# Steering Model-Based Oracles to Admit Real Program Behaviors*

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

## ABSTRACT

The *oracle*—an arbiter of correctness of the system under test (SUT)—is a major component of the testing process. Specifying oracles is challenging for real-time embedded systems, where small changes in time or sensor inputs may cause large differences in behavior. Behavioral models of such systems, often built for analysis and simulation, are appealing for reuse as oracles. However, these models typically provide an *idealized* view of the system. Even when given the same inputs, the model's behavior can frequently be at variance with an acceptable behavior of the SUT executing on a real platform. We therefore propose *steering* the model when used as an oracle, to admit an expanded set of behaviors when judging the SUT's adherence to its requirements. On detecting a behavioral difference, the model is backtracked and then searched for a new state that satisfies certain constraints and minimizes a dissimilarity metric. The goal is to allow non-deterministic, but bounded, behavior differences while preventing future mismatches, by guiding the oracle—within limits—to match the execution of the SUT. Early results show that steering significantly increases SUT-oracle conformance with minimal masking of real faults and, thus, has significant potential for reducing development costs.

## Categories and Subject Descriptors

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

## General Terms

Verification

## Keywords

Software Testing, Test Oracles, Model-Based Testing

## 1. INTRODUCTION

The *oracle* is a judge that determines the correctness of the execution of a given system under test (SUT) against a test suite. Despite increased attention in recent years, the *test oracle problem* [5]—constructing efficient and robust oracles—remains a major challenge for many domains. Real-time process control systems—embedded systems that interact with physical processes such as pacemakers or power management systems—are particularly difficult to build oracles for, as their behavior depends not only on the values of inputs and outputs, but also on their time of occurrence [2], and minor behavioral distinctions may have significant consequences [6]. When executing on an embedded hardware platform, several sources of non-determinism, such as input processing delays, execution time fluctuation, and hardware inaccuracy, can make the SUT exhibit varying but acceptable behaviors.

Behavioral models [8], typically expressed as state-transition systems, represent requirements by prescribing the behavior (the system state) to be exhibited in response to each input. Such models are used for many purposes in industrial software development and so their reuse as test oracle is highly desirable. However, these models provide an abstract view of the system that typically simplifies the actual conditions in the execution environment. On a real hardware platform, the SUT may exhibit behavior that differs from what the model prescribes for a given input. Over time, these differences can build to the point where the execution paths of the model and the SUT diverge enough to flag "failure", even if the system is operating within the system requirements.

We take inspiration for addressing this problem from *program steering*, the process of adjusting the execution of live programs in order to improve performance, stability, or correctness [9]. We hypothesize that behavioral models can be adapted for use as oracles for real-time systems through the use of *steering actions* [13]. By comparing the state of the model-based oracle (MBO) with that of the SUT following an output event, we can guide the model to match the state of the SUT, as long as a set of constraints are met. The result would be a widening of the behaviors accepted by the MBO, thus compensating for allowable non-determinism, without impairing the ability of the MBO to judge the SUT.

We propose an automated framework for comparing and steering the MBO with respect to the SUT, discuss the current prototype implementation of the framework, and assess its capabilities on a model for the control module of an infusion pump—a real-world system with complex, time-based behaviors. Results indicate that steering successfully accounts for all allowable differences between the MBO and the SUT, eliminating a large number of spurious "failure" verdicts. However, steering also masks a small number of faults, indicating a need to future work.

To the best of our knowledge, this is the first work proposing the automated steering of the test oracle. While this research is still in its infancy, the initial results are promising. If successful, this approach would lower testing costs and reduce development

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


*ICSE* '14, May 31—June 7, 2014, Hyderabad, India

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

**Figure 1: Steps in the Steering Process**

effort for real-time control systems by enabling reuse of behavioral models as oracles.

## 2. PROBLEM DEFINITION

A *test oracle* is defined as a predicate on a sequence of stimuli to and reactions from the SUT that judges the resulting behavior according to some specification of correctness [5]. Although oracles can be derived from many sources of information, we are particularly interested in using behavioral models, such as those often built for purposes of simulation, analysis and testing [8].

