2015 30th IEEE/ACM International Conference on Automated Software Engineering

Executing Model-based Tests on Platform-specific Implementations

Dongjiang You[1], Sanjai Rayadurgam[1], Mats P.E. Heimdahl[1], John Komp[2], BaekGyu Kim[3], Oleg Sokolsky[3]
[1]Department of Computer Science and Engineering, University of Minnesota, USA
[2]Medtronic PLC, USA
[3]Department of Computer and Information Science, University of Pennsylvania, USA
Email: [djyou, rsanjai, heimdahl]@cs.umn.edu, john.komp@medtronic.com, [baekgyu, sokolsky]@cis.upenn.edu

***Abstract*—Model-based testing of embedded real-time systems is challenging because platform-specific details are often abstracted away to make the models amenable to various analyses. Testing an implementation to expose non-conformance to such a model requires reconciling differences arising from these abstractions. Due to stateful behavior, naive comparisons of model and system behaviors often fail causing numerous false positives. Previously proposed approaches address this by being reactively permissive: passing criteria are relaxed to reduce false positives, but may increase false negatives, which is particularly bothersome for safety-critical systems. To address this concern, we propose an automated approach that is proactively adaptive: test stimuli and system responses are suitably modified taking into account platform-specific aspects so that the modified test when executed on the platform-specific implementation exercises the intended scenario captured in the original model-based test. We show that the new framework eliminates false negatives while keeping the number of false positives low for a variety of platform-specific configurations.**

I. Introduction

Advances in automated test generation from system models do not always translate to realizable benefits in terms of testing an implementation of the system. While it is now routinely possible to generate hundreds or even thousands of test cases from models, the ability to use those for testing a particular realization of the system is hampered by two main bottlenecks:

1) Translating the tests generated from the model – which, by definition, abstracts away some implementation specific details – into equivalent scenarios for the actual system.
2) Deriving an oracle – an arbiter of correctness – that can decide whether the actual system passed or failed such a test when it is executed.

At a conceptual level, these do not appear to be problematic. Translating tests derived from a model to a particular implementation is simply a matter of concretizing the abstract test scenario. And, the model itself is a good oracle for judging the correctness of the implementation: in fact, typically tests generated from executable models include not only the inputs used to trigger a particular model behavior, but also the corresponding outputs produced by the model, which can then be considered as the output expected from any implementation of the system for that test, modulo abstraction. Figure 1 shows this view: if f is an abstraction function that maps the system to the model, then, to address (1), for each abstract in_i^m in the generated test, pick some concrete in_i^s in $f^{-1}(in_i^m)$ and to address (2), check that the concrete output produced by the system maps to the abstract output produced by the model, i.e., check that $f(out_i^s) = out_i^m$.

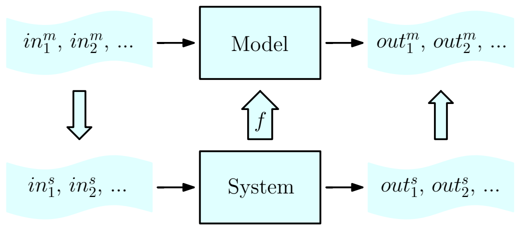

Fig. 1: Concretizing model-based tests: A simplified view

However, the situation is often more complex, especially for testing real-time control systems, which is our focus. In practice, real-time systems often exhibit non-deterministic behaviors such as run-to-run variations in timing. Executing model-based tests on such systems can lead to false positives – a system may behave correctly but still not match the model's behavior exactly as captured in the test. The effects of the hardware platform on which the system executes have to be taken into account. Figure 2 shows a better reflection of the typical situation.

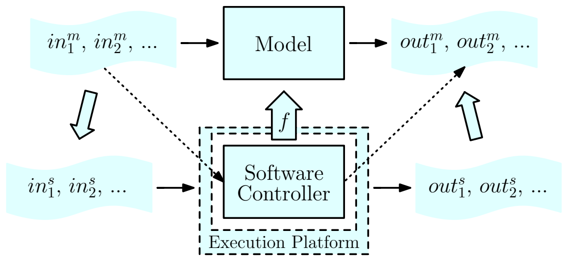

Fig. 2: Concretizing model-based tests: A typical scenario

The model we deal with here is that of an embedded software controller, which continually processes inputs, updates its state and produces outputs in discrete steps. Abstracted away typically are the notion of time (specifically, the exact relationship between the execution steps and real time) and the analog hardware interfaces in the sensors and actuators that interact with the real-world. The system under test (SUT) is an implementation (i.e., software controller) of the model. Ideally, for conformance testing the behavior of the model must be compared to that of the SUT (represented by dotted lines

This work has been partially supported by NSF grant CNS-1035715.

in Figure 2). In practice, the inputs and outputs have to be mediated through the hardware execution platform.

Tests generated from models provide the inputs and outputs of the model at each execution step. Replicating the test scenario on an actual system in a test environment that simulates the real world, requires accounting for these differences which usually involves more than a simple step-wise concretization of inputs. Timing differences may require that the actual test environment must stimulate the hardware interfaces within a certain real-time window. Further, the environmental inputs that would lead to a particular input required for the model-based test may depend on the real time instant at which the test environment provides the stimulus. In these circumstances, finding a concrete sequence of inputs to the target system to replicate the scenario produced by the inputs in the model-based test is non-trivial. Further, the output produced by the system, even when it is behaving correctly, could be different from the outputs produced by the model, because of timing and abstraction induced differences. This makes using the model as the oracle problematic.

The majority of existing model-based testing approaches are concerned with generating test cases and demonstrating that the SUT conforms to the model. Techniques specifically for testing real-time behaviors such as extending the model with non-deterministic real-time behaviors [1] using timed automata [2] or UPPAAL [3] can potentially characterize system non-determinism accurately. These approaches, however, do not scale well, since the introduced non-determinism can increase reachable state space exponentially, which also contradicts the original intent of using platform-independent models. Oracle steering [4] is an alternative approach that attempts to *slightly* change the model behaviors in order to accept non-deterministic real-time behaviors. In oracle steering, the system is deemed to have passed the test if some model behavior can be found that is *similar* to the observed system behavior, but there is no guarantee that the modified test inputs used to steer the model still retain the intent of the original test scenario. If, as is typical, the test was generated to achieve a certain purpose, we need a way to ensure that the purpose is indeed realized when each time the test is executed on the target platform.

In this work, we propose a complementary approach that translates a sequence of abstract model inputs to an *equivalent* sequence of concrete system inputs, which would considerably advance the utility and practicality of model-based testing. Equivalence here is to be construed broadly as a relation between finite sequences of test inputs (and outputs) that is sufficient to capture the notion of test scenario.

