Ensuring the Observability of Structural Test Obligations

Ying Meng, Gregory Gay, *Member, IEEE*, Michael Whalen, *Senior Member, IEEE*

Abstract—Test adequacy criteria are widely used to guide test creation. However, many of these criteria are sensitive to statement structure or the choice of test oracle. This is because such criteria ensure that execution *reaches* the element of interest, but impose no constraints on the execution path *after* this point. We are not guaranteed to observe a failure just because a fault is triggered. To address this issue, we have proposed the concept of *observability*—an extension to coverage criteria based on Boolean expressions that combines the obligations of a host criterion with an additional path condition that increases the likelihood that a fault encountered will propagate to a monitored variable.

Our study, conducted over five industrial systems and an additional forty open-source systems, has revealed that adding observability tends to improve efficacy over satisfaction of the traditional criteria, with average improvements of 125.98% in mutation detection with the common output-only test oracle and per-model improvements of up to 1760.52%. Ultimately, there is merit to our hypothesis—observability reduces sensitivity to the choice of oracle and to the program structure.

Index Terms—Software Testing, Automated Test Generation, Test Adequacy Criteria, Model-Based Test Generation

1 Introduction

Test adequacy criteria defined over program structures—such as statement, branches, or atomic conditions—are widely used as measures to assess the efficacy of test suites. Such criteria are essential in offering guidance in the testing process, as they offer clear checklists of goals—called *test obligations*—to testers and the means to automate the creation of test suites. However, many of these criteria are highly sensitive to how statements are structured [1], [2] or the choice of test oracle—the artifact used to judge program correctness [3]–[5].

Consider the Modified Condition/Decision Coverage (MC/DC) coverage criterion [6]. MC/DC is used as an exit criterion when testing software in the avionics domain. For certification, a vendor must demonstrate that the test suite provides MC/DC coverage of the source code [7]. However, the efficacy of test suites created to satisfy MC/DC—particularly when test suite creation is automated—is highly dependent on the syntactic structure of the code under test. A complex Boolean expression, for example, could be written as a series of simple expressions, or as a single *inlined* expression. This simple transformation can dramatically improve the efficacy of MC/DC-satisfying test suites, increasing fault detection efficacy by orders of magnitude [1].

Such results are worrying, particularly given the importance of coverage criteria in safety certification, and the improvements made in terms of automated generation. When examining the discrepancy in efficacy between test suites for non-inlined and inlined programs, we often found that the test case encountered a fault in the code—such as an erroneous Boolean operator—leading to a corrupted internal state. However, this corruption was *masked out* in a subsequent expression, and did not propagate to an output. This effect was prevalent in programs containing large numbers of simple Boolean expressions stored in local variables. Even in cases where a non-masking path could theoretically be found, it was common for a test case to not allow sufficient execution time for corrupted state to propagate.

This sensitivity to structure and choice of oracle is caused by the fact that the obligations of structural coverage criteria are only posed over specific syntactic elements—statements, branches, conditions. Such obligations ensure that execution *reaches* the element of interest, and exercises it in the prescribed manner. However, no constraints are imposed on the execution path *after* this point. We are not guaranteed to observe a failure just because a fault is triggered.

To address this issue, we have proposed the concept of *observability*—an extension to coverage criteria based on Boolean expressions that has the potential to eliminate masking. Observable coverage criteria combine the test obligations of their host criterion with an additional path condition that increases the likelihood that a fault encountered when executing the element of interest will propagate to a variable monitored and checked for correctness by the test oracle. Unlike many extensions to coverage criteria [8], this path condition does not increase the number of test obligations over its host criterion. Instead, it makes the existing obligations more stringent to satisfy, as the possibility of propagating a

Y. Meng and G. Gay are with the Department of Computer Science & Engineering, University of South Carolina. E-Mail: ymeng@email.sc.edu, greg@greggay.com

M. Whalen is with the Department of Computer Science and Engineering, University of Minnesota. E-Mail: mwwhalen@umn.edu

This work has been partially supported by NSF grant CCF-1657299. We also thank the Advanced Technology Center at Rockwell Collins Inc. for granting access to industrial case examples.

fault revealed by the original obligation must also be demonstrated. We hypothesize that observability will improve the effectiveness of the host criterion—no matter which criterion is chosen—particularly when used as a test generation target, paired with test oracles that only examine the values of output variables.

Focusing on safety-critical applications, we have implemented observable versions of Branch, Condition, Decision, and MC/DC Coverage as part of model-based test generation for the Lustre dataflow language [9]. While our implementation is for dataflow languages, the *concept* of observability is not restricted to any one language, generation paradigm, or product domain, and our semantic model should be valid for imperative languages such as C or Java.

This work is an extension of our prior work defining and exploring the concept of observability [10]–[12]. We first proposed the concept of observability as an extension of the MC/DC coverage criterion [10]. An extended study found that OMC/DC was more effective—and overcame many of the weaknesses of—traditional coverage criteria for a small set of studied systems [11]. We extend previous efforts by decoupling the notion of observability from MC/DC and exploring its application as a generic addition to any coverage criterion. This decoupling allows us to explore the impact of the choice of host criterion, and to explore the efficacy of observability as a general extension to adequacy criteria. Our new experimental studies also consider a wider range of programs than previously explored in order to better understand the efficacy of observability-based coverage criteria as the target of automated generation.

Our study, conducted over five industrial systems from Rockwell Collins and an additional forty open-source systems, has revealed the following insights:

> Test suites satisfying Observable MC/DC are generally the most effective, killing 95.61% of mutants on average (maximum oracle strategy)/87.03% (output-only oracle strategy) for the inlined Rockwell models, 98.85% (maximum)/85.88% (output-only) for the non-inlined Rockwell models, and 89.62% (maximum)/65.14% (output-only) for the Benchmarks models.
>
> Adding observability tends to improve efficacy over satisfaction of traditional criteria, with average improvements of up to 392.44% in mutation detection and per-model improvements of up to 1654.38%.
>
> Factors that can harm efficacy—generally resulting in a reduction in the number of fulfilled obligations—include expression complexity, the length of the combinatorial path from expression to output, and the length of the delayed path from expression to output.
>
> The addition of observability requires a longer test generation process, with average increases ranging from 129.39%-2422.91%. However, this increase tends to be relatively small—seconds to minutes.
>
> The addition of observability results in an increase in the size of test suites. The magnitude of that increase depends on the length of the paths from each expression to the output.
>
> The addition of observability results in an decrease in the number of fulfilled obligations. This loss is due to either the discovery of dead code that cannot influence the output or obligations that are too complex for the test generator to solve.
>
> The choice of host criterion influences efficacy, but the largest increase in complexity comes from the addition of observability. Varying both criterion and observability may allow testers to find an optimal level of efficacy and complexity.
>
> Observability reduces sensitivity to the choice of oracle, by ensuring a masking-free path from expression to the variables monitored by the test oracle.
>
> Observability reduces sensitivity to the program structure by capturing the complexity benefits of inlining in the path from expression to output.

Based on our results, *observability* is a valuable extension—regardless of the chosen host criterion—to coverage criteria for dataflow languages. Requiring observability increases test efficacy and produces suites that are robust to changes in the structure of program or the variables being monitored by the test oracle.

The remainder of this article is structured as follows. Section 2 introduces important background material. Section 3 presents the concept of observability and offers formal definitions and implementation details. Section 4 presents the details of our experiments, and Section 5 discusses our observations. Section 6 discusses threats to validity. Section 7 presents related work. Finally, Section 8 summarizes and concludes the manuscript.

2 Background

In this research, we are interested in improvements to the criteria used to judge the adequacy of testing efforts—and to automatically generate test suites. In particular, we are focused on the safety-critical reactive systems that power our society. In this section, we will discuss background material on both topics.

2.1 Adequacy Criteria

The concept of adequacy is important in providing developers with the guidance needed to test effectively. As we cannot know what faults exist without verification, and as testing cannot—except in simple cases—conclusively prove the absence of faults, a suitable approximation must be used to measure the adequacy of our testing efforts. If existing tests have not surfaced any faults, is the software correct, or are the tests *inadequate*?

The most common methods of measuring adequacy involve coverage of structural elements of code, such as branches of control, and Boolean expressions [13]–[15]. Each adequacy criterion prescribes a series of *test*

obligations (or requirements) that tests must fulfill. For example, branch coverage requires that all outcomes of expressions that can result in different code segments being executed—such as if-then-else and loop conditions—be executed. The idea of measuring adequacy through coverage is simple, but compelling: unless code is executed, many faults are unlikely to be found. If tests execute elements as prescribed by the criterion, then testing is deemed "adequate" with respect to faults that manifest through such structures.

