Automated Model-based Test Generation for Platform-specific Implementations

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

Dongjiang You

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

Dr. Mats P.E. Heimdahl

June, 2016

Acknowledgements

I would like to thank all those people who have made this dissertation possible. My deepest gratitude is to my advisor, Mats Heimdahl, for his invaluable support and guidance of my graduate study and research. I have been amazingly fortunate to have an advisor who has taught me innumerable lessons and insights on academic research and gave technical and editorial advice to the completion of this dissertation.

I would like to express my very great appreciation to Sanjai Rayadurgam and Michael Whalen, who devoted their time and practical experience to helping me sort out technical details of my work. They have been an excellent source of guidance and motivation. I am grateful to Pen-Chung Yew and Kia Bazargan for serving on my doctoral final exam committee and for their advice on this dissertation.

I would like to thank the following current and former members of the Critical Systems Group at the University of Minnesota for being both great colleagues and great friends: Matt Staats, Hung Pham, Gregory Gay, Ian De Silva, Jason Biatek, Anitha Murugesan, Lian Duan, Kevin Wendt, Andreas Katis, Elaheh Ghassabani, and Taejoon Byun.

I am thankful to Rockwell Collins for granting access, through Michael Whalen, to industrial case examples. I am especially thankful to Andrew Gacek for providing help in their tools and systems. I am grateful to BaekGyu Kim, Oleg Sokolsky, John Komp, Isaac Amundson, and Scott Hareland for providing opportunities to collaborate on research and for their guidance and insightful comments. Without their support, this dissertation would be far less practically relevant.

Finally, I owe my gratitude to my parents, Pingshun Wang and Baozhong You, and especially my wife, Bohan Shao, for always supporting and encouraging me.

Abstract

In model-based testing of safety-critical systems, structural coverage criteria have been widely used to measure test suite adequacy as well as a target when generating tests. We have found that the fault-finding effectiveness of tests satisfying structural coverage criteria is highly dependent on program structure; and even if the faulty code is exercised, its effect may not be observable at the output. To address these problems, we define observability as a desirable attribute of testing to mandate that the effect of exercising a structural part must manifest itself at a subsequent observable point in the program. We further propose an incremental test generation approach that combines the notion of observability and dynamic symbolic execution. Our results show that the notion of observability together with the incremental test generation approach are effective at detecting faults, robust to program restructuring, and efficient in generating tests.

On the other hand, advances in automated test generation from system models do not always translate to realizable benefits in terms of testing an implementation of the system, 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. 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 our framework eliminates false negatives while keeping the number of false positives low for a variety of platform-specific implementations.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Model-based development is a paradigm in which software and system development is focused on high-level executable models. These models represent the desired system behaviors and allow various types of analysis including model-checking [1, 2], theorem proving [3, 4], and test case generation [5, 6]. Specifically, model-based testing is an application of using models as the source for deriving test inputs and test oracles to perform software and system testing. Automated test generation has been applied widely in both academia and industry.

Structural test adequacy criteria – which specify that various structural parts of a program must be exercised – such as branch coverage and modified condition/decision coverage (MC/DC) often serve as useful benchmarks for assessing the thoroughness of testing software and for meeting regulatory requirements [7]. Of particular interest to us are criteria used to evaluate testing effort for safety-critical systems, such as MC/DC [8], which is mandated by the U.S. standard DO-178C [7] for testing the most critical avionics software. In previous investigations, we have found that the effectiveness of structural coverage criteria is highly dependent on the structure of the program under test [9]. Simple syntactic transformations, such as inlining of variables, have a dramatic effect on the fault finding effectiveness of test suites designed to satisfy the MC/DC criterion.

This effect is not surprising – structural criteria, as the name suggests, require that certain code structures, such as branches or Boolean expressions, be exercised to a certain level of thoroughness. However, the degree to which fault finding was impacted by simple syntactic changes to the program is bothersome, especially given that such criteria are used to assess the testing effort for safety-critical software. These structural criteria specify that various structural parts of a program must be exercised, they do not, however, mandate that the effect of exercising that part must manifest itself at a subsequent *observable* point in the program; a frequent problem is that the effect of a corrupted variable gets masked out through its use in subsequent operations.

To address this problem, *observability* has been proposed as a desirable attribute of testing in both hardware [10] and software domains [11]. The notion of observability mandates that the effect of exercising a structural part must manifest itself at a subsequent *observable* point in the program. Specifically, we have proposed Observable MC/DC – a combination of traditional MC/DC with the notion of observability [11]. In practical terms, OMC/DC adds an additional path constraint to the coverage obligation. This constraint concretizes the idea that the *effect* of satisfying the MC/DC obligation must propagate to an observable output. The OMC/DC criterion defines propagation by specifying what constitutes *masking* and mandating a non-masking path from the point of exercising the code structure to some monitored output variable. This greatly increases the likelihood of faults triggered during testing to be observed as failures.

While OMC/DC offers a significant improvement over MC/DC with respect to fault finding and robustness to syntactic transformations [11, 12], it makes generating tests to satisfy the criterion more difficult; in particular, to give a corrupted state the opportunity to propagate to an output in a stateful system (common in the critical systems domain), a test case may need to execute for a large number of test steps. In our work, we have generated tests by formulating test obligations – properties that must be satisfied by the paths that are executed – and using model checkers to find the concrete tests by asserting that such obligations cannot be satisfied [9, 11]. The counterexamples generated are test cases – paths that satisfy those obligations. The path constraint added by observability for effect propagation makes this search harder and often prohibitively expensive even for relatively small systems.

We present an incremental test generation strategy [13] that addresses this scalability problem using *dynamic symbolic execution*, in which test inputs are generated incrementally by combining concrete and symbolic executions of the program under test [14, 15]. Starting with some concrete values for inputs to obtain a concrete execution path, path constraints are then generated for that execution path by treating (some of the) variables as symbolic. These constraints are then systematically modified and solved to force the program to take different (but feasible) execution paths. There has been a large volume of research on dynamic symbolic execution that demonstrates encouraging improvement over classical symbolic execution [16, 17, 18, 19, 20]. Efficiency is gained by modifying the path constraints along a known concrete path, which localizes the search for newer feasible paths. This reduces the high computational cost associated with symbolic search and, in practice, leads to better coverage of the program paths and higher likelihood of revealing faults.

In the test generation approach presented in this work, we adopt the dynamic symbolic execution idea in a novel approach to generate tests that, first, satisfy a state-based coverage criterion to exercise a particular part of the code (MC/DC) and, second, satisfy a dataflow criterion propagating the possibly corrupted state to a point where it is used by an observable output [13]. To this effect, we employ a tagging semantics [11] where a tag is assigned to each condition and the propagation of tags to outputs approximates observability. Instead of invoking a model checker to generate a complete test, which can be prohibitively expensive due to the test case length, the incremental test generation starts with concrete test inputs that satisfy an obligation and then invokes the model checker at each test step repeatedly to solve path conditions in an attempt to propagate tags through non-masking paths towards outputs. A test case satisfying an obligation incrementally grows and eventually tags associated with the condition of interest (the condition the MC/DC obligation exercised) reach output variables (to satisfy the observability constraint).

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 large numbers of test cases from models, the ability to use those for testing a particular realization of the system is hampered by two main problems: (1) translating the tests generated from the model – a model that, by definition, abstracts away some implementation specific details – into equivalent scenarios for the actual system, and (2) deriving an oracle – an arbiter of correctness – that can decide whether the actual system, as opposed to the model, 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 1.1 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 $f(out_i^s) = out_i^m$.

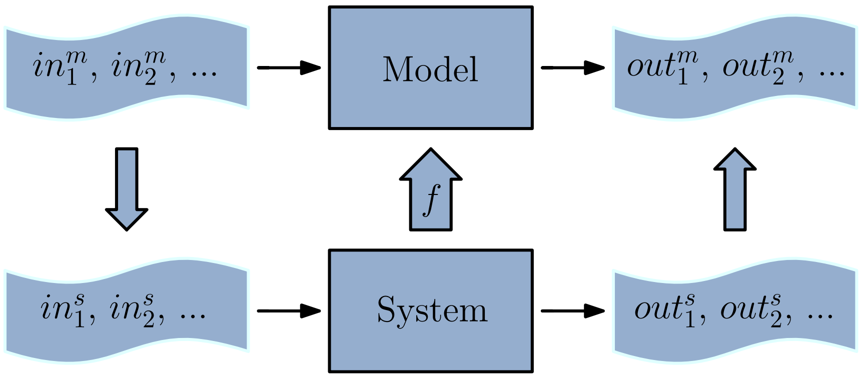


Figure 1.1: Concretizing model-based tests: A simplified view

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 nondeterministic 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 1.2 shows a better reflection of the typical situation.

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.

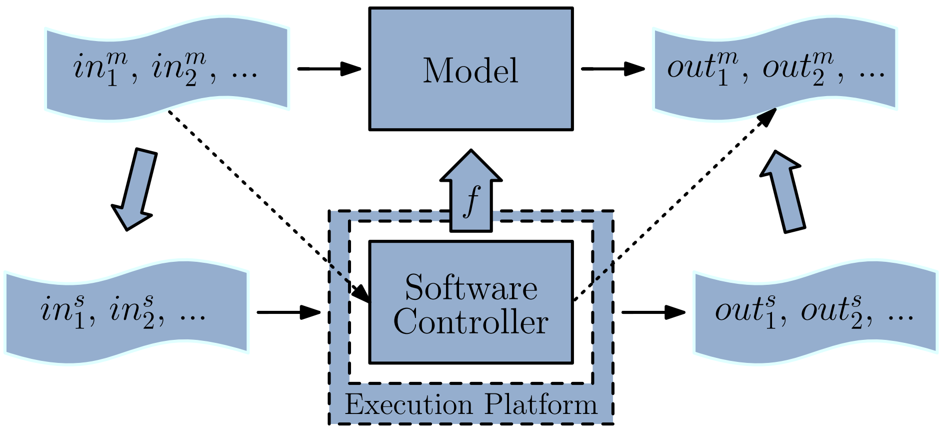


Figure 1.2: Concretizing model-based tests: A typical scenario

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 in Figure 1.2). 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; 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. Furthermore, the output produced by the system, even when it is behaving correctly, could be different from the output 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 [21] using timed automata [22] or UPPAAL [23] can potentially characterize system non-determinism accurately. These approaches, however, do not scale well, since the introduced non-determinism can increase the reachable state space exponentially. Oracle steering [24] 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.

We present 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 [25]. 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.

1.1 Objective and Contributions

Our **long-range goal** is to improve the applicability of model-based testing in practice. The majority of existing model-based testing approaches are concerned with generating test cases and demonstrating that the system under test (SUT) conforms to the model. In order for model-based testing to have the dramatic real-world impact, the **objective of this dissertation** is to address the challenges of effectively and efficiently using software models as the source for test data generation and translating these generated tests to equivalent scenarios for the execution platform. We have developed an automated testing framework that can serve as a foundation for testing complex software models and their platform-specific implementations.

The **intellectual merit** of the presented research lies in its new and novel approach to generating test cases that are effective at detecting faults and robust to program restructuring, and applying these model-based tests on the actual execution platform. While the topics of automated test generation and conformance testing have been well explored, to the best of our knowledge, little work has been done in terms of efficiently generating tests satisfying observability-based coverage criteria and adaptively reconciling abstraction differences between software models and system implementations. The results will be **significant** because of the potential improvements to the model-based testing process for real-time systems. Such systems are becoming more and more common, and the proposed research would advance the applicability of model-based testing in this domain. Developers of existing model-based testing tools may adopt our techniques in their tools and the adoption could result in better fault finding effectiveness and lower testing costs.

We have made the following **specific contributions** to the software testing literature in this dissertation:

1. **Developed an observability-based test generation approach.** We have formally defined a tagging semantics to generate observability constraints for monitoring and generating test cases. The notion of observability has significantly improved the effectiveness of fault finding and the robustness to syntactic transformations as compared to traditional structural coverage criteria. We further developed an incremental test generation approach using dynamic symbolic execution to allow scalable and efficient test generation satisfying observability-based coverage criteria.

2. **Developed an automated framework to reconcile abstraction differences.** We have formally defined timing at different system boundaries to allow effective and safe model-based test translation to enable the generated tests to be executed on the target platform. We have developed an automated framework that translates a sequence of abstract model behaviors to an *equivalent* sequence of concrete system behaviors, which would considerably advance the utility and practicality of model-based testing.

3. **Implemented our approaches in an open source framework.** This framework is able to efficiently generate tests satisfying observability-based coverage criteria,

and effectively translate model-based tests to match the level of abstraction of an execution platform.

4. **Empirically evaluated the use of our testing framework on industrial systems.** We performed empirical evaluations of the effectiveness, efficiency, and risks of our approaches as applied to industrial avionics and medical device systems. Specifically, we conducted experiments to evaluate the exact impact of the the notion of observability and the use of incremental test generation in terms of efficiency, test suite size, satisfied test obligations, and fault finding effectiveness. Besides, we examined the impact of using our automated framework to reconcile abstraction differences on platform-specific implementations in terms of the number of false positives and false negatives.

1.2 Structure of This Document

The remainder of this dissertation is organized as follows:

- Chapter 2 outlines the background of our approaches and surveys related work, including structural coverage analysis, dataflow programming languages, dynamic symbolic execution, and model-based testing.

- Chapter 3 presents our formal definition of the tagging semantics for observability and the incremental approach for test generation.

- Chapter 4 presents our formal definition of timing at different system boundaries and our automated framework to reconcile abstraction differences.

- Chapter 5 details our research questions and the design of experiments to evaluate our approaches.

- Chapter 6 discusses the results of our evaluation and their implications with respect to the effectiveness and efficiency of our approaches, comparisons with other potential solutions, and threats to validity in our empirical evaluations.

- Chapter 7 concludes the dissertation and overviews future work.

Chapter 2

Background and Related Work

Software testing research contains a large variety of areas that have been ongoing for decades. In this chapter, we will discuss the following four research areas that are closely related to our research objectives.

- Structural coverage analysis in Section 2.1, including code coverage criteria, observability-based coverage, information flow analysis, and program slicing.
- Dataflow programming languages in Section 2.2, which are commonly used in model-based development of reactive systems.
- Dynamic symbolic execution techniques for test data generation in Section 2.3.
- Model-based testing of platform-specific implementations in Section 2.4.

2.1 Structural Coverage Analysis

Structural coverage analysis provides a means to confirm that the code structure has been exercised by test procedures. The amount of code structure that has been executed can be measured by different criteria. Various structural coverage criteria have been defined and can be used to measure test suite adequacy as well as to generate test cases using automated tools.

Structural coverage criteria can be defined based on the data flow or control flow of a program. Control flow criteria measure the flow of control between statements and sequences of statements (e.g., branch coverage). Data flow criteria measure the flow of data between variable assignments and references to the variables (e.g., all DU pairs coverage).

We will first briefly discuss several commonly used coverage criteria specifically defined to evaluate Boolean expressions. We will then focus on the MC/DC criterion since it is used as an exit criterion when testing critical software in the avionics domain [7].

2.1.1 Code Coverage Criteria

A set of coverage criteria forming a subsumption hierarchy have been defined and widely used to evaluate test suite adequacy, including statement coverage, decision coverage, condition coverage, condition/decision coverage, multiple condition coverage, and modified condition/decision coverage (MC/DC). In our context, a *condition* is an atomic Boolean expression containing no Boolean operators (e.g., `and` or `or`); and a *decision* is a Boolean expression composed of conditions and zero or more Boolean operators [7]. A decision without a Boolean operator is a condition. Furthermore, if a condition appears more than once in a decision, each occurrence is a distinct condition.

Statement coverage requires that every executable statement in the program is executed at least once. Statement coverage is generally considered as a weak criterion since it is insensitive to control structures.

Decision coverage[1] requires two test cases to evaluate the decision to both *true* and $false$ outcomes. Decision coverage can ensure complete testing of control structures for simple decisions (i.e., decisions without Boolean operators). For complex decisions, such as (a `and` b), the test suite $\{(T, T), (F, T)\}$ will satisfy decision coverage, but the test suite cannot distinguish the decision (a `and` b) from the decision a (i.e., the effect of b is not tested).

Condition coverage requires two test cases to evaluate each condition in a decision to both *true* and $false$ outcomes, but does not require that the decision evaluates to both *true* and $false$ outcomes. In this case, for the decision (a `and` b), the test suite $\{(T, F), (F, T)\}$ will satisfy condition coverage, but the decision does not take on all

[1] Decision coverage is not a synonym for branch coverage. Decision coverage is concerned with all Boolean expressions, while branch coverage is only concerned with branch points such as *if* statements. Therefore, for example, the statement $c = (a$ `and` $b)$ contains a decision, but there is no branch point.

possible outcomes.

Condition/decision coverage combines the requirements for decision coverage and condition coverage. Therefore, a minimum of two test cases are required to evaluate each condition and the decision to both *true* and *false* outcomes. For the decision (a `and` b), the test suite $\{(T, T), (F, F)\}$ will satisfy condition/decision coverage, but the test suite cannot distinguish the decision (a `and` b) from any of the decisions a, b, and (a `or` b).

Multiple condition coverage requires that each combination of inputs to a decision is executed at least once. Therefore, multiple condition coverage requires exhaustive testing of the input combinations to a decision, and specifically, 2^n tests for a decision with n inputs[2] . Although multiple condition coverage would be the most desirable structural coverage measure, it is impractical in many cases because of the exponential growth in the number of required test cases.

2.1.2 Modified Condition/Decision Coverage

Modified condition/decision coverage (MC/DC) is a structural coverage criterion used in some safety-critical systems domains and developed as a compromise between the benefits of multiple condition coverage and the lower number of test cases required by condition/decision coverage [8]. In order to satisfy MC/DC, the following requirements have to be met:

- Each decision in the program has to take on all possible outcomes.
- Each condition in a decision has to take on all possible outcomes.
- Each condition in a decision has to be shown to independently affect the outcome of the decision.

The MC/DC criterion enhances condition/decision coverage by requiring that each condition has to be shown to independently affect the outcome of the decision. That is, the condition of interest cannot be masked out by other conditions in the decision.

[2] The number of inputs can be different from the number of conditions in a Boolean expression, because each occurrence of an input is a distinct condition. For example, the expression ((a `and` b) `or` (a `and` c)) has three inputs but four conditions. The maximum number of possible test cases is always 2^n, where n is the number of inputs rather than the number of conditions.

Masking is a concept that specific inputs in a logic expression can hide the effects of other inputs. For example, a $false$ input to an `and` operator masks all other inputs, and a $true$ input to an `or` operator masks all other inputs. The independence requirement ensures that the effect of each condition is tested relative to other conditions and changing a single condition must result in a change in the outcome of the decision. For example, for the condition a to independently affect the outcome of the decision (a `and` b), b has to be $true$, otherwise, (a `and` b) would evaluate to $false$ regardless of the value of a. Furthermore, MC/DC was developed as a practical alternative, which achieves many of the benefits of multiple condition coverage, while keeping the number of required test cases linear in terms of the number of inputs. Specifically, MC/DC requires a minimum of $(n+1)$ test cases for a decision with n inputs. For example, for the decision (a `and` b), the test suite $\{(T,T),(T,F),(F,T)\}$ will satisfy MC/DC.

In MC/DC, there are two different approaches[3] called *unique-cause* MC/DC and *masking* MC/DC [26]. For unique-cause MC/DC, in order to show the independent effect of a condition in a decision, the condition varies between $true$ and $false$ while all other conditions remain fixed. For masking MC/DC, in order to show the independent effect of a condition in a decision, it only requires that no other condition affects the decision and there is no requirement for other conditions to remain fixed. For example, given the decision ((a `and` b) `or` c), in order to show the independent effect of the condition c, there are three possible unique-cause MC/DC test suites: $\{(T,F,T),(T,F,F)\}$, $\{(F,T,T),(F,T,F)\}$, and $\{(F,F,T),(F,F,F)\}$. However, masking MC/DC only requires (a `and` b) to be $false$ and thus there are nine possible masking MC/DC test suites.

Generally, unique-cause MC/DC and masking MC/DC require the same number of test cases. However, as the above example shows, all test cases that satisfy unique-cause MC/DC will also satisfy masking MC/DC and masking MC/DC has more possible test suites per condition than unique-cause MC/DC. Having more possible test suites has the potential to reduce time and human effort to determine whether a given test suite can satisfy MC/DC [27]. On the other hand, unique-cause MC/DC cannot be applied to decisions with coupled conditions (e.g., the decision (a `and` b) `or` ((`not` a) `and` c) where

[3] The combination of unique-cause MC/DC and masking MC/DC may also be considered as the third approach.

a appears more than once), while masking MC/DC can be applied. In this dissertation, we use the masking form of MC/DC.