II. Background and Problem Statement

A. Model-based Testing

Model-based testing broadly refers to the use of models of software to perform software testing. In particular, models are frequently used to derive test suites and oracles. The model describes, in some abstract fashion, input/output sequences that are possible or acceptable. The SUT is considered a black box whose input/output sequences must conform to that described by the model [5].

Of particular interest to us are the use – in testing – of models of reactive systems which are typically specified as (extended) finite state machines [6] or labeled transition systems [7]. Such models provide an operational view of the system that enables execution/simulation of the model over a series of steps. Tests generated from such models capture the input provided to the model and the corresponding output produced by the model which becomes the oracle information against which the output produced by the SUT is compared. Notionally one may view a test execution as providing the *same* input to the SUT and the model and checking that the corresponding outputs match at each step. There is a rich body of research in the use of such behavioral models for test generation [5], [8].

These models by necessity abstract away implementation details which makes them amenable to various automated analyses. We call these *platform independent models*. However, since these models also provide an operational view, executable representations are often derived from these models using automated translation as well as manual coding. Executable implementations for the target hardware environment can often be derived from these models. Such implementations are called *platform-specific implementations*. These typically have additional components (e.g., input and output devices) and details that are not represented in platform-independent models. In particular, there are typically timing delays associated with input and output devices, code execution, and communications between components. Furthermore, these timing aspects are non-deterministic (e.g., a sensor's sampling routine may take a non-deterministic amount of time to process data). However, the platform-independent model is specified as a discrete-time transition system where time progresses in discrete steps between computations. Reliably reproducing a test scenario on the platform-specific implementation that is equivalent to a given test scenario for the platform-independent model requires reconciling differences induced by the timing abstraction and non-determinism, which we address in the present work.

B. The Four-Variable Model

In order to precisely characterize timing at different system boundaries, our approach uses and extends Parnas' four-variable model. Figure 3 shows the four-variable model defined by Parnas et al. [9].

Monitored variables are used to express physical environment changes that can be *observed* by the execution platform. The execution platform typically uses input devices (i.e., sensors) to observe the status of monitored variables. *Controlled variables* are used to express physical environment changes that can be *enforced* by the execution platform. The execution platform typically uses controlled variables to characterize changes and uses output devices (i.e., actuators) to enforce them. *Input variables* and *output variables* are used to express inputs and outputs of the software controller or the platform-independent model, which are represented by *SOF*.

For a given variable v, we use v^t to represent the time-function of its value [9]. Given definitions of the four variables, the following relations can be defined.

IN Relation: $(\underset{\sim}{m}^t, \underset{\sim}{i}^t) \in IN$, where $\underset{\sim}{m}^t$ and $\underset{\sim}{i}^t$ represent the vectors of monitored and input variables, respectively,

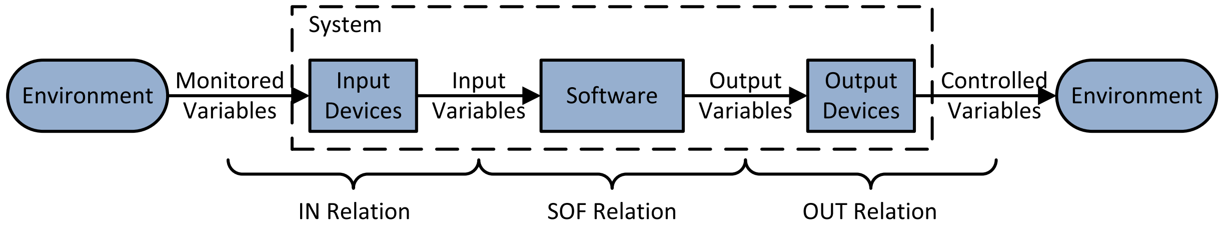

Fig. 3: The four-variable model

represents the physical interpretation of the input devices.

SOF Relation: $(\underset{\sim}{i}^t, \underset{\sim}{o}^t) \in SOF$, where $\underset{\sim}{i}^t$ and $\underset{\sim}{o}^t$ represent the vectors of input and output variables, respectively, represents the software system with input-output behavior.

OUT Relation: $(\underset{\sim}{o}^t, \underset{\sim}{c}^t) \in OUT$, where $\underset{\sim}{o}^t$ and $\underset{\sim}{c}^t$ represent the vectors of output and controlled variables, respectively, represents the effects of the output devices.

C. Problem Statement

We can now describe the problem as follows. Given a test case $(\underset{\sim}{i}^t, \underset{\sim}{o}^t)$ for SOF, we want to find a test case $(\underset{\sim}{m}^t, \underset{\sim}{c}^t)$ for $IN \cdot SOF \cdot OUT$, such that $(\underset{\sim}{i}^t, \underset{\sim}{o}^t) \in IN(\underset{\sim}{m}^t) \times OUT^{-1}(\underset{\sim}{c}^t)$. Further, executing the system-level test $(\underset{\sim}{m}^t, \underset{\sim}{c}^t)$ on $IN \cdot SOF \cdot OUT$ should exercise SOF in a way that is "equivalent" to executing the test $(\underset{\sim}{i}^t, \underset{\sim}{o}^t)$ on SOF. We will leave the notion of equivalence to be informally understood as the "intended scenario" in the test $(\underset{\sim}{i}^t, \underset{\sim}{o}^t)$.

In words, we seek a system test case that is equivalent to a given software test case. However, in practice, we do not need an equivalent system test case but rather a method to test the system in an equivalent way. This can be achieved by first finding equivalent system inputs, then executing the system with those inputs and finally verifying that the output produced by the system is equivalent to the output expected of the software. The problem can then be formulated as:

1) Given input variable vector $\underset{\sim}{i}^t$ and the relation IN, find monitored variable vector $\underset{\sim}{m}^t$, such that $(\underset{\sim}{m}^t, \underset{\sim}{i}^t) \in IN$.
2) Similarly, given controlled variable vector $\underset{\sim}{c}^t$ and the relation OUT, find output variable vector $\underset{\sim}{o}^t$, such that $(\underset{\sim}{o}^t, \underset{\sim}{c}^t) \in OUT$.

The first goal would enable execution of model-based tests on platform-specific implementations and the second goal would enable the use of the model as the oracle when executing those tests on platform-specific implementations.

As used in the literature of testing real-time systems, we define *false positive* and *false negative* as the following:

1) **False Positive**: If a test fails on a system that is acting correctly, we call it a false positive.
2) **False Negative**: If a test passes on a system that is acting erroneously, we call it a false negative.

