objective functions described in section 3.1.2.

Output: The output of the algorithm in Figure 3.6(a) is a set P of (p, f) tuples where p is a point in the search space and f is the objective function value for p. The set P is visualized as a HeatMap diagram [Grinstein et al., 2001] where the axes are the initial and final desired values. In HeatMaps, each region is assigned the average value of the values of the points within that region. The intervals of the region values are then mapped into different shades, generating a shaded diagram such as the one in Figure 3.6(b). In our work, we generate four HeatMap diagrams corresponding to the four objective functions F_{st}, F_{sm}, F_{ns} and F_r discussed in Section 3.1.2. The HeatMap diagrams generated in the first step are reviewed by domain experts. They select a set of regions that are more likely to include realistic and critical inputs. For example, the diagram in Figure 3.6(b) is generated based on an air compressor controller model evaluated for the smoothness objective function F_{sm}. This controller compresses the air by moving a flap between its open position (indicated by 0) and its closed position (indicated by 1.0). There are about 10 to 12 dark regions, i.e., the regions with the highest F_{sm} values in Figure 3.6(b). These regions have initial flap positions between 0.5 and 1.0, and final flap positions between 0.1 and 0.6. Among these regions, domain experts tend to focus on regions with initial values between 0.8 and 1.0, or final values between 0.8 and 1.0. This is because, in practice, there is more probability of damage when a closed (or a nearly closed) flap is being moved, or when a flap is about to be closed.

Algorithm: Figure 3.6(a) shows a generic description of our exploration algorithm. This algorithm

(a)

Algorithm. Exploration

Input: A controller-plant model M with input search space S.
An objective function F.
An observation time T.
Output: An overview diagram (HeatMap) HM.
The set P of HM points.

1. Partition S into equal sub-regions
2. Let $P = \{\}$
3. **repeat**
4. Let $p = (\texttt{Initial Desired}, \texttt{Final Desired})$ be a random point in S
5. Let `Desired` be a step function generated based on p and T
6. Run M with the `Desired` input to obtain the `Actual` output
7. $f = F(\texttt{Desired}, \texttt{Actual})$
8. $P = \{(p, f)\} \cup P$
9. **until** there are at least N points in each region of S **do**
10. Create a HeatMap diagram HM based on P
11. Return HM and P

(b)

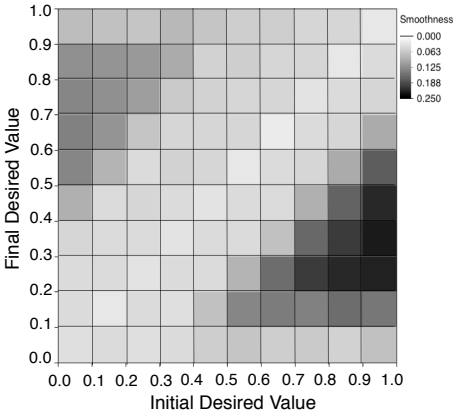


Figure 3.6. The first step of our approach in Figure 3.5: (a) The exploration algorithm. (b) An example HeatMap diagram produced by the algorithm in (a)

can be implemented in different ways by devising various heuristics for generating random points in the input search space S (line 4 in Figure 3.6(a)). In our work, we use two alternative heuristics for line 4 of the algorithm: (1) a simple random search, which we call naive random search, and (2) an adaptive random search algorithm [Luke, 2013]. The naive random search simply calls a random function to generate points in line 4 in Figure 3.6(a). Adaptive random search is an extension of the naive random search that attempts to maximize the euclidean distance between the selected points. Specifically, it explores the space by iteratively selecting points in areas of the space where fewer points have already been selected. To implement adaptive random search, the algorithm in Figure 3.6(a) changes as follows: Let P_i be the set of points selected by adaptive random search at iteration i (line 8 in Figure 3.6(a)). At iteration $i+1$, at line 4, instead of generating one point, adaptive random search randomly generates a set X of candidate points in the input space. The search computes distances between each candidate point $p \in X$ and the points in P_i. Formally, for each point $p = (v_1, v_2)$ in X, the search computes a function $dist(p)$ as follows:

$$dist(p) = \text{MIN}_{(v_1', v_2') \in P_i} \sqrt{(v_1 - v_1')^2 + (v_2 - v_2')^2}$$

The search algorithm then picks a point $p \in X$ such that $dist(p)$ is the largest, and proceeds to the lines 5 to 7. Finally, the selected p together with the value f of objective function at point p is added to P_i to generate P_{i+1} in line 8.

The algorithm in Figure 3.6(a) stops when at least N points have been selected in each region. We anticipate the adaptive random heuristic reaches this termination condition faster than a naive random heuristic because it generates points that are more evenly distributed across the entire space. Our work is similar to *quasi-random number generators* that are available in some languages, e.g., MATLAB [The MathWorks Inc., 2003a]. Similar to our adaptive random search algorithm, these number generators attempt to minimize the discrepancy between the distribution of generated points.

We evaluate efficiency of naive random search and adaptive random search in generating HeatMap diagrams in Section 3.3.

3.2.2 Single-State Search

Figure 3.7(a) presents our single-state search algorithm for the second step of the procedure in Figure 3.5. The single-state search algorithm starts with the point with the worst (highest) objective function value among those computed by the random search in Figure 3.6(a). It then iteratively generates new points by tweaking the current point (line 6) and evaluates the given objective function on the newly generated points. Finally, it reports the point with the worst (highest) objective function value. In contrast to random search, the single-state search is guided by an objective function and performs a tweak operation. Since the search is driven by the objective function, we have to run the search four times separately for F_{st}, F_{sm}, F_{ns} and F_r.

At this step, we rely on single-state exploitative algorithms. These algorithms mostly make local improvements at each iteration, aiming to find and exploit local gradients in the space. In contrast, explorative algorithms, such as the adaptive random search we used in the first step, mostly wander about randomly and make big jumps in the space to explore it. We apply explorative search at the beginning to the entire search space, and then focus on a selected area and try to find worst case scenarios in that area using exploitative algorithms.

Input: The input to the single-state search algorithm in Figure 3.7(a) is the controller-plant model M, an objective function F, an observation time T, a HeatMap region r, and the set P of points generated by exploration algorithm in Figure 3.6(a). We already explained the input values M, F and T in Section 3.2.1. Region r is chosen among the critical regions of the HeatMap identified by the domain expert. The single-state search algorithm focuses on the input region r to find a worst-case test scenario of the controller. The algorithm, further, requires P to identify its starting point in r, i.e., the point with the worst (highest) objective function in r.

Output: The output of the single-state search algorithm is a worst-case test scenario found by the search after K iterations. For instance Figure 3.7(b) shows the worst-case scenario computed by our algorithm for the smoothness objective function applied to an air compressor controller. As shown in the figure, the controller has an undershoot around 0.2 when it moves from an initial desired value of 0.8 and is about to stabilize at a final desired value of 0.3.

Algorithm: The algorithm in Figure 3.7(a) represents a generic single-state search algorithm. Specifically, there are two placeholders in this figure: *Tweak*() at line 6, and *Replace*() at line 13. To implement different single-state search heuristics, one needs to define how the existing point p should be modified (the *Tweak*() operator), and to determine when the existing point p should be replaced with a newly generated point *newp* (the *Replace* operator).

In our work, we instantiate the algorithm in Figure 3.7(a) based on three single-state search heuristics: standard Hill-Climbing (HC), Hill-Climbing with Random Restarts (HCRR), and Simulated Annealing (SA). Specifically, the *Tweak*() operator for HC shifts p in the space by adding values x' and y' to the dimensions of p. The x' and y' values are selected from a normal distribution with mean $\mu = 0$ and variance σ^2. The *Replace* operator for HC replaces p with *newp*, if and only if *newp* has a worse

(a)

Algorithm. SingleStateSearch

Input: A controller-plant model M with input search space S.
A region r.
The set P computed by the algorithm in Figure 3.6(a).
An objective function F.
An observation time T.
Output: The worst-case test scenario *testCase*.

1. $P' = \{(p,f) \in P \mid p \in r\}$
2. Let $(p,f) \in P'$ s.t. for all $(p',f') \in P'$, we have $f \geq f'$
3. $worstFound = f$
4. $AdaptParameteres(r,P)$
5. **for** K iterations **:**
6. $newp = Tweak(p)$
7. Let `Desired` be a step function generated by *newp*
8. Run M with the `Desired` input to obtain the `Actual` output
9. $v = F(\texttt{Desired},\texttt{Actual})$
10. **if** $v > worstFound$ **:**
11. $worstFound = v$
12. $testCase = newp$
13. $p = Replace(p,newp)$
14. **return** *testCase*

(b)

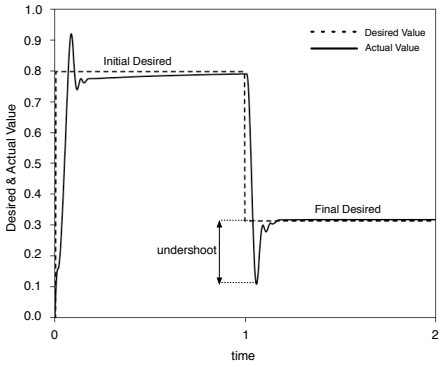


Figure 3.7. The second step of our approach in Figure 3.5: (a) The single-state search algorithm. (b) An example output diagram produced by the algorithm in (a)

(higher) objective function than p. HCRR and SA are different from HC only in their replacement policy. HCRR restarts the search from time to time by replacing p with a randomly selected point. Like HC, SA always replaces p with *newp* if *newp* has a worse (higher) objective function. However, SA may replace p with *newp* even if *newp* has a better (lower) objective function. This latter situation occurs only if another condition based on a random variable (temperature t) holds. Temperature is initialized to some value at the beginning of the search and decreases over time, meaning that SA replaces p with *newp* more often at the beginning and less often towards the end of the search. That is, SA is more explorative during the first iterations, but becomes more and more exploitative over time.

In general, single-state search algorithms, including HC, HCRR, and SA, have a number of configuration parameters (e.g., variance σ in HC, and the initial temperature value and the speed of decreasing the temperature in SA). These parameters serve as knobs with which we can tune the degree of exploitation (or exploration) of the algorithms. To be able to effectively tune these parameters in our work, we visualized the landscape of several regions from our HeatMap diagrams. We noticed that the region landscapes can be categorized into two groups: (1) Regions with a clear gradient between the initial point of the search and the worst-case point (see e.g., Figure 3.8(a)). (2) Regions with a noisier landscape and several local optima (see e.g., Figure 3.8(b)). We refer to the regions in the former group as *regular regions*, and to the regions in the latter group as *irregular regions*. As expected, for regular regions, like the region in Figure 3.8(a), exploitative heuristics work best, while for irregular regions, like the region in Figure 3.8(b), explorative heuristics are most suitable [Luke, 2013].

Note that the number of points generated and evaluated in each region in the first step (the ex-

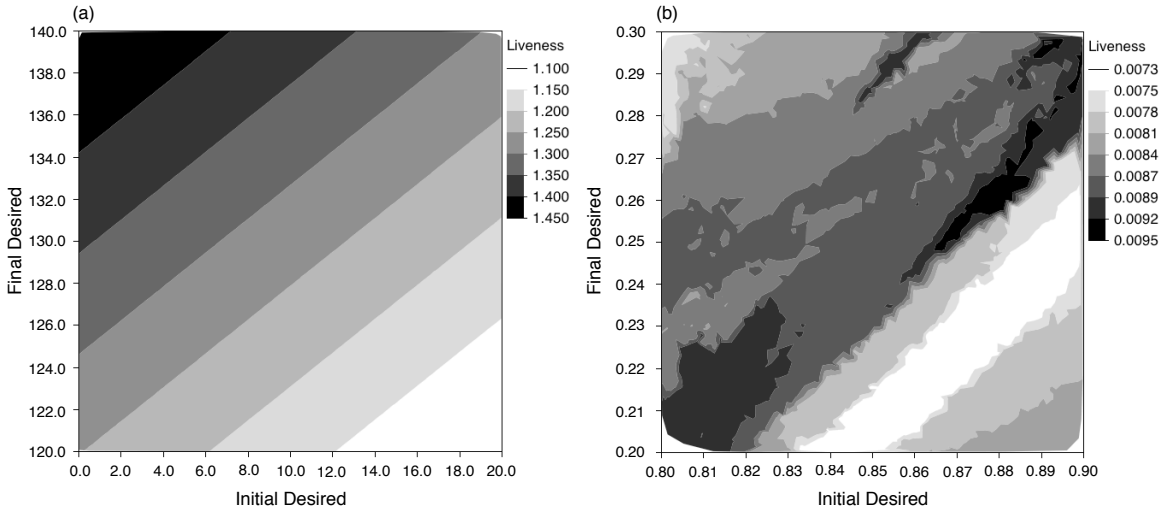


Figure 3.8. Diagrams representing the landscape for regular and irregular HeatMap regions: (a) A regular region with a clear gradient between the initial point of the search and the worst-case point. (b) An irregular region with several local optima.

ploration step) is not sufficiently large so that we can conclusively determine whether a given region belongs to the regular group or to the irregular group above. Therefore, in our work, we rely on a heuristic that attempts to predict the region group based on the information available in HeatMap diagrams. Specifically, our observation shows that dark regions mostly surrounded by dark shaded regions belong to regular regions, while dark regions located in generally light shaded areas belong to irregular regions. Using this heuristic, we determine whether a given region belongs to a regular group or to an irregular group. For regular regions, we need to use algorithms that exhibit a more exploitative search behavior, and for irregular regions, we require algorithms that are more explorative. In Section 3.3, we evaluate our single-state search algorithms, HC, HCRR and SA, by applying them to both groups of regions, and comparing their performance in identifying worst-case scenarios in each region group.

3.3 Evaluation

In this section, we present the research questions that we set out to answer (Sections 3.3.1), relevant information about the industrial case study (Section 3.3.2), and the key parameters in setting our experiment and tuning our search algorithms (Section 3.3.3). We then provide answers to our research questions based on the results obtained from our experiment (Section 3.3.4). Finally, we discuss practical usability of the HeatMap diagrams and the worst-case test scenarios generated by our approach (Section 3.3.5).

3.3.1 Research Questions

RQ1: How does adaptive random search perform compared to naive random search in generating HeatMap diagrams?

RQ2: How do our single-state search algorithms (i.e., HC, HCRR, and SA) compare with one another

in identifying the worst-case test scenarios? How do these algorithms compare with random search (baseline)?

RQ3: Does our single-state search algorithm (step 2 in Figure 3.5) improve the results obtained by the exploration step (step 1 in Figure 3.5)?

RQ4: Does our MiL testing approach help identify test cases that are useful in practice?

Any search-based solution should be compared with random search which is a standard "baseline" of comparison. If a proposed search-based solution does not show any improvement over random search, either something is wrong with the solution or the problem is trivial for which a random search approach is sufficient. In **RQ1** and **RQ2**, we respectively compare, with random search, our adaptive random search technique for HeatMap diagram generation, and our single-state search algorithms in finding worst-case test scenarios. In **RQ2**, in addition to comparing with random search, we compare our three single-state search algorithms with one another to identify if there is an algorithm that uniformly performs better than others for all the HeatMap regions. In **RQ3**, we argue that the second step of our approach (the search step) is indeed necessary and improves the results obtained during the exploration step considerably. In **RQ4**, we compare our best results, i.e., test cases with highest (worst) objective function values, with the existing test cases used in practice.

3.3.2 Case Studies

To perform our experiments, we applied our approach in Figure 3.5 to two case studies: A simple publicly available case study (DC Motor controller), and a real case study from Delphi (SBPC). Having one publicly available case study allows other researchers to compare their work with ours and to replicate our study. This is the main reason we included the DC Motor case study, even if it is simpler and less interesting than SBPC.

- **DC Motor Controller**: This case study consists of a Simulink PID Controller block (controller model) connected to a simple model of a DC Motor (plant model). The case study is taken from a Matlab/Simulink tutorial provided by MathWorks [The MathWorks Inc., 2009]. The controller model in this case study essentially controls the speed of a DC Motor. Specifically, the controller controls the voltage of the DC Motor so that it reaches a desired angular velocity. Hence, the desired and actual values (see Figure 3.1(a)) represent the desired and actual angular velocities of the motor, respectively. The angular velocity of the DC Motor is a float value bounded within $[0...160]$.
- **Suppercharger Bypass Position Controller**: Supercharger is an air compressor blowing into a turbo-compressor to increase the air pressure supplied to the engine and, consequently, increase the engine torque at low engine speeds. The air pressure can be rapidly adjusted by a mechanical bypass flap. When the flap is completely open, supercharger is bypassed and the air pressure is minimum. When the flap is completely closed, the air pressure is maximum. ***S**upercharger **B**ypass Flap **P**osition **C**ontroller (SBPC)* is a component that determines the position of the bypass flap to reach to a desired air pressure. In SBPC, the desired and actual values (see Figure 3.1(a)) represent the desired and actual positions of the flap, respectively. The flap position is a float value bounded within $[0...1]$ (open when 0 and closed when 1.0).

The DC Motor controller, the SBPC controller, and their corresponding plant models are all imple-

Table 3.1. Size and complexity of our case study models.

Model features	DC Motor (M)		SBPC (M)	
	Controller	Plant	Controller	Plant
Blocks	8	13	242	201
Levels	1	1	6	6
Subsystems	0	0	34	21
Input Var.	1	1	21	6
Output Var.	1	2	42	7
LOC	150	220	8900	6700

Table 3.2. Requirements parameters and simulation time for the DC Motor and SBPC case studies

Requirements Parameters	DC Motor	SBPC
Stability	$t_1 = 3.6s$	$t_1 = 0.8s$
Smoothness Normalized Smoothness	$v_1 = 8$	$v_1 = 0.05$
Responsiveness	$v_3 = 4.8$	$v_3 = 0.03$
Observation Time	$T = 8s$	$T = 2s$
Actual Simulation Running Time on Amazon	$50ms$	$31s$

mented in Matlab/Simulink. Table 3.1 provides some metrics representing the size and complexity of the Simulink models for these two case studies. The table shows the number of Simulink blocks, hierarchy levels, subsystems, and input/output variables in each controller and in each plant model. In addition, we generated C code from each model using Matlab auto-coding tool [The MathWorks Inc., 2011], and have reported the (estimated) number of lines of code (excluding comments) generated from each model in the last row of Table 3.1.

Note that the desired and actual angular velocities of the DC Motor, and the desired and actual bypass flap positions are among the input and output variables of the controller and plant models. Recall that desired values are input variables to controller models, and actual values are output variables of plant models. SBPC models have several more input/output variables representing configuration parameters.

3.3.3 Experiments Setup.

Before running the experiments, we need to set the controller requirements parameters introduced in Figure 3.3. Table 3.2 shows the requirements parameter values, the observation time T used in our experiments, and the actual simulation times of our case study models. For SBPC, the requirements parameters were provided as part of the case study, but for DC Motor, we chose these parameters based on the maximum value of the DC Motor speed. Specifically, T is chosen to be large enough so that the actual value can stabilize at the desired value. Note that as we discussed in Section 3.1.2, since we do not have pass/fail conditions, we do not specify v_2 from the smoothness and t_2 from the responsiveness properties.

We ran the experiments on Amazon micro instance machines which are equal to two Amazon EC2 compute units. Each EC2 compute unit has a CPU capacity of a 1.0-1.2 GHz 2007 Xeon processor. A single 8-second simulation of the DC Motor model and a single 2-second simulation of the SBPC Simulink model (e.g., Figure 3.7(b)) respectively take about 50 msec and 31 sec on the Amazon machine (See Table 3.2 last row).

Table 3.3. Parameters for the Exploration step

Parameters for Exploration	DC Motor	SBPC
Size of search space	$[0..160] \times [0..160]$	$[0..1] \times [0..1]$
HeatMap dimensions	8×8	10×10
Number of points per region (N in Figure 3.6(a))	10	10

We now discuss the parameters of the exploration and search algorithms in Figures 3.6(a) and 3.7(a). Table 3.3 summarizes the parameters we used in our experiment to run the exploration algorithm. Specifically, these include the size of the search space, the dimensions of the HeatMap diagrams, and the minimum number of points that are selected and simulated in each HeatMap region during exploration. Note that the input search spaces of both case studies are the set of floating point values within the search spaces specified in Table 3.3 first row.

We chose the HeatMap dimensions and the number of points per region, i.e., the value of N, (lines 2 and 3 in Table 3.3) by balancing and satisfying the following criteria: (1) The region shades should not change across different runs of the exploration algorithms. (2) The HeatMap regions should not be so fine grained such that we have to generate too many points during exploration. (3) The HeatMap regions should not be too coarse grained such that the points generated within one region have drastically different objective function values.

For both case studies, we decided to generate at least 10 points in each region during exploration ($N = 10$). We divided the search space into 100 regions in SBPC (10×10), and into 64 regions in DC Motor (8×8), generating a total of at least 1000 points and 640 points for SBPC and DC Motor, respectively. We executed our exploration algorithms a few times for SBPC and DC Motor case studies, and for each of our four objective functions. For each function, the region shades remained completely unchanged across the different runs. In all the resulting HeatMap diagrams, the points in the same region have close objective function values. On average, the variance over the objective function values for an individual region was small. Hence, we conclude that our selected parameter values are suitable for our case studies, and satisfy the above three criteria.

Table 3.4 shows the list of parameters for the search algorithms that we used in the second step of our work, i.e., HC, HCRR, and SA. Here, we discuss these parameters and justify the selected values in Table 3.4:

Number of Iterations (K): We ran each single-state search algorithm for 100 iterations, i.e., $K = 100$ in Figure 3.7(a). This is because the search has always reached a plateau after 100 iterations in our experiments. On average, iterating each of HC, HCRR, and SA for 100 times takes a few seconds for DC Motor and around one hour for SBPC on the Amazon machine. Note that the time for each iteration is dominated by the model simulation time. Therefore, the time required for our experiment was roughly equal to multiplying 100 by the time required for one single simulation of each model identified in Table 3.2 (50 msec for DC Motor and 31 sec for SBPC).

Exploitative and Explorative Tweak (σ): Recall that in Section 3.2.2, we discussed the need for having two Tweak operators: One for exploration, and one for exploitation. Specifically, each Tweak operator is characterized by a normal distribution with μ (mean) and σ (variance) values from which random values are selected. We set $\mu = 0$ in our experiment. For an exploitative Tweak, we choose $\sigma = 2.0$ for DC Motor, and $\sigma = 0.01$ for SBPC. As intended, with a proba-

Table 3.4. Parameters for the Search step

Single-State Search Parameters		**DC Motor**	**SBPC**
Number of Iterations (HC, HCRR, SA)		100	100
Exploitative Tweak (σ) (HC, HCRR, SA)		2	0.01
Explorative Tweak (σ) (HC)		-	0.03
Distribution of Restart Iteration Intervals (HCRR)		$U(20,40)$	$U(20,40)$
Initial Temperature (SA)	Stability	0.0220653	0.000161
	Smoothness	12.443589	0.0462921
	Normalized Smoothness	0.08422266	0.1197671
	Responsiveness	0.0520161	0.0173561
Schedule (SA)	Stability	0.0002184	0.0000015940594
	Smoothness	0.12320385	0.00045833762
	Normalized Smoothness	0.00083388772	0.0011858129
	Responsiveness	0.00051501089	0.00017184257

bility of 99% the result of tweaking a point in the center of a HeatMap region stays inside that region in both case studies. Obviously, this probability decreases when the point moves closer to the borders. In our search, we discard the result of Tweak when it generates points outside of the regions, and never generate simulations for them. In addition, with these values for σ, the search tends to be exploitative. Specifically, the Tweak has a probability of 70% to modify individual points' dimensions within a range defined by σ.

To obtain an explorative Tweak operator, we triple the above values for σ. Note that in our work, we use the explorative Tweak option only with the HC algorithm. HCRR and SA are turned into explorative search algorithms using *restart* and *temperature* options discussed below. In addition, in the DC Motor case study, we do not need an explorative Tweak operator because all the HeatMap regions belong to regular regions for which an exploitative Tweak is expected to work best.

Restart for HCRR: HCRR is similar to HC except that from time to time it restarts the search from a new point in the search space. For this algorithm, we need to determine how often the search is restarted. In our work, the number of iterations between each two consecutive restarts is randomly selected from a uniform distribution between 20 and 40, denoted by $U(20,40)$.

Initial Temperature and Schedule: The SA algorithm requires a temperature that is initialized at the beginning of the search, and is incremented iteratively based on the value of a schedule parameter. The values for the temperature and schedule parameters should satisfy the following criteria [Luke, 2013]: (1) The initial value of temperature should be comparable with differences between the objective function values of pairs of points in the search space. (2) The temperature should converge towards zero without reaching it. We set the initial value of temperature to be the standard deviation of the objective function values computed during the exploration step. The schedule is then computed by dividing the initial value of temperature by 101, ensuring that the final value of temperature after 100 iterations does not become equal to zero.

3.3.4 Results Analysis

RQ1. How does adaptive random search perform compared to naive random search in generating HeatMap diagrams? To answer this question, we compare (1) the HeatMap diagrams generated by naive random search and adaptive random search, and (2) the time these two algorithms take to

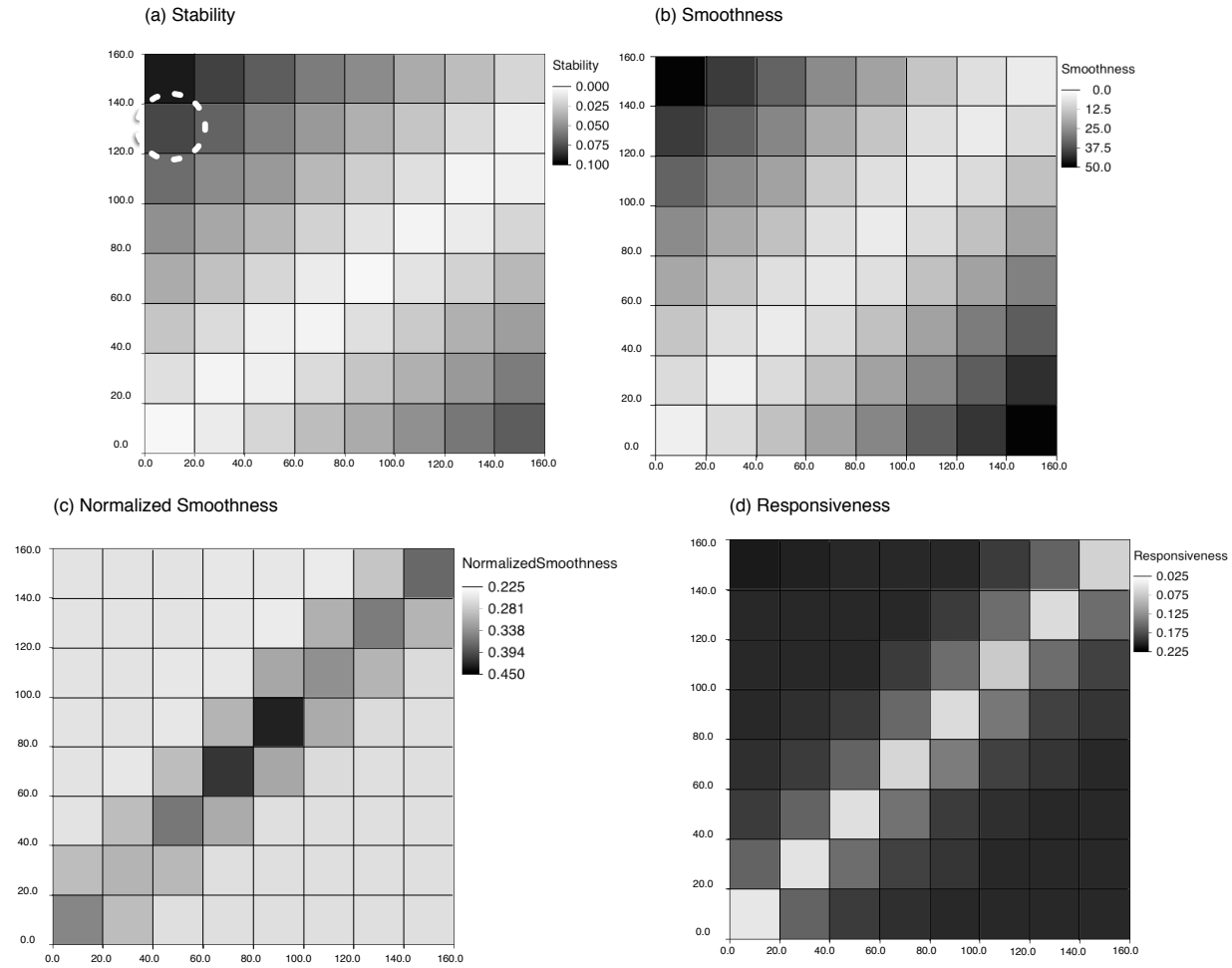


Figure 3.9. HeatMap diagrams generated for DC Motor for the (a) Stability, (b) Smoothness, (c) Normalized Smoothness and (d) Responsiveness requirements. Among the regions in these diagrams, we applied our single-state search algorithms to the region specified by a white dashed circle.

generate their output HeatMap diagrams.

For each of our case studies, we compared three HeatMap diagrams randomly generated by naive random search with three HeatMap diagrams randomly generated by adaptive random search. Specifically, we compared the region colors and value ranges related to each color. We noticed that all the HeatMap diagrams related to DC Motor (resp. SBPC) were similar. Hence, we did not observe any differences between these two algorithms by comparing their generated HeatMap diagrams.

Figures 3.9 and 3.10 represent example sets of HeatMap diagrams generated for DC Motor and SBPC case studies, respectively. In each figure, there are four diagrams corresponding to our four objective functions. Note that as we discussed above, because the HeatMap diagrams generated by random search and adaptive random search were similar, we have shown only one set of diagrams for each case study here.

In order to compare the speed of naive random and adaptive random search algorithms in generating HeatMap diagrams, we ran both algorithms 100 times to account for their randomness. For each

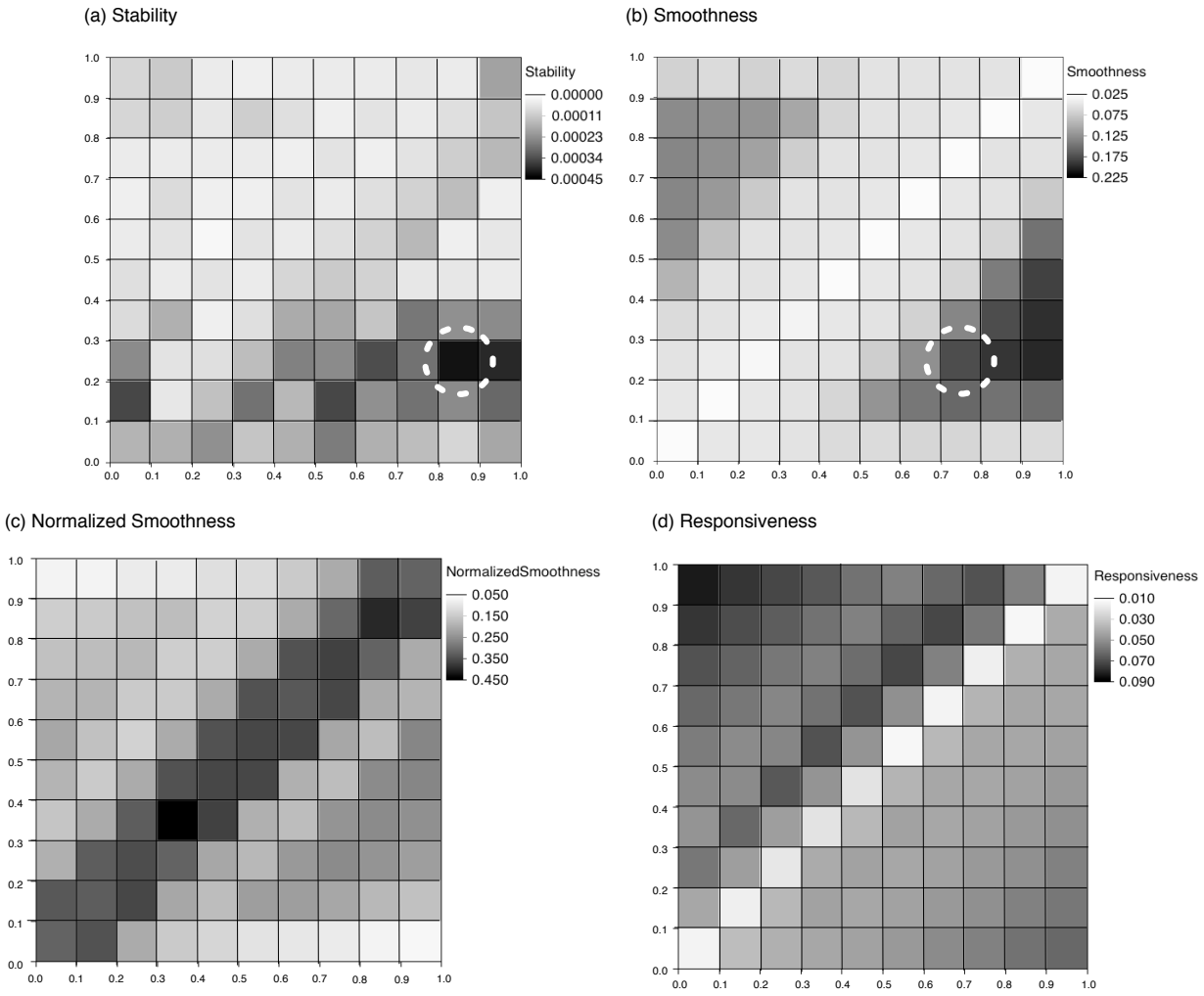


Figure 3.10. HeatMap diagrams generated for SBPC for the (a) Stability, (b) Smoothness, (c) Normalized Smoothness and (d) Responsiveness requirements. Among the regions in these diagrams, we applied our single-state search algorithms to the two regions specified by white dashed circles.

run, we recorded the number of iterations that each algorithm needed to generate at least N points in each HeatMap region. Figure 3.11(a) shows the distributions of the number of iterations obtained by applying these two algorithms to DC Motor, and Figure 3.11(b) shows the distributions obtained when the algorithms are applied to SBPC. As shown in the figures, in both case studies, adaptive random search was considerably faster (required fewer iterations) than naive random search. On average, for DC Motor, it took 1244 iterations for naive random search to generate HeatMap diagrams, while Adaptive random search required 908 iterations on the same case study. For SBPC, the average number of iterations for naive random search and adaptive random search were 2031 and 1477, respectively.