Executable behavioral models can be divided into *declarative* models created in formal specification languages, and *constructive* models built with state-transition languages such as Simulink & Stateflow, Statemate, finite state machines, or other automata structures [8]. Presently, we focus on constructive state-transition systems, as these are frequently used to model real-time control systems. Real-time control systems monitor and interact with a physical environment. Examples of such systems include pacemakers, traffic-control systems, and power plant management software.

Non-determinism is a major concern for systems that interact with physical processes. The task of monitoring the environment and pushing signals through multiple layers of sensors and actuators can introduce additional points of failure, delay, and unpredictability. Input and observed output values may be skewed by hardware noise, 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 such behavior is not what was captured in the model—which, by its very nature, incorporates a simplified view of the problem domain. A common abstraction when modeling is to elide any details that distract from the core system behavior in order to ensure that model-based analyses are feasible and useful. Yet these details manifest as differences between the model and the implemented system.

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. However, these solutions have their limitations. Assertion-based approaches only ensure that a limited set of properties hold at *particular points in the program*. Further, such oracles may not be able to account for the same range of testing scenarios as a model that prescribes behavior for all inputs. Declarative models that express the computation as a theory in a formal logic allow for more sophisticated forms of verification and can better account for time-constrained behaviors [3]. However, Miller et al. have found that developers are more comfortable building constructive models than formal declarative models [10]. Constructive models are visually appealing, easy to analyze without specialized knowledge, and suitable for analyzing failure conditions and events in an isolated manner [3]. The complexity of declarative models and the knowledge needed to design and interpret such models make widespread industrial adoption of the paradigm unlikely.

While there are challenges in using constructive model-based oracles, it is a widely held view that such models are indispensable in other areas of development and testing, such as automatic test generation [7]. From this standpoint, the motivational case for models as oracles is clear—if these models are already being built, their reuse as test oracles could save a large amount of time and money. Practitioners are well-versed in these models and so these are likely to be less error-prone compared to building a new unfamiliar type of oracle. Therefore, we seek a way to use constructive model-based oracles that can handle the non-determinism introduced during system execution on the target hardware.

## 3. ORACLE STEERING

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

An approach to address this would be to augment the comparison procedure with a filtering mechanism to detect and discard acceptable differences on a per-step basis. The issue with this is that the effect of non-determinism may linger on for several steps, leading to irreconcilable differences between the SUT and MBO. Filters may not be effective at handling growing behavioral divergence.

We take inspiration from *program steering*—however, instead of steering the SUT, 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, delay or withhold an input—that changes the state of the MBO to a state more similar to the SUT (as judged by a dissimilarity metric). Oracle steering, unlike filters, is *adaptable*. Such actions provide flexibility to handle non-determinism, while still retaining the power of the oracle as an arbiter. Improper steering can bias the behavior of the MBO, masking both acceptable deviations and actual indications of failures. However, we believe that it is possible to sufficiently bound steering such that the ability to detect faults is still retained, using a series of constraints:

**1:** A set of **tolerance constraints** governing the allowable changes to certain variables (input, internal, or output) in the model that can be effected by steering. These rules define the level of non-determinism or behavioral deviation that can be accounted for with steering. "No change allowed to the system's operational mode" or "computed output may be produced between $x$ and $y$ seconds" are examples of such tolerance constraints.

**2:** A **dissimilarity function** $Sym(model\ state, SUT\ state)$, that compares the state of the model to the observable state of the SUT. We seek a minimization of $Sym(s_m^{new}, s_{sut}) < Sym(s_m, s_{sut})$.

**3:** A further set of general **policy decisions** on when to steer. For example, one might decide not to steer unless $Sym(s_m^{new}, s_{sut}) = 0$ — that is, unless there exists a steering action that results in a model state identical to that observed in the SUT.

In other words, the new state of the MBO following the application of a steering action must be one that is possible to reach within

a limited number of transitions from the current state of the model, must fall within the boundaries set by the tolerance constraints, and must minimize the dissimilarity function.

