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

Gregory Gay, Sanjai Rayadurgam, Mats P.E. Heimdahl, *Senior Member, IEEE*

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.

Index Terms—Software Testing, Test Oracles, Model-Based Testing, Model-Based Development, Verification

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 embedded systems—especially those that interact with a physical domain such as implanted medical devices. 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]. 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 exhibiting non-deterministic—but acceptable—behaviors.

Behavioral models [4], 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 [5], Statemate [6], and Rhapsody [7]. 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.

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 [8]. We hypothesize that behavioral models can be adapted for use as oracles for real-time systems through the use of *steering actions* that override

G. Gay is with the Department of Computer Science & Engineering, University of South Carolina. E-Mail: greg@greggay.com

S. Rayadurgam and M. Heimdahl are with the Department of Computer Science and Engineering, University of Minnesota. E-Mail: [rsanjai, heimdahl]@cs.umn.edu

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

the current execution of the model [9], [10]. 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. 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 oracle steering, expanding on previous research [10], [11]. In this manuscript, we present an updated steering framework, discuss the theory, current implementation, and implications of oracle steering in detail, and present a case study where we examine the effectiveness of steering on two systems with complex, time-based behaviors—the control software of a patient controlled analgesia pump (a medical infusion pump) and a pacemaker. We also present an automated method for learning the constraints that guide the steering process.

Case study results indicate that steering improves the accuracy of the final oracle verdicts—outperforming both default testing practice and a filtering technique. 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.

Our results indicate that the choice of constraints limiting the choice of steering actions has a major impact on the ability of steering to account for allowable non-determinism. Relatively strict, well-considered constraints strike the best balance between the ability to account for non-determinism and the risk of masking faults. As constraints are loosened, steering may be able to account for more acceptable deviations, but it will often mask more faults or even choose suboptimal steering actions that cause undesired side-effects.

For developers are unsure of what constraints to employ, we offer a technique that can automatically learn a set of constraints through a process known as *treatment learning*. Given a set of developer-classified test cases, we can extract information on the steering actions chosen when no constraints are employed, and apply a treatment learner to identify the steering actions highly correlated to correct test verdicts. 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 recommend the use of steering as a tool for focusing and streamlining the testing process.

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* [12], [13]. A *test oracle* is a predicate 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 [4].

Although behavioral models are useful at all stages of the development process, they are particularly effective in addressing testing concerns. Models allow testing activities to begin before the actual implementation is constructed, and models are suited to the application of verification and automated test generation techniques [16]. As models are often executable, in addition to serving as the basis of test generation [4], models can be used as a source of expected behavior—as a *test oracle*.

An example of such a model can be see in Figure 1. This model, written in the Stateflow notation [5], represents the behavior of a simplified pacemaker—a medical device that regulates the heart rate of a patient by issuing electrical stimuli (paces) in the absence of natural activity. This pacemaker regulates a ventricle by polling sensors every millisecond (timestamped with $timeIn$) for a sensed event ($sense$), and issuing paces ($pace$) as regulated by the Lower Rate Limit (LRL)—a user-specified desired pace per minute rate. Paces are timestamped using the variable $timeOut$. Following a sense or pace, the pacemaker will ignore all sensor values for a user-specified length of time—VRP, or the Ventricular Refractory Period—to avoid acting on electrical noise. We will refer to this example throughout this work as SimplePacing.

Non-determinism is a major concern in embedded time-based 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 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 that 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. It is common 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.

As a simple example, in this model, the time that the sensor is polled is the same as the time that output is issued ($timeIn = timeOut$). This is unlikely to be true in the actual software—differences may arise from computation time, clock drift, and the difficulty of synchronizing the paral-

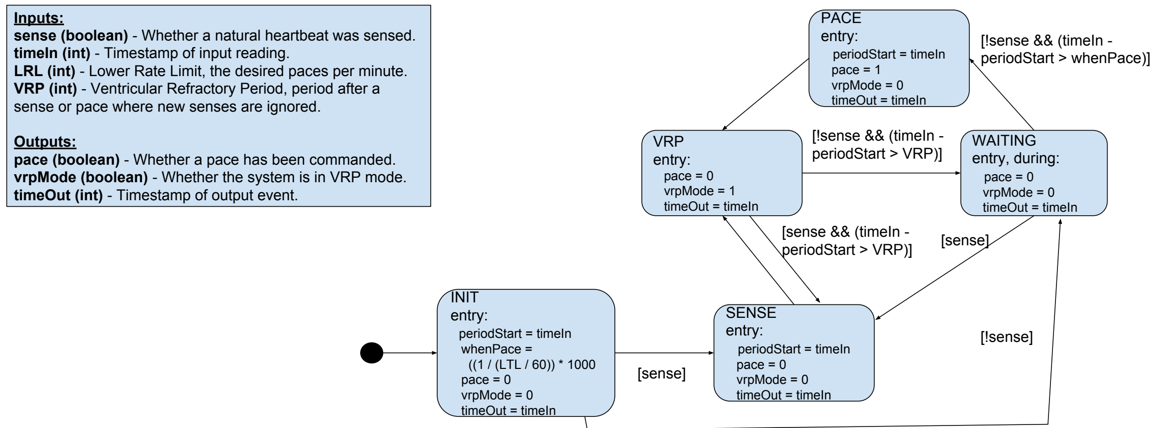


Fig. 1: SimplePacing model—a system that delivers paces (electrical impulses) at a prescribed rate in the absence of sensed ventricular activity.

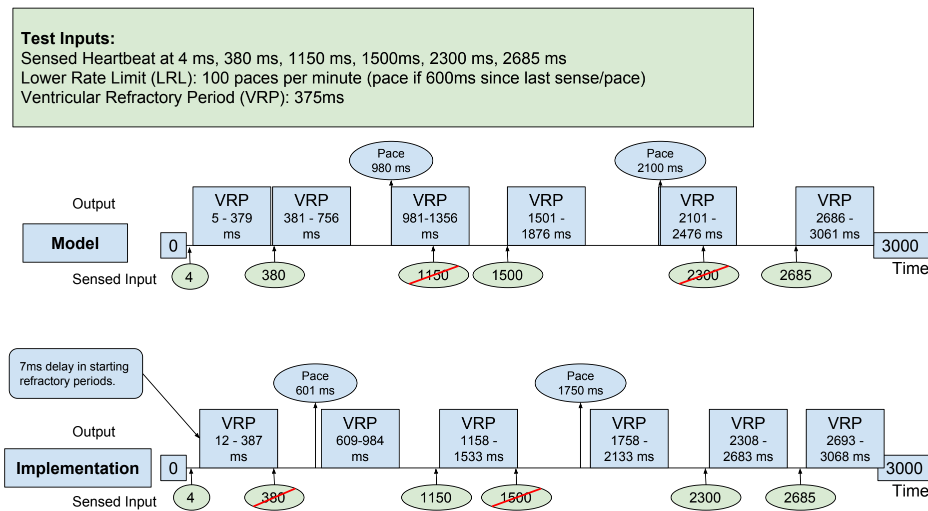


Fig. 2: Abstraction-induced behavioral differences between SimplePacing's model and implementation during execution of a test case. The Lower Rate Limit describes the minimum number of paces per minute to be issued by the pacemaker. In the absence of a sensed heartbeat, a pace will be delivered every 600 ms. The Ventricular Refractory Period is a length of time after a sense or pace where further senses are ignored, as they might be aftershocks or other noise from the last measured activity.

lel components of the software—but assuming instantaneous computation time (or a constant computation time) is a common abstraction when modeling. The SimplePacing model receives a simple binary sense. However, in the real world, the electrical impulses being sensed are complex noise-prone analog readings that the system must decide how to interpret. These omitted or simplified details may manifest themselves as differences between the behavior defined in the model and the behavior observed in the implementation during test execution. Furthermore, such behaviors are commonly non-deterministic. Repeated application of the same test stimulus may not result in the same output if, say, processing time varies.

