Automated Steering of Model-Based Test Oracles to Admit Real Program Behaviors

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Gregory Gay

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Mats Heimdahl, Ph.D.

May, 2015

Acknowledgements

Many, many people were pivitol in the creation of this thesis, and I would like to take a moment to thank them (and numerous other people that I have inevitably forgotten to mention). I would like to thank Mats Heimdahl—mv advisor—whose support and guidance made this entire project possible. I'd like to thank him in equal measure for telling me when I made a bone-headed mistake and for trusting me to find the method in my madness. I could not have asked for a better mentor. Thanks also go to Mike Whalen and Sanjai Rayadurgam, who devoted their time, experience, and expertise to providing feedback on my ideas. I would also like to thank Eric Van Wyk and Galin Jones for serving on my thesis committee and providing excellent advice over the past few years.

Research never takes place in a vacuum, and I would like to thank Matt Staats, Ian De Silva, Lian Duan, Hung Pham, Anitha Murugesan, Dongjiang You, Jason Biatek, Kevin Wendt, and the rest of the Crisys crew for being both great coworkers and great friends. I am grateful to Tim Menzies, Mark Harman, Phil McMinn, and Misty Davies for providing opportunities to collaborate on research and take a leading role in the software engineering research community.

Last—but certainly not least—I would like to thank my wife, my friends, and my family for their neverending work in keeping me sane. Michelle, you're a saint.

This work has been partially supported by NSF grants CNS-0931931 and CNS-1035715, and an NSF Graduate Research Fellowship.

Abstract

The *test oracle*—a judge of the correctness of the system under test (SUT)—is a major component of the testing process. Specifying test oracles is challenging for some domains, such as real-time embedded systems, where small changes in timing or sensory input may cause large behavioral differences. Models of such systems, often built for analysis and simulation, are appealing for reuse as test oracles. These models, however, typically represent an *idealized* system, abstracting away certain issues such as non-deterministic timing behavior and sensor noise. Thus, even with the same inputs, the model's behavior may fail to match an acceptable behavior of the SUT, leading to many false positives reported by the test oracle.

We propose an automated *steering* framework that can adjust the behavior of the model to better match the behavior of the SUT to reduce the rate of false positives. This *model steering* is limited by a set of constraints (defining the differences in behavior that are acceptable) and is based on a search process attempting to minimize a dissimilarity metric. This framework allows non-deterministic, but bounded, behavioral differences, while preventing future mismatches by guiding the oracle—within limits—to match the execution of the SUT. Results show that steering significantly increases SUT-oracle conformance with minimal masking of real faults and, thus, has significant potential for reducing false positives and, consequently, testing and debugging costs while improving the quality of the testing process.

Contents

List of Tables

List of Figures

Automated Steering of Model-Based Test Oracles to Admit Real Program Behaviors

Gregory Gay

May 5, 2015

Chapter 1

Introduction

When running a suite of tests, the *test oracle* is the judge that determines the correctness of the execution of a given system under test (SUT). Over the past decades, researchers have made remarkable improvements in automatically generating effective test stimuli [1], but it remains difficult to build an automated method of checking behavioral correctness. Despite increased attention, the *test oracle problem* [2]—the set of challenges related to the construction of efficient and robust oracles—remains a major problem in many domains.

One such domain is that of real-time process control systems—embedded systems that interact with physical processes such as implanted medical devices or power management systems. Systems in this domain are particularly challenging since their behavior depends not only on the values of inputs and outputs, but also on their time of occurrence [3]. In addition, minor behavioral distinctions may have significant consequences [4]. When executing the software on an embedded hardware platform, several sources of non-determinism, such as input processing delays, execution time fluctuation, and hardware inaccuracy, can result in the SUT non-deterministically exhibiting varying—but acceptable—behaviors for the same test case.

Behavioral models [5], typically expressed as state-transition systems, represent the system specifications by prescribing the behavior (the system state) to be exhibited in response to given input. Common modeling tools in this category are Stateflow [6], Statemate [7], and Rhapsody [8]. Models built using these tools are used for many purposes in industrial software development, particularly during requirements and specification analysis. Behavior modeling is common for the analysis of embedded and real-time systems, as the requirements for such systems are naturally *stateful*—their outcome depends strongly on the current system mode and

a number of additional factors both internal and external to the system. Because such models can be "executed", a potential solution to the need for a test oracle is to execute the same tests against both the model and the SUT and compare the resulting behaviors.

These models, however, provide an abstract view of the system that typically simplifies the actual conditions in the execution environment. For example, communication delays, processing delays, and sensor and actuator inaccuracies may be omitted. Therefore, on a real hardware platform, the SUT may exhibit behavior that is acceptable with respect to the system requirements, but differs from what the model prescribes for a given input; the system under test is "close enough" to the behavior described by the model. Over time, these differences can build to the point where the execution paths of the model and the SUT diverge enough to flag the test as a "failure," even if the system is still operating within the boundaries set by the requirements. In a rigorous testing effort, this may lead to tens of thousands of false reports of test failures that have to be inspected and dismissed—a costly process.

One simple potential solution would be to filter the test results on a step-by-step basis—checking the state of the SUT against a set of constraints and overriding a failing verdict if those constraints are met. However, filter-based approaches are *inflexible*. While a filter may be able to handle isolated non-conformance between the SUT and the oracle model, it will likely fail to account for behavioral divergence that builds over time, growing with each round of input.

Instead, we take inspiration for addressing this model-SUT mismatch problem from *program steering*, the process of adjusting the execution of live programs in order to improve performance, stability, or correctness [9]. We hypothesize that behavioral models can be adapted for use as oracles for real-time systems through the use of *steering actions* that override the current execution of the model [10, 11]. By comparing the state of the model-based oracle (MBO) with that of the SUT following an output event, we can guide the model to match the state of the SUT through a search process that seeks a steering action that transitions the model to a reachable state that obeys a set of user-specified constraints and general steering policies and that minimizes a dissimilarity metric. For example, if mismatches occur due to sensor inaccuracy in the real system, then we could incorporate the sensor's expected accuracy range as a constraint and try different values within that range. The steering process, by using a search algorithm and the provided constraints, widens the set of behaviors that the model will accept while retaining the power of the oracle as an arbiter on test results. Unlike a filter, steering is *adaptable*,

adjusting the live execution of the model—within the space of legal behaviors—to match the execution of the SUT. The result of steering is a widening of the behaviors accepted by the oracle, thus compensating for allowable non-determinism, without unacceptably impairing the ability of the model-based oracle to correctly judge the behavior of the SUT.

We present an automated framework for comparing and steering the model-based oracle with respect to the SUT. In this dissertation, we include a detailed discussion of the implementation and implications of oracle steering, and assess the capabilities of the steering framework on two systems with complex, time-based behaviors—the control software of a patient controlled analgesia pump (a medical infusion pump) and a pacemaker.

Case study results indicate that steering improves the accuracy of the final oracle verdicts—outperforming both default testing practice and step-wise filtering. Oracle steering successfully accounts for within-tolerance behavioral differences between the model-based oracle and the SUT—eliminating a large number of spurious "failure" verdicts—with minimal masking of real faults. By pointing the developer towards behavior differences more likely to be indicative of real faults, this approach has the potential to reduce testing effort and reduce development cost.

As the choice of constraints intuitively has an impact on the ability of steering to account for allowable non-determinism, we have also examined the performance of steering when several different sets of constraints are used. These constraints vary in the level of freedom given to the steering algorithm to adjust the oracle verdict, ranging from a scenario where no constraints are used at all to one where the steering algorithm is given very little ability to adjust the state of the system. As a result, we confirmed the importance of the choice of constraints in determining the effectiveness of steering. Relatively strict, well-considered constraints strike the best balance between enabling steering to focus developers and preventing steering from masking faults. As constraints are loosened, steering may be able to account for more and more acceptable deviations, but at the cost of also masking more faults. Alternatively, loose constraints may also impair steering from performing its job by allowing the search process the freedom to choose a suboptimal steering action.

Given the importance of selecting constraints, there is a lot of pressure on developers to choose the correct options. Fortunately—even if developers are unsure of what variables to set constraints on—as long as they can classify the outcome of a set of tests, a set of constraints can be automatically learned through a process known as *treatment learning*. Given a set of classified test cases, we can execute steering with no constraints, extract information on the

induced changes, and apply a treatment learner to discover what changes were imposed by steering when it acted correctly. We can then apply these learned constraints when steering for a broader set of test executions. For our case examples, the derived constraint sets were small, strict, and able to successfully steer the model with only minimal tuning.

We have found that steering is able to automatically adjust the execution of the oracle to handle non-deterministic, but acceptable, behavioral divergence without covering up most fault-indicative behaviors. We, therefore, recommend the use of steering as a tool for focusing and streamlining the testing process.

1.1 Statement of Thesis

Industrial developers seek to use model-based oracles when testing systems, but abstraction issues and unexpected non-determinism make the achievement of that goal challenging. Yet, we also know from anecdotal experience that some developers already conduct an ad-hoc steering of model-based oracles to work around such issues. These two points provide motivation for the **objective of this dissertation**, which—as addressed above—is to *address the challenges of using behavioral models as test oracles through the use of general, controlled steering procedures*. The **central hypothesis** of this thesis is that oracle steering, if conducted in a controlled manner, is an effective and low-cost method of reusing behavioral models as an information source for test oracles.

To that end, we have proposed steering methods and an automated framework to control the execution of a behavioral model. By comparing the behaviors of both the oracle and system under test following a cycle of input and output, we can backtrack and choose a valid state in the oracle that more closely matches the current state of the SUT. We believe that such an approach, as long as our hypothesis is true, will allow us to compensate for the non-determinism observed in real-world execution, while retaining the power of the oracle as an arbiter on behavior.

1.2 Contributions of this Thesis

Our **long-range goal** is to improve the practice of software testing. The test oracle is not a well-understood artifact, and improvements in the construction, composition, and use of such oracles will lead to improvements in testing practices and a lessening of the burden on the human tester.

Of the paired problems of test oracle construction and test input construction, substantially more research has been devoted to the latter. While interest in this area is increasing [2], the test oracle problem remains a major challenge for many domains. The research presented in this dissertation does not solve the test oracle problem—for instance, we do not offer methods of automatically creating oracles—but, it does represent a major step forward by enabling the use of behavioral models as test oracles for real-world embedded systems. Embedded systems represent a large portion of the software produced in industrialized societies, and such models could lead to large improvements in the cost and effectiveness of testing efforts.

The **intellectual merit** of the presented research lies in its new and novel contributions to the body of literature on software testing and test oracles. This work is significant because it (a) addresses a real-world testing challenge that has not been solved in a satisfactory manner, and (b), our solution can potentially bring about major improvements to the quality and cost of the software testing process for embedded and real-time systems. Such systems are becoming more and more common, and the presented research enables the use of models as oracles during their testing. We have addressed a real industrial problem. Developers already build these models, and their reuse as oracles is desired, but issues with abstraction and non-determinism have made the achievement of this goal daunting. Adoption of our techniques by industrial developers could result in lower testing costs and reduced development effort.

This dissertation has made the following contributions to the software testing literature:

1. **We have developed steering procedures for model-based test oracles.** We have devised an automated search process for steering the test oracle that accounts for acceptable deviations from the predicted system behavior by backtracking the execution of the model-based oracle, constraining the list of reachable states through a set of user-specified constraints, and selecting a new candidate state using a numeric dissimilarity metric.

2. **We have developed output comparison procedures to quantify behavioral deviations.** The search for a candidate search solution is guided by a dissimilarity metric that compares the state of the model-based oracle to the state of the system under test. We have examined the use of multiple metrics and their role in determining the effectiveness of the search procedure.

3. **We have created an automated method to learn steering constraints.** The user-specified constraints on the steering process have an important role in controlling the

search process. We have used treatment learning, a data mining technique focused on crafting small sets of rules that select for particular types of data, to extract candidate steering constraints. Experiments indicate that this process is effective in producing powerful constraints.

4. **We detail the implementation of these steering, comparison, and learning procedures as part of an automated testing framework.** This framework is able to make effective use of behavioral models as a source of oracle information while addressing the complications presented in the real-world use of such models.

5. **We empirically evaluate the use of oracle steering on medical devices.** We have conducted rigorous experiments to ascertain the impact of the use of oracle steering to compensate for non-determinism in the testing of real-time systems—including an infusion pump and a pacemaker. In addition to determining the safety and effectiveness of our steering procedures, we examine the impact on the steering process from different comparison procedures and differing strictness levels in the search constraints.

The **outcomes** of this research are (1) an automated framework that can be used to execute the proposed techniques on actual systems, and (2), authoritative studies of the capabilities, efficacy, and risks of these procedures as applied to industrial-quality software in the real-time system domain.

1.3 Publications from this Thesis

The following publications have resulted from the research reported in this thesis:

- Gregory Gay, Sanjai Rayadurgam, and Mats P. E. Heimdahl. 2014. *Steering Model-Based Oracles to Admit Real Program Behaviors*. In Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014). ACM, New York, NY, USA, 428-431.

- Gregory Gay, Sanjai Rayadurgam, and Mats P.E. Heimdahl. 2014. *Improving the Accuracy of Oracle Verdicts through Automated Model Steering*. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (ASE '14). ACM, New York, NY, USA, 527-538.

- Gregory Gay, Sanjai Rayadurgam, and Mats P.E. Heimdahl. 2015. *Automated Steering of Model-Based Test Oracles to Admit Real Program Behaviors*. Under submission to ACM Transactions on Software Engineering & Methodology.

1.4 Structure of this Document

The remainder of this dissertation is organized as follows:

- Chapter 2 presents some background material, including information on test oracles, examples of model-based oracles, and an extended problem statement.
- Chapter 3 outlines our approach to the search-based steering of model-based test oracles, detailing the steering process, how state comparisons are made, sources of information for the constraints on the search process, and an automated method of learning steering constraints.
- Chapter 4 surveys related work on model-based testing, program steering, and search-based software testing.
- Chapter 5 details a set of research questions and experimental studies that we use to evaluate the effectiveness of our oracle steering framework.
- In Chapter 6, we explore the results of our evaluation and discuss thier implications with regard to the effectiveness of our approach, how it compares to other potential solutions, the impact of different types of constraints on the search process, and whether effective constraints can be learned.
- Chapter 7 examines potential threats to validity in our experimental studies.
- Finally, Chapter 8 concludes the dissertation.

Chapter 2

Background

There are two key artifacts necessary to test software, the *test data*—inputs given to the system under test—and the *test oracle*—a judge on the resulting execution [12, 13]. A *test oracle* can be defined as a predicate on a sequence of stimuli to and reactions from the SUT that judges the resulting behavior according to some specification of correctness [2].

The most common form of test oracle is a *specified oracle*—one that judges behavioral aspects of the system under test with respect to some formal specification [2]. Commonly, such an oracle checks the behavior of the system against a set of concrete expected values [14] or behavioral constraints (such as assertions, contracts, or invariants) [15]. However, specified oracles can be derived from many other sources of information; we are particularly interested in using behavioral models, such as those often built for purposes of simulation, analysis and testing [5].

Although behavioral models are useful at all stages of the development process, they are particularly effective in addressing testing concerns: (1) models allow analysis and testing activities to begin before the actual implementation is constructed, and (2) models are suited to the application of verification and automated test generation techniques that typically allow us to cover a larger class of scenarios than we can cover manually [16]. As such, models are often executable; thus, in addition to serving as the basis of test generation [5], models can be used as a source of expected behavior—as a *test oracle*.

Executable behavioral models can be created using many different notations. Presently, we focus our work on state-transition languages such as Stateflow, Statemate, or other finite state machine and automata structures [5].

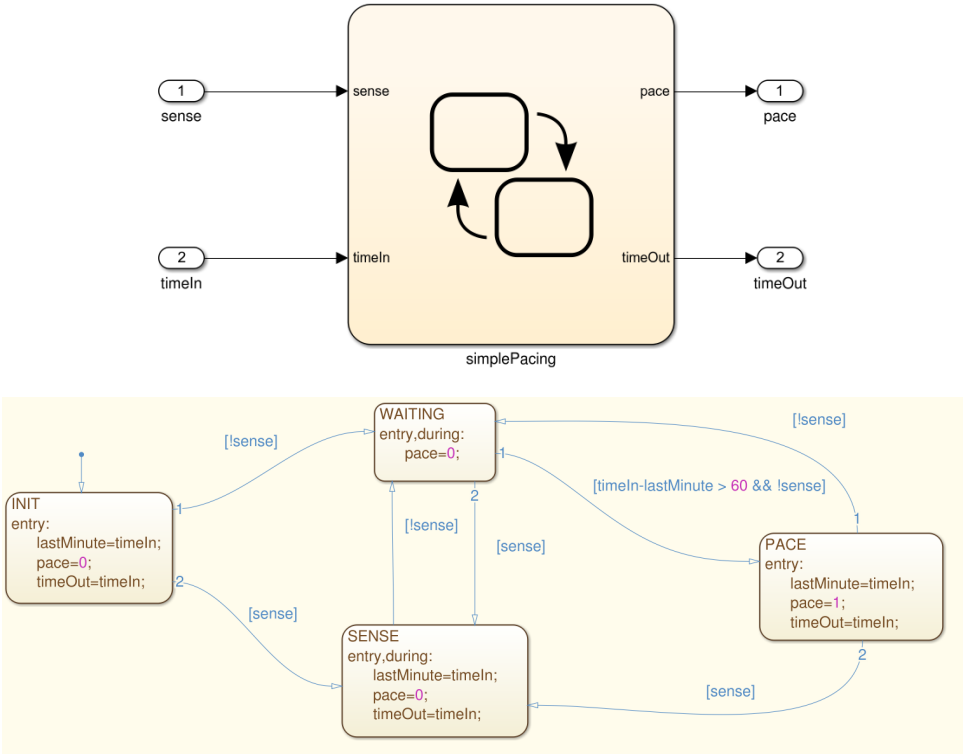

Figure 2.1: Model of software behavior for SimplePacing.

An example of such a model can be see in Figure 2.1. This model, written in the Stateflow notation [6], represents the behavior of the a simplified version of a pacemaker. The system takes in two inputs—a sensor reading that represents whether or not a natural heartbeat was sensed ($sense$), and the timestamp of the latest sensor reading ($timeIn$). It uses this information to determine, at one-second intervals, whether or not to issue an electrical pulse (a pace) to the heart chamber that it is implanted into. If a minute [1] goes by without a sensed event from the sensor, a pace is issued. The output of the software includes the outcome of the pace determination ($pace$)—issue a pace or do not issue a pace—and the time that the output is issued ($timeOut$). In the model, the time that the sensor is polled is the same as the time that output is issued ($timeIn = timeOut$). This will not be true in the actual software, but assuming

[1] A realistic pacemaker, such as the one used in our actual case study, makes decisions at the millisecond level—we use time values such as a "minute" in this example in order to explain the problems that we are interested in, rather than to accurately model an actual pacemaker

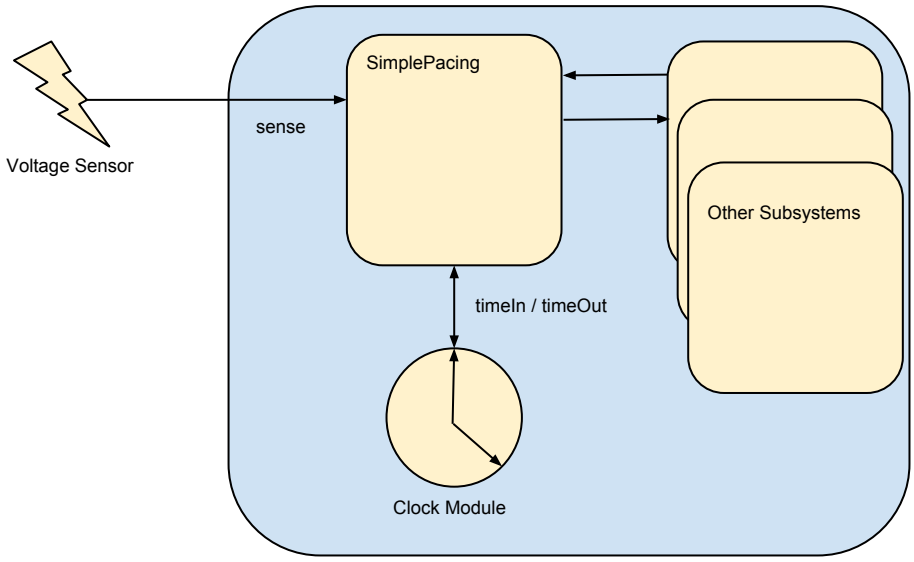

Figure 2.2: Illustration of the operating environment of SimplePacing's implementation.

instantaneous computation time (or a constant computation time) is a common abstration when modeling. We will refer to this example throughout this work as SimplePacing.

Non-determinism is a major concern in embedded real-time systems. The task of monitoring the environment and pushing signals through layers of sensors, software, and actuators can introduce points of failure, delay, and unpredictability. Input and observed output values may be skewed by noise in the physical hardware, timing constraints may not be met with precision, or inputs may arrive faster than the system can process them. Often, the system behavior may be acceptable, even if the system behavior is not exactly what was captured in the model—a model that, by its very nature, incorporates a simplified view of the problem domain. A common abstraction when modeling is to omit any details that distract from the core system behavior in order to ensure that analysis of the models is feasible and useful. Yet these omitted details may manifest themselves as differences between the behavior defined in the model and the behavior observed in the implementation.

To give an example of how these differences manifest themselves, consider the actual operating environment of the implementation of the SimplePacing software depicted in Figure 2.2. Although the model shown in Figure 2.1 demonstrates the behavior expected from the final SimplePacing system, a number of abstractions have been made. For example, the model receives

a simple binary sense. However, in the real world, the electrical impulses being sensed in the heart are complex analog readings, prone to noise. A physical sensor, or even the SimplePacing implementation itself, will need to take that reading and decide whether it is strong and clear enough to be considered as an actual sensed event. Similarly, the time stamps used as both input and output from SimplePacing will be taken by polling a clock module on the computing platform. While the timestamp for the input and output in the model are always the same, this may not be true for the actual software. Differences may arise from computation time, clock drift, and the difficulty of synchronizing the parallel components of the software. In systems such as pacemakers, time is a crucially important piece of data, and delays can lead to a behavior that is very different from the one produced by a model that abstracts such delays. Furthermore, clock issues are commonly non-deterministic. Repeated application of the same test stimulus may not result in the same output if, say, processing time varies. Additionally, while the SimplePacing model may be executed by itself, the final implementation is simply one subsystem in a larger pacemaker, and its execution may be impacted by the execution of the other subsystems. It may not be as easy to isolate the behavior of the SimplePacing module.

As illustrated in Figure 2.3, the behavior of the SimplePacing model and implementation might differ. In the first test depicted, no sensed events occur. Both the model and the implementation should deliver a pace every sixty seconds. However, for the actual implementation, minor computation delays result in a pace being delivered slightly off-schedule. The system requirements for SimplePacing likely recognize the commonality of such a scenario and prescribe a *tolerance period*, a bounded period of time where a pace is legal (that is, SimplePacing is acting correctly if the time between paces, in the absence of sensed events, is no more than 65 seconds). In this case, the implementation—despite the small delay—delivers paces withing the allowed timing window. However, as the comparison procedure expects the actual timestamp to exactly the predicted timestamp, the test will fail. It would not be easy to account for this difference in the model or behavior comparison, as the behavior at one step of execution is dependent on the execution history. As can be seen in the first test, the differences between the model and implementation grow larger as time passes and the delays accumulate. The time between paces in the implementation is still within the legal period, but the timestamps drift further from those produced by the model.

In the second test shown in Figure 2.3, the heart sensor picks up an unexpected signal at $timeIn = 110$ and registers it as a sensed event. As this was not a planned test stimulus, the

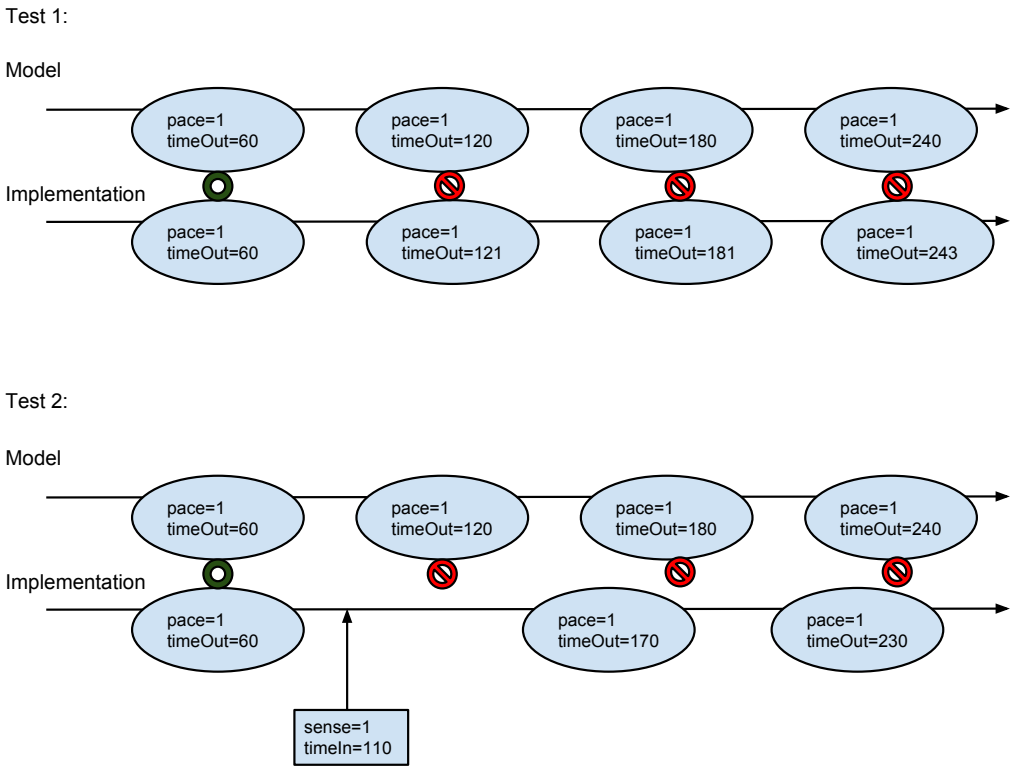

Figure 2.3: Abstraction-induced behavioral differences between SimplePacing's model and implementation.

executions of the model and implementation differ significantly from that point. The behavior of the implementation might be completely legal with respect to the sequence of events that follow after the additional sensed event, but as the executions of the model and implementation fail to conform, the test results in a failure. One could argue that picking up that additional input was, in itself, an error. However, tuning analog sensors is a complex process, and a small number of false positives may be accepted by testers if the result is never missing a real sensed event.