Adequacy criteria have seen widespread use, as they offer objective, measurable checklists [16] and—importantly—*stopping criteria* for the testing process. For that same reason, they are ideal as test generation targets [17]–[19], as coverage can be straightforwardly measured and optimized for [20].

2.2 Structural Coverage

Structural coverage criteria serve as a means to determine that the structure of system under test—the various elements making up the source code—have been thoroughly exercised by test cases. Many structural coverage criteria, defined with respect to specific syntactic elements of a program, have been proposed and studied over the past decades [11], [21]. These have been used to measure suite adequacy—as a means to assess the quality of existing test suites, and whether developers can stop adding tests. They are also commonly used as as targets for automated test generation.

In this study, we are primarily concerned with reactive systems—safety-critical embedded systems that interact with the physical world. Such systems often have sophisticated logical structures in the code. Therefore, in this work, we are primarily concerned with structural coverage criteria defined over Boolean expressions. In particular, we are focused on Condition Coverage, Branch Coverage, Decision Coverage, and Modified Condition/Decision Coverage (MC/DC).

Decision Coverage: A decision is a Boolean expression. Decisions are composed of one or more conditions—atomic Boolean subexpressions—connected by operators (`and`, `or`, `xor`, `not`). Decision Coverage requires that all decisions in the system under test evaluate to both the `true` and `false`. Given the expression `((a and b) and (not c or d))`, tests would need to be produced where the expression evaluates to true and the expression evaluated to false. In this case, the test input `(TTTT),(TTTF)` would satisfy Decision Coverage.

Branch Coverage: A branch is a particular type of decision that can cause program execution to diverge down a particular control flow path, such as that in `if` or `case` statements. Branch Coverage is defined in the same manner as Decision Coverage, but is restricted to branches, rather than all decision statements. Branch Coverage is arguably the most commonly used coverage criterion, with ample tool support[1] and industrial adoption. Improving Branch Coverage is a common goal in automated test generation [18], [22].

Condition Coverage: A condition is an atomic Boolean subexpression within the broader decision. Condition Coverage requires that each condition evaluate to `true` and `false`. Given the expression `((a and b) and (not c or d))`, achieving Condition Coverage requires tests where the individual atomic Boolean conditions `a`, `b`, `c`, and `d` evaluate to `true` and `false`. For this decision, test input `(TTTF),(FFFT)` would satisfy the obligations of Condition Coverage.

Note that satisfying the obligations of one form of coverage does not always imply that the obligations of others are fulfilled as well. The test input given above would satisfy Condition Coverage, but not Decision Coverage, as both test inputs result in the decision evaluating to `false`. Similarly, the input provided earlier for Decision Coverage—`(TTTT),(TTTF)`—would not satisfy Condition Coverage, as only `d` evaluates to both outcomes. Therefore, stronger criteria—such as Modified Condition/Decision Coverage—require that the obligations of both Decision and Condition Coverage be met.

Modified Condition/Decision Coverage (MC/DC): The MC/DC criterion is used as an exit criterion when testing software for critical software in the avionics domain, and is required for safety certification in that domain [23]. MC/DC further strengthens Condition and Decision Coverage by requiring that each decision evaluate to all possible outcomes, each condition take on all possible outcomes, and that each condition be shown to independently impact the outcome of the decision.

Independent effect is defined in terms of *masking*—a masked condition has no effect on the value of the decision; for example, given a decision of the form `x and y`, the truth value of `x` is irrelevant if `y` is false, so we state that `x` is masked out. A condition that is not masked out has *independent effect* for the decision.

Consider again the expression `((a and b) and (not c or d))`. Suppose we examine the independent affect of `d` in the example; if `(a and b)` evaluates to false, than the entire decision will evaluate to false, masking the effect of `d`; Similarly, if `c` evaluates to false, then `(not c or d)` evaluates to true regardless of the value of `d`. Only if we assign `a`, `b`, and `c` the value of true does the value of `d` affect the outcome of the decision. Showing independent impact requires a pair of test cases where all other conditions hold fixed values and our condition of interest flips values. If changing the value of the condition of interest changes the value of the decision as a whole, then the independent impact has been shown. In this example, the test inputs `(TTTT)`, `(TTTF)`, `(FTTT)`, `(TFTT)`, and `(TTFF)` satisfies MC/DC. Tests inputs 1 and 3 show the effect of `a`, 1 and 4 show `b`, 2 and 5 show `c`, and 1 and 2 show `d`.

1. Such as the Cobertura and EMMA IDE plug-ins—see http://cobertura.github.io/cobertura/ and http://www.eclemma.org/

MC/DC can be satisfied in ($number\ of\ conditions$ + 1) test cases if care is taken in selecting test input.

Because both decisions and conditions are covered, we state that MC/DC *subsumes* the previously-defined forms of coverage. Satisfying the obligations of MC/DC also satisfies the obligations of Decision and Condition Coverage. This comes at a cost—satisfying MC/DC requires more test cases and more effort than satisfying any of the above criteria. Therefore, if no benefit is perceived from the additional requirements of MC/DC, testers often elect to satisfy a simpler criterion instead.

Several variations of MC/DC exist—for this study, we use Masking MC/DC, as it is a common criterion within the avionics community [24].

2.3 Mutations and Mutation Coverage

Mutation [25] is a technique in which a user generates many faulty implementations through small modifications of the original implementation, either through automated code transformation or by hand [26], [27]. Usually a single modification is made to each *mutant* implementation, such as changing a single expression (substituting addition for subtraction , e.g.), permuting the order of two statements, or many other possible changes. The mutations introduced generally match one or more models of the types of mistakes that real developers make when building code. Generally, mutants are introduced with the intent that they not be trivially detected—they are both syntactically and semantically valid [15]. That is, the mutants will compile, and no mutant will crash the system.

Mutations can be used to assess the effectiveness of a test suite by examining how many mutants are *killed* (that is, detected) by the tests within the test suite. Detection of mutants has also been the basis of multiple adequacy criteria [28], [29]. In theory, if a suite detects more mutants, it will also be more adequate at fault detection. To kill a mutant using *strong mutation*, the following conditions must be met [30]:

(R) the test must *reach* the mutation.
(I) the test must *infect* program state by causing it to differ between the original and mutated program.
(P) incorrect state must *propagate* to program output.
(R) the test oracle must *reveal* the difference.

In strong mutation coverage, the resulting corruption must influence an output variable. *Weak mutation* only requires the (R,I) steps. In weak mutation coverage, a mutant is considered detected if the mutated statement is reached, and the value of *that expression* is corrupted [31]. *Firm mutation* requires propagation to some point of observation in the system, but not necessarily an output [32]. For each of the metrics, a *mutation coverage* score can be determined by dividing the number of killed mutants by the number of all mutants.

2.4 Reactive Systems and Dataflow Languages

Increasingly, our society is powered by sophisticated software systems—such systems manage factories and power plants, coordinate the many systems driving automobiles and airplanes, and even make life-saving decisions as part of medical devices implanted in human bodies. Many of these systems are what we refer to as *reactive systems*—-embedded systems that interact with physical processes. Reactive systems operate in cycles—receiving new input from their environment, to which they react by issuing output.

Such systems are commonly designed using modeling languages, which are translated into C code that can be directly flashed to hardware. Models can be developed using visual notations, such as Simulink [33], Stateflow [34] and SCADE [35]. They can also be directly expressed using *dataflow languages*, such as Lustre.

Lustre is a synchronous dataflow language used in a number of domains to model or directly implement embedded systems [9]. It is a declarative programming language for manipulating *data flows*—infinite streams of variable values. These variables correspond to traditional data types, such as integers, booleans, and floating point numbers. Lustre offers an intermediate representation between behavioral model and traditional source code that is useful for specification, design, and analysis purposes. Because of the simplicity and declarative nature of Lustre, it is well-suited to model checking and verification, in particular with regards to its safety properties [36]. Lustre programs can be automatically generated from visual notations such as Simulink, and can be automatically compiled to target languages such as C/C++, VHDL, as well as to input models for verification tools such as model checkers.

A Lustre program is structured into a network of control modules (nodes) that specify relations between inputs and outputs of a system. A node specifies a stream transformer, mapping streams of input variables to streams of internal and output variables using a set of defined expressions. Lustre nodes have cyclic behavior—at execution cycle i, the node takes in the values of the input streams at instant i, manipulates those values, and issues new values for the internal and output variables. Nodes have a limited form of memory, and can access input, internal, and output values from previous instants (up to a statically-determined finite limit). To update program state within one computational step, combinatorial variables are used; to store current program state for the reference by later cycle or cycles, delay variables are used (i.e., $\frac{1}{z}$ blocks in Simulink). During a cycle, all variables are calculated according to their definitions: combinatorial variables are computed combinatorially using values at the current computational step, and delay variables are computed combinatorially using values from previous step or steps.