Recall from Table 3.2 that a single model simulation for DC Motor takes about 50 ms. Hence, an average run of adaptive random search is about a few seconds faster than that of naive random search for DC Motor. This speed difference significantly increases for SBPC, which is a more realistic case, where the average running time of naive random search is about five hours more than that of adaptive

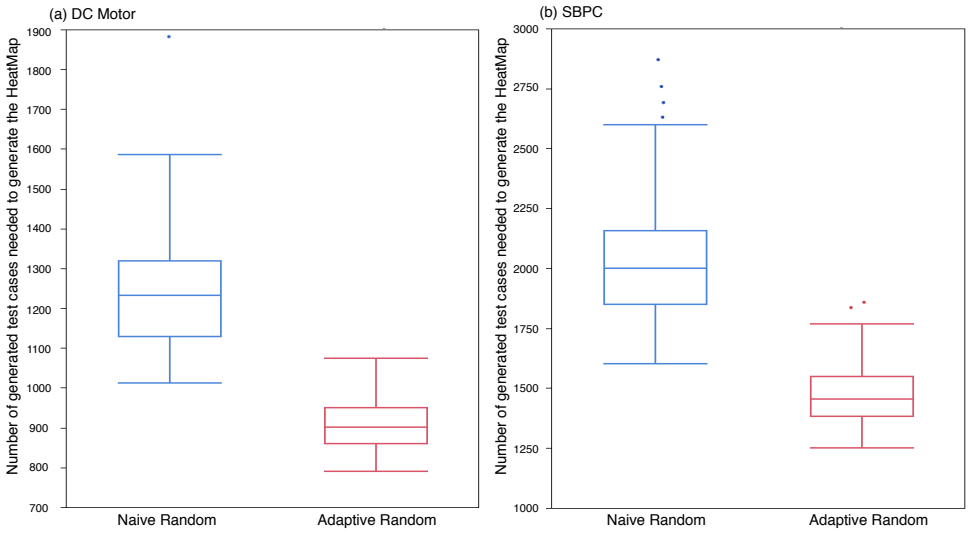


Figure 3.11. Comparing execution times of naive random search and adaptive random search algorithms for HeatMap diagram generation (the Exploration step).

random search. Based on results, given realistic cases and under time constraints, adaptive random search allows us to generate significantly more precise HeatMap diagrams (if needed), within a shorter time.

RQ2. How do our single-state search algorithms (i.e., HC, HCRR, and SA) compare with one another in identifying the worst-case test scenarios? How do these algorithms compare with random search (baseline)? To answer this question, we compare the performance of our three different single-state search algorithms, namely HC, HCRR, and SA, which we used in the search step of our framework. In addition, we compare these algorithms with random search which is used as a baseline of comparison for search-based algorithms. Recall that in section 3.2.2, we identified two different groups of HeatMap regions. Here, we compare the performance of HC, HCRR, SA, and random search in finding worst-case test scenarios for both groups of HeatMap regions, separately. As shown in Figure 3.7(a) and mentioned in section 3.2.2, for each region, we start the search from the worst point found in that region during the exploration step. This not only allows us to reuse the results from the exploration step, but also makes it easier to compare the performance of the single-state search algorithms as these algorithms all start the search from the same point in the region and the same objective function value.

We selected three regions from the set of high risk regions of each one of the HeatMap diagrams in Figures 3.9 and 3.10, and applied HC, HCRR, SA, and random search to each of these regions. In total, we applied each single-state search algorithm to 15 regions from DC Motor and to 15 regions from SBPC. For DC Motor, we selected the three worst (darkest) regions in each HeatMap diagram, and for SBPC case study, the domain expert chose the three worst (darkest) regions among the critical operating regions of the SBPC controller.

We noticed that all the HeatMap regions in the DC Motor case study were from group regular with a clear gradient from light to dark. Therefore, for the DC Motor regions, we ran our single-state search algorithms only with the exploitative parameters in Table 3.4. For the SBPC case study, we

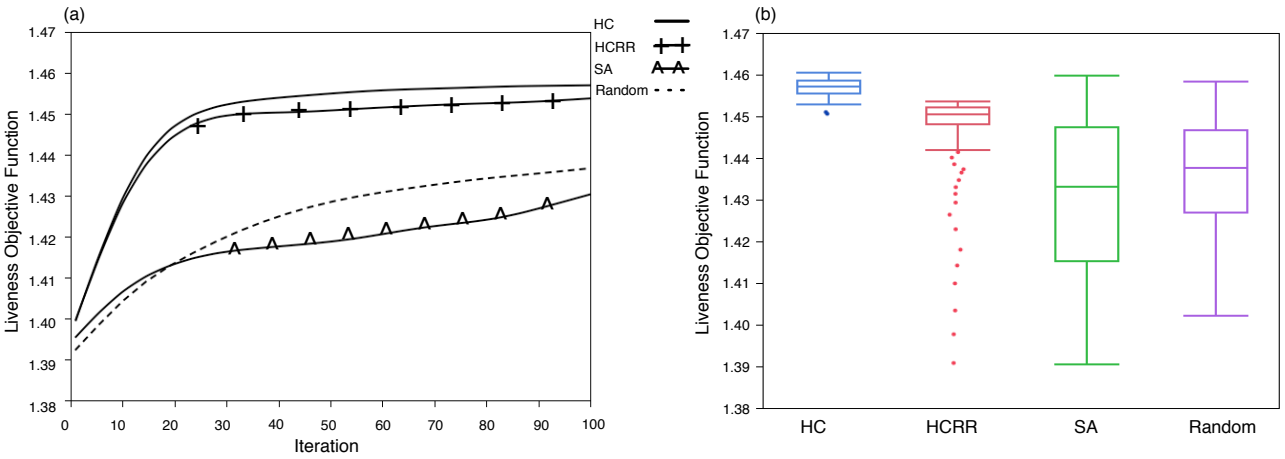


Figure 3.12. The result of applying HC, HCRR, SA and random search to a regular region from DC Motor (specified by a dashed white circle in Figure 3.9(a)): (a) The averages of the output values (i.e., the highest objective function values) of each algorithm obtained across 100 different runs over 100 iterations. (b) The distributions of the output values obtained across 100 different runs of each algorithm after completion, i.e., at iteration 100.

ran the search algorithms with the exploitative parameters for nine regions (regions from smoothness, normalized smoothness, and responsiveness HeatMap diagrams), and with the explorative parameters for three other regions (regions from stability HeatMap diagrams).

Figure 3.12 shows the results of comparing our three single-state search algorithms as well as random search using a representative region from the DC Motor case study. Specifically, the results in this figure were obtained by applying these algorithms to the region specified by a white dashed circle in Figure 3.9(a). The results of applying our algorithms to the other 14 HeatMap regions selected from the DC Motor case study were similar. As before, we ran each of these algorithms 100 times. Figure 3.12(a) shows the averages of the highest (worst) objective function values for 100 different runs of each of our algorithms over 100 iterations. Figure 3.12(b) compares the distributions of the highest (worst) objective function values produced by each of our algorithms on completion (i.e., at iteration 100) over 100 runs.

Figure 3.13 represents the results of applying our algorithms to two representative HeatMap regions from the SBPC case study. Specifically, Figures 3.13(a) and (b) show the results related to a regular region, (i.e., regions with clear gradients from light to dark), and Figures 3.13(c) and (d) represent the results related to an irregular region, (i.e., regions with several local optima and no clear gradient landscape). The former region is from Figure 3.10(c), and the latter is from Figure 3.10(a). Both regions are specified by a white dashed circle in Figures 3.10 (c) and (a), respectively. For the regular region, we executed HC, HCRR, and SA with the exploitative tweak parameter in Table 3.4, and for the irregular region, we used the explorative tweak parameter for HC from the same table.

Similar to Figure 3.12(a), Figures 3.13(a) and (c) represent the averages of the algorithms' output values obtained from 100 different runs over 100 iterations, and similar to Figure 3.12(b), Figures 3.13(b) and (d) represent the distributions of the algorithms' output values obtained from 100 different runs. As before, the results in Figures 3.13(a) and (b) were representative of the results we

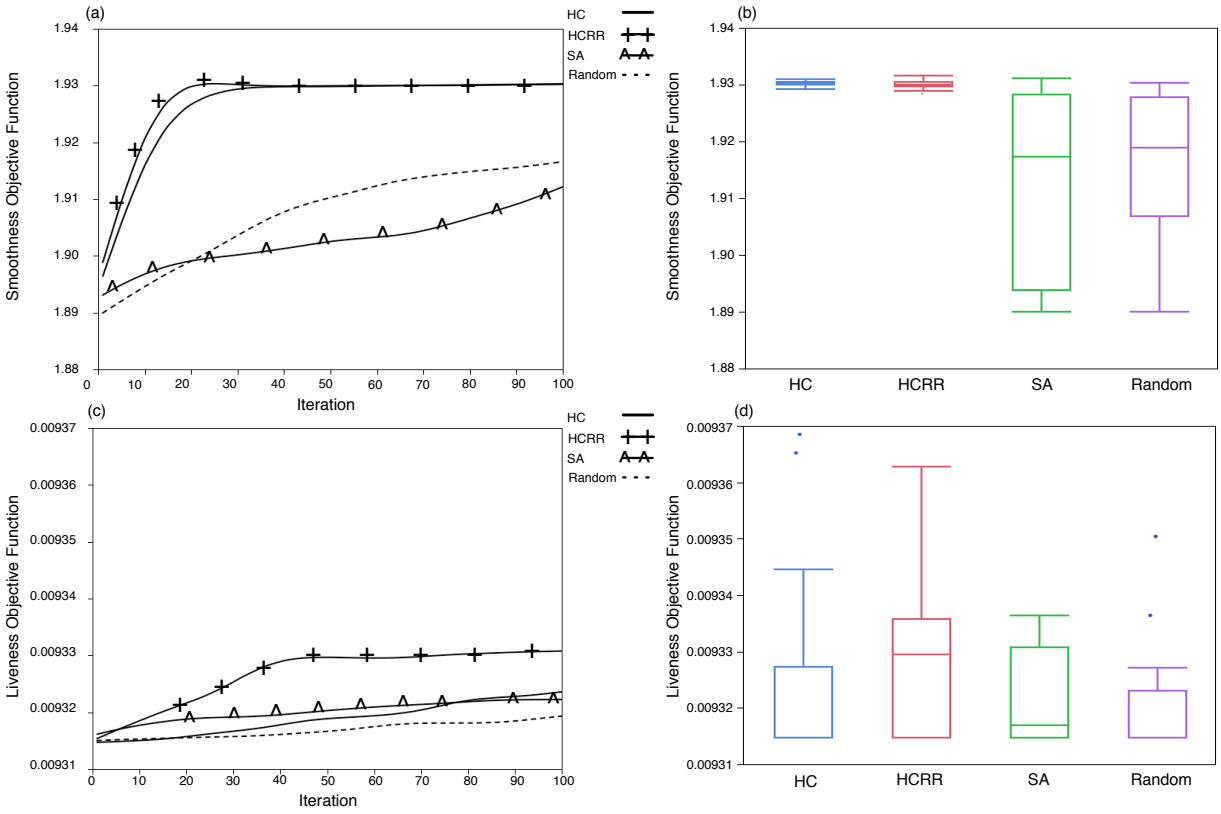


Figure 3.13. The result of applying HC, HCRR, SA and random search to a regular and an irregular region from SBPC: (a) and (c) show the averages of the output values (i.e., the highest objective function values) of each algorithm obtained across 100 different runs over 100 iterations. (b) and (d) show the distributions of the output values obtained across 100 different runs of each algorithm over 100 iterations. Diagrams (a,b) are related to the region specified by a dashed white circle in Figure 3.10(c), and Diagrams (c,d) are related to the region specified by a dashed white circle in Figure 3.10(a).

obtained from other eight SBPC regular regions in our experiment, while the results in Figures 3.13(c) and (d) were representative for the other five SBPC irregular regions.

The diagrams in Figures 3.12 and 3.13 show that HC and HCRR perform better than random search and SA for regular regions. Specifically, these two algorithms require fewer iterations to find better output values, i.e., higher objective function values, compared to SA and random search. HC performs better than HCRR on the regular region from DC Motor. As for the irregular region from SBPC, HCRR performs better than other algorithms.

To statistically compare the algorithms, we performed a *two-tailed, student t-test* [Capon, 1991] and other non-parametric equivalent tests, but only report the former as results were consistent. The chosen level of significance (α) was set to 0.05. Applying t-test to the regular region from DC Motor (Figures 3.12 (a) and (b)) resulted in the following order for the algorithms, from best to worst, with significant differences between all pairs: HC, HCRR, random search, and SA. The statistical test for both regular and irregular regions from SBPC (Figures 3.13(a) to (d)) showed that the algorithms are divided into two equivalence classes: (A) HC and HCRR, and (B) random search and SA. The p-values between pairs of algorithms in the same class are above 0.05, and the p-values between

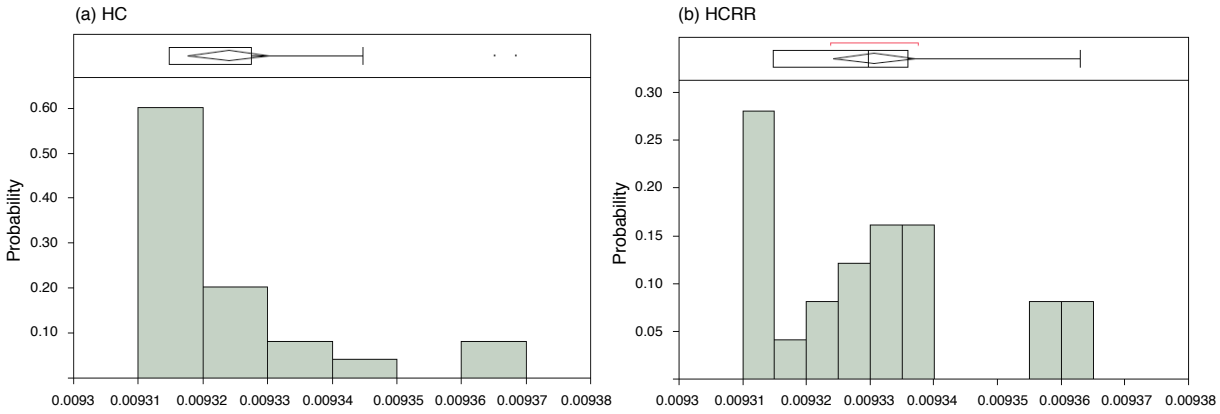


Figure 3.14. The distribution of the highest objective function values found after 100 iterations by HC and HCRR related to the region specified by a dashed white circle in Figure 3.10(a)

those in different classes are below 0.05. Table 3.5 provides the p-values comparing HC with other algorithms for regular regions, and comparing HCRR with other algorithms for an irregular region from SBPC. In addition, to computing p-values, we provide effect sizes comparing HC and HCRR with other algorithms for regular and irregular regions, respectively. In [Arcuri and Briand, 2012], it was noted that it is inadequate to merely show statistical significance alone. Rather we need to determine whether the effect size is noticeable. Therefore, in Table 3.5, we show effect sizes computed based on Cohen's d [Cohen, 1977]. The effect sizes are considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$, and high for $d \geq 0.8$.

To summarize, for regular regions exploitative algorithms work best. Specifically, HC, which is the most exploitative algorithm, performs better than other algorithms on regular regions from DC Motor. For the regular region from SBPC, HCRR manages to be as good as HC because it can reach its plateau early enough before it is restarted from a different point. Hence, HCRR and HC perform the same on the regular regions from SBPC.

For irregular regions, search algorithms that are more explorative do better. Specifically, HCRR is slightly better than HC on irregular regions from SBPC. The histogram diagrams in Figure 3.14 compare the distributions of the highest objective function values found by HC and HCRR. Specifically, with 50% probability, HCRR finds an objective function value larger than 0.00933. If we run HCRR for three times, with a probability of 87.5%, HCRR finds at least one value around or higher than 0.00933. For HC, however, the probability of finding an objective function value higher than 0.00933 is less than 20% in one run, and less than 49% in three runs. That is, even though we do not observe a statistically significant difference between the results of HC and HCRR for the irregular region of SBPC, by running these algorithms three times, we have a higher chance to find a larger value by HCRR than by HC. Finally, even though we chose SA parameters according to the guidelines in the literature (see Section 3.3.3, and [Luke, 2013]), on our case study models, SA is never better than random. This can be due to the fact that SA merely explores the space at the beginning of its search and becomes totally exploitative at the end of its search, but the exploitation is not necessarily performed close to a local optimum. However, HCRR periodically spends a number of iterations improving its candidate solution (exploitation) after each random restart (exploration). As a result, HCRR is more likely to perform some exploitation in parts of the search space close to a local optimum.

Table 3.5. The p-values obtained by applying t-test to the results in Figures 3.12 and 3.13 and the effect sizes measuring the differences between these results: Each algorithm is compared with HC (i.e., the best candidate for regular regions) for the regular DC Motor and SBPC regions, and with HCRR (i.e., the best candidate for irregular regions) for the irregular SBPC region.

Algorithm		**DC Motor Regular Region**	**SBPC Regular Region**	**SBPC Irregular Region**
p-value with HC (effect size)	HCRR	<0.0001(High)	0.2085(Low)	-
	SA	<0.0001(High)	<0.0001(High)	-
	Random	<0.0001(High)	<0.0001(High)	-
p-value with HCRR (effect size)	HC	-	-	0.0657(Medium)
	SA	-	-	0.0105(High)
	Random	-	-	0.0014(High)

RQ3. Does our single-state search algorithm (step 2 in Figure 3.5) improve the results obtained by the exploration step (step 1 in Figure 3.5)? To answer this question, we compare the output of single-state search algorithm with the output of exploration step for two regular regions from DC Motor and SBPC, and one irregular region from SBPC. Relying on RQ1 and RQ2 results, for this comparison, we use the adaptive random search algorithm for the exploration step, HC for the single-state search in regular regions, and HCRR for the single-state search in irregular regions.

Let A be the highest objective function value computed by adaptive random search in each region, and let B_i be the output (highest objective function value) of HC and HCRR at run i on regular and irregular regions, respectively. We compute the relative improvement that the search step could bring about over the results of the exploration step for run i of the search by $\frac{B_i - A}{A}$. Figure 3.15 shows the distribution of these relative improvements for the three selected regions and across 100 different runs.

The results show that the final test cases computed by our best single-state search algorithm have higher objective function values compared to the best test cases identified by adaptive random search during the exploration step. The maximum relative improvement, for regular regions, is around 5.8% for DC Motor, and 7.9% for SBPC, which is a larger and more realistic case study and has a more complex search space. This value for the irregular region from SBPC is about 3.9%.

RQ4. Does our MiL testing approach help identify test cases that are useful in practice? To demonstrate practical usefulness of our approach, we show that the test cases generated by our MiL testing approach had not been previously found by manual testing based on domain expertise. Specifically in our industry-strength case study (SBPC), we were able to generate 12 worst-case test scenarios for the SBPC controller. Figure 3.7(b) shows the simulation for one of these test scenarios concerning smoothness. Delphi engineers reviewed the simulation diagrams related to these worst-case scenarios to determine which ones are acceptable as they exhibit small deviations that can be tolerated in practice, and which ones are critical and have to be further investigated. Among the 12 worst-case test scenarios, those related to stability requirements were considered acceptable by the domain expert. The other nine test scenarios, however, indicated violations of the controller requirements. None of these critical test cases had been previously found by manual, expertise-based MiL testing by Delphi engineers. For example, Figure 3.7(b) shows an undershoot scenario around 0.2 for the SBPC con-

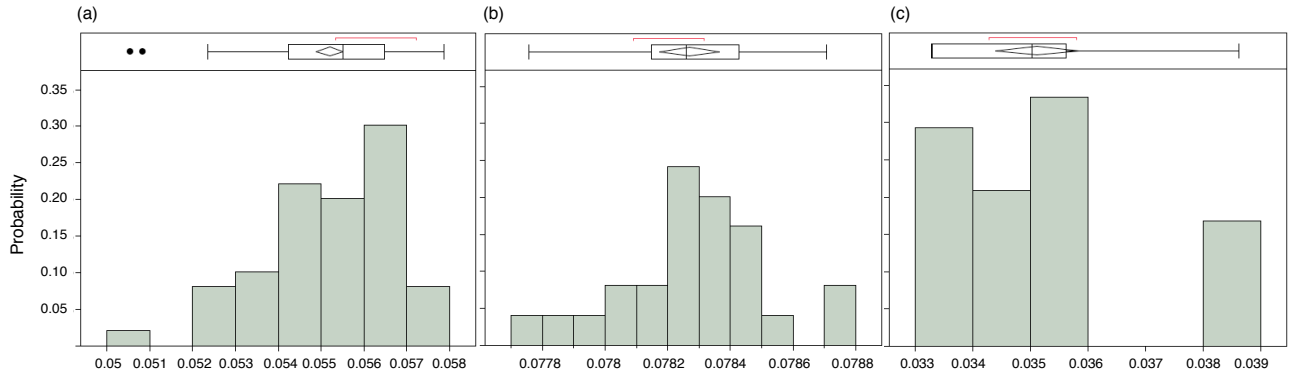


Figure 3.15. The distribution of the improvements of the single-state search output compared to the exploration output across 100 different runs of the search algorithm: (a) and (b) show the improvements obtained by applying HC to two regular regions from DC Motor and SBPC, and (c) shows the improvements obtained by applying HCRR to an irregular region from SBPC. The results are related to the three regions specified by dashed white circles in Figures 3.9(a), 3.10(a), and 3.10(c), respectively.

troller. The maximum undershoot/overshoot for the SBPC controller identified by manual testing was around 0.05. Similarly, for the responsiveness property, we found a scenario in which it takes 150*ms* for the actual value to get close enough to the desired value while the maximum corresponding value in manual testing was around 50*ms*.

3.3.5 Practical Usability

To better understand practical usability of our work, we made our case study results and tool support available to Delphi engineers through interactive tutorial sessions and our frequent meetings with them. In general, the engineers believe that our approach can help them effectively identify bugs in their controller models, and in addition, can be seen as a significant aid in designing controllers.

To receive feedback on specific output items of our work, we presented HeatMap diagrams shown in Figure 3.10 to Delphi engineers. They found the diagrams visually appealing and useful. They noted that the diagrams, in addition to enabling the identification of critical regions, can be used in the following ways: (1) The engineers can gain confidence about the controller behaviors over the light shaded regions of the diagrams. (2) The diagrams enable the engineers to investigate potential anomalies in the controller behavior. Specifically, since controllers have continuous behaviors, we expect a smooth shade change over the search space going from white to black. A sharp contrast such as a dark region immediately neighboring a light shaded region may potentially indicate an abnormal behavior that needs to be further investigated.

As discussed in response to **RQ4**, we identified 12 worst case scenarios (test cases) for SBPC in total. Nine out of these 12 test cases indicated requirements violations at the MiL level. According to Delphi engineers, the violations revealed by our nine test cases could be due to a lack of precision or errors in the controller or plant models. In order to determine whether these MiL level errors arise in more realistic situations, Delphi engineers applied these nine test scenarios at the Hardware-in-the-Loop (HiL) level where the MiL level plant model is replaced with a combination of hardware devices and more realistic HiL plant models running on a real-time simulator. This study showed that:

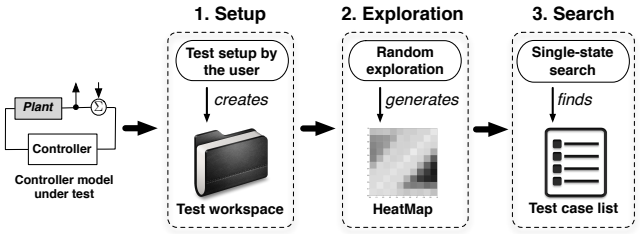


Figure 3.16. An overview of the MiL testing approach for controllers implemented in CoCoTest.

- Three (out of nine) errors that were related to the responsiveness requirement disappeared at the HiL level. This indicates that the responsiveness MiL level errors were due to lack of precision or abstractions made in the plant model, as they do not arise in a more realistic HiL environment.
- Six (out of nine) errors that were related to the smoothness and normalized smoothness requirements repeated at the HiL level. Since Delphi engineers were not able to identify defects in the controller model causing these errors, they conjecture that these errors might be due to configuration parameters of the controllers, and disappear once the controller is configured with proper parameters taken from real cars. As discussed in Section 3.1.1, we do not focus on controller configuration parameters in this chapter.

In addition, we also applied at the HiL level the six test cases out of the original 12 test cases that had passed the MiL testing stage. These six test cases were related to stability. The HiL testing results for these test cases were consistent with those obtained on MiL. That is, these test cases passed the HiL testing stage as well. To summarize, our approach is effective in finding worst-case scenarios that cannot be identified manually. Furthermore, such scenarios constitute effective test cases to detect potential errors in the controller model or in the plant model or in controller configuration parameters.

3.4 Tool Support

We have fully automated and implemented our approach in a tool called CoCoTest (`https://sites.google.com/site/cocotesttool/`). Figure 3.16 shows an overview of test generation process implemented by CoCoTest. Specifically, CoCoTest provides users with the following main functions: (1) Creating a workspace for testing a desired Simulink model. (2) Specifying the information about the input and output of the model under test. (3) Specifying the number of regions in a HeatMap diagram and the number of test cases to be run in each region. (4) Allowing engineers to identify the critical regions in a HeatMap diagram. (5) Generating HeatMap diagrams for each requirement. (6) Reporting a list of worst-case test scenarios for a number of regions. (7) Enabling users to run the model under test for any desired point in the input search space. In addition, CoCoTest can be run in a maintenance mode, allowing an advanced user to configure sophisticated features of the tool. This, in particular, includes choosing and configuring the algorithms used for the exploration and single-state search steps. Specifically, the user can choose between random search or adaptive random search for exploration, and between Hill-Climbing, Hill-Climbing with Random Restarts and Simulated Annealing for single-state search. Finally, the user can configure the specific parameters of each of these algorithms as discussed in Section 3.2.2.

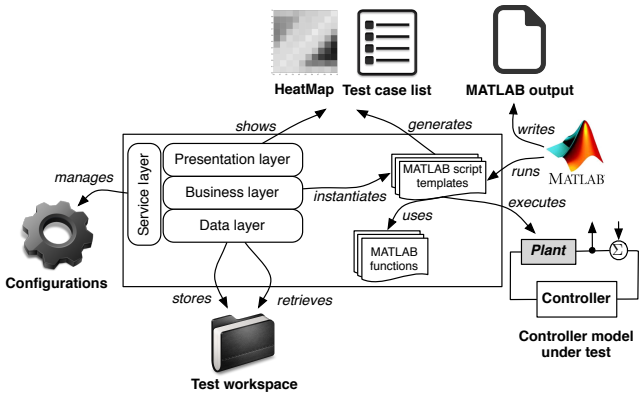


Figure 3.17. CoCoTest architectural view.

As shown in Figure 3.5, the input to CoCoTest is a controller-plant model implemented in Matlab/Simulink and provided by the user. We have implemented the five generic objective functions discussed in Section 3.1.2 in CoCoTest. The user can retrieve the HeatMap diagrams and the worst-case scenarios for each of these objective functions separately. In addition, the user can specify the critical operating regions of a controller under test either by way of excluding HeatMap regions that are not of interest, or by including those that he wants to focus on. The worst-case scenarios can be computed only for those regions that the user has included, or has not excluded. The user also specifies the number of regions for which a worst-case scenario should be generated. CoCoTest sorts the regions that are picked by the user based on the results from the exploration step, and computes the worst-case scenarios for the ones that are top in the sorting depending on the number of worst-case scenarios requested by the user. The final output of CoCoTest is five HeatMap diagrams for the five objective functions specified in the tool, and a list of worst-case scenarios for each HeatMap diagram. The user can examine the HeatMap diagram and run worst-case test scenarios in Matlab individually. Further, the user can browse the HeatMap diagrams, pick any point in the diagram, and run the test scenario corresponding to that point in Matlab.

Figure 3.17 shows the architectural view of CoCoTest. CoCoTest adopts a three-tier architecture with an extra service layer which handles the configuration management. The exploration and search algorithms are implemented in MATLAB scripts. The scripts execute model simulations to compute the objective function values. CoCoTest calls MATLAB from command line to run the MATLAB scripts. It redirects MATLAB output to a file and periodically reads the file to know when the test execution is finished and the test results are ready.

CoCoTest is implemented in Microsoft Visual Studio 2010 and Microsoft .NET 4.0. It is an object-oriented program in Visual C# with 65 classes and roughly 30K lines of code excluding comments. The main functionalities of CoCoTest have been tested with a test suite containing 200 test cases. CoCoTest requires Simulink to be installed and operational on the same machine to be able to execute controller-plant model simulations. We have tested CoCoTest on Windows XP and Windows 7, and with MATLAB 2007b. MATLAB 2007 was selected due to compatibility with Delphi models.

3.5 Conclusions

In this chapter, we identified and formalized a set of common requirements for closed-loop continuous controllers. Our proposed technique relies on a combination of explorative and exploitative search algorithms, which aim at finding worst-case scenarios in the input space with respect to the controller requirements. Our technique is implemented in a tool, named CoCoTest. We evaluated our approach by applying it to an automotive air compressor module and to a publicly available controller model. Our experiments showed that our approach automatically generates several worst-case scenarios, which can be used for testing purposes, that had not been previously found by manual testing based on domain expertise. The test cases indicated potential violations of the requirements at the MiL level, and were applied by Delphi engineers at the HiL level to identify potential discrepancies between plant models, and the HiL plant model and hardware. In addition, we demonstrated the effectiveness and efficiency of our search strategy by showing that our approach computes significantly better test cases and is significantly faster than a pure random test case generation strategy.

Chapter 4

MiL Testing of Highly Configurable Continuous Controllers: Scalable Search Using Surrogate Models

In this chapter, we extend our approach in the previous chapter to support MiL testing of all feasible configurations of continuous controllers. Specifically, we use a combination of *dimensionality reduction* [Campolongo et al., 2007] and *surrogate modeling* techniques based on supervised learning [Jin, 2011, Ong et al., 2003, Caballero and Grossmann, 2008, Douguet, 2010] to scale our search to large multi-dimensional spaces. Given an objective function, we first use dimensionality reduction techniques to identify the input variables that do not have a significant impact on the output of the objective functions, i.e., varying the values of those variables does not cause a significant change in the objective function output. We then apply an explorative random search [Arcuri and Briand, 2011] and focus the explorative search only on significant variables. Using the exploration results, we select some partitions of the input space that are more likely to include worst case input scenarios. We then apply a single-state search [Luke, 2013] to the selected partitions to identify worst case scenarios in each partition. Our objective functions require us to simulate Simulink models and are computationally expensive. Therefore, for each objective function and for each partition, we use the exploration results to build a surrogate model [Jin, 2011] based on supervised learning techniques [Witten et al., 2011]. The surrogate model is faster to compute than the objective function, and is able to predict its output within some confidence interval. Our single-state search uses the surrogate model to predict the output of the objective function when the decision as to which point the search should move to can be made based on the surrogate model.

We evaluated our approach by applying it to a complex, industrial controller, consisting of 443 Simulink blocks from the automotive domain. Our experiment showed that applying dimensionality reduction prior to exploration helps generate more accurate and predictive surrogate models for two out of three requirements. In addition, combining single-state search with surrogate modeling remarkably improves our approach for the same two requirements. Specifically, for one requirement, the search combined with surrogate modeling is eight times faster than the search without surrogate modeling, and for the other requirement, the search with surrogate modeling computes higher output values that could not be computed by the search without surrogate modeling. Finally, our approach identified critical violations of the controller requirements that had been found neither by our earlier

Controller Objective Functions	F_{st}, F_{sm}, F_r
Input Space (d=8)	$R_{ID} \times R_{FD} \times R_{Cal1} \times R_{Cal2} \times R_{Cal3} \times R_{Cal4} \times R_{Cal5} \times R_{Cal6}$ $[0..1] \times [0..1] \times [3.5..4] \times [2..4] \times [0..0.13] \times [0.3..0.7] \times [0..0.05] \times [1..1.2]$

Figure 4.1. Controller objective functions and the ranges of the input variables and configuration parameters for our industrial controller.

work [Matinnejad et al., 2015a] nor by manual testing based on domain expertise.

Organization. This chapter is organized as follows. Section 4.1 describes our extend MiL testing approach for continuous controllers to include their configuration parameters. Our experiment design and the results of our evaluation are presented in Sections 4.2 and 4.3, respectively. Finally, Section 4.4 concludes the chapter.

4.1 MiL Testing Using Search

In this Section, we describe our extended MiL testing approach to include not only initial desired (`ID`) and final desired (`FD`) variables, but also the controller configuration parameters. For example, the industrial controller used as a case study in this chapter has six configuration parameters referred to as `Cal1` to `Cal6`, respectively. The type of variables `ID`, `FD`, and `Cal1` to `Cal6` is float. We denote the range of variables `ID` and `FD` by R_{ID} and R_{FD} respectively, and the range of each `Cali` by R_{Cali}. Figure 4.1 provides value ranges for each of the configuration variables and `ID` and `FD` variables in our industrial controller. To compute highest values of F_{st}, F_{sm} and F_r for any controller configuration, our search algorithm has to handle the search space size of $|R_{ID} \times R_{FD} \times R_{Cal1} \times \ldots \times R_{Cal6}|$. Due to sheer size of the search space, we cannot effectively solve our problem by simply applying existing search algorithms. Instead, we combine *dimensionality reduction* and *surrogate modeling* techniques (based on supervised learning) to perform search in large and multi-dimensional input spaces and to reduce the cost of computing our objective functions for individual points in the search space.

Figure 4.2 shows an overview of our search-based approach to MiL testing of continuous controllers. Similar to our previous work [Matinnejad et al., 2015a], our approach is composed of an *exploration* and a *search* step. The input to our approach includes a set of objective functions, and a controller Simulink model required to compute the objective functions. Specifically, in our work, objective functions are F_{st}, F_{sm} and F_r as described in Chapter 3. The input spaces of these functions are the same and equal to the cross product of the ranges for `ID` and `FD` variables and the configuration variables. For example, the input space (objective function domains) in our case study is $R_{ID} \times R_{FD} \times R_{Cal1} \times \ldots \times R_{Cal6}$ (See Figure 4.1). The range of the objective functions is the set of real numbers $\mathbb{R}$.