It has been shown that structural coverage criteria are sensitive to program structures [9]. Simple syntactic transformations can have a dramatic effect on the generated test suite as well as its fault finding effectiveness. Furthermore, while MC/DC ensures that a condition will not be masked out in a decision, it is still possible that the condition will ultimately be masked out within some sequence of statements in a program. Therefore, even if the faulty code is exercised, the corrupted program state may not be propagated to the output as observable failures, i.e., faults may still remain undetected. As an example, consider the following trivial program fragment:

```
expr = in1 or in2;      // statement1
out = expr and in3;   // statement2
```

Based on the definition of MC/DC, Table 2.1 shows two sample test suites that satisfy MC/DC on the program fragment.

`TestSuite1`	**(T, F, F)**	**(F, T, F)**	(F, F, T)	(T, T, T)
`TestSuite2`	(T, F, T)	(F, T, T)	(F, F, T)	(T, T, F)

Table 2.1: Sample test suites for (in1, in2, in3)

In `TestSuite1`, the test cases with $in3 = false$ (bold faced) contribute towards MC/DC of ($in1$ `or` $in2$) in statement1. Nevertheless, if our test oracle monitors the output variable *out*, the effect of $in1$ and $in2$ cannot be observed in the output since it will be masked out by $in3 = false$. Thus, `TestSuite1` satisfies MC/DC on the program fragment, but a fault on the first line will never propagate to the output. On the other hand, `TestSuite2` will also satisfy MC/DC on the program, but since $in3 = true$ in the first two test cases, faults in the first statement can propagate to the output.

The masking problem can be addressed by monitoring all internal state variables, but the use of such a strong test oracle is often cost-prohibitive (or outright infeasible) [28]. An alternative approach is to strengthen the coverage criterion to include a notion of *observability* of expressions in the variables monitored by the test oracle, which we will discuss in the following section.

2.1.3 Observability-based Coverage

Observability is a measure of how well internal states of a system can be inferred by knowledge of its external outputs. The observability of a program is the degree to which we can observe the program's internal states [29]. The observability property in a program can influence the construction of test oracles. If some variable is not observable, we cannot use it to determine the correctness of the system under testing. Gaudel discussed this issue and pointed out that certain parts of the program necessary to ascertain correctness are not always observable [30].

Observability has been studied in testing of hardware logic circuits. Observability-based code coverage metric (OCCOM) is a technique in which tags are attached to internal states in a circuit and the propagation of tags is used to predict the actual propagation of errors (i.e., corrupted state) [10, 31]. A variable is tagged when there is a possible change in the value of the variable due to an error. Both positive and negative tags are introduced on every assignment statement. The observability-based coverage can be used to determine whether erroneous effects that are triggered by the inputs can be observed at the outputs, i.e., whether a tag introduced at any particular location is propagated to the outputs.

Besides positive and negative tags, an unknown tag is also defined and introduced when the sign of a tag cannot be determined [31]. This happens when both positive and negative tags exist in the same assignment. Variables with unknown tags are not considered to carry an observable error. For example, in the assignment $c = a + b$, if a is tagged by a positive tag and b is tagged by a negative tag, then c will be tagged by an unknown tag, meaning that the effect of neither a nor b can be observed at c. This tag cancelation mechanism aims to eliminate optimistic inaccuracy (i.e., a variable that is actually not observable is considered to be observable). In the above example, if a is erroneously added by 1 and b is erroneously subtracted by 1, then the outcome of c does not change. However, this mechanism also introduces pessimistic inaccuracy (i.e., a variable that is actually observable is considered to be not observable). In the above example, if a is erroneously added by 2 and b is erroneously subtracted by 1, then the outcome of c is erroneously added by 1 due to the error at a. Thus, a should have been considered to be observable at c. The pessimistic inaccuracy in OCCOM may mislead the testing procedure because developers have to spend more unnecessary time to cover certain variables which actually have been covered.

Furthermore, the magnitude of an error determines whether the error will be propagated. For example, consider the following code fragment in Figure 2.1:

```
void f(int a, int b, int c){
  if(a + b > c){
    print("then branch");
  }
  else{
    print("else branch");
  }
}
```

Figure 2.1: Example code fragment for tag propagation

Suppose $a = 3$, $b = 4$, and $c = 5$, a positive error on a cannot be propagated to the output because the comparison decision provides the same outcome (i.e., $true$). A negative error on a may be propagated to the output. For example, if $a = 2$ (i.e., erroneously subtracted by 1), the error cannot be propagated; but if $a = 1$ (i.e., erroneously subtracted by 2), the error will be propagated because the outcome of the decision $(a + b > c)$ will change from $true$ to $false$. Therefore, the detection of an error in an assignment depends on the magnitude and sign of the error. The definition of OCCOM assumes that the error will be of the right magnitude to be propagated, but faults with lower opportunities to be observed may still be judged as observable in real situations and the observability coverage may be overestimated. Therefore, the optimistic inaccuracy in OCCOM may also mislead the testing procedure because developers will probably terminate testing after satisfying test adequacy criteria – OCCOM in this case.

Subsequent studies on OCCOM mainly focused on automated techniques that support test generation using observability measures. Various engines have been applied for test generation such as linear programming, automatic test pattern generator, and standard satisfiability. Specifically, Fallah et al. [32] proposed a constraint solving approach called Hybrid SAT, which is a combination of linear programming and standard satisfiability. In their approach, a SAT problem that corresponds to sensitization requirements on signal values and module-input module-output relationships for every module is solved. Sensitizing a path means that the value at the input to the path should affect the value at the output of the path. This approach produces an input assignment that satisfies the sensitization requirement and is consistent with the behavior of the logic circuit. Linear programming equations are written for data path modules and satisfiability clauses are generated for the control modules. Since they use Boolean clauses and linear arithmetic constraints to model the test generation problem, the resulting SAT problem is called Hybrid SAT. Although the original Hybrid SAT approach is based on path coverage, it can potentially be coupled with an observability metric by changing the specifics of the generated SAT program. Therefore, a function vector generation approach that combines Hybrid SAT and OCCOM was proposed and shown that it can generate test suites with high coverage [33].

The *testability* information for a program has been studied based on the Propagation, Infection, and Execution (PIE) analysis technique [34, 35]. The PIE analysis is designed to determine the probability of faults at each program location propagating to the output and thus similar to the notion of observability. The testability information for a program can be used to guide test case selection [36] and the construction of test oracles (i.e., assertions in their case) [37]. The PIE analysis suggests that more testing effort should be focused on the program location where it is less likely to reveal faults. However, empirical studies on the fault finding effectiveness of incorporating fault-exposure-potential estimates in generating test cases indicate that, although statistically significant, the improvements are quite small [38].

Regarding the optimistic inaccuracy in OCCOM, Jiang et al. improved OCCOM by defining a probabilistic observability measure using the concept similar to the propagation probability in the PIE approach without extending tag coverage [39]. This topology-based observability computation algorithm can quickly produce results and provide a closed lower bound of observability measures. Although this approach avoids optimistic inaccuracy and is faster than PIE, it may still be infeasible since it suffers from the limitation that it may result in too many internal observation signals.

Although OCCOM has the potential limitations in terms of optimistic inaccuracy, pessimistic inaccuracy, and scalability for practical applications, it has been well demonstrated that observability-based coverage criteria outperform traditional coverage criteria in terms of covering program execution paths and detecting faults [31].

2.1.4 Information Flow Analysis

Information flow analysis is an approach that marks and tracks certain data in a program at runtime. Dynamic taint analysis [40, 41] is an information flow analysis that has been used in security as well as software testing and debugging. Similar to observability, dynamic taint analysis is also a tag-based technique. Taint propagation can occur in explicit information flow (i.e., data dependencies) and implicit information flow (i.e., control dependencies). For example, consider the following code fragment in Figure 2.2:

```
1. void f(bool a, bool b){
2.   bool x = a && b;
3.   int y = 0;
4.   if (a || b){
5.     y = 1;
6.   }
7.   print(x);
8.   print(y);
9. }
```

Figure 2.2: Example code fragment for dynamic taint analysis

Suppose that we have tainted variables a and b with tag_a and tag_b, and the input for function f is (F, T). Then, both of the taint markings associated with variables x and y are $\{tag_a, tag_b\}$ at the end of the function. Specifically, x is tainted at line 2 by direct involvement of tainted variables a and b in the computation of x; and y is tainted at line 5 because tainted variables a and b affect the value of y indirectly, i.e., the assignment of y depends on the *true* evaluation of the *if* statement. In other words, x is tainted by data dependencies and y is tainted by control dependencies.

Taint analysis is defined over data/control dependencies and masking is generally not considered. Although the way in which taint marking are combined in an assignment may vary based on the application, the most common behavior is to union them [41]. This conservative behavior in security ensures that a variable is marked as long as there is a control/data dependency from an already tainted source, regardless whether the tainted source actually affects the variable. Thus, in terms of analyzing the propagation of a program's internal states, taint analysis may have far more optimistic inaccuracy than an observability-based approach.

Consider the example in Figure 2.2 with the same input (F, T) for function f. At line 2, the taint markings associated with x is $\{tag_a, tag_b\}$, but as long as a is $false$, x will be $false$ regardless of the value of b (i.e., b is masked by the $false$ value of a). Therefore, the observability markings associated with x should be $\{tag_a\}$ rather than $\{tag_a, tag_b\}$. Meanwhile, the taint markings associated with y is $\{tag_a, tag_b\}$, but as long as b is $true$, line 5 will execute and assign 1 to y. Therefore, the observability markings associated with y should be $\{tag_b\}$ rather than $\{tag_a, tag_b\}$.

2.1.5 Program Slicing

Program slicing is a technique that computes the set of statements, known as program slice, that can potentially influence the variables used in a given location of interest [42]. Program slicing is often used in debugging to localize errors more easily, but it can also be considered as an approach that computes the set of variables and statements that can potentially affect the test oracle, including output. Program slicing consists of static program slicing and dynamic program slicing [43, 44].

Static program slicing computes consecutive sets of indirectly relevant statements, according to data and control dependencies. Static program slicing is generally not applicable in practice because it takes all possible dependencies into consideration. Dynamic program slicing computes a set of statements that influence the variables used at a program point for a particular execution. This can identify all variables that contribute to a specific program point, including output. Compared with static program slicing, dynamic program slice is usually much smaller than static program slice, since it only consists of the statements that actually influence the variables of interest. Therefore, the result of dynamic program slicing may vary depending on the inputs. For example, consider the following code fragment in Figure 2.3:

Suppose the input for function f is (F, T) and coverable statements are (2, 3, 4, 5,

```
1. void f(bool a, bool b){
2.   bool x = a && b;
3.   int y = 0;
4.   if (a || b){
5.     y = 1;
6.   }
7.   print(x);
8. }
```

Figure 2.3: Example code fragment for program slicing

7), one typical dynamic program slice from the output contains statements (2, 7).

Dynamic program slicing and dynamic taint analysis are very similar, but they compute different kinds of information. Dynamic program slicing computes the subset of statements in a program that can affect particular variables, while dynamic taint analysis computes the set of statements in the program that can be affected by a given set of variables. Although dynamic program slicing can find all the variables that contribute to the location of interest, it can be inefficient to determine whether certain variables can be propagated to the output because dynamic program slicing is a backward analysis. Furthermore, similar to dynamic taint analysis, dynamic program slicing is also defined over data/control dependencies and does not consider masking.

Checked coverage is an approach built upon dynamic program slicing and defined to assess oracle quality [45]. Checked coverage attempts to ensure that the code structure under test can be observed by test oracles. Given a test suite, checked coverage computes dynamic program slices from test oracles. Specifically, this approach first identifies all statements that check the computation inside the test suite – termed *check statements* – which are essentially assertions used as test oracles. Then it traces the execution of a test suite and computes a dynamic slice from all assertions. This gives the set of statements that have a control or data dependency to at least one of the assertions. The checked coverage is then a percentage of all statements that contribute to the value of any assertion (i.e., are observable at that assertion) vs. the total number of statements covered by the test suite.

Checked coverage is very similar to the notion of observability except that markings are backward propagated from test oracles rather than (in observability) forward propagated from the location of interest to test oracles.

2.2 Dataflow Programming Languages

Dataflow languages, which assign values to a set of equations in response to periodic inputs, are popular in model-based development using tools such as Simulink [46] and SCADE [47]. The dataflow language we are using – Lustre – is a synchronous dataflow language for programming reactive systems [48]. It is typically used as an intermediate representation between behavioral models and source code. Lustre programs can be automatically generated from models in notations such as Simulink, and can be automatically translated further to implementations in languages such as C/C++ and also as input models to verification tools such as model checkers.

2.2.1 The Dataflow Language Lustre

A Lustre program consists of assignments to *combinatorial* and *delay* variables, and evaluates in cycles (i.e., computational *steps*). Combinatorial variables are used to update program state within one computational step. Delay variables are used to store program state (i.e., $\frac{1}{z}$ blocks in Simulink), such that an expression can refer to a variable's value from the *previous* step (i.e., values of delay variables).

During a cycle, variables are assigned values based on their defining equations – a combinatorial computation involving values at the current step for combinatorial variables and values from the previous step for delay variables. Within a computational step, Lustre does not impose a sequential order on evaluation of equations, but a partial order based on data dependencies. An equation can be evaluated as long as values of all variables it uses have been computed.

Suppose we have the following code fragment in Figure 2.4, in which $in1$, $in2$, and $in3$ are input variables, $v1$, $v2$, $v3$, and $v4$ are internal state variables, and out is an output variable. All variables are of type Boolean. Variable $v1$ at the initial step will have the value $false$ followed by (shown as an arrow), at each subsequent step, $in1$'s value from the previous step.

```
v1 = (false -> (pre in1));
v2 = (in2 and v1);
v3 = (false -> (pre v2));
v4 = (if in3 then v2 else v3);
out = (false -> (pre v4));
```

Figure 2.4: Sample Lustre code fragment

When testing such reactive systems, a test case typically contains multiple steps, each of which specifies values for input variables. Internal and output variables are then computed by the program as described above. Values stored in delay variables (i.e., $in1$, $v2$, and $v4$) will be used by other variables (i.e., $v1$, $v3$, and out, respectively) in the next computational step. Table 2.2 shows an example of a test case of 4 steps together with values of internal and output variables.

Step	**Input Vars. (in1, in2, in3)**	**Internal Vars. (v1, v2, v3, v4)**	**Output Vars. (out)**
1	(T, F, F)	(F, F, F, F)	(F)
2	(F, T, F)	(T, T, F, F)	(F)
3	(F, F, F)	(F, F, T, T)	(F)
4	(F, F, F)	(F, F, F, F)	(T)

Table 2.2: Sample Lustre program evaluation

2.2.2 Coverage Criteria for Dataflow Languages

Dataflow languages are designed to write synchronous programs and they describe how inputs are transformed into outputs rather than describe the control flow of the program. Thus, the coverage criteria discussed in Section 2.1 are not suitable for dataflow languages since they do not provide meaningful information. To deal with this problem, structural coverage criteria specifically for Lustre programs have been defined [49, 50, 51] together with an approach to generate test cases [52].

Structural coverage criteria for Lustre are based on *activation conditions* that are defined as the condition upon which a data flow is transferred from the input to the

$s =$ `AND` (a, b)	$AC(a, s) =$ (`not` a) `or` b; $AC(b, s) =$ (`not` b) `or` a
$s =$ `OR` (a, b)	$AC(a, s) = a$ `or` (`not` b); $AC(b, s) = b$ `or` (`not` a)
$s =$ `ITE` (c, a, b)	$AC(c, s) = true$; $AC(a, s) = c$; $AC(b, s) =$ `not` c

Table 2.3: Activation conditions for Boolean and conditional operators

output of a path. When the activation condition of a path is $true$, any change in input causes modification of the output within a finite number of steps [49]. Therefore, coverage criteria defined over Lustre implicitly address observability by examining variable propagation from the input to the output. Table 2.3 shows several examples of activation conditions for Boolean and conditional operators. Specifically, each expression $s =$ `op` (e) – which uses the input e, the operator `op`, and the output s – is paired with the activation condition $AC(e, s)$ for the path (e, s). Furthermore, for a given path p of length n such as $(e_1, e_2, ..., e_n)$, the activation condition $AC(p)$ is a conjunction of activation conditions associated with each unit path (e_i, e_{i+1}), where i is an integer and $i \in [1, n-1]$. When n is 1, $AC(p)$ is $true$.

Suppose P_n is the set of all paths in a Lustre program whose length is no more than n and $in(p)$ is the input of a path p, then the following three coverage criteria can be defined for Lustre [49]:

- Basic coverage criterion.
- Elementary conditions criterion.
- Multiple conditions criterion.

Basic coverage criterion requires that each path in P_n is activated at least once, or formally, $\forall p \in P_n$, $AC(p) = true$. Basic coverage criterion ensures that all the dependencies between inputs and outputs have been exercised at least once.

Elementary conditions criterion requires that each path in P_n is activated by both $true$ and $false$ input values (only Boolean variables are considered), or formally, $\forall p \in P_n$, $in(p)$ `and` $AC(p) = true$, (`not` $in(p)$) `and` $AC(p) = true$. Elementary conditions criterion is stronger than basic coverage criterion in that it also takes into consideration the effect of inputs on the outputs.

Multiple conditions criterion requires that each path in P_n is activated by both $true$ and $false$ values of each unit path along the path, or formally, $\forall p \in P_n$, $\forall e \in p$, e `and` $AC(p) = true$, (`not` e) `and` $AC(p) = true$. Multiple conditions criterion is the strongest criterion in that it checks whether the outputs of a path depend on all value combinations of each unit path along the path.

Coverage criteria for Lustre are similar to observability since they also check if specific inputs affect the outputs and measure variable propagations. The difference is that the approach measures the coverage of variable propagation on *all* possible paths, while observability checks whether each atomic condition in a Boolean expression affects the monitored variables and determines whether such a path exists. Furthermore, observability (as well as MC/DC) is stronger in terms of how a decision must be exercised – the masking problem in Boolean expressions is not fully addressed in coverage criteria for Lustre. For example, in the assignment $c = a$ `and` b, MC/DC requires that in order for a to independently affect c, b has to be $true$. However, even the strongest coverage criterion for Lustre – multiple conditions criterion – only requires that a is $false$ or b is $true$, then if both a and b are $false$, it can still satisfy multiple conditions criterion but cannot satisfy MC/DC for a.

Coverage criteria defined for Function Block Diagram (FBD) [53] are very similar to those defined for Lustre. Specifically, coverage criteria for FBD are based on a *d-path condition* that is similar to activation conditions in Lustre. Three coverage criteria have been defined for FBD with different levels of rigor to satisfy, which are similar to the above three coverage criteria for Lustre.

2.3 Dynamic Symbolic Execution

Symbolic execution is an approach that uses *symbolic* values, rather than concrete values, as test inputs and uses symbolic expressions to represent the values of program variables over the symbolic input values. Although the main idea behind symbolic execution was introduced decades ago [54, 55], it only recently received a lot of attention and became practically effective as a result of the advances in constraint solving techniques.

The classical symbolic execution is defined to execute all paths of a program. An

```
1. void f(int x, int y){
2.   z = y * 4;
3.   if (x == z){
4.     if (x > y + 20){
5.       print("error");
6.     }
7.   }
8. }
```

Figure 2.5: Symbolic execution code example

execution path can be considered as a sequence of Boolean values, where each value (i.e., *true* or *false*) represents whether the program takes the "then" or "else" of the corresponding branch. Therefore, the execution paths of a program can also be represented as an execution tree. For example, symbolic execution of the code fragment in Figure 2.5 can result in an execution tree shown in Figure 2.6.

The function f in Figure 2.5 has three execution paths – numbered from 1 to 3 in the execution tree in Figure 2.6. The three paths can be executed by running the concrete test suite $\{(x = 40, y = 10), (x = 4, y = 1), (x = 0, y = 1)\}$.

In order to generate a test suite that executes all feasible paths, symbolic execution maintains a symbolic state that maps program variables to symbolic expressions and a symbolic constraint that represents the path conditions to execute a certain path. During symbolic execution, both the symbolic state and the symbolic constraint are updated; and at the end of an execution path of the program, the symbolic constraint can be solved to generate concrete test input values.

