Improving the Accuracy of Oracle Verdicts Through Automated Model Steering*

Gregory Gay, Sanjai Rayadurgam, Mats P.E. Heimdahl
Department of Computer Science & Engineering
University of Minnesota, USA
greg@greggay.com, [rsanjai,heimdahl]@cs.umn.edu

ABSTRACT

The *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 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 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 acceptable differences in behavior) and is based on a search process attempting to minimize a dissimilarity metric. This framework allows non-deterministic, but bounded, behavior 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, development costs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification

Keywords

Software Testing, Test Oracles, Model-Based Testing

*This work has been partially supported by NSF grants CNS-0931931 and CNS-1035715.

ASE'14, September 15-19, 2014, Västerås, Sweden.

http://dx.doi.org/10.1145/2642937.2642989.

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). Despite increased attention in recent years, the *test oracle problem* [16]—a 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 [9]. In addition, minor behavioral distinctions may have significant consequences [18]. 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 [21], typically expressed as state-transition systems, represent the system requirements by prescribing the behavior (the system state) to be exhibited in response to given input. Common modeling tools in this category are Stateflow [22], Statemate [15], and Rhapsody [1]. Models built using these tools are used for many purposes in industrial software development and, thus, their reuse as a test oracle is highly desirable. 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 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 the oracle verdict as needed. 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 [23]. 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, 31]. 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, as long as a set of constraints defining acceptable deviation are met. 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, building on ideas first proposed in [10]. We detail the implementation of the framework, and assess its capabilities on a model for the control software of an patient controlled analgesia pump (a medical infusion pump)—a real-world systems with complex, time-based behaviors. 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 lower testing costs and reduce development effort.

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 [17, 32]. 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 [16].

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 [16]. Commonly, such an oracle checks the behavior of the system against a set of concrete expected values [30] or behavioral constraints (such as assertions, contracts, or invariants) [7]. 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 [21].

A common practice during the development and testing of software is to build models of the intended behavior of the final system. Although such models are useful at all stages of the development process—particularly during requirements analysis—they are particularly useful for addressing two problems in testing: (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 allow us to cover a larger class of scenarios than we can cover manually [26]. As such, models are often executable; thus, in addition to serving as the basis of test generation [21], models can be used as a source of expected behavior, that is, used as a test oracle.

Executable behavioral models can be divided into *declarative* models created in formal specification languages, and *constructive* models built with state-transition languages such as Simulink and Stateflow, Statemate, finite state machines, or other automata structures [21]. Presently, we focus our work on constructive state-transition systems since these are frequently used to model real-time control systems—software that monitors and interacts with a physical environment [18].

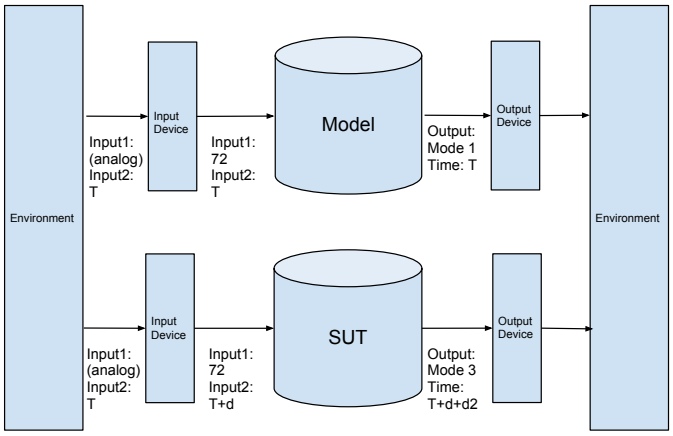

Figure 1: Illustration of abstraction induced behavioral differences between the model and the system under test.

Non-determinism is a major concern in embedded real-time systems. The task of monitoring the environment and pushing signals through multiple 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 model and a corresponding system depicted in Figure 1. A common abstraction when designing a model is to assume that all actions take place instantly, ignoring the reality of *computation time*. Both the model and the actual implementation receive the same stimuli from the physical environment, both process that input through the system, and then both issue output back to the environment. In the model, all actions are considered to have taken place at time step T. However, in the system, computations take time to be completed, and the processing of environmental stimulus through hardware layers and software subsystems adds an additional delay to the time at which the system receives that input. By the time that the system finishes responding to the stimulus, at time $T + (2\ delay\ periods)$, it is unlikely that the output fed to the environment matches the output that the model issued (in this case, not only do the output timestamps not match, but the SUT made a mode selection that is entirely different from that of the model). In systems such as pacemakers or infusion pumps, 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, such delays are commonly non-deterministic. Repeated application of the same stimulus may not result in the same output if processing time varies.