D. Motivating Example

We use a PCA (Patient-Controlled Analgesia) infusion pump system as an example to illustrate the problem and our approach throughout this paper. A PCA infusion pump system is a safety-critical medical device that physically interacts with a patient by injecting medication for the purpose of pain-relief. The infusion is controlled using several sensors and actuators. The patient can control the device via a user interface and a pump motor is used to apply force so that the medication can flow from the syringe to the patient through intravenous tubes. Various sensors are used to detect abnormal conditions such as empty reservoirs and air in line, when happened, the patient is notified by actuators such as buzzers and LED lights.

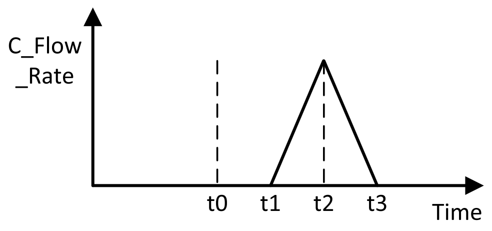

Fig. 4: Flow rate change due to starting infusion and detecting air in line on the platform-specific implementation

In a simplified scenario from the infusion system, suppose there are two monitored variables *M_Start_Infusion*, which is a button that the patient can press to start infusing, and *M_Air_in_Line*, which is a sensor that monitors if there is air in the flow of the medication. Correspondingly, there are two input variables *I_Start_Infusion*, which indicates if the start-infusion button has been pressed, and *I_Air_in_Line*, which indicates if air-in-line has been detected by the sensor. Furthermore, *O_Flow_Rate* is an output variable that represents the computed infusing flow rate and *C_Flow_Rate* is the actual flow rate enforced by the pump motor.

Suppose we have a model-based test scenario in which *I_Start_Infusion* and *I_Air_in_Line* become *true* at the same time. When executing this test scenario on the model, *O_Flow_Rate* is always 0, which is the expected behavior. Otherwise, the air could be infused and cause serious consequences to the patient.

When executing this test scenario on a platform-specific implementation, however, this test may pass or fail depending on the platform. Specifically, an execution platform may have a longer delay in converting the quantity of *M_Air_in_Line* into *I_Air_in_Line* than converting the quantity of *M_Start_Infusion* into *I_Start_Infusion*, which can happen because data sampling and processing in the sensor can take longer time than transmitting an electrical signal in the button. Then Figure 4 illustrates how *C_Flow_Rate* changes. Specifically, *M_Start_Infusion* and *M_Air_in_Line* become $true$ at $t0$; *I_Start_Infusion* becomes $true$ at $t1$, which starts infusing; *I_Air_in_Line* becomes $true$ at $t2$, which tries to stop infusing by setting *O_Flow_Rate* to 0; eventually, *C_Flow_Rate* becomes 0 at $t3$ because of the delay in the pump motor. As a result, a failure is rendered even if the system is acting correctly, which is a false positive, because the model oracle does not take into account delays from the platform-specific implementation and the original test scenario has also been changed.

The effect of such shifting may affect testing effectiveness in multiple ways. First, although the scenario at system level is "pressing start-infusion button with air-in-line detected", the scenario at the model level becomes "air-in-line is detected during infusion". Thus, it is not surprising that the model-based oracle does not match system outputs. Second, depending on the types of tests generated from the model, for example, model-based tests may execute a certain part or behavior of the system with a specific combination of inputs. When such a combination is lost, testing may fail to find the faults that should have been found.

Alternatively, considering the longer delay the air-in-line sensor has, we could potentially schedule *M_Air_in_Line* to be $true$ before *M_Start_Infusion* becomes $true$, such that *I_Air_in_Line* and *I_Start_Infusion* can be $true$ at the same time on the software controller.

III. Approach

A. Framework Definitions

We build on the four-variable model and introduce a few additional definitions. For a given variable v, we use v^t to represent the time-function of its value [9], where the domain consists of real numbers (i.e., time) and the range consists of all possible values of v in a real-time environment. We also define v^k to represent the step-function of its value, where the domain consists of integers (i.e., steps) and the range consists of all possible values of v in a discrete-time environment. Furthermore, the value of v at time t and step k are represented by $v^t(t)$ and $v^k(k)$, respectively.

1) Model-based Tests and SOF-Delay: In a system with r input variables and s output variables, we characterize a specific model-based test interaction in the following form:

$$(i_1^k(k_1), i_2^k(k_1), ..., i_r^k(k_1))$$

and its oracle in the form:

$$(o_1^k(k_1), o_2^k(k_1), ..., o_s^k(k_1))$$

where k_1 is a specific step number (e.g., the first step).

Test execution on the model and its implementation is considered to be step-wise (i.e., in terms of discrete time). For example, when executing a test case for conformance testing of a Simulink model and a corresponding C implementation running on the target platform, at each step, a test interaction is executed on both the model and the implementation, and their outputs are compared. The implementation conforms to the model for this test case if there is no discrepancy between their outputs at every step. Thus defined, model-based tests and oracles implicitly have the following two features.

1) Model-based tests and oracles are in terms of discrete time, which makes it far easier to record the actual outputs and compare them with the oracle. A naive value comparison for each output variable at each step would do the work.
2) The test and oracle execute at the *same* time, which abstracts away the fact that the execution itself takes time and this zero-delay assumption can be problematic in model-based testing.

Now we lift this test interaction and its oracle from discrete-time to real-time:

$$(i_1^t(t_1), i_2^t(t_1), ..., i_r^t(t_1))$$

and its oracle would be

$$(o_1^t(t_1 + \Delta_{SOF}), o_2^t(t_1 + \Delta_{SOF}), ..., o_s^t(t_1 + \Delta_{SOF}))$$

where t_1 is the time of step k_1 and Δ_{SOF} is the execution delay of the software controller. Note that, while we consider timing of different sensor and actuator components separately, we view the software controller as a single synchronous component. That is, the software controller takes all input variable values at the same time t_1 and produces all output variable values at the same time $t_1 + \Delta_{SOF}$[1], where Δ_{SOF} represents the execution delay of the software controller and it is a platform-specific non-deterministic value.

2) IN-Delay: The IN relation maps monitored variables to input variables and we define the delay from a change to monitored variables to a change to input variables as *IN-Delay*.

IN-Relation is defined between two vectors, i.e., $\underset{\sim}{m}^t$ and $\underset{\sim}{i}^t$, while the exact mapping between elements of $\underset{\sim}{m}^t$ and $\underset{\sim}{i}^t$ are implicit. When it comes to *IN-Delay*, informally, within an input device component, Δ_{i_j} is the delay from the time that the corresponding monitored variable changes values, to the time that input variable i_j changes values. And we do not explicitly define which monitored variable(s) map to i_j in order to simplify our definition, but such a mapping indeed exists. For example, if we have an input device component that monitors flow rate, the *M_Flow_Rate* variable represents the actual flow rate and the *I_Flow_Rate* variable is a sampled flow rate from the flow rate sensor. Therefore, $\Delta_{M\_Flow\_Rate}$, representing the delay from the time *M_Flow_Rate* changes value to the time its input variable(s) change values, would be equivalent to $\Delta_{I\_Flow\_Rate}$, representing the delay from its monitored variable(s) change values to the time *I_Flow_Rate* changes value.