An example of this is illustrated in Figure 2, where the behavior of the SimplePacing model and implementation differ during test execution. In this case, the difference is simple—minor computation delays result in a situation where there is a 7ms delay in starting a Ventricular Refractory Period following a sense or pace. The developers of systems often recognize the likelihood of such scenarios and prescribe a *tolerance period*, a bounded period of time where the activity is still legal, in the specification. In this case, they might declare that the SUT is correct as long as the start and end of the VRP falls in a +/- 8ms window of time of when it is expected to occur. In fact, this is the actual tolerance prescribed in a publicly available pacemaker specification [17]. In this case, the implementation—despite the small delay—initiates a VRP within the required window of time. However, as the comparison procedure expects the model and system to conform, the test will fail.

If this were the sole difference between the model and system's behavior, it would be easy enough to adjust the model or comparison to account for this difference. However, the behavior of SimplePacing—and many critical embedded or cyber-physical systems—is *stateful*. The behavior at one

step of execution is dependent on the actions and reactions of the system at all previous execution steps. As can be seen, execution over the entire life of the test case differs greatly between the two systems. Because of the delay in starting the VRP, the input at 380 ms is ignored by the implementation. The model receives this input 5 ms after its VRP ends, and it starts a new VRP. However, because that input was ignored, the implementation issues a pace at 601 ms, while the model does not do so until 980 ms. In both cases, the behavior exhibited by that artifact is "correct" with respect to the specifications, but the two diverge significantly as execution proceeds. In this case, the source of the divergence is clear, and the implementation is consistent in how it acts. In practice, such issues are often more complex—the implementation might vary when the VRP starts and ends non-deterministically, it might react to a sense later than intended, or it might issue a pace either earlier or later than expected.

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 [18]. 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]. In some cases, models can more easily account for a wider range of input scenarios. Xie and Memon found that oracles that incorporate state information (such as models) can have higher fault-detection capabilities than simpler oracles—even when shorter test cases are used [19]. 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 some forms of non-deterministic behavior [20]. However, Miller et al. have found that developers are more comfortable building constructive models than formal declarative models [21]. Constructive models are easier to analyze without specialized knowledge and suitable for analyzing failure conditions and events in an isolated manner [20]. The complexity of declarative models and the knowledge needed to design and interpret such models make widespread industrial adoption of the paradigm unlikely.

More importantly, it is a widely held view that constructive models are indispensable in other areas of development and testing, such as requirements analysis or automated test generation [16], [22]. From this standpoint, the motivational case for models as oracles is clear. If these models are already being built—and are useful throughout development and testing—their reuse as 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. For these reasons, model-based approaches have become common in the development of critical systems [23]—systems that are particularly likely to demonstrate the type of non-determinism discussed previously. Therefore, we seek a way to use constructive model-based oracles even when faced with non-determinism introduced during system execution on the target hardware.

3 Oracle Steering

We would like to distinguish between correct, but non-conforming, and fault-indicative behaviors when using a model-based oracle. A simple approach that could potentially address this would be to augment the comparison with a filter that detects and ignores acceptable differences on a per-step basis. For example, if the model and SUT produce behavior that differs only by the timestamp, the filter could allow this difference as long as it falls within a bounded time range. Such filters are relatively common in GUI or graphic rendering testing [24][1]. This is an example of an *indirect* solution—a filter selectively overrides the oracle, but does not interfere with the internal execution of the model.

However, an issue with indirect solutions like filtering is that many of the systems we are interested in not only demonstrate non-determinism, but are stateful. As can be seen in Figure 2, the non-deterministic shift in a single refractory period can influence the execution of the system for the entire test case, leading to irreconcilable differences between the SUT and the model-based oracle. If state has a minimal impact, or the sources of non-determinism are simple and isolated, then a filter is an appropriate solution. If the refractory period is always delayed in the same way, then a filter could simply account for that delay. In practice, however, the beginning and end of that period may change constantly. At the same time, input interactions and other timed events are still occurring that influence system behavior. That shift in the refractory period is likely to be just the first of a series of divergences introduced because the real system is more complex than the model. Indirect actions may not be effective at handling these behavior-impacting events that grow and build off of each other as time progresses. Any potential solution must account for not just a divergence between the SUT and the model-based oracle at a single time step, but with all previous divergences as well.

Rather than indirect actions, such as filtering, we believe that the solution is *direct* action that adjusts the behavior of the model throughout execution to better reflect the details of the actual operating environment of the system. We take inspiration from *program steering*—the process of adjusting the execution of live programs in order to improve performance, stability, or behavioral correctness [25]. 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.

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

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

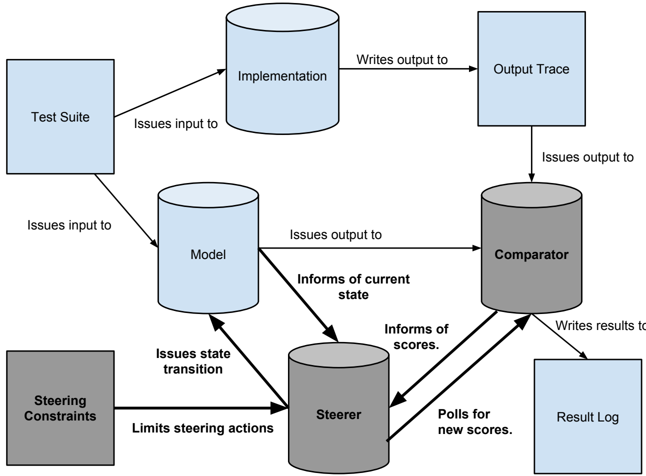


Fig. 3: An automated testing framework employing steering.

Input: $Model, SUT, Tests$
1 **for** $test \in Tests$ **do**
2 $stepNumber = 0$
3 **for** $step \in test$ **do**
4 $previousState = state(Model)$
5 $applyInput(SUT, step)$
6 $applyInput(Model, step)$
7 $S_m = state(Model)$
8 $S_{sut} = state(SUT)$
9 **if** $Dis(S_m, S_{sut}) > 0$ **then**
10 $instrumentedModel = instrument(Model, previousState)$
11 $steer(instrumentedModel, S_m, S_{sut})$
12 $S_m^{new} = state(Model)$
13 **if** $Dis(S_m^{new}, S_{sut}) > 0$ **then**
14 $verdict = fail$
15 $break$
16 $verdict = pass$

Fig. 4: Testing process when steering is employed.