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 declar-

ative behavioral models in a formal notation such as Modelica. These solutions, however, have their limitations. Assertion-based approaches only ensure that a limited set of properties hold at *particular points in the program* [7]. 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 better account for time-constrained behaviors [11]. However, Miller et al. have found that developers are more comfortable building constructive models than formal declarative models [24]. Constructive models are visually appealing, easy to analyze without specialized knowledge, and suitable for analyzing failure conditions and events in an isolated manner [11]. 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 [26, 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. Practitioners are well-versed in these models and so these are likely to be less error-prone compared to building a new unfamiliar type of oracle. 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.

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.

One approach to 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 the simple computation-time abstraction in Figure 1, a filter could simply allow any timestamp within a certain range. 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. 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 judgement after the first few comparisons.

To address this problem, we take inspiration from *program steering*—the process of adjusting the execution of live programs in order to improve performance, stability, or behavioral correctness [12]. 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

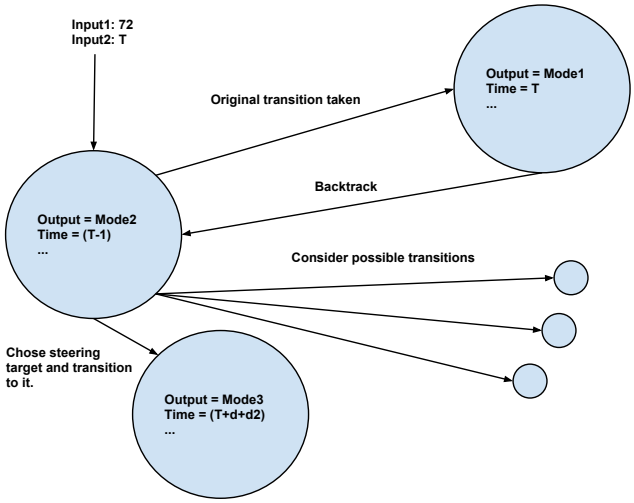

Figure 2: Illustration of steering process.

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). Oracle steering, unlike filters, is *adaptable*. Such 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 series of constraints it is possible to sufficiently bound steering so that the ability to detect faults is still retained.

First, a set of **tolerance constraints** governing the allowable changes to certain variables (input, internal, or output) in the model that can be affected by steering. These constraints define bounds on the non-determinism or behavioral deviation that can be accounted for with steering. For example, a pacemaker takes as input time-stamped sensed cardiac events and responds by setting the time of the next commanded pace (the pulse delivered to the heart). Then, if there are no additional sensed events within that time frame, the pacemaker will issue an electrical pulse to the appropriate chamber of the heart. If the model acts on the input after a slight delay and the model acts instantly, we might allow steering to adjust the time stamp on an input by up to four milliseconds. However, we might forbid steering from changing the pacemaker's response to that input (i.e., if the pacemaker responds with a status "event acknowledged and acted on", we might not allow the response to be changed to "premature sensed event"). If a pace command occurs at a different time in the system than it does in the model, we might—within a similar time window—allow steering to attempt to adjust the time that the pace command occurs.

Second, 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 $Dis(s_m^{new}, s_{sut}) < Dis(s_m, s_{sut})$. That is, within the bounds on the search space set by the tolerance constraints, we seek the candidate solution with the lowest dissimilarity score to the state of the SUT. There are many different functions that can be used to calculate dissimilarity. Cha provides a good primer on the calculation of dissimilarity [6].

Third, a further 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 ex-

```
1. for test in Tests
2.    for step in test
3.    initialVerdict = Dis(S_m, S_sut)
4.    if initialVerdict > 0
5.       oldState = S_m
6.       targetState=searchForNewState(Model,S_m,S_sut,Constraints,Dis())
7.       while Dis(targetState, S_sut) < Dis(oldState, S_sut)
8.          oldState = targetState
9.          targetState=searchForNewState(Model,S_m,S_sut,Constraints,Dis())
10.      transitionModel(Model,targetState)
```

Figure 3: Steps in the Steering Process

ists a steering action that results in a model state identical to that observed in the SUT.