In a special case where p monitored variables have a one-to-one mapping to p input variables, we would have that Δ_{m_j} is equivalent to Δ_{i_j} where $1 \leq j \leq p$.

Given a specific test interaction at system level:

$$(m_1^t(t_1), m_2^t(t_1), ..., m_p^t(t_1))$$

the corresponding test interaction at the software controller level would be

$$(i_1^t(t_1 + \Delta_{i_1}), i_2^t(t_1 + \Delta_{i_2}), ..., i_r^t(t_1 + \Delta_{i_r}))$$

where Δ_{i_j} represents the timing delay defined above.

Ideally, if all the delays are known values and if we want an input variable vector:

$$(i_1^t(t_1), i_2^t(t_1), ..., i_r^t(t_1))$$

[1]Technically, it is impossible even for two consecutive assignments to happen at exactly the same time, but the difference is often too small to be captured, so we still treat them as at the same time.

the monitored variable vector would be

$$(m_1^t(t_1-\Delta_{m_1}), m_2^t(t_1-\Delta_{m_2}), ..., m_p^t(t_1-\Delta_{m_p}))$$

Therefore, if we know all the exact values of Δ (which is unfortunately non-deterministic but can often be characterized), all monitored variable values can be perfectly aligned such that, given this monitored variable vector, the software controller will get the input variable vector, and produces an output variable vector that matches its oracle.

3) OUT-Delay: The OUT relation maps output variables to controlled variables and we define the delay from a change to output variables to a change to controlled variables as *OUT-Delay*, which can further be defined in the same way as *IN-Delay*, by replacing monitored variables with output variables, input variables with controlled variables, and sensors with actuators.

In a system with q controlled variables, when executing the above monitored variable vector, we will get a controlled variable vector as the following:

$$(c_1^t(t_2), c_2^t(t_2), ..., c_q^t(t_2))$$

Similarly, the output variable vector would be

$$(o_1^t(t_2-\Delta_{o_1}), o_2^t(t_2-\Delta_{o_2}), ..., o_s^t(t_2-\Delta_{o_s}))$$

Other relationships can be characterized similarly as in monitored and input variables.

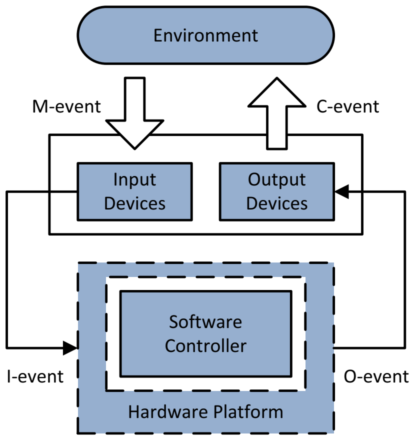

Fig. 5: The architectural view of the platform-specific implementations

4) Events: We define an event as setting any one of the *m-i-o-c* variables to a specific value at a specific time. We do not require that two consecutive events to be different, that is, if we take flow rate as an example, samples obtained within a certain period of time may all be the same. Physical events (e.g., pressing a button) can be defined similarly, that is, a button pressed electrical signal will set the button pressed input variable to be true.

Specifically, if we take the flow rate sensor as an example, the sensor itself runs a sampling routine at a certain rate. The actual flow rate is a monitored variable and the sampled flow rate is an input variable. The software controller always takes existing input variables' values to update its internal state. In our case of executing model-based tests, each test interaction has events for all the *m-i-o-c* variables. We use *M-I-O-C-event* to represent events that set values of *m-i-o-c* variables, respectively.

B. Platform-specific Implementations

Figure 5 shows an overview of the execution environment of platform-specific implementations. Specifically, a platform-independent model can often be translated and compiled to the code that runs on the hardware platform and we refer to this piece of code as *software controller*. Besides, a platform-specific implementation also contains input devices (i.e., sensors) and output devices (i.e., actuators). The sensor takes stimuli from the physical environment, while a stimulus can be a physical event (e.g., pressing a button) or pre-processed inputs (e.g., sampling flow rate), and the actuator causes some effect in the physical environment (e.g., starting a motor or turning on a light).

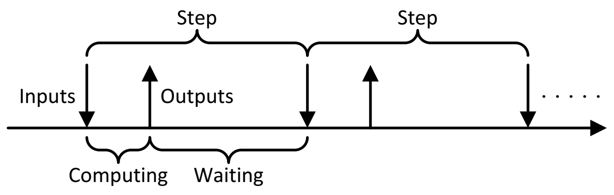

Fig. 6: The input/output timing in real-time step

Figure 6 shows typical real-time steps in the software controller. The step starts with obtaining inputs to update its internal state, then producing outputs, and waiting for a fixed time frame. We define *step size* on the implementation as the time interval between two consecutive I-events or O-events.

We have described how monitored and output variables can be derived from input and controlled variables, respectively, in an environment of known delays of input and output devices. However, all the delays are non-deterministic on the execution platform. That is, when executing a reverse mapping $\underset{\sim}{m}^t$ from $\underset{\sim}{i}^t$ on the platform, the resulting input variable vector $\underset{\sim}{i}'^t$ is not always equivalent to $\underset{\sim}{i}^t$.

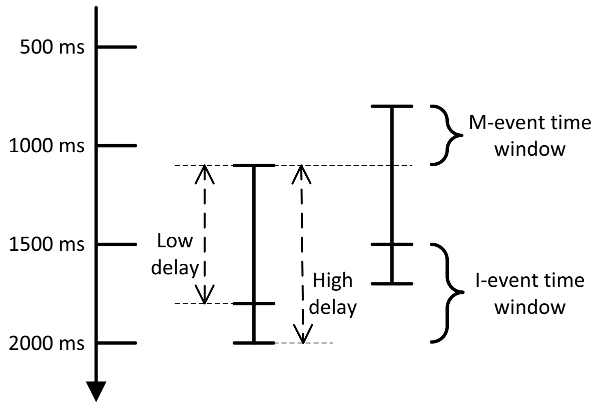

Fig. 7: Time window for M- and I-events

Here, we define the notion of *time window* as a period of time during which an event should occur, such that when executed on the platform with non-deterministic delays, it can trigger expected effects. Figure 7 shows an example. We assume that one step is mapped to 500 ms in this example. Suppose that an I-event is expected between the time 1500 ms and 2000 ms as shown, then the time window for the corresponding M-event is estimated based on the upper-bound (shown as *high delay*) and lower-bound (shown as *low delay*) of delays. Note that, however, when the variance of delays is too large, the time window for M-event can be too small or even does not exist.