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, rather than immediately issuing a failure verdict, we will attempt to correct the behavior of the model.

Each time a divergence occurs, the model is steered to account for the mismatch between model and system. Unlike with indirect solutions, the complexity of handling non-determinism is reduced because the model is guided to adapt to the real-world execution of the system as such events occur. By doing so, steering provides the 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 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 artifacts:

1) A set of **tolerance constraints** limiting the acceptable values for the *steering variables*—a set of *model* variables that may be directly manipulated.
2) A **dissimilarity function**—a numerical function that compares a candidate model state to the state of the SUT and delivers a score. We seek the candidate solution that minimizes this function.
3) A set of generic **policies** that can control 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 (a *failure* if any discrepancies are detected and a *pass* if a test executes without revealing any differences between the model and SUT).

Such a framework can be modified to incorporate automated oracle steering. The updated testing process is detailed in Figure 4 and illustrated in Figure 3, with steering-related components shaded in dark gray. In this updated framework, the comparison mechanism issues a dissimilarity score instead of a boolean verdict. If the output does not match, the steering algorithm will instrument the model and attempt to find an action that minimizes that score. If the model and SUT cannot be aligned, the comparator will log the final dissimilarity score.

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 [5]. When steering, we translate the Stateflow models to the Lustre synchronous programming language to more easily perform automated transformations (see Section 5.6).

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 must be steerable. Variables internal to the model and output variables may be

specified as steerable, but they are transformed into input variables for the computation steps where steering is applied. This transformation enables search algorithms to directly assign values to these variables. On subsequent, unassisted execution steps, the model will be again transformed such that those variables are again internal to the model.

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 chosen steering action is an assignment to the variables in Π that satisfies the tolerance constraints, minimizes the dissimilarity function, and follows any additional steering policies (for example, it may need to meet some dissimilarity threshold to be chosen).

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. However, implicitly, these constraints limit the values that can be assigned to Π.

Consider the scenario outlined in Figure 2. We could use steering to correct VRP-related mismatches by defining VRP as a member of Π and allowing the search algorithm to assign a new value to it. As we want to limit the change that can be imposed on VRP, we must set a constraint. One reasonable choice would be to allow the new value of VRP to fall anywhere between a minimum of $(VRP^{original} - 8ms)$ and a maximum of $(VRP^{original} + 8ms)$.

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. By adjusting the model each time it diverges, we eliminate the need to track the entire execution history.

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 $Dis(s_m^{new}, s_{sut})$. There are many functions that can be used to calculate dissimilarity. Cha provides a good primer on the calculation of dissimilarity [26]. As we primarily used models 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]) are 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, there must exist a steering action that results in a model state identical to the SUT state.

These policies are intended to capture constraints that can be applied to control steering, regardless of the system under test. While such policies could, in many cases, be expressed as part of the tolerance constraints, they have been separated and expressed as generic options that can be easily enabled or disabled when executing the steering framework. The ability to invoke options like the one above gives a user a set of options to try when they begin to explore steering.

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 process is illustrated in Figure 5 for one test step from the test in Figure 2 (when input occurs 380 ms into the test). SimplePacing is instrumented such that 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 VRP, with the tolerance constraint that the chosen value of VRP must fall between $VRP^{original} - 8ms$ and $VRP^{original} + 8ms$. 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 6. Our search process is based on bounded model checking [28], an automatic approach to property verification for concurrent, reactive systems [29]. The problem of identifying an appropriate steering action can be expressed as a Satisfiability Modulo Theories (SMT) instance, a generalization of a boolean satisfiability instance in which atomic boolean variables are replaced by predicates expressed in classical first-order logic [30]. 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 then the original. We have made use of the Kind [28] model checker, and the Z3 constraint solver [31].

When we execute tests, each test step is checked explicitly for conformance violations by comparing the state of the oracle and the SUT. If a mismatch is detected, we allow the

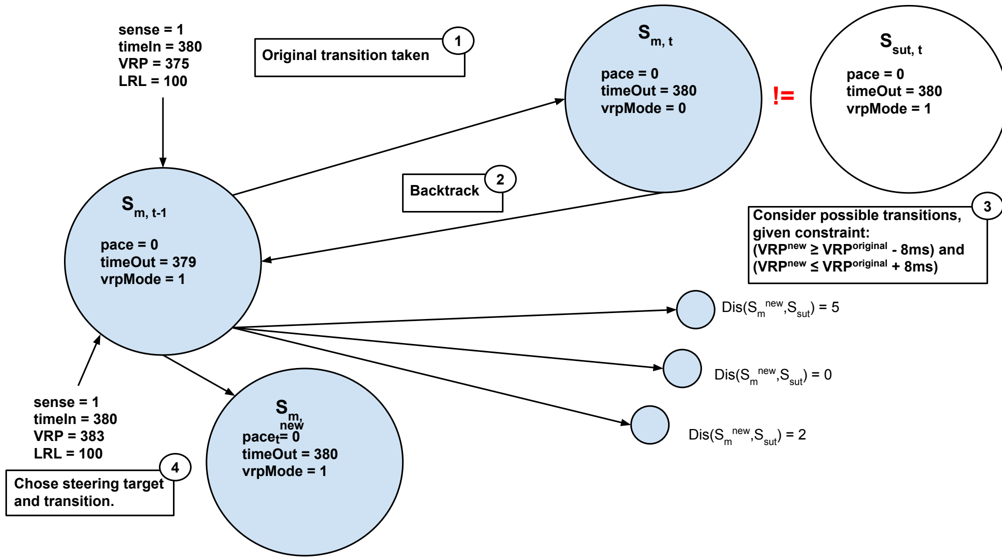


Fig. 5: Illustration of steering process for SimplePacing (Figure 1) for one step of the test depicted in Figure 2.

Input: $Model, S_m, S_{sut}$

1 **if** $Dis(S_m, S_{sut}) > 0$ **then**

2 $\quad DisConstraint := \lambda threshold \rightarrow \lambda state \rightarrow Dis(state, S_{sut}) \leq threshold$

3 $\quad targetState := searchForNewState(Model, S_m, S_{sut}, Constraints, DisConstraint(0))$

4 $\quad$ **if** *targetState* = *NULL* **then**

5 $\quad\quad newState := S_m$

6 $\quad\quad T := 1$

7 $\quad\quad$ **while** $newState \neq NULL$ **do**

8 $\quad\quad\quad targetState := newState$

9 $\quad\quad\quad newState := searchForNewState(Model, targetState, S_{sut}, Constraints, DisConstraint(T \times Dis(targetState, S_{sut}))$

10 $\quad\quad\quad T := 0.5 \times T$

11 $\quad transitionModel(Model, targetState)$

Fig. 6: Outline of steering process.

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. Constants are also embedded in the model for each steerable variable and output variable in the model and each output variable in the SUT describing what occurred originally during test execution. That is, they tell us what happens when we do not steer. These values are used both for calculating the value of the dissimilarity score and in the tolerance constraints[2].

An expression is formulated containing the tolerance constraints and the threshold that we want the new dissimilarity score to beat. This expression is then 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 the new values of the steerable variables) to the instrumented model, record new variable values in the trace, and continue test execution.