In other words, the new state of the model-based oracle following the application of a steering action must be a state that is possible to reach within a limited number of transitions from the current state of the model, must fall within the boundaries set by the tolerance constraints, and must minimize the dissimilarity function.

We can illustrate steering using the system depicted in Figure 1. This system takes as input some environmental factor and a time stamp on when the reading was taken. It then outputs a mode choice and a time stamp on when the mode choice is issued. The model has abstracted computation time, and as a result, the model issues a mode choice and time stamp that differs from the SUT (the SUT is subjected to the time delay of the initial input to pass from the environment to the software and the time to perform computations). Although there is clearly non-conformance between the model-based oracle and the SUT, the SUT may still be operating within the bounds of the system requirements. Thus, as depicted in Figure 2, we can attempt to steer the model-based oracle.

As outlined in Figure 3, after obtaining our initial verdict by calculating the dissimilarity function (i.e., the Euclidean distance between the state of the SUT and the state of the model-based oracle), the steering procedure would backtrack to the state of the model before receiving the input $Input1 = 72$, $Input2 = T$ and evaluate the set of possible transitions from that state through the use of a search algorithm. The set of candidate solutions is limited by the set of tolerance constraints—for example, we might allow $Input1 <= Input1^{new} <= (Input1+5)$ (the value $Input1^{new}$ may fall up to 5 seconds after the original value of $Input1$). Then, the search selects from the remaining candidates through the use of the dissimilarity function. Finally, the candidate solution that results in the lowest observed value of $Dis(s_m^{new}, s_{sut})$ is chosen as the new state of the model-based oracle. Execution then resumes from the chosen state.

We have implemented the basic steering approach outlined in Figure 3. Our search process is based on SMT-based bounded model checking [13], which is a natural choice for this problem. We have a series of constraints that govern steering actions and seek to locate a model state reachable in a limited number of transitions that satisfies those constraints and minimizes a dissimilarity metric. Specifically, we make use of the Kind model checker [13] and the Z3 constraint solver [8].

A solution to the constraint $Dis(s_m^{new}, s_{sut}) < Dis(s_m, s_{sut})$ would give us a model state that is more similar to the behavior of the SUT than the original transition taken by the model-based oracle, but carries no guarantee that the satisfying state minimizes the dissimilarity metric. Thus, we first check the constraint $Dis(s_m^{new}, s_{sut}) = 0$; then, if an exact match cannot be found and if the steering policy allows inexact matches, we apply the model checker with this constraint in order to get an initial threshold, then iteratively reapply the model checker with new thresholds until we can no longer find a better solution. The best solution found then becomes the new state of the model-based oracle.

3.1 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, specifiying 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 nondeterminism into the model, steering decouples the specification of nondeterminism 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.

3.2 Automated Testing Framework

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

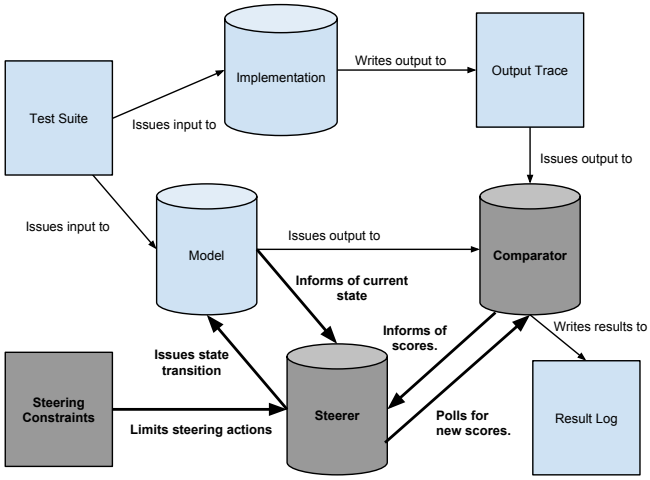

Figure 4: An automated testing framework employing steering.

the model and SUT). As illustrated in Figure 4, such a framework can be modified to incorporate automated oracle steering.