C. Direct Execution of Model-based Tests

Figure 8 shows executing a test case on a platform-specific implementation. When an M-event is given, a C-event (i.e., a response) is expected during the time we refer to as estimated C-event time window. Due to various delays (i.e., IN-delay, SOF-delay, and OUT-delay), the actual C-event cannot be produced within the expected time frame. Thus, there is a mismatch and this test fails, although the software controller works as expected.

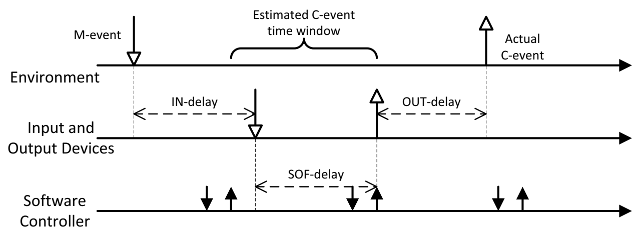

Fig. 8: Real-time execution

D. Scheduling M-events

Figure 9 shows how our approach schedules M-events.

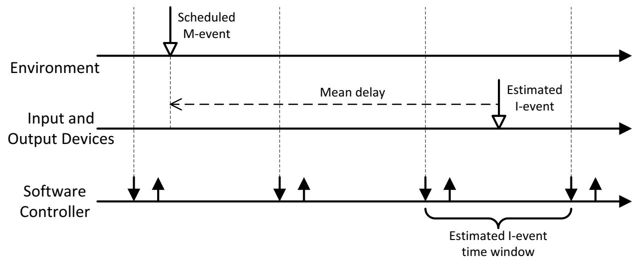

Fig. 9: Scheduling M-events

Given a model-based test interaction, we start with estimating a future time window during which all I-events should occur for all input variables. Suppose the largest delay of all input variables is Δ_{max}, and the step size is $step$. Both Δ_{max} and $step$ are mean delay values obtained from the execution history [10].

In order to maximize the possibility that an input event can occur during our expected time window, the estimated future I-event time should be in the middle of two consecutive steps. Therefore, if we have $current$ as the timestamp at which the software controller finishes one step, then the future time $time_{\textit{I-events}}$ when all I-events of the current test interaction should happen is estimated as:

$$\begin{aligned} time_{\textit{I-events}} = {} & current - step/2 \\ & + ceil((\Delta_{max} + step/2)/step) * step \end{aligned}$$

then a monitored variable m_j with delay Δ_{m_j} should be sent at time

$$time_{\textit{M-event}_j} = time_{\textit{I-events}} - \Delta_{m_j}$$

which is shown as *scheduled M-event* in Figure 9.

Although this is also an online real-time system testing approach, the overhead is minimized. Model-based test generation is offline to take advantages of many automated tooling support. During online execution, the scheduling overhead is often too small to create timing discrepancies. However, when too many events are scheduled to happen at the same time, the overhead can be significant enough to affect system behavior. Therefore, we also set a threshold such that events with similar scheduled time (specifically, similar scheduled time is in this case all subsequent events that have a scheduled time no later than 10 ms from the current event) will be combined and sent together to mitigate the scheduling overhead.

E. Adjusting C-events

Figure 10 shows how our approach adjusts C-events.

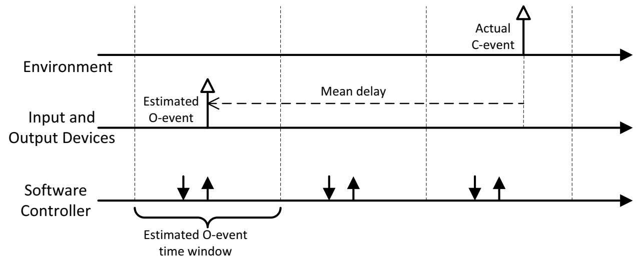

Fig. 10: Adjusting C-events

If scheduling M-events has been done successfully, the software controller should have executed the expected test scenario originated from model-based tests and produced O-events that match model-based test oracles. But what the environment receives are C-events. Then all we need to do is to bring the timing of C-events back to the model level and compare to see if they match the oracle output.

Specifically, given the same $time_{\textit{I-events}}$ from scheduling M-events, we define a time window during which the corresponding O-event should happen, i.e., between $time_{\textit{I-events}}$ and $time_{\textit{I-events}} + step$. Therefore, given a C-event at time $time_{\textit{C-event}_j}$, an output variable o_j with delay Δ_{o_j} should have been produced at time

$$time_{\textit{O-event}_j} = time_{\textit{C-event}_j} - \Delta_{o_j}$$

which would fit into one of the expected O-event time windows and its value is compared with oracles to determine if the test passes or fails.

F. Benefits of Our Approach

Test cases for system testing are usually written manually, which is a time consuming process and often, only a small number of test cases can be written and executed, leaving potentially many aspects of the system untested. Although the software itself may have been tested heavily, it is unlikely to have been exercised in the context of system-level test scenarios.

When model-based tests are used to test an implementation, frequently the "wrong" test cases may be executed (because the scenario exercised is quite different) and then compared with the "wrong" oracle (because the model and the implementation exercised different behaviors). Our approach provides a way to execute model-based tests (or any test cases that can be generated from a platform-independent model) in a way that is appropriate for the specific implementation platform.

While test cases can be generated relatively easily from platform-independent models, without the proposed framework, detailed models of platform specific components are needed to derive tests that can be executed on the implementation. This makes test generation harder because of model complexity and affects scalability. The proposed approach addresses this by abstracting the platform specific details to a minimal set of parameters (e.g., the mean and variance of timing delays) and using that information during test execution instead of test generation.

IV. Evaluation

A. Research Questions

We wish to evaluate the following two aspects:

First, to what extent does the proposed approach reduce false positives? False positives naturally arise during testing platform-specific implementations using model-based tests. Starting with a test suite that passes on the platform-independent model, we would like to understand the percentage of tests that can still pass on the platform-specific implementations with different timing configurations. Then we can evaluate how well the false positive rate can be reduced using the proposed approach. Furthermore, what are the platform-specific characteristics that affect the effectiveness of our approach? Specifically, we would like to characterize the applicability of the proposed framework in terms of timing delays at different system boundaries.

Second, does the proposed approach introduce additional false negatives? Some existing techniques that test platform-specific implementations aiming at reducing false positives often introduce additional false negatives [4], which can be particularly bothersome for safety-critical systems. Empirical evidence of limiting false negatives using the proposed approach will strengthen the case for its applicability to safety-critical system domains.