For the code example in Figure 2.5, initially, the symbolic state is $\{x \mapsto x_0, y \mapsto y_0\}$, where x_0 and y_0 represent the symbolic values for x and y, respectively; and after executing line 2, the symbolic state is $\{x \mapsto x_0, y \mapsto y_0, z \mapsto y_0 * 4\}$. Furthermore, the path constraints at the end of the three execution paths will be the following:

1. $(x_0 = y_0 * 4)$ `and` $(x_0 > y_0 + 20)$
2. $(x_0 = y_0 * 4)$ `and` $(x_0 <= y_0 + 20)$
3. $(x_0 \neq y_0 * 4)$

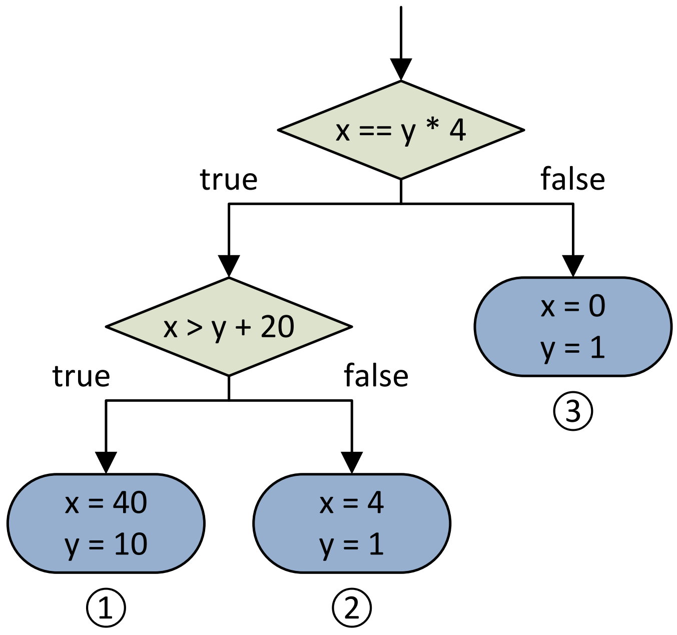


Figure 2.6: Symbolic execution tree

Although symbolic execution has been shown to be an effective approach for generating test cases with high code coverage and detecting faults, the path exploration problem is obvious – for a program with n branch points, there would be 2^n possible execution paths. At the end of symbolic execution, those symbolic constraints may not be efficiently solved to generate concrete test input values [56, 57].

Dynamic symbolic execution provides improved scalability over classical symbolic execution by combining concrete and symbolic executions. The notion of dynamic symbolic execution has been used in many applications such as DART [14], CUTE and jCUTE [15, 17], KLEE [18], PEX [19], Randoop [58], and Pathcrawler [16], as well as combined with fuzz testing to detect exploitable security issues [20]. By using concrete values, dynamic symbolic execution is able to simplify constraints, which helps it generate test inputs for execution paths that classical symbolic execution is incapable of analyzing.

Dynamic symbolic execution often starts with some given or random input, then symbolic constraints are generated in the execution of the concrete input. These constraints are then modified and solved to force the program to take different (but feasible) execution paths. If we take the program in Figure 2.5 as an example, initially, some given or random input is executed. For example, using the random input $\{x = 15, y = 5\}$, dynamic symbolic execution will execute the program both concretely and symbolically. The concrete execution will take the "else" branch of the first *if* statement; and the symbolic execution will generate the path constraint $(x_0 \neq y_0{*}4)$. That is, the execution path 3 in Figure 2.6 has been executed.

In order for the program to take a different branch (i.e., the "then" branch of the first *if* statement), dynamic symbolic execution negates the path constraint $(x_0 \neq y_0{*}4)$ and solves $(x_0 = y_0{*}4)$ to get the concrete test input $\{x = 4, y = 1\}$. Then dynamic symbolic execution repeats concrete and symbolic executions using this new input. Concretely, the program takes the "then" branch of the first *if* statement and the "else" branch of the second *if* statement; and symbolically, the generated path constraint is $(x_0 = y_0 * 4)$ `and` $(x_0 <= y_0 + 20)$. That is, the execution path 2 in Figure 2.6 has been executed.

Similarly, dynamic symbolic execution will attempt to force the program to take a path that has not been executed before. Thus, dynamic symbolic execution negates the path constraint $(x_0 <= y_0 + 20)$ and solves $(x_0 = y_0 * 4)$ `and` $(x_0 > y_0 + 20)$ to get the concrete test input $\{x = 40, y = 10\}$. Finally, all possible paths have been explored and dynamic symbolic execution terminates.

The above example has used a depth-first search strategy to explore execution paths. Applications may use a different heuristic as the search strategy. Furthermore, dynamic symbolic execution can often simply path constraints by replacing symbolic values with concrete values. For example, when solving the path constraint $(x_0 = y_0 * 4)$ for the program to take the "then" branch of the first *if* statement, dynamic symbolic execution could have replaced y_0 with its concrete value 5 and solved the path constraint $(x_0 = 20)$; and the generated concrete test input would be $\{x = 20, y = 5\}$.

Because dynamic symbolic execution can simply path constraints, it is able to explore execution paths and generate test inputs that classical symbolic execution is incapable of searching. However, this simplification also makes dynamic symbolic execution lose completeness as a result of concretizing certain variables. For example, if we have replaced y_0 with its concrete value 5 in the above example, then dynamic symbolic execution will not be able to explore the "then" branch of the second *if* statement, i.e., the path constraint $(x_0 = 20)$ `and` $(x_0 > 25)$ is infeasible. In practice, however, dynamic symbolic execution is more preferable than classical symbolic execution since it can explore program paths and generate test inputs that classical symbolic execution would have timed out and failed to accomplish.

Concatenating test cases – an approach similar to dynamic symbolic execution – has been investigated specifically for dataflow languages [59, 60]. In a common approach to generating tests satisfying a coverage criterion, each obligation is solved at the initial program state. In Hamon et al.'s approach [59], after creating a test case for an obligation, the final state (i.e., a state where every program variable has a concrete value) of the test case serves as the initial state for the next test case. That is, the next test case can be interpreted as an extension of the previous test case. Fraser et al. [60] further investigated how test case length affects fault finding effectiveness by concatenating test cases to create test suites with different test case length, while achieving similar coverage.

2.4 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 system under test (SUT) is considered a black box whose input/output sequences must conform to those described by the model [61, 62].

Of particular interest to us are the use – in testing – of models of reactive systems which are typically specified as (extended) finite state machines [63] or labeled transition systems [64]. Such models provide an operational view of the system that enables execution and 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 further 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 [61, 65].

These models by necessity abstract away implementation details, making 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.

Several existing modeling frameworks are able to account for non-deterministic behaviors induced during test execution on platform-specific implementations [66, 67, 21, 68], by incorporating more sophisticated model formats such as timed automata with real-valued clocks [22, 23]. For example, a timed automata could allow an event to happen within a bounded time window rather than at a specific time to allow more non-deterministic behaviors. These techniques can be effective if the software model can explicitly account for the same amount of non-determinism as the platform-specific implementation. Other approaches decouple non-deterministic behaviors from software models [69, 70, 24], and attempt to relax the passing criteria to allow certain non-deterministic behaviors induced from the execution platform. We will discuss them in the following sections.

2.4.1 UPPAAL-based Techniques

UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata [23]. Figure 2.7 and 2.8 show a simple UPPAAL model of an infusion pump system as a parallel composition of two automata. Specifically, Figure 2.7 describes the timed behavior of the infusion pump and Figure 2.8 describes its environment.

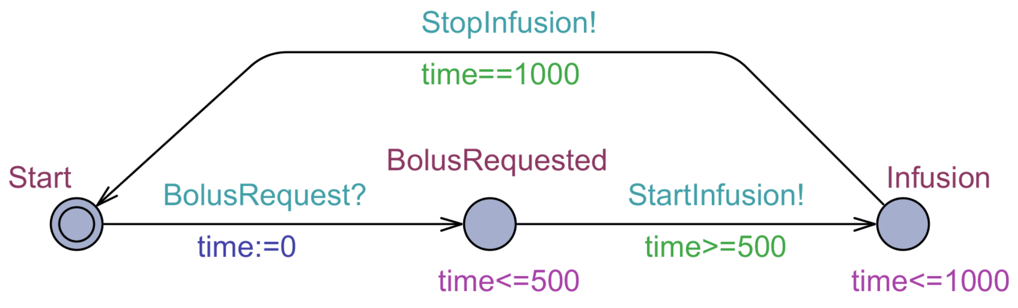


Figure 2.7: Infusion pump example with deterministic behavior

Figure 2.7 models the system using a clock variable (*time*), input synchronization (*BolusRequest*), and output synchronization (*StartInfusion* and *StopInfusion*).

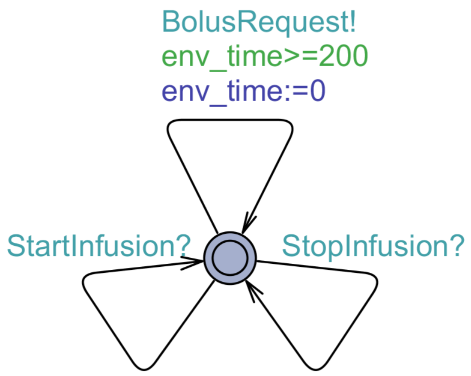


Figure 2.8: Environment model for the infusion pump example

Figure 2.8 models the environment using a clock variable (*env_time*) and the complementary synchronization with that in Figure 2.7. For example, the environment model in Figure 2.8 indicates that the bolus request event (represented by "BolusRequest!") can occur at any time as long as 200 ms have elapsed since the last clock reset. The system model in Figure 2.7 listens for the bolus request event (represented by "BolusRequest?").

The model in Figure 2.7 is deterministic. That is, when a patient requests a bolus, the bolus infusion starts exactly after 500 ms have elapsed since the last clock reset. This behavior is accomplished through (1) the clock invariant on the *BolusRequested* state forces the transition to *Infusion* state after no more than 500 ms; and (2) the clock guard on the transition between *BolusRequested* state and *Infusion* state allows a start infusion no sooner than 500 ms. This model, although can satisfy the system requirements, is not useful for testing purpose. When executing the system on the hardware platform, for example, the non-deterministic timing delay makes the infusion impossible to start at the specified time (i.e., 500 ms in this case).

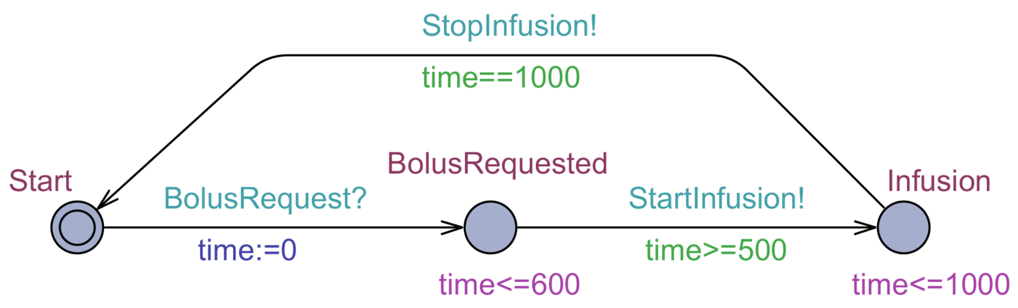


Figure 2.9: Infusion pump example with non-deterministic behavior

A more sophisticated UPPAAL model shown in Figure 2.9 can account for this mismatch by allowing non-deterministic behaviors. In this model, the clock invariant on the *BolusRequested* state forces the transition to *Infusion* state after no more than 600 ms. When combined with the clock guard between *BolusRequested* state and *Infusion* state, the model may start infusion anywhere between 500 and 600 ms after the patient requests a bolus. This model allows a short timing delay that would appear on the execution platform, while bounding the acceptable non-determinism.

Testing frameworks based on UPPAAL can use the similar models that allow a bounded amount of non-determinism to perform conformance testing between software models and platform-specific implementations [68]. Although these approaches can potentially account for non-determinism, under the assumption that the non-deterministic execution behaviors can be expected and planned, they have certain limitations.

Building a software model to account for planned non-determinism is different from building a software that also accounts for possible non-determinism induced from the execution platform. If the system is expected to demonstrate non-deterministic behaviors as detailed in the system requirements, the non-determinism can be planned and modeled. Using such a model to perform conformance testing can reveal defective behaviors in the platform-specific implementation, if tests fail to demonstrate conformance, assuming that no additional non-determinism is induced from the execution platform. Furthermore, introducing system non-determinism negates the original intent of using platform-independent models – models that abstract away platform-specific details to make the models amenable to various analyses. 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.

Software models often are more deterministic than implementations such that conformance testing can be sound, i.e., a test suite must render a failure only if the implementation is defective. Thus, there could be system behaviors that satisfy the requirements but do not match the behaviors of the model. These situations depend on the specific execution platform and such non-determinism is difficult to anticipate. On the other hand, software models are created at early development stages and can hardly be equipped with all the planned non-determinism that would appear in later development stages such as implementation and testing. Therefore, if such non-determinism is not planned or planned incorrectly, modifying the models at later stages would require a large amount of work. Each time the system is deployed and executed on a different hardware platform, the software model needs to be modified accordingly to account for the non-determinism.

To account for the problem of introducing non-determinism to the software models, several other approaches attempt to decouple platform-specific non-deterministic behaviors from the software models. Larsen et al. defined an online testing framework based on UPPAAL models for testing real-time systems [21, 68]. 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.

Kim et al. defined testing and verification frameworks based on the four-variable model that attempt to capture timing at different system boundaries [69, 70]. 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 an industrial scale case example system, creating a large number of test interactions would be impractical without automation.

2.4.2 Oracle Steering

Oracle steering – an approach that *carefully* steers the model to exhibit behavior that is closer to the observed behavior of the SUT – was recently proposed as an alternative approach to reducing false positives (i.e., reported mismatches do not attributable to real system defects) in model-based testing [24]. 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 is greatly reduced, but the number of false negatives (i.e., real system defects are not reported as mismatches) has also increased as a result of accepting possibly invalid non-deterministic system behaviors. 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, the approach is defined and built over discrete time, and may not be able to handle the richness of real-time behaviors.

The advantage of the oracle steering approach is that it relies on a set of rule-based constraints that can easily be revised over time, while the software model itself rarely needs to be modified directly. Developers may impose different constraints on what is considered to be the correct/incorrect behavior, depending on (evolved) system requirements or specific execution platform. The steering approach can apply the constraints to adjust the model during test execution. Since oracle steering decouples non-determinism from the models, it can easily support the reuse of models for multiple development purposes. Additionally, oracle steering does not require changes to the models or rely on a specific model format, thus it can be applied to extend many existing model-based testing approaches.

While model steering can be useful for addressing the oracle problem, there is no guarantee that the modified test inputs used to steer the model still retain the intent of the original test scenario. That is, it overlooks the possibility that the original test scenario may get changed in the steering process. For example, suppose that a test for a cardiac device controller is intended to stimulate a sequence of fast heartbeats to activate a specific therapy mode. If the threshold for *fastness* is not met for some heartbeats due to small timing variations on an execution of the test on the target platform, the specific therapy mode may not be activated at all and so the purpose of the test would be lost. However, the oracle model may be successfully steered to account for timing variations and the behavior of the system will be (rightly) deemed correct. 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.

Chapter 3

Observability-based Test Generation

In this chapter, we will first illustrate our formal definitions on the notion of *observability* [11], then we will describe how our incremental approach using *dynamic symbolic execution* can improve scalability and efficiency in generating tests satisfying observability-basd coverage criteria [13].

3.1 Observable MC/DC

The masking problem discussed in Section 2.1 can be addressed by strengthening structural coverage criteria to include the notion of observability of expressions in the variables monitored by the test oracle. To this effect, we have proposed a new test adequacy coverage criterion – observable modified condition/decision coverage (OMC/DC) [11]. Informally, OMC/DC establishes observability of decisions by requiring that the variable whose assignment contains a particular Boolean decision remains unmasked through a path to a variable monitored by the test oracle (commonly an output variable).

Observability is a measure of how well internal states of a system can be inferred, usually through the values of its external outputs. We state that an expression in a program is *observable* in a test case if we can modify its value, leaving the rest of the program intact, and observe changes in the output of the system. If we cannot find such a value, then the expression is *not observable* for that test case.

More formally, we can view a deterministic program P containing expression e as a transformer from inputs to outputs: $P : I \rightarrow O$. We write $P[v/e_n]$ for program P where the computed value for the n^{th} instance of expression e is replaced by value v. Note that this is not substitution but akin to mutation; we are replacing a *single* instance of expression e rather than all instances. We say e is observable in test t if $\exists v.P(t) \neq P[v/e_n](t)$. This idea can be straightforwardly lifted from test cases to test suites.

This formulation is a generalization of the semantic idea behind masking MC/DC [27], lifted from decisions to programs. In masking MC/DC, given decision D, for each condition c in D, we want a pair of test cases that ensure c is observable for both *true* and *false* values. Given test suite T, the MC/DC obligations are:

$$\begin{aligned}
&(\forall c_n \in Cond(D)\ .\\
&\qquad (\exists t \in T\ .\ (D(t) \neq D[true/c_n](t)))\ \wedge\\
&\qquad (\exists t \in T\ .\ (D(t) \neq D[false/c_n](t))))
\end{aligned}$$

where $Cond(D)$ is the set of all conditions in decision D.

Given this definition, one can directly lift MC/DC obligations to OMC/DC obligations by moving the observability obligation from the decision to the program output. Given test suite T, the OMC/DC obligations are:

$$\begin{aligned}
&(\forall c_n \in Cond(P)\ .\\
&\qquad (\exists t \in T\ .\ (P(t) \neq P[true/c_n](t)))\ \wedge\\
&\qquad (\exists t \in T\ .\ (P(t) \neq P[false/c_n](t))))
\end{aligned}$$

where $Cond(P)$ is the set of all conditions in program P.

3.1.1 Tagging Semantics

Unfortunately, the semantic definition for observability is unwieldy both for test generation and especially for test measurement. First, the analysis requires that two versions

$$\begin{aligned} E \quad &::= \quad \mathit{Val} \mid \mathit{Id} \mid E\ \texttt{op}\ E \mid \texttt{not}\ E \mid \\ &\qquad E\ \texttt{?}\ E\ \texttt{:}\ E \mid \texttt{tag}(E, T) \mid (\mathit{Val}, \mathit{TS}) \mid \texttt{addTags}(E, \mathit{TS}) \\ \mathit{Context} \quad &::= \quad \square \mid \mathit{Context}\ \texttt{op}\ E \mid E\ \texttt{op}\ \mathit{Context} \mid \texttt{not}\ \mathit{Context} \mid \\ &\qquad \mathit{Context}\ \texttt{?}\ E\ \texttt{:}\ E \mid \texttt{addTags}(\mathit{Context}, \mathit{TS}) \mid \\ &\qquad \langle \mathcal{K} : \mathit{Context}, \mathcal{E} : \mathit{Env}, \ldots \rangle \end{aligned}$$

lit	$n \Rightarrow (n, \emptyset)$
var	$\langle \mathcal{E} : \sigma \rangle[x] \Rightarrow \langle \mathcal{E} : \sigma \rangle[(\sigma\ x)] \quad \text{if } x \in \mathrm{dom}(\sigma)$
op	$(n_0, l_0) \oplus (n_1, l_1) \Rightarrow (n_0 \oplus n_1, l_0 \cup l_1)$
and$_1$	$(\texttt{tt}, l_0)\ \texttt{and}\ (\texttt{tt}, l_1) \Rightarrow (\texttt{tt}, l_0 \cup l_1)$
and$_2$	$(\texttt{tt}, l_0)\ \texttt{and}\ (\texttt{ff}, l_1) \Rightarrow (\texttt{ff}, l_1)$
and$_3$	$(\texttt{ff}, l_0)\ \texttt{and}\ \_ \Rightarrow (\texttt{ff}, l_0)$
or$_1$	$(\texttt{ff}, l_0)\ \texttt{or}\ (\texttt{ff}, l_1) \Rightarrow (\texttt{ff}, l_0 \cup l_1)$
or$_2$	$(\texttt{ff}, l_0)\ \texttt{or}\ (\texttt{tt}, l_1) \Rightarrow (\texttt{tt}, l_1)$
or$_3$	$(\texttt{tt}, l_0)\ \texttt{or}\ \_ \Rightarrow (\texttt{tt}, l_0)$
ite$_1$	$(\texttt{tt}, l_0)\ \texttt{?}\ e_t\ \texttt{:}\ e_e \Rightarrow \texttt{addTags}(e_t, (e_t =_v e_e)\ \texttt{?}\ \emptyset\ \texttt{:}\ l_0)$
ite$_2$	$(\texttt{ff}, l_0)\ \texttt{?}\ e_t\ \texttt{:}\ e_e \Rightarrow \texttt{addTags}(e_e, (e_t =_v e_e)\ \texttt{?}\ \emptyset\ \texttt{:}\ l_0)$
tag	$\texttt{tag}((v, l), t) \Rightarrow (v, l \cup \{(t, v)\})$
addt	$\texttt{addTags}((v, l_0), l_1) \Rightarrow (v, l_0 \cup l_1)$

Table 3.1: Expression syntax, context, and semantics

of the program run in parallel to check that the results match. Second, for test measurement, the test suite must be run separately for *each pair* of modified programs.