The body of a Lustre node consists of a set of equations of the form $x = expr$ where x is a variable identifier, and $expr$ is the expression defining the value of x at instant i. Like in most programming languages, expression t can make use of any of the other input, internal, or output variables in defining x—as long as that variable

```
v1 = (0 -> (pre in1));
v2 = (v1 > 1);
v3 = (false -> (pre v2));
out = (if in2 then v2 else v3);
```

Fig. 1: Sample Lustre code fragment

has already been assigned a value during the current cycle of computation. The order of equations does not matter in Lustre, except for data dependencies. That is, within a computational step, as long as all the variables involved in an equation have already been computed, the equation can be evaluated.

Lustre supports many of the traditional numerical and boolean operators, including +, , , =, <, >, %, etc. Lustre also supports two important *temporal* operators: *pre*(*x*) and ! . The *pre*(*x*) operator, or "previous", evaluates to the value of *x* at instant (*i* 1). The ! operator, or "followed by", allows initialization of variables in the first instant of execution. For example, the expression *x* = 0 ! *pre*(*x*) + 1 defines the value of *x* to be 0 in instant 0, then defines it as 1 at instant 1—or, the value at instant 0 plus one—and so forth.

For example, consider the code fragment in Figure 1, in which *in1* and *in2* are input variables, *v1*, *v2*, and *v3* are internal variables, and *out* is an output variable. Variables *in1* and *v1* are type of int and all the rests are type of boolean. Variables *in1* and *v2* are delay variables, values stored in them will be used by *v1* and *v3* in the next cycle, respectively. Variable *v1* is initially assigned to *0* followed by (represented by operator *arrow*) *in1*'s value from the previous cycle, at each subsequent cycle. Similarly, values of variable *v3* is a stream of boolean values, which starts with a *false* followed by *v2*'s value from the previous computational step.

2.4.1 Test Case Structure for Reactive Systems

There are two key artifacts necessary to construct a test case, the *test inputs, or test data*—inputs given to the system under test—and the *test oracle*—a judge on the resulting execution [37], [38]. A *test oracle* can be defined as a predicate on a sequence of stimuli to and reactions from the SUT that judges the resulting behavior according to some specification of correctness [39].

As reactive systems compute in cycles, multiple test inputs must generally be provided. Therefore, tests are divided into a series of *test steps*, where input and expected output is provided for *each step*. In each step, specific values are given for each input variable, then the internal and output variables are computed accordingly. The output at each step is compared to the expected output provided as part of the test oracle. Table 1 shows example test input that contains four steps together with corresponding evaluations of all internal and output variables. From this example, we can see how the values of delay variables impact other variables.

TABLE 1: Sample Lustre Program Evaluation

step	inputs (in1, in2)	internals (v1, v2, v3)	outputs (out)
1	(1, T)	(0, F, F)	(F)
2	(2, T)	(1, F, F)	(F)
3	(3, F)	(2, T, F)	(F)
4	(4, F)	(3, T, T)	(T)

3 Observability-Based Test Creation

In this chapter, we will illustrate the common issue impacting the efficacy of test suites generated to satisfy structural coverage criteria—*masking*—and formally define our solution—*observability* [10]. We then will describe how extending common structural coverage criteria to require observability can overcome masking, and consequently, sensitivity to program structure and oracle. Finally, we will describe how we implemented our tool to generate test obligations for observability-based coverage criteria.

3.1 Masking

Previous research has shown that the efficacy of test suites satisfying structural coverage criteria—defined over specific program elements such as control-flow branches, conditions, or decisions—can be highly sensitive to how expressions are written [1], [2], [11] and the selection of variables monitored by the test oracle [3]–[5]. This is due to *masking*, when the value of a variable or subexpression is prevented from influencing the outcome of another expression. In other words, masking prevents the *propagation* of this effect, in the sense of mutation, to a program output.

In this work, we are primarily concerned with masking in terms of Boolean expressions. Masking occurs when the value of a condition (an atomic variable or subexpression) in a Boolean decision hides the effects of other conditions. We state a condition is masked if the outcome of a Boolean decision cannot be changed by varying the value of the condition while holding the rest of the conditions fixed (i.e., no matter what value the condition is, the final outcome of syntactic element of interest does not change). For example, input `a = true` masks `b` in the decision `(a or b)`. As the decision's outcome is always `true`, regardless of the value of `b`, `a` is masked. Similarly, `a = false` masks `b` in decision `(a and b)`, as the decision will always evaluate to `false`.

By requiring that each condition demonstrate an independent influence on its decision's outcome, MC/DC is designed to prevent masking *within* an expression. Test cases must exist where, if we flip the value of a single condition while the others are held constant, the outcome of that decision must be changed. Branch, Decision, and Condition Coverage lack any such guarantee. This is one reason MC/DC is often required for testing of avionics and other safety critical systems—its requirements are more strenuous, but the additional assurances of the independent impact requirement theoretically increase the probability that logic faults will be detected.

```
1. v1 = in1 or in2;
2. out = v1 and in3;
```

Fig. 2: Non-inlined sample code

```
1. out = ((in1 or in2) and in3);
```

Fig. 3: Inlined sample code

However, how the code is structured has a major impact on the formulation of the test obligations—the prescribed goals—for a criterion and the efficacy of the suites satisfying such obligations. Consider the code fragments in Figures 2 and 3. The two code fragments are *semantically* identical—they offer the same outcome—but are written in two different styles. The fragment in Figure 2 is split over two separate, simple equations (a *non-inlined* style). The fragment in Figure 3 is *inlined*—written as a single, complex expression.

As the obligations for criteria such as MC/DC are posed over individual program elements, the MC/DC obligations for the non-inlined version will be much simpler—and more trivially satisfied—than obligations for the inlined version. In the non-inlined version, for example, `in1` must be shown to overcome any masking from the value of `in2`. However, in the inlined version, `in1` must overcome masking from both `in2` and `in3`. As a result, MC/DC is much harder to satisfy over inlined implementations, and requires a larger number of test cases. The produced test suites tend to be far more effective [1], [2]. Therefore, we can see that traditional coverage criteria are sensitive to program structure.

Further, just because a condition is shown to influence the outcome of the decision it resides within, there is no assurance that the condition will influence the *program output*. Consider again the sample code fragment in Figure 2. Based on the definition of MC/DC, `TestSuite1` in Table 2 provides MC/DC over the program fragment in Figure 2; the test cases with `in3` = false (bold faced) contribute towards MC/DC of `in1` or `in2` in `v1`. Nevertheless, if we monitor 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` gives us MC/DC coverage of the non-inlined program fragment, but a fault on the first line will never propagate to the output. On the other hand, `TestSuite2` will also give MC/DC coverage of the program, but since `in3` = true in the first two test cases, faults in the first statement can propagate to an output.

Because coverage obligations are posed over individual program elements, and make no demands on what happens after that element is executed, masking can prevent triggered faults from being observed. Masking can prevent an infection from propagating, hindering the oracle's ability to reveal a fault.

Masking can be partially mitigated through selection of the correct oracle strategy. For instance, by monitoring all internal state variables as well as all the outputs, masking between statements is not an issue [3], [5], [40]. In the case of Figure 2, if we monitor the value of *v1* during testing, failures introduced by `in1` or `in2` can be detected without changing test suites. However, monitoring and specifying expected values for all variables is generally prohibitively expensive (or outright infeasible). A subset of variables could be used, if carefully chosen, but this selection is also non-trivial to make.

TABLE 2: Sample test suites satisfying MC/DC for the code in Figures 2-3

TestSuite1 =	*f* **(T, F, F)**, **(F, T, F)**, (F, F, T), (T, T, T)*g*
TestSuite2 =	*f*(T, F, T), (F, T, T), (F, F, T), (T, F, F)*g*

An alternative approach is to strengthen the coverage criteria with conditions on execution along the path from the program element of interest to the output (or other chosen oracle variables). Such path conditions can ensure the *observability* of such elements when we test.

3.2 Observability

The observability of a program is the degree to which it is possible to infer the internal state of a system given the information that we can monitor from the program—generally through program output [5]. We say an expression in a program is *observable* in a test case if we can change only the expression's value—keeping the rest of the program fixed—and see the influence of this change in the result of the test case. Otherwise, if this update has no visible influence, we say the expression is *not observable* in that test case. As we will see, observability is closely related to strong mutation.

For example, consider the program fragment in Figure 2. We could replace the expression defining variable `v1` with a fixed value of `false`. Normally, execution of the first test case in `TestSuite1` in Table 2 would cause `v1` to evaluate to `true`. However, as the effect of executing `v1` is masked by `in3` in expression `out`, we would not notice the substitution—we lack *observability* when executing this test case. On the other hand, if we executed the first test case from `TestSuite2` instead, we would detect the substitution. In that case, we can establish observability.