B. Case Study Example

We used a PCA (Patient-Controlled Analgesia) infusion pump system as our case study example [11]. This system is modeled using MathWorks Simulink and Stateflow [12], [13].

The top level system has 76 input variables and 31 output variables. This system also contains seven subsystems. Table I shows basic information of the subsystems. Some of the subsystems have been used as case examples in previous studies [14], [4], [15].

C. Device Configuration

Code for platform testing was generated from the Simulink and Stateflow models using the MatLab Simulink Coder [16].

TABLE I: Infusion pump subsystem information

Subsystem	# Input Vars	# Output Vars
Alarm	102	5
Config	63	25
Infusion_Manager	53	5
Logging	43	2
System_Statistics	69	5
System_Monitor	3	1
Top_Level_Mode	30	3

This tool produces platform agnostic C code. To execute this code on a target, platform specific code is needed to configure the target peripherals and memory. The target platform manufacturer provides this code in the form of a Board Support Package (BSP). Other than a small assembly language bootstrap the remainder of the BSP consists of libraries of short routines for interfacing with the platform's peripherals (e.g. clocks, I/O, timers). A small amount of handwritten code was required to create the Interrupt Service Routines (ISR) for asynchronous sensor inputs such as button presses and limit switch activations. The top-level executive is a simple infinite loop that repeatedly executes the Simulink generated code and then sleeps until the next execution cycle time. During sleep, ISRs are still supported to prevent sensor inputs from being lost between increments.

The platform used for this testing was an Atmel ARM91SAM7X development board. In order to show the generic nature of the development procedure described here, the same code was recompiled and executed on a Pololu Orangutan SVP Robot Controller. External hardware was added to simulate the functions of the infusion pump including a motor, buttons to simulate user input and limit sensors, annunciators, and LED displays. The on-chip USART was used to create a simple monitor port to allow printf statements to be observed during execution.

A 500 ms timing loop was used to slow down execution to allow ample time for user interaction with the system. The selection was arbitrary and could have been significantly shorter. In order to automate interactions with the device during testing for sending sensor events and receiving actuator events, the device was connected to a PC through a serial port.

D. Simulating Sensors and Actuators

We identified multiple sources of timing delays and we wanted to assess how the delays affect testing and how our approach reconciles timing induced mismatches. In order to create a variety of platform-specific configurations, we further created additional sensor and actuator timing delays.

Specifically, the modeled sensor has two threads. The first thread constantly reads M-events from the test driver and the second thread creates additional delays for each M-event. When an M-event's delay time has elapsed, the M-event will be sent to the software controller as an I-event. I-events will update the software controller's input variable states used to update the software controller's internal state in the next step.

Similarly, the modeled actuator has two threads in which the first thread constantly reads O-events from the software controller and the second thread creates additional delays for each O-event. When an O-event's delay time has elapsed, the O-event will be sent to the test driver as a C-event.

In this study, randomized sensor and actuator delays follow a normal distribution. For a platform-specific configuration, we characterize it using *max mean* and *standard deviation*. *Max mean* is used to provide an upper bound of mean delay for each event. In a specific platform, each event would have a random mean delay between 0 and *max mean* and a fixed *standard deviation*. The mean delay and standard deviation of each event is used to create a randomized delay during execution. Note that on both the sensor and the actuator, if the generated random delay is negative, 0 is used.

In our study, *max mean* values used were 250, 500, 1000, and 2000 ms and *standard deviation* can be 0, 25, 50, 100, and 200 ms. Therefore, we would have 20 different platform-specific configurations, plus the ideal situation in which both max mean and standard deviation are 0, i.e., there is no delay[2].

E. Test Case Generation

Though test cases from any source can be used with our approach, we generated our test suite using Simulink Design Verifier [17] to take advantage of its automated model-based test generation capability. Tests were generated for the branch coverage criterion to provide a rich and realistic set of cases.

Specifically, there were a total of 592 branch coverage objectives for the model. Among them, test cases could be generated to satisfy 524 objectives. 41 objectives were proven to be unsatisfiable and 27 objectives were undecidable within the given time budget (10 hours in our experiment). *CombinedObjectives (Nonlinear Extended)* option was used to reduce the test suite size while maintaining coverage. As a result, the test suite was reduced to 116 test cases with a total of 1048 test interactions. Test oracles were also generated from the model and later used to check if each test interaction passes or fails on platform-specific implementations.

We executed the test suite 5 times for each of the 21 different platform-specific configurations, once with the framework mediated timing adjustments and once directly executed, for a total of 210 times, with a total of more than 220K test interactions.

F. Mutation Generation

We generated mutants using Milu, a mutation testing tool for C programs [18]. We generated 50 mutants evenly distributed across the seven subsystems based on the total number of possible mutants in each subsystem.

Equivalent mutants, which are behaviorally equivalent to the original system, can jeopardize the use of mutation testing. Although detecting equivalent mutants is possible, specifically in the case of finite state systems [19], [20], it does not scale well. Therefore, we did not attempt to remove equivalent mutants since it is cost-prohibitive for our case example systems.

[2]This is essentially the same as executing test cases on the platform-independent model, although the system still exhibits negligible delays due to, e.g., transition.

We ran our test suite on all the mutants in an execution platform with *max mean* to be 500 ms and *standard deviation* to be 50 ms.

We first executed the test suite directly on the implementation (without delays) to determine if the mutant can ever be killed. If so, we then executed the test suite using our approach in the above platform-specific configuration, and check if those failed tests can still fail.

V. Results & Discussion

In this section, we address our research questions and discuss the implications of our results. We begin by presenting false positives that can be reduced using our approach.

A. Reducing False Positives

We ran the full test suite with 1048 test interactions on 21 different platform-specific configurations. On each configuration, we ran the full test suite 5 times in order to account for non-determinism in terms of timing from the hardware platform as well as from the injected sensor and actuator delays. All test interactions are supposed to pass on the platform-specific implementations.