This raises the question—why use models as oracles? Alternative approaches could be to turn to an oracle based on explicit behavioral constraints—assertions or invariants—or to build declarative behavioral models in a formal notation such as Modelica [17]. These solutions, however, have their limitations. Assertion-based approaches only ensure that a limited set of properties hold at *particular points in the program* [15]. Further, such oracles may not be able to

account for the same range of testing scenarios as a model that prescribes behavior for all inputs. Declarative models that express the computation as a theory in a formal logic allow for more sophisticated forms of verification and can potentially account for timing-based non-deterministic behaviors [18]. However, Miller et al. have found that developers are more comfortable building constructive models than formal declarative models [19]. Constructive models are visually appealing, easy to analyze without specialized knowledge, and suitable for analyzing failure conditions and events in an isolated manner [18]. The complexity of declarative models and the knowledge needed to design and interpret such models make widespread industrial adoption of the paradigm unlikely.

While there are challenges in using constructive model-based oracles, it is a widely held view that such models are indispensable in other areas of development and testing, such as requirements analysis or automated test generation [16, 20]. From this standpoint, the motivational case for models as oracles is clear—if these models are already being built, their reuse as test oracles could save significant amounts of time and money, and allow developers to automate the execution and analysis of a large volume of test cases. Therefore, we seek a way to use constructive model-based oracles that can handle the non-determinism introduced during system execution on the target hardware.

Chapter 3

Oracle Steering

In a typical model-based testing framework, the test suite is executed against both the SUT and the model-based oracle, and the values of certain variables are recorded to a trace file after each execution step. The oracle's comparison procedure examines those traces and issues a verdict for each test (*fail* if test reveals discrepancies, *pass* otherwise). When testing a real-time system, we would expect non-determinism to lead to behavioral differences between the SUT and the model-based oracle during test execution. The actual behaviors witnessed in the SUT may not be incorrect—they may still meet the system requirements—they just do not match what the model produced. We would like the oracle to distinguish between correct, but non-conforming behaviors introduced by non-determinism and behaviors that are indicative of a fault.

The simplest approach that could potentially address this would be to augment the comparison procedure with a filtering mechanism to detect and discard acceptable differences on a per-step basis. For example, to address some of the computation-induced delays discussed in Section 2 and illustrated in Figures 2.2 and 2.3, a filter could simply allow behaviors that fall within a bounded time range to be acceptable, as long as they are the *same behaviors* predicted by the model. Such filters are relatively common in GUI or graphic rendering testing [21][1]. However, the issue with filtering on a per-step basis is that the effect of non-determinism may linger on for several steps, leading to irreconcilable differences between the SUT and the model-based oracle. Filters may not be effective at handling growing behavioral divergence. While a filter may be a perfectly appropriate solution for static GUIs, the cumulative build-up of differences in complex, time-based systems, will likely render a filter ineffective on longer time

[1] Filters could easily be implemented using assertion statements checked following the oracle procedure.

scales. Even if that filter was modified to track, say, the average response time for the SimplePacing system of Figure 2.1, the growing difference in timestamps may lead the whole system down an entirely different path of execution. If the time of input or output impacts behavior, such as in the case of a pacemaker—a system where even a single delayed input may impact all future commanded paces—a filter is unlikely to make an accurate judgment after the first few comparisons. Therefore, any potential solution to the issues we are concerned with must account for not just the current divergence between the SUT and the model-based oracle, but with all previous divergences as well. This is a non-trivial problem, as many systems are now expected to execute over a long period of time.

To address this challenge, we instead take inspiration from *program steering*—the process of adjusting the execution of live programs in order to improve performance, stability, or behavioral correctness [22]. Instead of steering the behavior of the SUT, however, we *steer the oracle* to see if the model is capable of matching the SUT's behavior. When the two behaviors differ, we backtrack and apply a *steering action*—e.g., adjust timer values, apply different inputs, or delay or withhold an input—that changes the state of the model-based oracle to a state more similar to the SUT (as judged by a dissimilarity metric).

We steer the oracle rather than the SUT because these deviations in behavior result not necessarily from the incorrect functioning of the system, but from a disagreement between the idealized world of the model and the reality of the execution of the system under test. In cases where the system is actually acting incorrectly, we don't want to steer at all—we want to issue a failure verdict so that the developer can change the implementation. In many of these deviations, however, it is not the system that is incorrect. If the model does not account for the real-world execution of the SUT, then *the model is the artifact that is incorrect*. Therefore, we can take inspriation from program steering for this situation—rather than immediately issuing a failure verdict, we can attempt to correct the behavior of the model.

Oracle steering, unlike filtering, is *adaptable*. Steering actions provide flexibility to handle non-determinism, while still retaining the power of the oracle as an arbiter. Of course, improper steering can bias the behavior of the model-based oracle, masking both acceptable deviations and actual indications of failures. Nevertheless, we believe that by using a set of appropriate constraints it is possible to sufficiently bound steering so that the ability to detect faults is still retained.

To steer the oracle model, we instrument the model to match the state it was in during the

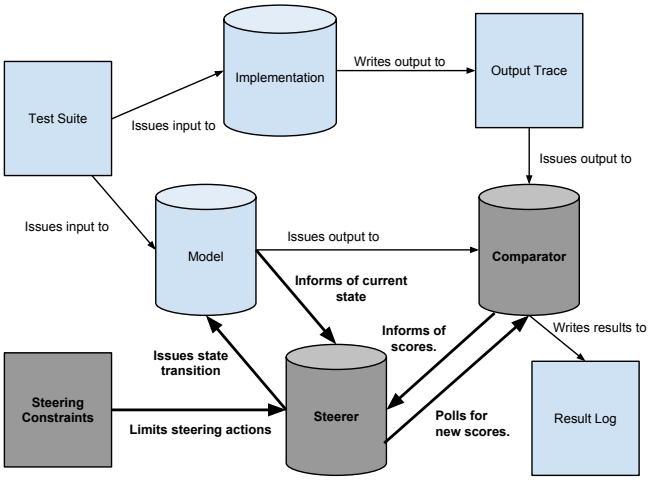

Figure 3.1: An automated testing framework employing steering.

previous step of execution, formulate the search for a new model state as a boolean satisfiability problem, and use a powerful search algorithm to select a target state to transition the model to. This search is guided by three types of constraints:

1. A set of **tolerance constraints** limiting the acceptable values for the *steering variables*—a set of *model* variables that the steering process is allowed to directly manipulate.

2. A **dissimilarity function**—a numerical function that compares a candidate model state to the state of the SUT and delivers a numeric score. We seek the candidate solution that minimizes this function.

3. A set of **additional policies** dictating the limits on steering.

In a typical testing scenario that makes use of model-based oracles, a test suite is executed against both the system under test and the behavioral model. The values of the input, output, and select internal variables are recorded to a *trace file* at certain intervals, such as after each discrete cycle of input and output. Some comparison mechanism examines those trace files and issues a verdict for each test case (generally a *failure* if any discrepancies are detected and a *pass* if a test executes without revealing any differences between the model and SUT). As illustrated in Figure 3.1, such a framework can be modified to incorporate automated oracle steering. The testing process is detailed in Figure 1. Informally, the test framework follows these steps:

ALGORITHM 1: Testing Process

```
Input: Model, SUT, Tests
for test ∈ Tests do
    stepNumber = 0
    for step ∈ test do
        previousState = state(Model)
        applyInput(SUT, step)
        applyInput(Model, step)
        S_m = state(Model)
        S_sut = state(SUT)
        if Dis(S_m, S_sut) > 0 then
            instrumentedModel = instrument(Model, previousState)
            steer(instrumentedModel, S_m, S_sut)
            S_m^new = state(Model)
            if Dis(S_m^new, S_sut) > 0 then
                verdict = fail
                break
            end
        end
    end
    verdict = pass
end
```

1. Execute each test against the system under test, logging the state of certain variables at each test step.

2. Execute each test against the model-based oracle, and for each step of the test:

   (a) Feed input to the oracle model.

   (b) Compare the model output to the SUT output.

   (c) If the output does not match, the steering algorithm will instrument the model and attempt to steer the model's execution within the specified constraints by searching for an appropriate steering action.

   (d) Compare the new output of the model to the SUT output and log the final dissimilarity score.

3. Issue a final verdict for the test.

More information about model instrumentation can be found in Section 3.2, and the algorithm for steering is detailed in Figure 2 (and explained in Section 3.2).

In the following sections, we will explain the search process in more detail. In Section 3.1, we describe our assumptions on the format of model used as well as the model format that we operate on in our implementation of the steering framework. In Section 3.2, we detail the search process and discuss how it selects a steering action. In Section 3.3, we offer advice on the selection of tolerance constraints and policies. In Section 3.4, we present a process for automatically learning candidate constraints from observing unconstrained steering actions.

3.1 System Model

In the abstract, we define a model as a transition system $M = (S, S_0, \Sigma, \Pi, \rightarrow)$, defined as:

- S is a set of states—assignments of values to system variables—with initial state S_0.
- Σ is an input alphabet, defined as a set of input variables for the model.
- Π is a specially-defined *steering alphabet*—$\Pi \subseteq \Sigma$—a set of *steerable variables*—the variables that the steering procedure is allowed to directly control and modify the assigned values of.
- $\rightarrow$ is a transition relation (a binary relation on S), such that every $s \in S$ has some $s' \in S$ with $s \rightarrow s'$.

Any model format that can be expressed as such an M, in theory, can be the target of a steering procedure. In this work, our models are written in the Stateflow notation from Mathworks [6]. However, many modeling notations can be formulated in terms of a transition system.

The primary difference of this definition from a standard state-transition system is the steering alphabet, Π. By definition, $\Pi \subseteq \Sigma$. That is, the steerable variables are considered to be input variables, but not all input variables are required to be steerable. Variables internal to the model may (and, likely, will often be specified as steerable), but they must be transformed into input variables for the computation steps where steering is applied. This transformation enables search algorithms to directly perform steering operations on the variables. On subsequent, unassisted execution steps, the model will be again transformed such that those variables are again internal to the model.

In practice, we perform test execution and steering by translating models from their native language into Lustre, a synchronous dataflow language used in a number of manufacturing

```
node simplePacing(sense: bool;
      timeIn: int)
   returns (pace: bool;
      timeOut: int);
var
  internal1: int;
  (...)
  internal42: bool;
let
  timeOut = (if (not (internal2 = 0)) then internal8 else timeIn);
  pace = ((not (internal2 = 0)) and internal7);

  internal42 = (true -> false);
  internal12 = (if (internal2 = 2) then internal29 else internal16);
  internal29 = (if ( not sense) then internal4 else internal27);
  internal15 = (if (internal2 = 3) then internal28 else internal19);
  internal24 = (if (((timeIn - internal1) > 60) and (not sense)) then timeIn
                                                                 else internal27);
  internal14 = (if (internal2 = 3) then internal32 else internal18);
  internal28 = (sense and internal30);
  internal19 = ((not (internal2 = 4)) and internal3);
  internal17 = (if (internal2 = 4) then internal33 else internal1);
  internal13 = (if (internal2 = 3) then internal31 else internal17);
  internal33 = (if ( not sense) then internal1 else timeIn);
  internal18 = (if (internal2 = 4) then internal34 else internal2);
  internal32 = (if ( not sense) then 1 else internal26);
  internal5 = (if (internal2 = 1) then internal21 else internal9);
  internal8 = (if (internal2 = 1) then internal24 else internal12);
  internal9 = (if (internal2 = 2) then internal31 else internal13);
  internal25 = (if sense then timeIn else internal1);
  internal6 = (if (internal2 = 1) then internal22 else internal10);
  internal23 = (((timeIn - internal1) > 60) and ( not sense));
  internal26 = (if sense then 4 else internal2);
  internal3 = (( not internal42) and internal40);
  internal20 = (if (internal2 = 4) then internal35 else internal4);
  internal11 = (if (internal2 = 2) then internal28 else internal15);
  internal35 = (if ( not sense) then internal4 else timeIn);
  internal16 = (if (internal2 = 3) then internal29 else internal20);
  internal22 = (if (((timeIn - internal1) > 60) and (not sense)) then 3
                                                                 else internal26);
  internal30 = (( not sense) and internal3);
  internal21 = (if (((timeIn - internal1) > 60) and (not sense)) then timeIn
                                                                 else internal25);
  internal27 = (if sense then timeIn else internal4);
  internal7 = (if (internal23 = 1) then internal23 else internal11);
  internal31 = (if ( not sense) then internal1 else internal25);
  internal34 = (if ( not sense) then 1 else internal2);
  internal10 = (if (internal2 = 2) then internal32 else internal14);
  internal4 = (if internal42 then 0 else internal41);
  internal2 = (if internal42 then 0 else internal39);
  internal1 = (if internal42 then 0 else internal38);
  internal37 = (if (not (internal2 = 0)) then internal6 else 2);
  internal36 = (if (not (internal2 = 0)) then internal5 else timeIn);
  internal41 = (0 -> pre(timeOut));
  internal40 = (false -> pre(pace));
  internal39 = (0 -> pre(internal37));
  internal38 = (0 -> pre(internal36));
tel;
```

Figure 3.2: SimplePacing, translated to the Lustre language

industries to model or directly implement embedded systems [23][2]. Lustre is a declarative programming language for manipulating data flows—infinite streams of variable values. These variables correspond to traditional data types, such as integers, booleans, and floating point numbers. A Lustre program—or a node—is the specification of a stream transformer, mapping the streams of input variables to the streams of internal and output variables using a set of defined expressions. Lustre nodes have cyclic behavior—at execution cycle i, the node takes in the values of the input streams at instant i, manipulates those values, and issues values for the internal and output variables. Nodes have a limited form of memory, and can access input, internal, and output values from previous instants (up to a statically-determined finite limit).

Figure 3.2 depicts the Lustre translation of SimplePacing, shown in the original Stateflow format in Figure 2.1. The body of a Lustre node consists of a set of equations of the form $x = expr$—as can be seen in Figure 3.2—where x is a variable identifier, and t is the expression defining the value of x at instant i. Like in most programming languages, expression t can make use of any of the other input, internal, or output variables in defining x—as long as that variable has already been assigned a value during the current cycle of computation.

Lustre supports many of the traditional numerical and boolean operators, including $+$, $-$, $*$, $/$, $<$, $>$, $\%$, etc. Lustre also supports two important *temporal* operators: $pre(x)$ and $\rightarrow$. The $pre(x)$ operator, or "previous", evaluates to the value of x at instant $(i-1)$. The $\rightarrow$ operator, or "followed by", allows initialization of variables in the first instant of execution. For example, the expression $x = 0 \rightarrow pre(x) + 1$ defines the value of x to be 0 in instant 0, then defines it as 1 at instant 1—or, the value at instant 0 plus one—and so forth.

Because of the simplicity and declarative nature of Lustre, it is well-suited to model checking and verification, in particular with regards to its safety properties [25]. This also makes it an ideal language to use as a target for steering because, as we will elaborate on in the next section, the steering constraints and dissimilarity function can both be encoded directly into the models, then solved using the same algorithms that are regularly used to prove safety properties over the programs expressed in the Lustre language.

Additionally, because typical discrete state-transition systems are semantically similar to Lustre, it is easy to translate from other modeling paradigms to Lustre while preserving the semantic structure of those models.

[2]This translation is conducted using the Gryphon framework, licensed from Rockwell Collins [24].

3.2 Selecting a Steering Action

If the initial comparison of model and SUT states s_m and s_{sut} reveals a difference, we attempt to steer the model. Fundamentally, we treat steering as a search process. We backtrack the model, instrumenting it with the previous recorded state as the new initial state S_0, and seek a steering action—a set of values for the steerable variables Π that, when combined with the assigned values to the remaining input variables $\Sigma - \Pi$, transitions the model to a new state s_m^{new}. Note that, if the steering process fails to produce a solution, $s_m^{new} = s_m$. The choice of a steering action is a solution to a search problem where we seek an assignment to the variables in Π that satisfies the **tolerance constraints**, minimizes the **dissimilarity function**, and follows any **additional policies**.

The set of **tolerance constraints** governs the allowable changes to values of the steerable variables Π. These constraints define bounds on the non-determinism or behavioral deviation that can be accounted for with steering. Constraints can be expressed over any internal, input, or output variables of the model—not just the members of Π. Constraints can even refer to the value of a variable in the SUT.

For example, consider the scenario outlined in the first test in Figure 2.3. We could use steering to correct this mismatch by signifying $timeIn$ as a member of Π and allowing the search algorithm to assign a new value to it. However, we want to differentiate allowable time delays from fault-induced time delays, so we must also set a constraint on the change in value of $timeIn$. A reasonable constraint might be to allow the new value of $timeIn$ to fall anywhere between a minimum of $timeIn^{original}$ and $(timeIn^{original} + 4ms)$.

This differs from setting a filter to compare the values of s_m and s_{sut} because, by changing the state of the model, we impact the state of the model in future steps as well, allowing us to match the behavior of the model and the system, while limiting the ability of steering to mask faults. By adjusting the execution as execution commences, we also eliminate the need to track the entire execution history, as divergences are accounted for as they appear.

Consider the second test in Figure 2.3. In this test, a spike in electrical interference caused the SUT to sense an input event that was not an explicit part of the test sequence. Although this sensitivity to noise might be considered a *hardware* fault, the *software* in the SUT acted correctly in its response to the event. We can use steering to account for this noise as well, by designating $sense$ as a member of Π and allowing the search algorithm to adjust its value.

However, letting the steering process freely ignore or create senses might be dangerous, so we might—for example—set a constraint that $if\ (sense^{original} = 1)\ then\ (sense^{new} = 1)$. That is, the search algorithm can indicate that there is a sense when we had not previously indicated one as part of the test inputs, but the search algorithm can never ignore a sense that was supposed to be there.

We could also create a constraint that combines multiple members of Π, that takes into account model variables not in Π, or that depends on the value of a variable in the SUT. For example, we could establish a constraint on $sense$ that depends on the output variable $timeOut$ as follows: $if\ ((timeOut^{sut} \geq timeOut^{original})\ and\ (timeOut^{sut} \leq timeOut^{original} + 4ms)\ then\ ((sense^{new} = 0)\ or\ (sense^{new} = 1))\ else\ (sense^{new} = sense^{original})$. That is, we can freely change whether an event was sensed, as long as the value of $timeOut$ in the SUT is within a certain range of the value of $timeOut$ in the model.

After using the tolerance constraints to limit the number of candidate solutions, the search process is guided to a solution through the use of a **dissimilarity function** $Dis(model\ state, SUT\ state)$, that compares the state of the model to the observable state of the SUT. We seek a minimization of $Dis(s_m^{new}, s_{sut})$. That is, within the bounds on the search space set by the tolerance constraints, we seek the candidate solution with the lowest dissimilarity score. There are many different functions that can be used to calculate dissimilarity. Cha provides a good primer on the calculation of dissimilarity [26]. As we primarly worked with numeric variables, dissimilarity functions such as the Euclidean distance—the average difference between two variable vectors—were found to be sufficient to guide this selection. When considering variable comparisons over—for instance—strings, more sophisticated dissimilarity metrics (such as the Levenshtein distance [27]) may be more appropriate.

We can further constrain the steering process by employing a set of general **policy decisions** on when to steer. For example, one might decide not to steer unless $Dis(s_m^{new}, s_{sut}) = 0$. That is, one might decide not to steer at all unless there exists a steering action that results in a model state identical to that observed in the SUT.

To summarize, the new state of the model-based oracle following the application of a steering action must be a state that is reachable from the current state of the model, must fall within the boundaries set by the tolerance constraints, and must minimize the dissimilarity function.

This is illustrated in Figure 3.3 for the test step corresponding to the second output event in the first test in Figure 2.3. The Lustre translation of SimplePacing is instrumented such that

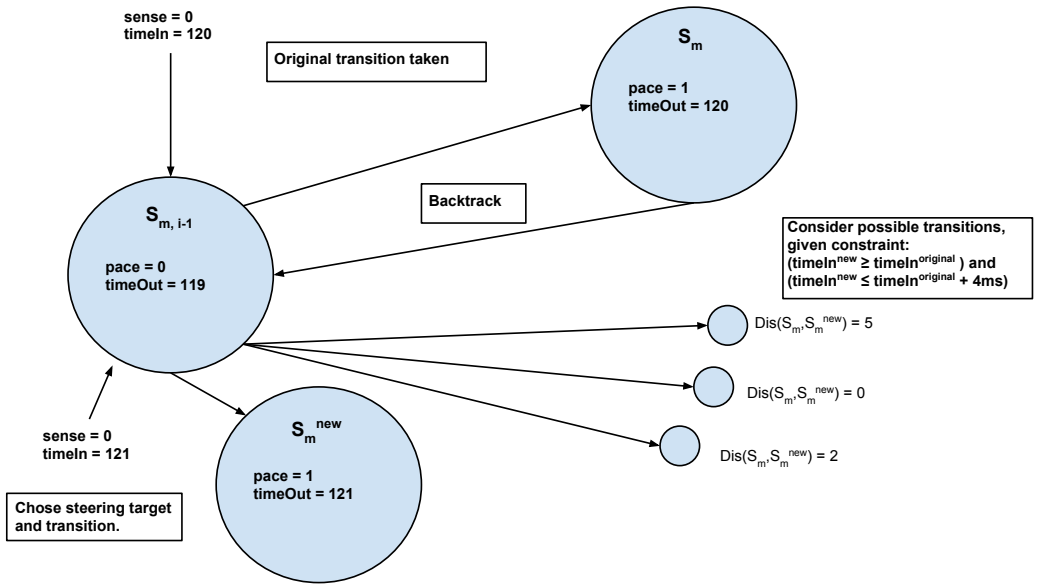

Figure 3.3: Illustration of steering process.

the initial state matches the state that the model was in, following the application of input in the immediately-preceding time step. The steerable variable set Π consists of the input $timeIn$, with the tolerance constraint that the chosen value of $timeIn$ must fall between $timeIn^{original}$ and $timeIn^{original} + 4ms$. We examine the candidate states, evaluate their dissimilarity score, then choose the steering action that minimizes this score. We transition the model to this state and continue test execution.

We have implemented the basic search approach outlined in Figure 2. Our search process is based on SMT-based bounded model checking [25], an automatic approach to property verification for concurrent, reactive systems [28]. The problem of identifying an appropriate steering action can be expressed as a Satisfiability Modulo Theories (SMT) instance. An SMT instance is a generalization of a boolean satisfiability instance in which atomic boolean variables are replaced by predicated expressed in classical first-order logic [29]. A SMT problem can be thought of as a form of a constraint satisfaction problem—in our specific case, we seek a set of values for the steerable variables that obeys the set of tolerance constraints and has a lower dissimilarity score than the original. By negating the set of constraints—asserting that we *can not* select a steering action—a bounded model checker can search for a solution to the SMT instance that proves that we *can* bring the model into a state more closely resembling that of the

ALGORITHM 2: Steering Process

Input: $Model, S_m, S_{sut}$
if $Dis(S_m, S_{sut}) > 0$ **then**
    $DisConstraint := \lambda threshold \rightarrow \lambda state \rightarrow Dis(state, S_{sut}) \leq threshold$
    $targetState := searchForNewState(Model, S_m, S_{sut}, Constraints, DisConstraint(0))$
    if *targetState = NULL* **then**
        $newState := S_m$
        $T := 1$
        while $newState \neq NULL$ **do**
            $targetState := newState$
            $newState :=$
            $searchForNewState(Model, targetState, S_{sut}, Constraints, DisConstraint(T \times$
            $Dis(targetState, S_{sut}))$
            $T := 0.5 \times T$
        end
    end
    $transitionModel(Model, targetState)$
end

system within a limited number of transitions (generally, just a single transition). In this work, we have made use of Kind [25], a model checker for the Lustre language, and the Z3 constraint solver [30].

We execute tests using an in-house Lustre interpreter. Each test step is checked explicitly for conformance violations by comparing the state of the model and the oracle. If a mismatch is detected, we allow the steering framework to search for a new solution. In order to achieve this, we explicitly instrument the model such that the current state (before applying the chosen steering action) is the "initial" state. This instrumentation also embeds the calculation of the dissimilarity function and the tolerance constraints directly into the model as an SMT instance that the search algorithm must find a solution to.

An example of this instrumentation can be see in Figure 3.4. If the mismatch occurred after the first step of execution, any expressions containing a "followed-by" operator (e.g., $x = 0 \rightarrow y$), is modified to only contain the right-hand side of the $\rightarrow$ ($x = 0 \rightarrow y$ is transformed into $x = y$). This is because execution is now past the "initial state," and thus, the expression needs to be calculated using the expression that should be used following the first execution step. Any use of the "previous" operator is also translated out—the $pre(y)$ in $x = 0 \rightarrow pre(y)$ is replaced with the stored value of y in the previous step of the execution trace. The replacement of pre operators with constants eliminates the need to share both the model and trace memory with the

```
node simplePacing(sense: bool;
      timeIn: int)
   returns (pace: bool;
      timeOut: int);
var
  internal1: int;
  (...)
  internal42: bool;
  concrete_oracle_sense: bool;
  concrete_oracle_timeIn: int;
  concrete_oracle_pace: bool;
  concrete_oracle_timeOut: int;
  concrete_sut_pace: bool;
  concrete_sut_timeOut: int;
  score_new_pace: real;
  score_original_pace: real;
  score_new_timeOut: real;
  score_original_timeOut: real;
  score_steered: real;
  score_original: real;
  prop: bool;
let
  timeOut = (if (not (internal2 = 0)) then internal8 else timeIn);
  pace = ((not (internal2 = 0)) and internal7);
  (...)
  internal42 = false;
  internal41 = 119;
  internal40 = false;
  internal39 = 2;
  internal38 = 0;

  concrete_oracle_sense = false;
  concrete_oracle_timeIn = 120;
  concrete_oracle_pace = true;
  concrete_oracle_timeOut = 120;
  concrete_sut_pace = true;
  concrete_sut_timeOut = 121;
  score_new_pace = (if (concrete_sut_pace != pace)
    then 1.0 else 0.0);
  score_original_pace = (if (concrete_sut_pace != concrete_oracle_pace)
    then 1.0 else 0.0);
  score_new_timeIn = (if (concrete_sut_timeIn > timeIn)
    then (concrete_sut_timeIn - timeIn) else (timeIn - concrete_sut_timeIn));
  score_original_timeIn = (if (concrete_sut_timeIn > concrete_original_timeIn)
    then (concrete_sut_timeIn - concrete_original_timeIn)
    else (concrete_original_timeIn - concrete_sut_timeIn));

  score_steered = score_new_pace + score_new_sense;
  score_original = score_original_pace + score_original_sense;
  prop = not ((timeIn >= concrete_oracle_timeIn) and
    (timeIn <= concrete_oracle_timeIn + 4.0) and
    (score_steered < score_original));

  --%PROPERTY prop;
tel;
```

Figure 3.4: SimplePacing, instrumented for steering. Truncated from Figure 3.2 to conserve space.

search algorithm. Instead, we explicitly embed the earlier values of variables.