In theory, masking can be overcome by requiring observability from a test suite—in addition to the existing test obligations of a host coverage criterion. Informally, we can obtain observability of test obligations by requiring that the variable whose assignment contains a particular element of interest remains unmasked through a path to a variable monitored by the test oracle.

Although this notion of observability was previously defined as an explicit extension to MC/DC [10], such requirements can be imposed on any existing criterion over Boolean expressions. The path conditions of observability establish a masking-clear path from an expression containing a program element of interest—one with obligations defined over it—to a monitored variable. In this study, we apply observability to Branch, Condition, Decision, and MC/DC Coverage.

To formally define how observability is established, we can view a deterministic program P containing expression e as a transformer from inputs I to outputs $O : P : I \to O$. 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 a substitution. Rather, we replace a single instance of expression e rather than all instances, which is more akin to mutation. We state e is observable in test t if $\exists v{:}P(t) \not\equiv P[v{=}e_n](t)$.

This formulation is a generalization of the semantic idea behind masking MC/DC [24], lifted from decisions to programs. In masking MC/DC, the main obligation is that, for each condition c in given decision D, there are a pair of test cases t_i and t_j ensuring that c is observable in D's outcome for both outcomes (`true` and `false`): $((D(t_i) \not\equiv D[true{=}c](t_i)) \wedge ((D(t_j) \not\equiv D[false{=}c](t_j)))$.

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

$$\begin{aligned}&(\forall c \in Cond(P)):\\ &\quad((\exists t \in T:\ (P(t) \not\equiv P[true{=}c](t)))\\ &\quad\wedge(\exists t \in T:\ (P(t) \not\equiv P[false{=}c](t))))\end{aligned} \tag{1}$$

where Cond(P) is the set of all conditions in program P. This formula can be straightforwardly generalized from conditions to several different program structures: decisions to perform Observable Decision Coverage, branches for Observable Branch Coverage, or Boolean variable assignments if we wish to pair Observability with Condition Coverage.

There are strong connections between MC/DC, observability, and mutation testing. From the definition above, it is clear that masking MC/DC corresponds to weak or firm mutation (depending on whether or not a decision is observable) for mutations that replace each condition with constant *true* and *false*. Similarly, Observable MC/DC corresponds to strong mutation for these mutants. Observability offers the means to ensure a path from an expression to the program output, ensuring that the effect of a fault is detected. By explicitly defining a path constraint, Observable criteria offer feedback to the test generation process. In fact, one could use the same path constraints along with Boolean-based mutation operators to satisfy a subset of strong mutation.

This relationship shows a connection between equivalent mutants and "dead code". If an MC/DC (resp. Observable MC/DC) obligation cannot be satisfied, it means that the expression can be simplified in such a way that the program behaves equivalently, as has been examined in the literature on vacuity [41].

3.3 Tagged Semantics

The semantic definition for observability, defined above, is unwieldy for test generation and test measurement. The analysis would require two versions of the program running in parallel to check that the results match. Then, for test measurement, the test suite must be executed separately for *each pair* of modified programs.

In order to define an observability constraint that efficiently supports monitoring and test generation, we can approximate semantic observability using a tagged semantics approach [42]. Each variable corresponding to a Boolean expression or atomic value in the program is assigned a tag, the observability of which is tracked through the execution of the a program. If a tag is propagated to the output—or any "monitored" internal variable—the corresponding path condition is considered to be fulfilled. More precisely, we track pairings of tag and concrete outcome. If a tagged variable appears more than once in a decision, a tag is assigned to each occurrence uniquely. We then examine the number of all possible pairs that have reached as output in some test in order to evaluate the coverage level for a test suite.

Formal tagging semantics have been defined for a set of expressions, an imperative command language, as well as a simple dataflow language (shown in Table 3). A reduction semantics with evaluation contexts (RSEC) [43] is used for presentation, and the K tool suite [44] is used to check for consistency. The rules, which run over *configurations* containing *K* (the syntax being evaluated) and a set of configuration parameters being labeled, operate by applying rewrites at positions in syntax where the evaluation context allow. A *context* can be a program or program fragment with a *hole* (represented by $\square$)—a placeholder where a rewrite can occur. In their definition, maps are assumed to have operations—lookup $\rho(x)$ and update $\rho[x \leftarrow \ldots]$, the empty map $\cdot$, and lists with concatenation $x{:}y$ and cons $elem :: x$, and operators. Additional syntax, which will be formatted as gray background to distinguish from user-level syntax, may be introduced during rewriting.

In Table 3, expressions yield $(Val; TS)$ pairs, where TS is a set of tags, and are evaluated in a context containing environment ρ of type $Env = (id \to (Val \times TS))$. The expressions are standard, except the $tag(E; T)$ which adds a tag to the set of tags associated with the expression e. For any structural coverage, it is assumed that each Boolean variable is wrapped in a tag expression. Masking is defined by operators: 1) and—given $(a\ and\ b)$, for a is not masked out, b has to be *true*, so the tag assigned to a propagates only when b is *true* (and vice-versa); 2) or—given $(a\ or\ b)$, for a is not masked out, b has to be *false*, so the tag assigned to a propagates only if b is *false* (and vice-versa); 3) ite—given $(if\ a\ then\ b\ else\ c)$, for b is not masked out, a must be *true*, therefore b's tag propagates when a is *true*; similarly, c's tag propagates when a is *false*; 4) relation expressions such as $a > b$, a and b are never masked out by each other; these will not be shown in Table 3.

The imperative language semantics define the way tags broadcast through commands: tags need to propagate through all variables assigned in either branch in conditional statements, for the value of a variable can be influenced by not being assigned by the condition.

TABLE 3: Syntax and tagging semantics for imperative and dataflow programs

Expression syntax, context, and semantics:

$E ::=$	$Val \mid Id \mid E\ op\ E \mid not\ E \mid$
	$E\ ?\ E : E \mid tag(E,\ T) \mid$ (Val, TS) $\mid$ addTags(E, TS)
$Context ::=$	$\mid\ Context\ op\ E \mid E\ op\ Context \mid not\ Context \mid$
	$Context\ ?\ E : E \mid addTags(Context,\ TS) \mid$
	$\langle \kappa : Context,\ \epsilon : Env, ... \rangle$
lit	$n \rightarrow (n,\ \varnothing)$
var	$\langle \epsilon : \sigma \rangle [x] \rightarrow \langle \epsilon : \sigma \rangle [(\sigma x)] \quad if\ x \in dom(\sigma)$
op	$(n_0,\ l_0) \quad (n_1,\ l_1) \rightarrow (n_0 \quad n_1,\ l_0 \sqcup l_1)$
and$_1$	$(tt,\ l_0)\ and\ (tt,\ l_1) \rightarrow (tt,\ l_0 \sqcup l_1)$
and$_2$	$(tt,\ l_0)\ and\ (ff,\ l_1) \rightarrow (ff,\ l_1)$
and$_3$	$(ff,\ l_0)\ and\ \_ \rightarrow (ff,\ l_0)$
or$_1$	$(ff,\ l_0)\ and\ (ff,\ l_1) \rightarrow (ff,\ l_0 \sqcup l_1)$
or$_2$	$(ff,\ l_0)\ and\ (tt,\ l_1) \rightarrow (tt,\ l_1)$
or$_3$	$(tt,\ l_0)\ and\ \_ \rightarrow (tt,\ l_0)$
ite$_1$	$(tt,\ l_0)\ ?\ e_t\ :\ e_e \rightarrow addTags(e_t,\ l_0)$
ite$_2$	$(ff,\ l_0)\ ?\ e_t\ :\ e_e \rightarrow addTags(e_e,\ l_0)$
tag	$tag(t,\ (v,\ l)) \rightarrow (v,\ l \sqcup f(t,\ v))$
adt	$addTags((v,\ l_0),\ l_1) \rightarrow (v,\ l_0 \sqcup l_1)$

Imperative command syntax, context, and semantics:

$S ::=$	$skip \mid S;\ S \mid if\ E\ then\ S\ else\ S \mid$
	$Id\ :=\ E \mid while\ E\ do\ S \mid$ end(**List** Id, TS)
$Context ::=$	$\mid\ Id\ :=\ Context \mid if\ Context\ then\ S\ else\ S \mid$
	$Context;\ S \mid \langle \kappa : Context,\ \epsilon : Env,\ C : TS \rangle$
asgn	$\langle \epsilon : \sigma > [x\ :=\ (n,\ l)] \rightarrow < \epsilon : \sigma [x \quad (n,\ l)] \rangle [skip]$
seq	$skip;\ s_2 \rightarrow s_2$
cond$_1$	$\langle C : c \rangle [if\ (tt,\ l)\ then\ s_1\ else\ s_2] \rightarrow$
	$\langle C : c \sqcup l \rangle [s_2;\ end\,(V,\ c)]$
	$where\ V = (Assigned\ s_1)\,.\,(Assigned\ s_2)$
cond$_2$	$\langle C : c \rangle [if\ (ff,\ l)\ then\ s_1\ else\ s_2] \rightarrow$
	$\langle C : c \sqcup l \rangle [s_1;\ end\,(V,\ c)]$
	$where\ V = (Assigned\ s_1)\,.\,(Assigned\ s_2)$
while	$while\,(e)\ s \rightarrow if\,(e)\ then\,(s;\ while\,(e)\ s)\ else\ skip$
endcond$_1$	$\langle C : c' \rangle [end\,(nil,\ c)] \rightarrow \langle C : c \rangle [skip]$
endcond$_2$	$\langle \epsilon : \sigma,\ C : c' \rangle [end\ (x\ ::\ V,\ c)] \rightarrow \langle \epsilon : \sigma',\ C : c' \rangle$
	$[end\,(V,\ c)]\ where\,(\sigma\ x)\ =\ (n,\ l)\ and$
	$\sigma' = \sigma [x \rightarrow (n,\ l \sqcup c')]$
prog	$s \rightarrow \langle \kappa :\ s,\ \epsilon :\ \varnothing,\ C\ :\ \varnothing \rangle$

Dataflow program syntax, context, and semantics:

$EQ ::=$	$Id\ =\ E \mid id\ =\ pre\,(E)$
$Prog ::=$	$(I,\ Env,\ \mathbf{List}\ EQ)$
$Context ::=$	$\mid\ Context;\ \mathbf{List}\ EQ \mid Context\ ::\ \mathbf{List}\ EQ \mid$
	$EQ\ ::\ Context \mid Id\ =\ Context \mid$
	$Id\ =\ pre(Context) \mid \langle \kappa : Context,\ \tau :$
	$\mathbf{List}\ Env,\ O : \mathbf{List}\ Env,\ \epsilon\ :\ Env,\ S\ :\ Env \rangle$
comb	$\langle \epsilon : \sigma \rangle\ eqs_0.\,((x\ =\ (n,\ l))\ ::\ eqs_1) \rightarrow$
	$\langle \epsilon :\ \sigma [x \quad (n,\ l)] \rangle\ eqs_0.eqs_1$
state	$\langle S : \sigma \rangle\ eqs_0.\,((x\ =\ (n,\ l))\ ::\ eqs_1) \rightarrow$
	$\langle S : \sigma [x \rightarrow (n,\ l)] \rangle\ eqs_0.eqs_1$
write	$\langle O : \kappa,\ \epsilon : c \rangle\ nil;\ eqs \rightarrow \langle O : \kappa.[c],\ \epsilon : c \rangle\ eqs$
cycle	$\langle \tau : \sigma_i\ ::\ l,\ \epsilon :\ \_,\ S : \sigma_l \rangle\ eqs \rightarrow$
	$\langle \tau : i,\ \epsilon : (\sigma_i \rightarrow \sigma_l),\ S :\ \varnothing \rangle\ eqs;\ eqs$
prog	$(i,\ s,\ eqs) \rightarrow \langle \tau :\ i,\ O :\ nil,\ S : s,\ \epsilon : \varnothing,\ \kappa :\ eqs \rangle$

$C : TS$ is introduced into the expression configuration to store the set of variable tags. Once a statement has been executed, the tags added to C by conditional statements will be removed. An *end* statement is introduced to implement that—it is appended to clear C and propagate the conditional tags to all variables assigned in the conditional body. A helper function (*Assigned s*) will then return the list of variables assigned in s. Given a program (or program fragment) containing inputs, the rules defined in table 3 will determine the set of tags propagating to output.

Dataflow languages, such as Simulink and SCADE, are popular for model-based development, and assign values to a set of equations in response to periodic inputs. To store system state, state variables ($\frac{1}{z}$ blocks in Simulink) are used. 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.

Note that both the tagging semantics are *optimistically inaccurate* with respect to observability; that is, they may report that certain conditions are observable when they are not. This is easily demonstrated by a code fragment:

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

The semantic model of observability will correctly report that `c` is not observable; it cannot affect the outcome of this code fragment. However, the tagging model propagates the tags of `c` to the assignments in the `then` and `else` branches.

We have implemented observable versions of Branch, Condition, Decision, and MC/DC Coverage as part of model-based test generation for the Lustre dataflow language. However, the *concept* of observability is not restricted to any one language, generation paradigm, or product domain. The semantic model described in this section should be valid for any imperative language, such as C or Java. Ongoing research efforts are in progress to extend these ideas to Java and to assembly language. In addition, Colaco et al. have implemented observability through an extended tag semantics that takes into account certain non-Boolean faults as part of the Scade 6 language [45].

3.4 Implementation: Model-Based Test Generation

In model-based test generation, models are annotated with *trap properties*. A property of interest is negated, then the model checker returns a counterexample—a test input sequence demonstrating that the property can be met. In order to generate tests that meet the conditions

of observability, we need to be able to annotate the program with trap properties that track the tags described above. This is accomplished by conjoining the coverage obligations of the host criterion with a path condition representing the variable in which the test obligation's target resides. Observability can be attained either *immediately*—within the current computational cycle—or *after a delay.* Path conditions must reflect either case. In this section, we describe this annotation for the Lustre dataflow language [9].

3.4.1 Immediate Non-Masking Paths

A variable `x` is observable if a computational path can be found from `x` to a monitored variable `z` in which `x is not masked.` If such a path can be taken entirely within one computational step, we call it an *immediate non-masking path*, and variable `x` is *immediately observable*. Such paths can be defined inductively by examining the variables that use `x` in their definition. If `x` is used in the definition of variable `y`, and `x` is not masked by other variables within that definition, then `x` is immediately observable at `y`. We then consider the variables that use `y` in their definitions, and apply the same criteria.

We track such notions by introducing additional variables. First, *combinatorial usage expressions*—`x_COMB_USED_BY_y`—determine whether a variable is masked within a definition. The variable is `true` if `x` is not masked by other elements of `y`'s definition. Second, *immediate observability expressions*—`x_COMB_OBSERVED`—which offer a way to check the status of the non-masking path. For each Boolean variable in the program, there could exist one or more immediate non-masking paths.

Consider the code fragment in Figure 4, where `out` is an output variable, `in1`, `in2`, and `in3` are input variables, and `v1`, `v2`, and `v3` are internal variables.

```
v1 = in1 and in2;
v2 = if (in3) then v3 else v1;
v3 = not in2;
out = v1 or v2;
```

Fig. 4: Sample Lustre code

We can generate additional definitions to track the observability of variables as in Figure 5. Variable `v1` is used by two variables—`v2` and `out`—in their definitions and therefore has two potential immediate non-masking paths: directly through the output variable `out` or through `v2`. Variable `in2` also has two potential immediate non-masking paths through its use in defining `v1` and `v3`. All the other variables are each used once, so each has only one immediate non-masking path.

3.4.2 Delayed Non-Masking Paths

Reactive systems compute in cycles, and variable values from the previous cycle can be referred to. As a result, the effect of a variable on output may not be

```
in1_COMB_USED_BY_v1 = in2;
in2_COMB_USED_BY_v1 = in1;
in3_COMB_USED_BY_v2 = true;
v3_COMB_USED_BY_v2 = in3;
v1_COMB_USED_BY_v2 = (not in3);
in2_COMB_USED_BY_v3 = true;
v1_COMB_USED_BY_out = (not v2);
v2_COMB_USED_BY_out = (not v1);

out_COMB_OBSERVED = true;
in1_COMB_OBSERVED = (in1_COMB_USED_BY_v1
    and v1_COMB_OBSERVED);
in2_COMB_OBSERVED = ((in2_COMB_USED_BY_v1
    and v1_COMB_OBSERVED) or
    (in2_COMB_USED_BY_v3 and
    v3_COMB_OBSERVED));
in3_COMB_OBSERVED = (in3_COMB_USED_BY_v2
    and v2_COMB_OBSERVED);
v3_COMB_OBSERVED = (v3_COMB_USED_BY_v2 and
    v2_COMB_OBSERVED);
v1_COMB_OBSERVED = ((v1_COMB_USED_BY_v2
    and v2_COMB_OBSERVED) or
    (v1_COMB_USED_BY_out and
    out_COMB_OBSERVED));
v2_COMB_OBSERVED = (v2_COMB_USED_BY_out
    and out_COMB_OBSERVED);
```

Fig. 5: Introduced variables to track immediate non-masking paths

observed until several computation cycles after a value is computed. In each of these intermediate computational steps, the system state is stored in a delay variable, until it propagates to an output eventually. We call such a path—propagating influence through a delay variable to an output—a *delayed non-masking path* and the variable is *delay observable*. A delayed non-masking path can be built over multiple immediate non-masking paths: from a variable to a latch—a delay variable—then from the latch to another latch until an output is reached.