TABLE II: Median number of passed tests and percentage point decrease in false positives

Max Mean	Std. Deviation	Scheduled Execution	Direct Execution	FP Decrease
0	**0**	1048 (100%)	1048 (100%)	0%
250	**0**	1048 (100%)	884 (84.35%)	15.65%
	25	1048 (100%)	463 (44.18%)	55.82%
	50	1048 (100%)	682 (65.08%)	34.92%
	100	899 (85.78%)	177 (16.89%)	68.89%
	200	197 (18.8%)	68 (6.49%)	12.31%
500	**0**	1048 (100%)	273 (26.05%)	73.95%
	25	1048 (100%)	6 (0.57%)	99.43%
	50	1048 (100%)	67 (6.39%)	93.61%
	100	840 (80.15%)	3 (0.29%)	79.86%
	200	161 (15.36%)	8 (0.76%)	14.6%
1000	**0**	1048 (100%)	3 (0.29%)	99.71%
	25	1048 (100%)	16 (1.53%)	98.47%
	50	1048 (100%)	14 (1.34%)	98.66%
	100	843 (80.44%)	4 (0.38%)	80.06%
	200	117 (11.16%)	5 (0.48%)	10.68%
2000	**0**	1048 (100%)	3 (0.29%)	99.71%
	25	1048 (100%)	2 (0.19%)	99.81%
	50	1048 (100%)	3 (0.29%)	99.71%
	100	831 (79.29%)	2 (0.19%)	79.1%
	200	134 (12.79%)	2 (0.19%)	12.6%

Table II shows the number and percentage of passed test interactions using the proposed approach (i.e., scheduled execution) and direct execution of model-based tests on each platform-specific configuration. The percentage point decrease in false positives is simply calculated by subtracting failed percentage of test interactions using scheduled execution from that using direct execution.

It is not surprising that direct test execution and output comparison are very sensitive to time fluctuation. As timing delays are randomly injected, direct execution would start to fail quickly when delays increase. In general, mean delay dominates the number of passed/failed tests in direct execution, which is also intuitively straightforward since an event is more likely to miss its time window with larger delays, leading to unexpected output. Standard deviation may affect direct execution in multiple ways. A larger standard deviation in general leads to more failed tests with small max mean delays (i.e., 250 and 500 ms), but since direct execution is completely unguided, a larger standard deviation may also have more passed tests (e.g., 250-25 and 250-50) for the same max mean delay. This happens because the configuration 250-25 randomly assigned higher delays to those variables that are more sensitive to time fluctuation. With large max mean delays (e.g., 1000 and 2000 ms), around 99% tests would fail in spite of standard deviation.

The proposed approach reduces false positives in all cases (except the 0-0 case where there is no false positive). As shown in Table II, our approach is robust to absolute delay values (i.e., max mean), but can still be sensitive to delay variance (i.e., standard deviation) when it is large. Specifically, our approach can reduce most false positives with a standard deviation less than 200 ms despite mean delays, but start accumulating false positives with the 200 ms standard deviation. The event scheduling mechanism in our approach ensures that each event can fit in the right time window, but if we recall the definition of time window in Figure 7, for example, a larger delay variance would lead to a narrower time window for M-events. When the delay variance is large enough, the time window for the corresponding M-event may not exist, thus the event scheduling cannot guarantee that the I-event can be received by the software controller at the right step.

Our approach always estimates and expects I-events to happen in the middle of two consecutive steps, which already give both direct and scheduled executions certain tolerance to timing delays. In our study, we used an almost fixed delay to schedule M-events from I-events, and convert C-events to O-events. Since the actual sensor/actuator have non-deterministic timing delays, it would be difficult for the event scheduling to be accurate when the delay variance is too large. It would still be possible to schedule input events accurately if there are more known characteristics of the delay distribution. Then applying a more sophisticated scheduling algorithm would reduce false positives further. For example, the delay may vary widely overall, but may not change much during a short period of time. In such a case, using a short period of history delays would make the scheduling much more accurate. We leave such improvements as future work.

We used large variances of delays for the purpose of evaluation. Although our approach did not perform well on the 200 ms standard deviation settings, they may not even be realistic in practice. For example, even if we assume normal distribution of delays with a mean of 1000 ms and a standard deviation of 200 ms, then 68.3% of the values will be within the range of 800 - 1200 ms, and 95.4% of the values will be within the range of 600 - 1400 ms, which are already unrealistically large ranges of delays.

B. Eliminating False Negatives

Reducing false positives in our approach is essentially accepting *good* system behaviors that would be rejected otherwise, but it may also run the risk of accepting *bad* system behaviors.

We first performed a mutation testing on the platform-independent model. For each of the 50 generated mutants, we ran the full test suite with 1048 test interactions and recorded the test interactions that failed on the mutant. Specifically, in our 52400 (i.e., 1048 test interactions on 50 mutants) test interaction executions, 13081 test interaction executions failed due to injected mutants. We selected a representative platform-specific configuration with a *max mean* delay of 500 ms and a *standard deviation* of 50 ms. We then performed the same mutation testing on the platform-specific implementation and observed whether each of the 13081 failed test interaction executions still fails and we call it false negative if it passes. In our experiment, we did not observe any false negative as a result of using our approach.

Existing approaches, e.g., oracle steering, may accept system behaviors that are similar enough to the model behaviors, while a similarity threshold is often set manually in order to constrain and balance the number of false positives and negatives. Unlike prior work, our approach does not modify the model or the oracle for the purpose of accepting system behaviors with discrepancy, avoiding the risk of accepting bad system behaviors.

Nevertheless, since testing itself is incomplete and has false negatives (i.e., testing can only show the presence of faults, not their absence), our approach only ensures that no additional false negatives can be introduced. Thus, the quality of the original model-based tests plays an important role in finding faults and reducing overall fault negatives.

VI. Threats to Validity

External Validity: We used only one system as the case example to evaluate our approach. Our experiment could be limited, but we actively worked with domain experts to set up a realistic experimental environment. Although there is only one case example in our study, the infusion pump system consists of subsystems some of which have been used in previous studies [14], [4], [15] as standalone systems.

We used several MathWorks tools for building models and generating code and test cases. We also used the actual hardware device in our study. Since these are widely used commercial tools, we believe that our results can be generalizable to other systems in these domains.

Besides the actual delays on the device, we also simulated additional delays in order to create a variety of platform-specific configurations. These simulated delays, however, may not represent realistic hardware specifications, but they do show that our approach can work in a variety of platform-specific configurations.

Internal Validity: We used a specific test suite generated from Simulink Design Verifier satisfying branch coverage on the model in our experiments. The results may change if other forms of test suites are used. However, both branch coverage and model-based tests are commonly used in practice.

We used measured delay information in our approach to schedule and adjust events. The measured delays in our experiments may not reflect the precision that one could obtain in practice. However, we also simulated unrealistically large delays and large variances of delays in order to account for possible biases.

Construct Validity: We used mutants rather than real faults to demonstrate that our approach can eliminate false negatives that can possibly be introduced by other similar techniques. It is possible that using real faults would lead to different results, although mutation testing has been widely used and has been shown to be similar to real faults [21].

VII. Related Work

Kim et al. defined testing and verification frameworks, using an approach similar to the one described here, that can precisely capture timing at different system boundaries [10], [22]. Specifically, they manually created a limited number of system level tests from system timing requirements. Additional delays due to other components such as hardware are measured and added to the timing requirements to relax passing criteria, reducing false positives from executing model-based tests on platform-specific implementations. However, the frameworks do not attempt to reproduce the intended scenario of the original model-based tests and use only single stimulus (e.g., pressing a button). The goals also do not include automated test generation and execution. For a system at the scale of our case example, creating thousands of test interactions each of which involves more than 100 variables would be impractical without automation.