In the exploration step (Figure 4.2(a)), we apply a random (unguided) search to the entire input space of the objective functions, and then based on the results we build a *regression tree* [Witten et al., 2011] partitioning the input space such that the variance of the objective function values within each partition is minimized. Before performing exploration, for each objective function, we use a dimensionality reduction strategy to identify dimensions that have the most impact on that objective

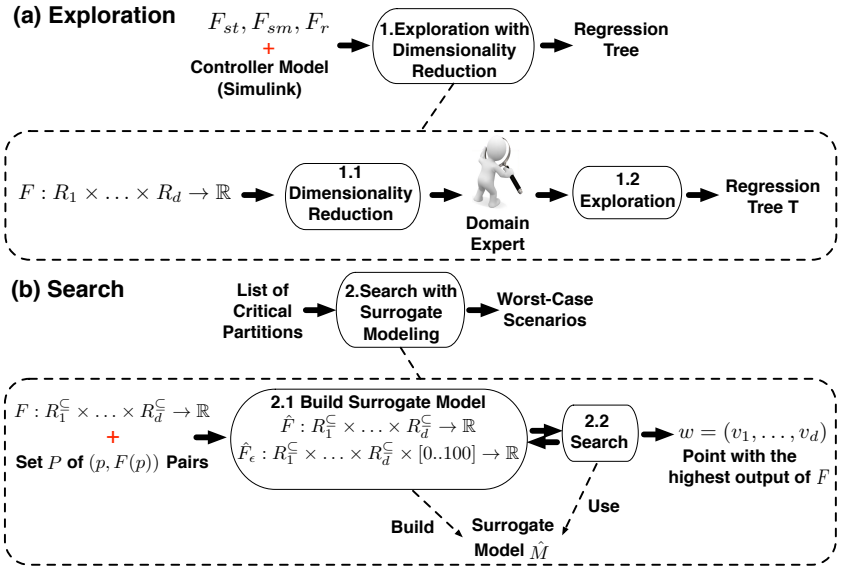


Figure 4.2. An overview of our automated approach to MiL testing of different configurations of continuous controllers: (a) Exploration step, (b) Search step.

function. This allow us to focus the exploration step only on dimensions with most impact on the objective functions, and hence, increase the scalability of our approach. The regression tree built based on the exploration results enables us to divide the space into partitions such that, for each partition, the value of the objective function is predictable within a certain confidence interval. In addition, regression trees allow the engineers to visualize partitions from a multidimensional space. We then use regression trees to identify higher risk partitions, i.e., those partitions that contain input values that are likely to violate controller requirements. Regression trees replace the heatmap diagrams we used in the previous chapter, which can no longer be used for a space that has more than three dimensions. From the regression trees, we select the partitions with the highest mean for the objective function, which are considered to be higher risk as they are more likely to contain critical errors.

In the search step (Figure 4.2(b)), we focus our search on the selected partitions and employ single-state search algorithms to identify, within those partitions, the worst case scenarios to test the controller. In this step, we build surrogate models to minimize the need for running simulations of Simulink controller models so as to make the search more scalable. Based on our previous experience, the main cause of computation time of our search is such simulations. Recall that to compute objective functions, we have to simulate the input controller model. In this work, for each objective function and for each input space partition, we create a surrogate model that predicts the objective function values within that partition. We use the exploration results related to each partition to build surrogate models. A surrogate model built for an objective function is able to predict the output of that function within a confidence interval. Using this model, our single-state search can determine whether the decision as to which point the search has to move to can be made without resorting to running simulations or not. In Sections 4.1.1 (Exploration) and 4.1.2 (Search), we describe the first and second steps of our approach, respectively.

4.1.1 Exploration

Figure 4.2(a) shows the exploration step of our approach. The input of this step is an objective function $F : R_1 \times \ldots \times R_d \rightarrow \mathbb{R}$. Function F can be any of the objective functions in Figure 4.1. The goal of this step is to efficiently select a set of points in the space of $R_1 \times \ldots \times R_d$ and compute the output of F for each point in this set. The output of this step is used to identify critical parts of the $R_1 \times \ldots \times R_d$ space (i.e., those partitions for which F produces the most critical (highest) values), and further, to create a surrogate model for individual critical partitions that can estimate as precisely as possible the output of F for any arbitrary point in that partition.

Given an objective function F, we first use dimensionality reduction techniques to identify search input dimensions that have the least impact on the output of F. A dimension i ($1 \leq i \leq d$) has a low impact on the output of F if varying the input vector $(v_1, \ldots, v_i, \ldots, v_d)$ of F by varying v_i within the range R_i does not yield a significant change in the output of F.

In addition, this step uses *adaptive random search* [Luke, 2013] to explore the input space of F by focusing on the dimensions with most impact on F. Here, we briefly discuss adaptive random search applied to the entire input space without considering dimensionality reduction. In Section 4.1.1.2, we show how this algorithm is modified to focus on significant search dimensions only. Adaptive random search is an extension of the naive random search that attempts to maximize the euclidean distance between the selected points. Adaptive random search explores the space by iteratively selecting points in areas of the space where fewer points have already been selected. Let $R_1 \times \ldots \times R_d$ be the input space, and let P_i be the set of points selected by adaptive random search at iteration i. At iteration $i+1$, adaptive random search randomly generates a set P of candidate points in the input space. The search computes distances between each candidate point p and points already selected in P_i. Formally, for each point $p = (v_1, \ldots, v_d)$ in P, the search computes a function $dist(p, P_i)$ as follows:

$$dist(p, P_i) = \text{MIN}_{(v'_1,\ldots,v'_d) \in P_i} \sqrt{\sum_{j=1}^{d} (v_j - v'_j)^2}$$

The search algorithm then adds to P_i a point p in P such that $dist(p, P_i)$ is the largest. The algorithm terminates after generating a specific number of points. Adaptive random search is similar to *quasi-random number generators* that are available in some languages, e.g., MATLAB [The MathWorks Inc., 2003a]. Similar to our adaptive random search algorithm, these number generators attempt to generate points that are evenly distributed across the entire space. Below, we describe how we use dimensionality reduction and adaptive random search to efficiently select a set of points in the input space of F that can be utilized in the search step (Figure 4.2(b)) for building an effective surrogate model.

4.1.1.1 Dimensionality Reduction

Using dimensionality reduction, we identify the dimensions of the domain of F that have the most impact on the output of F. To do so, we rely on *sensitivity analysis* which is the study of how the variations in the outputs of a function are related to the variations in its inputs [Campolongo et al., 2007]. An application of sensitivity analysis is identifying input variables with the most and the least significant impact on a given function. Among different sensitivity analysis techniques, we use the *elementary effects* method [Morris, 1991]. This method is intuitive, and compared to other

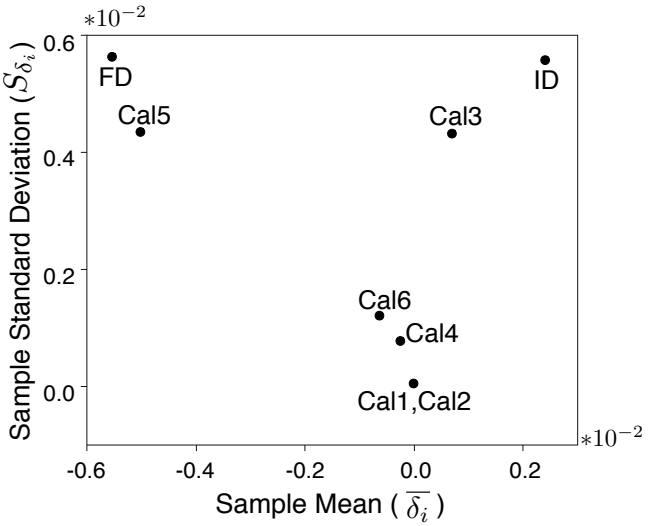


Figure 4.3. The elementary effect analysis results for the stability objective function (F_{st}) with an eight-dimension input space.

techniques such as variance-based methods [Campolongo et al., 2007], requires fewer number of function evaluations, and hence, is well-suited for functions that are expensive to compute.

The elementary effects method works as follows: Using adaptive random search, we generate r points in the input space of $R_1 \times \ldots \times R_d$. For each dimension i ($1 \leq i \leq d$), and for each point j, we vary v_i in the input vector $(v_1, \ldots, v_d)$ of F by a parameter Δ and measure the resulting variation δ_{ij} in the output of F. We then compute the sample mean $\overline{\delta_i}$, and the sample standard deviation S_{δ_i} for each input space dimension i to assess the impact of that dimension on F. Figure 4.3 shows an example output of the elementary effects method for the stability objective function F_{st} with an eight-dimension input space. Provided with this diagram, engineers choose the dimensions with significant impact on F_{st}. For example, they may decide that `Cal1`, `Cal2`, `Cal4`, and `Cal6` (which have sample means and standard deviations close to zero) are not significant.

4.1.1.2 Exploration in the Reduced Dimensional Space

To explore the input space, we use the adaptive random search algorithm as described at the beginning of Section 4.1.1. The difference is that we modify the *dist* function to ensure that the search maximizes diversity along the dimensions with significant effect on F. Otherwise, note that exploration with and without dimensionality reduction take about the same time and operate in the same space. Let D_r be the set of dimensions with significant impact on F, e.g., D_r in Figure 4.3 is $\{\texttt{ID}, \texttt{FD}, \texttt{Cal3}, \texttt{Cal5}\}$. For each candidate point p, we compute $dist_{D_r}(p, P_i)$ as follows:

$$dist_{D_r}(p, P_i) = \text{MIN}_{(v'_1,\ldots,v'_d)\in P_i} \sqrt{\sum_{j\in D_r} (v_j - v'_j)^2}$$

In contrast to the *dist* function presented at the beginning of Section 4.1.1, in $dist_{D_r}$, we consider only the values related to the dimensions in D_r. That is, we focus on maximizing the diversity of the selected points along the dimensions in D_r, and the values of the variables along the dimensions in

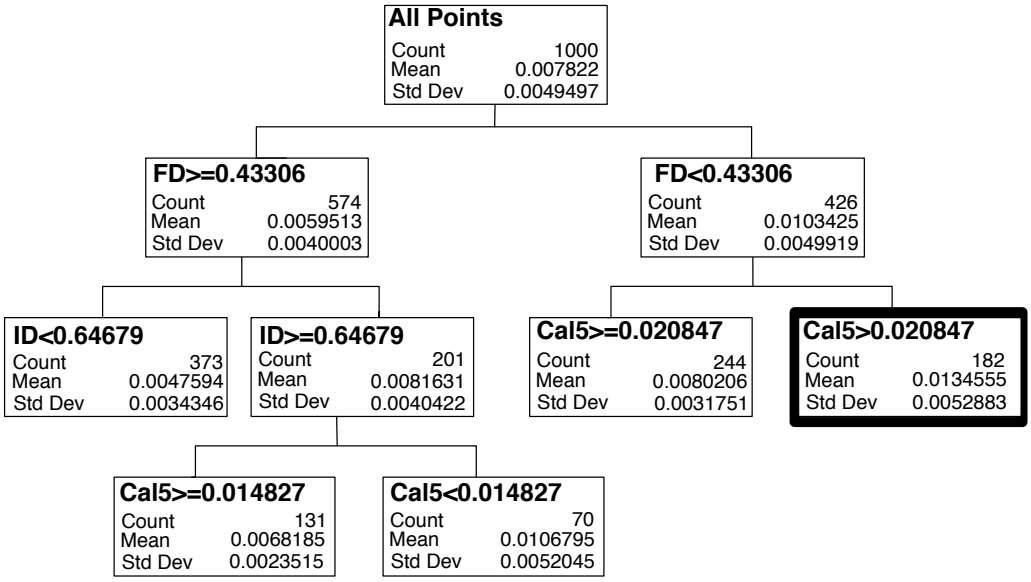


Figure 4.4. An example of a regression tree generated for F_{st}.

$\{1,\ldots,d\} \setminus D_r$ can either be fixed or set arbitrarily. This allows us to intensively explore parts of the space that results in the most variations in the output of F.

Having selected N points in the input space of F and having computed F for each point, we build a *regression tree* based on these points, e.g., see Figure 4.4. The regression tree represents a stepwise partition of the input space aimed at getting increasingly homogeneous partitions with respect to F. Such a representation is a convenient and intuitive way to visualize the impact of input space dimensions on F [Witten et al., 2011].

Figure 4.4 shows an example of the regression tree generated from the exploration results for F_{st} ($N = 1000$). Each node in the tree corresponds to a space partition and is labeled by the number of the points in that partition as well as the mean and standard deviation of the values of F_{st} for those points. For example, the highlighted node in Figure 4.4 corresponds to a partition where $min_{FD} \leq \texttt{FD} < 0.43306$ and $0.020847 < \texttt{Cal5} \leq max_{Cal5}$, and it includes 182 points selected during exploration. The mean and standard deviation of F_{st} for these points are 0.0134555 and 0.0052883, respectively.

We select the partition with the highest mean value for the objective function among the leaf nodes of the regression tree. The partition with the highest mean is more likely to include errors and critical scenarios. We denote input space partitions by $\mathcal{R}^{\subseteq}$, and define it as $\mathcal{R}^{\subseteq} = R_1^{\subseteq} \times \ldots \times R_d^{\subseteq}$ such that $R_i^{\subseteq} \subseteq R_i$ for $1 \leq i \leq d$. For example, in Figure 4.4, we select the highlighted partition that has the highest mean value, and denote it by

$$\mathcal{R}^{\subseteq} = R_{ID}^{\subseteq} \times R_{FD}^{\subseteq} \times R_{Cal1}^{\subseteq} \times R_{Cal2}^{\subseteq} \times R_{Cal3}^{\subseteq} \times R_{Cal4}^{\subseteq} \times R_{Cal5}^{\subseteq} \times R_{Cal6}^{\subseteq}$$

such that $R_{FD}^{\subseteq} = [min_{FD}..0.43306)$, $R_{Cal5}^{\subseteq} = (0.020847..max_{Cal5}]$, and $R_v^{\subseteq} = R_v = [min_v..max_v]$ for every other variable v. After selecting a partition $\mathcal{R}^{\subseteq}$, this partition together with the points generated during exploration inside this partition are passed to step 2.

4.1.2 Search

Figure 4.2(b) shows the search step of our approach. The search step takes as input a function F, a partition $\mathcal{R}^{\subseteq}$, and a set P of $(p, F(p))$ pairs computed during the exploration step. Specifically, the space $\mathcal{R}^{\subseteq}$ is a critical input space partition identified in the previous step, and P includes pairs of $(p, F(p))$ where p is a point in $\mathcal{R}^{\subseteq}$ that was selected by the adaptive random search in step **1.2**, and $F(p)$ is the output of F applied to p. We refer to P as an *observation set*. In step **2.1** in Figure 4.2(b), we first build a surrogate model $\hat{M}$ using *supervised learning* techniques [Witten et al., 2011]. The surrogate model $\hat{M}$ consists of two functions: A predictive function $\hat{F} : \mathcal{R}^{\subseteq} \rightarrow \mathbb{R}$ that estimates the output of F for any point in $\mathcal{R}^{\subseteq}$, and an error function $\hat{F}_{\varepsilon} : \mathcal{R}^{\subseteq} \times [0..100] \rightarrow \mathbb{R}$. For each point $p \in \mathcal{R}^{\subseteq}$ and a given confidence level cl, $\hat{F}_{\varepsilon}(p, cl)$ is the prediction error indicating that, with a confidence level of cl, the actual value of $F(p)$ is within $\hat{F}(p) \pm \hat{F}_{\varepsilon}(p, cl)$.

In step **2.2**, we search the points in $\mathcal{R}^{\subseteq}$ to find a point that maximizes F. Since computing F is expensive, we expedite the search by checking whether we can conclusively decide the next point that the search should move to using $\hat{F}$. Specifically, the output of $\hat{F}$ is conclusive, if the prediction error $\hat{F}_{\varepsilon}$ is less than the difference between the output of $\hat{F}$ and the existing highest value found by the search. Otherwise, we have to compute the actual output of F by simulating the Simulink controller related to F.

4.1.2.1 Surrogate Modeling

We use supervised learning techniques to build a surrogate model of the function F. Given a point p, supervised learning predicts $F(p)$ using a set P of observations with known output values [Witten et al., 2011]. We divide the observation set P into a *training* set and a *test* set. The training set is used to infer a predictive function $\hat{F}$. This is done by estimating the parameters of $\hat{F}$ such that $\hat{F}$ fits the training data as well as possible, i.e., for the observations in the training set, the differences between the output of F and that of $\hat{F}$ are minimized. The test set is then used to evaluate the accuracy of the predictions produced by $\hat{F}$ when applied to observations outside the training set.

Supervised learning techniques are categorized into *regression* and *classification* techniques where the goal is to predict real-valued and categorical outputs, respectively. We use regression techniques because F is a real-valued function. Specifically, we use the following regression techniques:

Linear Regression (LR). Linear regression assumes that F is linear. Given an observation point $p = (v_1, \ldots, v_d)$, linear regression infers $\hat{F}$ as a linear function

$$\hat{F}(v_1, \ldots, v_d) = \beta_0 + \beta_1 v_1 + \beta_2 v_2 + \ldots + \beta_d v_d$$

Exponential Regression (ER). Exponential regression assumes that F is non-linear but monotonic, and infers $\hat{F}$ in the following form:

$$\hat{F}(v_1, \ldots, v_d) = \beta_0 v_1^{\beta_1} v_2^{\beta_2} \ldots v_d^{\beta_d}$$

Polynomial Regression (PR). Polynomial regression assumes that F is neither linear nor monotonic. The inferred function $\hat{F}$ takes the form of an nth-degree polynomial:

$$\hat{F}(v_1,\ldots,v_d) = \beta_0 + \sum_{i=1}^{d} \beta_{i1} v_i + \beta_{i2} v_i^2 + \ldots + \beta_{in} v_i^n$$

In this chapter, we consider PR($n = 2,3$) because based on our experiments the predictability of our surrogate models decreases for $n > 3$. In the above three regression methods, parameters β_i ($0 \leq i \leq d$) and β_{ij} ($1 \leq i \leq d$ and $1 \leq j \leq n$) have to be estimated using the training data. The goal is to estimate these parameters such that the sum squared error of the predicted outputs for the training points is minimized. That is, $\sum_{i=1}^{m}(F(p_i) - \hat{F}(p_i))^2$ where m is the number of observations in the training set, is minimized. In addition, we use *step-wise regression* to build $\hat{F}$ in the above three regression methods [Witten et al., 2011]. Instead of including all variables v_1 to v_d at once, step-wise regression aims at selecting a minimal subset of variables that are statistically significant at explaining the variation in F. The variables may be selected in a forward or backward way. In the forward selection, variables are iteratively selected and added as long as the sum squared error over the training data decreases, i.e., the predictive power of $\hat{F}$ improves. At each iteration, a statistical test (*F-test*) is used to determine which variable best improves the predictive power of $\hat{F}$ [Capon, 1991]. Dually in the backward elimination, variables are iteratively removed as long as the predictive power of $\hat{F}$ over training data does not decrease. Most implementations of step-wise regression, e.g., *stepwiselm* in MATLAB [The MathWorks Inc., 2003c], combine the backward and forward methods by iteratively switching between them. That is, they add variables using forward selection for some iterations, and then switch to backward elimination after a while to remove unnecessary variables.

In addition to function $\hat{F}$, all the above regression methods provide a function $\hat{F}_\varepsilon : \mathcal{R}^{\subseteq} \times [0..100] \rightarrow \mathbb{R}$. Function $\hat{F}_\varepsilon(p,cl)$ estimates the prediction error for a point p based on a given confidence level cl, which is a percentage value between 0 and 100, usually above 80%. For example the input $cl = 95$ implies that the actual value of $F(p)$ lies in the interval of $\hat{F}(p) \pm \hat{F}_\varepsilon(p,cl)$ with a confidence level of 95%.

4.1.2.2 Single-State Search Using Surrogate Model

As discussed in the previous chapter to compute highest values of our objective functions, among the existing meta-heuristic search techniques, we opt for *single-state* algorithms as opposed to *population-based* ones. This is because population-based search algorithms compute fitness functions for a set of points (a population) at each iteration [Luke, 2013]. Hence, they are less likely to scale when objective functions are computationally expensive.

We propose a new Hill Climbing (HC) single-state search algorithm [Luke, 2013] extended to use surrogate models. Our algorithm speeds up the search by avoiding simulations when it is possible to decide the next move for search, based on the surrogate model predictions. The algorithm is shown in Figure 4.5. It takes as input the function F, the set P of observations in the partition $R_1^{\subseteq} \times \ldots \times R_d^{\subseteq}$, the surrogate model $\hat{M} = (\hat{F}, \hat{F}_\varepsilon)$, and a confidence level cl . The output of the algorithm is a point w with the highest output of F found in T_s seconds.

The algorithm first identifies the observation $(p, F(p)) \in P$ such that $F(p)$ is the largest in P, and sets the variable *highest* to $F(p)$ (lines 1, 2). At the beginning of each iteration of this algorithm, *highest* is the highest output of F computed so far. The algorithm then iteratively generates a new point *newp* by tweaking the current point p (line 4). The tweak operator is similar to the one used in the previous chapter. That is, we tweak a point $p = (v_1, \ldots, v_d)$ by shifting each v_i with a value x randomly selected from a normal distribution with mean $\mu = 0$ and variance $\sigma^2 = 0.1 \times |R_i^{\subseteq}|$. In our previous work, we showed that this tweak operator yields effective results for two dimension input space functions [Matinnejad et al., 2015a]

For each new point *newp*, the algorithm computes surrogate model functions $\hat{F}(newp)$ and $\hat{F}_\varepsilon(newp, cl)$ (lines 5, 6). At line 7, the algorithm determines whether it needs to compute the actual value of $F(newp)$, or it can decide the next move only using $\hat{F}(newp)$. Figure 4.6 depicts the conditions under which we have to compute $F(newp)$. Specifically, if *highest* is less than or equal to $\hat{F}(newp) - \hat{F}_\varepsilon(newp, cl)$, or more than or equal to $\hat{F}(newp) + \hat{F}_\varepsilon(newp, cl)$, with a confidence level of *cl*, *highest* is less or greater than $F(newp)$, respectively. In the case of *highest* greater than $\hat{F}(newp) + \hat{F}_\varepsilon(newp, cl)$, the search does not move to *newp*, and hence, no need to compute $F(newp)$. In the case of *highest* less than $\hat{F}(newp) - \hat{F}_\varepsilon(newp, cl)$, the search may move to *newp* depending on the value of $F(newp)$. Thus, we compute $F(newp)$. If *highest* is between $\hat{F}(newp) - \hat{F}_\varepsilon(newp, cl)$ and $\hat{F}(newp) + \hat{F}_\varepsilon(newp, cl)$, we cannot confidently compare *highest* with the actual value of $F(newp)$ using $\hat{F}(newp)$, and hence, have to compute $F(newp)$.

Line 7 in Figure 4.5 summarizes the condition for determining whether $F(newp)$ has to be computed or not. If yes, the algorithm computes $F(newp)$, and refines the surrogate model $\hat{M}$ using the new observation $(newp, F(newp))$ (lines 8–10). Otherwise, at line 11, the algorithm decides the next point that the search should move to. When it decides to move (lines 12, 13), it updates the current point p with *newp*, and *highest* with the highest value of F computed so far and stored in y. In addition, it keeps a copy of *newp*. Finally, once the loop at line 3 terminates, the algorithm reports the point w with the highest output of F found in T_s seconds.

Note that the higher the value of *cl*, the algorithm in Figure 4.5 is more likely to compute the actual value of F by running simulations. For $cl = 100$, the interval of $\hat{F}(newp) \pm \hat{F}_\varepsilon(newp, cl)$ is equal to $(-\infty, +\infty)$, and hence, the algorithm behaves like a conventional Hill Climbing algorithm and runs a simulation at each iteration. For $cl = 0$, we have $\hat{F}_\varepsilon(newp, cl) = 0$, and hence, the algorithm runs fewer simulation, i.e., only when $highest < \hat{F}(newp)$ (see Figure 4.6).

4.2 Experiment Setup

In this section, we present the research questions, some information about our industrial subject, the metrics used to evaluate surrogate models, and information about our experiment design.

4.2.1 Research Questions

RQ1 How do the different surrogate modeling techniques perform compared to one another?

RQ2 Does dimensionality reduction improve prediction accuracy of the best surrogate modeling technique identified in **RQ1**?

Algorithm.
SingleStateSearch

Input: $F : R_1^{\subseteq} \times \ldots \times R_d^{\subseteq} \rightarrow \mathbb{R}$.
The set P of $(p, F(p))$ pairs.
The surrogate model $\hat{M} : (\hat{F}, \hat{F}_\varepsilon)$.
A confidence level cl.
Output: Point $w = (v_1, \ldots, v_d)$ with the highest output of F.

1. Let $(p, f) \in P$ s.t. for all $(p', f') \in P$, we have $f \geq f'$
2. $highest = f$
3. **for** T_s seconds **do:**
4. $newp = Tweak(p)$
5. $y = \hat{F}(newp)$
6. $\varepsilon = \hat{F}_\varepsilon(newp, cl)$
7. **if** $highest < (y + \varepsilon)$ **:** /*If $highest$ is less than $\hat{F}(newp) + \hat{F}_\varepsilon(newp, cl)$, we
8. $y = F(newp)$ simulate and compute $F(newp)$, as shown in Figure 4.6.
9. $P = P \cup (newp, y)$ Otherwise, we bypass simulation.*/
10. $(\hat{F}, \hat{F}_\varepsilon) = BuildSurrogateModel(P)$
11. **if** $y > highest$ **:**
12. $highest = y$
13. $p = w = newp$
14. **return** w

Figure 4.5. Single-state Hill Climbing (HC) search algorithm with surrogate modeling.

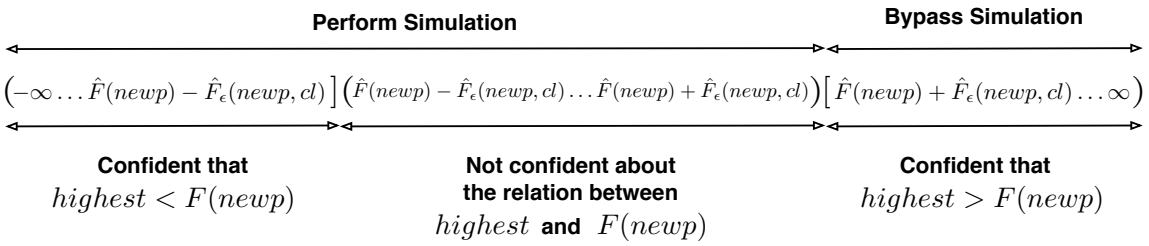


Figure 4.6. Depicting the conditions used by the algorithm in Figure 4.5 to perform or to bypass simulations.

RQ3 How do our single-state search algorithms, with and without surrogate modeling, perform compared to each other?

RQ4 Does our approach help identify testing results that are useful in practice?

In **RQ1**, for each objective function, we identify, among the four regression methods discussed in Section 4.1.2.1, the method that yields a surrogate model with highest prediction accuracy. For each objective function, we then use this best surrogate model for the single-state search. In **RQ2**, we determine whether focusing exploration on significant dimensions of each fitness function improves the prediction accuracy of the surrogate models. Recall that exploration with and without dimensionality reduction take about the same time and operate in the same space. However, the exploration results with dimensionality reduction are more diversely distributed along the dimensions with significant impact on the objective functions. The question is whether this gives rise to more accurate and predictive surrogate models? In **RQ3**, we compare the performance and results of our single state search algorithms with and without surrogate modeling, in order to determine whether our new approach scales better in large search spaces. Finally, in **RQ4**, we compare our best results, i.e., test cases with

highest objective function values, with those obtained in our previous work [Matinnejad et al., 2015a] as well as the existing test cases used in practice.

4.2.2 Industrial Subject

Supercharger is an air compressor blowing into a turbo compressor to increase the air pressure supplied to the engine. The air pressure is controlled by a mechanical bypass flap: When the flap is completely open (resp. closed), the air pressure is minimum (resp. maximum). Our industrial subject is the ***S**upercharger **B**ypass Flap **P**osition **C**ontroller (SBPC)* which determines the position of the bypass flap to achieve a desired air pressure. SBPC is one of the most complex controllers among those dedicated to engine management. In SBPC, the *desired* and *actual* variables represent the desired and actual positions of the flap, respectively. The flap position is bounded within [0..1] (0 for open and 1.0 for closed), i.e., R_{ID}=R_{FD}=[0..1] in our experiments. The SBPC controller and plant models are both implemented in Simulink and include 443 blocks in total. SBPC has six configuration parameters `Cal1` to `Cal6` impacting the PID controller terms, and hence, the controlling behavior of SBPC. Figure 4.1 shows the ranges for the SBPC configuration parameters.

4.2.3 Surrogate Modeling Evaluation Metrics

To assess the goodness of fit and predictive power of the generated surrogate models, we use two well-known evaluation metrics for supervised learning methods: (1) *coefficient of determination* or R^2 [Witten et al., 2011] and (2) *Mean of Relative Prediction Error* or MRPE [Park and Stefanski, 1998]. Specifically, R^2 measures the proportion of the total variance of F explained by $\hat{F}$ for the observations in the training set. In other words, R^2 is a measure of goodness of fit on the training set. The value of R^2 is always less than 1. The higher the value of R^2, the higher the goodness of fit of $\hat{F}$ predictions.

The *Mean of Relative Prediction Error (MRPE)* [Park and Stefanski, 1998] measures the predictive power of $\hat{F}$ using the observations from the test set. Let k be the number of observations in the test set. We compute the MRPE as follows:

$$\frac{1}{k} \times \sum_{i=1}^{k} \left| \frac{F(p_i) - \hat{F}(p_i)}{F(p_i)} \right|$$

That is, MRPE measures the average of the relative prediction errors of $\hat{F}$ for the observations in the test set. The lower the value of MRPE, the higher the predictive power of $\hat{F}$. Note that a surrogate model $\hat{F}$ with high R^2 may not yield low MRPE values due to *overfitting*, i.e., when $\hat{F}$ is excessively tailored to the training observations, but does not generalize well to the test observations. Therefore, to compare different surrogate models, we have to take into account both R^2 and MRPE metrics.

In theory, both R^2 and MRPE can be computed on either the training set or the test set. When applied to the training set, they measure the goodness of fit, and when applied to the test set, they assess the prediction accuracy. However, it is more common to use R^2 to measure goodness of fit, to get an idea of how much variance remains unexplained in the data used to fit the model, and MRPE to address prediction accuracy as it is more readily interpretable to assess the applicability of a prediction model. Therefore, we chose to compute R^2 on the *training* set and MRPE on the *test* set.

4.2.4 Experiment Design and Analysis Strategy

We implemented the elementary effect analysis, adaptive random search, surrogate modeling techniques including LR, ER, and PR($n = 2,3$), and our single-state search algorithm in MATLAB. Both adaptive random search and single-state search are required to call simulations of the SBPC Simulink model. The latter, in addition, may resort to surrogate modeling by calling and refining surrogate models. We ran each Simulink simulation for 2 sec to give SBPC enough time to stabilize. We ran all the experiments on Amazon micro instance machines which is equal to two Amazon EC2 Compute Units. Each unit has a CPU capacity of a 1.0-1.2 GHz 2007 Xeon processor. Each 2-sec Simulink simulation of SBPC takes about 31 sec on the Amazon machine. While calling the surrogate models is negligible (less than 5ms) and rebuilding them takes about 2 sec on average.

To investigate **RQ1**-**RQ4**, we designed and performed the following experiments. Below, we use abbreviations DR and SM for dimensionality reduction and surrogate models, respectively.

EXP-I. To answer **RQ1** and **RQ2**, we computed the output of exploration once with and once without DR. To compute exploration results with DR, we applied the elementary effect analysis to our three objective functions with parameters $r = 20$ and $\Delta = 0.556$. Recall from Section 4.1.1 that r is the number of points, and Δ is the size of the modification applied to each dimension of each point. These values are selected based on the guidelines in [Campolongo et al., 2007]. To account for randomness we repeated the elementary effect analysis 10 times. The results across different repetitions were consistent, and out of the eight input space dimensions, four were significant for F_{st}, and three were significant for F_r and F_{sm}. We then applied our adaptive random search to each of these functions by focusing the exploration on their significant dimensions only. We let $N = 1000$, and executed our adaptive random search to generate 1000 points for F_{st}, and 1000 points for F_r and F_{sm}. Note that since F_r and F_{sm} have the same significant dimensions, we generated the same points for both functions, but kept two output values for each point.

To compute exploration results without DR, we let $N = 2000$, and generated 2000 points across the eight dimensions of the input space, but computed F_{st}, F_r, and F_{sm} separately for each point. Note that the total number of points generated during exploration with and without DR was the same and equal to 2000.

For each objective function, we built two regression trees: one from the exploration results with DR, and one from the exploration results without DR. Each regression tree node corresponds to an input space partition. Suppose that σ and μ respectively denote the standard deviation and mean values related to each partition. We expand each regression tree until $\frac{\sigma}{\mu}$ for every leaf node falls below 0.1. Expanding the trees further often results in leaf nodes containing very few observations and corresponding to very small space partitions. By expanding the tree, the variance of the objective function values (σ) in leaf node partitions decrease. In each tree, among all the leaf nodes with $\frac{\sigma}{\mu} < 0.1$, we select the one that has the highest μ for further search of worst case scenarios.

For the partitions with the highest μ, based on their observation points, we created four surrogate models (SMs) using LR, ER, and PR ($n = 2,3$) techniques. To effectively apply surrogate modeling techniques, we want to have at least 200 points in the selected partitions. If lower, we generate additional points before building a SM. We then use these 200 points as training data to build SMs. For the test sets, we generate an additional 50 points using a naive random selection technique. That

is, the test points are chosen the same way irrespective of the use of DR. This allows us to use the same test sets to facilitate the comparison of the SMs generated with and without DR.

In total, to obtain the exploration results with and without DR, we performed 24 experiments (3 objective functions, 4 surrogate modeling techniques, with/without DR). To account for randomness, we repeated each of the 24 experiments 10 times.

EXP-II. To answer **RQ3** and **RQ4**, we performed single state search, for a given input space partition, once with and once without using SMs. To compute the search results with SM, we first built a SM using the best technique identified in **RQ1**. Then, for each objective function, we applied our SM-based single state search algorithm in Figure 4.5 to a given input space partition. We ran this algorithm for three confidence levels (cl): 80, 90, and 95. For each objective function and each confidence level, we ran the search for 3000 sec.

To compute the search results without SM, for each objective function, we applied a Hill Climbing (HC) algorithm similar to that used in our previous work [Matinnejad et al., 2015a] to a given input space partition.We let the search run for 3000s, i.e., the total search budget time for the single state search with and without SM was the same and equal to 3000s. We refer to our SM-based single state search algorithm in Figure 4.5 as HC-SM, and to our single-state search algorithm without SM as HC-NoSM.