Suppose we have a sample code fragment in Figure 6, where `delay1` and `delay2` are delay expressions.

```
delay1 = (0 -> pre(in1));
v1 = (if (delay1 > 0) then true else in2);
delay2 = (false -> pre(v1));
```

Fig. 6: Sample Lustre code

As with immediate non-masking paths, we can inductively build paths involving delay expressions. An example can be seen in Figure 7. Variable `v1`, which uses `delay1` and `in2` in its definition, is used in the definition of delay expression `delay2`. Therefore, a delayed non-masking path from `delay1` to `delay2` is composed of the immediate non-masking path from `delay1` to `v1`, then a delayed non-masking path from `v1` to `delay2`.

This annotation gives us the means to track immediate paths to latches. However, it is still necessary to establish the means to knit these paths together to form the sequential path over one or more delays passed on the path to output. To do so, we introduce a *to-*

```
delay1_COMB_USED_BY_v1 = true;
in2_COMB_USED_BY_v1 = (not (delay1 > 0));

in1_SEQ_USED_BY_delay1 = true;
v1_SEQ_USED_BY_delay2 = true;
delay1_SEQ_USED_BY_delay2 =
    (delay1_COMB_USED_BY_v1 and
    v1_SEQ_USED_BY_delay2);
in2_SEQ_USED_BY_delay2 =
    (in2_COMB_USED_BY_v1 and
    v1_SEQ_USED_BY_delay2);
```

Fig. 7: Introduced variables to track delayed non-masking paths

ken mechanism—a special variable to mark the current delay location. Once the token is initialized to a delay variable `x`, it can non-deterministically move to any other delay location—as long as `x` can be sequentially used by that location. It can also move to a special TOKEN_OUTPUT_STATE, is a monitored variable is reached or TOKEN_ERROR_STATE is the token can no longer possible be observed through a monitored variable or another delay.

```
v1 = (false -> (not (pre v2)));
v2 = (false -> (pre v1));
v3 = (0 -> (if ((pre v3) = 3)
          then 0
          else ((pre v3) + 1)));
out = (((v1 and v2) and (v3 = 2)) or
     ((not (v1 and v2)) and (not (v3 =
         2))));
```

Fig. 8: Sample Lustre code

We generate token equations to track the path taken through delay variables. Consider the code fragment in Figure 8. We can then generate the token equations shown in Figure 9. In this case, if we are currently at `TOKEN_D1`, and `v1` is immediately observable, then we reach the output. Otherwise, if `v1` can be delay observed through `v2`, then the token moves to `TOKEN_D3`.

3.4.3 Test Obligations

Test obligations are the goals prescribed by an adequacy criterion, establishing properties deemed important to thorough testing. Consider the expression: `v1 = ((v2 and in1) and delay2)`. If we wanted to satisfy MC/DC coverage, we would need to establish a set of test cases where each condition (`v2`, `in1`, and `delay2`) is `true` and `false`, where the entire decision `v1` evaluates to `true` and `false`, and where each condition is not masked within that decision. These obligations can be established as Boolean properties over the conditions. For example, we could achieve both outcomes for condition `v2` and show non-masking with these two properties: `v2_AT_v1_TRUE = ((v2 and in1) and delay2)` and `v2_AT_v1_TRUE = (((not v2) and in1) and delay2)`. If we can show that

```
token_next = (if ((pre token) =
    TOKEN_INIT_STATE) then token_first
   else (if ((pre token) =
       TOKEN_ERROR_STATE) then
       TOKEN_ERROR_STATE
   else (if ((pre token) =
       TOKEN_OUTPUT_STATE) then
       TOKEN_OUTPUT_STATE
   else (if ((pre token) = TOKEN_D1) then
     (if v1_COMB_OBSERVED then
         TOKEN_OUTPUT_STATE
     else (if ((token_nondet = TOKEN_D3)
         and v1_SEQ_USED_BY_v2)
     then TOKEN_D3 else TOKEN_ERROR_STATE))
   else (if ((pre token) = TOKEN_D2) then
     (if v3_COMB_OBSERVED then
         TOKEN_OUTPUT_STATE
     else (if ((token_nondet = TOKEN_D2)
         and v3_SEQ_USED_BY_v3)
     then TOKEN_D2 else TOKEN_ERROR_STATE))
   else (if ((pre token) = TOKEN_D3) then
     (if v2_COMB_OBSERVED then
         TOKEN_OUTPUT_STATE
     else (if ((token_nondet = TOKEN_D1)
         and v2_SEQ_USED_BY_v1)
     then TOKEN_D1 else TOKEN_ERROR_STATE))
   else TOKEN_ERROR_STATE))))));
```

Fig. 9: Example token equations.

each obligation—each property—is satisfied by at least one test case, we can show that the criterion is satisfied.

Observability-based test obligations conjoin the base obligations of the host criterion (e.g., MC/DC) with the path conditions required to establish either an immediate non-masking path or a delayed non-masking path from the expression where the base obligation is established to a monitored variable. An extension of MC/DC obligation `v2_AT_v1_TRUE` to the equivalent Observable MC/DC obligation is shown in Figure 10.

```
v2_AT_v1_TRUE = ((v2 and in1) and delay2);
v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE =
    v2_AT_v1_TRUE and (v1_SEQ_USED_BY_delay1
    and token=delay1);
v2_AT_v1_TRUE_CAPTURED =
    v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE ->
    (v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE or
    pre(v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE));

 obligation_0 = ((v2_AT_v1_TRUE and
     v1_COMB_OBSERVED) or
     (v2_AT_v1_TRUE_CAPTURED and token =
     TOKEN_OUTPUT_STATE));
```

Fig. 10: Sample test obligations

Expression `v2_AT_v1_TRUE` is a base obligation from the host criteria, defining an MC/DC obligation in expression `v1`. For delayed non-masking paths, we have to define the instant in which the expression would be immediately observable at a delay (the moment of *capture*). We then must latch this fact for the remainder of execution, in case

the execution path hits a monitored variable. Expressions `v2_AT_v1_TRUE_AFFECTING_AT_CAPTURE` and `v2_AT_v1_TRUE_CAPTURED` define this concept of capture for delayed non-masking paths. Finally, the full obligation is defined in expression `obligation_0`. In the obligation, the subexpression before the `or` operator defines immediate observability, and the second subexpression defines delayed observability. If either path is observed, then the obligation is met.

4 Case Study

We wish to assess the quality—in terms of fault finding—of test suites generated to satisfy both observable and traditional versions of the studied coverage criteria. We also want to evaluate the effect of observability on the effectiveness of test suites. Thus, we address the following questions:

1) Which criterion has the highest average likelihood of fault detection?
2) Are test suites generated to satisfy observable variants of coverage criteria more effective than the test suites generated to satisfy the original criterion?

The first question allows us to establish a baseline for discussion, and a general ranking of criteria. Which criterion—whether observable or traditional—returns the best results, on average? In the second case, we wish to understand whether observability generally offers a *beneficial* effect—does it consistently improve the likelihood of fault detection?

Additionally, we are interested in the nature of the tests generated to satisfy observable and traditional coverage criteria, and the effect of adding observability constraints to a coverage criterion:

3) What impact does observability have on the generation cost, the average size of the generated test suites, and the average percentage of satisfied obligations for each criterion?
4) Across studied criteria, does observability have a consistent effect on factors such as likelihood of fault detection, oracle and structure sensitivity, and satisfiability of obligations?

Question 3 allows us to examine how the addition of observability impacts generation cost, suite size, and the ability of the test case generation process to satisfy the imposed test obligations. Question 4 allows us to examine the *impact of the choice of criterion*. Does it matter whether we start with MC/DC or Branch Coverage? Does observability consistently impact test suites?

In order to answer these questions, we have performed the following experiment:

1) **Gathered case examples:** We have assembled two sets of software models, written in the Lustre language (Section 4.1).
2) **Generated mutants:** We generated up to 500 mutants, each containing a single fault. (Section 4.2)
3) **Generated structural tests:** We generated test suites intended to satisfy Branch, Condition, Decision, and MC/DC Coverage—as well as observable variants of each—using counterexample-based test generation. (Section 4.3)
4) **Reduced test suites:** We generated 50 reduced test suites using the full test suite generated in the previous step. (Section 4.4)
5) **Computed effectiveness:** We computed the fault finding effectiveness of each test suite using both an output-only oracle and an oracle considering all program variables (a *maximally powerful* oracle) against the set of mutants. (Section 4.5)

TABLE 4: Rockwell (non-inlined) example information

Model	# Inputs	# Internal Variables	# Outputs
DWM1	11	569	7
DWM2	31	115	9
Latctl_Batch	23	128	1
Microwave	13	162	4
Vertmax	40	30	2

TABLE 5: Rockwell (inlined) example information

Model	# Inputs	# Internal Variables	# Outputs	Average Complexity
DWM1	11	21	7	95.89285714
DWM2	31	10	9	21.36842105
Latctl_Batch	23	19	1	5.714285714
Microwave	13	99	4	9.15
Vertmax	40	30	2	720.5

4.1 Case Examples

In this study, we have made use of two pools of systems. The studied systems were originally modeled using the Simulink and Stateflow notations [33], [34]. Then, each was translated to the Lustre synchronous programming language [46] to take advantage of existing automation. In practice, Lustre would be automatically translated to C code. This is a syntactic transformation, and if applied to C, the results of this study would be identical.