No matter the size of the model, steering will add additional overhead to the time required to execute tests. In practice, depending on the strictness of the constraints employed and the complexity of the model, execution with steering can take anywhere between a few additional seconds of execution to a few additional minutes. Therefore, we recommend that testing take place *offline*, rather than attempting to run the model and system in lockstep.

A framework would follow these steps:

1. Execute each test against the system under test, logging the values of select 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 interface with the model, backtracking execution and attempting to steer the model's execution within the specified constraints (perhaps consulting dissimilarity metrics used by the comparison module)
   (d) Compare the new output of the model to the SUT output and log the new dissimilarity score.
3. Issue a final verdict for the test.

4. RELATED WORK

Several authors have examined the use of behavioral models as test-generation targers for real-time systems [4, 20, 9, 29, 5]. If such models are used to generate tests, they can implicitly serve as a test oracle. For example, Larsen et al. [20] model a system as a non-deterministic timed automata that is constrained by an environment model. The combined model serves as an oracle during test execution. Arcuri et al. also examine the impact of the environment when testing process-control systems [4]. Their framework, which only models the environment that the system interacts with and eschews the system itself, allows limited non-determinism in the time that an action can take place, as well as certain forms of hardware-related non-determinism (through tester-provided probabilities of hardware failure). Savor and Seviora proposed a framework where the behavior of the SUT is compared to expected behaviors produced by a finite state machine derived from the system requirements [29]. Their framework can handle non-determinism in process communication by appending signals with an interval on the time of occurence. This interval is used to construct alternative legal event orderings. Briones and Brinksma treat quiescence—the lack of system output—as a special form of system output, and thus, offer a behavioral model that is able to capture a range of non-deterministic response times from the system under test [5].

While the above approaches consider forms of non-determinism, there are a few key differences with our proposed approach since it decouples the model from the rules governing steering. This decoupling makes non-determinism implicit and the approach more generally applicable. Explicitly specified non-deterministic behavior 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. Additionally, by not relying on a specific model format, steering can be made to work with models created for a variety of purposes.

Oracle steering is conceptually similar to *dynamic* program steering, the automatic guidance of program execution [12, 23]. Much of the 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 [12]. However, Kannan et al. have proposed a framework to assure the correctness of software execution at runtime through corrective steering actions [19]. Their framework serves as a supervisor on program execution, checking observed behavior at runtime against a set of assertions and adjusting the behavior of the program whenever an assertion is violated. This is done under the assumption that the SUT is mostly correct, and that only minimal control should be exercised. Although their system bears similarities to what we are proposing, our goals are very different—rather than adjusting the behavior of the live system, we apply steering to the test oracle in order to better identify fault-indicitive behaviors.

More relevant to our work is Microsoft Research's Spec Explorer [31] test generation framework. Their framework explores the possible runs of the executable model by applying steering actions to the model and guiding it through various execution scenarios with the goal of systematically generating test suites. It can then use the model as an oracle for the generated test by checking whether the SUT produces the same behaviors, steering through different execution scenarios for test generation as opposed to adjudging system execution. Although Spec Explorer also steers the actions of a behavioral model, their application and goals greatly differ from ours. They use steering to create tests, and do not apply it when checking conformance. Therefore, their framework cannot be used to address the instances of non-conformance related to non-deterministic execution of interest in this report.

5. CASE STUDY

We aim to gain an understanding of 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?

5.1 Experimental Setup Overview

We have based our model-based oracle on the management subsystem of a generic Patient-Controlled Analgesia (GPCA) infusion pump [25]. 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. This case example, developed in the Simulink and Stateflow notations and translated into the Lustre synchronous programming language [14], is a 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 1.

Table 1: Case Example Information

	# States	# Transitions	Lustre LOC
Infusion	23	50	6299

To evaluate oracle steering, we performed the following:

1. **Generated system implementations:** We approximated the behavioral differences expected from systems running on real embedded hardware by creating alternate versions of the 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 30 test steps 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 (Section 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 5.4).

5.2 System and Test Generation

To produce "implementations" of the example system, we created alternative versions of the model, introducing realistic non-deterministic timing changes to the systems. 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 original model and the "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—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).

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 1 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 a previous report on empirical software testing research [28], 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 [3].

Using a random testing algorithm, we generated 100 tests, each thirty test steps long. Each test represents thirty seconds of system activity—a length appropriate 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 in order to collect traces. In the models 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 labeled each test that fails pre-steering as failing due to an "acceptable timing deviation", an "unacceptable timing deviation", or a "seeded fault."