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 with a lower dissimilarity score. As outlined in Figure 6, 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 $S_m^{new} = S_{sut}$. If an exact minimization cannot be found, and our policies still allow steering in that situation, 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 found. Once that constraint can no longer be satisfied, we take the last viable solution 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 better solution. The best solution found will be selected as the steering action.

It should also be noted that the results and stability of the search process depend on the algorithm employed. The solver used in our implementation, Z3, returns deterministic results. Other algorithms may not return the same steering action each time a test is executed.

3.3 Defining Constraints

The efficacy of the steering process depends 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 important that the constraints to be employed are carefully considered.

2. An example of this instrumentation and additional technical details can be seen in [32].

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 constraint on that sensor input and allow the steering algorithm to try values within that range.

We recommend that users err toward strict constraints to avoid masking faults. To give an example, while the inputs of a medical device may include a patient prescription, we would recommend that steering be prohibited from altering the prescription values, as manipulation of those might threaten the life of a patient. Instead, the focus of constraints should be on areas where adding flexibility to the oracle cannot harm a patient, such as minor timer or sensor adjustments.

Constraints may be defined over any of the variables of the system (input, output, or internal to a function). When no constraints are specified for a variable, no steering actions may be taken. We recommend that testers start with a minimal set of strict constraints, then loosen them until the number of false-failure verdicts is sufficiently reduced. While it is possible to steer the value of output variables directly, we do not generally recommend doing so. It is safer to steer the factors leading to unexpected output than to directly overwrite that output. As steering decouples the specification of allowable non-determinism from the model, testers can experiment with different sets of constraints and policies. Alternative options can easily be explored by swapping in a new constraint file and executing the test suite again until they are confident in their selections.

As the software changes during development, tests must often be altered to account for changes in SUT interfaces or other internal behavioral changes. As the constraints employed by steering are independent of the test cases themselves, the use of steering does not increase the required *test* maintenance effort. However, if changes to the SUT make certain steering actions illegal, alter the behavior of system variables, or remove variables used in constraints, then some *constraint* maintenance must take place. As we recommend minimal sets of constraints, the required level of constraint maintenance should be low.

3.4 Handling Non-Determinism Through Steering or the Underlying Model

By incorporating a sophisticated model of time or other forms of non-determinism, some of these modeling approaches discussed in Section 8 could account for a subset of the non-deterministic behaviors induced during execution of the SUT—particularly variance related to timing issues. A natural question, then, is whether to use steering or to simply incorporate this non-determinism into the underlying model.

In such cases, the model *must* be built with those execution behaviors in mind. These behaviors depend specifically on the particular details of the real-world execution—such as the underlying hardware platform. Such details are difficult to anticipate, and assumptions would need to be made at the time that the model is constructed. If the model is built—as many are—during requirements engineering, then it is difficult, if not impossible, to make correct assumptions. Fixing incorrect assumptions may require extensive overhaul of the model's structure. This task may need to be performed on a regular basis as the 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.

There is an argument to be made for not building these details directly into the model in the first place. Effective requirements analysis requires *clarity*. Being forced to incorporate the full scope of the non-deterministic behavior of the SUT into the model could muddle its clarity, making the analysis of the requirements more difficult and potentially leading to an incorrect implementation.

Through the use 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. Additionally, by not relying on a specific model 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.

Once allowable behavioral deviations have been observed, if a non-deterministic model is employed, one could choose to either steer to temporarily account for such situations or update the model to permanently account for such situations. These options are not mutually exclusive. Steering could be used to refine the tester's understanding of the behavioral divergences they see in practice. After using steering, they could update the model. In that case, steering is useful as a discovery tool. If, for example, different hardware platforms are being considered, it may be wise to use steering until stable decisions have been reached. This prevents the need to make frequent—potentially difficult—model revisions.

4 Learning Constraints

Choosing the correct constraints is essential to accurate steering, but it is not always clear what those constraints should be, or even what variables should be manipulated 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, as the developers will have a more

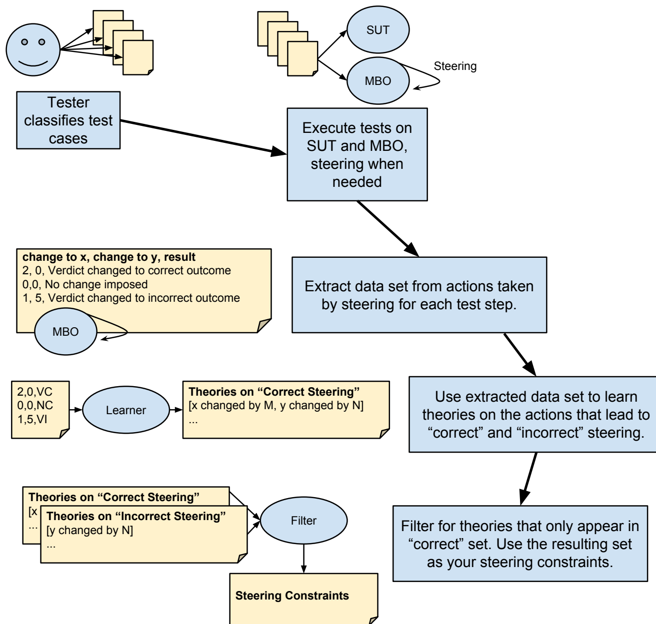


Fig. 7: Outline of learning process.

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.

It is possible to use human-classified test verdicts to *learn* an initial set of constraints. We can treat constraint elicitation as a machine learning problem where we 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 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 data set that can be explored by a variety of learning algorithms. This process is illustrated in Figure 7. The data we extract at each test step is detailed in Table 1. For each steerable variable, we record how much it was altered by steering. For each oracle-checked variable, we record both how much it differed between model and system before steering and how much it was changed after steering. Then we record a classification—did steering perform the correct action? The classification $ChangedCorrect$ is assigned when steering acted and arrived at the same verdict as the human. The classification $ChangedIncorrect$ is assigned when the post-steering verdict does not match the human-assigned verdict. We also record data when steering does not act at all or does not change the pre-steering verdict. This is assigned the classification $NotChanged$.

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 [33]. As new evidence is examined, the accuracy of these categorizations is then refined.

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 [34]. Treatment learning approaches take the classification of an observation and try to reverse engineer the evidence that led to that categorization. Such learners produce a *treatment*—a small set of data value boundaries that, if imposed, will produce a subset of the original data matching the desired classification.

Ultimately, classifiers strive to increase accuracy by growing a 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, when imposed, causes the largest impact. This focus is exactly what makes treatment learning interesting as a method of producing constraints. We wish to constrain steering to a small set of steerable variables, with strict limitations on allowed value changes. Treatment learning can be used to generate a minimal initial set of constraints or to tune an existing set of constraints.

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 on the left in Table 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—when it changes the oracle verdict in a way that matches the human-assigned verdict. We can extract treatments, ranked by their score assigned by the treatment learner's objective function.