Several techniques support limited forms of non-determinism [23], [24], [1], [25] by introducing extended model formats such as timed automata with real-valued clocks [2] and UPPAAL [3]. Specifically, non-deterministic timing behaviors are introduced in the models. False positives can be reduced if the software model can explicitly account for the same amount of non-determinism as the system has. These approaches, however, often result in an exponential increase in the difficulty of demonstrating conformance, restricting the amount and type of non-determinism that can be handled.

Our approach does not require changes to the original software model to account for non-determinism. In fact, adding system non-determinism contradicts the original intent of using platform-independent models, which abstract away platform-specific details to make the models amenable to various analyses. Besides, introducing non-determinism can often increase false negatives. While false positives in testing embedded real-time systems often lead to wasted manual effort, false negatives (i.e., missed faults) can lead to catastrophic consequences. Furthermore, our approach decouples test generation and execution, as well as platform-independent models and platform-specific non-determinism, and therefore can be used to extend many existing approaches.

Larsen et al. defined an online testing framework based on UPPAAL models for testing real-time systems [1], [25]. In their work, test cases can be generated and executed, and test results can be checked, all online. A test step is generated from the model and executed on the SUT at one time, which reduces the size of reachable state space and thus improves scalability. Despite the fact that their work also requires changes to the original model to accept non-deterministic system behaviors, this online testing relies on fast test and oracle generation, and behavior comparison, which often introduce significant overhead that can change input/output events and thus system behaviors. Therefore, its application is restricted by the size and complexity of systems. While our approach to test execution and comparison is also online, model-based tests and oracles are generated *offline*. Event scheduling happens online, but the overhead is typically negligible.

Oracle steering was recently proposed as an alternative approach to reduce false positives [4], in which the model is *carefully* steered to exhibit behavior that is closer to the observed behavior of the SUT. To ensure that undesirable system behaviors are not overlooked by the oracle, constraints are placed on how far the model could be steered to accommodate the observed system behavior. This is achieved by first attempting to execute equivalent tests, one step at a time, on both the system and the model, and when there is a mismatch of the corresponding outputs, modifying the model inputs (and often parts of the model state) within reasonable bounds so that the model behavior can be nudged closer to the system behavior. Empirical assessment of the approach confirmed that such a steered model is more effective as an oracle – the number of false positives (reported mismatches not attributable to real system defects) is greatly reduced, but the number of false negatives has also increased. Furthermore, the approach is defined and built over discrete time, and may not be able to handle the richness of real-time behaviors.

VIII. Conclusions

We have described our testing framework based on the four-variable model defined by Parnas et al. [10] and the framework for testing timing defined by Kim et al. [9]. Our testing framework precisely captures timing of different hardware components. Our scheduling-based approach enables executing model-based tests on platform-specific implementations. Our approach brings together the advantages of existing automated model-based testing and oracle generation tools. Tests and oracles are generated offline, while test scheduling, execution, and test result checking are online. This approach has the benefit of online testing – the scheduling can be adaptive to the timing change at runtime, but avoids common problems in online testing – tests are generated offline using existing tools to reduce online overhead.

Future work is based on the following observations.

(1) Our approach takes a fixed mean delay of each hardware component and schedules events. This approach does not perform very well in an execution platform where delays have large variance. Instead, a more sophisticated scheduling algorithm can be applied that can make better use of delay information.

(2) Event sequences generated by our approach can contain important timing information. For example, the timing gap between two events (specifically, between an M-event and a C-event) can reflect whether the platform-specific implementation satisfies system level timing requirements.

Acknowledgment

We would like to thank Anitha Murugesan at the University of Minnesota, for providing support for the Simulink and Stateflow models.

References

[1] Kim G Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using UPPAAL. In *Formal Approaches to Software Testing, 4th International Workshop*, pages 79–94, 2005.

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

[3] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on UPPAAL. In *Formal methods for the design of real-time systems*, pages 200–236. 2004.

[4] Gregory Gay, Sanjai Rayadurgam, and Mats P. E. Heimdahl. Improving the accuracy of oracle verdicts through automated model steering. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 527–538, 2014.

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

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

[7] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In *Modeling and verification of parallel processes*, pages 187–195. 2001.

[8] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[9] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer programming*, 25(1):41–61, 1995.

[10] BaekGyu Kim, Hyeon I Hwang, Taejoon Park, Sang H Son, and Insup Lee. A layered approach for testing timing in the model-based implementation. In *Proceedings of the 2014 Design, Automation & Test in Europe Conference & Exhibition*, pages 1–4, 2014.

[11] Anitha Murugesan, Michael W Whalen, Sanjai Rayadurgam, and Mats PE Heimdahl. Compositional verification of a medical device system. In *Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology*, pages 51–64, 2013.

[12] MathWorks Simulink. http://www.mathworks.com/products/simulink.

[13] MathWorks Stateflow. http://www.mathworks.com/products/stateflow.

[14] Gregory Gay, Matt Staats, Michael Whalen, and Mats Heimdahl. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*, 2015.

[15] Anitha Murugesan, Michael W Whalen, Neha Rungta, Oksana Tkachuk, Suzette Person, Mats PE Heimdahl, and Dongjiang You. Are we there yet? Determining the adequacy of formalized requirements and test suites. In *NASA Formal Methods*, pages 279–294. 2015.

[16] MathWorks Matlab Coder. http://www.mathworks.com/products/matlab-coder.

[17] MathWorks Simulink Design Verifier. http://www.mathworks.com/products/sldesignverifier.

[18] Yue Jia and Mark Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *TAIC PART'08*, pages 94–98, 2008.

[19] CAJ Van Eijk. Sequential equivalence checking based on structural similarities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(7):814–819, 2000.

[20] Michael Whalen, Gregory Gay, Dongjiang You, Mats PE Heimdahl, and Matt Staats. Observable modified condition/decision coverage. In *Proceedings of the 35th International Conference on Software Engineering*, pages 102–111, 2013.

[21] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[22] BaekGyu Kim, Lu Feng, Linh TX Phan, Oleg Sokolsky, and Insup Lee. Platform-specific timing verification framework in model-based implementation. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 235–240, 2015.

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

[24] Abdeslam En-Nouaary, Rachida Dssouli, and Ferhat Khendek. Timed Wp-method: Testing real-time systems. *IEEE Transactions on Software Engineering*, 28(11):1023–1038, 2002.

[25] Kim G Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using UPPAAL-TRON: An industrial case study. In *Proceedings of the 5th ACM International Conference on Embedded Software*, pages 299–306, 2005.