In total, to obtain the search results with and without SM, we performed 9 experiments with SM (3 objective functions, 3 confidence levels), and 3 experiment corresponding to the 3 objective functions without SM. To account for randomness, we repeated each of the 12 experiments 30 times.

4.3 Experiment Results

This section provides responses, based on our experiments, for research questions **RQ1** to **RQ4** described in Section 4.2.1.

RQ1. To answer **RQ1**, we use the SMs generated by the **EXP-I** experiments (Section 4.2.4). Since **EXP-I** includes 24 experiments, and each experiment was repeated 10 times, we obtain 24 different groups of SMs where each group consists of 10 different SMs. For all the SMs, we compute R^2 and MRPE values. Figure 4.7(a) shows the average R^2 and average MRPE values for 12 SM groups generated for F_{sm}, F_{st}, and F_r based on the exploration results with DR. As shown in Figure 4.7(a), the SMs built using PR ($n = 3$) have the best goodness of fit (highest R^2) and best predictive accuracy (lowest MRPE). The results for the other 12 SM groups generated from the exploration results without DR are consistent with those shown in Figure 4.7(a). Specifically, our results confirm that, compared to LR, ER and PR($n = 2$), PR ($n = 3$) generates the most accurate SMs for our objective functions.

In addition, the results in Figure 4.7(a) show that while the surrogate models generated by PR ($n = 3$) for F_{sm} and F_r have high goodness of fit and predictive power, this is not the case for F_{st}. Additionally, though due to space constraints this cannot be shown here, we observed that applying PR with $n > 3$ results in less accurate SMs, i.e., lower R^2 and higher MRPE.

RQ2. To answer **RQ2**, we focus on the best SMs in the **EXP-I** experiments, i.e., the SMs generated

(a) Mean of R^2/MRPE values for different surrogate modeling techniques

	LR R^2/**MRPE**	ER R^2/**MRPE**	PR(n=2) R^2/**MRPE**	PR(n=3) R^2/**MRPE**
F_{sm}	0.66/**0.0526**	0.44/**0.0791**	0.95/**0.0203**	0.98/**0.0129**
F_r	0.78/**0.0295**	0.49/**1.2281**	0.85/**0.0247**	0.85/**0.0245**
F_{st}	0.26/**0.2043**	0.22/**1.2519**	0.46/**0.1755**	0.54/**0.1671**

(b) Distribution of MRPE for three objective functions with and without DR

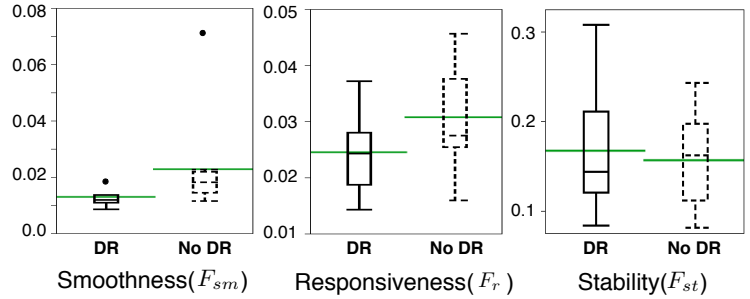


Figure 4.7. Experiment results for **RQ1** and **RQ2**: (a) Comparing different surrogate modeling techniques, and (b) comparing exploration results with and without dimensionality reduction (DR).

by PR($n = 3$). Figure 4.7(b) shows the MRPE distributions related to the best SMs generated from the exploration results both with and without DR, and for each of F_{sm}, F_r and F_{st}. To statistically compare the MRPE values, we performed the non-parametric pairwise Wilcoxon Pairs Signed Ranks test [Capon, 1991], and calculated the effect size using Cohen's d [Cohen, 1977]. 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 $d \geq 0.8$ [Cohen, 1977]. Testing differences in MRPE distributions shows that for F_{sm} and F_r, the SMs built *with* DR have significantly better predictive power than those built *without* DR. In addition, the effect size is "high" for both F_{sm} and F_r. For F_{st}, however, there is no statistically significant difference between the MRPE values of the SMs generated with and without DR. Specifically, focusing exploration on a reduced-dimension space significantly improves (with a high effect size) the predictive power of the SMs for F_{sm} and F_r, but does not have a significant impact on the predictive power of the SMs related to F_{st}. This may be due to the SMs for F_{st} being less accurate than those for F_{sm} and F_r. Since based on our results, DR never decreases the predictive power of the resulting SMs, our results suggest to focus exploration on the significant dimensions identified by DR.

RQ3. To answer **RQ3**, we use the **EXP-II** experiments (Section 4.2.4). Recall that in **EXP-II**, each run of each algorithm was executed for 3000 sec. In each run, we recorded the value of the variable *highest*, i.e., the highest found output of the objective function (see Figure 4.5), at every 100 sec time interval. Figure 4.8 compares the value distributions of *highest* obtained from HC-SM with *cl* = 80, 90, and 95, and from HC-NoSM at some selected (representative) time points for each of the objective functions F_{sm}, F_{st}, and F_r. Specifically, Figure 4.8(a) shows the value distributions of *highest* for F_{sm} at 800s, 1500s, 2500s, and 3000s. Figure 4.8(b) shows the value distributions of *highest* for F_r at 200s, 300s, and 3000s, and Figure 4.8(c) shows the value distributions of *highest* for F_{st} at 3000s. Time points, for each fitness function, were selected to make the overall trends visible for HC-SM and HC-NoSM.

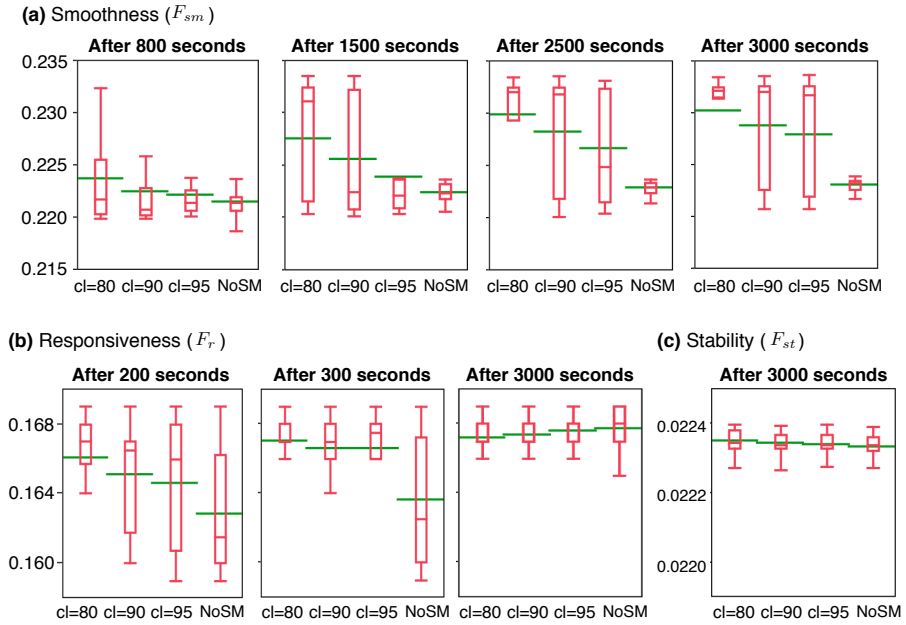


Figure 4.8. Boxplots for single-state search output values with and without surrogate modeling at some selected time points and applied to: (a) Smoothness (F_{sm}), (b) Responsiveness (F_r), and (c) Liveness (F_{st}).

Figures 4.9(a) and (b) respectively represent the differences between the mean values found for F_{sm} and F_r by each of the HC-SM algorithms and by HC-NoSM over 3000s of time. For F_{sm}, as shown in both Figures 4.8(a) and 4.9(a), within 3000s, all the HC-SM algorithms are able to find higher values compared to those found by HC-NoSM. Among the HC-SM algorithms, $cl = 80$ finds highest values for F_{sm} at 2500s with a mean of 23% and a median of 23.4%. HC-SM with $cl = 90$ and $cl = 95$ reach slightly lower values for F_{sm} at around 3000s. HC-NoSM, however, is not able to go higher than 22.3% (both mean and median) in the 3000s allotted time. That is, in 3000s and across 30 different runs, half of the values computed by HC-SM indicate an over/undershoot of 23.4% or higher, while the highest smoothness violation found by HC-NoSM is about 22.3%.

The one percent improvement of HC-SM over HC-NoSM is important in practice. This is because, depending on the hardware configuration, engineers specify a maximum over/undershoot that can be tolerated for the smoothness requirement. Exceeding this value, even slightly, is not in general acceptable. In the particular case of our case study (SBPC), slight deviation for the smoothness causes the flap to hit other hardware parts, generating noise and damaging hardware over time. We note that, even after running HC-NoSM for 5000s, its average and median output remained at around 22.3%. In general, the overall increase in the output of HC-NoSM over 5000s was very small.

For F_r, as shown in both Figures 4.8(b) and 4.9(b), the HC-SM algorithms find their highest values within the first 300s of time with $cl = 80$ being the fastest again. Specifically, at 300s, on average, HC-SM identifies a worst response time of 167s, while the average output of HC-NoSM indicates a response time of 163s. In contrast to the results of F_{sm}, for F_r, HC-NoSM is able to match the HC-SM algorithms in around 2500s of time (see the area shown by a dashed circle in Figure 4.9(b)). That is, the HC-SM algorithms are about 8 times faster than HC-NoSM. Finally, for F_{st} (Figure 4.8(c)), we did not observe any noticeable difference between the values found by the HC-SM algorithms and those found by HC-NoSM within 3000s of time. That is, within this time, all the algorithms behaved the

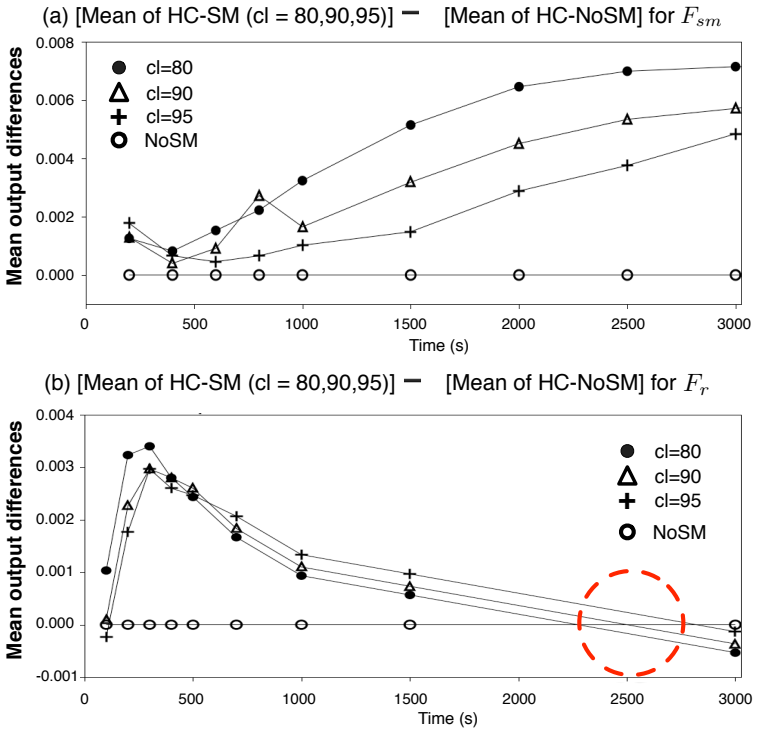


Figure 4.9. Differences of mean output values of search with surrogate modeling (HC-SM with *cl* = 80, 90, 95) and search without surrogate modeling (HC-NoSM).

same. This is due to the SM for F_{st}, which is clearly less accurate than those for F_{sm} and F_r. Hence, for F_{st}, the HC-SM algorithms almost run Simulink simulations at every search iteration, producing the same results as HC-NoSM.

We conclude that for accurate SMs with high predictive power (i.e., those built for F_{sm} and F_r), HC-SM outperforms HC-NoSM. Specifically, for F_{sm}, HC-SM computes higher output values that could not be computed by HC-NoSM, and for F_r, HC-SM is about eight times faster than HC-NoSM in finding the same output. This is because, compared to HC-NoSM, HC-SM runs fewer Simulink simulations, relying on the SM output in many search iterations. In addition, HC-SM with cl = 80 is slightly faster than HC-SM with cl = 90 and 95 because it runs fewer simulations.

RQ4. To demonstrate practical usefulness of our approach, we argue that MiL testing for *different* configurations of the SBPC controller finds requirements violations that have neither been identified via MiL testing of *fixed* configurations, nor by manual testing based on domain expertise.

Figure 4.10 compares our results with the results of our previous work on MiL testing with fixed configuration parameters [Matinnejad et al., 2015a, Matinnejad et al., 2013], and the results of manual expertise-based MiL testing. As shown in the figure, by extending our approach to test different configurations within some given ranges, we were able to identify a critical violation of the stability requirement with a deviation of 2.2%. Note that our previous results [Matinnejad et al., 2015a, Matinnejad et al., 2013] as well as the results from manual testing never indicated any stability error in SBPC. In addition, for smoothness and responsiveness, our current work found more critical violations: Specifically, the maximum observed over/undershoot was 24% compared to the 20% found by

	MiL-Testing different configurations	MiL-Testing fixed configurations	Manual MiL-Testing
Stability	2.2% deviation	-	-
Smoothness	24% over/undershoot	20% over/undershoot	5% over/undershoot
Responsiveness	170 ms response time	80 ms response time	50 ms response time

Figure 4.10. Comparing our MiL testing results with the results of MiL testing fixed controller configurations [Matinnejad et al., 2015a], and the results of manual MiL testing.

our previous work, and 5% found by manual testing. Finally, we computed a worst response time of 170*ms* compared to 80*ms* found by our previous work and 50*ms* identified via manual testing.

We conclude our results by noting that, due to limitations of manual testing, MiL testing in practice mostly focuses on a single controller configuration which is typically the one specified based on HiL configuration parameters. This obviously falls short when the controller is configured and deployed on a hardware with parameters that differ from those used on HiL. Since existing MiL testing does not consider different configurations, the errors that could have been found at MiL level go unnoticed until the very late development stages. Our work attempts to alleviate this shortcoming by enabling MiL testing for various configurations obtained by varying configuration parameters within their given ranges.

4.4 Conclusions

In this chapter we proposed an approach for MiL testing of closed-loop continuous controllers in large configuration spaces, based on meta-heuristic search, with respect to smoothness, responsiveness, and stability requirements. The main challenge is to scale search to large multi-dimensional input spaces made up of possibly many configuration parameters. To scale search, we combined techniques for dimensionality reduction and supervised learning to build surrogate models that accurately predict simulation results without resorting to simulation in many cases. Our evaluation shows that our approach is able to identify critical violations of the controller requirements that had neither been found by our test generation algorithm in chapter three nor by manual testing. Further, we showed that combining search with surrogate modeling remarkably improves our approach for two out of three requirements. Specifically, for one requirement, the search combined with surrogate modeling is eight times faster than the search without surrogate modeling, and for the other requirement, the search with surrogate modeling computes more critical requirements violations than what could be detected by the search without surrogate modeling.

Chapter 5

Effective Test Suites for Mixed Discrete-Continuous Stateflow Controllers

In this chapter, we focus on providing test case generation algorithms for Stateflow models. Given that test oracles for Stateflow models are not amenable to full automation mostly due to their complex continuous-time behaviors [Zander et al., 2012], our algorithms help engineers develop small test suites with high fault revealing power for continuous behaviors, effectively reducing the cost of human test oracles [Barr et al., 2015, McMinn et al., 2010]. Note that our approach for testing closed-loop controllers fails to automate test oracles for Stateflows because for closed-loop controllers, the environment (plant) feedback and the desired controller output (setpoint) are both available [Heimdahl et al., 2013] . Hence, test oracles could be formalized and automated in terms of feedback and setpoint. For Stateflow models, which typically implement *open-loop controllers* [Nise, 2004], the plant feedback is not available.

We present and evaluate six test generation 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 [Binder, 2000] 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 [Alshahwan and Harman, 2012, Alshahwan and Harman, 2014] to output signals of Stateflow models. Output uniqueness has been studied for web applications and has shown to be a useful surrogate to white-box generation 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 meta-heuristic search [Luke, 2013] 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 disconti-

nuity pattern captures fast, short-duration and upward or downward pulses (i.e., spikes [Wikipedia., 2016]) 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.

Organization. This chapter is organized as follows. Section 5.1 presents a mixed discrete-continuous Stateflow example that motivates our work. Section 5.2 presents our test generation algorithms for Stateflow models. Sections 5.3 and 5.4 describes our experiments setup and experiments results, respectively. Finally, Section 5.5 concludes the chapter.

5.1 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 5.1(a) represents the discrete behavior of SCC specifying that the supercharger clutch can be in two *quiescent* states: engaged 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 5.1(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 5.1(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 5.1(b) extends the discrete transition system in Figure 5.1(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 5.1(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 5.1(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 function `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

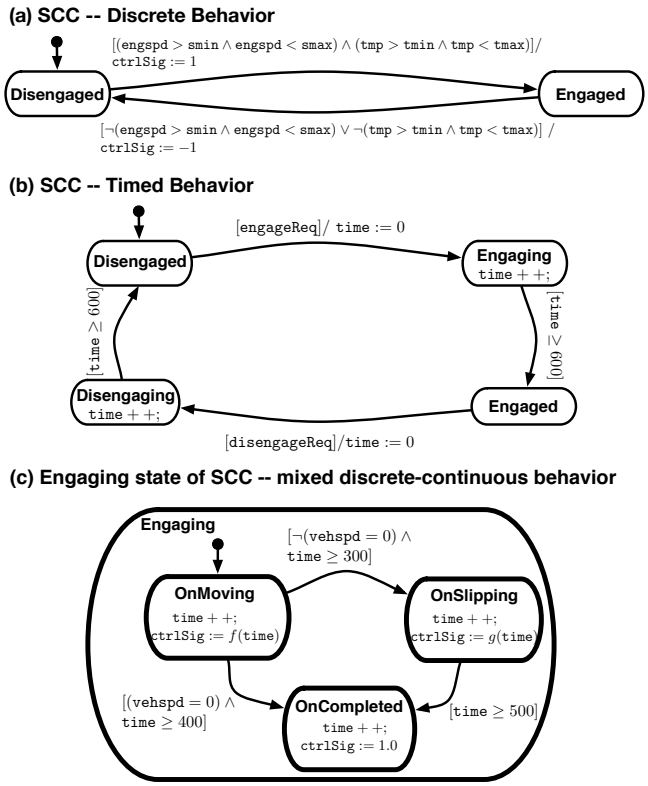


Figure 5.1. Supercharge Clutch Controller (SCC) Stateflow.

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 5.2 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 5.1(c)). Our focus in this chapter 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 5.1(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

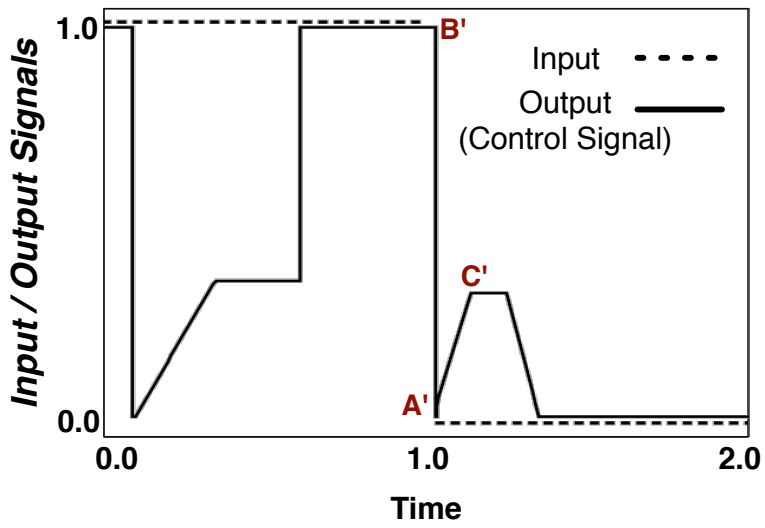


Figure 5.2. An example input (dashed line) and output (solid line) signals for SCC in Figure 5.1. The input signal represents `engageReq`, and the output signal represents `ctrlSig`.

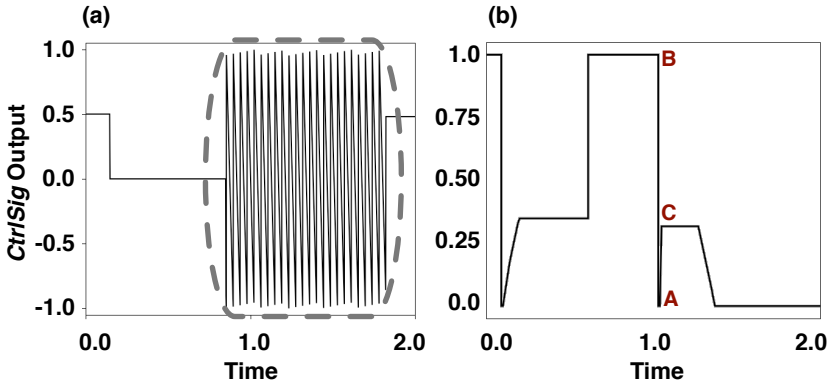


Figure 5.3. The output signals (`ctrlSig`) for two faulty versions of SCC: (a) an unstable output, and (b) a discontinuous output.

and rely on the `time` variable (see Figure 5.1(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 [Nise, 2004] 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 [Clarke et al., 1999] largely focuses on properties that fall in groups one and two above [Nardi, 2014]. 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.

Failure patterns. Figure 5.3 shows two specific patterns of failures in continuous output signals, violating requirements on desired physical behaviors of controllers (group three). The failure in Figure 5.3(a) shows *instability*, and the one in Figure 5.3(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 5.2, we provide a number of test generation algorithms to generate test cases that reveal failures in mixed discrete-continuous Stateflow outputs including the two failure patterns in Figure 5.3.

5.2 Test Generation Algorithms

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

Black-box input diversity (ID). Our input diversity generation algorithm is adaptive random and attempts to maximize diversity of test inputs selected from the input search space. Adaptive random test generation [Arcuri and Briand, 2011, Chen et al., 2010] is a simple strategy that is commonly used as a baseline for comparison. A generation 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 [Namin and Andrews, 2009, Inozemtseva and Holmes, 2014]. We consider the two well-known *state coverage (SC)* and *transition coverage (TC)* criteria for Stateflows [Binder, 2000] 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 [Alshahwan and Harman, 2012, Alshahwan and Harman, 2014]. Stateflow outputs provide a primary source of data for engineers to find faults. Hence, in our work, we adapt the output uniqueness proposed by [Alshahwan and Harman, 2012, Alshahwan and Harman, 2014] and define a notion of output diversity over continuous control signals.

Black-box failure-based. The goal of failure-based test generation algorithms is to select test inputs that are able to reveal common failures specific to a particular domain [Pretschner et al., 2013]. 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 generation 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 5.3(a). A period of insta-

bility 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 5.2 and 5.3(b), there is a discrete jump at around time 1.0 sec (i.e., point A$'$ in Figure 5.2, and point A in Figure 5.3(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 5.2 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 5.3(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 [Wikipedia., 2016]. Specifically, the failure shown in Figure 5.3(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 generation problem, and we then present our test generation algorithms.

Test generation 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 generation 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 [Satpathy et al., 2008]. 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[1]. 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, $\mathcal{R}_{sg}$

[1] In case a variable-step solver is used to simulate *SF*, we choose the least common divisor (lcd) of simulation steps as Δt. Note that according to Nyquist-Shannon sampling theorem [Gold et al., 1969], with a Δt of $1ms$, continuous signals with a frequency up to 500 HZ can be discretized without any information loss. In this thesis, we mostly deal with input/output signals that are aperiodic, e.g., driver's commands, and do not have high frequencies. Studying scalability of our approach to systems involving signals with very large frequencies is left for future work.

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 5.1, to ensure the feasibility of the generated input signals, in this chapter, 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 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*.

5.2.1 Input Diversity Test Generation

The input diversity generation 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:

$$\hat{dist}(sg_j, sg'_j) = \frac{\sqrt{\sum_{i=0}^{k} (sg_j(i \cdot \Delta t) - sg'_j(i \cdot \Delta t))^2}}{\sqrt{k+1} \times (max_{\mathcal{R}_{sg}} - min_{\mathcal{R}_{sg}})} \tag{5.1}$$

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 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:

$$dist(I, I') = \sum_{j=1}^{d} \hat{dist}(sg_j, sg'_j) \tag{5.2}$$

Figure 5.4 shows the ID-Generation 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 $7-9$).

5.2.2 Coverage-based Test Generation

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.

Algorithm. ID-Generation

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

Figure 5.4. The input diversity test generation algorithm (ID).

Algorithm. SC-Generation

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

Figure 5.5. The state coverage (SC) generation algorithm. The algorithm for transition coverage, TC, is obtained by replacing S (state coverage report) with T (transition coverage report).

The state coverage test generation algorithm, SC-Generation, is shown in Figure 5.5. The algorithm for transition coverage, TC-Generation, 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 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).

Algorithm. OD-Generation

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

Figure 5.6. The output diversity test generation algorithm (OD).

5.2.3 Output Diversity Test Generation

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 [Alshahwan and Harman, 2014]. 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 5.1 (i.e., $dist_o(sg_o, sg'_o) = \hat{dist}(sg_o, sg'_o)$).

Figure 5.6 shows the OD algorithm, i.e., OD-Generation. 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).

5.2.4 Failure-based Test Generation

The goal of failure-based test generation 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 [Luke, 2013] that generate test inputs maximizing the likelihood of presence of failures in outputs.

We propose two failure-based test generation algorithms, output stability and output continuity that respectively correspond to instability and discontinuity failure patterns introduced in Section 5.1. 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 generation algorithms that guide the search to identify test inputs that maximize these objective functions, and hence, are more

likely to reveal faults.

Output stability.[2] 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 5.3(a) is higher than that of the *stability* function applied to the signal in Figure 5.3(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. An alternative way to define the *stability* function is to count the number of ups and downs in the signal. That is, to count the number of times the signal changes from being ascending to descending, and vice versa. Such a function, however, does not take the magnitude of oscillations into account. In our work, we opt for the above definition of the *stability* function which incorporates the magnitude of oscillations in the definition of the *stability* function.

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 5.3(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 5.2 is lower than that in Figure 5.3(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)* algo-

[2]We note that the stability function defined here is different from the stability objective function defined for testing closed-loop controllers in Chapters 3 and 4.

Algorithm. OS-Generation

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. $All = All \cup \{newI\}$
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. $All = All \cup \{I\}$
15. Let TS be the test inputs in All with the q-highest values of $stability$ function
16. return TS

Figure 5.7. The test generation algorithm based on output stability. The algorithm for output continuity, OC-Generation, is obtained by replacing $stability(sg_o)$ with $continuity(sg_o)$.

rithm [Luke, 2013]. In our earlier work on computing test cases violating stability, smoothness, and responsiveness requirements for closed-loop controllers [Matinnejad et al., 2015a], HCRR performed best among a number of alternative single-state search heuristics. Figure 5.7 shows our output stability test generation algorithm, OS-Generation, based on HCRR. The algorithm for output continuity, OC-Generation, is obtained by replacing $stability(sg_o)$ with $continuity(sg_o)$ in OS-Generation. 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 generation algorithms in Figures 5.4 to 5.6. This is to ensure that OC-Generation spends the same test execution budget as the other generation algorithms. The OC-Generation 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.

5.3 Experiment Setup

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

5.3.1 Research Questions

RQ1 (Fault Revealing Ability). *How does the fault revealing ability of our proposed test generation algorithms compare with one another?* We start by comparing the ability of the test suites generated using the different test generation algorithms discussed in Section 5.2 in revealing faults in Stateflow models. In particular, we are interested to know (1) if our generation algorithms outperform input diversity (baseline)? and (2) if there is any generation algorithm that consistently reveals the most faults across different study subjects and different fault types?

RQ2 (Fault Revealing Subsumption) *Is any of our generation algorithms subsumed by other algorithms? or for each generation algorithm, are there some faults that can be found by that algorithm, but not by others?* This question investigates if any of the generation algorithms discussed in Section 5.2 is subsumed by other algorithms, i.e., if any generation 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 generation algorithms?* This question investigates whether any of our generation algorithms has a tendency to reveal a certain type of failures better than others. This shows whether our generation 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 generation 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 generation algorithms already perform well with small test suite sizes, while some may require to enlarge test suites to better reveal faults.

5.3.2 Study Subjects

We use three Stateflow models in our experiments: Two industrial models from Delphi, namely, SCC (discussed in Section 5.1) and Auto Start-Stop Control (ASS); and one public domain model from Mathworks website [The MathWorks Inc., 2016d], (i.e., Guidance Control System (GCS)). Table 5.1 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 5.1 are those obtained after model flattening.

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 [The MathWorks Inc., 2016i]. 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 [The MathWorks Inc., 2016i], 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

Table 5.1. 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

some configuration parameters and some difference equations in some states. We have made the modified version available at [Matinnejad, 2016].

5.3.3 Measuring Fault Revealing Ability

In our study, we measure the fault revealing ability of test suites generated by different generation algorithms. To automate our experiments, we use fault-free versions of our subject models to generate test oracles (i.e, the ground truth oracle [Barr et al., 2015]). Let *TS* be a test suite generated by one of our generation 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* 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 5.1, 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*.

5.3.4 Experiment Design

Figure 5.8 shows the overall structure of our experiments consisting of the following two steps:

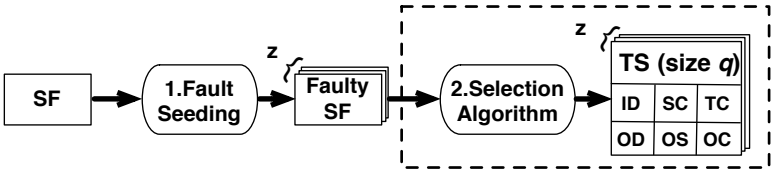


Figure 5.8. Our experiment design: Step 2 was repeated for 100 times due to the randomness in our generation algorithms.

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 Generation. As shown in Figure 5.8, after seeding faults, for each faulty model, we ran our six generation algorithms, namely Input Diversity (ID), State Coverage (SC), Transition Coverage (TC), Output Diversity (OD), Output Stability (OS), and Output Continuity (OC) test generation algorithms. For each faulty model and each generation 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 generation step of our algorithm for 100 times to account for the randomness in the generation 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 generation 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.

5.4 Results and Discussions

This section provides responses, based on our experiment design, for research questions **RQ1** to **RQ4** described in Section 5.3.

RQ1 (Fault Revealing Ability). To answer **RQ1**, we ran the experiment in Figure 5.8 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 5.9(a) shows four plots comparing the fault revealing ability of the test generation algorithms discussed in Section 5.2 with *THR*=0.2. Each plot in Figure 5.9(a)

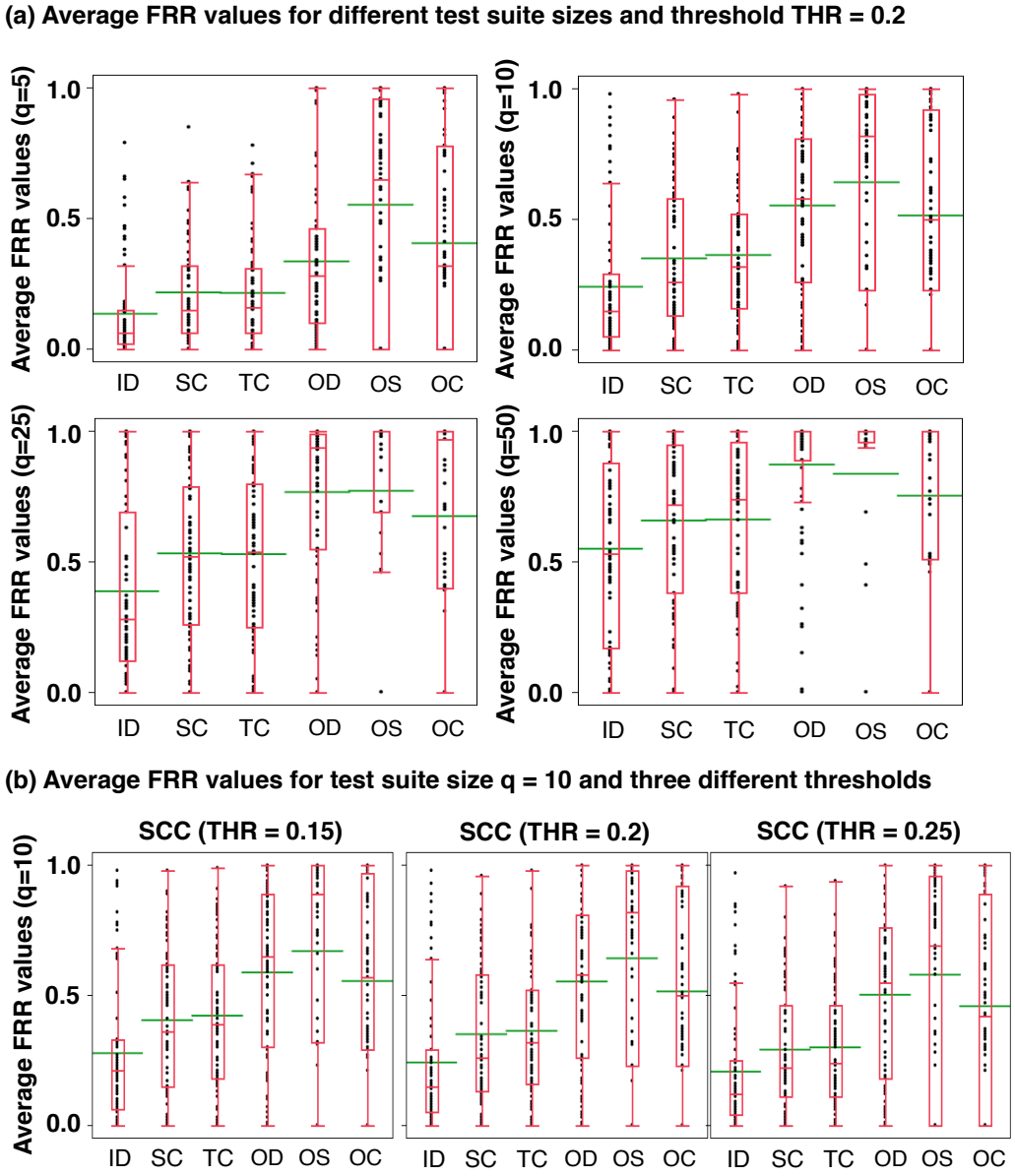


Figure 5.9. Boxplots comparing fault revealing abilities of our test generation algorithms for different test suite sizes and different thresholds.

compares six distributions corresponding to our six test generation 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 generation algorithms to that faulty model. For example, a point with (x = SC) and (y = 0.32) in the (q = 5) plot of Figure 5.9(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 generation algorithms, we performed the non-parametric pairwise Wilcoxon Pairs Signed Ranks test [Capon, 1991], and calculated the effect size using Cohen's d [Cohen, 1977]. 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 $d \geq 0.8$ [Cohen, 1977].