Constants are also embedded in the model for each input variable and output variable in the model and each output variable in the SUT describing what occured originally during test execution. That is, they tell us what happened before we steered. These values are used both for calculating the value of the dissimilarity score (in the $score_original$ statement) and in the tolerance constraints (embedded in the $prop$ statement). All of these expressions are used within the set of constraints that we want the search algorithm to satisfy in its chosen steering action, all combined in the $prop$ expression. This expression includes the tolerance constraints—in our example in Figure 3.4, that the chosen timestamp on input falls within four milliseconds of the one originally exhibited by the model—and the threshold that we want the new dissimilarity score to beat. In this case, we want a solution that obeys the tolerance constraint (the timestamp falls in the chosen interval) and where the state of the model exactly matches the state of the system, calculated using the Manhattan distance.

The $prop$ expression is negated because we want a *counterexample*—we assert that the constraints *can not* be satisfied, and ask the search algorithm to find a set of values for the steerable variables that *can* satisfy those constraints within a single transition. We take the counterexample offered by the search algorithm, extract the values of the steerable variables, and replace the original values of those variables in the trace. We then apply the new set of input (non-steerable inputs that retain their original values, and we append the new values of the steerable variables) to the instrumented model in the Lustre interpreter, record the new values of all of the internal and output variables, and continue test execution.

The use of SMT-based complete search techniques is ideal for many Lustre programs because such solvers can efficiently decide the validity of formulas with linear numerical terms [29]. The main limitation, however, is that SMT-based solvers are limited in their ability to prove or disprove non-linear mathematical operations within properties. In general, this is not an issue—the tolerance constraints for a model do not generally need to contain sophisticated multiplication or division (if either operation is used, it is generally only to multiply or divide by a constant). We use Microsoft Researcher's Z3 solver [30] because it was found to offer the best support for non-linear expressions—enabling the ability to directly calculate certain dissimilarity metrics as part of the process of producing a counterexample.

It should be noted that an SMT solver may not be able to directly minimize $Dis(s_m^{new}, s_{sut})$. Instead, such solvers will offer any solution that satisfies that constraint—that is, any solution

that offers a smaller score and satisfies our other constraints. As outlined in Figure 2, we instead find a minimal solution by first using the constraint $Dis(s_m^{new}, s_{sut}) = 0$—we ask the solver for a solution where, at least for the compared output variables, $S_m^{new} = S_{sut}$. If an exact minimization can not be found, we then attempt to narrow the range of possible solutions by setting a threshold value T and using the constraint $Dis(s_m^{new}, s_{sut}) < (T * Dis(s_m, s_{sut}))$. If a solution is possible, we continue to set $T = 0.5 * T$ until a solution is no longer offered. Once that constraint can no longer be satisfied, we take the solution offered for the previous—satisfiable—value of T and iteratively apply the constraint $Dis(s_m^{new}, s_{sut}) < Dis(s_m^{previous\ new}, s_{sut})$ until we can no longer find a solution offering a lower value for the dissimilarity function. The best solution found will be selected as the steering action.

3.3 Selecting Constraints

The efficacy of the steering process depends heavily on the tolerance constraints and policies employed. If the constraints are too strict, steering will be ineffective—leaving as many "false failure" verdicts as not steering at all. On the other hand, if the constraints are too loose, steering runs the risk of covering up real faults in the system. Therefore, it is crucially important that the constraints to be employed are carefully considered.

Often, constraints can be inferred from the system requirements and specifications. For example, when designing an embedded system, it is common for the requirements documents to specify a desired accuracy range on physical sensors. If the potential exists for a model-system mismatch to occur due to a mistake in reading input from a sensor, than it would make sense to take that range as a tolerance constraint on that sensor input and allow the steering algorithm to try values within that range of the canonical test input.

We recommend that users err toward strict constraints. While it is undesirable to spend time investigating failures that turn out to be acceptable, that outcome is preferable to masking real faults. Steering will not fully account for a model that produces incorrect behavior, so steering should start with a mature, verified model.

To give an example, consider a pacemaker. The pacemaker might take as input a set of prescription values, event indicators from sensors in the patient's heart, and timer values. We would recommend that steering be prohibited from altering the prescription values at all, as manipulation of those values might cover faults that could threaten the life of a patient. However,

as electrical noise or computation delays might lead to issues, steering should be allowed to alter the values of the other inputs (within certain limits). The system requirements might offer guidance on those limits—for instance, specifying a time range from when a pace command is supposed to be issued to when it must have been issued to be acceptable. This boundary can be used as to constrain the manipulation of timer variables. Furthermore, given the critical nature of a pacemaker, a tester might also want to employ a policy where steering can only intervene if a solution can be found that identically matches the state of the SUT.

Unlike approaches that build non-determinism into the model, steering decouples the specification of non-determinism from the model. This decoupling allows testers more freedom to experiment with different sets of constraints and policies. If the initial set of constraints leaves false failure verdicts or if testers lack confidence in their chosen constraints, alternative options can easily be explored by swapping in a new constraint file and executing the test suite again. Using the dissimilarity function to rate the set of final test verdicts, testers can evaluate the severity and number of failure verdicts remaining after steering with each set of constraints and gain confidence in their approach.

3.4 Learning Constraints

While it is clear that choosing the correct constraints is essential to accurate steering, it is not always clear what those constraints should be, or even what variables should be manipulated by steering in the first place. However, even in these situations, the developers of the system under test *should* at least have an idea of whether a test should pass or fail. A human oracle is often the ultimate judge on the correctness of the produced plans, as the developers or other domain experts will likely have a more comprehensive idea of what constitutes correctness than any constructed artifact, even if they are not completely sure of the specific factors that should lead to that verdict.

There are heavy restrictions on the volume of testing that a human oracle can judge [31]. This is why we wish to employ model-based test oracles in the first place. However, it may be possible to use human-classified test verdicts to *learn* an initial set of constraints. We can treat constraint elicitation as a machine learning problem. We can execute a series of tests against the SUT, steer with no value constraints at all—the only limitation being what states are reachable within one transition through changes to the steerable variables—and record information on

Variables Involved	Attribute	Values
For each steerable variable	How much was it changed?	Continuous
For each oracle-checked variable	How much did it differ before steering?	Continuous
For each oracle-checked variable	How much was it changed?	Continuous
(class variable)	Did steering change the verdict correctly?	NotChanged, ChangedIncorrect, ChangedCorrect

Table 3.1: Data Extracted for Tolerance Elicitation

what changes were imposed by the steering algorithm. If a human serves as an oracle on those tests, we can then evaluate the "correctness" of steering.

For purposes of constraint elicitation, we care about the effects of steering in two situations: successfully steering when we are supposed to steer and successfully steering when we *are not* supposed to steer. By observing the framework-calculated oracle verdict before and after steering and comparing it to the human-classified oracle verdict, we can determine what test steps correspond to those two situations. Using that correctness classification and a set of data extracted from each test step, we can form a dataset that can be explored by a variety of learning algorithms. This process is illustrated in Figure 3.5. The data we extract is detailed in Table 3.1.

We can use this extracted set of data to elicit a set of tolerance constraints. A standard practice in the machine learning field is to *classify data*—to use previous experience to categorize new observations [32]. As new evidence is examined, the accuracy of these categorizations is refined and improved. An example of a classification problem might be the process by which a test oracle arrives at a "pass" or "fail" verdict in the first place, and one could imagine sophisticated machine learning algorithms replacing traditional oracles completely for certain types of systems and testing scenarios.

We are instead interested in the reverse scenario. Rather than attempting to categorize new data, we want to work backwards from the classifications to discover *why steering acted correctly or incorrectly*—a process known as treatment learning [33]. Treatment learning approaches take the classification of an observation and try to reverse engineer the statistical evidence that led to that categorization. Such learners produce a *treatment*—a small set of data value boundaries that, if imposed, will change the expected class distribution. By filtering the data for entries that follow the rules set forth by the treatment, one can identify how a particular classification is reached.

What chiefly differentiates treatment learning from classification is the focus of the approach. Ultimately, classifiers strive to increase the representational accuracy by growing a

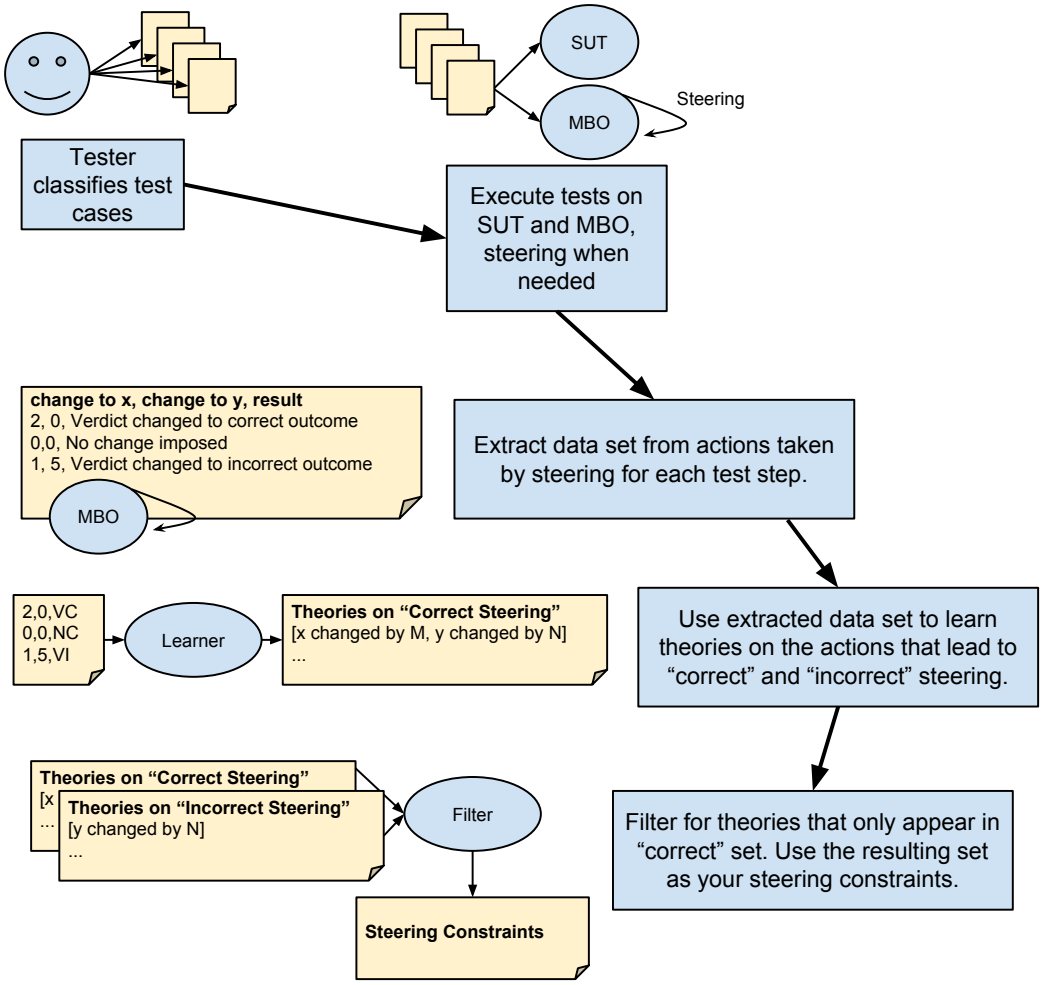

Figure 3.5: Outline of learning process.

collection of statistical rules. As a result, if the data is complex, the model employed by the classified will also be complex. Instead, treatment learning focuses on minimality, delivering the smallest rule that can be imposed to cause the largest impact. This focus is exactly what makes treatment learning interesting as a method of eliciting tolerance constraints. We wish to constrain steering to a small set of steerable variables, with strict limitations on what value changes are allowed. Treatment learning can deliver exactly this, and can be used to both generate an initial set of constraints and to tune an existing set of constraints.

Class	Percentage
NotChanged	96%
ChangedIncorrect:	1%
ChangedCorrect:	2%

Table 3.2: Base class distribution

Rank	Worth	Treatment
1	3.428298	[changeToVariable1=[-4.000000..0.000000] [changeToVariable2=[1.000000..2.000000]
2	3.384346	[changeToVariable2=[1.000000..2.000000] [changeToVariable3=[-10.000000..0.000000]
3	3.352951	[changeToVariable2=[1.000000..2.000000]
4	3.352951	[changeToVariable4=[-15.000000..0.000000] [changeToVariable2=[1.000000..2.000000]
5	3.352951	[changeToVariable2=[1.000000..2.000000] [changeToVariable5=[-176.000000..0.000000]

Table 3.3: Examples of learned treatments.

Class	Percentage
NotChanged	0%
ChangedIncorrect:	14%
ChangedCorrect:	86%

Table 3.4: Class distribution after imposing Treatment 1 from Table 3.3

To give an example, consider the base class distribution after steering a model-based oracle for a set of classified tests and extracting the data detailed above, as shown in Table 3.2. This sort of base class distribution makes conceptual sense—on many test steps, steering does nothing. It only kicks in when the oracle and model differ, makes a change, and likely reduces the number of future steps in the same test case where differences occur. By targeting the class $ChangedCorrect$, we can attempt to elicit a treatment that details what happens when steering acts correctly. We can extract treatments, ranked by their score assigned by the treatment learner's objective function. Example treatments are shown in Table 3.3.

The first treatment in Table 3.3 states that the most major indicators of a correct change are when the value of Variable1 is reduced between 0-4 seconds and the value of Variable2 is increased by 1-2 seconds. By imposing that treatment, we end up with the class distribution shown in Table 3.4. This class distribution shows strong support for the produced treatment.

In order to create a set of tolerance constraints, we first create 10 treatments like those seen

```
((Variable3 >= concrete_oracle_Variable3 - 10.0) and
     (Variable3 <= concrete_oracle_Variable3))
((Variable2 = concrete_oracle_Variable2) or
     ((Variable2 >= concrete_oracle_Variable2 + 1.0) and
     (Variable2 <= concrete_oracle_Variable2 + 2.0)))
((Variable1 >= concrete_oracle_Variable1 - 4.0) and
     (Variable1 <= concrete_oracle_Variable1))
((Variable6 >= concrete_oracle_Variable6 - 13.0) and
     (Variable6 <= concrete_oracle_Variable6))
(Variable7 = concrete_oracle_Variable7)
(Variable8 = concrete_oracle_Variable8)
...
(Variable20 = concrete_oracle_Variable20)
```

Figure 3.6: Examples of produced tolerances.

in Table 3.3 using "ChangedCorrect" as our target class (we wish to know what actions steering takes when it works correctly) and extract all of the individual variable and value range pairings. Some of these items may not actually be indicative of successful steering—they may be variable values selected by biases in the algorithm that selects the steering actions that appear in both *correct* and *incorrect* steering. Thus, we also produce 10 treatments using "ChangedIncorrect" as the target class. This produces a set of treatments indicating what happens when steering incorrectly changes an oracle verdict. We remove any variable and value range pairings that appear in both the "good" and "bad" sets, leaving only those that appear in the good set. We then form our set of elicited tolerance constraints by locking down any variables that constraints were not suggested for. This results in a set of tolerances similar to that shown in Figure 3.6.

Conceptually, a treatment learner explores all possible subsets of the attribute ranges of a dataset searching for optimal treatments. Such a search is infeasible in practice, so much of the effectiveness of a treatment learning algorithm lies in quickly pruning unpromising attribute ranges—ignoring rules that lead to a class distribution where the target class is in the minority [34]. Notable treatment learning algorithms include the TAR family (TAR2 [33], TAR3 [35, 36, 37], and TAR4.1 [34])—a series of algorithms utilizing a common core structure, but employing different objective functions and search heuristics—and the STUCCO contrast-set learner [38]. Other optimization algorithms have also been applied to treatment learning, including simulated annealing and gradient descent [34]. For the study in this work, we employed the TAR3 algorithm, explored further in Section 5.5.

Chapter 4

Related Work

In this chapter, we will discuss three research areas that have informed the idea of oracle steering: model-based testing (Section 4.1), program steering (Section 4.2), and search-based software testing (Section 4.3).

4.1 Model-Based Testing

Model-based testing (MBT) is the practice of using models of software systems for the derivation of test suites [1]. MBT techniques began to see industrial use around 2000, and by 2011, significant industrial applications exist and commercial-grade tools are available [1]. Such techniques commonly take a model in the form of a finite-state machine [5] or other labeled transition system [39]—or another modeling notation that is transformed, "under the hood" into the appropriate automata structure—and generate a series of tests to apply to the system under test. An attempt is then made to establish conformance between the model and the system [40]. Often, the model also serves as the oracle on the test. Test inputs are "executed" against both the model and the SUT, and the resulting state is compared. However, in some cases, the model may only be used as a source of test inputs, and a separate oracle may be used. We are concerned with work where the model serves as the oracle.

Model-based testing is seen as a way to bridge the gap between *testing* and *verification*. Verification is aimed at proving properties about the system through formal reasoning about models of the system, often with the goal of arguing that the system will satisfy the user's

requirements [41]. Verification can give certainty about satisfaction of a required property—often through an exhaustive search of the state space of the model or explicit proof generation—but this certainty only applies to the model of the system. Any form of verification is only as good as the validity of the model itself [41]. Testing, however, is performed by exercising the full implementation of the system. Testing can never be complete—in practice, testing will only ever cover a tiny percentage of the state space of a real-world system. Testing can never prove the absence of errors, only their presence. By proving that a model conforms to the requirements, and demonstrating that the system conforms to the model—by generating tests from the model and comparing the resulting execution of the system to the behavior of the model—developers can make a strong argument that the final system also conforms to the requirements.

Much of the research on model-based testing is concerned explicitly with the generation of test cases with the goal of demonstrating that the SUT conforms to the model. In this scenario, the model serves implicitly as an oracle on the generated tests, being the basis on which correctness is judged. In addition to a formal model, these test generation algorithms require an *implementation relationship* [42, 43]. A system under test is a complex object that is not inherently amenable to formal reasoning. Comparing model behavior to SUT behavior is not a straight-forward task. In many cases, this comparison is only possible by treating the SUT as a black box that receives input from its environment and, at times, issues output. The system under test can only be formally reasoned about if the assumption is made that a formal model of behavior can be extracted from the SUT. That is, if we are to show that the model and system conform, then the system needs to be expressible in a format that is compatible with the model. This formal model of the SUT is hypothesized to exist, but is not known a priori [41, 44].

This hypothesis allows us to reason about the system under test as if it were a formal model, and thus, to express the correctness of that system by establishing a relationship between the model and the SUT. Jan Tretmans laid much of the groundwork for this field by proposing a theory of test generation for labelled transition systems, based on establishing this *implementation relationship* (referred to as **ioco**) between the model and the system under test [42, 41].

Tretmans theory of implementation relations relies on the expression of the specification model and SUT as input-output transition systems (IOTS), a special form of labelled transition systems. A labelled transition system is a structure consisting of states with transitions, labelled with actions, between them [45, 46]. An IOTS is defined as a 4-tuple $(S,s_0,L,\rightarrow)$, where:

- S is a finite, non-empty set of states.
- $s_0 \in S$ is the initial state.
- L is a finite set of labels (actions), partitioned into input (L_I) and output (L_U) actions, with $L_I \cap L_U = \emptyset$. The actions in L are the observable actions of the system. A special label $\tau \notin L$ represents unobservable, internal action.
- $\rightarrow \subseteq (S \times (L \cup \{\tau\}) \times S)$ is the transition relation.

Note that the model format used in steering (described in Chapter 3) can be expressed as an IOTS by considering an assignment to the input alphabet as an input action and any new assignment to the output and visible state variables as an output action.

The reason for the explicit distinction between input and output actions is because input actions are issued to the system from the environment—they are under the control of the environment—and output actions are issued to the environment from the system—under control of the system. A system can never refuse to perform its input actions, and therefore, an IOTS must be able to accept all input actions in any of its states. Similarly, the environment of the IOTS can never block an output action. As a result, deadlock is not allowed to exist in IOTS systems [47]. It is always assumed that a system will *eventually* issue output. An IOTS supports a special form of state that cannot produce an output action, and that can only be exited through new input action. These states represent quiescence—when a system is waiting for new input to perform an action [47, 20]. The idea of quiescence is essential when modeling real-time or embedded systems, as such systems often exhibit quiescent behavior. By including these special states, quiescence can be treated as on observable event, and testing can ensure that such periods do not violate the system requirements.

A transition $(s, \mu, s') \in T$ can be denoted as $s \xrightarrow{\mu} s'$. A computation is a finite set of transitions: $s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} s_2 \xrightarrow{\mu_3} ... \xrightarrow{\mu_n} s_n$. A trace, σ, captures a sequence of observable actions (input and output) during a computation. It follows, then, that the correctness of the system under test sut, expressed as an IOTS, with respect to a model-based oracle $model$—also expressed as an IOTS—is given as the implementation relationship $ioco$, defined as:

$$sut\ ioco\ model = \forall \sigma \in traces(model)\ : out(sut\ after\ \sigma) \subseteq out(model\ after\ \sigma) \quad (4.1)$$

where $(sut\ after\ \sigma)$ denotes the set of states reachable from sut by performing the trace σ.

Informally, this implementation relation states that the system sut is correct with respect to the model-based oracle $model$ if, and only if, after all possible traces of $model$, any output action performed by sut can also be an output action of $model$. Demonstrating this relationship is the basis of much of the subsequent work on model-based testing [48, 49, 50].

Given a behavioral model, a test generation algorithm must produce a set of test cases that can be used to judge whether the model conforms to the specification according to this implementation relationship. It is argued that, to firmly establish this implementation relationship, the test suite must be sound—it must render a failure verdict only if the implementation is incorrect. Thus, the SUT is generally assumed to execute deterministically. If the SUT is not deterministic, then soundness cannot easily be established, as a failure verdict might be rendered when the system is acting correctly. In reality, there are many situations where real-time, embedded, and concurrent systems will act in a non-deterministic manner, complicating the process of model-based conformance testing. Much of the work on model-based testing relies on the assumption that the SUT is *less deterministic* than the model-based oracle [51, 52, 53]. In practice, this relationship is generally reversed, limiting the applicability of many model-based testing techniques.

Even if the execution environment does not introduce the unexpected non-determinism that we are concerned with, this assumption of determinism often prevents the model-based testing of systems with any form of real-time behavior, as such systems invariably exhibit non-deterministic behavior during *normal* execution. To get around this limitation, several authors have examined the use of behavioral models as test-generation targets for real-time systems by proposing special model formats and modified implementation relationships that support limited forms of non-determinism [54, 20, 3, 55, 47]. In these cases, the authors propose that soundness can still be achieved if the model explicitly accounts for the same range of non-deterministic behavior that the system exhibits. We propose essentially the same relationship. However, our approach attempts to modify the model during test execution to replicate that non-determinism, decoupling the model from the additional specification of non-determinism. Thus, our approach could theoretically be used to extend many of the existing approaches to model-based testing to handle the non-determinism witnessed during testing of the implemented system. We will examine several of these related approaches in the Sections 4.1.1 and 4.1.2, then discuss how our oracle steering approach differs in Section 4.1.3.

4.1.1 Models with Real-Valued Clocks

Few researchers have examined the practical difficulties of using models to test real-time or embedded systems and the type of issues that arise in such scenarios—such as probe effects from monitoring overhead, computation delays, and sensor inaccuracy [56]. However, many of these issues manifest by introducing non-determinism in the timing behavior of these systems, and a number of researchers *have* examined model-based testing of systems with real-time behavior [20, 3, 55, 47, 57, 58]. If such work can be used to verify the *expected* timing behaviors of a system, it can also theoretically be adapted to account for certain timing issues that introduce unexpected non-determinism into the system under test.

In order to perform verification of real-time systems, several different model formats have been proposed that allow the examination of timing behaviors. Informally, the proposed modeling formats tend to be automata structures similar to Tretmans' IOTS transition system with the addition of real-valued clock variables [59]. Such automata can remain in a particular state for a variable period of time, guided by timing constraints that guard the transitions.

While sophisticated verification of properties can be performed on non-deterministic models, establishing conformance between the model and SUT is still more complicated. When model-based oracles allow non-determinism, the generated test cases cannot form a linear trace, but instead form a tree adapted to the actions of the SUT [20]. This results in an exponential increase in the difficulty of establishing conformance, as the reachable state space at any given point in execution is massively increased over that of a deterministic model. Therefore, the majority of model-based testing approaches for real-time systems attempt to restrict the amount and type of allowed non-determinism [3, 60, 61]—restricting the use of clocks, clock resets, or timing-based guards.

As currently implemented, our approach begins with a deterministic model and introduces the ability to handle non-determinism through the steering process. The degree of allowed non-determinism is determined through the user-specified constraints. As steering progresses one test step at a time—rather than attempting to check the correctness of the entire trace at once—our approach should be able to account for a large variety of non-deterministic behaviors without suffering to the same extent from the branching problems that limit these model-based testing frameworks. In practice, however, as loose constraints will often lead to faults being covered, we expect users to limit the non-determinism that steering will allow (also easing the process of establishing conformance).