In order to define an observability constraint that more efficiently supports monitoring and test generation, we approximate observability using a tagging semantics. We assign each condition a tag and then track the observability of these tags through the execution of a program. If a tag reaches the output, we consider the obligation satisfied. More accurately, we track pairs: the first is the condition tag (uniquely assigned for each condition instance in the program syntax), and the second is the Boolean outcome of the condition. The level of coverage for a test suite can be assessed by examining how many of all possible pairs within the program have reached an output in some test.

We formally define the tagging semantics for a set of expressions (shown in Table 3.1) and a simple dataflow language (shown in Table 3.2). For presentation, we use a reduction semantics with evaluation contexts (RSEC) [71] which we machine checked for consistency using the K tool suite [72]. The rules operate over *configurations* that contain the syntax being evaluated ($\mathcal{K}$) and a set of labeled configuration parameters. To simplify presentation, elements of the configuration are not shown in rules if not used or modified. The rules operate by applying rewrites at positions in the syntax that are allowed by the *evaluation context* (the *Context* definition). A context is a program or fragment of a program with a *hole*, where the hole (represented by $\square$) defines a placeholder where a rewrite can occur. We assume appropriate definitions for maps including lookup ($\sigma\ x$) and update $\sigma[x \leftarrow v]$ operations, the empty map $\emptyset$, and lists with concatenation $x.y$ and cons $elem :: x$, operators. During rewriting, additional syntax may be introduced; we distinguish this syntax from user-level syntax by formatting it against a gray background.

Expressions yield (Val, TS) pairs (where TS is a set of tags) and are evaluated in a context containing environment $\mathcal{E}$ of type $Env = (id \rightarrow (Val \times TS))$. The expressions are standard except the `tag`(e, t) expression, which adds a tag to the set of tags associated with the expression e. For observability, it is assumed that each condition is wrapped in a `tag` expression. The Boolean operators `and` and `or` define masking: given a `and` b, the value of a only matters if b is *true*, so a's tags only propagate if b is *true* (and vice-versa); a `or` b is similar, the value of a only matters if b is $false$, so a's tags only propagate if b is $false$ (and vice-versa).

$$
\begin{array}{rcl}
EQ & ::= & Id = E \mid Id = \texttt{pre}(E) \\
Prog & ::= & (I, Env, \textbf{List}\ EQ) \\
Context & ::= & \ldots \mid Context; \textbf{List}\ EQ \mid Context :: \textbf{List}\ EQ \mid \\
 & & EQ :: Context \mid Id = Context \mid Id = \texttt{pre}(Context) \mid \\
 & & \langle \mathcal{K} : Context, \mathcal{I} : \textbf{List}\ Env, \mathcal{O} : \textbf{List}\ Env, \\
 & & \quad \mathcal{E} : Env, \mathcal{S} : Env) \rangle
\end{array}
$$

$$
\begin{array}{rl}
\textbf{comb} & \langle \mathcal{E} : \sigma \rangle\ eqs_0.((x = (n, l)) :: eqs_1) \Rightarrow \\
 & \langle \mathcal{E} : \sigma[x \leftarrow (n, l)] \rangle\ eqs_0.eqs_1 \\
\textbf{state} & \langle \mathcal{S} : \sigma \rangle\ eqs_0.((x = pre\ (n, l)) :: eqs_1) \Rightarrow \\
 & \langle \mathcal{S} : \sigma[x \leftarrow (n, l)] \rangle\ eqs_0.eqs_1 \\
\textbf{write} & \langle \mathcal{O} : \kappa, \mathcal{E} : c \rangle\ \text{nil}; eqs \Rightarrow \langle \mathcal{O} : \kappa.[c], \mathcal{E} : c \rangle\ eqs \\
\textbf{cycle} & \langle \mathcal{I} : \sigma_i :: \iota, \mathcal{E} : \_, \mathcal{S} : \sigma_l \rangle\ eqs \Rightarrow \\
 & \langle \mathcal{I} : \iota, \mathcal{E} : (\sigma_i \cup \sigma_l), \mathcal{S} : \emptyset \rangle\ eqs; eqs \\
\textbf{prog} & (i, s, eqs) \Rightarrow \langle \mathcal{I} : i, \mathcal{O} : \text{nil}, \mathcal{S} : s, \mathcal{E} : \emptyset, \mathcal{K} : eqs \rangle
\end{array}
$$

Table 3.2: Dataflow program syntax, context, and semantics

In this dissertation, we have extended the tagging semantics defined in our original work [11] by removing the optimistic inaccuracy in *if-then-else* expressions [13]. Specifically, the original tagging semantics has optimistic inaccuracy in the following code fragment:

```
if (c) then out := 0 else out := 0 ;
```

where the condition c is reported to be observable, but c cannot affect the outcome of this code fragment. The extended tagging semantics is shown in Table 3.1 as $\mathbf{ite}_1$ and $\mathbf{ite}_2$ rules, where $=_v$ represents value equality of two expressions.

We have informally described dataflow languages in Section 2.2. Formally, our dataflow language consists of assignments to combinatorial and state variables, and the semantics are defined over lists (traces) of input variable values. The expression configuration is extended to contain an input trace I, output trace O, and state environments S. Evaluation proceeds by cycles: at the beginning of a cycle, the **cycle** rule constructs the initial evaluation environment. During a cycle, variable values are recorded using the **comb** and **state** rules. Note that the context does not force an ordering on evaluation of equations; instead, an equation can evaluate as soon as all variables it uses have been stored in the environment. When all equations have been computed, the **write** rule appends the environment to the output list. The **prog** rule, given an input list, an initial state environment, and a list of equations, initializes the configuration for the **cycle** rule. Coverage can be determined by examining the tags stored in the output environment list.

3.1.2 Comparison with Existing Work

The notion of observability has been studied in testing of hardware logic circuits [10]. Observability-based code coverage metric (OCCOM) is a criterion in which tags are attached to internal program states and the propagation of tags is used to predict the actual propagation of errors [31]. A variable is tagged when there is a change in the value of the variable due to an error. The observability-based coverage can be used to determine whether erroneous effects that are triggered by the inputs can be observed at the outputs.

The key differences between our definitions of observability and OCCOM are twofold: (1) Our approach investigates variable value propagation, while OCCOM investigates fault propagation; and (2) OCCOM has pessimistic inaccuracy because of tag cancelation. When both positive and negative tags exist in the same assignment (e.g., different tags in an `ADDER` or the same tags in a `COMPARATOR` cancel each other out), no tag is assigned [10] or an unknown tag is used [31]. Variables without tags or with unknown tags are not considered to carry an observable error. In our approach, since we do not make a distinction between positive and negative tags, we do not have tag cancelation or the corresponding pessimistic inaccuracy. Extended work [32] may fix pessimistic inaccuracy by producing test vectors with specific values, but is highly infeasible.

Since our formal definition of the tagging semantics is complete (and sound), our approach is guaranteed to find a non-masking path if it exists. In practice, however, the solving process may terminate before exploring all feasible paths since the time budget is always finite. Although it has been demonstrated that observability can lead to better fault finding effectiveness and robustness to program structures [31, 11], they often suffer from scalability problems. This is not surprising – with the notion of observability, the generated test cases are much longer than the ones generated using traditional approaches, and finding such non-masking paths is time consuming and even infeasible on systems of industrial scale. Therefore, an efficient test generation approach is also needed to make the notion of observability useful, which we will discuss in the next section.

3.2 Incremental Test Generation

In the traditional counterexample-based test generation, test obligations are first formulated and instrumented in the original program. Test obligations represent path constraints for the program to take certain paths (e.g., satisfying a coverage criterion). Trap properties are simply negations of test obligations that must be fulfilled. Asserting these properties on the program to a model checker leads to the generation of counterexamples that can be seen as test cases satisfying the obligations. For the purpose of generating tests satisfying observability-based coverage criteria, the path constraints describe *non-masking paths* starting from the initial program state to a state exercising the condition of interest in a specific way and then to a program state where the effect is observable at some output variable. This kind of generation can be expensive because the model checker has to search through a complete non-masking path which may involve many computational steps.

As we have introduced in Section 2.3, dynamic symbolic execution provides improved scalability over classical symbolic execution by combining concrete and symbolic executions. By using concrete values, dynamic symbolic execution is able to simplify constraints, which helps it generate test inputs for execution paths that classical symbolic execution is incapable of analyzing.

The notion of observability can be captured naturally using dynamic symbolic execution. That is, in order to propagate a tag_c from a condition c to an output out, which traverses a sequence of variables $(c, v_1, v_2, ..., v_n, out)$, we can propagate tag_c to any intermediate variable v_i and from v_i further propagate tag_c along the path. This process terminates when tag_c reaches out or there are no feasible paths.

Therefore, the incremental approach invokes the model checker multiple times; and it forms and solves path constraints *locally* based on the existing program state resulting from executing a concrete input. Specifically, at the initial step, our approach is similar to traditional counterexample-based test generation. Once we have a concrete input, it is immediately executed on the current program state and then concatenated with the existing test. If tags – which are associated with conditions and used to approximate observability – are propagated to outputs, then we have a complete test; otherwise, we record the set of variables that these tags can propagate to. In the next step, we start generating tests based on the program state and recorded set of variables. To enable test generation from a specific program state, the original trap property is combined with additional constraints that imply a desired initial state, which is essentially the program state at the end of the execution of the test case constructed thus far.

We will first describe constraint generation for non-masking paths using our tagging semantics defined in Section 3.1, then we will introduce the incremental test generation procedure with a motivating example.

3.2.1 Non-masking Path Constraints

The test obligations generated to satisfy observability will differ depending on the set of monitored variables. We define *observation points* as the set of variables where (internal) program state can be observed, i.e., an effect propagated to any of the observation points is *observable*. In an observability-based coverage criterion, observation points are usually the set of output variables – *output observation points*. In our incremental test generation approach, we also define *delayed observation points* – an effect propagated to delay variables is stored and may be further propagated. Therefore, when a tag is propagated to some output observation point, we have a complete test. Alternatively, when a tag is propagated to some delayed observation point, it will be further propagated in future steps.

A variable is *observable* if it is not masked along some computation path (as described in the tagging semantics) to an observation point. We call the path a *non-masking path* and the constraints for the program to take the path as *non-masking path constraints*. An *immediate* non-masking path is a dataflow path from a variable to an observation point that is entirely within one computational step (i.e., no delays) where the value of the variable is not masked. Immediate non-masking paths can be defined inductively by examining define-use relationships among variables. Although many variables can be immediately observed, often, the effect of a variable on an output can only be observed after several steps. In each of these intermediate steps, its tag is stored in a delay variable, until it eventually propagates to an output. We call this a *delayed* non-masking path. This situation can be broken into immediate observations: the first from a variable to a delay, the next from the delay to another delay, etc., until an output is reached. We track these relationships by instrumenting the original program with additional path constraint variables.

For the same Lustre code fragment from Section 2.2 in the following:

```
v1 = (false -> (pre in1));
v2 = (in2 and v1);
v3 = (false -> (pre v2));
v4 = (if in3 then v2 else v3);
out = (false -> (pre v4));
```

Suppose y is one of the variables that uses x, then tags associated with x can be propagated to y if x is not masked in the definition of y (i.e., x affects y). We use x_affect_y to represent the path constraint that x is not masked in the definition of y (i.e., tags associated with x can be propagated to y). The following *affect* relations can be generated for the above Lustre code example:

```
in1_affect_v1 = true;
in2_affect_v2 = v1;
v1_affect_v2 = in2;
v2_affect_v3 = true;
in3_affect_v4 = (v2 <> v3);
v2_affect_v4 = in3;
v3_affect_v4 = (not in3);
v4_affect_out = true;
```

Note that the path constraints differ from our previous work [11]: when $v2 = v3$, $in3$ does not play any role in determining the value of $v4$, creating optimistic inaccuracy. In our current approach, we have removed this inaccuracy by strengthening the path constraints for tag propagation in such cases. In the above example, to propagate tags from $in3$ to $v4$, we also require the inequality of $v2$ and $v3$. More precisely, we require the value inequality of expressions from two branches.

Furthermore, for each observation point (i.e., output and delay variables), we backward track from the observation point to the variables that it has control/data dependencies, i.e., a given variable can be observed at the observation point through a *sequential* path. In the above example, there are three observation point variables (i.e., $v1$, $v3$, and out) and the following *observed_at* relations can be generated for the above Lustre code example:

```
in1_observed_at_v1 = in1_affect_v1;

in2_observed_at_v3 = (in2_affect_v2 and v2_observed_at_v3);
v1_observed_at_v3 = (v1_affect_v2 and v2_observed_at_v3);
v2_observed_at_v3 = v2_affect_v3;
```

```
in2_observed_at_out = (in2_affect_v2 and v2_observed_at_out);
v1_observed_at_out = (v1_affect_v2 and v2_observed_at_out);
in3_observed_at_out = (in3_affect_v4 and v4_observed_at_out);
v2_observed_at_out = (v2_affect_v4 and v4_observed_at_out);
v3_observed_at_out = (v3_affect_v4 and v4_observed_at_out);
v4_observed_at_out = v4_affect_out;
```

Note that each sequential path is within one computational step. For example, variables $in2$, $v1$, and $v2$ may be observed at the delay variable $v3$ within one step, but the variable $in1$ has to go through at least two steps to be propagated to $v3$.

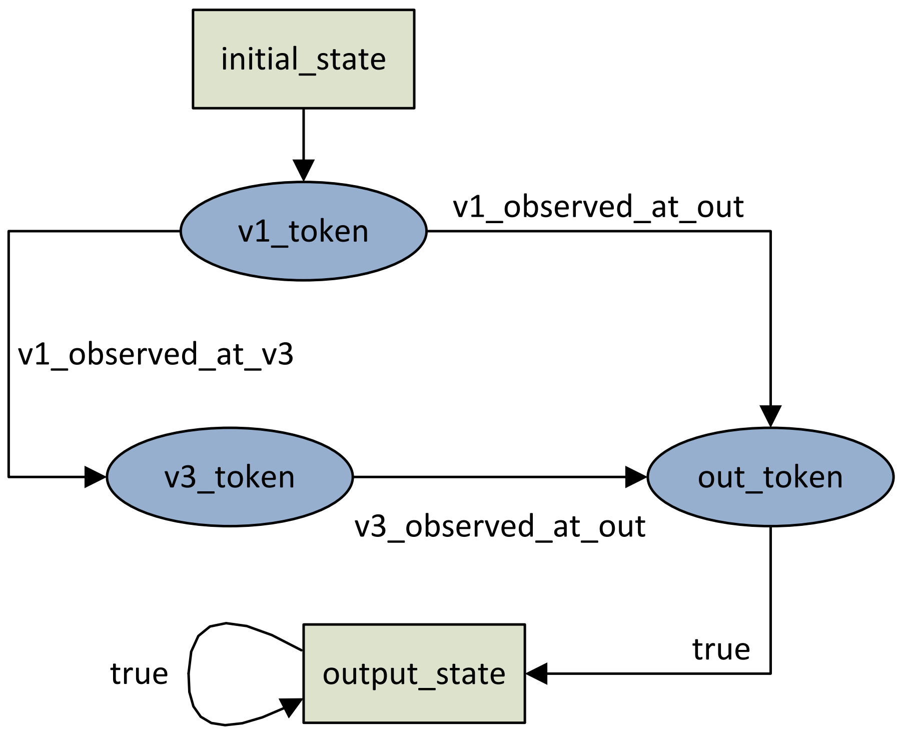


Figure 3.1: Token transition for the Lustre code example

We now have a mechanism that defines path constraints from a condition to an observation point (output or delay variables) as well as from a delay variable to an observation point (output or delay variables). What is necessary is some means to knit these paths together to define a complete path through (possibly) several delay variables to an output. We accomplish this using a *token* variable that describes the current delay location. Figure 3.1 shows token transition for the above Lustre code example.

Once the token is initialized to a delay variable, it can nondeterministically move to any other delay location. In the above Lustre code example, for example, $v1$ is sequentially used by $v3$ and thus the token can move from $v1$ to $v3$ as long as the path constraint $v1\_observed\_at\_v3$ can be satisfied. When a token moves to a special *output_state*, the variable of interest is observable and the token stays there. If the token cannot move, because it is not observable at another delay or the output, the token moves to an *error_state* (not shown in Figure 3.1).

3.2.2 Test Generation Procedure

Figure 3.2 shows an overview of the incremental test generation procedure satisfying observability-based coverage criteria.

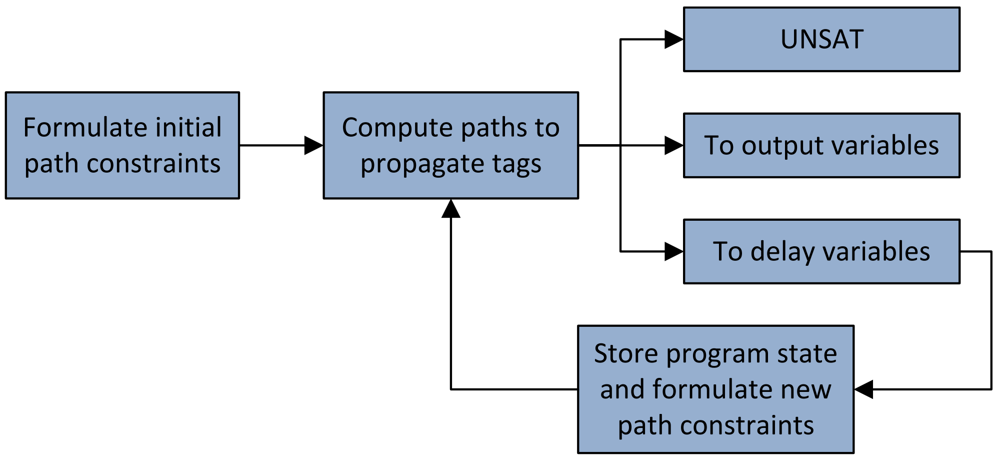

Formulate initial path constraints
Compute paths to propagate tags
UNSAT
To output variables
To delay variables
Store program state and formulate new path constraints

Figure 3.2: Incremental test generation procedure

For each condition c, our approach will execute one or more iterations and either produce a test case that can propagate the effect of c (i.e., tag_c) to some output observation points, or return an *UNSAT* indicating that such a test case cannot be found. During each iteration, our approach first attempts to generate a test that can propagate

tag_c to output observation points. If such a test can be found, tag_c has been propagated to the outputs. The newly generated test is concatenated with existing ones and we have a complete test. Otherwise, our approach attempts to generate a test that can propagate tag_c to delayed observation points. If such a test can be found, tag_c is propagated to delayed observation points and will be further propagated in the next iteration. Otherwise, there is no feasible path that can propagate tag_c further and thus $UNSAT$ is returned.

To be more specific, during each iteration: (1) existing tests are first executed from the current program state to obtain an updated program state, except that at the initial iteration, there is no existing tests and we start from the initial program state; (2) path constraints are formulated based on the tagging semantics and locations where tags have propagated; (3) path constraints are solved by a model checker to generate tests and propagate tags towards observation points.

Algorithm 1 shows the algorithm of our incremental test generation approach. The MAIN procedure takes a condition c and returns a test case that can propagate the effect of c to some output observation points, or *UNSAT*. The MAIN procedure first assigns the condition c as the current location of tag_c (line 9). During each iteration of the *while* loop, the algorithm first attempts to generate a test that can propagate tag_c to unvisited output observation points (line 11 – 14). If SAT, the newly generated test step is concatenated with existing test case and we have a complete test. Otherwise, the algorithm attempts to generate a test that can propagate tag_c to unvisited delayed observation points (line 15 – 17). If *UNSAT*, there is no feasible path that can propagate tag_c further and thus $UNSAT$ is returned. Otherwise, tag_c is propagated to delayed observation points and will be further propagated in the next iteration.

The GENERATE procedure takes the set of unvisited observation points (either output or delayed observation points) and returns a test that can propagate tag_c to any unvisited observation point, or *UNSAT*. Each non-masking path constraint is formulated as introduced above. Then pc is constructed as a disjunction of non-masking path constraints from the current location of tag_c to any observation points (line 24 – 27). The disjunction ensures that the tag can be propagated when *any* of the non-masking path constraint can be satisfied. Finally, the program *state* is conjoined such that the generation starts from an existing state (line 28).