Comparison with Input Diversity. Testing differences in *FRR* distributions with *THR*=0.2 shows that, for all the test suite sizes, all the test generation 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 generation 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 generation 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 5.9(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 generation 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

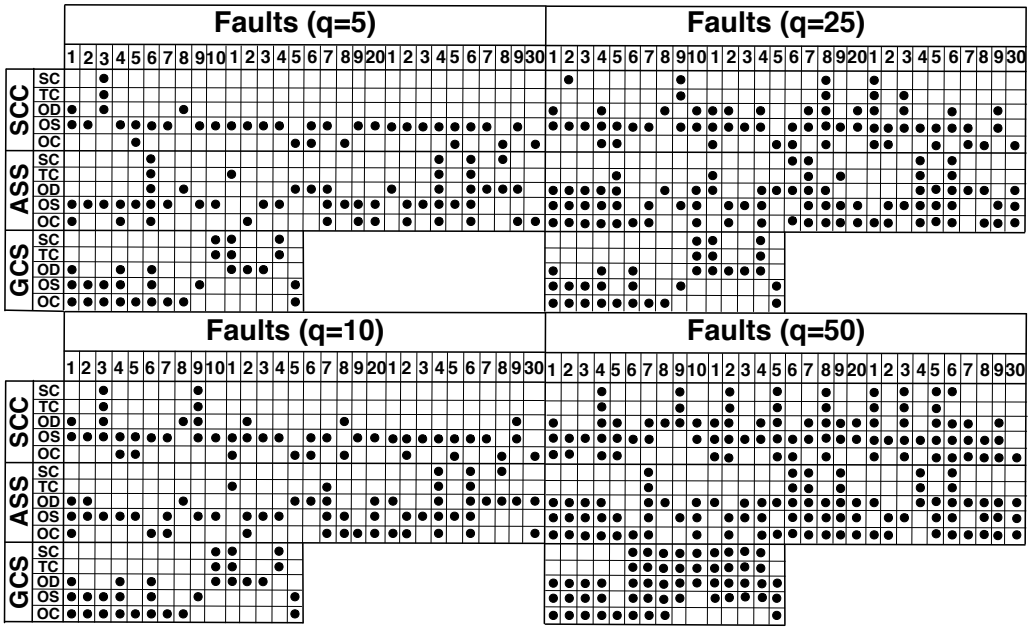


Figure 5.10. The best generation algorithm(s) for each of the 75 faulty models.

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 5.9(a). We applied the Wilcoxon test to identify, for each of the 75 faulty models, which generation algorithm yielded the highest fault revealing rate (i.e., the highest average *FRR* over 100 runs). Figure 5.10 and Table 5.2 show the results. Figure 5.10 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 5.2 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.

Coverage algorithms. As shown in Table 5.2, 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 5.10, SC and TC together are able to find three faults exclusively for $q = 5$ (11 of ASS, and 10 and 14 of

Table 5.2. 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

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 5.2, 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 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 5.11(a) to (c) compare the fault revealing ability of our test generation 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 5.11(c), for q = 50, the *FRR* value distribution for OD has the highest average compared to other algorithms. Further, the variance of

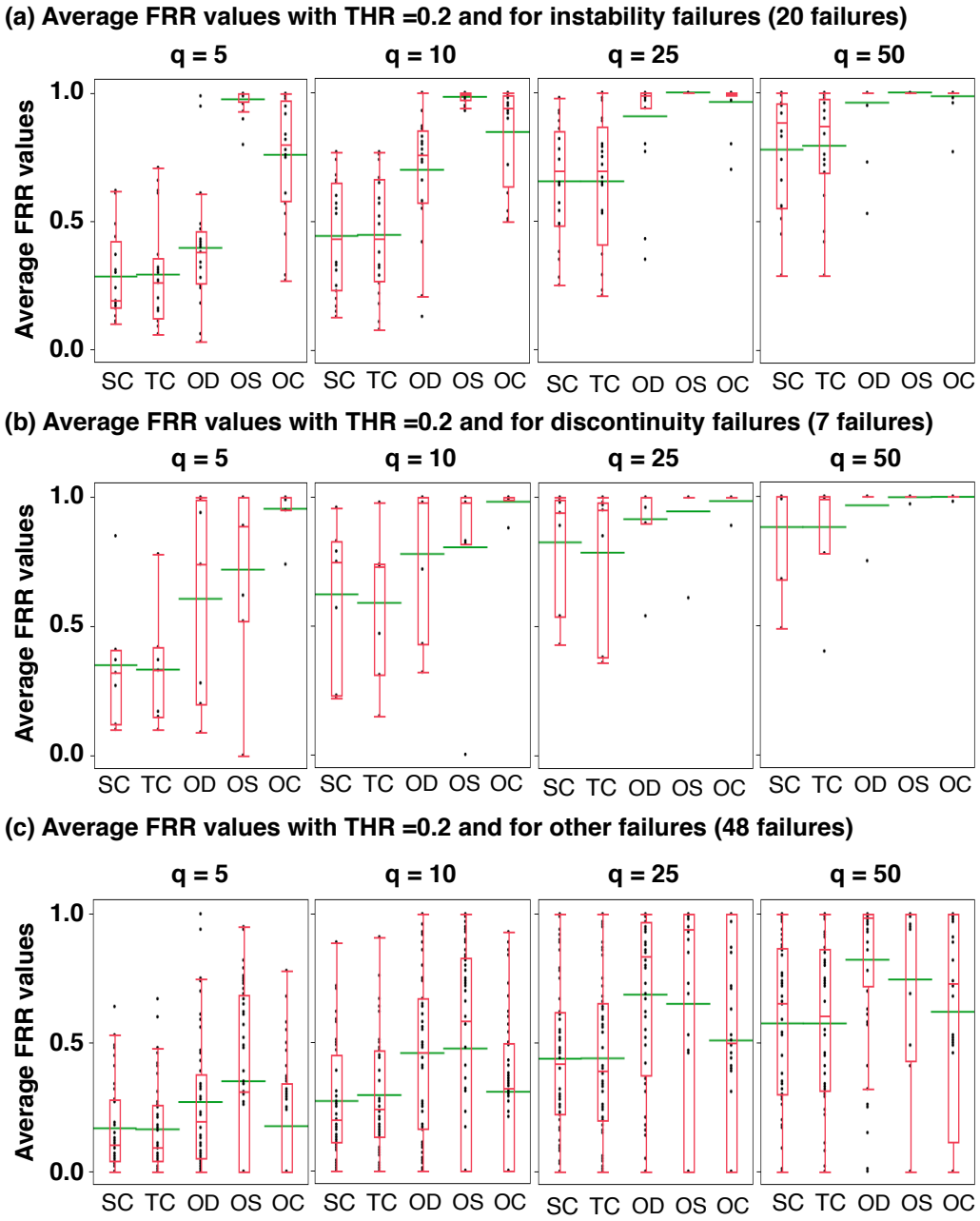


Figure 5.11. The average *FRR* values for different types of failures and for different test suite sizes.

FRR values for OD in Figure 5.11(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

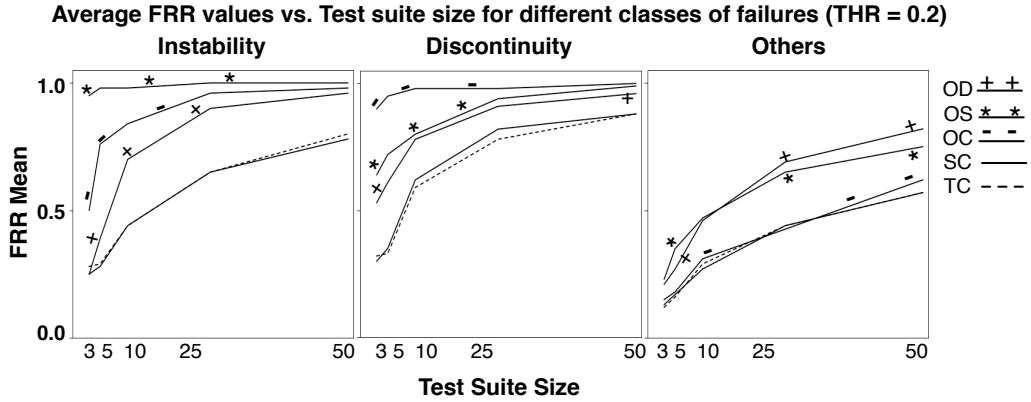


Figure 5.12. The impact of test suite size on the average *FRR* over 100 test suites of different faulty models.

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 5.9 to include $q = 3$, as well. Figure 5.12 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 5.12, 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 5.12, 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 generation 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 5.3(b)).

Combining output-based generation 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 generation algorithms.

5.5 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 generation algorithms are required. These algorithms aim at providing minimal test suites with high fault revealing power. We proposed and evaluated six test generation 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 generation 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.

Chapter 6

Test Generation and Test Prioritization for Simulink Models with Dynamic Behavior

In this chapter we focus on providing effective testing techniques to help engineers ensure correctness of Simulink/Stateflow models in their entirety. Recall from Section 2.1, that Simulink models are developed either for simulation or for code generation purposes. We propose a test generation algorithm and a test prioritization algorithm that are applicable to both continuous simulation as well as discrete code generation Simulink models. First, we discuss the key challenges of the existing techniques for testing simulation and code generation Simulink models and then discuss how our approach addresses these challenges.

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 that we discuss 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 [Zander et al., 2012, Pretschner et al., 2007a, Pretschner et al., 2007b], and are not compatible, and hence not applicable, to Simulink models with continuous behaviors (i.e., simulation models) [Zander et al., 2012, Pretschner et al., 2007a, Pretschner et al., 2007b]. 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 [Barnat et al., 2006], KLEE [Cadar et al., 2008] and JavaPathFinder [JPF, 2016]) or by SAT/Constraint/SMT solvers (e.g., PVS [Owre et al., 1996], Prover [Prover Technology, 2016])). The incompatibility challenge even extends to some features that are commonly used in the Simulink code generation models [Rao et al., 2011, Zander et al., 2012]. More specifically, existing techniques have difficulties to handle library code or system functions (implemented as Matlab S-functions) and floating point and non-linear arithmetic operators (e.g. square root or trigonometry functions). In particular, Simulink Design Verifier (SLDV) [The MathWorks Inc., 2016g], 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 [Lakhotia et al., 2010], techniques that rely on these solvers to verify or test Simulink [The MathWorks Inc., 2016g, Hamon, 2008, Balasubramanian et al., 2011, Holling et al., 2014, Cleaveland et al., 2008] often fall short when the underlying model contains floating point and non-linear math

operators.

The Oracle Challenge. The second challenge mostly has to do with unrealistic assumptions about test oracles for Simulink models in practical settings. Several existing techniques rely on *automatable test oracles* such as assertions (explicit oracles [Barr et al., 2015, McMinn et al., 2010]) or run-time errors (implicit oracles [Barr et al., 2015, McMinn et al., 2010]) to identify faults in Simulink models [Nardi et al., 2013, Nardi, 2014]. However, formal specifications from which assertions can be derived, are expensive and may not be available in practice. Run-time errors such as integer over-/underflows are not sufficient as many faults may not lead to run-time crashes. Furthermore, several important CPS requirements concern continuous dynamics aspects of systems under analysis [Heimdahl et al., 2013, Pretschner et al., 2007a, Pretschner et al., 2007b]. 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. There is little work on verifying or testing Simulink models against CPS continuous dynamics requirements, or on developing automated test oracles for such requirements [Heimdahl et al., 2013, Pretschner et al., 2007a, Pretschner et al., 2007b]. Finally, in practice, it is likely and common that engineers assess system outputs manually to identify failures (i.e., test oracles are manual). In this situation, existing tools focus on generating test suites that fulfill some notion of structural coverage [The MathWorks Inc., 2016j, The Reactive Systems Inc., 2016]. Several recent studies, however, demonstrate that structural coverage criteria alone may not be effective in finding faults in software code [Inozemtseva and Holmes, 2014, Staats et al., 2012, Namin and Andrews, 2009]. Therefore, effectiveness of these criteria for Simulink model testing has yet to be ascertained and empirically evaluated.

The Scalability Challenge. There are almost no study that focuses on demonstrating 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 that is already recognized by Mathworks [Hamon, 2008]. Further, as models grow larger and become more complicated, it is more likely that they contain features that are 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 test oracle assumptions (the oracle challenge). Hence, scalability remains to be an open problem for Simulink testing and verification.

In this chapter, we provide automated techniques to generate effective test suites for Simulink/Stateflow models in their entirety. Our goal is to alleviate the above three challenges:

First, in order to deal with the incompatibility challenge, we focus on Simulink models consisting of both continuous and discrete behaviors, and generate test inputs as *signals*, i.e., functions over time. Our test generation algorithm is black-box and builds on a combination of a single-state search optimizer [Luke, 2013] and the whole test suite generation approach [Fraser and Arcuri, 2013, Fraser and Arcuri, 2011].

Second, in our work, we assume that test oracles are manual. Instead of focusing on structural coverage alone as done in existing tools, we propose and evaluate a test generation approach that aims to maximize diversity in output signals of Simulink models. Output signals provide a useful source of information for detecting faults in Simulink models as they not only show the values of variables at discrete time instants, but they also capture continuous aspects, i.e., they show how variables change

over a time interval. By inspecting output signals, one can determine whether the model output reaches appropriate values at the right times, whether the time period that the model takes to change its values is within acceptable limits, and whether the amount and the frequency of changes of variables over time are acceptable and do not violate continuous dynamics requirements. Our intuition is that test cases that yield diverse output signals may likely reveal different types of faults in Simulink models. 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.

In our work, in order to lower the cost of manual test oracle, we propose a test prioritization algorithm to automatically rank test cases generated by our test generation algorithm. Engineers can then inspect the test outputs starting from the top ranked test cases in the ranking. We take a greedy test prioritization approach that uses the dynamic test execution information to prioritize the tests. Specifically, we take into account the output diversity of the generated test suites, in addition to the amount of structural coverage achieved by individual test cases, to predict how likely it is for each test case to reveal a fault. This enables test cases with slightly lower coverage but coming from test suites with higher output diversity to rank higher, and may likely increase the likelihood of detecting faults early in the execution of the tests. Note that this is not the case with a test prioritization technique solely driven by structural coverage. Further, in our algorithm the notion of dynamic test coverage is specific to a Simulink model output. That is, for a given model output and a given test case, the test coverage includes Simulink blocks or Stateflow states that are executed by the test case and, in addition, from which the output is reachable.

Third, we evaluate our test generation and our test prioritization algorithms using four industrial and public-domain Simulink models. Our evaluation assesses fault revealing ability of our test generation and effectiveness of our proposed test prioritization. We systematically compare our algorithms with baseline, state-of-the-art and commercial approaches. In particular, we compare our test generation approach with a random test generation baseline, with a coverage-based test generation algorithm and with the Simulink Design Verifier (SLDV), a commercial tool for testing Simulink models. We compare our test prioritization algorithm with a random test prioritization baseline and a state-of-the-art prioritization technique proposed based on structural coverage [Zhang et al., 2013].

Contributions. Our contributions are as follows:

(1) We propose a Simulink testing approach consisting of a test generation algorithm and a test prioritization algorithm for Simulink models. Our approach is applicable to Simulink models developed for both purposes of simulation and code generation. Our approach does not rely on automatable test oracles and is guided by heuristics that build on continuous dynamic aspects of Simulink outputs.

(2) We propose a new notion of diversity for output signals and develop a novel algorithm based on this notion to generate test suites for Simulink models. We show that our test generation approach based on output diversity outperforms random baseline testing and coverage-based testing.

(3) 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 [Yoo and Harman, 2012, Zhang et al., 2013]. We show that our test prioritization algorithm outperforms random test prioritization and a state-of-the-art

coverage-based test prioritization [Zhang et al., 2013].

(4) We compare our approach with the Simulink Design Verifier (SLDV), the only testing toolbox of Simulink. In contrast to our approach, SLDV supports a subset of Simulink models. We show that, when considering the SLDV-compatible subset, our output diversity approach is able to reveal significantly more faults compared to SLDV, and further, it subsumes SLDV in revealing faults: In our experiments, any fault identified by SLDV is also identified by our approach.

Organization. This chapter is structured as follows. Section 6.1 presents examples of simulation and code generation models and motivates our output diversity approach in comparison with coverage-based test generation. Section 6.2 provides background on test input generation and fixes our formal notation. Sections 6.3 and 6.4 describe our output diversity test suite generation and our test case prioritization algorithms, respectively. Section 6.5 explains how we estimate the test oracle cost in our approach. Our test generation and prioritization tool, called SimCoTest, is presented in Section 6.6. Sections 6.7 and 6.8 describe our experiments setup and experiments results, respectively. Finally, Section 6.9 concludes the chapter.

6.1 Motivation

In this section, we provide examples of simulation and code generation models. We then introduce SimuLink Design Verifier (SLDV) as an state-of-the-art commercial tool for testing Simulink models. We then provide some intuition for our output diversity test generation approach by contrasting it with the test generation approach of SLDV using an illustrative example.

6.1.1 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 [The MathWorks Inc., 2016e].

Simulation models. Figure 6.1(a) shows a very simplified simulation model for FLC adopted from [Zander et al., 2012] 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 6.1(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 6.1(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 6.1(a) is executable. Engineers can run the model for any desired input signal and inspect the output. Exam-

ples of input and output signals for this model are shown in Figures 6.1(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.

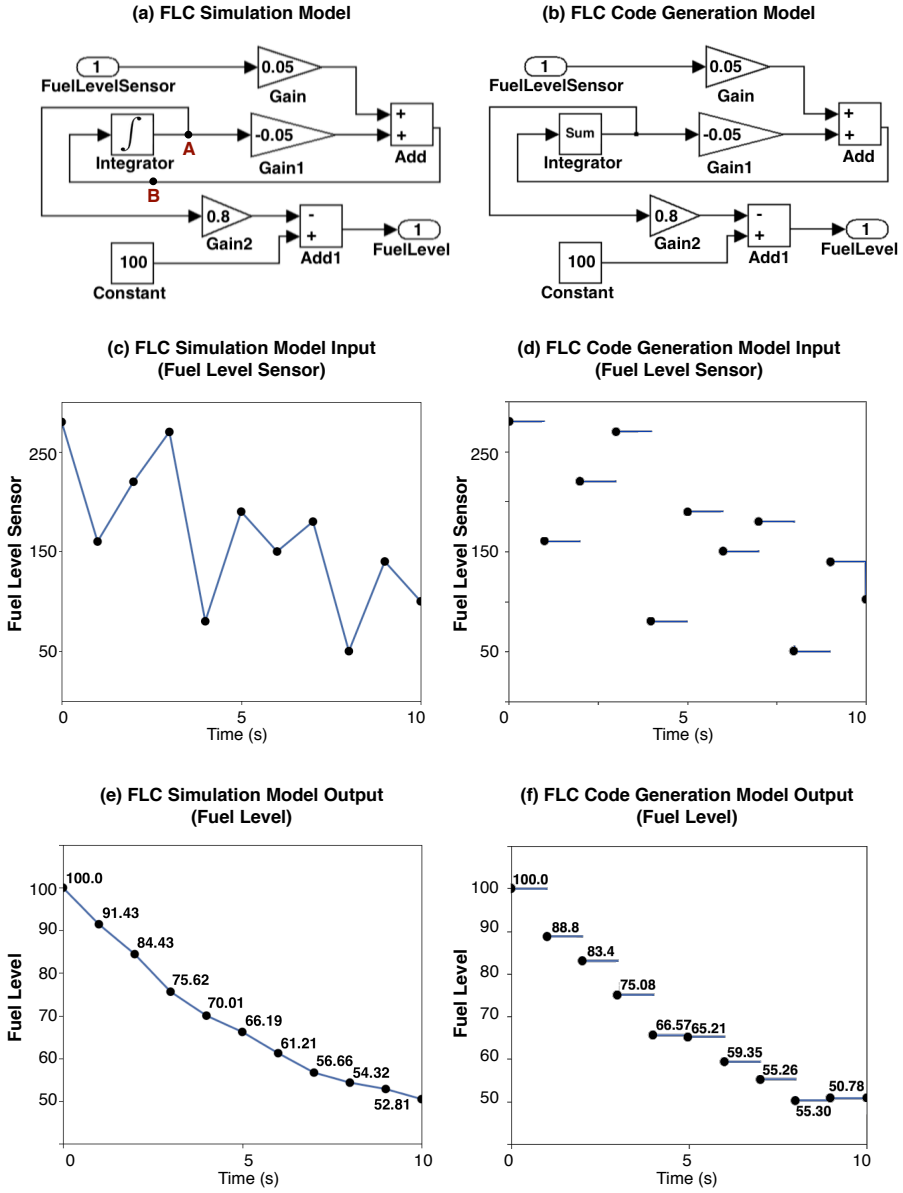


Figure 6.1. A Fuel Level Controller (FLC) example: (a) A simulation model of FLC; (b) a code generation model of FLC; (c) an input to FLC simulation model; (d) an input to FLC code generation model; (e) output of (a) when given (c) as input; (f) output of (b) when given (d) as input.

Code generation models. Figure 6.1(b) shows an example FLC code generation model, (i.e., the model from which software code can be automatically generated). The code generation model is time-discrete and magnitude-discrete. Further, note that the continuous integrator block ($\int$) in the

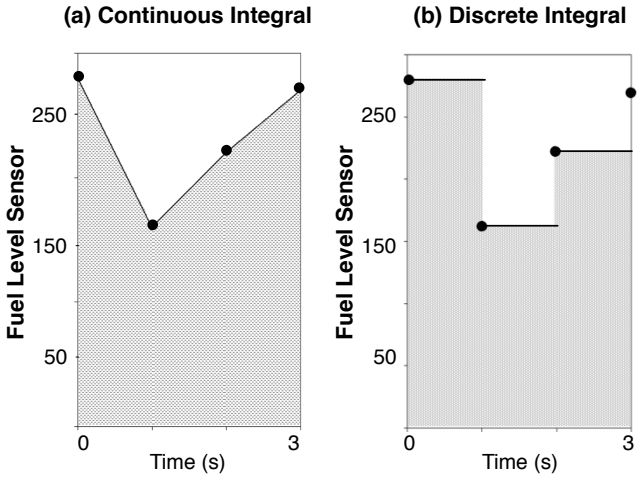


Figure 6.2. Comparing outputs of (a) continuous integral $\int$ and (b) discrete integral `sum` from models in Figures 6.1 (a) and (b), respectively.

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 6.1(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 6.1(c) shows a continuous input signal for the simulation model in Figure 6.1(a) over a 10 sec time period. Figure 6.1(d) shows the discrete version of the signal in Figure 6.1(c) that is used as input for the code generation model in Figure 6.1(b). Models in Figures 6.1(a) and (b) produce the outputs in Figures 6.1(e) and (f) once they are provided with the inputs in Figures 6.1(c) and (d), respectively. As shown in the figures, the percentages of fuel level in the continuous output signal (Figure 6.1(e)) differ from those in the discrete output signal (Figure 6.1(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 6.2. The grey area in Figure 6.2(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 6.2(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 chapter, we provide a testing technique that is applicable to both simulation and code generation models.

6.1.2 Limitations of Existing Simulink Testing Tools

In this chapter, we systematically compare our Simulink testing approach with existing alternative tools and techniques. In particular, we compare our work with SimuLink Design Verifier (SLDV) that is distributed by Mathworks and is among the most well-known commercial tools for testing and verification of Simulink models in industry. In this section, we introduce SLDV and contrast its underlying approach with our work. In Sections 6.7 and 6.8, we will provide an empirical study comparing our technique with SLDV as well as other alternative baseline approaches (i.e., random and coverage-based testing).

SimuLink Design Verifier (SLDV) is a product of Mathworks and a Simulink toolbox. It is the only Simulink toolbox that is dedicated to test generation. It automatically generates test input signals for Simulink models using constraint solving and model-checking techniques [The MathWorks Inc., 2016g]. SLDV provides two usage modes corresponding to two different assumptions about test oracles: (1) The first usage mode assumes that automatable test oracles are not available. SLDV then generates test suites to achieve some form of structural coverage (i.e., Decision, Condition, and MC/DC) [The MathWorks Inc., 2016j]. (2) The second usage mode assumes that automatable test oracles are available either as assertions or run-time errors. SLDV then generates test scenarios (counter-examples) violating some given assertions or leading to some run-time error. In the first usage mode, SLDV creates a test suite satisfying a given structural coverage criterion [The MathWorks Inc., 2016j]. In the second usage mode, SLDV tries to prove that assertions/run-time crashes cannot be reached, or otherwise, it generates inputs triggering the assertions/errors.

As discussed in the beginning of this chapter, in our work we assume that automatable test oracles are not available. Hence, we systematically compare the fault revealing ability of our test generation algorithm with that of the first usage mode of SLDV, i.e., test generation guided by structural coverage.

We note that another well-known commercial tool for Simulink testing called Reactis [Reactive Systems Inc., 2016b] has several commonalities with SLDV. In particular, both tools rely on formal/exhaustive verification techniques to generate test cases [Cleaveland et al., 2008]. Further, they both rely on structural coverage to generate test cases when automatable test oracles are absent. In this paper, we chose to compare our approach with SLDV as it is available as a Simulink toolbox, but Reactis is a standalone tool independent from Mathworks products. Large experiments with SLDV can be fully automated using MATLAB APIs, while Reactis lacks such APIs, hence large experiments cannot be properly automated.

Structural Coverage versus Output Diversity. Main limitations of SLDV fall under the three high-level categories of challenges discussed in the beginning of this chapter. Specifically, SLDV supports a subset of the Simulink language (i.e., discrete fragment of Simulink) [Chaturvedi, 2009], and is not applicable to continuous blocks of Simulink such as the continuous integrator in Figure 6.1(a). Further, it is not applicable to Simulink models containing floating point and non-linear arithmetic operations, and supports only some restricted forms of S-Functions. These issues make SLDV inapplicable to many industry and real-world models.

In this chapter, in order to compare SLDV with our approach, we focus on the Simulink subset that is supported by SLDV. Considering this subset, the main difference between our approach and that used by SLDV then lies in their underlying test generation algorithms. While SLDV aims to

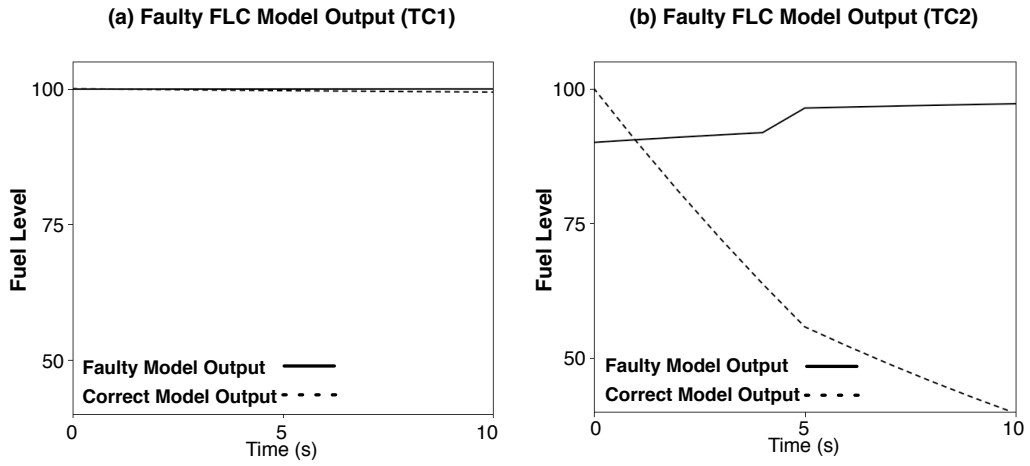


Figure 6.3. (a) A test output of a faulty version of model in Figure 6.1(a); and (b) another test output of the same faulty version of model in Figure 6.1(a).

maximize structural coverage, we focus on diversifying output signals. Here, we use an example to illustrate the workings of these two different test generation strategies. Consider a faulty version of the simulation model in Figure 6.1(a) where the line connected to point A is mistakenly connected to point B. Suppose we use SLDV to generate a test case (TC1) for this faulty model. Since the model in Figure 6.1(a) does not have any control behavior, a single test case can achieve full structural coverage. Figure 6.3(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. Given that in this domain small deviations from oracle are acceptable, 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 algorithm, the test suite size is not determined by structural coverage and is set independently. Suppose we choose to generate three test cases for the given faulty model. Figure 6.3(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. But TC2 is more fault revealing than TC1. Our approach attempts to generate test cases that yield diverse output signals to increase the probability of generating outputs that noticeably diverge from the expected result.

6.2 Background and Notation

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

6.2.1 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, $\mathcal{N} = \{n_1, \ldots, n_b\}$ 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.

As discussed in Section 6.1, in order to execute (simulate) a model M, a time step (Δt) should be given. Let k be the number of time steps in the simulation time interval $[0..T]$. To specify a signal sg, one has to provide the signal values at time points 0, Δt, $2 \cdot \Delta t$,..., $k \cdot \Delta t$. We denote these values by sg^0, sg^1, sg^2, ..., sg^k, respectively. As for input/output signals of a simulation model M, $sg(t)$ takes all the real values between sg^i and sg^{i+1} for $i \cdot \Delta t < t < (i+1) \cdot \Delta t$. In contrast, for input/output signals of a code generation model M, $sg(t)$ is equal to sg^i for $i \cdot \Delta t \leq t < (i+1) \cdot \Delta t$.

For the example in Figure 6.1, we have $T = 10s$, $\Delta t = 1s$, and $k = 10$. Note that in that example, to better illustrate the input and output signals, Δt is chosen to be larger than normal (e.g., in our experiment, we set $\Delta t = 0.001s$, $T = 2s$, and $k = 2000$). The signal in Figure 6.1(e) represents a signal for a simulation model, and the signal in Figure 6.1(f) represents a signal for a code generation model. The main difference between signals for simulation and code generation models is that signals for simulation models are continuous, i.e., their values are taken from a real interval (an infinite set), while signals for code generation models are discrete, i.e., their values are taken from a finite set of numbers.

6.2.2 Test Inputs

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.

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 6.2.1, each signal sg_{i_j} is characterized by a set of values $sg_{i_j}^0$, $sg_{i_j}^1$, $sg_{i_j}^2$, ..., $sg_{i_j}^k$ specifying

the values of signal sg_{i_j} at time steps 0, Δt, $2 \cdot \Delta t$, ..., $k \cdot \Delta t$, respectively. To generate an input signal sg_{i_j}, it is sufficient to generate values $sg^0_{i_j}$, $sg^1_{i_j}$, $sg^2_{i_j}$, ..., $sg^k_{i_j}$. The signal is then determined based on definitions in Section 6.2.1 depending on whether the signal is used for simulation models or for code generation models, respectively.

6.3 Test Generation Algorithms

We propose a search-based whole test suite generation algorithm 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.

6.3.1 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 [Matinnejad et al., 2015a], 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.

$$\hat{dist}(sg_o, sg'_o) = \frac{\sqrt{\sum_{i=0}^{k} (sg_o(i \cdot \Delta t) - sg'_o(i \cdot \Delta t))^2}}{\sqrt{k+1} \times (max_{\mathcal{R}} - min_{\mathcal{R}})} \tag{6.1}$$

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 $\hat{dist}(sg_o, sg'_o)$ 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, $\hat{dist}(sg_o, sg'_o)$ is relatively large when sg_o is constant at the maximum of the signal range while sg'_o is constant at the minimum of the signal range. A test suite that generates several output signals with similar shapes may not help with fault finding.

6.3.2 Feature-based Output Diversity

In machine learning, a feature is an individual measurable and non-redundant property of a phenomenon being observed [Witten et al., 2011]. 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*

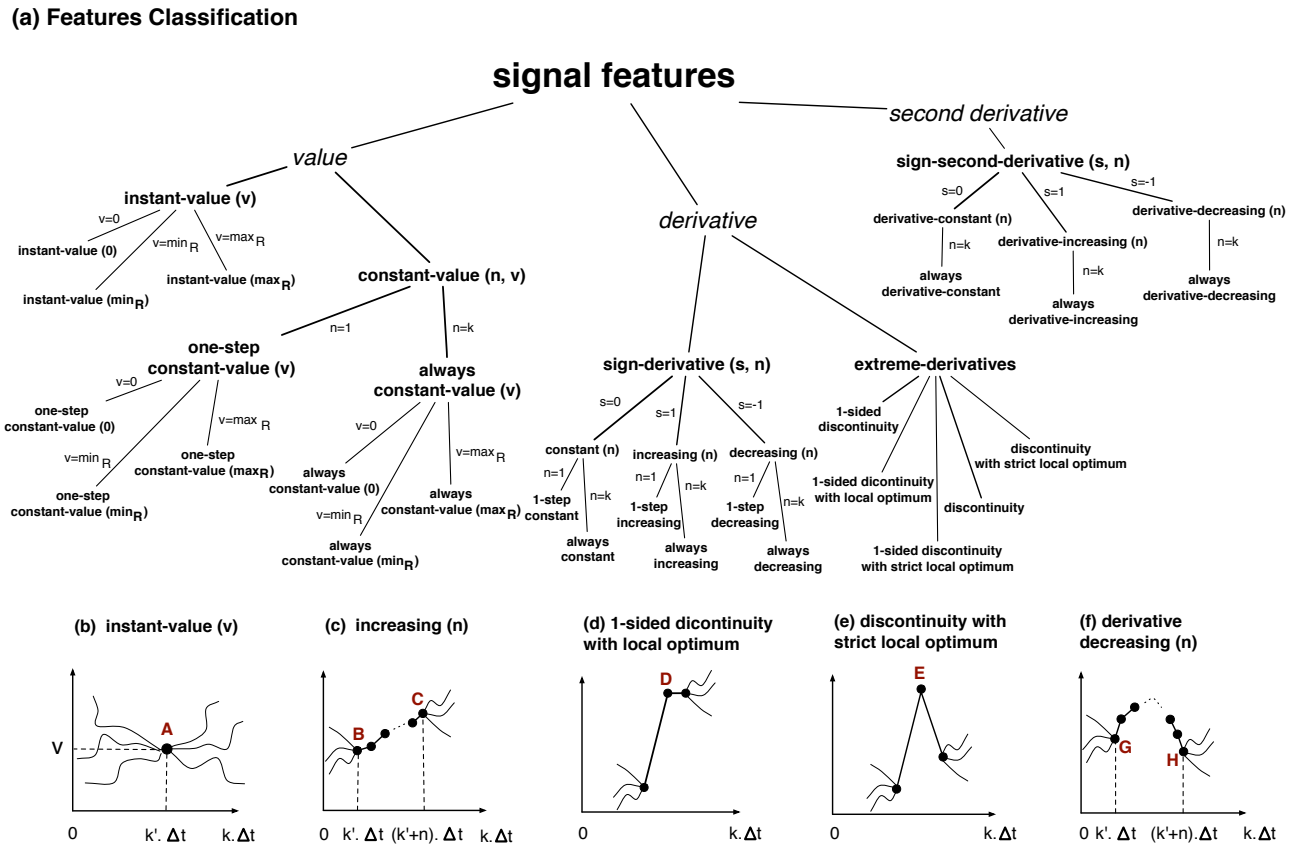


Figure 6.4. Signal Features: (a) Our signal feature classification, and (b)–(f) Examples of signal features from the classification in (a).