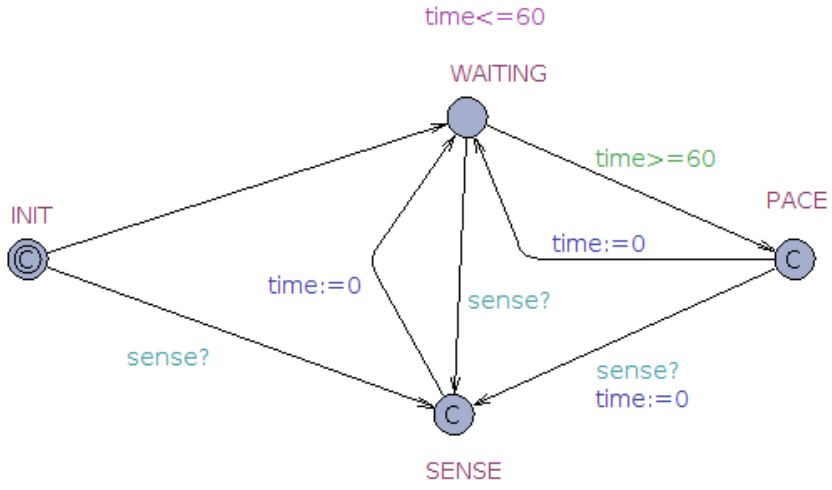

Figure 4.1: An Uppaal version of the SimplePacing model that delivers a pace deterministically after 60 seconds without a sensed heartbear.

Our hypothesis is that, regardless of strengths or weaknesses of the testing framework employed, developers would naturally restrict the allowed non-determinism when testing systems of the type we are primarily interested in. Non-determinism is not generally a desired property of a safety-critical system—in some cases, it is expected, but developers naturally will desire that a medical device or avionics system acts in a predictable manner. If there is risk of harm, then a system must be controlled. Although our approach could be used, in theory, to extend these non-deterministic models to handle the additional non-determinism introduced in the SUT during real-world execution, we would still expect the additional allowable non-determinism to be minimal.

UPPAAL-based Approaches

UPPAAL is a framework for modeling, simulation, and verification of real-time systems, developed as a collaboration between Aalbord University and Uppsala University[1] [62]. The framework can be used to model systems as collections of parallel non-deterministic processes—written as timed automata [59] with finite structures and real-valued clocks—that communicate

[1]Available for download from `http://www.uppaal.org/`

through channels and shared variables. UPPAAL is often used to model real-time parallel systems such as communication protocols. The widespread use of UPPAAL for verification of real-time systems [62, 63, 64, 65, 66] also makes it appealing as a format for testing those same type of systems [20, 57, 56].

UPPAAL models are written in a non-deterministic guarded transition language with support for data types including binary channels, bounded integers, and real valued clocks. A system is a network of timed automata, where the state of the system is defined by the location of all automata, the values of the shared clock variables, and the values of all of the discrete variables. Every automaton may transition or synchronize with another automaton, which leads to a new state. From a given state, an automaton may choose to take an action or a delay transition. Depending on the chosen delay, further actions may be forbidden. The network of automata may share clocks and actions.

An UPPAAL version of the SimplePacing system depicted in Figure 2.1 is shown in Figure 4.1. This version of the model is deterministic in the sense that, without heart input, it will deliver a pace every 60 seconds. This is accomplished through (1) the clock guard on the transition between "Waiting" and "Pace" allowing a pace *no sooner* than 60 seconds from the last clock reset and, (2), the clock invariant on the Waiting state forcing the transition to another state after *no more* than 60 seconds. This model, in its current form, is not useful for testing purposes if scenarios like those depicted in Figure 2.3 occur, causing output to occur after a non-deterministic timing delay in the SUT. If we want to account for this difference in the model, we must allow the model to act non-deterministically.

A version of the UPPAAL SimplePacing model with non-deterministic pacing is shown in Figure 4.2. In this version, the invariant on the Waiting state asserts that the model *must* transition to another state within 65 seconds of the last clock reset. This, combined with the guard on the transition to Pace, means that the model may deliver a pace anywhere between 60 and 65 seconds after the last sensed heartbeat. This allows us to account for a short timing delay in the SUT, while also defining the limit of what is to be accepted.

This simple model is still incomplete—it requires a source of input. Multiple branches are marked with the expression "`sense?`." This is what is known in UPPAAL as a channel synchronization. The SimplePacing automaton (and final system) listens for senses from a heart—a simulated heart during testing. The expression `sense?` indicates that this transition can be taken if a true value is broadcast to the listener over the "`sense`" channel. To perform

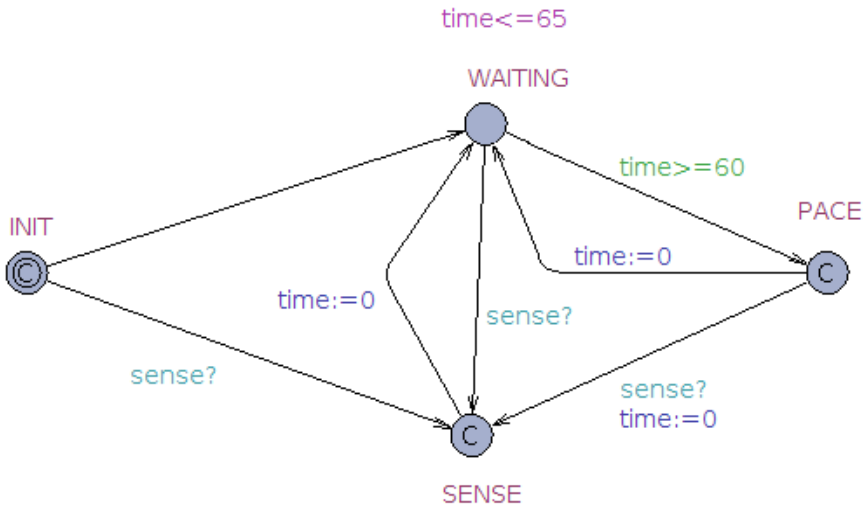

Figure 4.2: An Uppaal version of the SimplePacing model that delivers a pace non-deterministically between 60 and 65 seconds without a sensed heartbeat.

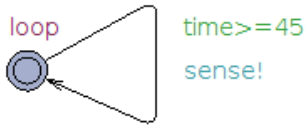

Figure 4.3: An simple timed automaton that sends a heartbeat to SimplePacing non-deterministically at any clock point more than 45 seconds after the last clock reset.

verification or testing, we need a source of input to that `sense?` listener. We can provide this by defining a second timed automaton that represents the heart. A simple heart model is pictures in Figure 4.3 that can send a sense at any point (chosen non-deterministically) as long as 45 seconds have elapsed since the last clock reset. This timed automata forms part of a network along with SimplePacing, sharing a clock and broadcasting `sense` events through the `sense!` channel synchronization. A common paradigm for verification in UPPAAL is to specify both the system and its environment using timed automata. Testing frameworks based on Uppaal can use a similar approach to provide test input to the model-based oracle or to constrain the non-determinism in the original system model to a limited form that reflects the

testing conditions [57].

This simple pacemaker example demonstrates that, in principle, timed automata models could be used to address a subset of the problems we are concerned about that introduce non-deterministic behavior into the system under test. In practice, there are several practical issues that make building the non-determinism induced by execution reality directly into the model unappealing. We will touch more on those in Section 4.1.3.

Larsen et al. proposed an approach for conformance testing of real-time systems using UPPAAL [20] (later extended in [57]). In their approach, given an abstract formal model of the behavior of the system written in UPPAAL, their test generation tool automatically explores the state-space of the model to generate test cases that can be executed against the SUT. Many model-based test generation frameworks execute tests against the model and SUT independently, perform the behavioral comparison *offline*. Their framework, UPPAAL-TRON, executes test *online*—combining test generation and execution. A single test input step is generated from the model at one time, which is then immediately executed againt the SUT. This process continues until an error is detected or the test generator decides to end the test. Online testing can be advantagous, as it allows for fewer constraints on the non-determinism expressed by the model. As discussed above, many model-based testing tools require that non-determinism be restricted while testing because of an exponential growth in the reachable state-space of the SUT. Online testing can automatically adapt to the current stage of execution, and—as a result—can reduce the state space to a manageable size.

UPPAAL-TRON replaces the environment of the SUT and stimulates and monitors the system. Based on timed sequences of input and output actions performed so far, it generates input to the SUT that is deemed relevant by the oracle model. Their approach—a generalization of the TORX framework [48] to real-time systems—continually computes the symbolic set of states that the model can occupy after the observations made so far. UPPAAL-TRON monitors the output of the SUT and checks conformance of the system output to the behavior specified in the model. Larsen et al. propose an implementation relation for establishing conformance that extends Tretmans ioco to include time-bounded quiescence. The inclusion of a more sophisticated quiescence model allows complex timing behaviors to be compared.

In order to generate tests and establish conformance, UPPAAL-TRON requires that the system model by combined with an additional environment model (similar to the SimplePacing and heart model example given above). The non-deterministic behavior of the system model is

constrained by the environment model to reflect the allowable behavior of the SUT in the environment it lives in. This idea is conceptually similar to our steering framework, but progresses in an opposite direction. Their assumption is that you have a model with large amount of freedom to select behavior non-deterministically, and that it should be constrained to match the expected environment. Our steering approach starts with a deterministic model (or, theoretically, with limited non-determinism) and transforms that to match the environmental reality.

Building an online testing framework does allow greater freedom in expressing timing-based non-determinism than many other model-based testing frameworks, but comes with its own limitations. Online testing requires keeping the SUT, test generation code, monitoring scaffolding, and the model-based oracle in lockstep during execution. This is a challenging task, and risks introducing probe effect—where the overhead from monitoring alters the behavior of the SUT. Online testing requires fast computation of expected behavior and the behavior comparison conducted by the test oracle, and therefore, can either limit the complexity of the model and test generation code, or limit the complexity of the SUT for which the approach can be applied. In practice, their testing approach did add latency and uncertainty to the timing of events, and after realizing this, they had to build an additional two second tolerance window into their case example [57].

Another interesting framework for model-based testing of real-time systems based on UPPAAL's timed automata was introduced by Rachel Cardell-Oliver [56]. Because of the non-deterministic nature of real-time systems, and because test infrastructure and monitoring overhead may intoduce additional timing non-determinism, Cardell-Oliver—like us—contends that it is not realistic to expect an exact match between the behavior of the SUT and traditional test oracles. Rather than generating tests from a model of the system, Cardell-Oliver's approach is based on expressing abstract test specifications as UPPAAL timed automata. These test specifications are transformed into program code for executing concrete tests and checking whether the SUT conforms to the expected behavior encoded into the test specification model.

Traditional test generation frameworks specify a test case as a particular series of inputs and outputs. Non-deterministic behavior makes this impractical. Instead, Cardell-Oliver's framework specifies input and output behaviors as parallel processes and uses program clocks and additional offline analysis to check whether the behavior exhibited by the SUT satisfies both the functional and timing behaviors built into the test specification. Because their tests are hand-crafted, they can tune a test to one particular function of the system—perhaps enabling them to

more easily account for the non-determinism introduced to *that function* by the environment or hardware platform. Their approach even allows for interactive execution, where a human operator provides input to the system (a major source of timing non-determinism—humans can rarely perform a task multiple times with identical timing). However, because each test must be hand-built, their approach suffers from a severe manual limitation that is addressed by frameworks that make use of a model of the system and generate tests from that model.

Other Timed Automata

Briones and Brinksma have presented their own extension to Tretmans' ioco implementation relation for real-time systems, along with a test generation algorithm [47]. Their extension, which is intended to be more general than Larsen et al.'s UPPAAL-based approach [20, 57], is based on an operational interpretation of the notion of quiescence. By treating quiescence as a special form of output, behavioral traces can include observations of these periods. This means that implementation relations—such as ioco—can be extended to account for non-deterministic timing of output behavior as long as the model-based oracle can also produce the same quiescent period. This interpretation of quiescence gives rise to a family of possible implementation relations parameterized by observation durations for the quiescent period.

The authors introduce a version of Tretmans' IOTS automata for real-time systems, called a Timed Input-Output Transition System (TIOTS). Real-valued timing properties are added to the model through the notion of *time-passage actions*—actions that take time. All other actions are still considered to take place instantaneously. Transitions must also satisfy certain timing properties—for example, if two different transitions exist from one state to another, they must take the same amount of time to perform. If input arrives, then it must be accepted immediately without additional timing delays. This retains Tretmans' requirement for input enabledness, but may not reflect all real-world execution scenarios. In contrast to Tretmans' untimed case, time delays can change system state without the application of new input. This allows the testing of real-time systems that can demonstrate the same behavioral phenomena. In order to demonstrate conformance in real-world testing, all timing-related behaviors must have a maximum quiescence period. Their test-generation framework generates tests randomly, by recursively choosing one of three test actions: (1) termination, (2) generation of an input, or (3), observation of output (or quiescence). This test generation approach is sound with respect to the set of implementation relations.

En-Nouaary et al. have also introduced a model-based testing framework for real-time systems [3]. Their approach is based off of models specified as Timed Input Output Automata (TIOA)—a format that is, like UPPAAL, dervied from timed automata [59]. TIOA, like the UPPAAL model format, models systems using automata with shared, real-valued clocks that increase synchronously at the same speed until reset. Models expressed as TIOA are non-deterministic, and this makes establishing conformance challenging. In order to test the complex implementations, their approach is based on using state characterization to partially determinize the model. Their approach samples the state space of the model at a user-specified level of granularity and constructs a testable subautomaton (called the Grid Automaton). The Grid Automaton is then transfomed into a special timed finite state machine, that is fed to a test generation algortihm. Test generation is performed through the Wp-Method, a common algorithm for test generation using finite state machines [67].

The testing framework presented by En-Nouaary et al. is potentially of limited applicability to handling the non-determinism introduced to the SUT during real-world execution. In order to operate efficiently, their framework places strict limitations on the non-determinism that can be modeled—limiting the number of active clocks, the scale of the non-determinism, and the size of the state space of the SUT. However, what makes their work relevant is the authors detailed examination of the faults expected from real-time systems.

Their fault model makes a distinction between timing-based faults and the traditional action and transition faults [68]. Timing faults are further distinguished as *effective* and *non-effective* faults, where effective faults are timing issues that impact the execution of the system. Timing faults are generalized to one of three issues:

- Clock Reset Faults: When clocks fail to be reset in the SUT when they are in the model, or if the SUT resets a clock that is not reset in the model.
- Time Constraint Restriction Faults: When the SUT rejects input satisfying a time constraint that is accepted by the model.
- Time Constraint Widening Faults: When the SUT increases the upper bound or decreases the lower bound on a timing constraint.

The timing non-determinism added to the implementation systems in our case study were added manually after considering potential problems seen in previous real-world applications. It would

be ideal, in future work, to create an approach to automated seeding of timing faults—similar to our automated approach to seeding functional faults (see Section 5.2). En-Nouaary's categorization of timing faults could be used as the basis for the automated seeding of timing faults for future experimentation with steering and other testing approaches to real-time systems.

4.1.2 Other Model-Based Approaches

Savor and Seviora contributed one of the earliest research efforts on developing model-based oracles for real-time systems [55]. Their approach is based on monitoring the execution of the system. A supervisor model—a finite state machine derived from the requirements—is fed the same inputs that the SUT receives. An expected behavior is returned from the model, which is added to an expected behavior buffer. The observed SUT behavior is added to an observed behavior buffer. Over time, a matching system compares expected and observed behaviors from these buffers. The use of these two buffers allows for varying complexity in the behavior comparison procedure. There is a trade-off in the computational complexity of monitoring the SUT and the latency of failure reporting. Thus, their monitors support both online (results checked during execution) and offline testing (results buffered and compared at the speed that the speed that they can be computed at). If failure detection must be reported immediately after the error appears, then the oracle model cannot be complex—it should be as deterministic as possible. If failures can be checked out of step with the system execution, then more sophisticated models can be employed.

Savor and Seviora briefly discuss the dangers of non-determinism. In their modeling notation, a network of processes grouped in a block communicate through zero-delay signal routes—communication takes place instantaneously—while processes in different blocks communicate through indeterminate-delay channels. These channels are a major source of non-determinism, occurring when signal routes merge. Their modeling notation—SDL—was primarly designed for modeling of communication protocols [69], and the authors primarly focus on non-determinism in process communication. The authors advise that the model employed within the oracle must be able to consider all legal behaviors allowed under non-determinism in the specification, but warn that such non-determinism could result in significant oracle time and space complexity. To preserve ordering for out-of-time comparisons, signals are appended with an occurrence interval that gives an interval during which inputs to or outputs from the SUT could have taken place. Their monitor uses this information to construct legal orderings of

events, and a belief-creation algorithm filters these orderings and constructs hypotheses about legitimate observable behaviors of the SUT.

Arcuri et al. have proposed a framework for testing real-time systems that explicitly takes the environment into account [54]. Their approach actually does not model the system at all, but instead models the environment as a network of parallel state machines, and uses those state machines for test generation and as oracles during test execution. The rationale for this approach is that, because the environment provides input and reacts to the output of systems, then effective testing relies more on an understanding of how the environment of a system works. By modeling the environment, Arcuri et al. argue that testing can still capture the correct behavior of the system while better accounting for unexpected environmental influence on the execution of the system.

Their approach allows system testers—who may not know the system design, but do know the application domain—to model the environment for test automation. These models are then used for code generation of an environment simulator, selection of test cases, and evaluation of the results. In their work, executing a test case is actually an execution of the environment simulator. The models then serve as test oracles during execution of the environment simulation. Within these environmental models, there are error states that should never be reached during execution of a test case. These error states reflect unsafe, undesirable, or illegal states in the environment. Their environment simulator can model certain forms of non-determinism. For example, a timeout transition could be triggered within a minimum and maximum time value. This allows the modeling of real-world scenarios where there is a natural variance when time-related behaviors are represented. Probabilities can also be assigned in the model to represent failure scenarios such as hardware breakdown. In the authors' framework, the input data for a test case includes the choice of actual values to use in these non-deterministic events. In their models, non-deterministic situations can only occur in the transitions between states. As such transitions can take place multiple times, for each instance of the state machines, for each non-deterministic choice, they define in the test case a range of possible values. Thus, the values chosen for the simulated non-determinism may not reflect the same non-determinism seen in the real world. However, this partial separation of non-determinstic scenarios from the chosen likelihood of occurrence allows for easier adjustments to match the real-world observations than some of the other modeling formats examined previously.

SpecExplorer is a framework for testing reactive, object-oriented software systems, developed by Microsoft Research [10]. This framework differs from many of the other frameworks discussed because it is primarily intended for testing. The other approaches take models and tools used for verification and repurpose them to be used in testing. This difference allows SpecExplorer to be used with models that would otherwise be too large and complex to be utilized in formal verification.

The inputs and outputs of object-oriented systems can be abstractly viewed as parameterized action labels. Thus, in SpecExplorer, state transitions can take the form of method invocations, with object instances and complex data structures as input and return values. From the testers perspective, the SUT is controlled by invoking methods on objects and monitored by observing invocations of other methods. Because the state space of a model may be infinitely large in an object-oriented framework, SpecExplorer reduces this space during test generation by separating the description of the model state space and finitizations provided by user scenarios and earlier steps in the test execution.

SpecExplorer is designed to incorporate non-determinism—anticipating issues related to parallel process communication and timing delays in output. The framework handles non-determinism by distinguishing between controllable actions invoked by the tester and observable actions that are outside of the testers control. While SpecExplorer could be used to account for some

What makes SpecExplorer relevant to our oracle steering framework is that SpecExplorer makes use of a form of model steering during the test generation process. These steering actions are used to selectively explore the state transitions of the model, resulting in test cases that are specialized for various goals that the user wants to achieve. The model is executed by choosing input actions, guided by user-specified testing scenarios. The user-controlled methods include:

- Parameter Selection: Testers can write expressions that constrain the selection of input method invocations and parameter values.
- Method Restriction: Testers can require that certain preconditions be met before an action is taken. For example, in a chat application, method restriction could prevent messages from being sent until all clients have been created and have entered the chat session.
- State Filtering: Testers can use filters to avoid certain states. In a chat application, a state filter could prevent the same message from being posted more than once.

- Directed Search: Weights can be applied to states to help direct test execution. Additionally, tests can be steered to achieve certain user goals, such as covering a minimum number of transitions.

Although Spec Explorer also makes use of steering to guide the execution of behavioral models, their application and goals differ from ours. They use steering to create tests, but the final test cases are effectively deterministic. Steering is not applied when checking conformance. As with the other approaches to model-based testing discussed in this chapter, SpecExplorer may be able to address some of the issues we are concerned with. However, any allowable non-conformance must be anticipated during test creation. Thus, their framework suffers from the same limitations that we will discuss in Section 4.1.3.

4.1.3 Comparison to Oracle Steering

By incorporating a sophisticated model of time or other forms of non-determinism, several of the modeling frameworks examined in this chapter could account for a subset of the non-deterministic behaviors induced during real-world execution of the system under test—particularly variance related to timing issues. For example, if a computation delay can result in a pace being delivered off-schedule in a pacemaker being tested, than a timed automata could be crafted as an oracle that allows that pace to occur at any point within a bounded window of time. This is a powerful addition to standard state-transition systems, and such models *could*—in theory—account for some of the same problems that steering accounts for. However, that power comes at a practical cost—the model *must* be built with those execution behaviors in mind.

There is a difference between building a model to account for *planned* non-deterministic behavior detailed in the system requirements and building a model that also accounts for possible non-determinism introduced to the SUT to use in conformance testing. If the system is expected to demonstrate some level of non-determinism in its functional behavior, then this non-determinism can be planned for and modeled. As such variation is planned for, then it is relatively unlikely that the model will need to change when testing the completed implementation. If the implementation doesn't conform to the model, then the requirements are not being satisfied. Tests that fail to demonstrate conformance—in the absence of the additional non-determinism that we are concerned with—indicate defective behavior in the system.

In contrast, the situations that we are concerned with will change the behavior of the system. That behavior may still satisfy the requirements, but as it does not conform to what the model predicts should happen, the test will fail. These situations can be deterministic or non-deterministic, but in either case, they depend specifically on the particular details of the real-world execution. Such details are difficult to anticipate, and decisions need to be made at the time that the model is constructed. If the model is built—as many of them are—during the requirements engineering phase of development, then it is difficult, if not impossible, to make the correct assumptions. Many of the behavior differences between model and SUT induced by non-determinism depend on the hardware being employed and the sophistication of the code written for the implementation.

If such assumptions are made and end up being incorrect, then making changes to the model may require a complete overhaul of the models structure—a potentially arduous task. Consider the simple UPPAAL model in Figure 4.2. In this case, a desire to allow a six second window on pacing would require changing the invariant on the state "Waiting" to $time <= 66$. In a model this simple, this is not a painful task. However, in a more complex model, making such changes might require adjusting many different invariants on states and guards on transitions. These invariants and guards may have been designed to model the combined product of multiple timing-based behaviors, and adjusting those would require clear understanding of how all of those behaviors and the non-determinism introduced by the SUT's environment interact. This task may need to be performed on a regular basis as the system hardware or software evolves. The alternative is to wait until the implementation is ready to be tested to build the model-based oracle. This is also an inefficient outcome if the model would be useful for early-lifecycle tasks such as requirements analysis. A large degree of manual effort is required when modeling. The ability to reuse a model built for requirements analysis during testing would be welcome.

In addition to the potential difficulty in adjusting the model to account for reality-induced non-conformance, there is an argument to be made for not building these details directly into the model in the first place. Being forced to incorporate the reality-induced non-deterministic behavior of the SUT into a model that is being built during requirements engineering may distract from the core purpose of the model at that time—to analyze the functional behavior of the system to be built. Effective requirements analysis requires *clarity*. This is why Miller at al. have found that engineers are more comfortable with building deterministic models—particularly in visual formats—over mathematical or non-deterministic models [70]. Such models are easy to

build, easy to read and understand, and can be used to explain the system behavior to other engineers, users, and management without requiring a deep understanding of the system.

This is not an argument against using non-deterministic models—timed automata can, and should, be used in cases where their real-valued clocks and timing model are needed to verify the correctness of the core functionality of the system requirements for systems with complex time-dependent behavior. Rather, there is an argument to be made that being forced to *also* take into account non-determinism induced by the real-world environment could muddle the clarity of the model, making the analysis of the requirements more difficult and potentially leading to an incorrect implementation. Abstraction allows engineers to reason about requirements and system properties in a clear manner, then to clearly explain those properties of the system to others. Such models can be reused to generate tests intended to cover certain system states and behaviors, and in some cases, even to generate source code. This is a large part of the appeal of frameworks such as Simulink and Stateflow. Even if a more complex non-deterministic model, built in a framework such as UPPAAL, is needed to analyze functional behavior, allowing engineers to abstract away execution details will likely lead to clarity in requirements analysis and a more stable development process.

Through the use of a specification of the tolerance constraints, we effectively decouple the model from the rules governing conformance. This decoupling makes non-determinism implicit and the approach more generally applicable. Explicitly specified non-deterministic behavior—as required by the model-based approaches described above—would limit the scope of non-determinism handled by the oracle to what has been planned for by the developer and subsequently modeled. It is difficult to anticipate the non-determinism resulting from deploying software on a hardware platform, and, thus, such models will likely undergo several revisions during development.

Steering instead relies on a set of rule-based constraints that may be easier to revise over time. If the assumptions made on the non-conformance introduced in the real-world prove to be incorrect, or the developers wish to impose different limitations on what is considered to be correct execution, then changing that criteria is as easy as writing a new set of tolerance constraints and using those when steering. The model itself rarely needs to be changed directly. Instead, the steering process can automatically use the constraint set to adjust the model during execution. Therefore, steering allows clarity to be retained during modeling, and allows the reuse of models for multiple development purposes.

Additionally, by not relying on a specific modeling format, steering can be made to work with models created for a variety of purposes. By not being tied to a specific test generation framework, we can make use of tests from a variety of tools, or more easily build steering into a number of existing frameworks. Many of the existing approaches to model-based testing propose both a modeling format and specific test generation framework. This is because their conformance relationship requires a compatible model. In contrast, the intent of our approach is more general. Our steering approach requires that tests exist, but imposes no requirements on the source of such tests. We take non-determinism into account explicitly during the test execution stage when we attempt to steer the model. By exploring non-determinism during execution, our approach allows more freedom in the generation of tests. In fact, our approach should be compatible with many of the frameworks discussed above for testing real-time systems. Regardless of the source of model or tests, if the model and system fail to conform when the system is still meeting its requirements, these model-based testing frameworks could incorporate a steering step during test execution to expand the behaviors allowed without making changes to the fundamental structure of the model. In many situations, this could be a preferable alternative to editing the model itself.