We have implemented the basic steering approach outlined in Figure 1. It is a search process based on SMT-based bounded model checking [4], which is a natural choice for this problem. We have a series of constraints that govern steering actions and seek to locate a model state reachable in a limited number of transitions that satisfies those constraints and minimizes a dissimilarity metric. Specifically, we make use of the Kind model checker [4].

Note that the constraint $Sym(s_m^{new}) < Sym(s_m)$ would give us a model state that is more similar to the behavior of the SUT than the original transition taken by the MBO, but carries no guarantee that the satisfying state minimizes the dissimilarity metric. Thus, we apply the model checker with this constraint in order to get an initial threshold, then iteratively reapply the model checker with new thresholds until we can no longer find a better solution. The best solution found then becomes the new model to that state.

# 4. RELATED WORK

Several authors have addressed model-based conformance testing for real-time systems [1, 7, 2]. For example, Larsen et al. [7] model a system as a non-deterministic timed automata that is constrained by an environment model. The combined model serves as an oracle during test execution.

While several approaches consider limited non-determinism, there are a few key differences with our proposed approach, which decouples the model from the rules governing steering. This makes non-determinism implicit and the approach more generally applicable. Explicitly specified non-deterministic behavior would limit the scope of non-determinism handled by the oracle to what has been 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. Also, by not relying on a specific model format, steering can be made to work with models created for a variety of purposes.

Oracle steering is conceptually similar to *dynamic* program steering, the automatic guidance of program execution [9]. The most relevant in terms of the techniques employed is Spec Explorer [13], where an executable behavioral model is steering through different execution scenarios for test generation as opposed to adjudging system execution.

# 5. PILOT STUDY

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

1. To what degree does steering lessen behavior differences that are legal under system requirements?
2. To what degree does steering mask behavior differences that fail to conform to the requirements?

**Models and Test Generation:** We have based our MBO on the management subsystem of a generic Patient-Controlled Analgesia (GPCA) infusion pump [11]. This model, developed in the Simulink and Stateflow notations and translated into the Lustre synchronous programming language, is a complex real-time system of the type common in the medical domain. This subsystem controls the operational mode of the infusion pump, including the flow rate of the medicine administered to the patient.

**Table 1: Verdicts: T(true)/F(false), P(positive)/N(negative).**

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

In order to produce "implementations" of this system, we created two clones of the model, each introducing a realistic non-deterministic timing change to the system: (1) a version of the system where the exit of the patient-requested dosage period may be delayed by a short length of time, and (2), a version of the system where the exit of an intermittent dosage period may be delayed. We have also created 50 *mutants* (faulty implementations) of the original system and the two implementation models by introducing a single fault into each model[1].

Using a random testing algorithm, we generated 100 tests—each 30 steps long—and ran them against each model in order to collect traces. In the models with timing fluctuations, we controlled those fluctuations through the use of an additional input variable. The value for that variable was generated non-deterministically, but we used the same value across all systems with the same -timing fluctuation. As a result, we know whether a behavior is due to a timing fluctuation or a seeded fault in the system.

**Steering Constraints:** For this system, we specified tolerance constraints in terms of the input variables of the system (although constraints can be placed on internal or output variables as well):

- Five of the input variables relate to timers within the system. For each of those, we placed an allowance of $(Current\ Value - 1) <= New\ Value <= (Current\ Value + 2)$. For example, a dosage duration is allowed to fall within a four second period—between one second shorter and two seconds longer than the prescribed duration.
- The remaining 15 input variables are not allowed to be steered.

These constraints reflect what we consider a realistic application of steering—we allow a small window around the behaviors that we accept that are related to timing, but as we do not expect any sensor noise, we do not allow freedom in adjusting sensor-based inputs.

The dissimilarity metric used in this experiment is the $L^1 - Norm$. Given vectors representing the state of the SUT and the MBO—where each member of the vector represents the value of a variable—the dissimilarity between the two vectors can be measured as the absolute numerical distance between the state of the SUT and the MBO. A constant difference of 1 is used for differences between boolean variables.

**Experiment Overview:** Using the generated artifacts—without steering—we monitored the outputs during each test, compared the results to the values of the same variables in the MBO to calculate the similarity score, and issued an initial verdict. Then, if the verdict was a failure (score > 0), we steered the MBO, and recorded a new verdict post-steering.