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 6.4(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 [Porat, 1997]. The classification in Figure 6.4(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 6.4(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 6.4(b) takes value v at point A. The signal in Figure 6.4(c) is increasing for n steps from B to C. The signal in Figure 6.4(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 6.4(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 6.4(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 6.4(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 6.4(b) and (c). Specifically, the feature function F_{f_b} related to the signal feature "instant-value(v)" in Figure 6.4(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 6.4(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 Fig-

ure 6.4(c) is defined as follows:

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

where $\texttt{lds}(sg,i)$ denotes the sign of the left derivative of sg at step i. Specifically, $\texttt{lds}(sg,i)$ 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 6.4 are available at [Matinnejad, Reza, 2016].

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 [Witten et al., 2011]. 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.

6.3.3 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 6.2.2, 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 6.4) 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:

$$O_v(TSO) = \frac{\sum_{i=1}^{q} \text{MIN}_{\forall sg \in TSO\setminus\{sg_i\}} \hat{dist}(sg_i,sg)}{q} \tag{6.2}$$

$$O_f(TSO) = \frac{\sum_{i=1}^{q} \text{MIN}_{\forall sg \in TSO\setminus\{sg_i\}} \hat{dist}(F^v(sg_i),F^v(sg))}{q} \tag{6.3}$$

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 [Fraser and Arcuri, 2013] 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 [Luke, 2013]. This is because our objective functions are computationally expensive as they require to simulate the underlying Simulink model and compute distance functions between every test input pair. When objective functions are time-consuming, population-based search may become less scalable as it may have to re-compute objective functions for several new or modified members of the population at each iteration.

Figure 6.5 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 [Luke, 2013]. Specifically, the algorithm generates an initial solution (lines 1-3), iteratively tweaks this solution (line 9), and selects a new solution whenever its objective function is higher than the current best solution (lines 13-15). 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. The tweak operator (line 10) is similar to the one used in (1+1) EA algorithm [Luke, 2013]. It receives as input the current test suite TS and a tweak parameter σ. For every test input $I \in TS$ such that $I = (sg_{i_0}, \ldots, sg_{i_n})$, the tweak operator shifts every signal sg_{i_j} by adding values x_{j_l}'s to signal values $sg_{i_j}(l \cdot \Delta t)$ such that $0 \leq l \leq k$ where k is the number of simulation steps. The x_{j_l} values are 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 sg_{i_j}.

Like the simulated annealing search algorithm, our OD algorithm in Figure 6.5 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 similar to a random explorative search. As the time proceeds, the temperature is lowered, eventually to zero, turning the search into a exploitative search algorithm such as Hill Climbing [Luke, 2013]. 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.

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

1. $TS \leftarrow$ GenerateInitialTestSuite(q) /*Test suite size q */
2. $TSO \leftarrow$ signals obtained for output o by simulating M for every test input in TS
3. $BestFound \leftarrow O(TSO)$
4. *whole-test-suite-coverage* $\leftarrow$ coverage achieved by test cases in TS over M
5. *initial-coverage* $\leftarrow$ *whole-test-suite-coverage*
6. *accumulative-coverage* $\leftarrow$ *initial-coverage*
7. $\sigma \leftarrow \sigma$-*exploration* /*Tweak parameter $\sigma \in [\sigma$-*exploitation*$\ldots\sigma$-*exploration*$]$
8. **repeat**
9. *newTS* = Tweak(TS, σ) /* generating new candidate solution */
10. $TSO \leftarrow$ signals obtained for output o by simulating M for every test input in *newTS*
11. *whole-test-suite-coverage* $\leftarrow$ coverage achieved by test cases in *newTS* over M
12. *accumulative-coverage* $\leftarrow$ *accumulative-coverage* + *whole-test-suite-coverage*
13. **if** $O(TSO) > highestFound$ **:**
14. $highestFound = O(TSO)$
15. $TS = newTS$
16. Reduce σ proportionally from σ-*exploration* to σ-*exploitation* as *accumulative-coverage* increases over *initial-coverage*
17. **until** maximum resources spent
18. **return** TS

Figure 6.5. Our output diversity (OD) test generation algorithm for Simulink models.

The reason that we opt for such search solution is that, based on our existing experience of applying search algorithms to continuous controllers [Matinnejad et al., 2015b], 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 (i.e., $\sigma = 0.01$), our result will be biased by the initially randomly selected solution. To reduce this bias, we start by performing a more explorative search (i.e., $\sigma = 0.5$). However, if we let the search remain explorative, it may reduce to a random search. Hence, we reduce σ iteratively in OD such that the amount of reduction in σ is proportional to the increase in the accumulative structural coverage obtained by the generated test suites (line 16).

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 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 $O(q \times T_M) + O(q^2 \times k)$[1].

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 6.4 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 6.8, we will provide average time that it takes to perform model simulations (T_M) and to execute one iteration of the OD algorithm using O_v and O_f functions based on our empirical evaluation.

[1]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.

6.4 Test Prioritization Algorithm

Our OD test generation algorithm discussed in Section 6.3 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, dynamic 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 dynamic test coverages, are restricted to static analysis of code or other artifacts [Qi Luo and Poshyvanyk, 2016, Thomas et al., 2014]. 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 per 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 [Qi Luo and Poshyvanyk, 2016]. 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 [Li et al., 2007, Qi Luo and Poshyvanyk, 2016].

Our test case prioritization algorithm is shown in Figure 6.6. 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. 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 dynamic 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 $faultiness : \mathcal{N} \rightarrow [0..1]$. Below, we first discuss these three functions. We then describe how our proposed prioritization algorithm works.

Dynamic Test Coverage. Recall from Section 6.3 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). Dynamic test coverage refers to the set of Simulink nodes (i.e., Simuilnk 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 dynamic test coverage of tc by $covers(tc)$ and define it as follows:

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

where $o \in \mathcal{O}$ and $static\_slice(o)$ is referred to as the *static backward slice* of output o and is the set of

Algorithm. Test case prioritization algorithm

Input: – $M = (\mathcal{I}, \mathcal{N}, \mathcal{O})$: Simulink Model
– $\mathcal{TC}$: A test suite for M
– $covers : \mathcal{TC} \rightarrow 2^{\mathcal{N}}$: Dynamic test coverage of test cases
– $FRP : \mathcal{TC} \rightarrow [0..1]$: Fault revealing probability of test cases
– $faultiness : \mathcal{N} \rightarrow [0..1]$: 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 **tc** that yields the highest aggregated fault revealing probabilities*/ and add it to *Rank*/
6. Let **tc** $\in \mathcal{TC}$ yield the largest $P(tc)$
7. $Rank[Ranked] =$ **tc**
   /* Lines 8-10: Update the faultiness probability of Simulink nodes covered by **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. **returen** *Rank*

Figure 6.6. Our test prioritization algorithm that is used to rank test cases generated for all the Simulink outputs.

all nodes in $\mathcal{N}$ from which o is reachable.

Note that our notion of dynamic 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 dynamic test coverage is the same as the notion of test execution slices defined in our previous work on fault localization of Simulink models [Liu et al., 2015]. 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 implementations of these concepts for Simulink models in this paper.

Fault revealing probability. Given a test case *tc*, the fault revealing probability of *tc*, denoted by $FRP(tc)$, is the probability of *tc* being able to reveal faults successfully. The fault revealing probabilities are not modified during test case prioritization.

Faultiness probability. Given a Simulink node n, the faultiness probability of n, denoted by $faultiness(n)$, is the probability of n containing a fault that is not yet detected by the currently ranked test cases. The faultiness probability is the highest at the beginning when *Rank* is empty and decreases as the size of *Rank* increases.

Having discussed the three input functions, we now describe the algorithm in Figure 6.6. As discussed earlier, this algorithm 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.

We assume that faults are evenly distributed in the underlying Simulink models and assign $faultiness(n)$ to one for every $n \in \mathcal{N}$. We consider the notion of output diversity defined in Section 6.3 as a proxy for the fault revealing ability of test cases, i.e., *FRP*. In particular, suppose that $tc \in \mathcal{TC}$ belongs to a test suite *TS* generated by the algorithm in Figure 6.5. 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 6.2 and 6.3, respectively.

Our proposed test prioritization algorithm (Figure 6.6) generalizes and extends the existing dynamic test prioritization techniques [Zhang et al., 2013, Yoo and Harman, 2012, Qi Luo and Poshyvanyk, 2016]. 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 6.6 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 6.6 (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.

6.5 Test Oracle

As discussed in the beginning of this chapter and Section 6.1, in our work, we make two important assumptions about test oracles: First, we assume that no automatable test oracle is available, a common situation in practice. Second, 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 *continuous* period of time. These two assumptions have the following two implications on our approach that we discuss in this section.

First, since we assume that test oracles 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 those test suites 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 6.5) 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 6.2 that a test input signal is characterized by values sg^0, sg^1 ... sg^k. If $sg^0 = sg^1 = \ldots = sg^k$, the resulting test input is a constant signal. If $sg^0 = sg^1 = \ldots = sg^i$, $sg^{i+1} = \ldots = sg^k$ and $sg^i \neq sg^k$, the resulting test input is a step signal. If $sg^0 = sg^1 = \ldots = sg^i$, $sg^{i+1} = \ldots = sg^j$, $sg^{j+1} = \ldots = sg^k$ and sg^i, sg^j and sg^k are all different, the resulting test input is a two-step signal, and so on. In the automotive domain, constant signals are considered the least complex and the most common test inputs for Simulink models. It is relatively easy for engineers to inspect the output signals generated by constant input signals. Moving from constant input signals to one-step input signals and from one-step input signals to multiple-step input signals, the resulting output signals become more complex and the cost of manual test oracle typically increases. To ensure that test suites $TS = \{I_1, \ldots I_{q_1}\}$ and $TS' = \{I'_1, \ldots I'_{q_2}\}$ have the same input complexity, the input signals in TS and TS' should have the same number of steps. That is, for every test input $I_i = (sg_1, \ldots, sg_n)$ in TS (respectively TS'), there exists some test input $I_j = (sg'_1, \ldots, sg'_n)$ in TS' (respectively TS) such that sg_l and sg'_l (for $1 \leq l \leq n$) have the same number of steps.

In our experiments described in Section 6.7.5, 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 steps.

Second, our test oracle takes into account all the output values obtained over the entire simulation time interval as opposed to focusing on discrete properties over outputs (e.g., invariants). Our aim is to design a test oracle that imitates the reasoning of domain experts when they inspect Simulink model output signals. 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 signal sg_o is more likely 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 6.1 for definition of $\hat{dist}$). The ground truth oracle is a conceptual oracle that always gives the "right answer" [Barr et al., 2015]. In our work, it encompasses the knowledge of domain expert about the expected model behavior. In Section 6.7.3, we will use our heuristic oracle function, *oracle*, to provide a metric to measure fault revealing ability of test generation techniques. Our fault revealing measure attempts to capture impacts of faults on output signal shapes over the entire simulation time interval as opposed to focusing on violation of discrete properties over model outputs.

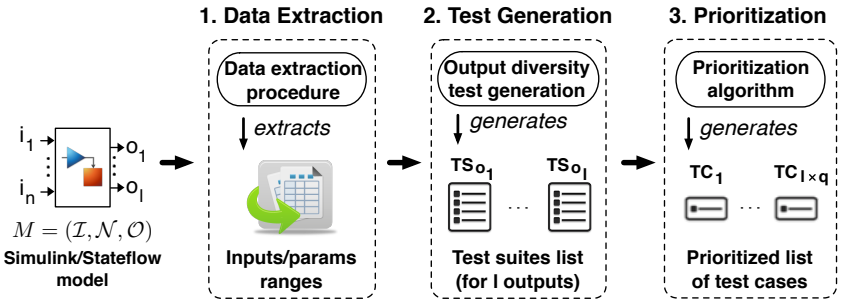


Figure 6.7. An overview of test suite generation for Simulink/Stateflow models in SimCoTest.

6.6 Tool Support

We have implemented our approach in a tool called Simulink Controller Tester (SimCoTest) (`https://sites.google.com/site/simcotesttool/`) [Matinnejad et al., 2016b]. Figure 6.7 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 6.5 (test generation), and (3) prioritizes the generated test cases obtained for different model outputs based on our prioritization algorithm in Figure 6.6 (prioritization).

Figure 6.8 shows the *Test Generation Results* form where different outputs of the model are listed in the left-hand side of the form (i.e., the *Prioritized Test Suites* list) based on the order specified by our prioritization algorithm. Users can click on each output, i.e., o_1 to o_l, to obtain the corresponding test suite in the right side of the form (i.e., the *Test Cases* list). Further, for each test case within a test suite, users can see the information about signals assigned to input variables and values assigned to configuration parameters. Finally, SimCoTest enables users to run each test case in Simulink and generate the simulation results. In particular, SimCoTest automatically plots output signals for each corresponding output variable.

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 and test generation, are partly implemented using MATLAB script functions, which are called from SimCoTest using the MLApp COM interface [The MathWorks Inc., 2016c]. Specifically, 64 MATLAB functions are implemented in roughly 7K lines of MATLAB script and are called from SimCoTest. The main functionalities of SimCoTest have been tested with a test suite containing 300 test cases. 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.

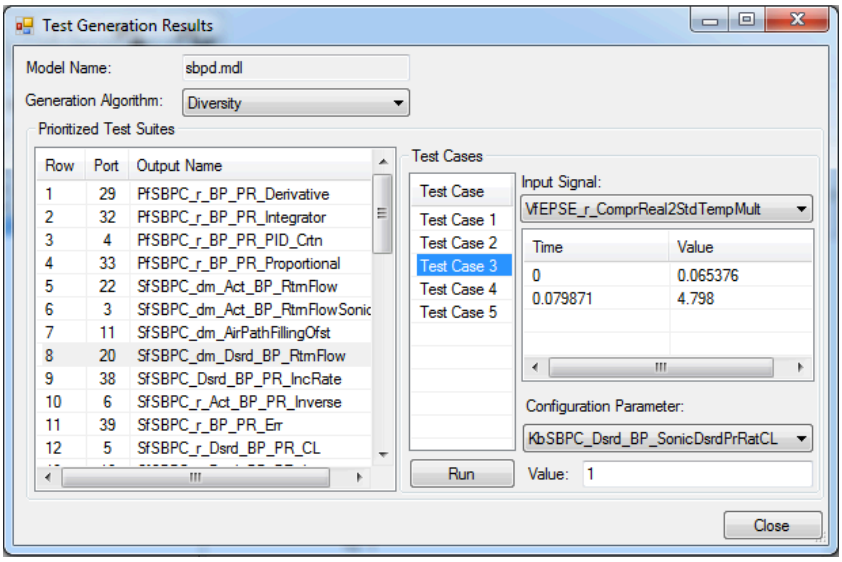


Figure 6.8. SimCoTest test generation results form.

6.7 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.

6.7.1 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? How does the fault revealing ability of the OD algorithm compare with that of a coverage-based test generation strategy?* We investigate whether OD test generation is able to perform better than random testing which is a baseline of comparison, or whether OD is able to perform better than a coverage-based test generation strategy. For coverage-based test generation, we replace the objective function O in our OD algorithm in Figure 6.5 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.

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 [Zhang et al., 2013].

RQ4 (Comparing Test Generation with Commercial Tools). *How does the fault revealing ability of the OD test generation algorithm compare with that of SLDV?* With this question, we compare our output diversity test generation approach (OD) with SLDV in generating effective test suites for Simulink models. Our comparison is performed under the assumption that test oracles are manual. This question enables us to provide evidence that our approach is able to outperform the most widely used industry strength Simulink model testing tool. Further, in contrast to **RQ1** and **RQ2**, where we applied OD to continuous Simulink models, this question has to focus on discrete models because SLDV is only applicable to discrete code-generation models. Hence, this question allows us to investigate the capabilities of OD in finding faults for discrete Simulink models, as well.

6.7.2 Study Subjects

We use four Simulink models in our experiments: Two industrial models, Clutch Position Controller (CPC) and Flap Position Controller (FPC), from Delphi Automotive Systems, our industry partner, and two public domain models, Cruise Controller (CC) [The MathWorks Inc., 2016f] and Clutch Lockup Controller (CLC), from the Mathworks website [The MathWorks Inc., 2016a]. Table 6.1 shows key characteristics of these models. CPC and CC include Stateflows, and FPC and CLC are Simulink models without Stateflows. FPC and CPC are continuous models and incompatible with SLDV. The CC model, which is the largest model from the SLDV tutorial examples, is compatible with SLDV. Since the other tutorial examples of SLDV were small, we modified the CLC model from the Mathworks website to become compatible with SLDV by replacing the continuous and other SLDV-incompatible blocks with their equivalent or approximating discrete blocks. We have made the modified version of CLC available at [Matinnejad, Reza, 2016]. Note that we were not able to make CPC, FPC or any other Delphi Simulink models compatible with SLDV since they contained complex continuous operations and S-Function blocks, and hence, they have to be almost re-implemented before SLDV can be applied to them. In order to compare our approach with SLDV, we considered two widely used coverage criteria for Simulink models, namely *decision coverage* and *modified condition/decision coverage (MC/DC)*, for test generation function of SLDV. Decision coverage [The MathWorks Inc., 2016j], also known as branch coverage, aims to ensure that each one of the possible branches from each decision point is executed at least once and thereby ensuring that all reachable blocks are executed. Examples of decision points in Simulink models are switch blocks and Stateflow states. MC/DC [The MathWorks Inc., 2016j] is a stronger coverage criterion and aims to ensure that a change in the outcome of each condition, independent of any other inputs, causes a change in the outcome of the corresponding decision. Table 6.1 reports the total number of decision points in our study subjects, and the number of MC/DC goals for the models we used in our comparison with SLDV. In addition, we report the total number of Simulink blocks and Stateflow states as well as input/output variables and configuration parameters for each model. CPC and FPC are representative models from the automotive domain with many input variables and blocks. In order to compare OD with SLDV, we use CC from the Mathworks website and the modified version of CLC as both models are reasonably large and complex, and yet compatible with SLDV.

Table 6.1. Characteristics of our study subject Simulink models.

Name	Publicly Available	No. Inputs	No. Configs	No. Outputs	No. Blocks/ States	No. Decision Points	No. MC/DC Goals
CPC	No	10	41	15	590	126	Not Used
FPC	No	21	65	35	810	120	Not Used
CC	Yes	6	4	2	43	12	29
CLC	Yes	2	0	4	81	10	18

6.7.3 Measuring Fault Revealing Ability

We use our heuristic test oracle function, *oracle*, defined in Section 6.5 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 6.5). 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. When we use our test generation approach introduced in Section 6.3, each test case in $\mathcal{TC}$ is related to a specific output of M. So, we only consider signals sg_o^{tc} where test case $tc \in \mathcal{TC}$ is generated for output o. In contrast, when we use SLDV (or other baseline techniques) to generate $\mathcal{TC}$, the set SG contains output signals generated by every test case in $\mathcal{TC}$ and for every output of M. In any case and as discussed in Section 6.5, to compare different test generation approaches, we ensure that the size of SG, i.e., the number of signals that are generated by $\mathcal{TC}$ and are required to be inspected by engineers, for different test generation approaches are the same.

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 conclusively detects a failure. Otherwise, FR returns false. In our work, we set THR to 0.2. We arrived at this value for THR based on our experience and discussions with domain experts. In our experiments, in addition, we obtained and evaluated the results for $THR = 0.15$ and $THS = 0.25$ and showed that our results were not sensitive to such small changes in THR.

6.7.4 Measuring Test Prioritization Effectiveness

To compare the effectiveness of different prioritization algorithms, we measure how early faults are 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 [Yoo and Harman, 2012]. Finally we note that similar to fault revealing measure *FR*, *NTE* values also depend on the fault revealing threshold *THR*. Hence, in our experiments we computed *NTE* values based on three different thresholds of 0.2, 0.15 and 0.25.

6.7.5 Experiment Design

We developed a comprehensive list of Simulink fault patterns and have made it available at [Matinnejad, Reza, 2016]. Examples of fault patterns include incorrect signal data type and incorrect math operations for Simulink models, and incorrect transition condition for Stateflow models. 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 [Zhan and Clark, 2005, Brillout et al., 2009, Binh et al., 2012, Yin et al., 2014]. We note that these fault patterns represent the faults observed in practice. We have developed an automated fault seeding program to automatically generate 44 faulty versions of CPC, 30 faulty versions of FPC, 15 faulty versions of CC, and 15 faulty versions of CLC (one fault per each faulty model). In order to achieve diversity in terms of the locations and the types of faults, our automation seeded faults of different types and in different parts of the models. We also ensured that every faulty model remains executable (i.e., no syntax error).

Having generated the fault-seeded models, we performed three experiments, **EXP-I** to **EXP-III**, to answer **RQ1** to **RQ4**, described below.

EXP-I focuses on answering **RQ1** and **RQ2** using the 74 faulty versions of the time-continuous models from Table 6.1, i.e., CPC and FPC. Figure 6.9(a) shows the overall structure of **EXP-I**. We ran the OD algorithm in Figure 6.5 with vector-based (O_v) and feature-based (O_f) objective functions. We also ran random test generation and coverage-based test generation algorithms. As discussed in Section 6.5, input signals with fewer steps produce simpler output signals and have lower manual oracle cost. However, they may fail to cover a large part of the underlying Simulink model. Hence, in **EXP-I**, we start by generating constant input signals in our algorithms and gradually increase the number of steps in input signals to increase structural coverage until adding more steps in input signals does not bring about additional structural coverage. Also, as mentioned in Section 6.7.1, for the coverage-based algorithm, we replace the objective function O in our OD algorithm with a coverage-based objective function that computes the dynamic test coverage obtained by test cases in TS. Recall from Section 6.4 that the notion of test coverage in our work is specific to model outputs. Accordingly, our coverage-based objective function considers the dynamic test coverages related to the output for which we are generating the tests. As shown in Figure 6.9(a), the OD and coverage-based algorithms generate l separate test suites for l outputs of the model under test, while random test generation generates one test suite for all the model outputs. As indicated in Table 6.1, the number of outputs, l, is equal to 15 for CPC and 37 for FPC models.

For each faulty model and each objective function, we ran OD and our coverage-based algorithm for 600 sec and created test suites of size q where q took the following values: 3, 5, and 10. We also used random test generation to generate test suites with the same size. We chose to examine

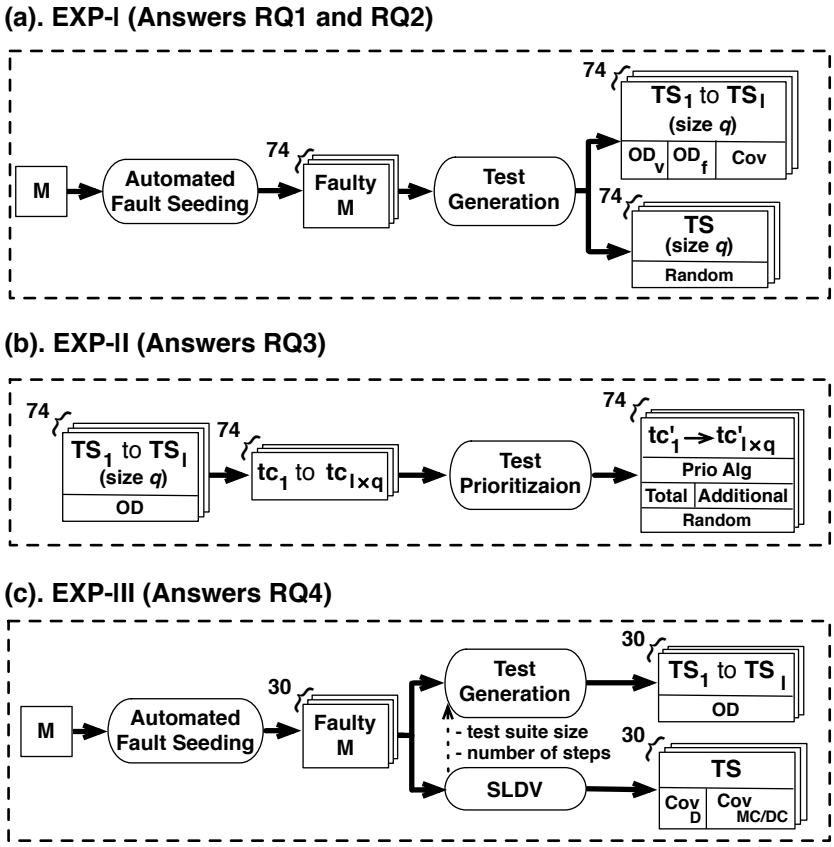


Figure 6.9. Our experiment design: test generation algorithms are repeated for 20 times to account for their randomness. EXP-II is repeated for all the fault revealing OD test suites from EXP-I. Further, random prioritization is repeated for 20 times.

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) [Varrette et al., 2014]. 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 6.9(b) shows the overall structure of **EXP-II**. We used our prioritization algorithm in Figure 6.6 to rank the test cases generated by the OD test generation algorithm for 74 faulty versions of CPC and FPC models. We also used random prioritization as well as total and additional coverage-based test prioritization strategies [Zhang et al., 2013] 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 and ignored those test suites obtained in EXP-I that were not able to detect any fault since test prioritization is irrelevant for them.

We ran random prioritization for 20 times to account for its randomness. Recall from Section 6.4 that our prioritization algorithm in Figure 6.6 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 coverage 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 each coverage node is 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**. We also note that the two other models from Table 6.1 have a few outputs. Further, the test suites are small in our experiments. As a result, test prioritization can have only a slight impact on the manual test oracle for these models. So, we chose to not run test prioritization for them.

EXP-III answers **RQ4** and is performed on the SLDV-compatible subject models from Table 6.1, i.e., CC and CLC. Figure 6.9(c) shows the overall structure of **EXP-III**. To answer **RQ4**, we compare the fault revealing ability of the test suites generated by SLDV with that of the test suites generated by OD test generation algorithm. We give SLDV and OD the same execution time budget (200 sec in our experiment). This time budget was sufficient for SLDV to achieve a high level of structural coverage over the subject models. Further, we ensure that the generated test suites have the same test oracle cost. Specifically, for each faulty model M, we first use SLDV to generate a test suite TS_M based on both the decision and the MC/DC coverage criteria within the time allotted (200 sec). We then apply OD to M to generate a test suite TS'_M such that TS_M and TS'_M have the same test oracle cost (see Section 6.5). We have implemented a Matlab script that enables us to extract the size of the test suites as well as the number of steps in input signals for each individual test input of TS_M. Further, we have modified the OD algorithm in Figure 6.5 so that it generates input signals containing a desired number of steps. Also note that, while SLDV generates one test suite for all model outputs, OD generates one test suite per each output. Nevertheless, the total number of output signals that need to be inspected by engineers is the same for both techniques, and hence, they have the same oracle cost. Finally, we note that while SLDV is deterministic and is executed once per each faulty model, OD is randomized, and hence, we rerun it 20 times for each faulty model. Overall, **EXP-III** took around 3.4 hours to run on our HPC clusters to generate test cases using both SLDV and our OD algorithm. It would have taken more than two months on a single node.

6.8 Results

This section provides responses, based on our experiment design, for research questions **RQ1** to **RQ4** described in Section 6.7.

RQ1 (Comparing Test Generation with State-of-the-art). To answer **RQ1**, we ran **EXP-I** to compare our OD algorithm with random and coverage-based test generation. Figures 6.10(a) to (c) show the fault revealing ability of random test generation (R), coverage-based test generation (Cov), and

OD with the objective functions O_v and O_f. Each distribution in Figures 6.10(a) to (c) contains 74 points. Each point relates to one faulty model and represents either the average aggregated oracle *Oracle* or the average fault revealing measure *FR* over 20 different sets of test suites with a fixed size and obtained by applying a test generation algorithm to that faulty model. Note that the *FR* values are computed based on three different thresholds *THR* of 0.2, 0.15, and 0.25. For example, a point with (x = R) and (y = 0.149) in the *Oracle* plot of Figure 6.10(a) indicates that the 20 different random test suites with size 3 generated for one faulty model achieved an average aggregated oracle *Oracle* of 0.149. Similarly, a point with (x = OD(O_f)) and (y = 0.85) in any of the *FR* plots of Figure 6.10(b) indicates that among the 20 different sets of l test suites with size 5 generated by applying OD with objective function O_f to l outputs of one faulty model, for 17 sets at least one test case was able to reveal the fault (i.e., *FR* = 1) while for 3 sets none of the test cases could reveal the fault (i.e., *FR* = 0).

To statistically compare the *Oracle* and *FR* values, we performed the non-parametric pairwise Wilcoxon Pairs Signed Ranks test [Capon, 1991], and calculated the effect size using Cohen's d [Cohen, 1977]. 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 $d \geq 0.8$ [Cohen, 1977].

Testing differences in the average *Oracle* and *FR* distributions, for all the three thresholds and with all the three test suite sizes, shows that OD with both objective functions O_f and O_v performs significantly better than both random and coverage-based test generation. 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 the comparisons between OD and coverage-based, the effect size is "high" for all the comparisons except for the comparisons of *FR* distributions for OD(O_v) with test suite sizes 5 and 10, where the effect size is "medium". *To summarize,* the fault revealing ability of OD significantly outperforms that of random testing and coverage-based test generation.

Comparing coverage-based and random. Using the results in Figure 6.10, we can also compare the fault revealing ability of coverage-based (Cov) and random (R) test generation. Testing differences in the average *Oracle* and *FR* distributions, for all the three test suite sizes and with all the three thresholds, indicates that coverage-based test generation performs significantly better than random test generation with one exception. Only for test suite size 3 and threshold 0.25, there is no statistically significant difference between their average *FR* distribution, though Cov still achieves higher mean and median *FR* values. Further, the effect size is consistently "small" for all the comparisons where there exist a statistically significant difference between Cov and random. *To summarize,* the fault revealing ability of coverage-based test generation outperforms that of random test generation.

Comparing dynamic test coverage. In Section 6.1, we motivated our output diversity approach by comparing two different test outputs of a faulty model which were equally desirable with respect to structural coverage but yielded highly different fault revealing abilities. Here, we report the structural coverages achieved by different test generation algorithms to provide evidence that achieving higher structural coverage does not necessarily lead to better fault revealing ability. Figure 6.11 compares the average of dynamic test coverage percentages obtained by 20 different runs of different test generation algorithms over the faulty CPC and FPC models. Recall from Section 6.4 that in our work the notion of dynamic test coverage is specific to a model output. Hence, in Figure 6.11 we compute

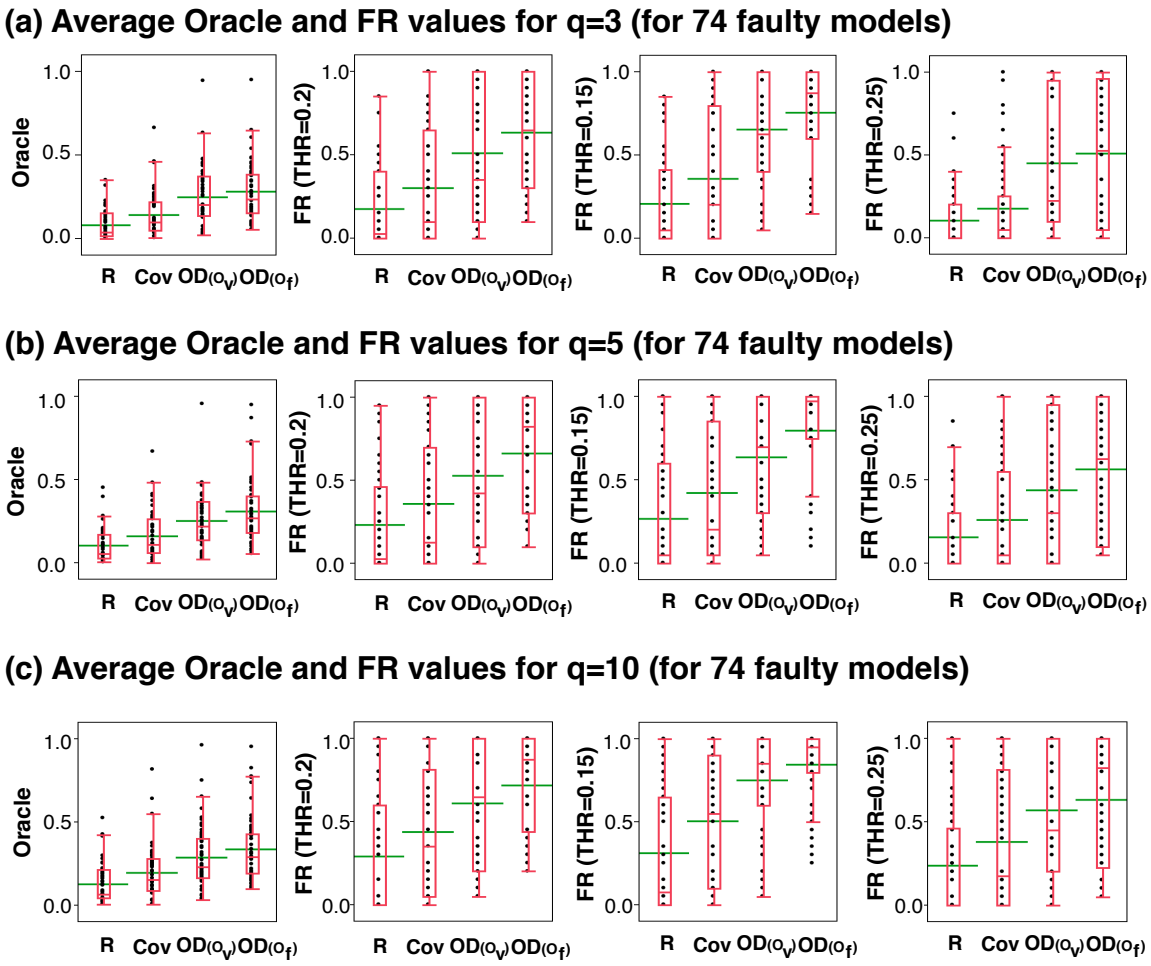


Figure 6.10. Boxplots comparing average aggregated oracle values (*Oracle*) and fault revealing measures (*FR*) of OD (with both diversity objectives), coverage-based (Cov) and random test suites (R) for different thresholds and different test suite sizes.

the percentages of structural coverage obtained by each test suite over the backward static slice of the output related to that test suite. That is, for a test suite TS related to an output o, we compute the percentage of branches in $static\_slice(o)$ which are covered by test cases in TS. For each faulty model and each test generation algorithm, we report the average of dynamic test coverages obtained by fault revealing test suites generated for that faulty model. As shown in the figure, coverage-based test generation was able to achieve higher structural coverages than other algorithms across all test suite sizes. Specifically, it achieved, on average, 89%, 91% and 93% of decision coverage for test suite sizes 3, 5 and 10, respectively. As shown in the figure, this is at least 3% higher than the structural coverages achieved by other test generation algorithms, across all test suite sizes.