4.2 Program Steering

Program steering is the capacity to control the execution of a program as it progresses in order to improve performance, reliability, or correctness [22]. Program steering typically comes in two forms—*dynamic*, or automatic steering, and *interactive*, where a human operator steers the program through some sort of interface [9]. The majority of research in dynamic program steering is concerned with automatic adaptation to maintain consistent performance or a certain reliability level when faced with depleted computational resources [22]. The use of steering to adjust program values is typically left to the domain of interactive program steering; however, there is some work of interest in this area as applied to software testing and monitoring. For example, Kannan et al. proposed a steering-based framework to assure the correctness of software execution at runtime [71].

The goal of their framework is to serve as a supervisor on program execution, checking observed behaviors against certain properties (this can be thought of as a form of invariant-based oracle), and adjusting the behavior of the program if a property is violated. Their framework

is an attempt to bridge formal model verification and testing. Verification is often infeasible on large systems (without greatly weakening the model), and as it is based on formal models, does not always ensure the correctness of a particular implementation. By incorporating the requirements-based properties used in formal verification into the execution monitor and steering, an attempt can be made to force the system to conform to its requirements. Their framework accomplishes steering through the use of three scripting languages. These languages specify what to observe from the running program, the requirements that the program should satisfy, and how to steer the program to a safe state when it fails to conform to these requirements. They use multiple languages in order to maintain a clear separation between the implementation-specific description of monitored objects and the high-level requirements specification.

The framework attempts to steer the SUT when a deviation is observed between the system execution and the requirements-based properties. The requirements of the system are expressed in terms of a sequence of abstract events (or trace). A monitoring script describes the mapping from observations to abstract events. A monitor uses this script to decide what and how to take readings from the SUT to extract abstract events. A checker verifies the sequence of abstract events with respect to the requirements specification, detects violations, and generates a "meta-event" as a result. A steerer uses the sequence of meta-events to decide how to adjust the system dynamically to a safe state through control events. Steering is conducted through the tuning of a small set of parameters. After a violation is detected, steering actions alter the state of the system until the violated properties are no longer false. This steering is done under the assumption that the SUT is mostly correct, and that only minimal control should be exercised. It should be noted that their use of steering is not necessarily to influence the testing process (if a property is violated during testing, it should be investigated whether or not it can be corrected), but is primarily intended for use after a system has been deployed to help correct for unexpected deviations from correct behavior.

Although their system bears similarities to what we are proposing, our goals are very different—we are not attempting to correct system behavior. We simply wish to identify the deviant behavior. Our goal, rather than to apply program steering, is to steer the oracle. This difference in goals and implementation frameworks also leads to a differences in the proposed techniques, which we will discuss further in Section 4.2.1.

Lin and Ernst proposed the steering of multi-mode systems—systems that base operational

parameters on a series of discrete modes, such as power management systems on laptops—making use of steering actions that force mode changes when the system is faced with unforeseen inputs and operating scenarios [72]. In addition to ensuring that defects in the system are corrected, the authors propose using the testing process to *train* their program steering method. The proposed mode selector examines data from a series of successful tests and derives relationships between operation modes and the situations in which those modes should be applied. In new situations where the system is either failing or under-performing, the mode selector can override the system's choice of operational mode with a new mode that the selector determined would better fit the examined scenario. This mode selector is a form of fault-tolerance, the practice of programs gracefully adjusting their behavior once a fault or failure has been detected [73]. Their approach could, in theory, be used to augment a system model or the oracle steering process rather than the actual system under test. For instance, we could use their learning approach to "cache" common steering actions. Then, rather than searching for a candidate steering action across the entire state space, steering could be sped up by first examining "common" steering actions.

4.2.1 Comparison to Oracle Steering

Program steering has been used in the past to correct the behavior of systems deemed to have deviated from their expected behavior [71]. In situations where a model-based oracle and the system fail to conform, it may be possible to steer the system to conform to the model. However, instead of attempting this, we attempt to steer the model to match the system under test. While steering of the oracle model is risky—the model is intended to serve as our idea of correctness—we conduct model steering for two reasons: for finer control of the steering process, and importantly, to better account for the source of the deviation.

In the first case, direct monitoring and steering of the system under test is not always possible in embedded or real-time systems. Monitoring the behavior of software is known to introduce a *probe effect*, where the computational overhead from monitoring introduces timing delays into the behavior of the system. In a time-dependent system, or a system with limited computational resources, this probe effect can cause erroneous behavior. If monitoring alone can cause a system to perform incorrectly, the computational cost of both monitoring and steering an embedded or real-time system can be prohibitively high. Therefore, the applicability of program steering techniques on such systems is limited.

More importantly, however, is the source of the deviations between the model and the system. These deviations result not necessarily from the incorrect functioning of the system, but from a disagreement between the idealized world of the model and the reality of the execution of the system under test. In cases where the system is actually acting incorrectly, we don't want to steer at all—we want to issue a failure verdict so that the developer can change the implementation. In many of these deviations, however, it is not the system that is incorrect. If the model does not account for the real-world execution of the SUT, then *the model is the artifact that is incorrect*. Therefore, we can take inspriation from program steering for this situation—rather than immediately issuing a failure verdict, we can attempt to correct the behavior of the model.

4.3 Search-Based Software Testing

In its current form, the process of oracle steering is formulated as a search problem. Given a set of constraints and an objective function (i.e., the state comparison performed by the dissimilarity function), we search for the optimal steering action. The field of search-based software engineering [74] is full of examples of such problems, particularly in the area of software testing [75, 76, 77, 78]. Search methods have been applied to a wide variety of testing goals including structural [79], functional [80], non-functional [81] and state-based testing [82]. Search-based approaches have even been used to help support oracle creation [31, 2].

The search process that we currently use to perform oracle steering is a form of bounded model checking. Model checking is an automatic approach to property verification for concurrent, reactive systems [28]. This process starts with a model described by the user (generally either a transition-based model or a format from which a transition-based model can be synthesized from) and discovers whether hypotheses asserted by the user are valid on the model. If such hypotheses can be violated, the model checker produces a counter-example showing a set of transitions that cause the violation. This process is at the heart of most verification frameworks, such as UPPAAL [62]. Properties are asserted, and a search algorithm attempts to find a violation of that property.

Model checking is based on the concept of temporal logic—that a formula is not statically true or false in a model [28]. Rather, a formula may be true in some states and false in others. Bounded model checking is a specific form of model checking where, given a transition-based model M, a temporal logic formula f, and a user-supplied time bound k, we can construct

a propositional formula that is satisfiable if and only if f is satisfiable over a transition path of length k. The oracle steering problem can naturally be seen as a form of bounded model checking. We have a series of constraints (the user-specified tolerances), and seek a *single* transition that brings the model to a state that satisfies those constraints. This is a bounded path of length 1[2].

A class of search algorithms known as SAT solvers often form the core of a bounded model checking approach [83]. Satisfiability, or SAT for short, is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. Satisfiability establishes if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to true [84, 85]. SAT is an NP-complete problem—no known polynomial-time algorithm exists that can solve all instances of SAT, and it is likely that no such algorithm can exist [86]. However, certain algorithms—called SAT solvers—can find solutions to a large enough subset of SAT instances to be useful for purposes of model analysis.

Many optimization problems can be transformed into instances of SAT [84], or more generally, into instances of SMT (satisfiability modulo theories). An SMT instance is a generalization of a SAT instance in which variables are replaced by predicates from a variety of underlying theories. SMT formulas provide a richer language than is possible with standard SAT formulas. Our problem of oracle steering can also be thought of in the framework of SMT—we have a set of constraints on what variables can be steered and how they can be transformed, and the additional constraint that the output of a candidate steering target must match the behavior of the target system more closely than the original state reached in the oracle. These constraints can be expressed in conjunctive normal form (CNF) as a SMT problem.

Search algorithms, including those that address SAT/SMT problems, typically come in two different forms—complete and metaheuristic methods [87]. Complete methods are exhaustive searches that guarantee an optimal solution if there exists one (these searches do not try all possible configurations, they instead use pruning methods to guide the search process). Metaheuristic search methods use different techniques to sample the search space and report the best observed result. Metaheuristic search methods tend to be much faster than complete searches and scale to larger problems, but do not come with a guarantee of optimality.

[2]we could make use of a longer path length if we wanted to allow the model a series to actions when moving into conformance with the system, but for testing purposes, a shorter path prevents steering from covering for faults.

4.3.1 Complete Search

One example of a complete method is the branch-and-bound algorithm [88]. This algorithm is conceptually simple: set a literal in the boolean formula to a particular value, apply that value to the formula, and check to see if the value satisfies all of the clauses that it appears in. If so, assign a value to the next variable. However, if setting a value unsatisfies a clause, then a backtracking step (a *bound*) is initiated and the other possible value is applied. This process prunes branches of the formed boolean decision tree. Consider the following example in CNF (from [87]):

$$f = (\neg x_2 \vee x_5) \wedge (x_1 \vee \neg x_3 \vee x_4) \wedge (x_4 \vee \neg x_5) \wedge (x_1 \vee x_2) \tag{4.2}$$

We first set a value of zero to x_1. This inserts a zero into clauses two and four, but does not satisfy or unsatisfy either clause yet. Next, we insert a value of zero for x_2. This satisfies the first clause, but unsatisfies the fourth clause (as both x_1 and x_2 are set to zero). Therefore, we stop and backtrack, assigning a new value of one to x_2. This satisfies the fourth clause. We can continue this process with all variables until the complete formula is satisfied.

Another common complete method is the Davis-Putnam-Logemann-Loveland, or DPLL, algorithm [89]. DPLL is conceptually similar to the branch-and-bound method, but a few key differences give it an edge on a number of SAT problems. Like with branch-and-bound, DPLL begins by selecting a variable and applying a value to it. If this value satisifies the clause, then all clauses containing that variable are removed from the formula. If the variable is made false due to negation, the algorithm instead remove that variable from only the cause that it is negated in. This process is repeated recursively until a solution is found. This induces a domino effect—as more variables are removed from clauses, more clauses turn into unit clauses. Again, we can consider the following example:

$$f = (\neg x_2 \vee x_5) \wedge (x_1 \vee \neg x_3 \vee x_4) \wedge (x_4 \vee \neg x_5) \wedge (x_1 \vee x_2) \tag{4.3}$$

If we assign a value of zero to x_2, the first clause is rendered true ($\neg x_2 = \neg 0 = 1$). We can eliminate the first cause from the formula and x_2 from clause four. This leaves the following:

$$f = (x_1 \vee \neg x_3 \vee x_4) \wedge (x_4 \vee \neg x_5) \wedge (x_1) \tag{4.4}$$

As the third clause is now a unit clause, we assign $x_1 = 1$. We can now remove both clauses one and three from the formula:

$$f = (x_4 \vee \neg x_5) \tag{4.5}$$

From this point, the example is trivially solved with $x_4 = 0$ and $x_5 = 0$.

A common approach of complete search in testing is the automated generation of test cases [90]. In software testing, the need to determine the adequacy of test suites has motivated the development of several classes of test coverage criteria [91]. For example, structural coverage criteria measure test suite adequacy using the coverage over the structural elements of the system under test, such as statements or control flow branches. Expressions related to how those structural elements need to be exercised can be encoded as clauses to be satisfied by the search algorithm.

In this case, how the complete search works is that each test obligation—an expression describing how to exercise a structural element of the SUT—is encoded into the SMT instance. These obligations are expressed in a negative form, called a *trap property* [92]. For example, we might assert that "We can never exercise this branch or statement." The solver then attempt to violate this property by finding a series of inputs to the SUT that will exercise that element in the manner that we have specified. Through this process, the search algorithm can improve coverage and reduce the cost associated with test creation [93].

The current implementation of oracle steering makes use of Microsoft Research's Z3 theorem prover [30]. Z3's SMT solver is based on the DPLL algorithm, with additional specialized solvers that handle equalities and uninterpreted functions, arithmetic operations, and quantifiers.

In order to choose a steering action, we encode the following into the SMT instance: (1) the tolerance constraints, and (2) a dissimilarity goal. As a complete search will always find a solution if it exists (unless the state space is too large to explore), rather than just accepting any dissimilarity score that is better than the original calculation, we encode a direct goal—i.e., a dissimilarity score of 0. If that cannot be matched, we can achieve partial conformance by encoding progressive larger scores as goals until we find a solution that matches that goal.

To find the steering action, we encode these constraints as trap properties. For example, we might assert that *the dissimilarity score can not be 0*. Then, we ask the solver to find a solution that violates that assertion within a bounded number of transitions. If a state that can be found that gives a score of 0, we know that the set of input that transitions the model to that state is

the steering action that we want to choose.

4.3.2 Metaheuristic Search

Incomplete methods use a heuristic-based search to quickly find a near-optimal solution [87]. Metaheuristic search methods are fast and can efficiently solve larger search problems than complete methods, but do not carry the same guarantee of optimality.

Oracle steering is currently based on complete search. A complete search was chosen to help choose optimal steering actions, but this choice places limitations on the size of the state space of the models we steer, the speed of the approach, and the complexity of the dissimilarity metric. Therefore, in the future, we would like to consider the use of metaheuristic search in oracle steering. The use of a metahueristic method would come at the potential cost in optimality, but could enable:

- Larger models. This is particularly useful if we want to consider steering over multiple test steps at once.
- Complex dissimilarity metrics. Models may require careful choice of a steering action, and this may require comparisons of weighted variables, strings, and other mathematically complex expressions that cannot easily be evaluated as part of a complete search.
- Online steering. Currently, steering takes place as part of offline testing. However, a fast search technique may enable the use of steering during live execution of a system.

The cornerstone of metaheuristic search methods is the *objective function*, a numeric metric used to score the optimality of observed solutions [94]. According to Clark and Harman [74, 94], there are four key properties that should be met for a metaheuristic approach to be successfully applied to a software engineering problem:

1. **A Large Search Space:** If there are only a small number of factors to compare, there is no need for a search-based approach. This is rarely the case, as software engineering typically deals in incredibly large search spaces.
2. **Low Computational Complexity:** If the first property is met, search algorithms must sample a non-trivial population. A typical run of one of these searches requires thousands of executions of a score evaluation. Therefore, the computational complexity of the evaluation method has a major impact on the overall search.

3. **Approximate Continuity of the Objective Function:** While it is not necessary for an objective function to be continuous, too much discontinuity can mislead a search. Any search-based optimization must rely on an objective function for guidance, and continuity will ensure that such guidance is accurate.

4. **No Known Optimal Solutions:** If an optimal solution to a problem is already known, then there exists no need to apply search techniques.

All four of these characteristics are prevalent in software engineering problems [95]—particularly properties 1 and 4—and our problem of oracle steering potentially meets all of these conditions. The set of reachable states in a model can be exponentially large. Yet, representations such as ordered binary decision diagrams [96] make computing this set computationally feasible and the comparison of a candidate state to the system state can be inexpensive to compute. In the case of oracle steering, the dissimilarity function is a natural choice for an objective function. Almost all possible options for a dissimilarity function are continuous numeric functions [26]. Finally, we do not know the optimal steering action in the vast majority of situations.

Numerous algorithms are used to conduct search-based software engineering experiments, including hill climbers [77], local searches such as simulated annealing [97] and tabu search [98, 99], generic and evolutionary algorithms [100, 101], and swarm optimization [102], among others.

Rather than asserting a dissimilarity goal—as was the case when using a complete method—we could use the dissimilarity score as an objective function. We could sample the search space for candidate solutions that meet the tolerance constraints, then accept the solution found with the lowest dissimilarity score.

Metaheuristic search techniques have been used for SAT/SMT problems as well. One example of a SAT solver using a metaheuristic search is the WalkSAT algorithm [103]. WalkSAT is a greedy hill-climbing search that starts by assigning a random value to each variable in the boolean formula. If this assignment satisfies all clauses, the algorithm terminates and returns the assignment. Otherwise, WalkSAT randomly picks a clause that is unsatisfied by the current assignment and flips the value of a variable to a value that will satisfy that clause. However, at a certain probability level, WalkSAT will instead pick a variable at random to flip from the whole formula. The occasional random flip prevents WalkSAT from becoming stuck in local plateaus

in the search space. If no solution is found within a certain number of flips, the algorithm will restart with a new random assignment.

Like with a complete search, a metaheuristic algorithm such as WalkSAT could be used to produce a suite of test cases that achieve some form of structural coverage. The set of test obligations for, say, branch coverage could be encoded into a conjunctive normal form statement. Then, the WalkSAT algorithm could try to find a solution that satisfies the largest number of obligations (clauses). In this case, our objective function could be the number of satisfied obligations [77]. We choose tests that cover a large percentage of the branches, statements, or conditions of a program. Another common objective function for structural coverage is the *percentage of additional coverage* added by a test. Thus, we would choose the test that covers the largest number of obligations not yet covered by another test.

Chapter 5

Experimental Studies

We aim to assess the capabilities of oracle steering and the impact it has on the testing process—both positive and negative. Thus, we pose the following research questions:

1. To what degree does steering lessen behavioral differences that are legal under the system requirements?
2. To what degree does steering mask behavioral differences that fail to conform to the requirements?
3. Are there situations where a filtering mechanism is more appropriate than actively steering the oracle, and vice-versa?

As we suspect that the success of the steering process is, at least partially, dependent on the quality of the constraints imposed, we also wish to explore the impact of different sets of steering constraints:

4. To what degree does the strictness of the employed tolerance constraints impact the correctness of the steered oracle verdict?
5. How accurate are tolerance constraints learned automatically from previously steered test execution traces?

5.1 Experimental Setup Overview

Our case study centers around models of two industrial-scale real-time medical device systems. The first system is the management subsystem of a generic Patient-Controlled Analgesia (GPCA) infusion pump [104]. This subsystem takes in a prescription for a drug—as well as several sensor values—and determines the appropriate dosage of the drug to be administered to a patient each second over a given period of time.

The second system is based on the pacing subsystem of an implanted pacemaker, built from the requirements document provided to the Pacemaker Challenge [105]. This subsystem monitors the patient's cardiac activity and, at appropriate times, commands the pacemaker to provide electrical impulses to the appropriate chamber of the heart.

These models, developed in the Simulink and Stateflow notations and translated into the Lustre synchronous programming language [106], are complex real-time system of the type common in the medical device domain. Details on the size of the Simulink model and the number of lines of code in the translated Lustre code are provided in Table 5.1.

	# States	**# Transitions**	**Lustre LOC**
Infusion_Mgr	23	50	6299
Pacing	48	120	24017

Table 5.1: Case Example Information

To evaluate the performance of oracle steering, we performed the following for each system:

1. **Generated system implementations:** We approximated the behavioral differences expected from systems running on real embedded hardware by creating alternate versions of each model with non-deterministic timing elements. We also generated 50 mutated versions of both the oracle and each "SUT" with seeded faults (Section 5.2).

2. **Generated tests:** We randomly generated 100 tests for each case example, each varying from 30-100 test steps (input to output sequences) in length (Section 5.2).

3. **Set steering constraints:** We constrained the variables that could be adjusted through steering and the values that those variables could take on, and established dissimilarity metrics to be minimized (Sections 5.4 and 5.3).

4. **Assessed impact of steering:** For each combination of SUT, test, and dissimilarity metric, we attempted to steer the oracle to match the behavior of the SUT. We compare the test results before and after steering and evaluate the precison and recall of our steering framework, contrasted against the general practice of not steering and a step-by-step filtering mechanism (Section 5.6).

5. **Assessed impact of tolerance constraints:** We repeated steps 3-4 for each SUT and five mutants using four different sets of tolerance constraints, varying in strictness, in order to assess the impact of the choice of constraints (Sections 5.4 and 5.6).

6. **Learned new tolerance constraints:** Using the original and steered traces for each model and the trace for each SUT, for each of the four constraint levels and both dissimilarity metrics, we extracted 10 sets of tolerance constraints (= 80 per SUT) using the TAR3 treatment learning algorithm (Section 5.5).

7. **Assessed performance of learned tolerance constraints:** We repeated steps 3-4 for each SUT using the extracted tolerance constraints in order to assess the quality of those constraints.

5.2 System and Test Generation

To produce "implementations" of the example systems, we created alternative versions of each model, introducing realistic non-deterministic timing changes to the systems. For the Infusion system, we built (1) a version of the system where the exit of the patient-requested dosage period may be delayed by a short period of time, and (2) a version of the system where the exit of an intermittent increased dosage period (known as a square bolus dose) may be delayed. These changes are intended to mimic situations where, due to hardware-introduced computation delays, the system remains in a particular dosage mode for longer than expected.

For the Pacing system, we introduced a non-deterministic delay on the arrival of sensed cardiac activity. As a pacemaker is a complex, networked series of subsystems that depend on strict timing conditions, a common source of mismatch between model and system occurs when sensed activity arrives at a particular subsystem later than expected. Depending on the extent of the delay, unnecessary electrical impulses may be delivered to the patient or the pacemaker may enter different operational modes than the model.

For each of the original models and "system under test" variants, we have also generated 50 *mutants* (faulty implementations) by introducing a single fault into each model. This ultimately results in a total of 152 SUT versions of the Infusion system—two versions with non-deterministic timing behavior, fifty versions with faults, and one hundred versions with both non-deterministic timing and seeded faults (fifty per timing variation)—and 101 SUT variants of the Pacing system—one SUT with non-deterministic timing, fifty with faults, and fifty with both non-deterministic timing and seeded faults.

The mutation testing operators used in this experiment include changing an arithmetic operator, changing a relational operator, changing a boolean operator, introducing the boolean $\neg$ operator, using the stored value of a variable from the previous computational cycle, changing a constant expression by adding or subtracting from int and real constants (or by negating boolean constants), and substituting a variable occurring in an equation with another variable of the same type. The mutation operators used are discussed at length in an earlier report [107], and are similar to those used by Andrews et.al, where the authors found that generated mutants are a reasonable substitute for actual failures in testing experiments [108].

Using a random testing algorithm, we generated 100 tests. For the Infusion system, each test is thirty steps in length, representing thirty seconds of system activity. For the pacing system, the tests range 30-100 steps in length, representing input and output events occurring over 3000 ms of activity. In both cases, the test length was chosen to be long enough to capture a relevant range of time-sensitive behaviors, but still short enough to yield a reasonable experiment cost. These tests were then executed against each model and SUT variant in order to collect traces. In the SUT variants with timing fluctuations, we controlled those fluctuations through the use of an additional input variable. The value for that variable was generated non-deterministically, but we used the same value across all systems with the same timing fluctuation. As a result, we know whether a resulting behavioral mismatch is due to a seeded timing fluctuation or a seeded fault in the system. Using this knowledge, we manually classified each test as an "expected pass" or as failing due to an "acceptable timing deviation", an "unacceptable timing deviation", or a "seeded fault."

5.3 Dissimilarity Metrics

In this experiment, we have made use of two different dissimilarity metrics when comparing a candidate state of the model-based oracle to the state of the SUT. The first is the Manhattan (or City Block) distance. Given vectors representing the state of the SUT and the model-based oracle—where each member of the vector represents the value of a variable—the dissimilarity between the two vectors can be measured as the sum of the absolute numerical distance between the state of the SUT and the model-based oracle:

$$Dis(s_m, s_{sut}) = \sum_{i=1}^{n} |s_{m,i} - s_{sut,i}| \tag{5.1}$$

The second is the Squared Euclidean distance. Given vectors representing the state, the dissimilarity between the vectors can be measured as the "straight-line" numerical distance between the two vectors. The squared variant was chosen because it places greater weight on states that are further apart in terms of variable values.

$$Dis(s_m, s_{sut}) = \sum_{i=1}^{n} (s_{m,i} - s_{sut,i})^2 \tag{5.2}$$

A constant difference of 1 is used for differences between boolean variables or values of an enumerated variable. All numerical values are normalized to a 0-1 scale using predetermined constants for the minimum and maximum values of each variable.

When formulating a dissimilarity metric—or, for the matter, an oracle verdict—we must choose a set of variables to compare between the oracle model and SUT. As we cannot assume a common internal structure between the SUT and the model, we calculate similarity using the output variables of both systems. For the infusion pump, this includes the commanded flow rate, the current system mode, the duration of active infusion, a log message indicator, and a flag indicating that a new infusion has been requested. For the pacemaker, this set of variables consists of an atrial event classification, a ventricular event classification, the time of the event, the time of the next scheduled atrial pace attempt, and the time of the next scheduled ventricular pace attempt.

5.4 Manually-Set Tolerance Constraints

In order to assess the performance and capabilities of steering, we have specified a realistic set of tolerance constraints for both systems: For both systems, we have specified the tolerance constraints on steering in terms of limits on the adjustment of the input variables of the system (our model-based oracle steering framework allows constraints to be placed on internal or output variables as well). The chosen tolerance constraints for the Infusion system include:

- Five of the input variables relate to timers within the system—the duration of the patient-requested bolus dose period, the duration of the intermittent square bolus dosage period, the lockout period between patient-requested bolus dosages, the interval between intermittent square bolus dosages, and the total duration of the infusion period. For each of those, we placed an allowance of $(CurrVal - 1) <= NewVal <= (CurrVal + 2)$. E.g., following steering, a dosage duration is allowed to fall within a three second period—between one second shorter and two seconds longer than the original prescribed duration.

- The remaining 15 input variables are not allowed to be steered.

and, for the Pacing system:

- The input for a sensed cardiac event includes a timestamp indicating when the system will process the event. For this event, we placed an allowance of $Current\ Value <= New\ Value <= (Current\ Value + 4)$. Following steering, the sensed event can be adjusted to have taken place anywhere within a four millisecond window following the original timestamp.

- There are also boolean input variables indicating whether an event was sensed in the atrial or ventricular chambers of the heart. These can be toggled on or off, to better match the noise level in the SUT.

- The remaining 17 input variables are not allowed to be steered.

These constraints reflect what we consider a realistic application of steering—we expect issues related to non-deterministic timing, and, thus, allow a small acceptable window around the behaviors that are related to timing. For the Infusion system, we do not expect any sensor

inaccuracy, so we do not allow freedom in adjusting sensor-based inputs. We expect a small amount of event reordering and noise for the Pacing system, so there we allow a small amount of freedom in changing sensor-based values. As these are medical devices that could harm a patient if misused, we do not allow any changes to the inputs related to prescription values. As these are restrictive constraints, we deem these the **Strict** tolerance constraints.