Algorithm 1 Incremental Test Generation

```
1: $\mathcal{O} \leftarrow$ output observation points
2: $\mathcal{D} \leftarrow$ delayed observation points
3: $state \leftarrow \emptyset$                                  ▷ The current program state
4: $location \leftarrow \emptyset$          ▷ The observation point that $tag$ has propagated to
5: $visited \leftarrow \emptyset$                    ▷ The set of visited observation points
6: $testcase \leftarrow \emptyset$                ▷ The incrementally generated test case
7:
8: procedure MAIN($c$)
9:     $location \leftarrow c$
10:    while $true$ do
11:        $teststep \leftarrow$ GENERATE($\mathcal{O} \setminus visited$)
12:        if $teststep = SAT$ then
13:            $testcase \leftarrow testcase \mathrel{++} teststep$          ▷ Concatenate
14:            return $testcase$
15:        $teststep \leftarrow$ GENERATE($\mathcal{D} \setminus visited$)
16:        if $teststep = UNSAT$ then
17:            return $UNSAT$
18:        $state \leftarrow$ EXECUTE($state$, $teststep$)             ▷ Ternary simulation
19:        $location \leftarrow token$ value in $state$
20:        $visited \leftarrow visited \cup \{location\}$
21:        $testcase \leftarrow testcase \mathrel{++} teststep$              ▷ Concatenate
22:
23: procedure GENERATE($unvisited$)
24:    $pc \leftarrow false$                                          ▷ Path constraint
25:    for each $var$ in $unvisited$ do
26:        if there exists a sequential path from $location$ to $var$ then
27:            $pc \leftarrow pc \lor (location\_observed\_at\_var \land token = var\_token)$
28:    $pc \leftarrow (pc) \land state$
29:    return SOLVE($pc$)
```

Each test generated in our approach may contain *uninitialized* input variables because only variables that are necessary to achieve test obligations are concretized. Therefore, when the *partial* test is executed on the program, the resulting program state may represent a larger reachable state space and thus increase the likelihood that future propagation is feasible. In Algorithm 1, when a tag can only be propagated to delayed observation points, the newly generated test is first executed from the current program state to obtain an updated program state (line 18). The Lustre language, however, does not support specifying ternary logic. We have implemented a ternary simulator to compute program states. Table 3.3 and 3.4 show ternary evaluations for `AND` and `OR` operators. Eventually, *location* will be updated to the *token* value in the current program state and added to the set of visited observation points, and the new test step is concatenated with existing test case (line 19 – 21).

	TRUE	**FALSE**	**UNKNOWN**
TRUE	true	false	unknown
FALSE	false	false	false
UNKNOWN	unknown	false	unknown

Table 3.3: Ternary evaluation for AND

	TRUE	**FALSE**	**UNKNOWN**
TRUE	true	true	true
FALSE	true	false	unknown
UNKNOWN	true	unknown	unknown

Table 3.4: Ternary evaluation for OR

Essentially, in an observability-based coverage for dataflow languages, tags are always located at delay variables, unless they are propagated to output variables. In the incremental test generation approach, however, it is possible that a tag is repeatedly propagated to the same set of delay variables, forming a cyclic propagation. Cyclic propagation is a result of concretizing program state and generating new tests locally. From a specific set of program states, test generation (and thus tag propagation) is deterministic. The locally optimal path found by the model checker may be in a subspace that will repeatedly propagate the tag inside the subspace – the particular path chosen need not necessarily be the best candidate for eventually reaching an output – making test generation not terminate.

In our present approach, we favor *incrementality.* To account for the problem of cyclic propagation, we include a simple heuristic (i.e., recording all visited variables in *visited* and not propagating *tag* to them again) that avoids repeatedly visiting the same delayed observation point. This heuristic forces the test case to take a different path – which intuitively would increase fault finding effectiveness – and guarantees that the generation process will terminate. Otherwise, as in the traditional generation, a model checker typically generates the shortest possible path that satisfies an obligation. This may not be desirable from a testing perspective: for it may lead to fewer program states being visited which can negatively impact fault finding [12]. Given all the benefits, however, the heuristic may potentially miss some feasible test cases that necessarily require passing through cyclic propagation to reach some observable output. These test cases could have been found (if feasible) by attempting to generate a complete test case to satisfy the test obligation.

3.2.3 Motivating Example

For the same Lustre code fragment from Section 2.2 in the following:

```
v1 = (false -> (pre in1));
v2 = (in2 and v1);
v3 = (false -> (pre v2));
v4 = (if in3 then v2 else v3);
out = (false -> (pre v4));
```

Suppose our goal is to propagate the effect of $in2$ with $true$ value not only to $v2$ (as MC/DC requires, the condition $in2$ has to evaluate to both $true$ and $false$ values, and be shown to independently affect the outcome of the decision), but also to out (as observability requires, the effect of $in2$ has to propagate to the output). In the following illustration, we will only show feasible path constraints and generated tests, as well as the values of global variables in Algorithm 1 before each iteration. To simplify the representation, the program state is not shown.

Iteration 1

At the first iteration, the path constraint is a conjunction of an MC/DC obligation over a single atomic condition plus a path constraint representing the variable's observability at one of the observation points. Propagating tag_{in2_true} to output observation points is infeasible, so it will be propagated to the delay variable $v3$ through $v2$ as shown in the following.

$$\begin{aligned}
location &= in2\\
visited &= \emptyset\\
testcase &= \emptyset\\
pc &= (in2 \text{ and } v1) \text{ and}\\
&\quad ((v2\_observed\_at\_v3 \text{ and } (token = v3\_token))\\
&\quad \text{or } (v2\_observed\_at\_out \text{ and } (token = out\_token)))
\end{aligned}$$

The variable $v2$ is used in two equations and therefore has two non-masking paths to an observation point: one to $v3$ and the other to out through $v4$. After the first iteration, we have generated the test case {(T, -, -), (-, T, -)} and propagated tag_{in2_true} from $in2$ to $v2$, which in the next step propagates to the delay variable $v3$.

Iteration 2

At the second iteration, tag_{in2_true} automatically propagates from $v2$ to $v3$, since the delay variable $v3$ uses $v2$. Propagating tag_{in2_true} to output observation points is still infeasible, so it will be propagated to the delay variable out through $v4$ as shown in the following. Note that since $v3$ has been visited, it is shown in the *visited* set of observation points.

$$\begin{aligned}
location &= v3\\
visited &= \{v3\}\\
testcase &= \{(\text{T}, \text{-}, \text{-}), (\text{-}, \text{T}, \text{-})\}\\
pc &= v3\_observed\_at\_out \text{ and } (token = out\_token)
\end{aligned}$$

After the second iteration, we have generated the test case {(T, -, -), (-, T, -), (-, -, F)} and propagated tag_{in2_true} from $v3$ to $v4$, which in the next step propagates to the delay variable *out*, which is also an output variable.

Iteration 3

At the third iteration, tag_{in2_true} automatically propagates from $v4$ to *out*, since the delay variable *out* uses $v4$. Propagating tag_{in2_true} to output observation points is feasible, so a complete test is generated and the incremental test generation algorithm terminates. Note that since both $v3$ and *out* have been visited, they are shown in the *visited* set of observation points.

$$\begin{aligned} location &= out \\ visited &= \{v3,\, out\} \\ testcase &= \{(\text{T, -, -}),\, (\text{-, T, -}),\, (\text{-, -, F})\} \\ pc &= true \end{aligned}$$

After the third iteration, we have generated the test case {(T, -, -), (-, T, -), (-, -, F), (-, -, -)} and propagated tag_{in2_true} to the output variable *out*. Table 3.5 shows a summary of the generated test case as well as the locations of tag_{in2_true} at each step. Note that this test case still contains uninitialized variables but is complete enough to satisfy our test obligations. In practice, we may concretize those uninitialized variables with default or random values. If we concretize all uninitialized variables with their default values (e.g., $false$ for Boolean and 0 for integers), we would get the same test case shown in Table 2.2.

	Locations of tag_{in2_true}	**in1**	**in2**	**in3**
Step 1	$in2$	$true$	–	–
Step 2	$v2$	–	$true$	–
Step 3	$v4$	–	–	$false$
Step 4	out	–	–	–

Table 3.5: Test case generated using incremental approach

Chapter 4

Platform-specific Test Execution

Our incremental test generation approach using dynamic symbolic execution allows scalable and efficient test generation satisfying observability-based coverage criteria. Our next objective is to allow effective and safe model-based test translation to enable the generated tests to be executed on the target platform [25].

4.1 The Four-Variable Model

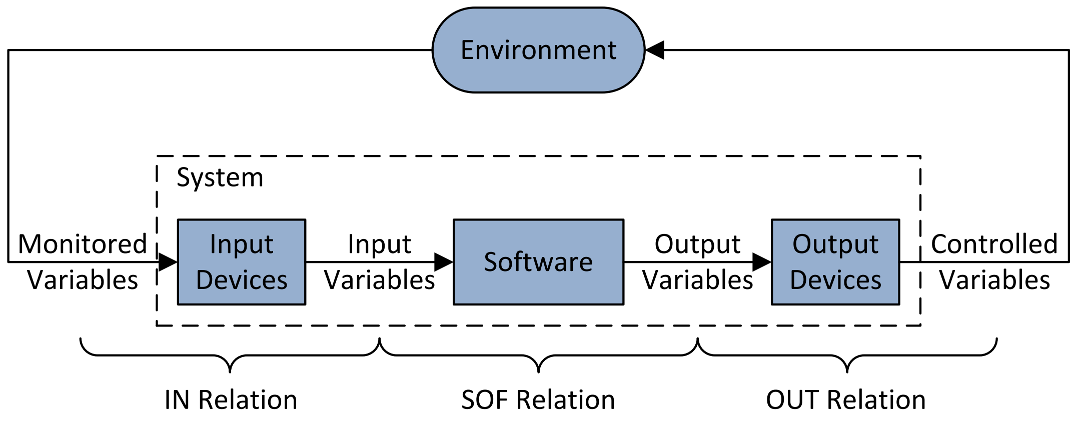


Figure 4.1: The four-variable model

In order to precisely characterize timing at different system boundaries, our approach will be based on Parnas' four-variable model. Figure 4.1 shows the four-variable model defined by Parnas et al. [73].

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 [73]. Given definitions of the four variables, the following relations can be defined.

IN Relation: $(\utilde{m}^t, \utilde{i}^t) \in IN$, where $\utilde{m}^t$ and $\utilde{i}^t$ represent the vectors of monitored and input variables, respectively, represents the physical interpretation of the input devices.

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

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

4.1.1 Extended Problem Statement

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

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 by 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.

4.1.2 Motivating Example

We use a PCA (Patient-Controlled Analgesia) infusion pump system – one of our existing case example systems – as an example to illustrate the problem and our approach. 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.

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 the following model-based test scenario:

Start-infusion button is pressed at the same time as air-in-line is detected.

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

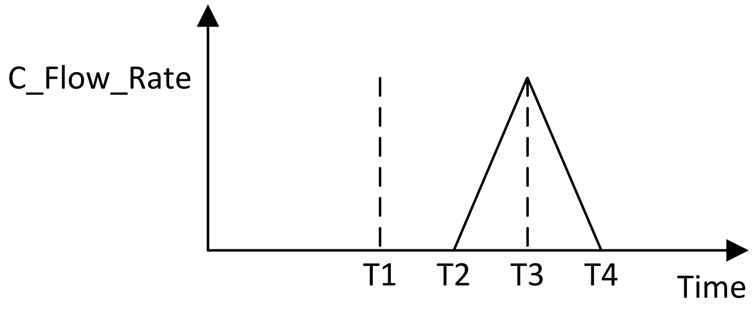


Figure 4.2: Flow rate change due to starting infusion and detecting air in line on the platform-specific implementation

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 4.2 illustrates how *C_Flow_Rate* (shown as y-axis) changes at different time (shown as x-axis):

1. At time $T1$, both *M_Start_Infusion* and *M_Air_in_Line* are set to *true*, as expressed by the test scenario.

2. At time $T2$, the start-infusion signal is received by the software controller. Then *I_Start_Infusion* becomes *true* and the software controller tries to start infusion by setting *O_Flow_Rate* to a calculated positive number, which subsequently increases *C_Flow_Rate*. As a result, *C_Flow_Rate* starts increasing from time $T2$.

3. At time $T3$, the air-in-line signal is received by the software controller. Then *I_Air_in_Line* becomes *true* and the software controller tries to stop infusion by setting *O_Flow_Rate* back to 0, which subsequently decreases *C_Flow_Rate*. As a result, *C_Flow_Rate* starts decreasing from time $T3$.

4. Eventually at time $T4$, *C_Flow_Rate* becomes 0 and infusion completely stops 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

Start-infusion button is pressed at the same time as air-in-line is 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.

4.2 Automated Framework

As used in the literature of testing real-time systems, we first 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.

4.2.1 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 [73], 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.

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 at step k in the following form:

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

and its oracle in the form:

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

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 of platform-specific implementations.

Now we lift this test interaction and its oracle from discrete-time (i.e., at step k) to real-time (i.e., at time t):

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

and its oracle would be:

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

where 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 and produces all output variable values at the same time $t+\Delta_{SOF}$[1] , where Δ_{SOF} represents the execution delay of the software controller and it is a platform-specific non-deterministic value.

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.

[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.

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 the system level at time t:

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

the corresponding test interaction at the software controller level would be:

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

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

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

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

the monitored variable vector would be:

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

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.

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.

Given a specific test output at the software controller level at time t:

$$(o_1^t(t), o_2^t(t), ..., o_s^t(t))$$

the corresponding test output at the system level would be:

$$(c_1^t(t + \Delta_{c_1}), c_2^t(t + \Delta_{c_2}), ..., c_q^t(t + \Delta_{c_q}))$$

where q is the number of controlled variables.

Similarly, when executing a system level test interaction and if we get a controlled variable vector:

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

the output variable vector would be:

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

which can be compared with the expected output produced by software models to determine whether the system is behaving correctly.

Events

We define an event as setting any one of the four variables (i.e., monitored, input, output, and controlled variables) to a specific value at a specific time. Note that multiple consecutive events can set a variable to the same value. 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 four variables. We use *M-event*, *I-event*, *O-event*, and *C-event* to represent events that set values for monitored, input, output, and controlled variables, respectively.

Figure 4.3 shows the four types of events at different system components in an architectural view of the execution environment, which we will further discuss in the following section.

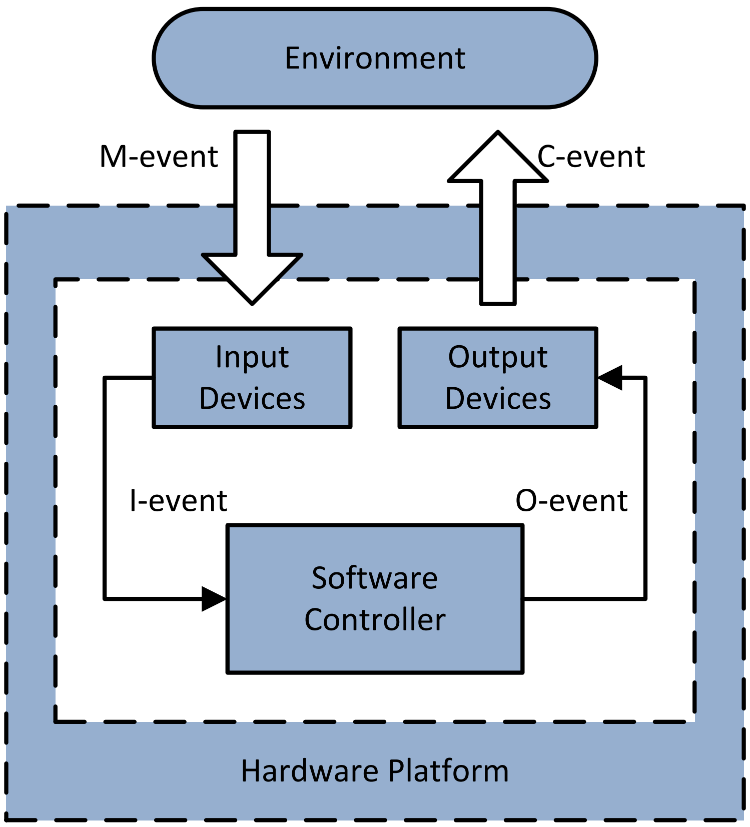


Figure 4.3: The architectural view of the execution environment

4.2.2 Platform-specific Implementations

A platform-independent model can often be translated and compiled to the code that runs on the hardware platform and we have been referring 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).

Figure 4.4 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 period of time. We define *step size* on the implementation as the time interval between two consecutive I-events.

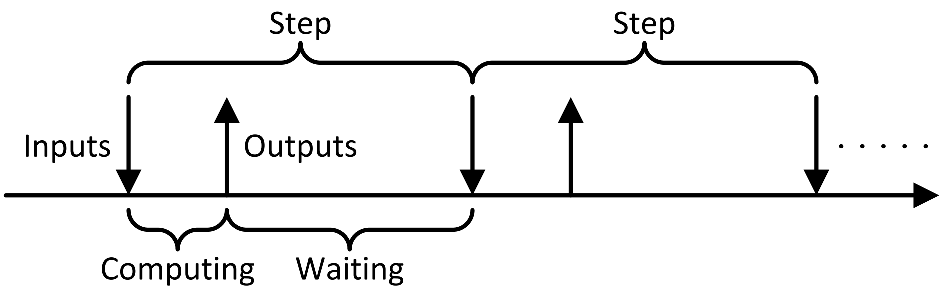


Figure 4.4: The input/output timing in real-time step

Figure 4.5 shows directly 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.

To execute model-based tests on platform-specific implementations, we have described how monitored and output variables can be derived from input and controlled variables, respectively, in an ideal situation where all delays are known and fixed values. However, all the delays are non-deterministic on the execution platform. That is, for

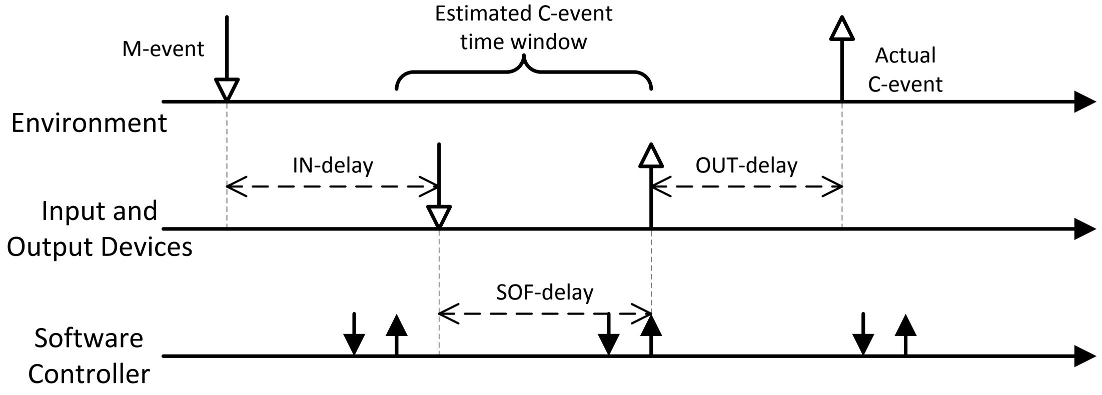


Figure 4.5: Real-time execution

example, when executing a reverse mapping $\underset{\sim}{m}^t$ from $\underset{\sim}{i}^t$ on the platform, the resulting input variable vector $\underset{\sim}{i}'^t$ may not always be equivalent to $\underset{\sim}{i}^t$ because the delay information used to derive monitored or output variables (1) may not accurately characterize the execution platform; and (2) may not match the non-deterministic delay induced from the execution platform at runtime.

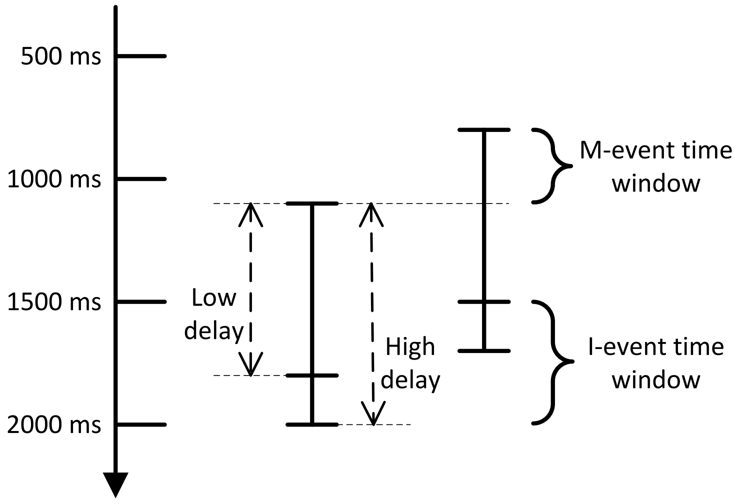


Figure 4.6: Time window for M-events and I-events

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 4.6 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.