RQ2 (Comparing O_f with O_v). The results in Figure 6.10 also compares the average *Oracle* and *FR* values for the feature-based, OD(O_f), and the vector-based, OD(O_v), output diversity algorithms. As for the average *Oracle* distributions, the statistical test results indicate that OD(O_f) performs significantly better than OD(O_v) for test suite sizes 5 and 10 with a "small" effect size. For test suite size 3, there is no statistically significant difference, but OD(O_f) achieves higher mean and median

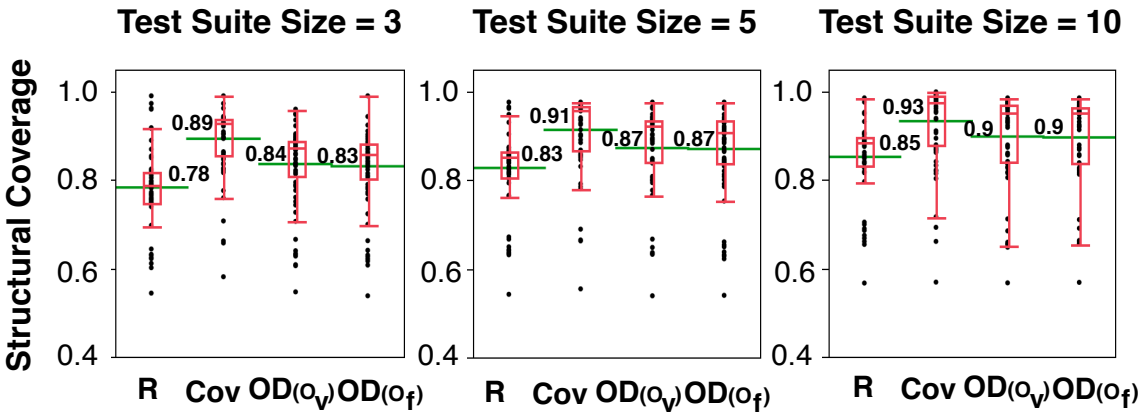


Figure 6.11. The percentages of dynamic test coverage achieved by different test generation algorithms over the faulty versions of CPC and FPC subject models for different test suite sizes.

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 test suite sizes 3, 5, and 10, respectively. That is, across all the faults and with all test suite sizes, the average probability of detecting a fault is at least %10 higher when we use OD(O_f) instead of OD(O_v). *To summarize,* the fault revealing ability of the OD algorithm with the feature-based diversity objective is higher than that of the OD algorithm with the vector-based diversity objective.

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 with both O_v and O_f ran for the same number of iterations in the given test execution budget. Recall that in Section 6.3.3, we discussed the asymptotic time complexity of individual iterations of OD(O_f) and OD(O_v) algorithms. Our investigation showed that executing the model with the given q test inputs takes the largest part of the execution time of each iteration. Specifically, executing a single model simulation in our experiments using the Matlab *cvsim* function [2] took 1.1 sec, on average, while computing O_f and O_v took only 12 and 5 msec, on average. Given that both OD(O_f) and OD(O_v) are given the same test execution budget in **EXP-I**, they almost always 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 vector-based output diversity objective function.

RQ3 (Comparing Test Prioritization with State-of-the art). To answer **RQ3**, we performed **EXP-II** with the fault revealing samples of test suites generated in **EXP-I** by OD with the better diversity objective (i.e., OD with the feature-based diversity). We used different prioritization algorithms to prioritize the combined set of all test cases generated by OD(O_f) for 15 outputs of CPC model as well as 37 outputs of FPC model. Figures 6.12 and 6.13 compare the average NTE distributions obtained by random prioritization (R), total (Tot) and additional (Add) coverage-based prioritization, and our

[2] *cvsim* function executes a Simulink model and computes the structural coverage at the same time.

test prioritization algorithm (PrioAlg) for CPC and FPC models, respectively. Note that in contrast to *Oracle* and *FR* measures used in **EXP-I**, *NTE* measure is not normalized. Hence, we present the results of **EXP-II** in separate plots for CPC and FPC case studies. Each distribution in Figures 6.12(a) to (c) (resp. in Figures 6.13(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 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 random prioritization algorithm. For example, a point with (x = Tot) and (y = 12.35) in any of the plots in Figure 6.12(c) indicates that among the 150 (i.e., 15×10) test cases generated for 15 outputs of CPC model, on average, 12.35 test cases need to be evaluated to find the fault when test cases are prioritized using total coverage algorithm. Similarly, a point with (x = PrioAlg) and (y = 9.8) in any of the plots in Figure 6.13(b) indicates that among all 185 (i.e., 37×5) test cases generated for 37 outputs of FPC model, on average, 9.8 test cases need to be evaluated to find the fault when test cases are prioritized using our test prioritization algorithm.

To statistically compare the *NTE* values, we used the same setting as in **EXP-I**. Recall that lower NTE values denote faster fault detection, 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 our prioritization algorithm performs significantly better than the other three prioritization algorithms. In addition, for all the comparisons between our prioritization algorithm and both random and total coverage-based prioritization, the effect size is consistently "high". For the comparisons between our prioritization algorithm and additional coverage-based prioritization, the effect size is "high" for test suite sizes 5 and 10, and "medium" for test suite size 3. *To summarize,* our prioritization algorithm significantly outperforms random and both total and additional coverage-based prioritization, as it yields significantly smaller *NTE* values compared to the other algorithms.

Comparing manual test effort. Using *NTE* measure enables us to directly compare the manual test effort when different test prioritization algorithms are used to order the tests. Specifically, for CPC model and with threshold 0.2, the average *NTE* is 19.5, 32.2, and 65.3 with random prioritization, 12.6, 17.7, and 27.7 with total coverage prioritization, 9.1, 13.8, and 19.9 with additional coverage prioritization, and 5.6, 5.9, and 10.3 with our prioritization algorithm, for test suite sizes 3, 5 and 10, respectively. Similarly, for FPC model and with threshold 0.2, the average *NTE* is 55.9, 90.4, and 166.4 with random prioritization, 27.0, 47.6, and 88.7 with total coverage prioritization, 19.7, 31.2, and 56.2 with additional coverage prioritization, and 9.3, 14.8, and 31.3 with our prioritisation algorithm, for test suite sizes 3, 5 and 10, respectively. That is, across all the faults and with all test suite sizes, the average number of tests that need to be evaluated is at least %45 less when we order the tests using our prioritization algorithm. Further, on average, 12.1 less test cases need be checked when our prioritization algorithm is used compared to additional coverage-based prioritization which is the second best algorithm. Our experience shows that inspecting output signals by engineers may take up to one or two hours for a single test case. Given that, our prioritization algorithm brings significant improvement in terms of the manual test effort.

RQ4 (Comparing Test Generation with Commercial Tools). To answer RQ4, we used the better diversity objective from **RQ2** (i.e., OD with the feature-based diversity objective) and performed **EXP-III** on 30 faulty models of CC and CLC. Recall that in **EXP-III**, we first use SLDV to generate

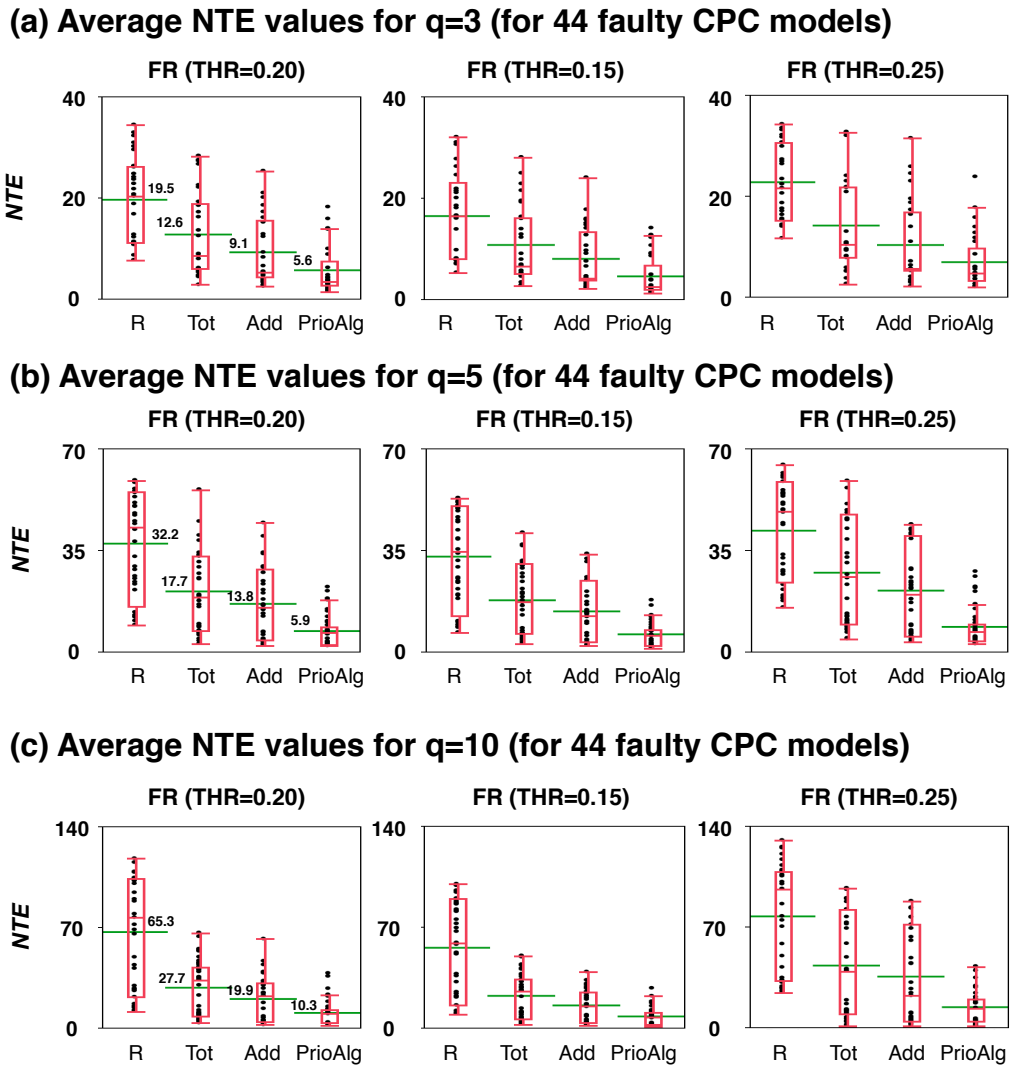


Figure 6.12. Boxplots comparing average number of tests to be evaluated (*NTE*) by our prioritization algorithm, coverage-based (total and additional) and random test prioritization with different thresholds and different test suite sizes for CPC case study.

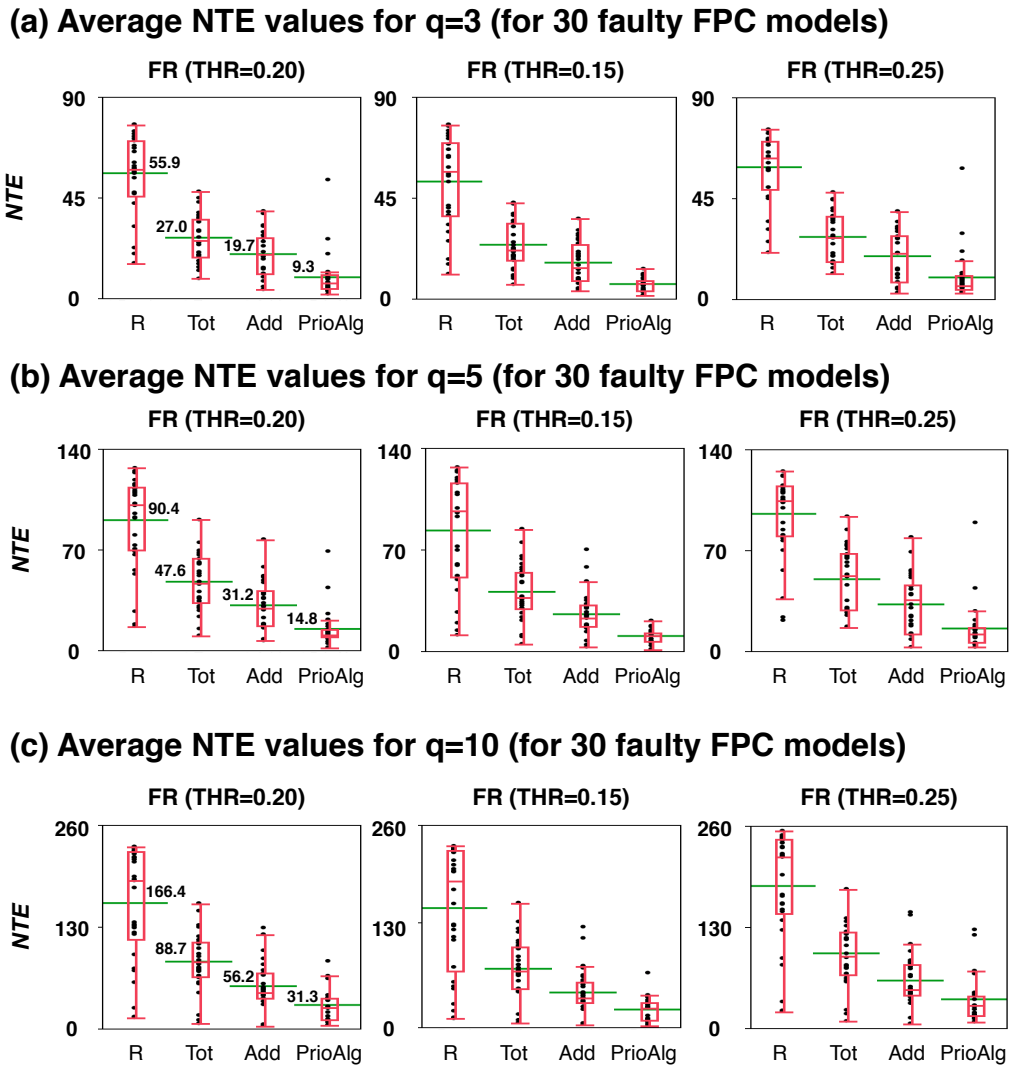


Figure 6.13. Boxplots comparing average number of tests to be evaluated (*NTE*) by our prioritization algorithm, coverage-based (total and additional) and random test prioritization with different thresholds and different test suite sizes for FPC case study.

test suites based on Decision and MC/DC coverage criteria, and then run OD to generate test suites with the same oracle cost, i.e., with the same size and test input complexity. We evaluate the fault revealing ability of SLDV and OD by comparing the aggregated oracle *Oracle* and the fault revealing measure *FR* values obtained over these 30 faulty models. In addition, we investigate if any of SLDV and OD subsumes the other technique (fault revealing subsumption). That is, we determine if any of OD and SLDV does not find any additional faults missed by the other technique. Finally, we compare the structural coverage achieved by each of SLDV and OD over these 30 faulty models. We report coverage results for two reasons: (1) We confirm our earlier claim that with the timeout of 120 sec used in **EXP-III**, SLDV has been able to achieve high structural coverage. (2) We provide further evidence that achieving higher structural coverage does not necessarily lead to better fault revealing ability.

Comparing fault revealing ability. We computed *Oracle* and *FR* values with three *THR* values of 0.15, 0.20, and 0.25 over the test suites generated by SLDV and OD. Figures 6.14(a) and (b) compare the distributions obtained for the 30 faulty models of CC and CLC, when SLDV was used with decision and MC/DC coverage criteria, respectively. Recall that SLDV is deterministic, while OD is randomized. So, in Figure 6.14, each point in distributions related to SLDV shows the value of *Oracle* or *FR* obtained for one and the only one test suite generated by SLDV for each faulty model. In contrast, each point in distributions related to OD shows the *average* value of *Oracle* or *FR* for 20 different sets of test suites generated by OD for different outputs of each faulty model. Further, for each faulty model, SLDV yields a *FR* value of one or zero, respectively indicating whether SLDV reveals the fault or not. However, for OD, we compute an average *FR* value over 20 different runs, which is between zero and one, indicating an estimated probability for OD to reveal a fault.

As shown in Figures 6.14(a) and (b), the *Oracle* and *FR* values obtained for SLDV are very small compared to those obtained by *OD*. Testing differences in *Oracle* and *FR* distributions with all the three thresholds shows that OD performs significantly better than SLDV for both coverage criteria. In addition, for all the eight comparisons depicted in Figure 6.14, the effect size is "high".

Fault revealing subsumption. Tables 6.2 to 6.3 compare performance of OD and SLDV for individual faults. Specifically, Tables 6.3 and 6.4 show, for each faulty model, the distribution of *Oracle* values obtained by OD and the single value of *Oracle* obtained by SLDV, when SLDV was used with Decision and MC/DC coverage, respectively. Note that $\bar{x}$ shows the mean, and Q_1, Q_2, and Q_3 refer to the three quartiles of the *QO* distribution obtained by 20 different runs of OD (i.e., Q_1, Q_2, and Q_3 are the 25th, 50th, and 75th percentiles, respectively). In addition, we report (as denoted by $\mathcal{P}$ in Tables 6.3 and 6.4) the percentages of OD runs that achieve a *QO* value greater than or equal to that obtained by SLDV. For example, for faults 1 and 5 in Table 6.3, 100% and 90% of the OD runs, respectively, yield *QO* values that are not worse than those generated by SLDV when it was used with Decision coverage. Similarly, for the same faults in Table 6.4, 100% of the OD runs yield *QO* values that are not worse than those generated by SLDV when it was used with MC/DC coverage. Table 6.2 shows, for each faulty model and when considering *FR* with threshold 0.2, whether SLDV is able to identify the fault (denoted by Yes) or not (denoted by No). Further, the table shows, out of the 20 runs, how many times OD is able to find the fault. Note that the results for the thresholds 0.15 and 0.25 were similar to those in Table 6.2.

Based on Tables 6.3 and 6.4, 9 and 6 faults go undetected by SLDV irrespective of the threshold

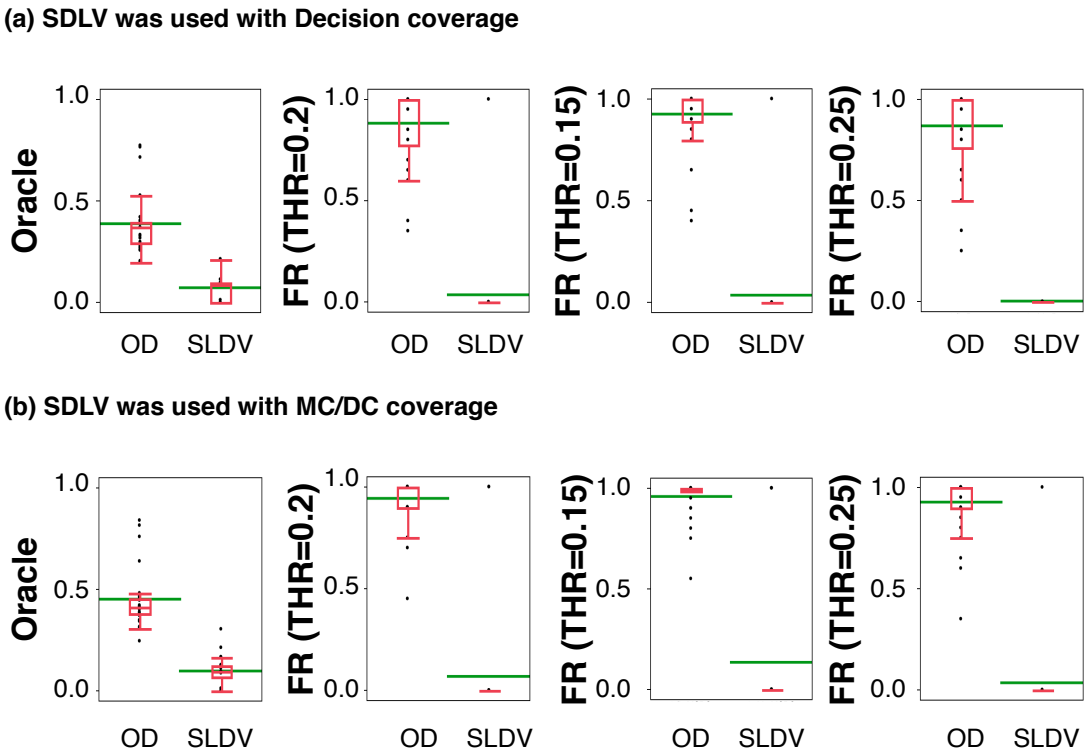


Figure 6.14. Boxplots comparing aggregated oracle values (*Oracle*) and fault revealing measures (*FR*) of OD and SLDV for different thresholds.

Table 6.2. The number of fault revealing runs of OD (out of 20) for our 30 faulty models, and the fault(s) that SLDV is able to find with a threshold (*THR*) of 0.2.

SDLV was used with Decision coverage															
Fault No.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
OD	**20**	**14**	**16**	**20**	**12**	**13**	**20**	**13**	**19**	**17**	**19**	**7**	**14**	**16**	**8**
SLDV	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
Fault No.	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
OD	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
SLDV	No	No	No	No	No	No	No	No	**Yes**	No	No	No	No	No	No
SDLV was used with MC/DC coverage															
Fault No.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
OD	**20**	**14**	**20**	**20**	**18**	**18**	**20**	**18**	**20**	**20**	**20**	**9**	**15**	**18**	**15**
SLDV	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
Fault No.	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**30**
OD	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
SLDV	No	No	**Yes**	No	No	No	No	No	**Yes**	No	No	No	No	No	No

Table 6.3. Aggregated oracle *Oracle* distributions for OD and single *Oracle* values for SLDV per each faulty model, when SLDV was used with decision coverage.

1			2		3		4		5		6		7		8	
OD		SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV
$\bar{x}$	0.772		0.419		0.256		0.714		0.288		0.259		0.764		0.297	
Q_1	0.667		0.189		0.166		0.614		0.178		0.192		0.674		0.025	
Q_2	0.762	0.012	0.406	0	0.272	0	0.759	0	0.296	0.096	0.217	0.096	0.782	0.096	0.22	0.096
Q_3	0.923		0.594		0.361		0.841		0.389		0.361		0.867		0.561	
$\mathcal{P}$	1.0		1.0		1.0		1.0		0.9		0.95		1.0		0.7	

9			10		11		12		13		14		15		16	
OD		SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV
$\bar{x}$	0.526		0.316		0.402		0.203		0.275		0.268		0.198		0.398	
Q_1	0.356		0.217		0.357		0.365		0.385		0.361		0.330		0.374	
Q_2	0.482	0.213	0.335	0.096	0.385	0.095	0.039	0	0.289	0.096	0.289	0.096	0.128	0	0.394	0
Q_3	0.771		0.386		0.482		0		0.110		0.192		0.046		0.435	
$\mathcal{P}$	0.85		0.95		1.0		1.0		0.85		1.0		1.0		1.0	

17			18		19		20		21		22		23		24	
OD		SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV
$\bar{x}$	0.539		0.534		0.536		0.531		0.541		0.403		0.544		0.538	
Q_1	0.531		0.520		0.529		0.518		0.534		0.376		0.546		0.536	
Q_2	0.547	0.087	0.541	0.087	0.544	0.087	0.535	0.087	0.547	0.087	0.407	0	0.547	0.113	0.547	0.087
Q_3	0.547		0.547		0.547		0.547		0.547		0.437		0.546		0.547	
$\mathcal{P}$	1.0		1.0		1.0		1.0		1.0		1.0		1.0		1.0	

25			26		27		28		29		30	
OD		SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV
$\bar{x}$	0.536		0.533		0.528		0.533		0.532		0.526	
Q_1	0.530		0.521		0.512		0.524		0.513		0.515	
Q_2	0.546	0.113	0.540	0	0.537	0.087	0.541	0.087	0.547	0.087	0.530	0
Q_3	0.547		0.547		0.547		0.547		0.547		0.547	
$\mathcal{P}$	1.0		1.0		1.0		1.0		1.0		1.0	

value (i.e., for these faults we have $QO = 0$ for SLDV), when SLDV was used with decision and MC/DC coverage, respectively. There is no fault that SLDV can possibly detect ($QO > 0$) but OD cannot. For 24 and 25 faults, all 20 runs of OD yield results that are at least as good as those of SLDV ($\mathcal{P} = 1$), when SLDV was used with decision and MC/DC coverage, respectively. For all the faults in Table 6.3 and 6.4, the average of QO obtained by OD (denoted by $\bar{x}$) is higher than the value of QO obtained by SLDV. Based on Tables 6.2, SLDV identifies only one fault with a threshold of 0.2, and that particular fault is also detected by all the 20 runs of OD. The results for thresholds 0.15 and 0.25 are similar.

Comparing coverage. Figure 6.15 compares the structural coverage percentages (i.e., decision and MC/DC coverage) achieved by test suites generated by OD and SLDV over the faulty models of CC and CLC. As before, the distribution for SLDV shows the percentages of structural coverage achieved by individual test suites generated for individual faulty models, while the distribution for OD shows the average of structural coverage percentages obtained by 20 different runs of OD for each faulty

Table 6.4. Aggregated oracle *Oracle* distributions for OD and single *Oracle* values for SLDV per each faulty model, when SLDV was used with MC/DC coverage.

1			2		3		4		5		6		7		8	
OD		SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV
$\bar{x}$	0.839		0.384		0.415		0.759		0.381		0.342		0.814		0.481	
Q_1	0.751		0.147		0.385		0.691		0.289		0.048		0.771		0.250	
Q_2	0.873	0.012	0.360	0	0.433	0	0.755	0	0.4	0.096	0.232	0.096	0.819	0.096	0.574	0.096
Q_3	0.936		0.603		0.482		0.864		0.481		0.615		0.867		0.675	
$\mathcal{P}$	1.0		1.0		1.0		1.0		1.0		0.65		1.0		0.85	

9			10		11		12		13		14		15		16	
OD		SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV
$\bar{x}$	0.638		0.421		0.445		0.245		0.315		0.308		0.348		0.431	
Q_1	0.482		0.385		0.385		0.561		0.386		0.386		0.542		0.335	
Q_2	0.676	0.213	0.408	0.096	0.482	0.096	0.105	0.096	0.386	0.096	0.289	0.096	0.269	0	0.421	0.166
Q_3	0.867		0.482		0.482		0		0.193		0.289		0.179		0.519	
$\mathcal{P}$	0.9		1.0		1.0		0.55		0.95		1.0		1.0		1.0	

17			18		19		20		21		22		23		24	
OD		SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV
$\bar{x}$	0.527		0.536		0.524		0.519		0.536		0.449		0.549		0.547	
Q_1	0.516		0.528		0.503		0.503		0.525		0.435		0.547		0.539	
Q_2	0.535	0.127	0.545	0.304	0.530	0.123	0.523	0.087	0.546	0.123	0.445	0.166	0.547	0.113	0.547	0.123
Q_3	0.547		0.547		0.547		0.547		0.547		0.460		0.547		0.547	
$\mathcal{P}$	1.0		1.0		1.0		1.0		1.0		1.0		1.0		1.0	

25			26		27		28		29		30	
OD		SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV	OD	SLDV
$\bar{x}$	0.561		0.544		0.512		0.521		0.510		0.531	
Q_1	0.547		0.542		0.498		0.495		0.479		0.524	
Q_2	0.554	0.113	0.547	0	0.523	0.123	0.529	0.123	0.515	0.087	0.544	0
Q_3	0.573		0.547		0.540		0.547		0.547		0.547	
$\mathcal{P}$	1.0		1.0		1.0		1.0		1.0		1.0	

model. As shown in the figure, SLDV was able to achieve, on average, 89% decision and 94% MC/DC coverage over CC and CLC models. In contrast, OD achieved, on average, 81% decision and 83% MC/DC coverage. In addition, SLDV and OD have been able to cover all fault-seeded blocks. This shows that within 200 sec, SLDV had sufficient time to cover the structure of the 30 faulty models. Indeed, for 44 out of 60 runs of SLDV, it terminated before the timeout period elapsed. Hence, by increasing the execution time, it is unlikely that SLDV's fault revealing ability would be impacted.

In summary, our comparison of SLDV and OD shows that: (1) For both studied models and both coverage criteria, OD is able to reveal significantly more faults compared to SLDV. (2) OD subsumes SLDV in revealing faults with both decision and MC/DC coverage: Any fault identified by SLDV is also identified by OD. (3) SLDV was able to cover a large part of the underlying models within the given timeout period (including all the fault-seeded blocks), and further, it achieved slightly higher decision and MC/DC coverage over study subjects compared to OD. However, covering a fault does

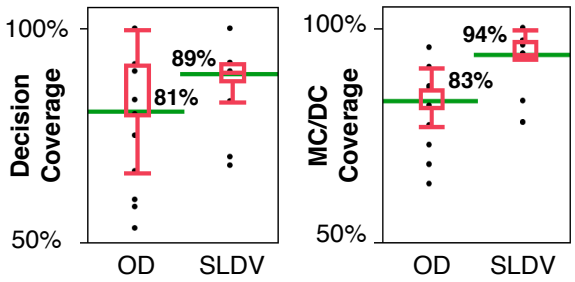


Figure 6.15. The percentages of decision and MC/DC coverages achieved by OD and SLDV over the 30 faulty versions of CC and CLC subject models.

not necessarily lead to detecting that fault. In particular, SLDV was able to reveal only one and two faults that it could cover, when used with decision and MC/DC coverage. (4) Finally, our results on comparing SLDV and OD are not impacted by small modifications in the threshold values used to compute the fault revealing measure *FR*.

Discussion. *Why does SLDV perform poorly compared to OD?* Our results in **RQ4** show that, compared to the output diversity (OD) algorithm, SLDV is less effective in revealing faults in Simulink models. In our experiment, even though test suites generated by SLDV cover most faulty parts of the Simulink models, the outputs produced by these test suites either do not deviate or only slightly deviate from the ground truth oracle, hence yielding very small *QO* values. In contrast, OD generates test suites with output signals that are more distinct from the ground truth oracle. Note that, as discussed in Section 6.1, any deviation should exceed some threshold to be conclusively deemed a failure. For example, Figures 6.16(b) and (d) show two output signals (solid lines) of a faulty model together with the oracle signals (dashed lines) generated by OD and SLDV, respectively. Note that the range of the Y-axis in Figure 6.16(b) is 1000 times larger than that in Figure 6.16(d). Hence, the deviation from the oracle in Figure 6.16(b) is much larger than that in Figure 6.16(d). In particular, the signals in Figures 6.16(b) and (d) respectively produce *QO* values of 0.43 and 0.01. Therefore, the output in Figure 6.16(b) is more fault revealing than the one in Figure 6.16(d).

Since SLDV is commercial and its technical approach description is not publicly available, we cannot precisely determine the reasons for its poor performance. We conjecture, however, that the reason for SLDV's poor fault finding lies in its input signal generation strategy. Specifically, all value changes in the input signals generated by SLDV typically occur during the very first simulation steps, and then, the signals remain constant for the most part and until the end of the simulation time. In contrast, changes in the input signals generated by OD can occur at any time during the entire simulation period. For example, Figures 6.16(a) and (c) show two examples of input signals generated by OD and SLDV, respectively. In both case, the signal value changes from one to zero. However, for the signal in Figure 6.16(a), the change occurs almost in the middle of the simulation (at 6 sec), while in Figure 6.16(c), the change occurs after the first step (at 0.01 sec). Signals in Figures 6.16(a) and (c) happen to cover exactly the same branches of the underlying model. However, they, respectively, yield the outputs in Figures 6.16(b) and (d) with drastically different fault revealing ability.

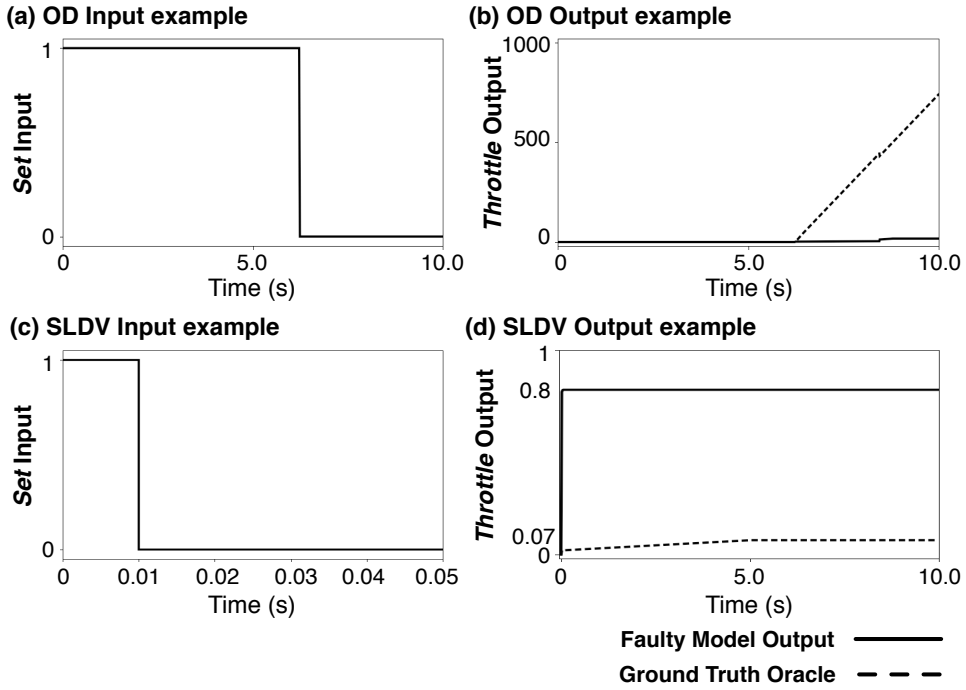


Figure 6.16. Examples of test inputs and output signals generated by SLDV and OD algorithm.