In order to assess the impact of different levels of strictness in the tolerance constraints, we took each SUT variant, five randomly-selected mutants of the original system, and five randomly-selected mutants for each SUT and attempted to steer them using the Strict tolerances and three additional sets of tolerances: **Medium, Minimal, and No Input Constraints**.

For the Infusion system, these are as follows:

- **Medium:** All time-based inputs, $(CurrVal - 2) <= NewVal <= (CurrVal + 5)$. All other variables are not allowed to be steered.
- **Minimal:** All time-based inputs completely unconstrained. All other variables are not allowed to be steered.
- **No Input Constraints:** All variables unconstrained.

and, for the Pacing system:

- **Medium:** Ventricle and atrial sensed events unconstrained. Event time, $Current\ Value <= New\ Value <= (Current\ Value + 25)$. Refractory periods, $Current\ Value <= New\ Value <= (Current\ Value + 10)$. All other variables are not allowed to be steered.
- **Minimal:** Ventricle and atrial sensed events and event time unconstrained. Refractory periods, $Current\ Value <= New\ Value <= (Current\ Value + 25)$. Lower and upper rate limit, $Current\ Value <= New\ Value <= (Current\ Value + 10)$. All other variables are not allowed to be steered.
- **No Input Constraints:** All input variables unconstrained.

These additional constraint sets represent a gradual relaxation of the limits on what steering is allowed to change, and are intended to demonstrate the impact that the choice of constraints has on the effectiveness of steering.

5.5 Learning Tolerance Constraints

In Section 3.4, we discussed the process of using treatment learning to extract a set of tolerance constraints. We wished to know whether we could learn tolerances for the case studies examined in this paper, and whether these tolerances are effective at guiding the steering process.

Using the process outlined previously, we generated tolerance constraints for the Infusion variant where the patient bolus period can be extended non-deterministically and for the Pacing system. Starting from the strict, medium, and minimal tolerance constraints and using no input constraints, we executed tests, steered the models, and extracted data from those executions and classifications on what the correct verdict should be post-steering. Because the treatment learners use a heuristic search process, we generate ten sets of constraints per preexisting constraint set (i.e., 10 sets generated after extracting data from steering with strict tolerance constraints, 10 sets generated after extracting data from steering with no input constraints, etc). We repeat this for each dissimilarity metric. This results in eighty sets of tolerance constraints learned from each system (4 constraint levels x 10 repeats x 2 metrics).

We generate tolerances using the TAR3 treatment learning algorithm. TAR3 is a well-known approach to treatment learning [34, 35, 36, 37]. It produces treatments by first being fed a set of training examples, E. Each example $e \in E$ consists of values, discretized into a series of ranges, for a given set of attributes. This set of value ranges is directly mapped to a specific classification, $R_i, R_j, ... \rightarrow C$. As in Section 3.4, each example corresponds to a test step, where the attributes of the data set represent the changes made to variable values by steering and the correctness of the changes.

In order to produce a treatment, a target classification C_{target} must be specified, and the set of class symbols C_1, C_2,...,C_{target} are ranked and sorted based on a utility score $U_1 < U_2 < ... < U_{target}$, where U_{target} represents the utility score of the target classification. Within the dataset E, each classification occur at a certain frequencies (F_1, F_2,..., F_{target}) where $\sum F_i$ = 1—that is, each class occupies a fraction of the overall dataset.

TAR3 produces treatments by utilizing two important scoring heuristics—lift and support. A treatment T of size M is a conjunction of attribute value ranges $R_1 \wedge R_2 ... \wedge R_M$ Some subset of the examples in the dataset ($e \subseteq E$) is contained within T. That is, if the treatment is used to filter E, $e \subseteq E$ is what will remain. In that subset, the possible classifications occur at frequencies f_1, f_2,..., f_C. The *lift* of a treatment is a measurement in the change in class

distribution in the entries of the dataset that remain after a treatment has been imposed. That is, TAR3 seeks the smallest treatment T that induces the biggest changes in the weighted sum of the utility scores multiplied by the frequencies of the associated classes. This score, calculated based on the subset $e \subseteq E$ that results when T has been imposed, is divided by the score of the baseline score (dataset E when no treatment has been applied). The lift is formally defined as:

$$lift = \frac{\sum_c U_c f_c}{\sum_c U_c F_c}. \tag{5.3}$$

Real-world datasets, especially those recorded from hardware systems [34], contain some *noise*—incorrect or misleading data. If these noisy examples are correlated with particular classifications, the treatment may become overfitted. An overfitted model result in a large lift score, but it does not accurately reflect the general conditions of the dataset's search space. To avoid overfitting, TAR3 adopts a threshold and reject all treatments that fall on the wrong side of this threshold. This is defined as the *minimum best support*.

The best support is defined as the ratio of the frequency of the target classification within the treatment subset to the frequency of that classification in the overall dataset. To avoid overfitting, TAR3 rejects all treatments with best support lower than a user-defined minimum (usually 0.2). As a result, treatments produced by TAR3 will have both a high *lift* and high *support*.

TAR3's lift and support calculations can assess the effectiveness of a treatment, but they are not what generates the treatments themselves. A naive approach to treatment learning might be to test all subsets of all ranges of all of the attributes. However, as a dataset of size N has 2^N possible subsets, this type of brute force attempt is inefficient. Instead, TAR3 employs a heuristic approach that begins by discretizing every continuous attribute into smaller ranges by sorting their values and dividing them into a set of equally-sized bins (producing the attribute value ranges contained within treatments). It then limits the size of a treatment, only building treatments up to a user-defined size. Past research [36, 37] has shown that treatments larger than four attributes lack support and are harder for humans to understand.

TAR3 will only build treatments from the discretized ranges with a high heuristic value. It determines which ranges to use by first determining the lift score of each individual attribute's value ranges (that is, the score of the class distribution obtained by filtering for the data instances that contain a value in that particular range for that particular attribute). These individual scores

are then sorted and converted into a cumulative probability distribution. TAR3 randomly selects values from this distribution, meaning that low-scoring ranges are unlikely to be selected in the first place. To build a treatment, n random ranges are selected and combined. These treatments are then scored and sorted. If no improvement is seen after a certain number of rounds, TAR3 terminates and returns the top treatments.

In order to create a set of tolerance constraints, we first create 10 treatments—like those seen in Table 3.3—that indicate the correct use of steering to adjust the state of the model. As some of these items may not actually be indicative of successful steering, coincidental steering actions that appear in both *correct* and *incorrect* steering, we also produce 10 treatments that correspond to incorrect steering actions. We remove any treatments that appear in both the "good" and "bad" sets, leaving only those that appear in the good set. We then form our set of elicited tolerance constraints by locking down any variables that constraints were not suggested for. This results in a set of tolerances similar to that shown in Figure 3.6. We repeat this process ten times for each constraint level and dissimilarity metric in order to control for the effects of variance.

5.6 Evaluation

Using the generated artifacts—without steering—we monitored the outputs during each test, compared the results to the values of the same variables in the model-based oracle to calculate the dissimilarity score, and issued an initial verdict. Then, if the verdict was a failure ($Dis(s_m.s_{sut}) > 0$), we steered the model-based oracle, and recorded a new verdict post-steering. As mentioned above, the variables used in establishing a verdict are the five output variables of the system.

In Section 3, we stated that an alternative approach to steering would be to apply a filter on a step-by-step basis. We have implemented such a filter for the purposes of establishing a baseline to which we can compare the performance of steering. This filter compares the values of the output variables of the SUT to the values of those variables in the model-based oracle and, if they do not match, checks those values against a set of constraints. If the output—despite non-conformance to the model—meets these constraints, the filter will still issue a "pass" verdict for the test.

For the Infusion_Mgr system, the filter will allow a test to pass if (despite non-conformance)

values of the output variables in the SUT satisfy the following constraints:

- The current mode of the SUT is either "patient dosage" mode or "intermittent dosage" mode, and has not remained in that mode for longer than $prescribed\ duration + 2$ seconds.
- If the above is true, the commanded flow rate should match the prescribed value for the appropriate mode.
- All other output variables should match their corresponding variables in the oracle.

As we expect a non-deterministic duration for the patient dosage and intermittent dosage modes (corresponding to the seeded issues in the SUT variants), this filter should be able to correctly classify many of the same tests that we expect steering to handle.

For the Pacing system, the filter will allow a test to pass if the values of the output variables satisfy:

- The event timestamp on the output and the scheduled time of the next atrial and ventricular events fall within four milliseconds of the time originally predicted by the model.
- All other output variables should match their corresponding variables in the oracle.

Similar to the Infusion_Mgr system, we expect short non-deterministic delays in when the Pacing system issues an output event.

We compare the performance of the steering approach to both the filter and the default practice of accepting the initial test verdict. We can assess the impact of steering or filtering using the verdicts made before and after steering by calculating:

- The number of *true positives*—steps where an approach does not mask incorrect behavior;
- The number of *false positives*—steps where an approach fails to account for an acceptable behavioral difference;
- And the number of *false negatives*—steps where an approach does mask an incorrect behavior.

Initial Verdict	Pass (Post-Steering)	Fail (Post-Steering)
Pass	TN	FP
Fail (Due to Timing, Within Tolerance)	TN	FP
Fail (Due to Timing, Not in Tolerance)	FN	TP
Fail (Due to Fault)	FN	TP

Table 5.2: Verdicts: T(true)/F(false), P(positive)/N(negative).

The testing outcomes in terms of true/false positives/negatives are listed in Table 5.2. Using these measures, we calculate the *precision*—the ratio of true positives to all positive verdicts—and *recall*—the ratio of true positives to true positives and false negatives:

$$Precision = \frac{TP}{TP + FP} \tag{5.4}$$

$$Recall = \frac{TP}{TP + FN} \tag{5.5}$$

We also calculate the *F-measure*—the harmonic mean of precision and recall—in order to judge the accuracy of oracle verdicts:

$$\text{Accuracy (F-measure)} = 2 * \frac{precision * recall}{precision + recall} \tag{5.6}$$

Chapter 6

Results and Discussion

As previously presented in Table 5.2, testing outcomes can be categorized according to the initial verdict as determined by the model-based oracle before steering; a "fail" verdict is further delineated according to its reason—a mismatch that is attributable to either an allowable timing fluctuation, an unacceptable timing fluctuation or a fault.

For the Infusion_Mgr system—when running all tests over the various implementations (containing either timing deviations or seeded faults as discussed in Section 5.2) using a standard test oracle comparing the outputs from the SUT with the outputs predicted by the model-based oracle (15,200 test runs)—11,364 runs indicated that the system under test passed the test (the SUT and model-based oracle agreed on the outputs) and 3,936 runs indicated that the test failed (the SUT and model-based oracle had mismatched outputs). In an industry application of a model-based oracle, the 3,936 failed test would have to be examined to determine if the failure was due to an actual fault in the implementation, an unacceptable timing deviation from the expected timing behavior, or an acceptable timing deviation that, although it did not match the behavior predicted by the model-based oracle, was within acceptable tolerances—a costly process. Given our experimental setup, however, we can classify the failed tests as to the cause of the failure: failure due to timing within tolerances, failure due to timing not in tolerance, and failure due to a fault in the SUT. This breakdown is provided in Table 6.2. As can be seen, 1,406 tests failed even though the timing deviation was within what would be acceptable—these can be viewed as false positives and a filtering or steering approach that would have passed these test runs would provide cost savings. On the other hand, the steering or filtering should not pass any of the 268 tests where timing behavior falls outside of tolerance or the 2,229 tests that

Table 6.1: Experimental results for the Infusion_Mgr system.

Verdict	Number of Tests
Pass	11364 (74.8%)
Fail (Due to Timing, Within Tolerance)	1406 (9.2%)
Fail (Due to Timing, Not in Tolerance)	268 (1.8%)
Fail (Due to Fault)	2229 (14.6%)

Table 6.2: Initial test results when performing no steering or filtering for Infusion_Mgr system. Raw number of test results, followed by percent of total.

Initial Verdict	Pass (Post-Steering)	Fail (Post-Steering)
Pass	11364 (74.8%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	1245 (8.2%)	152 (1.0%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	268 (1.8%)
Fail (Due to Fault)	43 (0.3%)	2186 (14.3%)

Table 6.3: Distribution of results for steering of Infusion_Mgr. Raw number of test results, followed by percent of total. Results same for both dissimilarity metrics

Initial Verdict	Pass (Post-Filtering)	Fail (Post-Filtering)
Pass	11364 (74.8%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	1245 (8.2%)	152 (1.0%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	268 (1.8%)
Fail (Due to Fault)	1252 (8.2%)	977 (6.4%)

Table 6.4: Distribution of results for step-wise filtering of Infusion_Mgr. Raw number of test results, followed by percent of total.

Technique	Precision	Recall	Accuracy
No Adjustment	0.64	1.00	0.78
Filtering	0.89	0.50	0.64
Steering (Both Metrics)	0.94	0.98	0.96

Table 6.5: Precision, recall, and accuracy values for Infusion_Mgr.

indicated real faults.

A similar breakdown can be found for the Pacing system in Table 6.7. About a third of the test executions—2,992 in total—pass initially. A further 21%, or 2,208 tests, fail due to

Table 6.6: Experimental results for the Pacing system.

Verdict	Number of Tests
Pass	2992 (29.6%)
Fail (Due to Timing, Within Tolerance)	2208 (21.2%)
Fail (Due to Timing, Not in Tolerance)	571 (5.7%)
Fail (Due to Fault)	4329 (42.9%)

Table 6.7: Initial test results when performing no steering or filtering for Pacing system. Raw number of test results, followed by percent of total.

Initial Verdict	Pass (Post-Steering)	Fail (Post-Steering)
Pass	2992 (29.6%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	2065 (20.4%)	143 (1.4%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	571 (5.7%)
Fail (Due to Fault)	297 (2.9%)	4032 (39.9%)

Table 6.8: Distribution of results for steering of Pacing. Raw number of test results, followed by percent of total. Results same for both dissimilarity metrics

Initial Verdict	Pass (Post-Filtering)	Fail (Post-Filtering)
Pass	2992 (29.6%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	1010 (10.0%)	1198 (11.9%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	571 (5.7%)
Fail (Due to Fault)	258 (2.6%)	4071 (40.3%)

Table 6.9: Distribution of results for step-wise filtering of Pacing. Raw number of test results, followed by percent of total.

Technique	Precision	Recall	Accuracy
No Adjustment	0.69	1.00	0.82
Filtering	0.79	0.95	0.86
Steering (Both Metrics)	0.97	0.94	0.95

Table 6.10: Precision, recall, and accuracy values for Pacing.

acceptable timing deviations. These should, ideally, pass following the application of steering or filtering. A further 571 test executions fail due to unacceptable timing differences, and 4,329 fail due to seeded faults. Steering and filtering should, ideally, not correct these tests.

Results obtained from the case study showing the effect of steering on oracle verdicts are summarized in Table 6.3 and 6.8 respectively, for the Infusion_Mgr and Pacing systems. For both systems, the two dissimilarity metrics performed identically. The raw results are presented as laid out in Table 5.2. For each category, the post-steering verdict is presented as both a raw number of test outcomes and as a percentage of total test outcomes. Tables 6.4 and 6.9 show the corresponding data for the step-by-step filtering approach for the two systems. Data from these tables lead to the precision, recall, and Accuracy values—shown for Infusion_Mgr in Table 6.5 and for Pacing in Table 6.10—for the default testing scenario (accepting the initial oracle verdict), steering, and filtering. In the following sections, we will discuss the results presented in these tables with regard to our central research questions.

6.1 Allowing Tolerable Non-Conformance

For the Infusion_Mgr system—according to Table 6.2—11% of the tests (1,674 tests) initially fail due to timing-related non-conformance. Of those, 1406 tests (9.2% of the total) fall within the tolerances set in the requirements. Steering should result in a pass verdict for all of those tests. Similarly, of the 2,779 tests (26.9%) that fail due to timing reasons in the Pacing system, 2,208 (21.2%) fail due to differences that are acceptable, and steering should account for these execution divergences (see Table 6.7).

As Table 6.3 shows, for both dissimilarity metrics, steering is able to account for almost all of the situations where non-deterministic timing affects conformance while both the model-based oracle and the implementation remain within the bounds set in the system specification. We see that steering using either distance metric correctly passes 1,245 tests where the timing deviation was acceptable—tests that without steering failed. Therefore, we see a sharp increase in precision over the default situation where no steering is employed (from 0.64 when not steering, to 0.94 when steering, according to Table 6.5).

Table 6.8 demonstrates similar results for steering on the Pacing system. Steering correctly changes the verdicts of 2,065 of the tests that initially failed. As shown in Table 6.10, this results in a large increase in precision—from 0.69 when not steering to 0.97.

Where previously developers would have had to manually inspect the more than 25% of all test execution traces for the Infusion_Mgr system (the sum of all “Fail” verdicts in Table 6.3) to determine the causes for their failures (system faults or otherwise), they could now narrow

their focus to the roughly 17% of test executions that still result in failure verdicts post-steering. For the Pacing system, steering drops this total from 70% inspection rate to a somewhat more managable 47%.

Particularly given the large number of tests in this study, this reduction represents a significant savings in time and effort, removing between potentially thousands of execution traces that the developer would have needed to inspect manually. Still, for both systems, there were a small number of tests that steering should have been able to account for (152, or 1% of the test executions, for Infusion_Mgr and 143, 1.4%, for Pacing). The reason for the failure of steering to account for allowable differences can be attributed to a combination of three factors: the tolerance constraints employed, the dissimilarity metric employed, and internal design differences between the SUT and the model-based oracle.

First, it may be that the tolerance constraints were too strict to allow for situations that should have been considered legal. As discussed in Section 3.3, the employed tolerance constraints play a major role in determining the set of candidate steering actions. By design, constraints should be relatively strict—after all, we are overriding the nominal behavior of the oracle while simultaneously wishing to retain the oracle's power to identify faults. Yet, the constraints we apply should be carefully designed to allow steering to handle these allowed non-conformance events. In this case, the chosen constraints may have prevented steering from acting in a relatively small number of situations in which it should have been able to account for a behavior difference. This is to be expected, and one of the strengths of steering is that it is relatively easy to tune the constraints and execute tests again until the right balance is struck. Fortunately, for both systems, the chosen constraints were able to account for the vast majority of situations that should have been corrected.

Second, the dissimilarity metric plays a role in guiding the selection of steering action. In our experiments, we noted no differences between the Manhattan and Squared Euclidean metrics in the solutions chosen—both took the same steering actions. By design, the metrics compare the output variables of the model and SUT (i.e., the set of variables that we use to determine a test verdict) and compute a numeric score. For the systems examined, the output variables were relatively simple numeric or boolean values, and we did not witness any situations where the metric could be "tricked" into favoring changes to one particular variable or another. In other types of systems, the choice of metric may play a more important role—particularly if, say, string comparisons are needed.

However, although the two metrics performed identically well, they may also both share the same blind spot. The metrics compare state in *this* round of execution, and do not consider the implications of a steering action in future test steps. It is possible that multiple candidate steering actions will result in the same score, but that certain choices will cause *eventual* divergences between the model and SUT that cannot be reconciled at that time. Such a possibility is limited in this particular experiment due to the strictness of the tolerance constraints employed, but will be discussed in more detail with regard to the tolerance experiment examined in Section 6.4. It is possible that the "wrong" steering actions were chosen in those cases where steering failed to correct the verdict—initially closing the execution gap, but causing further eventual divergence. This indicates a need for further research work on limiting future side effects when choosing a steering action, and may necessitate further development of dissimilarity metrics.

Third, as previously discussed, the tolerance constraints reduce the space of candidate targets to which the oracle may be steered. We then use the dissimilarity metric to choose a "nearest" target from that set of candidates. Thus, the relationship between the constraints and the metric ultimately determines the power of the steering process. However, no matter how capable steering is, there may be situations where differences in the internal design of the system and model render steering either ineffective or incorrect. We base steering decisions on state-based comparisons, but those comparisons can only be made on the portion of the state variables common between the SUT and oracle model (and, in particular, we limit this knowledge to the variables used for the oracle's verdict comparison, as these are the only variables we can assume the common existence of). As a result, there may be situations where we should have steered, but could not, as the state of the SUT depended on internal factors not in common with the oracle. In general, as the oracle and SUT are both ultimately based on the same set of requirements, we believe that some kind of relationship can be established between the internal variables of both realizations. However, in some cases, the model and SUT may be too different to allow for steering in all allowable situations. The inability of steering to account for tolerable differences for at least some tests in this case study can likely be attributed to the changes made to the SUT versions of the models.

In practice, when tuning the precision of steering, the choice of steering constraints seems to have the largest impact on the resulting accuracy of the steering process (see Section 6.4). While we do believe that the choice of metric and the relationship between the metric and the constraints both play an important role in determining the effectiveness of steering, in practice,

the set of constraints chosen showed the clearest correlation to the resulting precision. Therefore, if steering results in a large number of false failure verdicts, we would first recommend that testers experiment with different sets of constraints until the number of false failures has decreased (without covering up known faults).

6.2 Masking of Faults

As steering changes the behavior of the oracle and can result in a new test verdict, the danger of steering is naturally that it will mask actual faults in the system. Such a danger is concerning, but with the proper choice of steering policies and constraints, we hypothesize that such a risk can be reduced to an acceptable level.

As can be seen in Table 6.3, when steering the Infusion_Mgr model, we changed a fault-induced "fail" verdict to "pass" in forty-three tests. This is a relatively small number—only 0.3% of the 15,200 test executions. This, according to Table 6.5, results in a drop in recall from 1.0 (for accepting the initial verdict) to 0.98. For the Pacing model, as shown in Table 6.8, steering adjusted a fault-induced failure to a pass for a small, but slightly higher, percentage of test executions—258 runs, or 2.9% of the test executions. The resulting recall is 0.94 for steering (Table 6.10).

Although any loss in recall is cause for concern when working with safety-critical systems, given the small number of incorrectly adjusted test verdicts for both systems, we believe that it is unlikely for an actual fault to be entirely masked by steering on *every* test in which the fault would otherwise lead to a failure. Of course, we still would urge care when working with steering.

Just as the choice of tolerance constraints can explain cases where steering is unable to account for an allowable non-conformance, the choice of constraints has a large impact on the risk of fault-masking. At any given execution step, steering, as we have defined here, considers only those oracle post-states as candidate targets that are reachable from the the given oracle pre-state. However, this by itself is not sufficiently restrictive to rule out truly deviant behaviors. Therefore, the constraints applied to reduce that search space must be strong enough to prevent steering from forcing the oracle into an otherwise impermissible state for that execution step. It is, therefore, crucial that proper consideration goes into the choice of constraints. In some cases, the use of additional policies—such as not steering the oracle model at all if it does not

Initial Verdict	Pass (Post-Filtering)	Fail (Post-Filtering)
Pass	11311 (74.4%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	312 (2.1%)	1123 (7.4%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	268 (1.7%)
Fail (Due to Fault)	598 (3.9%)	1688 (11.1%)

Table 6.11: Distribution of results for step-wise filtering, (outputs + volume infused oracle), for Infusion_Mgr. Raw number of test results, followed by percent of total.

result in an exact match with the system—can also lower the risk of tolerating behaviors that would otherwise indicate faults.

Note that a seeded fault could *cause* a timing deviation (or the same behavior that would result from a timing deviation). In those cases, the failure is still labeled as being induced by a fault for our experiment. However, if the fault-induced deviation falls within the tolerances, steering will be able to account for it. In real world cases, where the faults are not purposefully induced, it is unlikely that even a human oracle would label the outcome differently, as they are working from the same system and domain knowledge that the tolerance constraints are derived from.

In real-world conditions, if care is taken when deriving the tolerance constraints from the system requirements, steering should not cover any behaviors that would not be permissible under those same requirements. Still, as steering carries the risk of masking faults, we recommend that it be applied as a *focusing tool*—to point the developer toward test failures likely to indicate faults so that they do not spend as much time investigating non-conformance reports that turn out to be allowable. The final verdict on a test should come from a run of the oracle model with no steering, but during development, steering can be effective at streamlining the testing process by concentrating resources on those failures that are more likely to point to faults.

6.3 Steering vs Filtering

In some cases, acceptable non-conformance events could simply be dealt with by applying a filter that, in the case of a failing test verdict, checks the resulting state of the SUT against a set of constraints and overrides the initial oracle verdict if those constraints are met. Such filters are relatively common in GUI testing [21].

Technique	Precision	Recall	Accuracy
No Adjustment	0.64	1.00	0.78
Filtering	0.64	0.76	0.70
Steering (Both Metrics)	0.94	0.98	0.96

Table 6.12: Precision, recall, and accuracy values for filtering (outputs + volume infused oracle) for Infusion_Mgr.

The use of a filter is tempting—if the filter is effective, it is likely to be easier to build and faster to execute than a full steering process. Indeed, for Infusion_Mgr, the results in Table 6.4 appear initially promising. The filter performs identically to steering for the initial failures that result from non-deterministic timing differences. It does not issue a pass verdict for timing issues outside of the tolerance limits, and it does issue a pass for almost all of the tests where non-conformance is within the tolerance bounds. As can be seen in Table 6.5, the use of a filter increases the precision from 0.64 for no verdict adjustment to 0.89.

However, when the results for tests that fail due to faults are considered, a filter appears much less attractive. The filter issues a passing verdict for 1,252 tests that should have failed—1,209 more than steering. This is because a filter is a *blunt instrument*. It simply checks whether the state of the SUT meets certain constraints when non-conformance occurs. This allowed the filter to account for the allowed non-conforming behaviors, but these same constraints also allowed a large selection of fault-indicating tests to pass.

This makes the choice of constraints even more important for filtering than it is in steering. The steering process, by backtracking the state of the system, is able to ensure that the resulting behavior of the SUT is even possible (that is, if the new state is reachable from the previous state). The filter does not check the possibility of reaching a state; it just checks whether the new state is globally acceptable under the given constraints. As a result, steering is far more accurate. A filter could, of course, incorporate a reachability analysis. However, as the complexity of the filter increases, the reasons for filtering instead of steering disappear.