Example treatments are shown in Table 3. Each treatment includes variables and their value ranges that are correlated to the targeted classification. The first treatment in Table 3 states that the major indicators of a correct steering action are when the value of Variable1 is reduced between 0-4 and the value of Variable2 is increased by 1-2 by steering. By imposing that treatment, we end up with the class distribution shown on the right in Table 2. This class distribution shows strong support for the produced treatment. By allowing steering to change Variable1 and Variable2 in the prescribed manner, steering tends to match the human-assigned verdict.

In order to create a set of tolerance constraints, we first create 10 treatments like those seen in Table 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 range pairings. Treatment learning algorithms operate scholastically, so 10 treatments are produced to account for randomness. Some of these items may not actually be indicative of successful steering—they

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 1: Data extracted for tolerance elicitation. The classification $ChangedCorrect$ means that the post-steering verdict matched the human-assigned verdict, while $ChangedIncorrect$ implies that the post-steering verdict does not match. $NotChanged$ means that steering does not act at all or does not change the pre-steering verdict.

Class	Percentage
NotChanged	96%
ChangedIncorrect	1%
ChangedCorrect	2%

Class	Percentage
NotChanged	0%
ChangedIncorrect	14%
ChangedCorrect	86%

TABLE 2: Base class distribution and class distribution of the remaining subset of the data after imposing a treatment.

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

TABLE 3: Examples of learned treatments. Treatments are assigned a "goodness" score and sorted by that score. Each treatment includes variables and their value range that is correlated to the targeted classification.

```
((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)
```

Fig. 8: Examples of produced tolerances. $concrete_oracle_X$ is a constant reflecting the value of that variable when steering is not employed.

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 8.

Notable treatment learning algorithms include the TAR family (TAR2 [34], TAR3 [35], [36], [37], and TAR4.1 [38])—a series of algorithms utilizing a common core structure, but employing different objective functions and search heuristics—and the STUCCO contrast-set learner [39]. Other optimization algorithms have also been applied to treatment learning, including simulated annealing and gradient descent [38]. In this work, we employed the TAR3 algorithm.

5 Case Study

We aim to assess the capabilities of oracle steering and the impact it has on the testing process. 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?
4) To what degree does the strictness of the employed constraints impact the correctness of steered oracle verdicts?
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 medical device systems. The first is the management subsystem of a generic Patient-Controlled Analgesia (GPCA) infusion pump [40]. 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 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 [17]. This subsystem monitors cardiac activity and, at appropriate times, commands the pacemaker to provide electrical impulses to the appropriate chamber of the heart.

These models are developed in the Stateflow notations and translated into the Lustre synchronous programming language to more easily perform automated transformations [41]. The simplicity and declarative nature of Lustre make it well-suited to model checking and verification [28]. This also makes it an ideal language to use as a target for steering because the steering constraints and dissimilarity function can be encoded directly into the model, and a steering action can be selected using the same algorithms that are regularly used to perform verification. 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.

Both case examples are complex real-time systems of the type common in the medical device domain. Details on the Stateflow and translated Lustre models are provided in Table 4.

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

	# States	# Transitions	Lustre LOC	Input Variables	Internal Variables	Output Variables
Infusion_Mgr	23	50	6,299	19	825	5
Pacing	48	120	24,017	18	545	5

TABLE 4: Case example information—number of states, number of transitions, lines of code when translated to Lustre, number of input variables, number of internal variables, and number of output variables.

1) **Generated system implementations:** We approximated systems running on embedded hardware by creating versions of each model with non-deterministic timing elements. We also generated 50 mutated versions of each version of each system 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 (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 precision 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. Non-determinism was simulated in this study in order to more clearly examine the effectiveness of steering under controlled experimental conditions. 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 SUT 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 integer 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 [42], and are similar to those used by Andrews et al, where the authors found that generated mutants are a reasonable substitute for real faults in testing experiments [43].

Using a probabilistic random testing algorithm, we generated 100 tests for each system. Each test captures a period of time long enough to observe complex time-sensitive behaviors, but still short enough to yield a reasonable experiment cost. 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. The generation algorithm allows testers to assign probabilities to particular input values or ranges. In this case, these probabilities were chosen to reflect "normal" execution of the system, rather than extreme events—for example, Infusion_Mgr's power is very unlikely to be turned off during a test case. For this study, normal execution was favored to clearly examine the influence of steering.

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 behavioral mismatch is due to a 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

We have made use of two dissimilarity metrics when comparing states. 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 minimum and maximum values for each variable.

When computing a dissimilarity metric—or, for the matter, an oracle verdict—we must choose a set of variables to compare. As we cannot assume a common internal structure between the SUT and the model, we calculate similarity using the output variables of both. 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. These constraints were developed using the actual software and hardware specifications for each system and by consulting with domain experts. The chosen tolerance constraints for the Infusion system include:

- There are five 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 between one second shorter and two seconds longer than the original prescribed duration.

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 take place up to four milliseconds after the original timestamp.
- Boolean input variables indicate 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 filtering conducted by the SUT.

These constraints reflect what we consider a realistic application of steering—we expect issues related to 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 input. For Pacing, we expect a small amount of event reordering and noise sensitivity, so we allow a small amount of freedom in changing sensor-based values. As these are restrictive constraints, we deem these the **Strict** tolerance constraints.

In order to assess the impact of different sets of 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 input 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 reflect a gradual relaxation of the limits on steering, and can demonstrate the impact that the choice of constraints has on the effectiveness of steering.

5.5 Learning Tolerance Constraints

In Section 4, we discussed the process of using treatment learning to extract a set of tolerance constraints. We wish to

know whether we can learn tolerances for our case studies, 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 stochastic 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 for each system (4 constraint levels x 10 repeats x 2 metrics).

We generate tolerances using the TAR3 treatment learning algorithm [38], [35], [36], [37]. It produces treatments by first being fed a set of training examples. Each example 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. As in Section 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 create a set of tolerance constraints, we first generate 10 treatments—like those seen in Table 3—that indicate the correct use of steering to adjust the state of the model. Some of these items may not actually indicate successful steering—they may be steering actions that occur at the same time as actions that are actually good. Thus, we also produce 10 treatments that correspond to incorrect steering actions, and remove any treatments that appear in both the "good" and "bad" sets. 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 8. 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 output during each test, compared the results to the values of the same variables in the model-based oracle at each time step to calculate the dissimilarity score, and issued an initial verdict[3]. 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. 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.

3. This corresponds to "oracle strategy 2" in Li and Offutt's hierarchy—checking the return values of methods invoked during each transition [44]

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

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.

The testing outcomes in terms of true/false positives/negatives are listed in Table 5. 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:

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

6 Results and Discussion

As previously presented in Table 5, 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 tests 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 the "No Adjustment" column of Table 6. 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 indicated real faults.

A similar breakdown can be found for the Pacing system in the "No Adjustment" column of Table 7. About a third of the test executions—2,992 in total—pass initially. A further 21%, or 2,208 tests, fail due to 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 or filtering on oracle verdicts are summarized in Table 6 and 7 respectively, for the Infusion_Mgr and Pacing systems. For both systems, the two dissimilarity metrics performed identically when steering. The raw results are presented as laid out in Table 5. 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. Data from these tables lead to the precision, recall, and accuracy values are shown in Table 8 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 Performance

By necessity, steering adds overhead to the test execution process. While steering is performed largely offline—using traces gathered from the SUT—the execution time required to perform steering should not be onerous. For tests where steering is not performed, the framework takes an average of 0.008 seconds of processing time. For tests where steering is employed, execution takes an average of 13.54 seconds for Infusion_Mgr and 162.91 seconds for Pacing on a workstation with an Intel Core i7-4790 four core CPU clocked at 3.60GHz and 16 GB of RAM. For simpler models, such as Infusion_Mgr, this is not a substantial increase in execution time. However, we would like to improve performance on larger models. This will be a focus of future work.