Note that Lustre systems, and the original Simulink and Stateflow systems from which they were translated, operate in a sequence of computational steps. In each step, input is received, internal computations are performed sequentially, and output is produced. Within a step, no iteration or recursion is done—each internal variable is defined, and the value for it computed, exactly once. The system itself operates as an large loop.

4.1.1 Rockwell Collins Dataset

The first set of systems consists of four industrial systems developed by Rockwell Collins engineers. Two of these systems, *DWM_1* and *DWM_2*, represent portions of a Display Window Manager for a commercial cockpit display system. The other two systems—*Vertmax_Batch* and *Latctl_Batch*—represent the vertical and lateral mode logic for a Flight Guidance System (FGS). In addition, we have used a Microwave System—control software for a generic microwave oven developed as a non-proprietary teaching aid at Rockwell Collins. This set of benchmarks has been used in previous model-based test generation research [1], [3], [5], [11], [40], [47], [48], including previous work studying Observable MC/DC [10].

TABLE 6: Benchmark example information

Model	# Inputs	# Internal Variables	# Outputs	Average Complexity
6counter	1	4	1	3.5
AlarmFunctionalR2012	44	182	5	9.086666667
CarAll	2	8	1	4.125
cd	1	6	1	3.833333333
DockingApproach	13	1410	11	1.853754941
DragonAll	13	22	1	19.47619048
DragonAll2	13	27	1	20.77272727
durationThm1	5	7	1	3.333333333
ex3	2	5	1	3.6
ex8	2	5	1	3.4
fast_1	14	19	1	4.166666667
fast_2	14	30	1	4.37037037
FireFly	9	17	1	9.125
Gas	2	8	1	2.444444444
HysteresisAll	2	5	1	5.4
IllinoisAll	10	16	1	11.85714286
Infusion	20	861	5	2.745823389
MesiAll	4	10	1	5.545454545
Metros1	3	16	1	3.533333333
Microwave01	13	126	1	6.417647059
MoesiAll	5	12	1	4.071428571
PetersonAll	12	28	1	14.65517241
ProducerConsumerAll	4	12	1	3.153846154
ProductionCell	3	15	1	3.214285714
Readwrit	9	24	1	12.04
RtpAll	12	24	1	15.96
Speed2	2	5	1	3.6
Stalmark	1	3	1	21
SteamBoilerNoArr1	33	99	1	14.85
SteamBoilerNoArr2	19	3	1	30.66666667
Swimmingpool1	8	21	1	8.1875
Switch	3	2	1	3.333333333
Switch2	3	2	1	3.333333333
SynapseAll	4	10	1	4.555555556
Ticket3iAll	13	20	1	11.45454545
Traffic	1	3	1	5.666666667
Tramway	4	23	1	2.727272727
TwistedCounters	1	4	1	5
Two Counters	1	3	1	2
UMS	5	39	1	2.837837838

Previous work has found that, due to masking, the structure of the model can have a significant impact on the resulting efficacy of generated test suites for MC/DC [1], [2]. In theory, observability can assist in overcoming masking. To study this, we have generated two variants of each of the Rockwell Collins systems:

Maximally Non-Inlined: Each expression is as simple as it can possibly be, with sub-expressions split into independent intermediate variable calculations.

Maximally Inlined: Each expression is as complex as it can possibly be, with no intermediate sub-expressions used.

We repeat the entire experiment with both variants, in order to more thoroughly study the interaction between program structure and observability.

Information related to the non-inlined version of each system is provided in Table 4, and information related to the inlined versions is provided in Table 5. In both cases, we list the number of input variables, number of internal variables, and number of output variables. The latter two numbers give an indication of the size of the model, as each internal and output variable corresponds to an expression that must be calculated each computational cycle. For the inlined versions, we also list the average *complexity* of the inlined expressions—that is, the average number of boolean operations in each expression.

4.1.2 Benchmarks Dataset

While the Rockwell Collins systems allow us to take a detailed look at the effect of program structure, the number of systems is relatively low. In order to more thoroughly analyze the effects of observability, we have also chosen an additional 40 systems from the open-source Benchmarks dataset. Several of these models have been used in previous work, including a NASA example, *Docking_Approach*, which describes the behavior of a space shuttle as it docks with the International Space Station [11]. Two other systems, *Infusion_Mgr* and *Alarms*—which represent the prescription management and alarm-induced behavior of an infusion pump device—were also used in previous work [3], [11], [49].

The Benchmark Lustre models are available from https://github.com/Greg4cr/Reworked-Benchmarks/tree/SingleNode.

Information related to each system is provided in Table 6, where we again list the number of input variables, number of internal variables, and number of output variables. In this case, we lack the original models, and cannot control the level of inlining. Therefore, we also list the average complexity of expressions to give an idea of how inlined each model is.

4.2 Mutant Generation

The following mutation operators were used in this study:

Arithmetic: Changes an arithmetic operator (+, -, /, *, mod, exp).

Relational: Changes a relational operator (=, ≠; <; >; ≤; ≥)[2].

Boolean: Changes a boolean operator (∧; ∨, XOR).

Negation: Introduces the boolean ¬ operator.

Delay: Introduces the delay operator on a variable reference (that is, 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.

The mutation operators used in this study are discussed at length in [50]. 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.

Note that the type of mutants used in the evaluation in this report are similar to those used by Andrews et al., where the authors found that generated mutants are a reasonable substitute for actual failures in testing experiments [27]. Additionally, recent work from Just et al. suggests a significant correlation between mutant detection and real fault detection [26].

In order to control experiment costs, we do not use all possible mutants for each model. Instead, we employ

2. Some definitions of this operator also replace the entire expression with `true` or `false`. We do not do this in this experiment.

the following rule-of-thumb—if a model has fewer than 500 possible mutations, we use all possible mutations. If over 500 mutations are possible, we choose 500 of them for use in the experiment. In order to select mutants, we first gather a list of all possible mutations. Then, we use the proportions of each mutation type in the full set to select the number of mutants for the reduced set of 500, or a little bit greater than 500 due to some calculating error. Mutants of each type are then chosen randomly until the determined number are chosen for that type. This process prevents biasing towards particular types of mutations. Instead, the proportion of each fault type is maintained, despite not using the full set of mutations.

4.3 Test Data Generation

In this research, we explore four structural coverage criteria: Condition Coverage, Decision Coverage, Branch Coverage, and Modified Condition/Decision Coverage (MC/DC) [6], [19]. These criteria are defined in Section 2.2. For each criterion, we generate tests for both the traditional criterion as well as a version requiring observability. We refer to the observable versions of each criterion as Observable Condition Coverage (**OCondition**), Observable Decision Coverage (**ODecision**), Observable Branch Coverage (**OBranch**), and Observable MC/DC (**OMC/DC**).

For our directed test generation approach, we used counterexample-based test generation to generate tests satisfying the four coverage criteria and their observable variants [17], [51]. In this approach, each coverage obligation is encoded as a temporal logic formula in the model, and a model checker is used to produce a counterexample illustrating how the coverage obligation can be covered. This counterexample offers test input—a series of values for each input variable for one or more test steps. By repeating this process for each coverage obligation for the system, we can use the model checker to derive test sequences intended to achieve the maximum possible coverage of the model.

We have extended our model-based test generation framework[3] to also generate test cases for observable criteria. This framework makes use of the JKind model checker [36], [52] as the underlying generation engine because we have found that it is efficient and produces tests that are easy to understand [53].

The test generation framework is available from https://github.com/MENG2010/lustre.

4.4 Test Suite Reduction

Counterexample-based test generation results in a separate test for each coverage obligation. This leads to a large amount of redundancy in the tests generated, as each test likely covers several obligations. Consequently, the test suite generated for each coverage criterion is generally much larger than is required to provide coverage. Given the correlation between test suite size and fault finding effectiveness [54], this has the potential to yield misleading results—an unnecessarily large test suite may lead us to conclude that a coverage criterion has led us to select effective tests, when in reality it is the size of the test suite that is responsible for its effectiveness. To avoid this, we reduce each naïvely generated test suite while maintaining the coverage achieved. To prevent us from selecting a test suite that happens to be exceptionally good or exceptionally poor relative to the possible reduced test suites, we produce 50 different reduced test suites for each case example.