5.3 Steering Constraints

For this particular case study, 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 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.

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. In this study, we do not expect any sensor inaccuracy, so we do not allow freedom in adjusting sensor-based inputs. Similarly, 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.

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:

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

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.

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

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.

We must choose a set of variables to compare when calculating a dissimilarity score (or oracle verdict). As we cannot assume that the internal variables of the SUT and the model are the same, we calculate similarity using the five output variables of the infusion pump: 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.

5.4 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 simply 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-based oracle—meets these constraints, the filter will still issue a "pass" verdict for the test. 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.

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.

Table 2: Verdicts: T(true)/F(false), P(positive)/N(negative).

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 3: Initial test results when performing no steering or filtering. Raw number of test results, followed by percent of total.

Verdict	Number of Tests
Pass	11364 (74.8%)
Fail (Due to Timing, Within Tolerance)	1438 (9.4%)
Fail (Due to Timing, Not in Tolerance)	268 (1.7%)
Fail (Due to Fault)	2230 (14.7%)

The testing outcomes in terms of true/false positives/negatives are listed in Table 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:

$$Precision = \frac{TP}{TP + FP} \quad (3)$$

$$Recall = \frac{TP}{TP + FN} \quad (4)$$

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

$$\text{F-measure} = 2 * \frac{precision * recall}{precision + recall} \quad (5)$$

6. RESULTS AND DISCUSSION

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 3. As can be seen, 1,438 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 2,230 tests that indicated real faults.

Results obtained from the case study showing the effect of steering on oracle verdicts are summarized in Tables 4 and 5 respectively, for the two different distance metrics studied. The numbers

Table 4: Distribution of results for steering (Squared Euclidean dissimilarity). 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)	1286 (8.4%)	152 (1.0%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	268 (1.7%)
Fail (Due to Fault)	43 (0.3%)	2187 (14.4%)

Table 5: Distribution of results for steering (Manhattan dissimilarity). 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)	1198 (7.9%)	240 (1.6%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	268 (1.7%)
Fail (Due to Fault)	43 (0.3%)	2187 (14.4%)

Table 6: Distribution of results for step-wise filtering. Raw number of test results, followed by percent of total.

Initial Verdict	Pass (Post-Filtering)	Fail (Post-Filtering)
Pass	11364 (74.8%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	1286 (8.4%)	152 (1.0%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	268 (1.7%)
Fail (Due to Fault)	1253 (8.2%)	977 (6.4%)

Table 7: Precision, recall, and F-measure values.

Technique	Precision	Recall	F-measure
No Steering	0.63	1.00	0.77
Filtering	0.89	0.50	0.64
Steering - Euclidean	0.94	0.98	0.96
Steering - Manhattan	0.91	0.98	0.94

are presented as laid out in Table 2. Testing outcomes are first categorized according to the initial verdicts 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, or an unacceptable timing fluctuation or a fault. 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. Table 6 shows the corresponding data for the step-by-step filtering approach. Data from these tables lead to the precision, recall, and F-measure values, shown in Table 7, for the default testing scenario (accepting the initial oracle verdict), steering utilizing two different dissimilarity metrics, and filtering.

6.1 Allowing Tolerable Non-Conformance

According to Table 3, 11.1% of the tests (1,706 tests) initially fail due to timing-related non-conformance. Of those, 1438 tests (9.4% of the total) fall within the tolerances set in the requirements. Steering should result in a pass verdict for all those tests.

As the tables show, 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. For example, in Table 4 we see that steering using the square Euclidian distance would have correctly passed 1,286 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 0.63 when not steering, to 0.94 for the Squared Euclidean metric, and to 0.91 for the Manhattan metric, see Table 7).

Where previously developers would have had to manually inspect the more than 25% of all the tests (the sum of all "Fail" Table 4) to determine the causes for their failures (system faults or otherwise), they could now narrow their focus to the roughly 17% that still result in failure post-steering. Particularly given the large number of tests in this study, this reduction represents a significant savings in time and effort, removing between 1,198 and 1,286 tests that the developer would have needed to inspect manually. Still, there were a small number of tests that steering should have been able to account for (152 for the Squared Euclidean metric and 240 for the Manhattan metric). 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 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. 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.