6.2 Allowing Tolerable Non-Conformance

For the Infusion_Mgr system—according to the "No Adjustment" category of Table 6—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 the "No Adjustment" column of Table 7).

As Table 6 shows, for both dissimilarity metrics, steering is able to account for almost all of the situations where it should be able to correct the model. 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 8).

Similar results for steering on the Pacing system can be seen in Table 7 . Steering correctly changes the verdicts of 2,065 of the tests that initially failed. As shown in Table 8, 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) 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 manageable 47% (the sum of all remaining post-steering "fail" verdicts in Table 7).

Given the large number of tests in this study, this reduction represents a significant savings in time and effort, removing

Initial Verdict	No Adjustment	Pass (Post-Adjustment)		Fail (Post-Adjustment)	
		Steering	Filtering	Steering	Filtering
Pass	11,364 (74.8%)			0 (0.0%)	
Fail (Due to Timing, Within Tolerance)	1,406 (9.2%)	1,245 (8.2%)		152 (1.0%)	
Fail (Due to Timing, Not in Tolerance)	268 (1.8%)	0 (0.0%)		268 (1.8%)	
Fail (Due to Fault)	2,229 (14.6%)	43 (0.3%)	1,252 (8.2%)	2186 (14.3%)	977 (6.4%)

TABLE 6: Results when accepting the default verdict, steering, and filtering for Infusion_Mgr. Steering results identical for both dissimilarity metrics.

Initial Verdict	No Adjustment	Pass (Post-Adjustment)		Fail (Post-Adjustment)	
		Steering	Filtering	Steering	Filtering
Pass	2,992 (29.6%)			0 (0.0%)	
Fail (Due to Timing, Within Tolerance)	2,208 (21.2%)	2,065 (20.4%)	1,010 (10.0%)	143 (1.4%)	1,198 (11.9%)
Fail (Due to Timing, Not in Tolerance)	571 (5.7%)	0 (0.0%)		571 (5.7%)	
Fail (Due to Fault)	4,329 (42.9%)	297 (2.9%)	258 (2.6%)	4,032 (39.9%)	4,071 (40.3%)

TABLE 7: Results when accepting the default verdict, steering, and filtering for Pacing.

	Infusion_Mgr			Pacing		
Technique	Precision	Recall	Accuracy	Precision	Recall	Accuracy
No Adjustment	0.64	**1.00**	0.78	0.69	**1.00**	0.82
Filtering	0.89	0.50	0.64	0.79	0.95	0.86
Steering (Both Metrics)	**0.94**	0.98	**0.96**	**0.97**	0.94	**0.95**

TABLE 8: Precision, recall, and accuracy results when accepting the initial oracle verdict, steering, and filtering. Best values are marked in bold for each system.

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, or 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 6.5. 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 6.5). 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).

6.3 Masking of Faults

As steering changes the behavior of the oracle and can result in a new test verdict, a risk 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.

As can be seen in Table 6, 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 8, results in a drop in recall from 1.00 (for accepting the initial verdict) to 0.98. For the Pacing model, as shown in Table 7, 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 8).

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

Initial Verdict	Pass (Post-Filtering)	Fail (Post-Filtering)
Pass	11,311 (74.4%)	0 (0.0%)
Fail (Due to Timing, Within Tolerance)	312 (2.1%)	1,123 (7.4%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	268 (1.7%)
Fail (Due to Fault)	598 (3.9%)	1,688 (11.1%)

TABLE 9: Results for step-wise filtering, (outputs + volume infused oracle), for Infusion_Mgr. Raw number, followed by percent of total.

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

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 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.

6.4 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 [24].

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 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 8, 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, even the initial success of filtering 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. However, the addition of this internal variable dramatically changes the results of filtering. The new results can be seen in Tables 9 and 10.

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 7. 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.

These results are not intended to indicate that indirect actions like filtering are *never* useful. When state does not have a major, persistent impact, or when the sources of non-determinism are limited, then a filter may be appropriate—and will be easier to implement and use. However, in the system domains we are interested in, state is important, and non-determinism is common. Therefore, direct actions, such as steering, are better able to handle the complex divergences between model and SUT.

6.5 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

Initial Verdict	Pass (Post-Steering)				Fail (Post-Steering)			
	Strict	Medium	Minimal	No Input Constraints	Strict	Medium	Minimal	No Input Constraints
Pass	1,270 (74.9%)				0 (0.0%)			
Fail (Due to Timing, Within Tolerance)	161 (9.4%)	156 (9.2%)	176 (10.4%)	180 (10.6%)	19 (1.1%)	24 (1.4%)	4 (0.2%)	0 (0.0%)
Fail (Due to Timing, Not in Tolerance)	0 (0.0%)	30 (1.8%)	35 (2.1%)		35 (2.0%)	5 (0.3%)	0 (0.0%)	
Fail (Due to Fault)	0 (0.0%)		1 (0.1%)	92 (5.4%)	210 (12.4%)		209 (12.3%)	118 (7.0%)

TABLE 11: Results for steering with various sets of constraints for the Infusion_Mgr system.

Initial Verdict	Pass (Post-Steering)			Fail (Post-Steering)		
	Strict	Medium/Minimal	No Input Constraints	Strict	Medium/Minimal	No Input Constraints
Pass	308 (28.0%)			0 (0.0%)		
Fail (Due to Timing, Within Tolerance)	284 (25.8%)		34 (3.1%)	22 (2.0%)		272 (24.7%)
Fail (Due to Timing, Not in Tolerance)	3 (0.2%)	70 (6.3%)	6 (0.5%)	81 (7.3%)	14 (1.2%)	78 (7.1%)
Fail (Due to Fault)	30 (2.7%)	40 (3.6%)	46 (4.1%)	372 (33.8%)	362 (32.9%)	356 (32.4%)

TABLE 12: Results for steering with various constraint sets for the Pacing system.

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

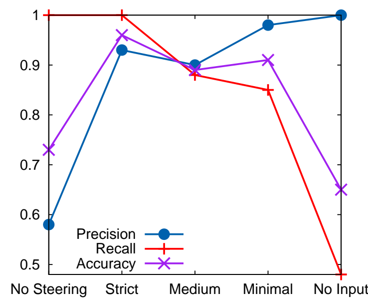


TABLE 13: Infusion_Mgr precision, recall, and accuracy for different tolerance constraint levels—as well as filtering and no adjustment.

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

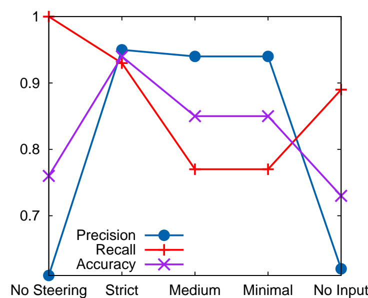


TABLE 14: Pacing precision, recall, and accuracy values for different tolerance levels—as well as filtering and no adjustment.