4.2.3 Scheduling M-events and Adjusting C-events

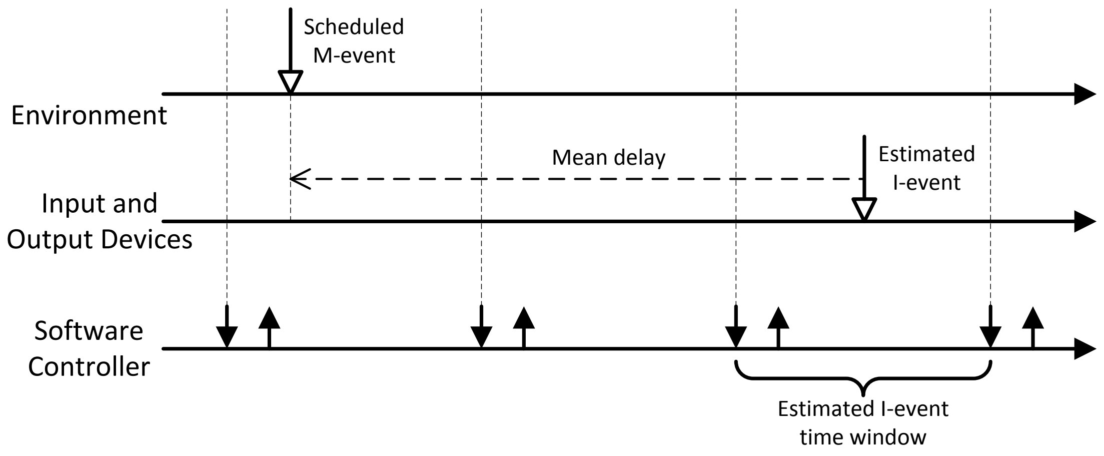


Figure 4.7: Scheduling M-events

Figure 4.7 shows how our approach schedules 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 [69].

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 $time_{I\text{-}events}$ when all I-events of the current test interaction should happen is estimated as:

$$time_{I\text{-}events} = current - step/2 + ceil((\Delta_{max} + step/2)/step) * step$$

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

$$time_{M\text{-}event_j} = time_{I\text{-}events} - \Delta_{m_j}$$

which is shown as *scheduled M-event* in Figure 4.7.

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.

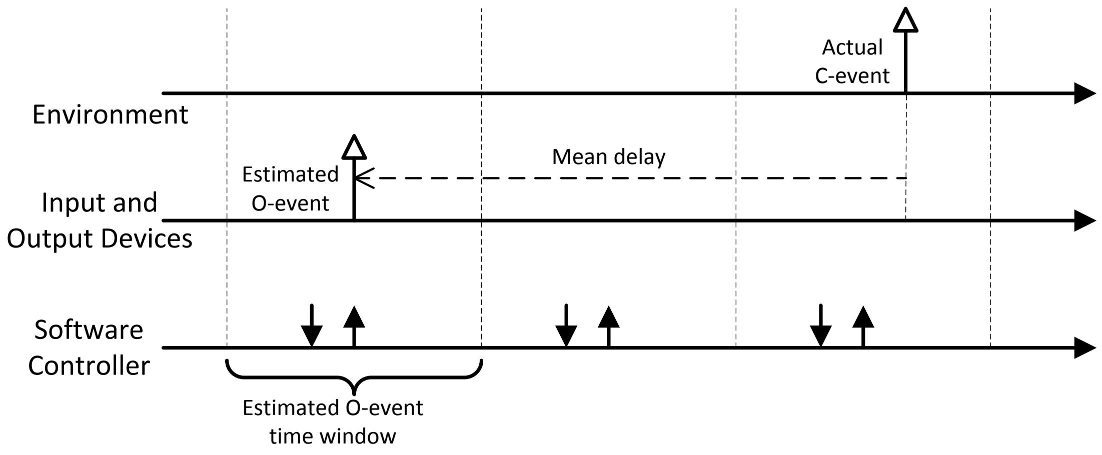


Figure 4.8: Adjusting C-events

Figure 4.8 shows how our approach adjusts 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 $time_{I\text{-}events}$ from scheduling M-events, we define a time window during which the corresponding O-event should happen, i.e., between $time_{I\text{-}events}$ and $time_{I\text{-}events} + step$. Therefore, given a C-event at time $time_{C\text{-}event_j}$, an output variable o_j with delay Δ_{o_j} should have been produced at time

$$time_{O\text{-}event_j} = time_{C\text{-}event_j} - \Delta_{o_j}$$

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.

There are many potential benefits of using this 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.

Chapter 5

Empirical Evaluation

We have described our approaches in Chapter 3 and 4. To assess the impact of our proposed approaches in practice, the next step is to implement them as parts of an automated framework that provides model-based test generation for platform-specific implementations. Specifically, We will need to evaluate the exact impact of the use of incremental test generation in terms of efficiency, test suite size, and fault finding effectiveness. Besides, we aim to examine the impact of platform-specific implementations on the test translation procedure in terms of the number of false positives and false negatives. We describe research questions on our proposed approaches and experimentation setup in this chapter.

5.1 Observability-based Test Generation

The quality in terms of fault finding of the test suites generated to satisfy OMC/DC and MC/DC has been evaluated [11]. In this dissertation, we are interested in comparing the two approaches – incremental and regular test generation satisfying observability-based coverage – in terms of efficiency and effectiveness. Therefore, we have the following research questions:

1. Is incremental test generation more efficient than regular test generation? What is the percentage of time needed to generate a test suite in the incremental approach compared with regular test generation?

2. What is the generated test suite size (i.e., the number of satisfied obligations) and how does it affect test suite effectiveness in the two approaches?

3. How well does the test suite generated by the incremental approach perform in terms of fault finding effectiveness compared with regular test generation?

5.1.1 Case Example Systems

In this study, we used four industrial systems (i.e., *DWM1*, *DWM2*, *Vertmax*, and *Latctl*) developed by Rockwell Collins Inc., two subsystems (i.e., *Alarm* and *Infusion Manager*) of a PCA (Patient-Controlled Analgesia) infusion pump system developed by the University of Minnesota and the University of Pennsylvania [74], and a last system (i.e., *Docking Approach*) created as a case example at NASA.

The Rockwell Collins systems were modeled using Simulink [46]. Two of the systems, *DWM1* and *DWM2*, represent distinct portions of a Display Window Manager for a commercial display system. The other two systems, *Vertmax* and *Latctl*, represent the vertical and lateral mode logic for a Flight Guidance System.

The infusion pump system was modeled using Simulink and Stateflow [46, 75]. The two largest systems, *Alarm* and *Infusion Manager*, represent the alarm-induced behavior and the prescription management of an infusion pump device.

The last system, *Docking Approach*, was created as a case example at NASA [12] and modeled using Stateflow [75]. The NASA system describes the behavior of a space shuttle as it docks with the International Space Station.

Table 5.1 and 5.2 show basic information of the systems measured on the original Simulink and Stateflow models.

	Subsystems	**Blocks**
DWM1	3109	11,439
DWM2	128	429
Vertmax	396	1,453
Latctl	120	718

Table 5.1: Simulink case example information

	States	**Transitions**	**Variables**
Alarm	78	107	60
Infusion Manager	27	50	36
Docking Approach	64	104	51

Table 5.2: Stateflow case example information

Each of the systems under test represents sizable, realistic industrial systems. All the systems were translated to the Lustre programming language [48] to take advantages of existing automation. Lustre code generation from Simulink/Stateflow models is analogous to the automated code generation offered by Mathworks Simulink Coder [76]. In practice, Lustre programs will be automatically translated to C code. Therefore, results of this study would be identical, if applied to their C implementations.

For each Lustre implementation of our case examples, we:

1. Generated 10 test suites each for incremental and regular test generation.
2. Generated 250 mutants for each system.
3. Ran reduced test suites on mutants with output-only test oracles.
4. Assessed fault finding effectiveness of each test suite and test generation approach combination.

5.1.2 Test Suite Generation

In order to generate tests satisfying observability-based coverage criteria, we used the counterexample-based approach [5, 6]. This approach is guaranteed to generate a test suite that achieves the maximum possible coverage of the system under test. Note that, however, in regular test generation, this is guaranteed as long as the model checker is given enough computing resource to terminate; in incremental test generation, the maximum propagation is only guaranteed at each step, which can have both advantages and disadvantages and will be discussed in detail in Chapter 6. To take advantages of existing automation on Lustre programs, we used JKind [77] as the underlying model checker. JKind is a multi-engine SMT-based automatic model checker for Lustre programs and uses Z3 as the underlying constraint solver [78].

The obligations generated to satisfy an observability-based coverage criterion will differ depending on the set of monitored variables. In this study, we generate test obligations with respect to the output variables of each system, as an output-only oracle is most likely to be used in practice [79]. When using the maximum oracle, all variables are observable. Thus, for example, the OMC/DC obligations would be equivalent to MC/DC obligations [11].

To account for variance in the generation time, we generated 10 test suites for each of the two approaches (i.e., incremental and regular test generation) for each of the seven systems under test. We measured time using Linux *time* utility and recorded the elapsed (i.e., wall-clock) time. The model checker JKind has multiple engines (e.g., bounded model checking [80], k-induction [81], and property directed reachability [82]) for different verification purposes and we only enabled bounded model checking for test generation. In this case, JKind itself runs a process and creates another process for the underlying constraint solver Z3, both of which are single-threaded in our experiments. Therefore, the addition of *user time* and *system time* measured from Linux *time* utility may be larger than the elapsed time in a multi-core environment.

Counterexample-based test generation results in a separate test for each generated coverage obligation. This results in a large amount of redundancy in the generated tests, as each test case likely covers several coverage obligations. Such an unnecessarily large test suite is unlikely to be used in practice. We therefore have reduced each generated test suite while maintaining a consistent level of coverage. To generate these reduced test suites, we make use of a simple randomized greedy algorithm. We begin by determining the coverage obligations satisfied by each test generated, and initializing an empty test set, *reduced*. We then select a test input at random from the full set of tests; if this test satisfies any obligations not satisfied by the existing test inputs in *reduced*, we add it to the set. This process continues until all tests have been removed from the full set.

For each of the 10 test suites we have generated for each approach and each system, we produced a reduced test suite for evaluating fault finding effectiveness. The 10 test suites will eliminate the possibility that we by accident create a very good (or very poor) test suite in the test suite reduction step. Furthermore, it is debatable whether test suite reduction will decrease fault finding effectiveness [83, 84], and the choice of reduction algorithms may also have an effect on fault finding and reduced test suite size. We have consistently used the simple randomized greedy algorithm, although a more sophisticated test suite reduction algorithm may produce an even smaller test suite.

5.1.3 Mutant Generation

Mutation testing is the practice of automatically generating *faulty* implementations of a system for the purpose of empirically examining the fault finding effectiveness of a test suite [85]. During mutation testing, clones of the system under test are created by introducing a single fault into the program. This method is designed such that all mutants produced are both syntactically and semantically valid. That is, the mutants will compile, and no mutant will "crash" the system under test.

Mutation testing has been shown to be an adequate proxy for real faults for the purpose of investigating fault finding effectiveness [86]. In our experiment, mutant generation is done by automatically introducing a single fault into the correct implementation. The mutation operators used in this experiment are similar to those used by other researchers [86, 87], for example, arithmetic, relational, Boolean operator replacement, Boolean variable negation, constant replacement, and delay introduction. The generated mutants have roughly the same distribution of fault types across the system occurring naturally and are evenly distributed across the system.

- **Arithmetic:** Changes an operator (+, -, /, *, mod, exp).
- **Relational:** Changes a operator ($=, \neq, <, >, \leq, \geq$).
- **Boolean:** Changes a operator ($\vee, \wedge$, XOR).
- **Negation:** Introduces the Boolean $\neg$ operator.
- **Delay:** On a variable reference, use the stored value of the variable from the previous computational cycle rather than the newly computed value.
- **Constant:** Changes a constant expression by adding or subtracting 1 from int and real constants, or by negating Boolean constants.
- **Variable Replacement:** Substitutes a variable occurring in an equation with another variable of the same type.

For each of our systems, we created 250 mutants. We removed mutants that cause compile-time or run-time errors (e.g., divide-by-zero), but we left possibly equivalent mutants. Although checking and removing equivalent mutants can be feasible on small systems [11], the cost of checking equivalent mutants for our last three systems is prohibitive. Therefore, we did not attempt to remove equivalent mutants in our experiments in order to keep our experimental configuration consistent across different case example systems.

5.1.4 Test Oracles and Data Collection

For the purpose of this study, we used output-only expected value oracles – the ones commonly used by our industrial partners in the testing of critical software systems [88]. When using an output-only expected value test oracle, for each test input, concrete values are specified that the system is expected to produce for each of the system's output variables.

For each case example, we ran the reduced test suites against each mutant and the original version of the system (i.e., the correct version). For each test suite, we recorded the value of all output variables at every test step of the execution of every test case using an in-house Lustre simulator.

To determine the fault finding effectiveness of the generated test suites, we simply compared output values produced by a faulty version against expected output values produced by the original version. The fault finding effectiveness of a test suite is computed as the percentage of mutants killed. We performed this analysis for each test suite for every case example.

Due to proprietary reasons, experiments on the first four systems were performed on an encrypted laptop with an Intel quad-core processor @ 2.4 GHz, 4 GB memory, and Ubuntu Linux. Experiments on the last three systems were performed on a server with four AMD eight-core processors @ 3.0 GHz, 192 GB memory, and Ubuntu Linux.

5.2 Platform-specific Test Execution

In this dissertation, we are interested in the effectiveness of our automated framework for platform-specific test execution. Specifically, we wish to evaluate the following two aspects:

- To what extent does the automated framework reduce false positives?
- To what extent does the automated framework introduce false negatives?

As used in the literature of testing real-time systems, *false positive* and *false negative* have been defined in Section 4.2 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.

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.

Some existing techniques that test platform-specific implementations aiming at reducing false positives often introduce additional false negatives [24], 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.

5.2.1 Case Example Systems

We used a PCA (Patient-Controlled Analgesia) infusion pump system developed by the University of Minnesota and the University of Pennsylvania [74] as our case study example. This system is modeled using MathWorks Simulink and Stateflow [46, 75]. The 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.

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

Table 5.3: Infusion pump subsystem information

The top level system has 76 input variables and 31 output variables. This system also contains seven subsystems. Table 5.3 shows basic information of the subsystems. Some of the subsystems have been used as case examples in previous studies [12, 24, 89].

5.2.2 Device Configuration

In order to perform platform testing, we have used MatLab Simulink Coder [76] to generate C code from the Simulink and Stateflow models. 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.

5.2.3 Sensor and Actuator Simulation

We identified multiple sources of timing delays and wanted to assess how the delays affect testing and how our approach reconciles timing induced mismatches. 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 where we ran model-based tests on platform-independent models.

5.2.4 Test Suite Generation

Though test cases from any source can be used with our approach, we generated our test suite using Simulink Design Verifier [90] 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 on each of the 20 different platform-specific implementations plus on the platform-independent model, 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.

5.2.5 Mutant Generation

For C programs, We generated mutants using Milu – a mutation testing tool developed by University College London [91]. Milu adopted the 77 C mutation operators of Agrawal et al. [92] and it also supports customized mutation operators. We generated 50 mutants using default mutation operators 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 [93, 11], it does not scale well. Therefore, we did not attempt to remove equivalent mutants since it is cost-prohibitive for our case example systems.

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 platform-independent model 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.

Chapter 6

Results and Discussion

We have described our experimental configuration to empirically evaluate our approaches in Chapter 5. In this chapter, we will discuss the results of our experiments. Specifically, we will first discuss the results of observability-based test generation using the incremental approach in Section 6.1. Then we will examine the results of using our automated framework to reconcile abstraction differences for platform-specific implementations in Section 6.2.

6.1 Observability-based Test Generation

Recall that in Section 5.1 we outlined three research questions related to the possible performance gains with the incremental test generation approach, the nature of the generated test suites (i.e., the number of tests generated), and the fault finding effectiveness of tests generated incrementally. Below we will present our experimental results and discuss the three questions in order.

6.1.1 Test Generation Time

We would first like to determine whether the incremental approach performs better than regular test generation in terms of test generation time.

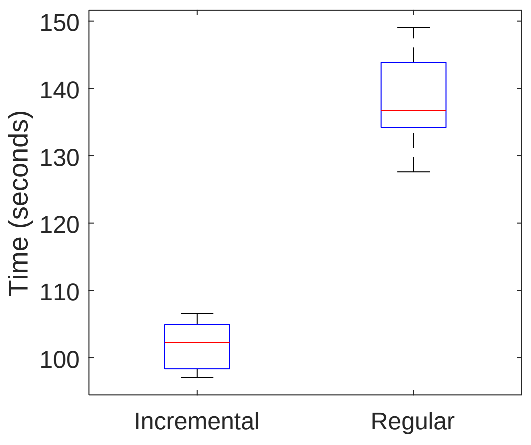
Table 6.1 shows the average test generation time for each case example system and each test generation approach. The *Time Ratio* column is the percentage of time needed to incrementally generate a test suite as opposed to the regular generation of

System	**Incremental Generation**	**Regular Generation**	**Time Ratio**	**p-value**
DWM1	101.7	137.6	73.9%	< 0.01
DWM2	14.1	41.1	34.3%	< 0.01
Vertmax	59.1	234.5	25.2%	< 0.01
Latctl	7.4	20.4	36.3%	< 0.01
Alarm	2844.8	10 hours*	N/A	< 0.01
Infusion Manager	354.6	2422.2	14.6%	< 0.01
Docking Approach	48 hours*	48 hours*	N/A	N/A

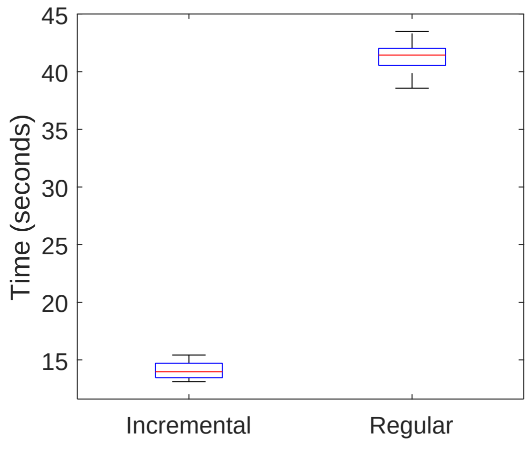
Table 6.1: Test generation time (in seconds) for each system, average over 10 test suites

the suite. The *p-value* is computed by a two-sided permutation test using *mean* as the test statistic, under the null hypothesis that test generation time from incremental and regular approaches are drawn from the same distribution. Note that certain test generation timed out (as starred in Table 6.1). Specifically, regular test generation timed out on *Alarm* after a time limit of 10 hours; and both incremental and regular test generation timed out on *Docking Approach* after a time limit of 48 hours. It is both time and memory prohibitive to obtain the actual test generation time. We set the timeout based on our experience from previous studies on the same systems [11, 12]. Setting a larger timeout may result in more test cases, but it may not have practical value since the growth is often too slow. Figure 6.1 and 6.2 further illustrate the distribution of test generation time for each system, excluding the cases where the test generation timed out.

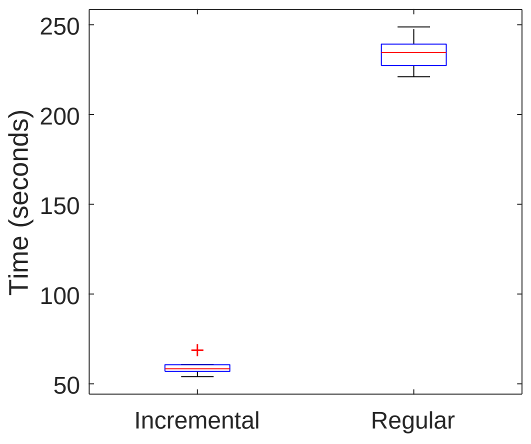
The incremental approach is significantly more efficient than regular test generation for all systems, while it achieved different improvements depending on the nature of each system. Specifically, the incremental approach achieved the least improvement on *DWM1*. We investigated this outlier and found that *DWM1* is purely combinatorial, i.e., there is no state maintained in the system and thus no delay operations. Therefore, all computation is within one step and finding an immediate non-masking path is generally cheap since the tag propagation is trivial. Using the incremental approach, however, can simplify the path constraints. Thus, the efficiency can still be gained.