6.9 Conclusions

In this chapter, we identified the key challenges concerning existing testing and verification techniques for Simulink models, and explained how we deal with them in our work. We distinguished Simulink simulation and code generation models and illustrated differences in their behaviors using examples. In contrast to the existing testing approaches that are only applicable to code generation Simulink models, we proposed a test generation and a test prioritization algorithm for both kinds of Simulink models based on our notion of feature-based output diversity for signals. Our test generation algorithm is implemented using a meta-heuristic search approach that is guided to produce test outputs exhibiting a diverse set of signal features. Our prioritization algorithm combines test coverage and test suite output diversity to rank the generated test cases. Our evaluation is performed using two industrial and two public domain Simulink models and showed that (1) Our test generation algorithm significantly outperforms random and coverage-based test generation. (2) Our test generation algorithm when used with the feature-based notion of output diversity generates higher fault revealing rates compared to when output diversity is measured based on the Euclidean distance between signal vectors. (3) Our test prioritization algorithm significantly outperforms random and coverage-based test prioritization. (4) Our approach is able to reveal significantly more faults compared to Simulink Design Verifier (SLDV), and further, it subsumes SLDV, as it is able to find the faults identified by SLDV with a 100% probability.

Chapter 7

Related Work

In this chapter, we focus on comparing our approach with the most related research threads to our work on controller testing and signal processing, surrogate modeling, Simulink model testing, source code testing, and test case prioritization.

7.1 Controller testing and Signal Generation

Recent work in the intersection of Simulink testing and signal processing has focused on test input signal generation using evolutionary search methods [Baresel et al., 2003, Windisch and Al Moubayed, 2009a, Windisch and Al Moubayed, 2009b, Lindlar et al., 2010, Wilmes and Windisch, 2010]. Complex continuous input signals are generated either by sequencing parameterized base signals [Baresel et al., 2003, Windisch and Al Moubayed, 2009a], or by modifying parameters of Fourier series [Windisch and Al Moubayed, 2009b]. These techniques, however, assume automated oracles, e.g., assertions, are provided. Since test oracles are automated, they do not pose any restriction on the shape of test inputs. In our work, however, we restrict variations in input signal shapes as more complex inputs increase the oracle cost. Similar to our work, the work of [Zander-Nowicka, 2008] proposes a set of signal features. These features are viewed as basic constructs which can be composed to specify test oracles. In our work, since oracle descriptions do not exist, we use features to improve test suite effectiveness by diversifying feature occurrences in test outputs.

Continuous controllers have been widely studied in the control theory domain [Nise, 2004, Araki, 2002, Wescott, 2000], where the focus has been to optimize the controller behavior for a specific application by design optimization [Nise, 2004], or a specific hardware configuration by configuration optimization [Araki, 2002]. Overall existing work in control theory 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 and cannot substitute systematic testing as addressed by our approach. Robust control toolbox [Mathworks, 2016] from Mathworks is an example of controller analysis tools used in control engineering. Robust is a Matlab toolbox that provides functions for analyzing and tuning performance and robustness of closed-loop control systems in the presence of uncertainty in plant models. The robust toolbox, however, tunes the plant model parameters only over a step signal input. Further, it is only applicable if the underlying model, i.e., the plant in this case, implements a Linear Time-Invariant (LTI) dynamic system. In reality, many controllers are neither linear not time-invariant.

7.2 Surrogate modeling

Surrogate modeling has been used to scale up computation in various application domains such as avionics [Ong et al., 2003], chemical systems [Caballero and Grossmann, 2008], and medical domain [Douguet, 2010]. Similar to our work, they use surrogate models in the context of evolutionary algorithms, and their goal is to approximate complex mathematical models with faster-to-compute models, i.e., surrogate models, to speed up highly time-consuming and costly simulations and experiments. Our work is the first to apply surrogate models for testing continuous controllers. Furthermore, our Hill Climbing (HC) single-state search algorithm represents a new combination of surrogate modeling and evolutionary algorithms by precisely showing when the search has to run real simulations and when the surrogate model is sufficient. Finally, our algorithm continuously refines surrogate models using the real simulations performed during the search.

In Chapter 4, we applied surrogate modeling in conjunction with dimensionality reduction to test controllers. *Dimensionality reduction* in our work differs from *input domain reduction* used in software code testing where the goal is to remove irrelevant variables prior to test case generation via static analysis [Harman et al., 2007, McMinn et al., 2012]. By dimensionality reduction, we mean identifying significant dimensions of each fitness function to focus exploration on those dimensions.

7.3 Simulink Model Testing

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

7.3.1 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 [Peranandam et al., 2012, Gadkari et al., 2008, Bohr and Eschbach, 2011]. 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 [Windisch, 2009, Windisch, 2010, Zhan and Clark, 2004]. Such fitness functions are typically defined in terms of metrics measuring the distance between input values and conditions characterizing the targeted block/state.

Reachability analysis is used to generate coverage-adequate test inputs or to provide proofs of correctness by showing unreachability of the faulty model parts [Hamon, 2008, Mohalik et al., 2014, Satpathy et al., 2012]. 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 [Reactive Systems Inc., 2016a, Cleaveland et al., 2008], a well-known commercial tool for model-based testing of Simulink models, adapts a guided random test generation strategy consisting of two steps [Satpathy et al., 2008, Sims and DuVarney, 2007]. 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 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 [Zhan and Clark, 2005, Zhan and Clark, 2008]. Alternatively, bounded reachability analysis techniques [Brillout et al., 2009, He et al., 2011] can be used to detect mutants by checking k-step (bi)similarity [Kuehlmann and van Eijk, 2002] 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 in [Philipps et al., 2003] 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 aim to generate test suites as well as oracles from models that are considered to be correct. In reality, however, Simulink models might contain faults. Hence, in our work, we propose techniques to help testing complex Simulink models for which automated and precise test oracles are not available. Further, even though in Simulink, every variable is described using signals, unlike our work, none of the above techniques generate test inputs in terms of signals.

7.3.2 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 that some given formal properties holds for a model, 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 [Clarke et al., 1999]. It has a long history of application in software and hardware verification. It has been previously used to detect faults in Simulink models [Hamon, 2008, Barnat et al., 2012] by showing that a path leading to an error (e.g., an assertion or a run-time 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 [Miller, 2009, Meenakshi et al., 2006, Scaife et al., 2004, Araiza-Illan et al., 2015]. For example, Barnat et al. [Barnat et al., 2012] transform Simulink models into the DiVinE model checker's input language [Barnat et al., 2006] to verify Simulink models against some linear temporal logic properties. Whalen et al. [Whalen et al., 2007, Miller, 2009] first translate Simulink models into the LUSTRE formal specification language [Halbwachs et al., 1991] and then transform the LUSTRE specifications into the input languages of several well-known model checkers such as NuSMV [Cimatti et al., 2000] and the SAL tool suite [Bensalem et al., 2000]. Finally, Simulink Design Verifier [Hamon, 2008] translates and feeds Simulink models into a commercial SMT-based model checker, called Prover [Prover Technology, 2016]. Some alternative techniques [Holling et al., 2014, Balasubramanian et al., 2011, Venkatesh et al., 2012] translate Simulink models into code and use existing code analysis tools to detect faults in the models. 8Cage [Holling et al., 2014] *marks* the Simulink models in places where specific fault models [Pretschner et al., 2013] are detected. It then converts the models into c-code and directs KLEE [Cadar et al., 2008] toward those markers to generate test inputs that raise failures corresponding to the fault models. Polyglot [Balasubramanian et al., 2011] transforms Stateflow models into java code and uses JavaPathFinder [JPF, 2016] to analyze and check properties on the generated java code.

Statistical model checking. Model checking approaches, being exhaustive, suffer form the *state explosion problem* [Clarke et al., 1999]. 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 [Younes and Simmons, 2006, Legay et al., 2010]. 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 [Younes and Simmons, 2006, Zuliani et al., 2013]. Statistical model checking has been previously applied to Simulink models to estimate the probability that properties specified in temporal logic hold over models [Zuliani et al., 2013, Clarke and Zuliani, 2011]. Note that in contrast to model checking, statistical model checking does not guarantee to produce exact results (i.e., true/false results) and only estimate 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 [Reactive Systems Inc., 2016b, S-Taliro, 2016]. Reactis validator [Reactive Systems Inc., 2016b, Cleaveland et al., 2008] adapts such an approach by running the coverage-adequate test suites generated by Reactis tester [Reactive Systems Inc., 2016a] and tracking whether any assertions are violated by the test cases. S-Taliro toolbox [S-Taliro, 2016, Annpureddy et al., 2011, Zutshi et al., 2015] has usage modes that rely on meta-heuristic search to falsify Metric Temporal Logic properties [Koymans, 1990] instrumented into Simulink models. Note that though these techniques look for possibility of assertion violations, they provide no guarantee to uncover all assertion violations.

Model checking techniques are applicable only to code-generation models with discrete behaviors. Further, they inherit the limitations of constraint/SAT/SMT solvers in handling floating point

and non-linear math operations. In our work, we generate test inputs as continuous signals and use black-box output-based algorithms which are applicable to both simulation Simulink models with mixed discrete-continuous behaviors as well as code generation models. In addition, all the model verification and testing approaches discussed above rely on the presence of formal specifications that characterize the expected behaviors of Simulink models and from which assertions can be derived. Formal specifications are expensive and may not be available in practice. Furthermore, existing formal specifications typically describe discrete system properties. Capturing continuous dynamic aspects of Simulink models using formal specifications is still an open problem [Pretschner et al., 2007a, Heimdahl et al., 2013]. Moreover, no completeness criteria exist for testing based on model properties. As a result, even in the presence of some model properties, engineers still need to check output signals to detect more failures. In fact, in practice it is likely and common that engineers assess system outputs manually to detect failures. Hence, in our work, we do not rely on the presence of formal specifications and assume that test oracles are manual. We generate small test suites with high fault revealing ability that effectively reduce the manual oracle cost. We also provide a test prioritization algorithm to further lower the manual oracle cost. Finally, we note that, in contrast to our work presented in this article, none of the papers discussed above include a systematic empirical evaluation of the proposed approach involving industrial Simulink models nor do they provide a systematic comparison with alternative techniques.

7.4 Source Code 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 [Fraser and Arcuri, 2013] that was proposed for testing software code. This approach evolves an entire test suite, instead of individual test cases, with the aim of covering all structural coverage goals at the same time. Our algorithm, instead, attempts to diversify test outputs by taking into account all the signal features at the same time. (2) *Output uniqueness/diversity*: The notion of output diversity in our work is inspired by the output uniqueness criterion [Alshahwan and Harman, 2012, Alshahwan and Harman, 2014]. As noted in [Alshahwan and Harman, 2014], effectiveness of this criterion depends on the definition of *output difference* and differs from one context to another. While in [Alshahwan and Harman, 2012, Alshahwan and Harman, 2014], output differences are described in terms of the textual, visual or structural aspects of HTML code, in our work, output differences are characterized by signal shape features.

7.5 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 [Qi Luo and Poshyvanyk, 2016, Yoo and Harman, 2012, Rothermel et al., 2001]. 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 [Qi Luo and Poshyvanyk, 2016]. In our prioritization algorithm in chapter 6, 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 technique typically rank test cases by relying on on *total* or *additional* structural coverage achieved by individual test cases [Qi Luo and Poshyvanyk, 2016, Yoo and Harman, 2012]. To unify the total and additional coverage-based strategies, Zhang et al. [Zhang et al., 2013] proposed 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. To compute the probability that a test case reveals a fault, we take into account the output diversity of the generated test suites in addition to structural coverage achieved by individual test cases. As a result, test cases with slightly lower coverage but coming from test suites with higher output diversity are ranked higher when compared with existing approaches. Further, in our algorithm the notion of dynamic test coverage is specific to a Simulink model output. That is, for a given model output and a given test case, the test coverage includes the model nodes that are connected to that output by control/data dependencies and are executed by that test case.

Chapter 8

Conclusions and Future Work

In this chapter we summarize the contributions of this dissertation and discuss some perspectives on potential future work in this area.

8.1 Summary

In this dissertation, we proposed several test generation techniques aimed at addressing the challenges of testing Simulink/Stateflow controllers with mixed discrete-continuous behaviors. The work presented in this dissertation has been done in collaboration with Delphi Automotive Systems [Delphi, 2016], a world leading part supplier company to the automotive industry, based in Luxembourg. Simulink models often have mixed discrete-continuous behaviors and their correct behavior crucially depends on time. Their inputs and outputs are signals rather than discrete values. Further, Simulink models are required to operate satisfactorily for a large variety of hardware configurations. In addition, developing test oracles for Simulink models is challenging, particularly for requirements capturing their continuous aspects.

In our work, we relied on discrete-continuous output signals of Simulink models to generate test cases with high fault revealing ability. Our algorithms are black-box and therefore applicable to Simulink/Stateflow models in their entirety. Further, we do not rely on the presence of formal specifications to automate test oracles. We presented two sets of test generation algorithms for closed-loop and open-loop controllers implemented in Simulink. For closed-loop controllers, test oracles can be formalized and automated relying on the controlled system's feedback. We identified a set of common requirements for closed-loop controllers, and used search to find the worst-case test scenarios of the controller with respect to each requirement. For open-loop controllers, the feedback is not available and, hence, test oracles are manual. As a result, we provide test generation algorithms to produce small effective test suites with high fault revealing ability. Our test generation algorithms either attempt to maximize the likelihood of specific failure patterns in the output signals or diversity between output signals of test inputs within a test suite. In order to lower the manual oracle cost, we further proposed a test prioritization algorithm to automatically rank test cases generated by our test generation algorithm.

Chapter three presented our automated test generation approach and our MiL testing tool (CoCoTest) for closed-loop controllers. We identified and formalized a set of common requirements for

closed-loop continuous controllers. Our proposed technique relies on a combination of explorative and exploitative search algorithms, which aim at finding worst-case scenarios in the input space with respect to the controller requirements. We evaluated our approach by applying it to an automotive air compressor module and to a publicly available DC motor controller. Our experiments showed that our approach automatically generates several worst-case scenarios, which can be used for testing purposes, that had not been previously found by manual testing based on domain expertise.

Chapter four described our approach for MiL testing of closed-loop controllers in large configuration spaces with respect to controller requirements. To scale search to large multi-dimensional, we combined techniques for dimensionality reduction and supervised learning to build surrogate models that accurately predict simulation results without resorting to simulation in many cases. Our evaluation shows that our approach is able to identify critical violations of the controller requirements that had neither been found by our algorithm in chapter three nor by manual testing. Further, we showed that combining search with surrogate modeling increases the search speed and enables search to compute more critical requirements violations than what could be detected by the search without surrogate modeling.

Chapter five presented our test generation algorithms focused on open-loop controllers that aim at providing minimal test suites with high fault revealing power. We proposed and evaluated six test generation algorithms for discrete-continuous Stateflows: three output-based, two coverage-based, and one input-based. 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 larger test suites, 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, output stability and output discontinuity test generation algorithms had very high fault revealing rates, even with small test suites, for instability and discontinuity failures, respectively. For the other failures, output diversity outperformed the other algorithms in finding faults for larger test suite sizes and, further, its fault detection rate kept improving at a faster rate than the others when increasing the test suite size.

In Chapter six we distinguished Simulink simulation and code generation models and illustrated differences in their behaviors using examples. We described the challenges corresponding to the existing testing approaches that are only applicable to code generation Simulink models, and proposed a test suite generation algorithm for both kinds of Simulink models based on our notion of feature-based output diversity. Our algorithm adapts the whole test suite generation approach and uses meta-heuristic search to produce test outputs exhibiting a diverse set of signal features. We further provided a test prioritization algorithm to autonomically rank the generated tests and lower manual oracle cost. Our evaluation was performed using two industrial and two public domain Simulink models and shows that (1) Our approach significantly outperforms random test generation and coverage-based test generation. (2) Our algorithm when used with the feature-based notion of output diversity generates higher fault revealing rates compared to when output diversity is measured based on vector-based output diversity. (3) Our prioritization algorithm outperforms random and coverage-based prioritization. (4) Our approach is able to reveal significantly more faults compared to Simulink Design Verifier. Further, it subsumes SLDV, as it is able to find the faults identified by SLDV with a 100% probability. Using our SimCoTest tool, which implements our test generation and prioritization algorithms, we were able to generate test cases that identified two real faults in representative SL/SF

models from our industry partner, Delphi. These faults had not been previously found by manual testing based on domain expertise or other Simulink testing tools used at Delphi.

8.2 Future Work

In this dissertation we focused on testing Simulink controllers developed in the automotive domain. To better assess the applicability and effectiveness of our approaches, future studies need to be made considering testing software controllers (1) developed in other CPS domains, e.g., avionics, communications, and medical systems, or (2) implemented in other system modeling notations, e.g., Labview from National Instruments [Instruments, 2016], SCADE Suite from ESTEREL Technologies [Technologies, 2016], or open-source alternatives such as SciLab [Scilab, 2016]. As an example, our input test generation strategy is geared towards the specific needs of the automotive domain where the dynamic behavior of the system is tested using sequences of step signals as input. This is sufficient for capturing automotive environment events, which are largely aperiodic, e.g., driver's commands. However, for other domains our test generation strategy may need to be adapted to effectively capture periodic signals with high frequency.

Developing coverage-adequate test inputs for Simulink models with time-continuous behaviours, and floating-point and non-linear operations remains an open problem. The existing coverage-based test generation tools are not compatible with time-continuous Simulink models and fall short when the models contain floating-point and non-linear computations. Further research is necessary to evaluate the applicability of search-based techniques to generate coverage-adequate test inputs for Simulink/Stateflow models in their entirety. We may also draw on reachability analysis techniques for hybrid automata [Frehse et al., 2011] and verification techniques studied in the control theory domain [Abate et al., 2010] to generate such coverage-adequate test inputs. This can further be combined with our output diversity algorithm, relying on multi-objective search to generate coverage-adequate test inputs with diverse output shapes.

Matlab simulator has several settings including the simulation time, the solver type (fixed-step or variable-step), the model solver (ode45 solver, ode23 solver, discrete solver, ...), and so on. In our work, we used the discrete solver for code-generation models and the ode45 solver for simulation models, as ode45 solver is the most accurate continuous solver in the version of Matlab used by Delphi. As for the simulation time and other simulation settings, we used the values recommended by Delphi engineers. In future, we plan to conduct experiments to examine the sensitivity of our test generation results to simulation settings.

In Simulink model testing, as in the more general case of automated testing, test oracle automation is the most difficult challenge. Formal specifications are expensive and rare, and not all faults lead to run-time errors. Moreover, engineers often bypass runtime errors in Simulink models by saturating the block output when overflow or division by zero errors occur. In our work, we focused on producing small test suites with high fault revealing ability when test oracles are manual. Further research is necessary to enhance the effectiveness of the test suites generated by our approach. For example, identifying more failure patterns, in addition to instability and discontinuity, in discrete-continuous outputs of Simulink models can help increase the fault revealing ability of our test generation algorithms. Further, the applicability of metamorphic testing can be studied in the context of Simulink testing. Metamorphic testing verifies the correctness of metamorphic relations between test cases to

automatically detect faults in the absence of automated test oracles.

Finally, we note that we plan to explore the possibility of building a commercial Simulink testing tool based on our approach. Delphi has already shown interest to use a commercial version of our SimCoTest tool. We were also awarded a Proof-of-Concept (POC) grant from National Research Fund Luxembourg (FNR) [FNR, 2016], which provides financial support for us within the next 16 months to build and commercialize the tool.

List of Papers

Published papers included in this dissertation:

- Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckman. "Automated Model-in-the-Loop Testing of Continuous Controllers using Search." In *Proceedings of the 28th Symposium on Search-Based Software Engineering (SSBSE 2013)*, pp. 141-157, 2013.
- Reza Matinnejad, Shiva Nejati, Lionel C. Briand, Thomas Bruckmann, and Claude Poull. "Search-Based Automated Testing of Continuous Controllers: Framework, Tool Support, and Case Studies" In *Information and Software Technology (IST)* Journal, pp. 705-722, 2015.
- Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. "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 (ASE 2014)*, pp. 163-174, 2014 (Received ACM/SIGSOFT distinguished paper award).
- Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. "CoCoTest: a tool for model-in-the-loop testing of continuous controllers" In *Proceedings of the 29th ACM/IEEE international conference on Automated Software Engineering (ASE 2014 - Tool Track)*, pp. 163-174, 2014.
- Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. "Effective Test Suites for Mixed Discrete-Continuous Stateflow Controllers" In *Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015)*, pp. 84-95, 2015 (Received ACM/SIGSOFT distinguished paper award).
- Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. "Automated Test Suite Generation for Time-Continuous Simulink Models" In *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*, pp. 595-606, 2016.
- Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. "SimCoTest: A Test Suite Generation Tool for Simulink/Stateflow Controllers" In *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016 - Tool Track)*, pp. 585-588, 2016.

Unpublished papers included in this dissertation:

- Reza Matinnejad, Shiva Nejati, Lionel C. Briand, and Thomas Bruckmann. "Test Generation and Test Prioritization for Simulink Models with Dynamic Behavior" submitted to *Transactions*

on Software Engineering (TSE) Journal.

Bibliography

[Matinnejad, 2016] Matinnejad (2016). Continuous Controller Tester (CoCoTest). https://sites.google.com/site/cocotesttool/.

[Abate et al., 2010] Abate, A., Katoen, J.-P., Lygeros, J., and Prandini, M. (2010). Approximate model checking of stochastic hybrid systems. *European Journal of Control*, 16(6):624–641.

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

[Alshahwan and Harman, 2014] Alshahwan, N. and Harman, M. (2014). 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.

[Annpureddy et al., 2011] Annpureddy, Y., Liu, C., Fainekos, G., and Sankaranarayanan, S. (2011). *S-taliro: A tool for temporal logic falsification for hybrid systems*. Springer.

[Araiza-Illan et al., 2015] Araiza-Illan, D., Eder, K., and Richards, A. (2015). 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.

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

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

[Arcuri and Briand, 2012] Arcuri, A. and Briand, L. (2012). A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*.

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

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

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

[Barnat et al., 2006] Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., and Šimeček, P. (2006). Divine – a tool for distributed verification. In *International Conference on Computer Aided Verification*, pages 278–281. Springer.

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

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

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

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

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

[Briand et al., 2016] Briand, L., Nejati, S., Sabetzadeh, M., and Bianculli, D. (2016). 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.

[Brillout et al., 2009] Brillout, A., He, N., Mazzucchi, M., Kroening, D., Purandare, M., Rümmer, P., and Weissenbacher, G. (2009). Mutation-based test case generation for simulink models. In *Formal Methods for Components and Objects*, pages 208–227. Springer.

[Caballero and Grossmann, 2008] Caballero, J. A. and Grossmann, I. E. (2008). An algorithm for the use of surrogate models in modular flowsheet optimization. *AIChE journal*, 54(10):2633–2650.

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

[Campolongo et al., 2007] Campolongo, F., Cariboni, J., and Saltelli, A. (2007). An effective screening design for sensitivity analysis of large models. *Environmental modelling & software*, 22(10):1509–1518.

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

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

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

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

[Clarke et al., 1999] Clarke, E., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

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

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

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

[Cormen et al., 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to algorithms*, volume 6. MIT press Cambridge.

[Delphi, 2016] Delphi (2016). Delphi Automotive Systems . http://www.delphi.com. [Online; accessed 17-Jul-2016].

[Douguet, 2010] Douguet, D. (2010). e-LEA3D: a computational-aided drug design web server. *Nucleic acids research*, 38(suppl 2):W615–W621.

[FNR, 2016] FNR (2016). National Research Fund Luxembourg. `http://fnr.lu`. [Online; accessed 17-Jul-2016].

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

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

[Frehse et al., 2011] Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., and Maler, O. (2011). SpaceEx: Scalable verification of hybrid systems. In *International Conference on Computer Aided Verification*, pages 379–395. Springer.

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

[Gold et al., 1969] Gold, B., Stockham, T. G., Oppenheim, A. V., and Rader, C. M. (1969). Digital processing of signals.

[Grinstein et al., 2001] Grinstein, G., Trutschl, M., and Cvek, U. (2001). High-dimensional visualizations. In *Proceedings of the 7th Data Mining Conference (KDD)*, pages 7–19.

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

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

[Harman et al., 2007] Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., and Wegener, J. (2007). The impact of input domain reduction on search-based test data generation. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 155–164. ACM.

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

[Heimdahl et al., 2013] Heimdahl, M. P., Duan, L., Murugesan, A., and Rayadurgam, S. (2013). 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.

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

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

[Instruments, 2016] Instruments, N. (2016). LabView. `http://www.ni.com/labview/`. [Online; accessed 17-Jul-2016].

[Jin, 2011] Jin, Y. (2011). Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 1(2):61–70.

[JPF, 2016] JPF (2016). Java pathfinder tool-set. `http://babelfish.arc.nasa.gov/trac/jpf`. [Online; accessed 17-Jul-2016].

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

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

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

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

[Ledin, 2001] Ledin, J. (2001). *Simulation engineering*. CMP books The Netherlands.

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

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

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

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

[Liu et al., 2015] Liu, B., Lucia, L., Nejati, S., and Briand, L. (2015). Simulink Fault Localization: an Iterative Statistical Debugging Approach. Technical report, University of Luxembourg.

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

[Mathworks, 2016] Mathworks (2016). Robust Toolbox. `http://nl.mathworks.com/products/robust/r`. [Online; last accessed 19-Jul-2016].

[Matinnejad, 2016] Matinnejad (2016). Simulink Controller Tester (SimCoTest). `https://sites.google.com/site/simcotesttool/`.

[Matinnejad, 2016] Matinnejad, R. (2016). The modified version of GCS Stateflow. `https://drive.google.com/file/d/0B104wPnuxJVhSU44WFlTVE84bmM/view?usp=sharing`. [Online; accessed 10-March-2016].

[Matinnejad et al., 2014a] Matinnejad, R., Nejati, S., Briand, L., and Brcukmann, T. (2014a). Cocotest: a tool for model-in-the-loop testing of continuous controllers. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 855–858. ACM.

[Matinnejad et al., 2014b] Matinnejad, R., Nejati, S., Briand, L., and Brcukmann, T. (2014b). 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.

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

[Matinnejad et al., 2016a] Matinnejad, R., Nejati, S., Briand, L., and Bruckmann, T. (2016a). Automated test suite generation for time-continuous simulink models. In *ICSE 2016*.

[Matinnejad et al., 2016b] Matinnejad, R., Nejati, S., Briand, L., and Bruckmann, T. (2016b). Simcotest: A test suite generation tool for simulink/stateflow controllers. In *Submitted to ICSE 2016*.

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

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

[Matinnejad, Reza, 2016] Matinnejad, Reza (2016). The paper extra resources (technical reports and the models). `https://sites.google.com/site/myicseresources/`.

[Mazzolini et al., 2010] Mazzolini, M., Brusaferri, A., and Carpanzano, E. (2010). Model-checking based verification approach for advanced industrial automation solutions. In *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pages 1–8. IEEE.

[McMinn et al., 2012] McMinn, P., Harman, M., Lakhotia, K., Hassoun, Y., and Wegener, J. (2012). Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation. *Software Engineering, IEEE Transactions on*, 38(2):453–477.

[McMinn et al., 2010] McMinn, P., Stevenson, M., and Harman, M. (2010). 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.

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

[Miller, 2009] Miller, S. P. (2009). 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.

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

[Morris, 1991] Morris, M. D. (1991). Factorial sampling plans for preliminary computational experiments. *Technometrics*, 33(2):161–174.

[Namin and Andrews, 2009] Namin, A. S. and Andrews, J. H. (2009). 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.

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

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

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

[Ong et al., 2003] Ong, Y. S., Nair, P. B., and Keane, A. J. (2003). Evolutionary optimization of computationally expensive problems via surrogate modeling. *AIAA journal*, 41(4):687–696.

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

[Park and Stefanski, 1998] Park, H. and Stefanski, L. (1998). Relative-error prediction. *Statistics & probability letters*, 40(3):227–236.

[Peranandam et al., 2012] Peranandam, P., Raviram, S., Satpathy, M., Yeolekar, A., Gadkari, A., and Ramesh, S. (2012). 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.

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

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

[Pretschner et al., 2007a] Pretschner, A., Broy, M., Krüger, I., and Stauner, T. (2007a). Software engineering for automotive systems: A roadmap. In *FOSE*, pages 55–71.

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

[Pretschner et al., 2007b] Pretschner, A., Salzmann, C., Schätz, B., and Stauner, T., editors (2007b). *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, Companion Volume*. IEEE Computer Society.

[Prover Technology, 2016] Prover Technology (2016). Prover Plug-In Software. `http://www.prover.com`. [Online; accessed 17-Jul-2016].

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

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

[Reactive Systems Inc., 2016a] Reactive Systems Inc. (2016a). Reactis Tester. `http://www.reactive-systems.com/simulink-testing-validation.html`. [Online; accessed 17-Jul-2016].

[Reactive Systems Inc., 2016b] Reactive Systems Inc. (2016b). Reactis Validator. `http://www.reactive-systems.com/simulink-testing-validation.html`. [Online; accessed 17-Jul-2016].

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

[S-Taliro, 2016] S-Taliro (2016). S-Taliro Toolbox. `https://sites.google.com/a/asu.edu/s-taliro/s-taliro`. [Online; accessed 17-Jul-2016].

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

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

[Scaife et al., 2004] Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., and Maraninchi, F. (2004). 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.

[Scilab, 2016] Scilab (2016). SciLab. `http://www.scilab.org`. [Online; accessed 23-Fev-2016].

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

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

[Technologies, 2016] Technologies, E. (2016). SCADE. `http://www.esterel-technologies.com/products/scade-suite/`. [Online; accessed 17-Jul-2016].

[The MathWorks Inc., 2003a] The MathWorks Inc. (2003a). Matlab quasi random numbers. `http://www.mathworks.nl/help/stats/generating-quasi-random-numbers.html`. [Online; accessed 17-Jul-2016].

[The MathWorks Inc., 2003b] The MathWorks Inc. (2003b). Simulink. `http://www.mathworks.nl/products/simulink`. [Online; accessed 17-Jul-2016].

[The MathWorks Inc., 2003c] The MathWorks Inc. (2003c). Stepwise Linear Regression Model. `http://www.mathworks.nl/help/stats/stepwiselm.html`. [Online; accessed 30-Mar-2016].

[The MathWorks Inc., 2009] The MathWorks Inc. (2009). DC Motor Simulink Model. `http://www.mathworks.com/matlabcentral/fileexchange/11587-dc-motor-model-simulink`. [Online; accessed 17-Jul-2016].

[The MathWorks Inc., 2011] The MathWorks Inc. (2011). Embedded Coder. `http://www.mathworks.nl/products/embedded-coder/`. [Online; accessed 17-Jul-2016.

[The MathWorks Inc., 2016a] The MathWorks Inc. (2016a). Building a Clutch Lock-Up Model. `http://nl.mathworks.com/help/simulink/examples/building-a-clutch-lock-up-model.html?refresh=true`. [Online; accessed 17-Jul-2016].

[The MathWorks Inc., 2016b] The MathWorks Inc. (2016b). C Code Generation from Simulink. `http://nl.mathworks.com/help/dsp/ug/generate-code-from-simulink.html`. [Online; accessed 17-Jul-2016].

[The MathWorks Inc., 2016c] The MathWorks Inc. (2016c). Call MATLAB Function from C# Client. `http://mathworks.com/help/matlab/matlab_external/call-matlab-function-from-c-client.html`.

[The MathWorks Inc., 2016d] The MathWorks Inc. (2016d). 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-2016].

[The MathWorks Inc., 2016e] The MathWorks Inc. (2016e). 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-Jul-2016].

[The MathWorks Inc., 2016f] The MathWorks Inc. (2016f). Simulink Deign Verifier Cruise Control Test Generation. http://nl.mathworks.com/help/sldv/examples/extending-an-existing-test-suite.html?prodcode=DV&language=en. [Online; accessed 17-Jul-2016].

[The MathWorks Inc., 2016g] The MathWorks Inc. (2016g). Simulink Design Verifier. http://nl.mathworks.com/products/sldesignverifier/?refresh=true. [Online; accessed 6-May-2016].

[The MathWorks Inc., 2016h] The MathWorks Inc. (2016h). Stateflow. http://www.mathworks.nl/products/stateflow. [Online; accessed 23-Fev-2016].

[The MathWorks Inc., 2016i] The MathWorks Inc. (2016i). Stateflow Model Examples. http://nl.mathworks.com/products/stateflow/model-examples.html. [Online; accessed 23-Fev-2016].

[The MathWorks Inc., 2016j] The MathWorks Inc. (2016j). Types of Model Coverage. http://nl.mathworks.com/help/slvnv/ug/types-of-model-coverage.html. [Online; accessed 17-Jul-2016].

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

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

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

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

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

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

[Whalen et al., 2007] Whalen, M., Cofer, D., Miller, S., Krogh, B. H., and Storm, W. (2007). 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.

[Wikipedia., 2016] Wikipedia. (2016). Voltage Spike. http://en.wikipedia.org/wiki/Voltage_spike. [Online; accessed 23-Fev-2016].

[Wilmes and Windisch, 2010] Wilmes, B. and Windisch, A. (2010). Considering signal constraints in search-based testing of continuous systems. In *International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pages 202–211. IEEE.

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

[Windisch, 2010] Windisch, A. (2010). Search-based test data generation from Stateflow statecharts. In *Genetic and Evolutionary Computation (GECCO)*, pages 1349–1356. ACM.

[Windisch and Al Moubayed, 2009a] Windisch, A. and Al Moubayed, N. (2009a). 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.

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

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

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

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

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

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

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

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

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

[Zhan and Clark, 2008] Zhan, Y. and Clark, J. A. (2008). A search-based framework for automatic testing of Matlab Simulink models. *Journal of Systems and Software*, 81(2):262–285.

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

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

[Zutshi et al., 2015] Zutshi, A., Sankaranarayanan, S., Deshmukh, J. V., Kapinski, J., and Jin, X. (2015). 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.