Second, the dissimilarity metric does appear to play a small role in the effectiveness of steering. The Squared Euclidean metric resulted in higher precision than the Manhattan metric (0.94 vs 0.91), indicating that the former was better able to account for tolerable non-conformance. As there is no difference in recall between the two metrics, it appears that—for this case example—the use of the Manhattan metric results in a slightly more conservative steering process. With the Manhattan metric, the steering approach is more likely to refuse to steer. However, given the similarity in the performance of the two metrics, it appears that the set of constraints employed plays a more dominant role in determining the final outcome of steering.

The tolerance constraints applied 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 powerful steering is, there may be situations where differences in the internal design of the systems 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. For this experiment, we only compared the output variables. 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.

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 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.

In this case study, steering 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 tests. Although loss in recall is cause for concern when working with safety-critical systems, given the small number of incorrectly adjusted test verdicts, we hypothesize 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.

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 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 correct it. In such cases, it is unlikely that even a human oracle would label the outcome differently, as they are working from the same tolerance limits.

If care is taken when deriving the tolerance constraints from the system requirements, steering should not cover any behaviors that would not be permissable 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, allowable 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. 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, the results in Table 6 appear 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.

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 1,253 tests that should have failed—1,210 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 this 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.

As the output variables reflect current conditions, mounting internal differences may be missed, and the filter may not be able to cope with large-scale behavior differences that result from this steady divergence. Steering is able to prevent and account for 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 Tables 4 and 5. However, the addition of this internal variable dramatically changes the results of filtering. The new results can be seen in Tables 8 and 9.

Table 8: Distribution of results for step-wise filtering, (outputs + volume infused oracle). Raw number of test results, followed by percent of total.

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%)

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

Table 9: Precision, recall, and F-measure values for filtering (outputs + volume infused oracle).

Technique	Precision	Recall	F-measure
Filtering	0.64	0.76	0.70

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 barely more effective than not doing anything at all.

6.4 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 in Table 7. 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, 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.

Steering performs similarly 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. The Squared Euclidean metric results in a higher precision, while both metrics obtain an equal recall value.

Ultimately, we find that steering—employing the Squared Euclidean dissimilarity metric—results in the highest accuracy for the final test results. 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.

7. THREATS TO VALIDITY

External Validity: Our study is limited to one case example. We are actively working with domain experts to produce additional systems for future studies. We believe this systems 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 Simulink and Lustre as our modeling and implementation languages 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. 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 [27, 28], 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 [2].

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 provide an *idealized* view of the system, and may struggle to differentiate unexpected—but still acceptable—behavior from behaviors indicitive 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 transitions the model-based oracle, through a search process, to a state 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-based oracle—within limits—to match the execution of the SUT. Experiments, conducted over an infusion pump system, 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.

There is still much room for future work. We plan to expand on the selection of case examples in terms of both program size and scope of non-determinism, as well as examining:

- The impact of different dissimilarity metrics, tolerance constraints, and steering policies on oracle verdict accuracy;
- Improvements to the speed and scalability of the steering framework, including the use of alternative search algorithms;
- The use of steering and dissimilarity metrics as methods of quantifying non-conformance and their utility in fault identification and location.

9. REFERENCES

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

[2] J. Andrews, L. 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.

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

[4] A. Arcuri, M. Z. Iqbal, and L. 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.

[5] L. B. Briones and E. Brinksma. A test generation framework for quiescent real-time systems. In *IN FATES'04*, pages 64–78. Springer-Verlag GmbH, 2004.

[6] S.-H. 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.

[7] D. Coppit and J. 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.

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

[9] 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.

[10] G. Gay, S. Rayadurgam, and M. P. 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.

[11] A. Gomes, A. Mota, A. Sampaio, F. Ferri, and E. Watanabe. Constructive model-based analysis for safety assessment. *Int'l Journal on Software Tools for Technology Transfer*, 14(6):673–702, 2012.

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

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

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

[15] 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.

[16] 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.

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

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

[19] 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.

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

[22] MathWorks Inc. Stateflow. http://www.mathworks.com/stateflow.

[23] D. 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.

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

[25] 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.

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

[27] A. Rajan, M. Whalen, and M. 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.

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

[29] T. Savor and R. 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.

[30] M. Staats, G. Gay, and M. 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.

[31] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer, 2008.

[32] E. Weyuker. The oracle assumption of program testing.