We can assess the impact of steering using the verdicts made before and after steering by calculating the number of *true positives*—steps where steering does not mask incorrect behavior—the number of *false positives*—the number of steps where steering fails to account for an acceptable behavioral difference—and the number of *false negatives*—the number of steps where steering does masks an incorrect behavior.

[1]The mutation operators used are discussed at length in [12].

**Table 2: Distribution of results.**

| Initial Verdict | Pass (Post-Steering) | Fail (Post-Steering) |
|---|---|---|
| Pass | 17835 (70.2%) | 0 (0.0%) |
| Fail (Due to Timing, Within Tolerance) | 2443 (9.6%) | 0 (0.0%) |
| Fail (Due to Timing, Not in Tolerance) | 266 (1.0%) | 536 (2.1%) |
| Fail (Due to Fault) | 399 (1.5%) | 3921 (15.4%) |

**Table 3: Precision, recall, and F-measure values.**

| Technique | Precision | Recall | F-measure |
|---|---|---|---|
| No Steering | 0.66 | 1.00 | 0.80 |
| With Steering | 1.00 | 0.87 | 0.93 |

The testing outcomes in terms of true/false positives/negatives are listed in Table 1. 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. We also calculate the *F-measure*—the harmonic mean of precision and recall—in order to judge the accuracy of oracle verdicts.

**Results:** Table 2 shows the results of steering—a pass or fail—following each of the pre-steering testing outcomes—a pass, a fail due to allowable timing fluctuations, a fail due to unacceptable timing fluctuations, or a fail due to a fault in the system. The precision, recall, and F-measure values for steering and the default testing scenario (issuing a verdict without steering) are listed in Table 3.

For the case example studied, steering is able to account for all situations where non-deterministic timing affects conformance while both the MBO and the implementation remain within the bounds set in the system specification. Therefore, we see a sharp increase in precision over the default situation where steering is not applied. Previously, a developer would have to manually inspect nearly 30% of the tests for faults in the system (all pre-steering failures in Table 2). Now, they would be asked to focus on 17.5% of the test results (all post-steering failures).

Note, however, that 665 tests that should have still been labeled as failures post-steering are now labeled as passing. This is a relatively small number—only 2.5% of the test results—but any loss in recall is cause for concern when working with safety-critical systems. Still, as can be seen from the F-measure, the application of steering results in greater *accuracy* in test classifications than not steering. These results seem promising, and further examination of constraints and dissimilarity metrics should improve recall.

As steering does carry the risk of masking faults, we recommend that it be applied as a *focusing tool*—to point the developer toward definite faults so that they do not spend as much time investigating potential faults. The increase in oracle verdict accuracy and the large decrease in failure verdicts—from 7565 to 4457 tests—after steering should result in a substantial decrease in the amount of time that developers spend investigating failure verdicts are actually instances of allowable non-conformance.

# 6. CONCLUSION AND FUTURE WORK

The results of the initial pilot study are quite promising. If steering can be scaled to larger systems without loss in accuracy, then the potential for improvements in the time and effort spent on testing real-time embedded systems are significant. The use of steering can allow developers to focus on addressing definite faults, rather than spending time examining test failure verdicts that can be blamed on a rigid oracle model.

However, there is much room for future work. We plan to expand the case examples in terms of both program size and scope of non-determinism. The dissimilarity metric used in this study was deliberately simplistic. In future work, we will study more sophisticated metrics, including ones that admit equivalent behaviors. We plan to examine the impact of different sets of tolerance constraints and overall steering policy decisions on the accuracy of steering. We would also like to examine the cost and time savings that could result from the application of steering.

# 7. REFERENCES


[1] 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*, pages 95–110. Springer-Verlag, 2010.

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

[3] 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*, 14(6):673–702, 2012.

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

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

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

[7] 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.

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

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

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

[11] 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.

[12] A. Rajan, M. Whalen, M. Staats, and M. Heimdahl. Requirements coverage as an adequacy measure for conformance testing. In *Proc. of the 10th Int'l Conf. on Formal Methods and Software Engineering*, pages 86–104. Springer, 2008.

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