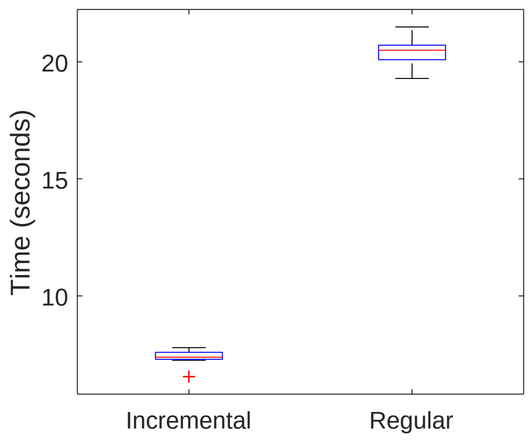
(a) DWM1



(b) DWM2

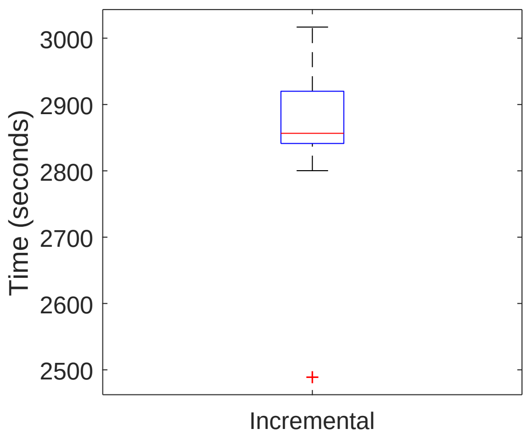


(c) Vertmax

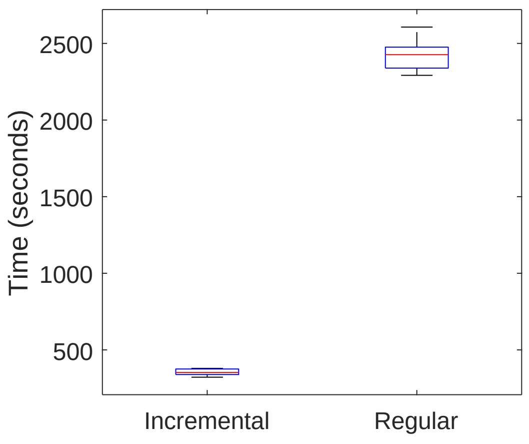


(d) Latctl

Figure 6.1: Test generation time for Rockwell Collins systems



(a) Alarm



(b) Infusion Manager

Figure 6.2: Test generation time for Alarm and Infusion Manager

On five of the systems (i.e., *DWM2*, *Vertmax*, *Latctl*, *Alarm* and *Infusion Manager*), the incremental approach required much less time as opposed to regular test generation with a trend that the larger/more complex the system is, the more is the improvement achieved. For smaller systems, the overhead associated with the incremental approach (e.g., executing generated tests to get a concolic state) takes a larger portion of the entire generation time. Additionally, our approach calls the model checker multiple times and JVM overhead can be significant on small systems with an execution time of tens of seconds. On *Docking Approach*, although both of the two approaches timed out, the incremental approach is able to search a larger state space of the system. Specifically, the incremental approach can reach an average depth of 34 steps, while the regular approach can only reach an average depth of 20 steps. We will further show in the next two sections that the incremental test generation approach achieved improvements on other aspects.

The observed improvement matches our expectation. As we generate new test inputs at each step, we concretize the inputs we have already computed and solve a much simpler path constraint to further propagate tags based on the execution results of existing test inputs. As a result, efficiency can be improved. Furthermore, the incremental approach separates "reaching a certain program state" from "propagating its effect to the output". When a program state is not reachable or the effect of reaching a program state cannot even be propagated to internal variables, the incremental approach can terminate early without wasting time searching deeper program states. However, as we have introduced in Section 2.3, this simplification also makes the incremental approach lose completeness as a result of concretizing certain variables. Therefore, we will further evaluate our approach in the following section in terms of satisfied coverage obligations and the nature of generated test suites.

6.1.2 Coverage Obligations and Test Suites

The incremental test generation approach may affect the number of satisfied coverage obligations in different ways. On the one hand, since the incremental approach concretizes all input values at each step, it may fail to propagate tags in later steps, because future paths may have been rendered infeasible based on the previous concretization. As result, the incremental approach may generate a test suite satisfying fewer coverage obligations than the regular approach. On the other hand, since the incremental approach is able to start from a concolic state to simplify path constraints, it may be able to search a much larger state space than the regular approach, and thus generate a test suite satisfying more coverage obligations. Due to the same reason, the generated test case length may also vary depending on the nature of the systems. We have observed both cases in our experiments and we will discuss the number of satisfied coverage obligations and test case length in this section.

Table 6.2 shows the total number of coverage obligations and the number of satisfied obligations for each test generation approach and each case example system[1]. On the four Rockwell Collins systems, (i.e., *DWM1*, *DWM2*, *Vertmax*, and *Latctl*), the incremental approach is able to satisfy the same number of coverage obligations as the regular approach, while using much less time as shown in the previous section.

On two of the systems (i.e., *Alarm* and *Infusion Manager*), the incremental approach did not satisfy as many coverage obligations as the regular approach. Compared with the regular approach, the incremental approach missed 14 obligations (1.0%) for *Infusion*

[1] The number of satisfied coverage obligations for each system is generally smaller than those in our previous studies [11, 13], since we have used the enhanced tagging semantics for observability as introduced in Section 3.1. Therefore, more obligations would become unsatisfiable and tests that in fact cannot propagate tags (i.e., optimistic inaccuracy in the original OMC/DC) would be eliminated.

System	Total Obligations	Incremental Generation	Regular Generation
DWM1	2038	1833	1833
DWM2	530	511	511
Vertmax	1730	1704	1704
Latctl	370	369	369
Alarm	1406	1012	1042
Infusion Manager	1666	816	830
Docking Approach	4562	1644	1310

Table 6.2: Satisfied obligations for each system, average over 10 test suites

Manager and 30 obligations (1.8%) for *Alarm*. In these cases, we have found that the incremental approach failed to propagate tags as a result of previous concretization of the program state, losing completeness. In practice, however, the incremental approach may still be more preferable. Compared with the regular approach, the incremental approach required only 14.6% of the generation time on *Infusion Manager*, and the regular approach timed out on *Alarm*.

On *Docking Approach*, the incremental approach is able to satisfy 334 more obligations on average than the regular approach. Because the incremental approach using dynamic symbolic execution can simply path constraints, it is able to explore execution paths and generate test inputs that the regular approach is incapable of searching. As a result, more coverage obligations can be satisfied and more test cases can be generated.

Besides the number of satisfied coverage obligations, test case length can also show insight of the generated test suites. We first compute the average test case length in a test suite, then we compute the average of the average test case length over 10 test suites for each approach and each system[2] . Table 6.3 shows this information.

As discussed in the previous section, *DWM1* is purely combinatorial and all of the generated test cases have only one step. On all systems except *Docking Approach*, the generated test suites have very similar test case length between the incremental approach and the regular approach. On *Docking Approach*, the incremental approach

[2] The average of the average test case length is different from the average of all test cases in the 10 test suites, because the number of test cases in each test suite may be different.

System	Incremental Generation	Regular Generation
DWM1	1.000	1.000
DWM2	2.004	1.988
Vertmax	2.334	2.334
Latctl	1.970	1.970
Alarm	2.139	2.111
Infusion Manager	3.800	3.916
Docking Approach	16.942	13.581

Table 6.3: Average test case length for each system, average over 10 test suites

generated test cases with an average of 3.4 more steps than the regular approach. As a result, more program states can be explored and more coverage obligations can be satisfied. Figure 6.3 further shows the distribution of average test case length of the 10 generated test suites for *Docking Approach*.

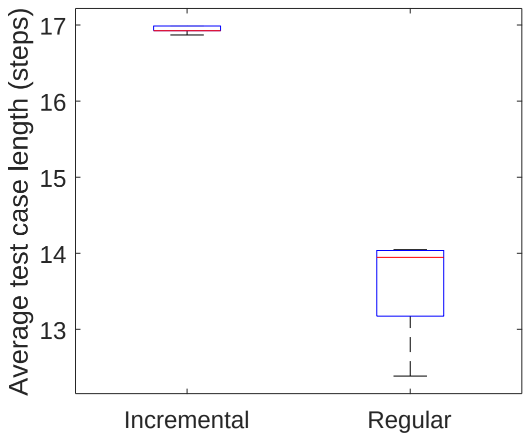


Figure 6.3: Average test case length for Docking Approach

Previous investigations on test suite size and fault finding effectiveness have shown that there is a moderate to high correlation between them [94, 12, 95]. In our experiments, however, the incremental approach generated smaller test suites for *Alarm* and *Infusion Manager* (thus, satisfying fewer coverage obligations), which may lead to worse fault finding effectiveness. On the other hand, the incremental approach generated larger test suites for *Docking Approach* (thus, satisfying more coverage obligations), which may lead to better fault finding effectiveness. Fault finding effectiveness of the generated test suites could be a concern since we do not wish to achieve better efficiency by losing fault finding effectiveness. Therefore, in order to fully understand our approach, we will evaluate fault finding effectiveness between incremental and regular test generation approaches in the next section.

6.1.3 Fault Finding Effectiveness

System	**Incremental Generation**	**Regular Generation**
DWM1	346.6	350.0
DWM2	68.9	70.6
Vertmax	216.3	218.0
Latctl	46.4	44.9
Alarm	210.1	210.1
Infusion Manager	171.3	146.5
Docking Approach	181.8	180.2

Table 6.4: Reduced test suite size, average over 10 test suites

System	**Incremental Generation**	**Regular Generation**
DWM1	95.4%	95.5%
DWM2	97.2%	98.0%
Vertmax	98.4%	98.4%
Latctl	95.5%	95.6%
Alarm	60.9%	62.0%
Infusion Manager	53.6%	52.0%
Docking Approach	35.0%	25.8%

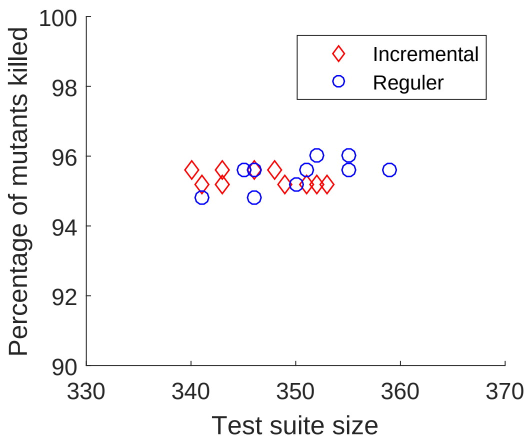
Table 6.5: Percentage of mutants killed for each system, average over 10 sets of mutants

As introduced in Section 5.1, both incremental and regular test generation approaches result in a separate test for each generated coverage obligation. This results in a large amount of redundancy in the generated tests, as each test case likely covers several coverage obligations. Such an unnecessarily large test suite is unlikely to be used in practice. Therefore, we have reduced each generated test suite while maintaining a consistent level of coverage using a simple randomized greedy algorithm. Table 6.4 and 6.5 show the average reduced test suite size and the average percentage of mutants killed for each case example system and each test generation approach, respectively. Figure 6.4 and 6.5 further illustrate the distribution of fault finding effectiveness together with reduced test suite size.

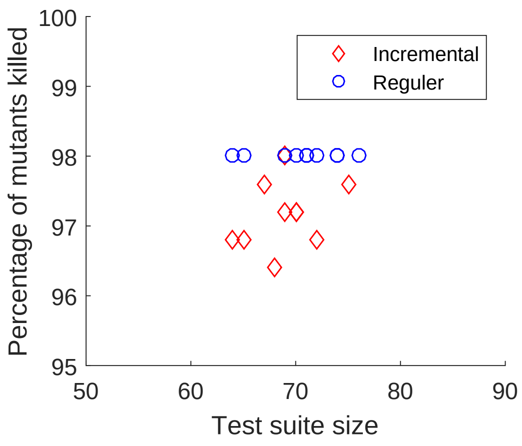
The test suite size can be a concern in our experiments because previous studies have shown that a larger test suite can often lead to better fault finding effectiveness [94, 95]. On the other hand, stronger coverage criteria often require more test cases to satisfy [11, 12]. The incremental approach, as shown in the previous section, can lead to more satisfied obligations and larger generated test suites (i.e., *Docking Approach*), or fewer satisfied obligations and smaller generated test suites (i.e., *Alarm* and *Infusion Manager*).

In the incremental approach, the state space represented as a concolic state is smaller than the actual reachable state. Therefore, the test cases for different coverage obligations in the incremental approach may diverge early as a result of concretizing certain variables. That is, as soon as a test case is created, it may represent a different concolic state from those represented by other test cases. On the other hand, the regular approach does not concretize intermediate program states and it tries to satisfy as many test obligations as possible when generating a test case. This effect can be seen on the system *Infusion Manager*, where the reduced test suite size for the incremental approach is larger than that for the regular approach. That is, each test case is *less likely* to cover multiple coverage obligations. The system *Docking Approach* implicitly has this effect, because the average test case length is much larger for the incremental approach and the total number of test steps is also much larger even if the reduced test suite size are the same.

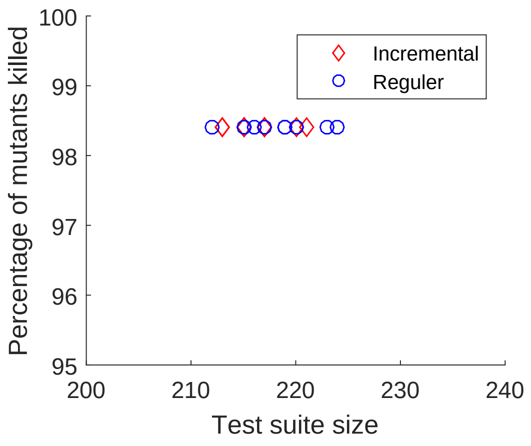
Fault finding effectiveness of incrementally generated tests is for our case example



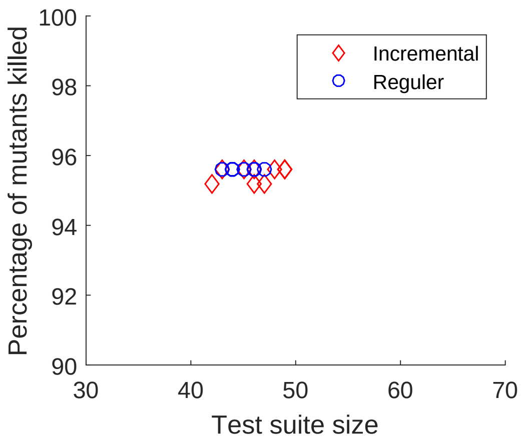
(a) DWM1



(b) DWM2

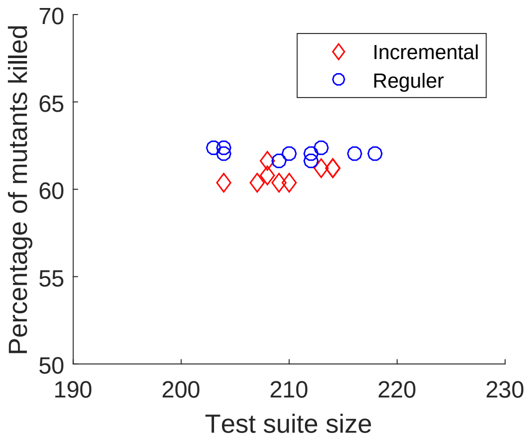


(c) Vertmax

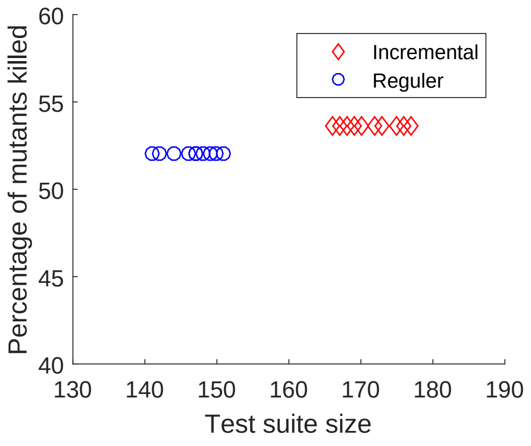


(d) Latctl

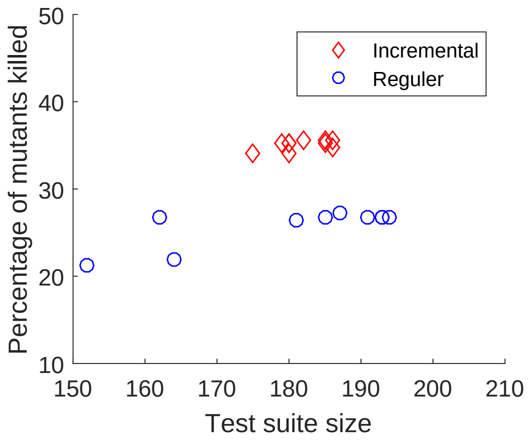
Figure 6.4: Fault finding effectiveness for Rockwell Collins systems



(a) Alarm



(b) Infusion Manager



(c) Docking Approach

Figure 6.5: Fault finding effectiveness for Alarm, Infusion Manager, and Docking Approach

systems very similar to regularly generated tests, except on the system *Docking Approach*. On *Docking Approach*, the incremental approach achieved much better fault finding effectiveness compared with regular approach, as a result of being able to search more and deeper of the program states and satisfy more coverage obligations.

Furthermore, JKind is an SMT-based model checker and will always generate the shortest test case possible to satisfy an obligation or a path constraint. In our experience, such tests technically satisfy the coverage criterion, but they are (1) susceptible to the masking effect discussed earlier and (2) likely to only vary the few input variables needed to cover the obligation and leave all other input variables at the model checker's default values (typically zero and false) [12]. Both issues are mitigated in our incremental approach. First, the incremental search for a propagation path allows the possibility to pursue a path longer than the shortest propagation path. Second, since we are restarting the search for a path in each incremental step, there is an opportunity for more variables to change values (i.e., we are no longer searching for the shortest path with the fewest variable changes). From our experimental results, we see that a combination of these two aspects of the incremental approach leads to longer and more diverse tests with better fault finding effectiveness.

On a related note, as introduced in Section 5.1, we did not remove equivalent mutants in this study in order to keep our experimental configuration consistent across different case example systems. As a result, fault finding effectiveness for the first four systems would be in general slightly lower than our previous results [11].

6.2 Platform-specific Test Execution

In this section, we address our research questions and discuss the implications of our results of using the automated framework to reconcile abstraction differences between platform-independent models and platform-specific implementations. We begin by presenting false positives that can be reduced using our approach.

6.2.1 Reducing False Positives

As introduced in Section 5.2, we have generated a reduced test suite satisfying branch coverage on the infusion pump system. Specifically, the reduced test suite contains

Max Mean	**Std. Deviation**	**Scheduled Execution**	**Direct Execution**	**FP Decrease**
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 (18.80%)	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%)	14.60%
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%)	79.10%
	200	134 (12.79%)	2 (0.19%)	12.60%

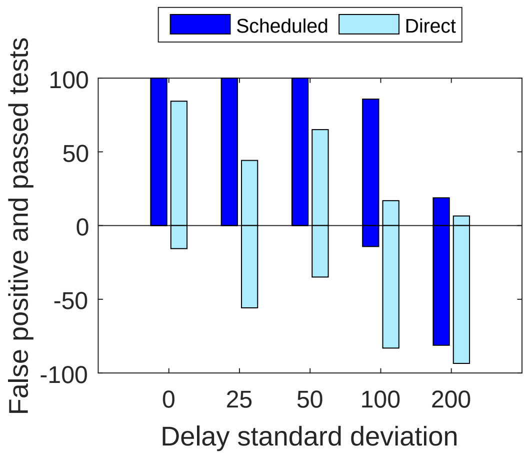
Table 6.6: Median number of passed tests and percentage point decrease in false positives

116 test cases with a total of 1048 test interactions. For the first research question, we ran the full test suite with 1048 test interactions on 20 different platform-specific implementations. On each implementation, 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 the injected sensor and actuator delays. All test interactions are supposed to pass on each platform-specific implementation and any failed test interaction is considered false positive.