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 separate 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 13. Exact values for accepting the initial verdict, filtering, and strict constraints differ slightly from those in Table 8, due to the lower number of mutants used, but the trends remain the same. As detailed in Table 11, steering with **strict** constraints improves precision by quite a bit by correcting almost all of the acceptable behavioral deviations, while masking no faults. Filtering, as in Section 6.4, 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 11—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 11 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 Table 11, 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 14. Again, the results for no adjustment, filtering, and **strict** steering follow the same trends as the earlier experiment (see Table 8). 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 Table 12—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 15.

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.

Variable Name	Explanation	When Behavior is Impacted
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 15: Inputs for the Pacing system, explanations of their utility, and when adjustments to those variables will impact observable system behavior.

6.6 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 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 16, 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 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 6.4, 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.

```
((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)
```

Fig. 9: Sample constraints learned for Pacing.

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.

At first, the results for the learned constraints for Pacing—shown in rows 3-10 of Table 16—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

	Infusion_Mgr			Pacing		
Technique	**Precision**	**Recall**	**Accuracy**	**Precision**	**Recall**	**Accuracy**
No Adjustment	0.18	1.00	0.30	0.24	1.00	0.39
Filtering	0.70	1.00	0.82	0.32	1.00	0.48
Learned from Strict (M)	0.70	1.00	0.82	0.26	1.00	0.42
Learned from Strict (SE)	0.65	1.00	0.77	0.26	1.00	0.42
Learned from Medium (M)	0.70	1.00	0.82	0.26	1.00	0.42
Learned from Medium (SE)	0.70	1.00	0.82	0.26	1.00	0.42
Learned from Minimal (M)	0.73	1.00	0.84	0.26	1.00	0.42
Learned from Minimal (SE)	0.76	1.00	0.86	0.26	1.00	0.42
Learned from No Tolerances (M)	0.82	1.00	0.89	0.24	1.00	0.39
Learned from No Tolerances (SE)	0.76	1.00	0.86	0.26	1.00	0.40
Learned, All Tolerances, After Widening (M/SE)				0.75	0.88	0.81

TABLE 16: Median precision, recall, and accuracy values for learned tolerance constraints for each system. M=Manhattan, SE=Squared Euclidean dissimilarity function.

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 17: Distribution of results for steering with tolerance constraints learned from strict constraints for the Pacing system. Raw number of test results.

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 executions, listed in Table 17. 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 9), 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 be done automatically by systematically shrinking or widening the learned tolerances until this threshold is met. In this case, we took the constraints and increased the bounds by one on both ends. For example, the IN_EVENT_TIME tolerance listed in Figure 9 transforms from 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 the last row of Table 16. 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.

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.

6.7 Summary of Results

The precision, recall, and F-measure for each method—accepting the initial verdict, steering (using two different dissimilarity metrics), and filtering—are shown in Table 8.

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.

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.

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 domains 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 be used to steer. We do not believe that the modeling language chosen has 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 for similarly-sized systems [45], [42].

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. It is possible that using real faults would lead to different results. Nevertheless, 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 [43].

8 Related Work

Model-based testing (MBT) is a formal method that uses models of software systems for the derivation of test suites [1]. Such techniques commonly take a model in the form of a labeled transition system (for example, a finite state machine) [46], [4] and generate a series of tests to apply to the SUT. An attempt is made to establish conformance between the model and the system [47]. Much of the research on model-based testing is concerned explicitly with the generation of test input, although some have explored model-based oracle generation [48]. Such models serve implicitly as oracle on the generated tests, being the basis on which correctness is judged.

Several authors have examined the use of behavioral models as test-generation targets for real-time systems [49], [22], [3], [50], [51]. These models are designed to handle limited forms of non-deterministic behavior—allowing some flexibility in terms of the time that output events occur [22], [49], [50], [51]. Arcuri et al. also model the impact of non-deterministic hardware failures [49]. Many of these approaches make use of non-deterministic or special time modeling formalism, such as UPPAAL [52].

Oracle steering is conceptually similar to *dynamic* program steering, the automatic guidance of program execution [25], [8]. 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 [25]. Kannan et al. have proposed a framework to assure the correctness of software execution at runtime through corrective steering actions [53]. Although their framework 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-indicative behaviors. The Spec Explorer [9] test generation framework explores the possible runs of the executable model by applying steering actions in order to guide the model through various execution scenarios. It can then use the model as an oracle for the generated test by checking whether the SUT produces the same behaviors. 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 deterministic. Steering is not applied when checking conformance. As with the other approaches to model-based testing of real-time systems discussed in this section, Spec Explorer may be able to address some of the issues we are concerned with, but it also suffers from the same limitations of being locked into a particular model format and requiring that non-determinism be built into that model.

9 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.

There is still much room for future work:

- We plan to further examine the impact of different dissimilarity metrics and tolerance constraints on oracle verdict accuracy;
- Policies defining when to attempt to steer could deeply impact the resulting testing process—we plan to define additional steering policies and explore their use;
- The need to invoke constraint solvers multiple times throughout execution limits the performance of the steering framework. We seek improvements to speed and scalability, and plan to experiment with the use of metaheuristic optimization algorithms in place of multiple calls to an exhaustive solver;
- 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;
- And, we plan to examine the use of steering to debug faulty or incomplete oracle models.

10 Source Code and Data

In the interest of allowing others to extend, reproduce, or otherwise make use of the work that we have conducted, the research prototype of our steering framework and experimental data—including the models, mutants, and tests—have been made freely available under the Mozilla Public License 2.0.

1) Experimental data for each system is available from the PROMISE repository [54]. This includes the original Stateflow models, the Lustre translation, fault-seeded mutants, and randomly-generated tests used in our experiments.
   a) The Infusion_Mgr data can be found at http://openscience.us/repo/test-generation/manager.html
   b) The Pacing data can be found at http://openscience.us/repo/test-generation/pacing.html.
2) The source code of the steering framework—and binaries of the required dependencies—can be obtained from https://github.com/Greg4cr/Steering-Framework.

References

[1] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey on automated software test case generation," *Journal of Systems and Software*, vol. 86, pp. 1978–2001, August 2013.

[2] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, pp. 507–525, May 2015.

[3] A. En-Nouaary, R. Dssouli, and F. Khendek, "Timed wp-method: testing real-time systems," *IEEE Transactions on Software Engineering*, vol. 28, no. 11, pp. 1023–1038, Nov.

[4] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines-a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.

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

[6] 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*, vol. 16, pp. 403–414, April 1990.

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

[8] 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.*, pp. 8–8, 2001.

[9] 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 *Formal Methods and Testing* (R. M. Hierons, J. P. Bowen, and M. Harman, eds.), vol. 4949 of *Lecture Notes in Computer Science*, pp. 39–76, Springer, 2008.

[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), ACM, 2014.

[11] G. Gay, S. Rayadurgam, and M. P. Heimdahl, "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, (New York, NY, USA), pp. 527–538, ACM, 2014.