In fact, the success of steering at accounting for allowable non-conformance is somewhat misleading for the Infusion_Mgr case example. Both filtering and steering base their decisions on the output variables of the SUT and oracle, on the basis that the internal state variables may differ between the two. For this case study, all of the output variables reflect *current conditions* of the infusion pump—how much drug volume to infuse *now*, the *current* system mode, and so forth. Internally, these factors depend on both the current inputs and a number of *cumulative*

factors, such as the total volume infused and the remaining drug volume. Over the long term, non-conformance events between the SUT and model will build, eventually leading to wider divergence. For example, the SUT or the model-based oracle may eventually cut off infusion if the drug reservoir empties. While a filter may be a perfectly appropriate solution for static GUIs, the cumulative build-up of differences in complex systems, will likely render a filter ineffective on longer time scales.

As the output variables reflect current conditions for this system, mounting internal differences may be missed, and the filter may not be able to cope with larger behavior differences that result from this steady divergence. Steering is able to prevent these long-term divergences by *actually changing the state of the oracle* throughout the execution of the test. A filter simply overrides the oracle verdict. It does not change the state of the oracle, and as a result, a filter cannot predict or handle behavioral divergences once they build beyond the set of constraints that the filter applies.

We can illustrate this effect by adding a single internal variable to the set of variables considered when making filtering or steering conditions—a variable tracking the total drug volume infused. Adding this variable causes *no change* to the results of steering seen in Table 6.3. However, the addition of this internal variable dramatically changes the results of filtering. The new results can be seen in Tables 6.11 and 6.12.

Because the total volume infused increases over the execution of the test, it will reflect any divergence between the model-based oracle and the SUT. As steering actually adjusts the execution of the model-based oracle, this volume counter also adjusts to reflect the changes induced by steering. Thus, steering is able to account for the growing difference in the volume infused by the model-based oracle and the volume infused by the SUT. However, as the filter makes no such adjustment, it is unable to handle the mounting difference in this variable (or any other considered variable that reflects change over time). The filter, even if initially effective, will fail to account for a large number of acceptable non-conformance events—ultimately resulting in a precision value no more effective than not doing anything at all (and a far lower recall).

Similar results can be seen for the Pacing system in Table 6.9. The output variables of the Pacing example include both immediate commands, but also scheduled times for the next pacing events in both heart chambers. As a result, the output variables reflect internally-growing divergences between the model and SUT far more quickly than they appear in Infusion_Mgr. Thus, the precision of filtering is far lower than that of steering for the Pacing system, as the

filter struggles to keep up with the time-dependent changes that mount over the execution of the test case. When we moved the internal volume counter variable to the outputs of Infusion_Mgr, we saw precision fall and recall rise. Similarly, for Pacing, precision is far lower for the filter than for steering, but the loss in recall is quite small as a result. The filter does not allow many divergences to pass—legal or illegal. Thus, filtering actually has a slightly higher level of recall than steering. However, it comes at a far higher cost to precision.

6.4 Impact of Tolerance Constraints

The tolerance constraints limit the choice of steering action. In essence, they define the specification of what non-determinism we will allow, bounding the variance between the model and SUT that can be corrected. As emphasized in Section 3.3, the selection of an appropriate set of constraints in likely a crucial factor in the success or failure of steering. If tolerance constraints are too strict, we hypothesize that they will be useless for correcting the allowable behavioral deviations; too loose, and dangerous faults may be masked. We saw hints of this in the initial experiment, which utilized strict tolerance constraints. We masked only a vanishingly small number of faults, but we also failed to account for a small number of tests that should have been handled. The choice of constraints was a key factor in both the success of steering—not masking faults—and the limitations of the process—not handling all acceptable tests.

Given the apparent importance of the selection of tolerance constraints, we wished to determine the exact impact of the choice of tolerance constraints. One advantage of using steering over, say, directly modeling non-determinism is that, by utilizing a seperate collection of rules to specify the bounds on acceptable non-deterministic deviations, we can easily *change* the constraints. By swapping in a new constraint file and re-executing the test suite, one can examine the effects of steering with the new limitations. For both Infusion_Mgr and Pacing, we took the implementations and five mutants for the original and each implementation and executed the test suite using four different sets of constraints. These are detailed in Section 5.4, and represent a steady loosening of the constraints from *strict* to *no constraints at all*.

Precision, recall, and accuracy results for Infusion_Mgr can be seen in Table 6.14. Exact values for accepting the initial verdict, filtering, and strict constraints differ slightly from those in Table 6.5, due to the lower number of mutants used, but the trends remain the same. As detailed in Table 6.16, steering with **strict** constraints improves precision by quite a bit by

Table 6.13: Constraint results for the Infusion_Mgr system (Overview).

Technique	Precision	Recall	Accuracy
No Adjustment	0.58	1.00	0.73
Filtering	0.89	0.67	0.76
Steering - Strict	0.93	1.00	0.96
Steering - Medium	0.90	0.88	0.89
Steering - Minimal	0.98	0.85	0.91
Steering - No Input Constraints	1.00	0.48	0.65

Table 6.14: Precision, recall, and accuracy values for different tolerance constraint levels—as well as filtering and no adjustment—for the Infusion_Mgr system. Results are the same for both dissimilarity metrics. Visualized below.

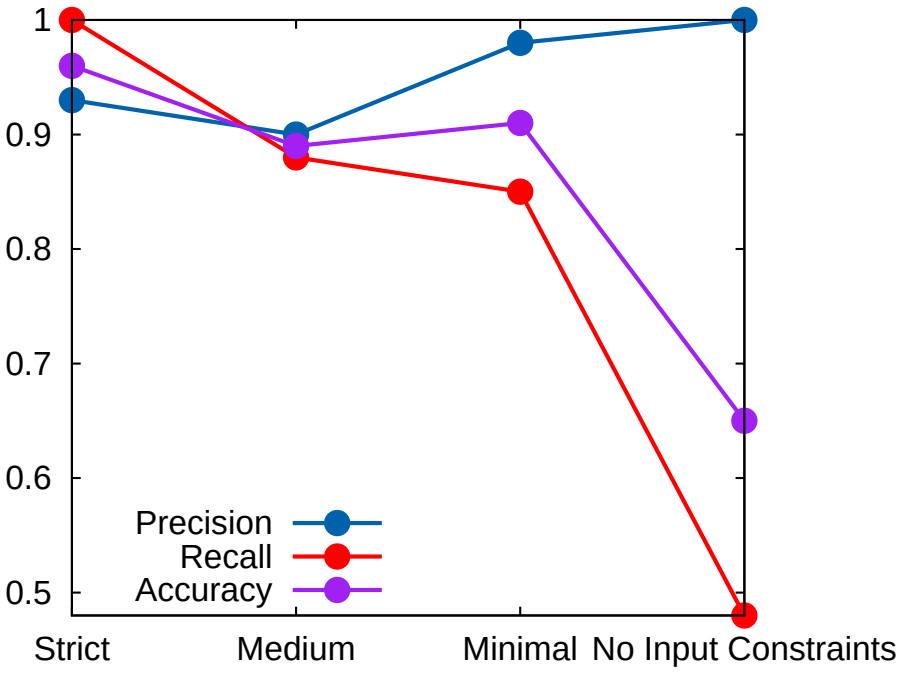

Table 6.15: Constraint results for the Infusion_Mgr system (Detailed).

Initial Verdict	Pass (Post-Filtering)	Fail (Post-Filtering)
Pass	1270 (74.9%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	161 (9.4%)	19 (1.1%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	35 (2.0%)
Fail (Due to Fault)	0 (0.0%)	210 (12.4%)

Table 6.16: Distribution of results for steering with strict tolerance constraints for the Infusion_Mgr system. Raw number of test results, followed by percent of total. Results are the same for both dissimilarity metrics.

Initial Verdict	Pass (Post-Filtering)	Fail (Post-Filtering)
Pass	1270 (74.9%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	156 (9.2%)	24 (1.4%)
Fail (Due to Timing, Not in Tolerance)	30 (1.8%)	5 (0.3%)
Fail (Due to Fault)	0 (0.0%)	210 (12.4%)

Table 6.17: Distribution of results for steering with medium tolerance constraints for the Infusion_Mgr system. Raw number of test results, followed by percent of total. Results are the same for both dissimilarity metrics.

Initial Verdict	Pass (Post-Filtering)	Fail (Post-Filtering)
Pass	1270 (74.9%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	176 (10.4%)	4 (0.2%)
Fail (Due to Timing, Not in Tolerance)	35 (2.1%)	0 (0.0%)
Fail (Due to Fault)	1 (0.1%)	209 (12.3%)

Table 6.18: Distribution of results for steering with minimal tolerance constraints for the Infusion_Mgr system. Raw number of test results, followed by percent of total. Results are the same for both dissimilarity metrics.

Initial Verdict	Pass (Post-Filtering)	Fail (Post-Filtering)
Pass	1270 (74.9%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	180 (10.6%)	0 (0.0%)
Fail (Due to Timing, Not in Tolerance)	35 (2.1%)	0 (0.0%)
Fail (Due to Fault)	92 (5.4%)	118 (7.0%)

Table 6.19: Distribution of results for steering with no input constraints for the Infusion_Mgr system. Raw number of test results, followed by percent of total. Results are the same for both dissimilarity metrics.

correcting almost all of the acceptable behavioral deviations, while masking no faults. Filtering, as in Section 6.3, improves precision, but at the cost of covering up many faults (resulting in a lower recall value).

As we loosen the constraints to the **medium** level—detailed in Table 6.17—we see a curious drop in precision. A small number of additional tests that fail due to acceptable non-determinism still fail after steering. It is likely that this is due to the sudden availability of additional steering actions. When presented with more choices, the search process chooses one of the several that minimizes the dissimilarity metric *now*, but causes side effects later on. We will revisit this when examining the results for the Pacing system. The recall also dips significantly. Inspecting Table 6.17 makes the reason for this clear, when given more freedom to adjust the timer values, the steering process will naturally cover up unacceptable timing differences. This underlines the importance of selecting constraints carefully.

When examining the results with **minimal** constraints and when there are **no constraints on the input variables** for Infusion_Mgr, shown in Tables 6.18 and 6.19, two clear trends emerge. The first is that, as the constraints loosen, the precision rises. Naturally, given the freedom to make larger and larger adjustments to the selected steerable variables, the search process can handle more and more of the tests that fail due to acceptable deviations. Unsurprisingly, this increase in precision comes at a heavy cost to recall. With minimal constraints, we now not only can handle more of the acceptable deviations, but we also cover up many of the unacceptable deviations. Fortunately, we still effectively *do not mask code-based faults*. This is an encouraging result for steering. Although minimal constraints do cover the bad behaviors induced by non-determinism, we are still not masking issues within the code of the system.

That changes when we move to steering with **no input constraints** on the input variables of Infusion_Mgr. Now, not only can we handle all of the timing-based failures—acceptable or illegal—we also mask many of the induced faults as well. This is not unexpected. Given complete freedom to deviate from the original test inputs, guided only by the use of the dissimilarity metric, steering will mask many illegal behaviors. This is a clear illustration of the importance of selecting the correct constraints. Give too much freedom, and faults will be masked; too little freedom, and acceptable deviations will distract testers. It is important to experiment and strike the correct balance. Fortunately, our initial set of tolerances seems to have hit a reasonable balance point for Infusion_Mgr.

The precision, recall, and accuracy figures for the Pacing system appear in Table 6.21.

Table 6.20: Constraint results for the Pacing system (Overview).

Technique	Precision	Recall	Accuracy
No Adjustment	0.61	1.00	0.76
Filtering	0.72	0.93	0.81
Steering - Strict	0.95	0.93	0.94
Steering - Medium	0.94	0.77	0.85
Steering - Minimal	0.94	0.77	0.85
Steering - No Input Constraints	0.62	0.89	0.73

Table 6.21: Precision, recall, and accuracy values for different tolerance levels—as well as filtering and no adjustment—for the Pacing system. Results are the same for both dissimilarity metrics. Visualized below.

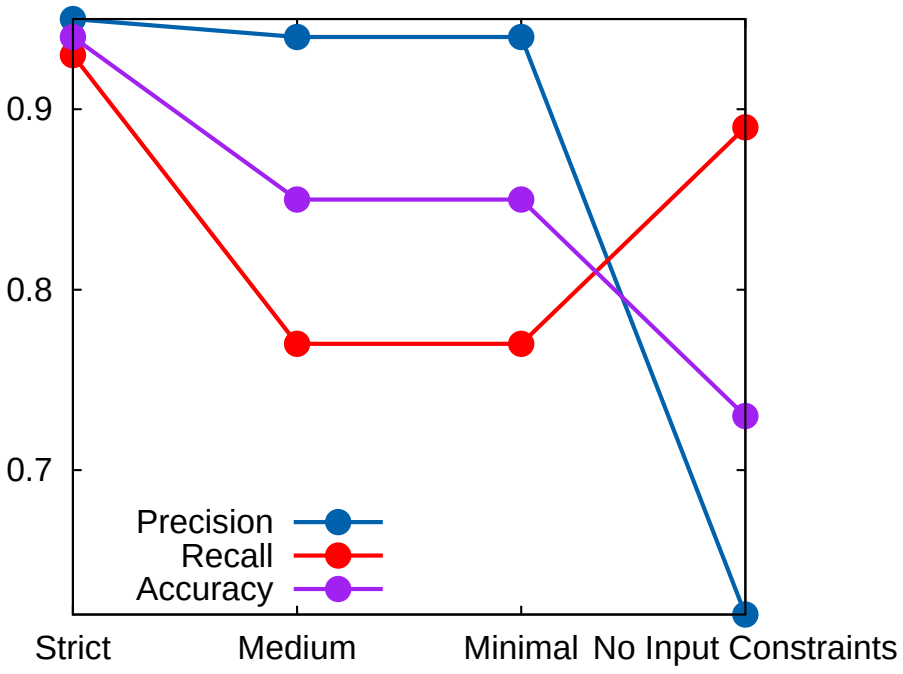

Table 6.22: Constraint results for the Pacing system (Detailed).

Initial Verdict	**Pass (Post-Filtering)**	**Fail (Post-Filtering)**
Pass	308 (28.0%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	284 (25.8%)	22 (2.0%)
Fail (Due to Timing, Not in Tolerance)	3 (0.2%)	81 (7.3%)
Fail (Due to Fault)	30 (2.7%)	372 (33.8%)

Table 6.23: Distribution of results for steering with strict tolerance constraints for the Pacing system. Raw number of test results, followed by percent of total. Results are the same for both dissimilarity metrics.

Initial Verdict	**Pass (Post-Filtering)**	**Fail (Post-Filtering)**
Pass	308 (28.0%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	284 (25.8%)	22 (2.0%)
Fail (Due to Timing, Not in Tolerance)	70 (6.3%)	14 (1.2%)
Fail (Due to Fault)	40 (3.6%)	362 (32.9%)

Table 6.24: Distribution of results for steering with medium and minimal tolerance constraints for the Pacing system. Raw number of test results, followed by percent of total. Results are the same for both dissimilarity metrics.

Initial Verdict	**Pass (Post-Filtering)**	**Fail (Post-Filtering)**
Pass	308 (28.0%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	34 (3.1%)	272 (24.7%)
Fail (Due to Timing, Not in Tolerance)	6 (0.5%)	78 (7.1%)
Fail (Due to Fault)	46 (4.1%)	362 (32.4%)

Table 6.25: Distribution of results for steering with no input constraints for the Pacing system. Raw number of test results, followed by percent of total. Results are the same for both dissimilarity metrics.

Variable Name	Explanation	When Impacts Behavior
IN_V_EVENT	Sensed event indicator for ventricle chambers	Immediate
IN_A_EVENT	Sensed event indicator for atrial chambers	Immediate
IN_EVENT_TIME	Time of sensor poll	Immediate
IN_SYSTEM_MODE	Current system mode	Immediate
IN_LRL	Lower rate limit on paces	Likely Delayed
IN_URL	Upper rate limit on paces	Likely Delayed
IN_HYSTERESIS_RL	Optional adaptation of artificial pacing rate to natural pacing	Likely Delayed
IN_VRP	Ventricular refractory period following a ventricular event	Likely Delayed
IN_ARP	Atrial refractory period following an atrial event	Likely Delayed
IN_PVARP	Atrial refractory period following a ventricular event	Likely Delayed
IN_PVARP_EXTENSION	Optional extension on PVARP following certain events	Likely Delayed
IN_FIXED_AVD	Fixed timing window between atrial event and ventricular reaction	Likely Delayed
IN_DYNAMIC_AVD	Enables a dynamic timing window between atrial events and ventricular reactions	Likely Delayed
IN_DYNAMIC_AVD_MIN	Minimum dynamically-determined value for AVD window	Likely Delayed
IN_ATR_MODE	Enables special mode to ease patient out of pacemaker-induced atrial tachycardia	Delayed
IN_ATR_DURATION	Defines minimum period before entering ATR mode	Likely Delayed
IN_ATR_FALLBACK_TIME	Defines duration of ATR mode	Likely Delayed

Table 6.26: Inputs for the Pacing system.

Again, the results for no adjustment, filtering, and **strict** steering follow the same trends as the earlier experiment (see Table 6.10). Filtering and steering do equally well on recall, but steering achieves far higher precision. The filter is unable to keep up with the behavior divergences that build over time, while steering keeps up by adjusting execution each time behaviors diverge.

As we shift to the **medium**, **minimal**, and **no input constraint** results—detailed in Tables 6.24 and 6.25—we see an interesting divergence from the results for Infusion_Mgr. Namely, that rather that improving, the precision actually significantly drops—from 0.95, to 0.94, and finally to 0.62. Steering the Pacing model with no input constraints is barely more accurate on the legal divergences than not steering at all.

By loosening the constraints, we gave the steering algorithm more freedom to manipulate the input values. On the Infusion_Mgr system, this resulted in us being able to handle more and more of the acceptable behavior differences, but at the cost of also covering up more and more of the unacceptable differences. On Pacing, we cover more faults as the constraints are loosened, but steering actually grows far *less capable* at accounting for the acceptable differences. This can be explained by examining the steerable variables for the Pacing system, detailed in Table 6.26.

Unless particularly specific conditions are met, changes to many of the steerable variables

for Pacing will have a delayed impact on the behavior of the system. For example, altering the length of one of the refractory periods will only immediately impact behavior if we are in a refractory period and decrease that period to be less than the current duration. To give a second example, enabling or altering ATR mode settings will only alter behavior if we are already in ATR mode, and even then, likely only after we have been in it for a longer period of time. Therefore, given very loose constraints (or worse, no constraints), it is incredibly easy for the steering algorithm to configure the immediately-effective variables to minimize the dissimilarity score, but to also alter one of the delayed variables in such a way that it eventually drives the model to diverge from the system. Infusion_Mgr, too, has prescription variables that can have delayed effects, but many of those could be adjusted again to "fix" the side effect. For Pacing, many of these side effects can only be "fixed" after they have damaged conformance.

This issue further highlights the importance of choosing good constraints, as many of the steering induced changes that can cause eventual side effects are also changes that *would not address hardware or time-based non-determinism*. Intuitively, changes to the majority of possible timing issues with Pacing would be restricted to a small number of those input variables and—even then—would only require small adjustments. You may want to correct a small delay in pacing, or slightly shift the end of a refractory period that has lasted too long, but it is unlikely that you would want to steer either of those factors by a significant amount, as the end result of a software fault could impact the health of a patient.

The possibility of choosing a steering action with an undesirable delayed side effect points to the need for further research on both tolerance constraints and dissimilarity metrics. We may want to add in a penalty factor on changes to certain variables to bias the search algorithm towards first trying the variables with immediate effects. We may also want to judge the impact of a steering action on both the immediate differences between the model and SUT and the impact on *eventual* differences. However, checking both current and future behavior is a difficult challenge, as the computational requirements to perform such a comparison may not be realistically obtainable.

Ultimately, what we see from both systems is that the choice of tolerance constraints is crucially important in determining the capabilities and limitations of steering. Relatively strict constraints seem to offer the best balance between accounting for acceptable deviations and masking fault-indicative behavior. As constraints loosen, we run a significantly increased risk of masking faults or choosing steering actions with undesirable long-term side effects. That

said, the key to determining a reasonable set of steering constraints is in understanding the requirements of the system being built and the domain the device must operate within. Choosing the correct set of constraints is important, but it is a task that reasonably experienced developers should be capable of conducting. Our framework allows experimentation with different sets of constraints, allowing developers to find and tune the tolerance constraints. By using domain knowledge and the software specifications to build reasonable, well-considered constraints, we can use steering to enable the use of model-based oracles and focus the attention of the developers.

6.5 Automatically Deriving Tolerance Constraints

As indicated in the previous section, tolerance constraints play an important role in the success of steering. Selecting the right constraints is clearly important; yet, one can imagine scenarios where the developers are uncertain of what boundaries to set or even what variables to loosen or constrain. Thus, we were interested in investigating whether constraints can be *learned* from steering against developer-classified test cases.

We took one of the time-delayed implementations of Infusion_Mgr (called "PBOLUS") and the implementation of Pacing, steered using no input constraints for both dissimilarity metrics, and derived ten sets of tolerance constraints using the learning process described in Section 3.4. As the same learning process can also be applied to refine existing constraints, we repeated the same process for the strict, medium, and minimal constraint sets.

The results for PBOLUS are shown in Table 6.27, where the reported calculations for the learned constraints are the median of ten trials. As expected, making no adjustments to the verdicts when steering PBOLUS results in the lowest precision. As we did not include any of the implementations with seeded faults in this experiment, filtering performs very well—handling the timing fluctuations specific to this implementation with relative ease. Across the board, the results for the learned constraints are very positive, on average generally matching or exceeding filtering.

When learning from the strict, medium, or minimal constraints, the learned constraints can only be a *tightening* of the constraints being learned from. That is, if particular variables are already locked down, then those variables will not suddenly be loosened. Variables that have a tolerance window will only have that window remain the same size or have that window

Technique	Precision	Recall	Accuracy
No Adjustment	0.18	1.00	0.30
Filtering	0.70	1.00	0.82
Learned from Strict (Manhattan)	0.70	1.00	0.82
Learned from Strict (Squared Euclidean)	0.65	1.00	0.77
Learned from Medium (Manhattan)	0.70	1.00	0.82
Learned from Medium (Squared Euclidean)	0.70	1.00	0.82
Learned from Minimal (Manhattan)	0.73	1.00	0.84
Learned from Minimal (Squared Euclidean)	0.76	1.00	0.86
Learned from No Tolerances (Manhattan)	0.82	1.00	0.89
Learned from No Tolerances (Squared Euclidean)	0.76	1.00	0.86

Table 6.27: Median precision, recall, and accuracy values for learned tolerance constraints for the Infusion_Mgr PBOLUS system.

shrink—learning will not further open that window. Thus, a certain ceiling effect forms where, if a developer missed a variable that should have been steerable, then the tightening can only help a small amount with performance. Here, that performance ceiling for tightening seems to line up with the performance of a filter. This was hinted at for the Infusion_Mgr system in Section 6.3, where the filter performed as well as strict steering for the allowable deviations. Here, we see the same effect—when learning from existing constraints, we can only improve performance to a limited degree.

This tightening may still be useful if a developer is sure of the variables to steer, but unsure of the bounds to set on those variables. However, the results of learning from *no preexisting input constraints* are interesting because they do not have this performance limitation. Given just a set of classified tests with no input constraints, we are free to derive constraints on any variables and freely set those boundaries. As a result, for the PBOLUS system, the best results emerge when given this freedom, with median steering performance after learning from no input constraints beating steering after learning from strict tolerance constraints on precision by up to 17%. This suggests that the strict constraints may actually be stricter than they need to be, potentially missing variables that should be adjustable.

Note that we did see minor differences between the executions using the Manhattan dissimilarity metric and the Squared Euclidean metric. However, given their identical performance in prior experiments, we believe the differences noted here are due to the stochastic nature of the learning process, rather than a difference induced by the choice of metric.

Technique	Precision	Recall	Accuracy
No Adjustment	0.24	1.00	0.39
Filtering	0.32	1.00	0.48
Learned from Strict (Manhattan)	0.26	1.00	0.42
Learned from Strict (Squared Euclidean)	0.26	1.00	0.42
Learned from Medium (Manhattan)	0.26	1.00	0.42
Learned from Medium (Squared Euclidean)	0.26	1.00	0.42
Learned from Minimal (Manhattan)	0.26	1.00	0.42
Learned from Minimal (Squared Euclidean)	0.26	1.00	0.42
Learned from No Tolerances (Manhattan)	0.24	1.00	0.39
Learned from No Tolerances (Squared Euclidean)	0.25	1.00	0.40

Table 6.28: Median precision, recall, and accuracy values for learned tolerance constraints for the Pacing system.

Initial Verdict	Pass (Post-Filtering)	Fail (Post-Filtering)
Fail (Due to Timing, Within Tolerance)	9	67
Fail (Due to Timing, Not in Tolerance)	0	24

Table 6.29: Distribution of results for steering with tolerance constraints learned from strict for the Pacing system. Raw number of test results. Results are the same for both dissimilarity metrics.

```
((real(IN_EVENT_TIME) = concrete_oracle_IN_EVENT_TIME) or ((real(IN_EVENT_TIME)
     >= concrete_oracle_IN_EVENT_TIME + 2.000000) and (real(IN_EVENT_TIME)
     <= concrete_oracle_IN_EVENT_TIME + 3.000000)))
((real(IN_AVD_OFFSET) >= concrete_oracle_IN_AVD_OFFSET) and (
     real(IN_AVD_OFFSET) <= concrete_oracle_IN_AVD_OFFSET + 1.000000))
(real(IN_ARP) = concrete_oracle_IN_ARP)
(real(IN_VRP) = concrete_oracle_IN_VRP)
...
(real(IN_URL) = concrete_oracle_IN_URL)
(real(IN_LRL) = concrete_oracle_IN_LRL)
```

Figure 6.1: Sample constraints learned for Pacing.

At first, the results for the learned constraints for Pacing—shown in Table 6.28—appear very poor. The constraints learned for Pacing score a median precision of around 0.26 in almost all cases, and as low as 0.24. Filtering does not do well, either, but does lead the pack with a precision of 0.32. We can see why the learning results are poor by examining the detailed test

Technique	Precision	Recall	Accuracy
No Adjustment	0.24	1.00	0.39
Filtering	0.32	1.00	0.48
Learned (All)	0.75	0.88	0.81

Table 6.30: Median precision, recall, and accuracy values for learned tolerance constraints for the Pacing system after widening learned constraints.

executions, listed in Table 6.29. The learned constraints, in almost all cases, are extremely strict. They never allow illegal behaviors to pass, but they also fail to compensate for the majority of the legal deviations.