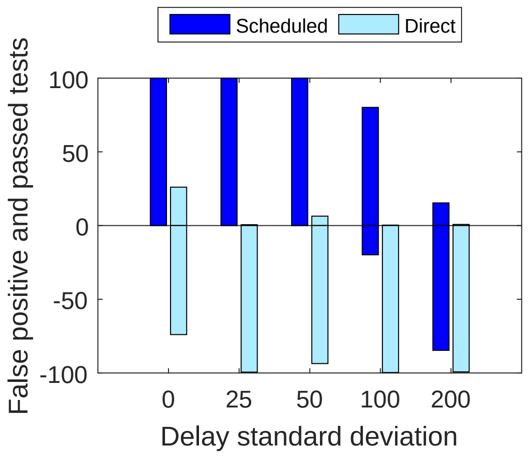
Table 6.6 shows the number and percentage of passed test interactions using the proposed approach (i.e., scheduled execution) and direct execution of model-based tests on each platform-specific implementation. The percentage point decrease in false positives is simply calculated by subtracting failed percentage of test interactions using scheduled execution from that using direct execution. Figure 6.6 further compares the numbers of false positive and passed tests between scheduled and direction executions of model-based tests. Specifically, each bar represents all test interactions for each approach and each platform-specific implementation. In each bar, the part above the 0 y-axis represents passed test interactions and the part below the 0 y-axis represents false positives.

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.

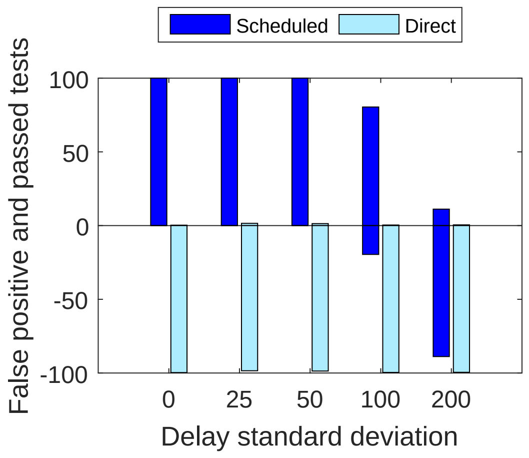
The proposed approach reduces false positives in all cases. As shown in Table 6.6, our approach is robust to absolute delay values (i.e., max mean), but can still be sensitive



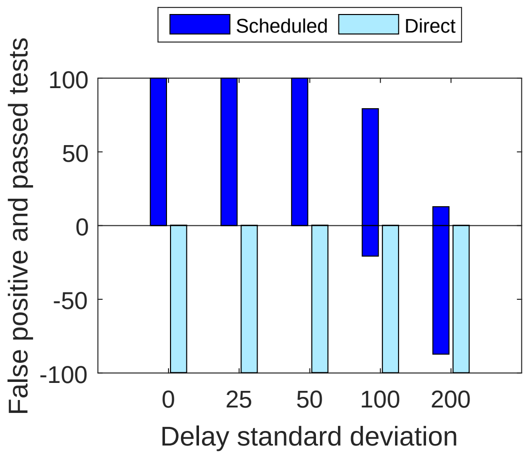
(a) Max mean 250 ms



(b) Max mean 500 ms



(c) Max mean 1000 ms



(d) Max mean 2000 ms

Figure 6.6: Comparison of passed test interactions and false positives

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 4.6, 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 experiment always estimates and expects I-events to happen in the middle of two consecutive steps, which already gives 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.

6.2.2 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.

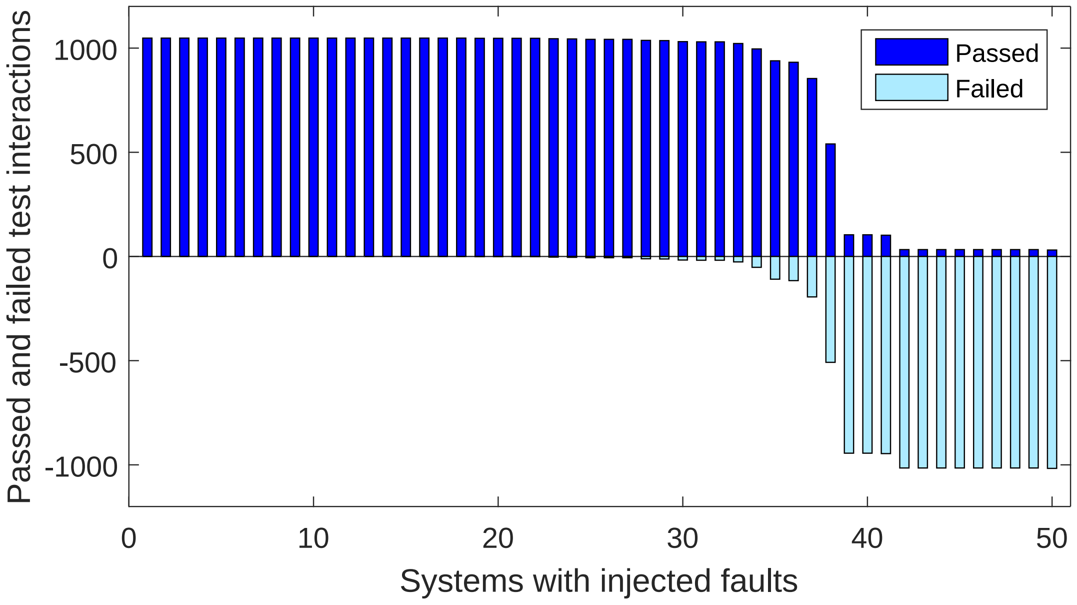


Figure 6.7: Passed and failed test interactions on each faulty system

To evaluate the number of false negatives that may be introduced by our approach, 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, the platform-specific implementation performed consistently with the platform-independent model in terms of passing/failing each test interaction and thus we did not observe any false negative as a result of using our approach.

Figure 6.7 shows the number of passed and failed test interactions for each system with one injected fault. Note that the first 18 faulty systems did not impose any failed test interactions. That is, the 18 injected faults were not detected by the test suite satisfying branch coverage on the system.

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 false negatives.

6.3 Threats to Validity

6.3.1 External Validity

We have used seven synchronous critical systems of different sizes from different domains (i.e., industrial avionics systems and medical device systems) in our test generation experiments. We believe that these system are representative of avionics and medical device domains. Thus, our results are generalizable to other systems in these domains. Although we used only one system as the case example to evaluate our automated framework for platform-specific testing, we actively worked with domain experts to set up a realistic experimental environment. Furthermore, the infusion pump system consists of subsystems some of which have been used in previous studies [12, 24, 89] as standalone systems, including our test generation experiments.

We have used the dataflow language Lustre [48] as the implementation language. Although Lustre is a less common language than C/C++ or Java, these Lustre programs are similar in structure to systems written in C/C++ in these domains. In practice, Lustre programs will be automatically translated to C code. Therefore, we believe that our results are applicable to systems written in more traditional imperative languages.

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.

We have generated certain numbers of mutants for each of our case example systems. The total number of mutants in each experiment and for each system was chosen to provide enough data points and yield a reasonable cost. Based on past experience, results using these numbers of mutants are representative [88]. The mutants were generated using tools that have been widely used in previous studies [11].

6.3.2 Internal Validity

We have used several tools to generate test cases, including the model checker JKind [77] with Z3 as the underlying constraint solver [78] and Simulink Design Verifier [90] to generate test cases satisfying branch coverage on the model in our experiments. The counterexample-based approach provides the shortest test cases possible to satisfy an obligation or a path condition. These automatically generated test cases may behave differently from test cases obtained by other means [12], and thus, it is possible that test cases derived by hand or random generation, may provide different results.

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.

6.3.3 Construct Validity

We have generated mutants for our case example systems and used mutants as proxy of real faults. Specifically, the fault finding effectiveness of generated tests is measured over mutants rather than real faults encountered during development. We also 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 it has been shown that the use of mutants leads to conclusions similar to those obtained using real faults [86].

Chapter 7

Conclusions and Future Work

Real-time systems are more and more common and model-based testing has been widely used in this domain. We have described our approaches that efficiently use software models as the source for test data generation and effectively translate these model-based tests to equivalent scenarios for the execution platform. Our framework will improve and advance the applicability of model-based testing in practice.

We have formally defined the notion of observability to address the masking problem in structural coverage criteria. Specifically, traditional coverage criteria specify that various structural parts of a program must be exercised, but they do not mandate that the effect of exercising that part must manifest itself at a subsequent observable point in the program. As a result, they are sensitive to program restructuring and the effect of a corrupted variable often gets masked out through its use in subsequent operations. The notion of observability mandates that the effect of exercising a structural part must manifest itself at a subsequent observable point in the program.

We have developed an incremental test generation approach that combines the notion of observability and dynamic symbolic execution. This approach is proposed to address the scalability problem in regular counterexample-based test generation satisfying observability-based coverage criteria. We have empirically evaluated our approaches on seven case example systems from avionics and medical device domains. Our results indicate that compared with traditional test generation approach, the incremental approach is far more efficient and feasible on large systems and is able to generate test suites satisfying more coverage obligations and detecting more faults in cases where the traditional counterexample-based approach is incapable of analyzing.

To execute model-based tests generated by our own approaches and other techniques on the platform-specific implementations, we have described our testing framework based on the four-variable model defined by Parnas et al. [69] and the framework for testing timing defined by Kim et al. [73]. 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.

While our results are encouraging, there are the following areas open for exploration in future research.

- Improving completeness of the incremental test generation approach. A coverage obligation that is satisfiable may not have a feasible path to outputs by searching from a concolic state, because future paths may have been rendered infeasible based on the previous concretization. While it is a common problem in dynamic symbolic execution that completeness is often traded off against efficiency, the incremental approach may still be preferable because it is able to explore more execution paths and satisfy more coverage obligations. Nevertheless, we may be able to reduce the number of concretized variables or use symbolic expressions rather than concrete values to represent a larger state space. Additional research is required since having more symbolic expressions will also slow down the entire test generation process. The extreme case would be the same as classical symbolic execution, where every variable is represented by a symbolic expression.
- Improving the accuracy of the automated framework to reconcile timing delays arising from execution platforms. Our current approach takes a measured 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 may be applied that can make better use of delay information. Furthermore, we have generated event sequences that can contain important timing information of the platform-specific implementation. 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.

References

[1] Orna Grumberg and David E Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[2] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[3] Myla Archer, Constance Heitmeyer, and Steve Sims. TAME: A PVS interface to simplify proofs for automata models. Technical report, DTIC Document, 1998.

[4] Saddek Bensalem, Paul Caspi, Catherine Parent-Vigouroux, and C Dumas. A methodology for proving control systems with Lustre and PVS. In *Dependable Computing for Critical Applications 7, 1999*, pages 89–107, 1999.

[5] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the 7th European Software Engineering Conference held jointly with 7th International Symposium on Foundations of Software Engineering*, pages 146–162, 1999.

[6] Sanjai Rayadurgam and Mats PE Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings of the Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 83–91, 2001.

[7] DO-178C, software considerations in airborne systems and equipment certification. RTCA, 2011.

[8] John Joseph Chilenski and Steven P Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[9] Ajitha Rajan, Michael W Whalen, and Mats Per Erik Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *Proceedings of the 30th International Conference on Software Engineering*, pages 161–170, 2008.

[10] Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer. An observability-based code coverage metric for functional simulation. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, pages 418–425, 1996.

[11] 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.

[12] Gregory Gay, Matt Staats, Michael Whalen, and Mats Heimdahl. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*, 41(8):803–819, 2015.

[13] Dongjiang You, Sanjai Rayadurgam, Michael Whalen, Mats PE Heimdahl, and Gregory Gay. Efficient observability-based test generation by dynamic symbolic execution. In *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering*, pages 228–238, 2015.

[14] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[15] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th International Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

[16] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In

Proceedings of the 5th European Dependable Computing Conference, pages 281–292, 2005.

[17] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423, 2006.

[18] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[19] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs*, pages 134–153, 2008.

[20] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, pages 151–166, 2008.

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

[22] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *Automata, Languages and Programming*, pages 322–335, 1990.

[23] 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.

[24] 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.

[25] Dongjiang You, Sanjai Rayadurgam, Mats PE Heimdahl, John Komp, BaekGyu Kim, and Oleg Sokolsky. Executing model-based tests on platform-specific implementations. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 418–428, 2015.

[26] John J Chilenski. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Technical report, DTIC Document, 2001.

[27] Kelly J Hayhurst, Dan S Veerhusen, John J Chilenski, and Leanna K Rierson. A practical tutorial on modified condition/decision coverage. Technical report, NASA Langley Research Center, 2001.

[28] Matt Staats, Gregory Gay, and Mats PE Heimdahl. Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 870–880, 2012.

[29] Matt Staats. The influence of multiple artifacts on the effectiveness of software testing. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, pages 517–522, 2010.

[30] Marie-Claude Gaudel. Testing can be formal, too. In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 82–96, 1995.

[31] Farzan Fallah, Srinivas Devadas, and Kurt Keutzer. OCCOM-efficient computation of observability-based code coverage metrics for functional verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(8):1003–1015, 2001.

[32] Farzan Fallah, Srinivas Devadas, and Kurt Keutzer. Functional vector generation for HDL models using linear programming and boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(8):994–1002, 2001.

[33] Farzan Fallah, Pranav Ashar, and Srinivas Devadas. Functional vector generation for sequential HDL models under an observability-based code coverage metric.

IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 10(6):919–923, 2002.

[34] Jeffrey M Voas. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, 1992.

[35] Jeffrey Voas. Dynamic testing complexity metric. *Software Quality Journal*, 1(2):101–114, 1992.

[36] Jeffrey M Voas and Keith W Miller. The revealing power of a test case. *Software Testing, Verification and Reliability*, 2(1):25–42, 1992.

[37] Jeffrey M Voas and Keith W Miller. Putting assertions in their place. In *Proceedings of the 5th International Symposium on Software Reliability Engineering*, pages 152–157, 1994.

[38] Wei Chen, Roland H Untch, Gregg Rothermel, Sebastian Elbaum, and Jeffery Von Ronne. Can fault-exposure-potential estimates improve the fault detection abilities of test suites? *Software Testing, Verification and Reliability*, 12(4):197–218, 2002.

[39] Tai-Ying Jiang, Chien-Nan Jimmy Liu, and Jing-Yang Jou. Observability analysis on hdl descriptions for effective functional validation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(8):1509–1521, 2007.

[40] Wes Masri, Andy Podgurski, and David Leon. Detecting and debugging insecure information flows. In *Proceedings of the 15th IEEE International Symposium on Software Reliability Engineering*, pages 198–209, 2004.

[41] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206, 2007.

[42] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[43] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. *ACM SIGPLAN Notices*, 25(6):246–256, 1990.

[44] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[45] David Schuler and Andreas Zeller. Assessing oracle quality with checked coverage. In *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation*, pages 90–99, 2011.

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

[47] SCADE Suite. `http://www.esterel-technologies.com/products/scade-suite/`.

[48] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[49] Abdesselam Lakehal and Ioannis Parissis. Structural test coverage criteria for Lustre programs. In *Proceedings of the 10th International Workshop on Formal Methods for Industrial Critical Systems*, pages 35–43, 2005.

[50] Virginia Papailiopoulou, Laya Madani, Lydie du Bousquet, and Ioannis Parissis. Extending structural test coverage criteria for lustre programs with multi-clock operators. In *Formal Methods for Industrial Critical Systems*, pages 23–36, 2008.

[51] Virginia Papailiopoulou, Ajitha Rajan, and Ioannis Parissis. Structural test coverage criteria for integration testing of LUSTRE/SCADE programs. In *Formal Methods for Industrial Critical Systems*, pages 85–101, 2011.

[52] Virginia Papailiopoulou. Automatic test generation for LUSTRE/SCADE programs. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 517–520, 2008.

[53] Eunkyoung Jee, Junbeom Yoo, Sungdeok Cha, and Doohwan Bae. A data flow-based structural testing technique for FBD programs. *Information and Software Technology*, 51(7):1131–1139, 2009.

[54] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[55] William E Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, SE-3(4):266–278, 1977.

[56] Corina S Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*, 11(4):339–353, 2009.

[57] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[58] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, 2007.

[59] Grégoire Hamon, Leonardo De Moura, and John Rushby. Generating efficient test sets with a model checker. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods*, pages 261–270, 2004.

[60] Gordon Fraser and Angelo Gargantini. Experiments on the test case length in specification based test case generation. In *Proceedings of the 2009 ICSE Workshop on Automation of Software Test*, pages 18–26, 2009.

[61] 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.

[62] Dongjiang You, Isaac Amundson, Scott A Hareland, and Sanjai Rayadurgam. Practical aspects of building a constrained random test framework for safety-critical embedded systems. In *Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation*, pages 17–25, 2014.

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

[64] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In *Modeling and Verification of Parallel Processes*, pages 187–195, 2001.

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

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[72] Grigore Rou and Traian Florin erbănută. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

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

[74] 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.

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

[76] MathWorks MATLAB Coder. `http://www.mathworks.com/products/matlab-coder/`.

[77] JKind. `https://github.com/agacek/jkind`.

[78] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[79] Matt Staats, Michael W Whalen, and Mats Per Erik Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 391–400, 2011.

[80] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 193–207, 1999.

[81] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 127–144, 2000.

[82] Niklas Een, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *Proceedings of the 2011 International Conference on Formal Methods in Computer-Aided Design*, pages 125–134, 2011.

[83] W Eric Wong, Joseph R Horgan, Saul London, and Aditya P Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineering*, pages 41–50, 1995.

[84] Gregg Rothermel, Mary Jean Harrold, Jeffery Von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 12(4):219–249, 2002.

[85] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[86] 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.

[87] Ajitha Rajan, Michael Whalen, Matt Staats, and Mats PE Heimdahl. Requirements coverage as an adequacy measure for conformance testing. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, pages 86–104, 2008.

[88] Gregory Gay, Matt Staats, Michael Whalen, and Mats PE Heimdahl. Automated oracle data selection support. *IEEE Transactions on Software Engineering*, 41(11):1119–1137, 2015.

[89] 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 *Proceedings of the 7th NASA Formal Methods Symposium*, pages 279–294, 2015.

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

[91] Yue Jia and Mark Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Testing: Academic & Industrial Conference – Practice and Research Techniques*, pages 94–98, 2008.

[92] Hiralal Agrawal, Richard DeMillo, R_ Hathaway, William Hsu, Wynne Hsu, Edward Krauser, Rhonda J Martin, Aditya Mathur, and Eugene Spafford. Design of

mutant operators for the C programming language. Technical report, SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[93] 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.

[94] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pages 435–445, 2014.

[95] Akbar Siami Namin and James H Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 57–68, 2009.

Appendix A

Source Code and Models

In order to benefit the community of software testing research, we have implemented our approaches as an open source framework as well as made certain case example systems public under the appropriate license.

- The set of tools for the Lustre programming language can be obtained from
  `https://github.com/djyou/lustre`
  This toolset is developed based on the JKind model checker [77]. Specifically, it supports static program analysis, coverage analysis and measurement, test case generation, test suite reduction, program simulation, and program optimization.

- The PCA (Patient-Controlled Analgesia) infusion pump system used in this dissertation can be obtained from
  `http://crisys.cs.umn.edu/gpca.shtml`
  The infusion pump system project contains the subsystems (i.e., *Alarm* and *Infusion Manager*) we used in observability-based test generation and the entire infusion pump system we used in the automated framework to execute model-based tests on platform-specific implementations.

Appendix B

Publications

The research reported in this dissertation has resulted in the following publications:

- Dongjiang You, Sanjai Rayadurgam, Mats Heimdahl, John Komp, BaekGyu Kim, and Oleg Sokolsky. **Executing Model-based Tests on Platform-specific Implementations.** In *Proceedings of the 30th International Conference on Automated Software Engineering*, pages 418–428, 2015.

- Dongjiang You, Sanjai Rayadurgam, Michael Whalen, Mats Heimdahl, and Gregory Gay. **Efficient Observability-based Test Generation by Dynamic Symbolic Execution.** In *Proceedings of the 26th International Symposium on Software Reliability Engineering*, pages 228–238, 2015.

- Dongjiang You, Isaac Amundson, Scott Hareland, and Sanjai Rayadurgam. **Practical Aspects of Building a Constrained Random Test Framework for Safety-critical Embedded Systems.** In *Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation*, in conjunction with *the 36th International Conference on Software Engineering*, pages 17–25, 2014.

- Michael Whalen, Gregory Gay, Dongjiang You, Mats Heimdahl, and Matt Staats. **Observable Modified Condition/Decision Coverage.** In *Proceedings of the 35th International Conference on Software Engineering*, pages 102–111, 2013.