[12] W. Howden, "Theoretical and empirical studies of program testing," *IEEE Transactions on Software Engineering*, vol. 4, no. 4, pp. 293–298, 1978.

[13] E. Weyuker, "The oracle assumption of program testing," in *13th International Conference on System Sciences*, pp. 44–49, 1980.

[14] 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*, pp. 870–880, IEEE Press, 2012.

[15] 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, (Washington, DC, USA), pp. 305–314, IEEE Computer Society, 2005.

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

[17] B. Scientific, "Pacemaker system specification," in *Pacemaker Formal Methods Challenge*, Software Quality Research Lab, 2007.

[18] T. M. Association, "Modelica - a unified object-oriented language for systems modeling," tech. rep., 2012.

[19] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for gui-based software applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, Feb. 2007.

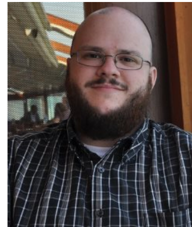
[20] A. Gomes, A. Mota, A. Sampaio, F. Ferri, and E. Watanabe, "Constructive model-based analysis for safety assessment," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 6, pp. 673–702, 2012.

[21] S. P. Miller, A. C. Tribble, M. W. Whalen, and M. P. E. Heimdahl, "Proving the shalls: Early validation of requirements through formal methods," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 4, pp. 303–319, 2006.

[22] K. G. Larsen, M. Mikucionis, and B. Nielsen, "Online testing of real-time systems using UPPAAL," in *International workshop on formal approaches to testing of software (FATES 04)*, Springer, 2004.

[23] J. Zander, I. Schieferdecker, and P. J. Mosterman, *Model-based testing for embedded systems*. CRC press, 2011.

[24] S. Mahajan and W. G. Halfond, "Finding HTML presentation failures using image comparison techniques," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, (New York, NY, USA), pp. 91–96, ACM, 2014.
[25] W. Gu, J. Vetter, and K. Schwan, "An annotated bibliography of interactive program steering," *ACM SIGPLAN Notices*, vol. 29, 1994.
[26] S.-H. Cha, "Comprehensive survey on distance/similarity measures between probability density functions," *International Journal of Mathematical Models and Methods in Applied Sciences*, vol. 1, no. 4, pp. 300–307, 2007.
[27] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, pp. 31–88, Mar. 2001.
[28] G. Hagen, *Verifying safety properties of Lustre programs: an SMT-based approach*. PhD thesis, University of Iowa, December 2008.
[29] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems, Second Edition*. Cambridge Press, 2006.
[30] A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185, Frontiers in Artificial Intelligence and Applications*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2009.
[31] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.
[32] G. Gay, *Automated Steering of Model-Based Test Oracles to Admit Real Program Behaviors*. PhD thesis, University of Minnesota, May 2015.
[33] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Houndations of Machine Learning*. MIT Press, 2012.
[34] T. Menzies and Y. Hu, "Data mining for very busy people," *Computer*, vol. 36, pp. 22–29, Nov. 2003.
[35] C. P. T. M. Johann Schumann, Karen Gundy-Burlet and A. 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] G. Gay, T. Menzies, M. Davies, and K. Gundy-Burlet, "Automatically finding the control variables for complex system behavior," *Automated Software Engineering*, vol. 17, pp. 439–468, Dec. 2010.
[39] S. D. Bay and M. 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, (New York, NY, USA), pp. 302–306, ACM, 1999.
[40] 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.
[41] N. Halbwachs, *Synchronous Programming of Reactive Systems*. Klower Academic Press, 1993.
[42] A. Rajan, M. Whalen, M. Staats, and M. Heimdahl, "Requirements coverage as an adequacy measure for conformance testing," 2008.
[43] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, pp. 608 –624, aug. 2006.
[44] N. Li and J. Offutt, "An empirical analysis of test oracle strategies for model-based testing," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 363–372, March 2014.
[45] A. Rajan, M. Whalen, and M. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 161–170, ACM, 2008.
[46] E. Brinksma and J. Tretmans, "Testing transition systems: An annotated bibliography," in *Modeling and Verification of Parallel Processes* (F. Cassez, C. Jard, B. Rozoy, and M. Ryan, eds.), vol. 2067 of *Lecture Notes in Computer Science*, pp. 187–195, Springer Berlin Heidelberg, 2001.
[47] J. Tretmans, "Model based testing with labelled transition systems," in *Formal methods and testing*, pp. 1–38, Springer, 2008.
[48] L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller, "Model-based test oracle generation for automated unit testing of agent systems," *IEEE Transactions on Software Engineering*, vol. 39, pp. 1230–1244, Sept 2013.
[49] 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 International Conference on Testing software and systems*, pp. 95–110, Springer-Verlag, 2010.
[50] 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*, pp. 136–146, 1997.
[51] L. Briones and E. Brinksma, "A test generation framework for quiescent real-time systems," in *International workshop on formal approaches to testing of software (FATES 04)*, pp. 64–78, Springer-Verlag GmbH, 2004.
[52] J. Bengtsson and W. Yi, "Timed automata: Semantics, algorithms and tools," in *Lectures on Concurrency and Petri Nets* (J. Desel, W. Reisig, and G. Rozenberg, eds.), vol. 3098 of *Lecture Notes in Computer Science*, pp. 87–124, Springer Berlin Heidelberg, 2004.
[53] S. Kannan, M. Kim, I. Lee, O. Sokolsky, and M. Viswanathan, "Runtime monitoring and steering based on formal specifications," in *Workshop on Modeling Software System Structures in a Fastly Moving Scenario*, 2000.
[54] T. Menzies, R. Krishna, and D. Pryor, "The promise repository of empirical software engineering data," 2015.



Gregory Gay is an Assistant Professor of Computer Science & Engineering at the University of South Carolina. His research interests include automated testing and analysis—with an emphasis on test oracle construction—and search-based software engineering. Greg received his Ph.D. from the University of Minnesota, working with the Critical Systems research group, and an M.S. from West Virginia University.

Sanjai Rayadurgam is a Research Staff Member at the University of Minnesota Software Engineering Center in the Department of Computer Science and Engineering. His research interests are in software testing, formal analysis and requirements modeling, with particular focus safety-critical systems development, where he has significant industrial experience. He earned a B.Sc. in Mathematics from the University of Madras at Chennai, and in Computer Science & Engineering, an M.E. from the Indian Institute of Science at Bangalore and a Ph.D. from the University of Minnesota at Twin Cities.

Mats P.E. Heimdahl is a Professor and Department Head in Computer Science and Engineering at the University of Minnesota. He earned an M.S. in Computer Science and Engineering from the Royal Institute of Technology (KTH) in Stockholm, Sweden and a Ph.D. in Information and Computer Science from the University of California at Irvine.

His research interests are in software engineering, safety critical systems, software safety, testing, requirements engineering, formal specification languages, and automated analysis of specifications.

He is the recipient of the NSF CAREER award, a McKnight Land-Grant Professorship, the McKnight Presidential Fellow award, and the awards for Outstanding Contributions to Post-Baccalaureate, Graduate, and Professional Education at the University of Minnesota