Interestingly, if we look at the learned constraints (an example set is listed in Figure 6.1), we see that the learning process has actually picked up on the *correct variables* to set constraints on. In particular, it almost universally allowed the sensed event indicators to be free and allowed a small window of adjustment on the event time. However, it set *too strict* of a limit on how much those variables could be adjusted. This is actually an easy "issue" to correct. As mentioned a number of times, the use of a separate tolerance constraint file means that it is easy to experiment with different constraints. Given that we are using a set of classified test results, we can simply adjust the tolerances and re-execute the tests until the performance meets a desirable threshold.

Such an adjustment can even be done automatically, by systematically shrinking or widening the learned tolerances until this threshold is met. In this case, we take the constraints and universally increased the windows by one on both ends. For example, the IN_EVENT_TIME tolerance listed in Figure 6.1 transforms from allowing an adjustment of 2-3 seconds to allowing an adjustment anywhere from 1-4 seconds. Even this small adjustment leads to markedly improved results, as can be seen in Table 6.30. Across the board, this small adjustment led to a median accuracy result of 0.81—far higher than no adjustment, filtering, or the original learned constraints.

For both systems, the results of learning tolerance constraints seem quite positive. Given a set of classified tests, we are able to extract a small, strict set of constraints that can be used—perhaps after a small amount of tuning—to successfully steer a model.

6.6 Summary of Results

The precision, recall, and F-measure (a measure of accuracy) for each method—accepting the initial verdict, steering (using two different dissimilarity metrics), and filtering—are shown for Infusion_Mgr in Table 6.5 and for Pacing in Table 6.10.

The default situation, accepting the initial verdict, results in the lowest precision value. Intuitively, not doing anything to account for allowed non-conformance will result in a large number of incorrect "fail" verdicts. However, the default practice does have the largest recall value. Again, not adjusting your results will prevent incorrect masking of faults. Filtering on a step-by-step basis results in higher precision than doing nothing, but due to the lack of reachability analysis and state adaptation—both of which used by the steering approach—the filter masks an unacceptably large number of faults for Infusion_Mgr. For Pacing, filtering is unable to keep up with the complex divergences that build over time. Although it is able to improve the level of precision over not adjusting the verdict, it fails to match the precision gains seen when steering.

Steering performs identically for both of the dissimilarity metrics used in this study. It is able to adapt the oracle to handle almost every situation where non-conforming behaviors are allowed by the system requirements, while masking only a few faults in a small number of tests. For both systems, steering results in a large increase in precision, with only a small cost in recall.

Ultimately, we find that steering results in the highest accuracy for the final test results for both systems. Steering demonstrates a higher overall accuracy—balance of precision and recall—than filtering or accepting the initial verdict, 0.96 to 0.64 and 0.78 for Infusion_Mgr and 0.95 to 0.86 and 0.82 for Pacing.

Tolerance constraints play a large role in determining the efficacy of steering, both limiting the ability of steering to mask faults and its ability to correct acceptable deviations. Relatively strict, well-considered constraints strike the best balance between enabling steering to focus developers and preventing steering from masking faults. As constraints are loosened, we observed that steering may be able to account for more and more acceptable deviations, but at the cost of also masking many more faults. Alternatively, loose constraints may also impair steering from performing its job by allowing the search process to choose a steering action that causes eventual side-effects.

Fortunately—even if developers are unsure of what variables to set constraints on—as long as they can classify the outcome of a set of tests, a set of constraints can automatically be learned. For our case examples, the derived set of constraints was small, strict, and able to successfully steer the model (albeit, for Pacing, with a small amount of tuning).

Steering is able to automatically adjust the execution of the oracle to handle non-deterministic, but acceptable, behavioral divergence without covering up most fault-indicative behaviors. We, therefore, recommend the use of steering as a tool for focusing and streamlining the testing process.

Chapter 7

Threats to Validity

External Validity: Our study is limited to two case examples. Although we are actively working with domain experts to produce additional systems for future studies, we believe that the systems we are working with to be representative of the real-time embedded systems that we are interested in, and that our results will generalize to other systems in this domain.

We have used Stateflow translated to Lustre as our modeling language. Other modeling notations can, in theory, be used to steer. We do not believe that the modeling language chosen would have a significant impact on the ability to steer the model. Similarly, we have used Lustre as our implementation language, rather than more common languages such as C or C++. However, systems written in Lustre are similar in style to traditional imperative code produced by code generators used in embedded systems development. A simple syntactic transformation is sufficient to translate Lustre code to C code.

We have limited our study to fifty mutants for each version of the case example, resulting in a total of 150 mutants for the Infusion_Mgr system and 100 for the Pacing system. These values are chosen to yield a reasonable cost for the study, particularly given the length of each test. It is possible the number of mutants is too low. Nevertheless, we have found results using less than 250 mutants to be representative [109, 107], and pilot studies have shown that the results plateau when using more than 100 mutants.

Internal Validity: Rather than develop full-featured system implementations for our study, we instead created alternative versions of the model—introducing various non-deterministic behaviors—and used these models and the versions with seeded faults as our "systems under

test." We believe that these models are representative approximations of the behavioral differences we would see in systems running on embedded hardware. In future work, we plan to generate code from these models and execute the software on actual hardware platforms.

In our experiments, we used a default testing scenario (accepting the oracle verdict) and stepwise filtering as baseline methods for comparison. There may be other techniques—particularly, other filters—that we could compare against. Still, we believe that the filter chosen was an acceptable comparison point, and was designed as such a filter would be in practice.

Construct Validity: We measure the fault finding of oracles and test suites over seeded faults, rather than real faults encountered during development of the software. Given that our approach to selecting oracle data is also based on the mutation testing, it is possible that using real faults would lead to different results. This is especially likely if the fault model used in mutation testing is significantly different than the faults we encounter in practice. Nevertheless, as mentioned earlier, Andrews et al. have shown that the use of seeded faults leads to conclusions similar to those obtained using real faults in similar fault finding experiments [110].

Chapter 8

Conclusion and Future Work

Specifying test oracles is still a major challenge for many domains, particularly those—such as real-time embedded systems—where issues related to timing, sensor inaccuracy, or the limited computation power of the embedded platform may result in non-deterministic behaviors for multiple applications of the same input. Behavioral models of systems, often built for analysis and simulation, are appealing for reuse as oracles. However, these models typically present an abstracted view of system execution that may not match the execution reality. Such models will struggle to differentiate unexpected—but still acceptable—behavior from behaviors indicative of a fault.

To address this challenge, we have proposed an automated *model-based oracle steering framework* that, upon detecting a behavioral difference, backtracks and selects—through a search-based process—a *steering action* that will bring the model in line with the execution of the system. To prevent the model from being forced into an illegal behavior—and masking a real fault—the search process must select an action that satisfies certain constraints and minimizes a dissimilarity metric. This framework allows non-deterministic, but bounded, behavior differences while preventing future mismatches by guiding the model, within limits, to match the execution of the SUT.

Experiments, conducted over complex real-time systems, have yielded promising results and indicate that steering significantly increases SUT-oracle conformance with minimal masking of real faults and, thus, has significant potential for reducing development costs. The use of our steering framework can allow developers to focus on behavioral difference indicative of real faults, rather than spending time examining test failure verdicts that can be blamed on a

rigid oracle model.

8.1 Future Work

There is still much room for future work. A few of the topics that we plan to continue to explore include:

- We plan to further examine the impact of different dissimilarity metrics, tolerance constraints, and steering policies on oracle verdict accuracy. Our results point to the importance of setting the right constraints, and we will work to improve both automated learning of constraints and on identifying advice on manually selecting and tuning constraints. With regard to dissimilarity functions, we have observed situations where the action selected minimizes the function in the current comparison, but causes undesired side effects later in execution. We plan to examine ways to avoid these side effects.

- We seek improvements to the speed and scalability of the steering framework, including the use of metaheuristic search algorithms. Although complete search methods offer a guarantee of optimality—if there is way to minimize the dissimilarity metric, the search will find it—they operate under strict limitations and can take a long period of time to find a solution. In some situations, it may be worth trading the optimality guarantee for the increased speed and ability to solve mathematically complex dissimilarity metrics offered by metahueristic search methods. We plan to explore the use and appropriateness of different search techniques.

- We would like to examine the use of steering and dissimilarity metrics as methods of quantifying non-conformance and their utility in fault identification and location. Using the dissimilarity metric to quantify the impact of behavioral deviation could potentially lead to powerful new techniques of test case generation, selection, and prioritization.

- And, we plan to examine the use of steering to debug faulty or incomplete oracle models. Generally, oracles are assumed to be correct. However, like programs, test oracles are developed by humans, and mistakes can be made in their construction. We have used steering to extend models to handle unexpected real-world situations. However, the very same techniques could be used to help automatically diagnose faults—or even correct—in faulty oracle models.

References

[1] Saswat Anand, Edmund Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, August 2013.

[2] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. A comprehensive survey of trends in oracles for software testing. Technical Report CS-13-01, University of Sheffield, Department of Computer Science, 2013.

[3] A. En-Nouaary, R. Dssouli, and F. Khendek. Timed wp-method: testing real-time systems. *Software Engineering, IEEE Transactions on*, 28(11):1023–1038, Nov.

[4] Matthew S. Jaffe, Nancy G. Leveson, Mats P.E. Heimdahl, and Bonnie E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.

[5] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[6] MathWorks Inc. Stateflow. http://www.mathworks.com/stateflow, 2015.

[7] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[8] IBM Rational Rhapsody. http://www.ibm.com/developerworks/rational/ products/rhapsody/, 2014.

[9] D.W. Miller, J. Guo, E. Kraemer, and Y. Xiong. On-the-fly calculation and verification of consistent steering transactions. In *Supercomputing, ACM/IEEE 2001 Conf.*, pages 8–8, 2001.

[10] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer, 2008.

[11] Gregory Gay, Sanjai Rayadurgam, and Mats P.E. Heimdahl. Steering model-based oracles to admit real program behaviors. In *Proceedings of the 36th International Conference on Software Engineering – NIER Track*, ICSE '14, New York, NY, USA, 2014. ACM.

[12] W.E. Howden. Theoretical and empirical studies of program testing. *IEEE Transactions on Software Engineering*, 4(4):293–298, 1978.

[13] E.J. Weyuker. The oracle assumption of program testing. In *13th Int'l Conf on System Sciences*, pages 44–49, 1980.

[14] M. Staats, G. Gay, and M.P.E. Heimdahl. Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 2012 Int'l Conf. on Software Engineering*, pages 870–880. IEEE Press, 2012.

[15] D. Coppit and J.M. Haddox-Schatz. On the use of specification-based assertions as test oracles. In *Proceedings of the 29th Annual IEEE/NASA on Software Engineering Workshop*, SEW '05, pages 305–314, Washington, DC, USA, 2005. IEEE Computer Society.

[16] M. Pezze and M. Young. *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, October 2006.

[17] The Modelica Association. Modelica - a unified object-oriented language for systems modeling. Technical report, 2012.

[18] Adriano Gomes, Alexandre Mota, Augusto Sampaio, Felipe Ferri, and Edson Watanabe. Constructive model-based analysis for safety assessment. *Int'l Journal on Software Tools for Technology Transfer*, 14(6):673–702, 2012.

[19] Steven P. Miller, Alan C. Tribble, Michael W. Whalen, and Mats P. E. Heimdahl. Proving the shalls: Early validation of requirements through formal methods. *Int. J. Softw. Tools Technol. Transf.*, 8(4):303–319, 2006.

[20] K. G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using UPPAAL. In *Int'l workshop on formal approaches to testing of software (FATES 04)*. Springer, 2004.

[21] Sonal Mahajan and William G.J. Halfond. Finding html presentation failures using image comparison techniques. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 91–96, New York, NY, USA, 2014. ACM.

[22] W. Gu, J. Vetter, and K. Schwan. An annotated bibliography of interactive program steering. *ACM SIGPLAN Notices*, 29, 1994.

[23] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[24] David Hardin, T Douglas Hiratzka, D Randolph Johnson, Lucas Wagner, and Michael Whalen. Development of security software: A high assurance methodology. In *Formal Methods and Software Engineering*, pages 266–285. Springer, 2009.

[25] G. Hagen. *Verifying safety properties of Lustre programs: an SMT-based approach*. PhD thesis, University of Iowa, December 2008.

[26] Sung-Hyuk Cha. Comprehensive survey on distance/similarity measures between probability density functions. *International Journal of Mathematical Models and Methods in Applied Sciences*, 1(4):300–307, 2007.

[27] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, March 2001.

[28] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems, Second Edition*. Cambridge Press, 2006.

[29] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[30] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[31] Phil McMinn, Mark Stevenson, and Mark Harman. Reducing qualitative human oracle costs associated with automatically generated test data. In *Proceedings of the First International Workshop on Software Test Output Validation*, STOV '10, pages 1–4, New York, NY, USA, 2010. ACM.

[32] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Houndations of Machine Learning*. MIT Press, 2012.

[33] Tim Menzies and Ying Hu. Data mining for very busy people. *Computer*, 36(11):22–29, November 2003.

[34] Gregory Gay, Tim Menzies, Misty Davies, and Karen Gundy-Burlet. Automatically finding the control variables for complex system behavior. *Automated Software Engg.*, 17(4):439–468, December 2010.

[35] Corina Pasareanu Tim Menzies Johann Schumann, Karen Gundy-Burlet and Anthony Barrett. Software v&v support by parametric analysis of large software simulation systems. In *2009 IEEE Aerospace Conference*, 2009.

[36] K. Gundy-Burlet, J. Schumann, T. Barrett, and T. Menzies. Parametric analysis of antares re-entry guidance algorithms using advanced test generation and data analysis. In *9th International Symposium on Artifical Intelligence, Robotics and Automation in Space*, 2007.

[37] K. Gundy-Burlet, J. Schumann, T. Barrett, and T. Menzies. Parametric analysis of a hover test vehicle using advanced test generation and data analysis. In *AIAA Aerospace*, 2009.

[38] Stephen D. Bay and Michael J. Pazzani. Detecting change in categorical data: Mining contrast sets. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, pages 302–306, New York, NY, USA, 1999. ACM.

[39] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In Franck Cassez, Claude Jard, Brigitte Rozoy, and MarkDermot Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer Berlin Heidelberg, 2001.

[40] Jan Tretmans. Model based testing with labelled transition systems. In *Formal methods and testing*, pages 1–38. Springer, 2008.

[41] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence, 1996.

[42] Jan Tretmans. A formal approach to conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test Systems VI*, pages 257–276, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co.

[43] ITU-T SG 10/Q.8 ISO/IEC JTC1/SC21 WG7. Information retrieval, transfer and management for osi; framework: Formal methods in conformance testing, 1996.

[44] Gilles Bernot. Testing against formal specifications: A theoretical view. In *TAPSOFT'91*, pages 99–119. Springer, 1991.

[45] Rob J van Glabbeek. The linear timebranching time spectrum ii. In *CONCUR'93*, pages 66–81. Springer, 1993.

[46] R De Nicola. Extensional equivalence for transition systems. *Acta Inf.*, 24(2):211–237, April 1987.

[47] Laura Brandn Briones and Ed Brinksma. A test generation framework for quiescent real-time systems. In *IN FATES'04*, pages 64–78. Springer-Verlag GmbH, 2004.

[48] Axel Belinfante, Jan Feenstra, RenG. de Vries, Jan Tretmans, Nicolae Goga, Loe Feijs, Sjouke Mauw, and Lex Heerink. Formal test automation: A simple experiment. In Gyula Csopaki, Sarolta Dibuz, and Katalin Tarnay, editors, *Testing of Communicating*

Systems, volume 21 of *IFIP The International Federation for Information Processing*, pages 179–196. Springer US, 1999.

[49] Jan Tretmans and Ed Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering*, pages 31–43, December 2003.

[50] Jean-Claude Fernandez, Claude Jard, Thierry Jeron, and Cesar Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1):123–146, 1997.

[51] Rance Cleaveland and Matthew Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5(1):1–20, 1993.

[52] Thierry Jéron and Pierre Morel. Test generation derived from model-checking. In *Computer Aided Verification*, pages 108–122. Springer, 1999.

[53] Brian Nielsen and Arne Skou. Automated test generation from timed automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 343–357. Springer, 2001.

[54] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *Proceedings of the 22nd IFIP WG 6.1 Int'l Conf. on Testing software and systems*, pages 95–110. Springer-Verlag, 2010.

[55] T. Savor and R.E. Seviora. An approach to automatic detection of software failures in real-time systems. In *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, pages 136–146, 1997.

[56] Rachel Cardell-Oliver. Conformance test experiments for distributed real-time systems. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 159–163. ACM, 2002.

[57] Kim G Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 299–306. ACM, 2005.

[58] Duncan Clarke and Insup Lee. Automatic generation of tests for timing constraints from requirements. In *Object-Oriented Real-Time Dependable Systems, 1997. Proceedings., Third International Workshop on*, pages 199–206. IEEE, 1997.

[59] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *Automata, languages and programming*, pages 322–335. Springer, 1990.

[60] Anders Hessel, Kim G Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal test cases for real-time systems. In *Formal Modeling and Analysis of Timed Systems*, pages 234–245. Springer, 2004.

[61] Jan Springintveld, Frits Vaandrager, and Pedro R D'Argenio. Testing timed automata. *Theoretical computer science*, 254(1):225–257, 2001.

[62] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on UPPAAL 4.0, 2006.

[63] Howard Bowman, Giorgio Faconti, Joost-Pieter Katoen, Diego Latella, and Mieke Massink. Automatic verification of a lip-synchronisation algorithm using uppaal-extended version. In *FMICS'98, Third Internatinoal Workshop on Formal Methods for Industrial Crtical Systems*, pages 97–124. CWI, 1998.

[64] Klaus Havelund, Kim Guldstrand Larsen, and Arne Skou. *Formal verification of a power controller using the real-time model checker Uppaal*. Springer, 1999.

[65] Anders P Ravn, Jiří Srba, and Saleem Vighio. Modelling and verification of web services business activity protocol. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–371. Springer, 2011.

[66] Anders Hessel and Paul Pettersson. Model-based testing of a wap gateway: An industrial case-study. In *Formal Methods: Applications and Technology*, pages 116–131. Springer, 2007.

[67] Susumu Fujiwara, F Khendek, M Amalou, A Ghedamsi, et al. Test selection based on finite state models. *Software Engineering, IEEE Transactions on*, 17(6):591–603, 1991.

[68] Abdeslam En-Nouaary, Ferhat Khendek, and Rachida Dssouli. Fault coverage in testing real-time systems. In *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*, pages 150–157. IEEE, 1999.

[69] Ferenc Belina, Dieter Hogrefe, and Amardeo Sarma. *SDL with applications from protocol specification*. Prentice-Hall, Inc., 1991.

[70] S. P. Miller, M. P.E. Heimdahl, and A.C. Tribble. Proving the shalls. In *Proceedings of FM 2003: the 12th Int'l FME Symposium*, September 2003.

[71] S. Kannan, M. Kim, I. Lee, O. Sokolsky, and M. Viswanathan. Run-time monitoring and steering based on formal specifications. In *Workshop on Modeling Software System Structures in a Fastly Moving Scenario*, 2000.

[72] L. Lin and M. D. Ernst. Improving the adaptability of multi-mode systems via program steering. In *Proceedings of the 2004 ACM SIGSOFT Int'l symposium on Software testing and analysis*, ISSTA '04, pages 206–216, New York, NY, USA, 2004. ACM.

[73] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10(2):123–165, June 1978.

[74] M. Harman and B.F. Jones. Search-based software engineering. *Journal of Information and Software Technology*, 43:833–839, December 2001.

[75] A. Arcuri. Insight knowledge in search based software testing. In *GECCO '09: Proceedings of the 11th Annual Conf. on Genetic and evolutionary computation*, pages 1649–1656, New York, NY, USA, 2009. ACM.

[76] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO '02: Proceedings of the 4th Annual Conf. on Genetic and evolutionary computation*, pages 1329–1336. Morgan Kaufmann Publishers, 2002.

[77] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[78] S. Rhys, S. Poulding, and J. Clark. Using automated search to generate test data for matlab. In *GECCO '09: Proceedings of the 11th Annual Conf. on Genetic and evolutionary computation*, pages 1697–1704, New York, NY, USA, 2009. ACM.

[79] Paolo Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*, pages 119–128, Boston, Massachusetts, USA, 11-14 July 2004. ACM.

[80] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Genetic and Evolutionary Computation Conference (GECCO 2004)*, pages 1400–1412, Seattle, Washington, USA, June 2004. LNCS 3103.

[81] Joachim Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275 – 298, 1998.

[82] Jungsup Oh, Mark Harman, and Shin Yoo. Transition coverage testing for Simulink/Stateflow models using messy genetic algorithms. In *Genetic Algorithms and Evolutionary COmputation Conference (GECCO 2011)*, pages 1851–1858, Dublin, Ireland, 2011.

[83] Mary Sheeran, Satnam Singh, and Gunnar Stlmarck. Checking safety properties using induction and a sat-solver. In Jr. Hunt, WarrenA. and StevenD. Johnson, editors, *Formal Methods in Computer-Aided Design*, volume 1954 of *Lecture Notes in Computer Science*, pages 127–144. Springer Berlin Heidelberg, 2000.

[84] C. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. In *Handbook of Knowledge Representation, Foundations of Artificial Intelligence*, pages 89–134. Elsevier, 2008.

[85] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conf. on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conf.*, pages 1194–1201, Menlo Park, August 4–8 1996. AAAI Press / MIT Press.

[86] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 216–226, New York, NY, USA, 1978. ACM.

[87] M. Qasem. Sat and max-sat for the lay-researcher. Technical report, The Public Authority for Applied Education and Training, Kuwait.

[88] T. Alsinet, F. Manya, and J. Planes. Improved branch and bound algorithms for max-sat. In *Proceedings of the Sixth Int'l Conf. on the Theory and Applications of Satisfiability Testing*, 2003.

[89] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[90] S. Rayadurgam and M.P.E. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*, pages 83–91. IEEE Computer Society, April 2001.

[91] H. Zhu and P.A.V. Hall. Test data adequacy measurement. *Software Engineering Journal*, 8(1):21–29, 1993.

[92] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *Software EngineeringESEC/FSE99*, pages 146–162. Springer, 1999.

[93] Matt Staats, Gregory Gay, Michael W Whalen, and Mats P.E. Heimdahl. On the danger of coverage directed test case generation. In *15th Int'l Conf. on Fundamental Approaches to Software Engineering (FASE)*, April 2012.

[94] M. Harman and J. Clark. Metrics are fitness functions too. In *10th Int'l Software Metrics Symposium (METRICS 2004)*, pages 58–69, Los Alamitos, CA, USA, 2004. IEEE Computer Society Press.

[95] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[96] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[97] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[98] F. Glover. Tabu search - part 1. *ORSA Journal on Computing*, 1:190–206, 1989.

[99] F. Glover. Tabu search - part 2. *ORSA Journal on Computing*, 2:4–32, 1990.

[100] J. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, MI, USA, 1975.

[101] Phil McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In *GECCO '09: Proceedings of the 11th Annual Conf. on Genetic and evolutionary computation*, pages 1689–1696, New York, NY, USA, 2009. ACM.

[102] J. Kennedy and R.C. Eberhart. Particle swarm optimization. In *IEEE Int'l Conf. on Neural Networks*, pages 1942–1948, 1995.

[103] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *DIMACS SERIES IN DISCRETE MATHEMATICS AND THEORETICAL COMPUTER SCIENCE*, pages 521–532, 1995.

[104] A. Murugesan, S. Rayadurgam, and M. Heimdahl. Modes, features, and state-based modeling for clarity and flexibility. In *Proceedings of the 2013 Workshop on Modeling in Software Engineering*, 2013.

[105] Boston Scientific. Pacemaker system specification. In *Pacemaker Formal Methods Challenge*. Software Quality Research Lab, 2007.

[106] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Klower Academic Press, 1993.

[107] A. Rajan, M. Whalen, M. Staats, and M.P. Heimdahl. Requirements coverage as an adequacy measure for conformance testing, 2008.

[108] J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608 –624, aug. 2006.

[109] A. Rajan, M.W. Whalen, and M.P.E. Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *Proc. of the 30th Int'l Conf. on Software engineering*, pages 161–170. ACM, 2008.

[110] J.H. Andrews, L.C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? *Proc of the 27th Int'l Conf on Software Engineering (ICSE)*, pages 402–411, 2005.

Appendix A

Obtaining the Source Code and Models

In the interest of allowing others to extend, reproduce, or otherwise make use of the work that we have conducted, the current version of our steering framework and experimental data—including the models, mutants, and tests—is freely available under the Mozilla Public License. This data and code is provided as-is, warts and all, but we are happy to answer questions or assist as much as possible in working with this information.

1. The experimental data is available from `http://crisys.cs.umn.edu/PublicDatasets.shtml`.
2. The source code—and binaries of the required dependencies—can be obtained from `https://github.com/Greg4cr`.

Appendix B

Glossary and Acronyms

Care has been taken in this thesis to minimize the use of jargon and acronyms, but this cannot always be achieved. This appendix defines domain-specific terms in a glossary, and contains a table of acronyms and their meaning.

B.1 Glossary

- **Software Test**—An artifact used to either argue for the correctness of software or to find faults in the software. A test case consists of test inputs (actions taken to stimulate the system), a test oracle (see below), and a set of steps needed to prepare the system for test execution.
- **Test Oracle**—An artifact that is used to judge the correctness of the system under test. A test oracle consists of *oracle information* and an *oracle procedure*.
  - **Oracle Information**—The data used by the oracle to judge the behavior of the system under test. In our work, we have used behavioral models as oracle information. Other oracles may use a set of assertions, or expected values for certain variables.
  - **Oracle Procedure**—A function that takes the oracle information and uses it to assess the behavior of the system under test, generally returning either a "pass" or "fail" verdict. This is often a comparison of the state of certain system variables to values given in the oracle information.
- **Test Suite**—A set of test cases.

B.2 Acronyms

Table B.1: Acronyms

Acronym	Meaning
FSM	Finite-State Machine.
MBO	Model-Based Oracle—a test oracle that uses a behavioral model as its source of oracle information.
MC/DC	Modified Condition & Decision Coverage—a form of structure-based testing based on exercising the conditional statements in a particular manner.
SBS(E/T)	Search-Based Software (Engineering/Testing).
SE	Software Engineering.
SUT	System Under Test.