Reduction is performed using a simple greedy algorithm. We determine the coverage obligations satisfied by each test generated, and initialize an empty test set *reduced*. We then randomly select a test from the full set of tests; if it satisfies obligations not satisfied by any test input in *reduced*, we add it to *reduced*. We continue until all tests have been examined in the full set of tests.

4.5 Computing Effectiveness

In order to compute effectiveness of the generated test suites, we produce *traces* of execution by executing each test case against the original program and each mutant—recording the value of all variables at each step.

In our study, we use what are known as *expected value oracles* as our test oracles [3]. Consider the following testing process for a software system: (1) the tester selects inputs using some criterion—structural coverage, random testing, or engineering judgment; (2) the tester then defines concrete, anticipated values for these inputs for one or more variables (internal variables or output variables) in the program. Past experience with industrial practitioners indicates that such oracles are commonly used in testing critical systems, such as those in the avionics or medical device fields.

We explore the use of two formulations of expected value oracle: an *output-only (OO) oracle strategy* defines expected values for all outputs, and a *maximum (MX) oracle strategy* that defines expected values for all outputs and all internal state variables. The OO oracle strategy represents the oracle most likely to be used in practice. Both oracle strategies have been used in previous work, and we use both to allow for comparison [1], [3].

To give an example, consider the model defined in Figure 1. This model has a single output variable, `out1`. Therefore, the output-only oracle strategy would define an expected value for `out1`. The model also has three internal state variables—`v1`, `v2`, and `v3`. The maximum oracle strategy would define expected values for all three of those variables *and* the output variable `out1`.

To produce an oracle, we use the values of the monitored variables from the traces gathered by executing test cases on the original program, and we compare

3. Used in past projects, such as [1], [3], [11].

TABLE 7: Average percent of mutants killed for each pairing of criterion and oracle.

	Rockwell (I)		Rockwell (NI)		Benchmarks	
	MX	OO	MX	OO	MX	OO
OMC/DC	95.61%	87.03%	98.85%	85.88%	89.62%	65.14%
MC/DC	94.85%	84.94%	94.87%	53.89%	88.36%	57.13%
OCondition	88.38%	68.50%	98.95%	85.61%	86.22%	62.93%
Condition	71.00%	55.71%	93.38%	47.64%	79.37%	50.50%
ODecision	83.38%	60.33%	98.37%	83.44%	86.21%	64.70%
Decision	85.03%	53.48%	93.32%	45.73%	78.81%	49.26%
OBranch	81.92%	57.10%	96.84%	70.19%	85.47%	62.70%
Branch	84.20%	47.91%	87.17%	32.05%	73.19%	46.50%

those values to those recorded for each mutant. The fault finding effectiveness of the test suite and oracle pair is computed as the number of mutants detected (or "killed"). For all studied systems, we assess the fault-finding effectiveness of each test suite and oracle combination by calculating the ratio of mutants killed to total number of mutants.

5 Results and Discussion

In this section, we will address our research questions and discuss the implications of the results. As a reminder, we are interested in the following:

1) Which criterion has the highest average likelihood of fault detection? (Section 5.1)
2) Are test suites generated to satisfy observable variants of coverage criteria more effective than the test suites generated to satisfy the original criterion? (Section 5.2)
3) What impact does observability have on the generation cost, the average size of the generated test suites, the average percentage of satisfied obligations for each criterion? (Section 5.4)
4) Across the studied criteria, does observability have a consistent effect on test suites in terms of factors such as likelihood of fault detection, oracle and structure sensitivity, and satisfiability of obligations? (Section 5.5)

5.1 Overall Efficacy

Table 7 lists the average percentage of faults detected by test suites generated for each of the eight coverage criteria, separated by oracle strategy, for the Rockwell and Benchmarks datasets. From these results, we can see that—on average—test suites generated to satisfy OMC/DC tend to kill a larger percent of mutants than test suites satisfying all other coverage criteria. For both variants of the Rockwell systems—with any oracle strategy—test suites generated to satisfy OMC/DC kill the most mutants. The sole exception is for the non-inlined variant—with the maximum oracle strategy—where OCondition suites outperform OMC/DC by 0.1%. For the Benchmark models—with any oracle strategy—OMC/DC-satisfying suites have the highest average possibility of revealing faults.

> Test suites satisfying Observable MC/DC are generally the most effective, killing 95.61% of mutants on average (MX oracle strategy) and 87.03% (OO oracle strategy) for the inlined Rockwell models, 98.85% (MX)/85.88% (OO) for the non-inlined Rockwell models, and 89.62% (MX)/65.14% (OO) for the Benchmarks models.

We can examine this question further through statistical analysis. To address this, we first formulate our hypothesis as follow:

H_1: For each system in our study—with any oracle strategy—the OMC/DC criterion produces test suites with the highest likelihood of fault detection.

The paired null hypothesis is,

H : For each system in our study—with any oracle strategy—the OMC/DC criterion produces test suites with a likelihood of fault detection drawn from the same distribution as another criterion's suites.

We have performed a one-sided (strictly greater) Mann-Whiteney-Wilconxon rank-sum test [55], a non-parametric hypothesis test used to determine whether two independent samples were selected from populations having the same distribution, to verify our hypothesis. Since we cannot generalize across non-randomly selected case examples, we apply the statistical test over pairs of coverage criteria (i.e., any of the coverage criteria versus the rest of the coverage criteria respectively, therefore, we have 56 pairs of metrics in total), for each pairing of model and oracle type, with = 0:05.

The statistical results are presented in Table 8. In this table, we list the percentage of cases for each dataset where we can reject H —that is, where we can confirm that OMC/DC outperforms the compared criterion. We also list the percentage of cases where the reverse is true—where we can state that the other criterion outperforms OMC/DC with significance. For example, for the Rockwell (Non-inlined) models, with an output-only oracle strategy, OMC/DC outperforms all criteria except OCondition in 100% of cases, with statistical significance.

For Benchmarks, with any oracle strategy, the percentage of cases where OMC/DC suites outperform suites satisfying other coverage criteria is always higher than the percentage of suites satisfying other criteria outperforming OMC/DC suites. That is, OMC/DC always has a highest average likelihood of fault detection. This is also true in all situations for both variants of the Rockwell models with an output-only oracle strategy. Results are a little less clear-cut for the Rockwell models when paired with a maximum oracle strategy, where other criteria occasionally tie or outperform OMC/DC. For instance, for the inlined variants, ODecision, OBranch, and Branch suites outperform OMC/DC suites as often as OMC/DC suites outperform their counterparts.

Intuitively, these results makes sense. There is a clear boost in performance from the addition of observability.

TABLE 8: Percent of cases where OMC/DC suites outperform suites satisfying other criteria with significance, and where suites satisfying other criteria outperform OMC/DC suites.

		MX Oracle		OO Oracle	
		More Effective	Less Effective	More Effective	Less Effective
Benchmarks	ODecision	45.00%	10.00%	45.00%	15.00%
	OCondition	52.50%	15.00%	47.50%	15.00%
	OBranch	37.84%	28.95%	45.95%	21.05%
	MC/DC	45.00%	15.00%	55.00%	5.00%
	Decision	57.50%	2.50%	67.50%	2.50%
	Condition	65.00%	5.00%	72.50%	2.50%
	Branch	65.79%	7.69%	71.05%	7.69%
Rockwell (Inlined)	ODecision	40.00%	40.00%	80.00%	20.00%
	OCondition	60.00%	0.00%	100.00%	0.00%
	OBranch	20.00%	20.00%	80.00%	20.00%
	MC/DC	40.00%	20.00%	80.00%	20.00%
	Decision	40.00%	20.00%	80.00%	20.00%
	Condition	80.00%	20.00%	80.00%	0.00%
	Branch	20.00%	20.00%	80.00%	20.00%
Rockwell (Non-inlined)	ODecision	80.00%	20.00%	100.00%	0.00%
	OCondition	20.00%	20.00%	40.00%	20.00%
	OBranch	40.00%	60.00%	100.00%	0.00%
	MC/DC	80.00%	0.00%	100.00%	0.00%
	Decision	100.00%	0.00%	100.00%	0.00%
	Condition	100.00%	0.00%	100.00%	0.00%
	Branch	60.00%	40.00%	100.00%	0.00%

As Table 7 shows, the observable versions of criteria almost always outperform both their non-observable counterpart and *all other non-observable criteria*, except the original MC/DC. MC/DC suites outperform all of the other non-observable versions of the studied criteria, and is the only non-observable criterion to produce suites that occasionally outperform the observable counterparts. The addition of observability boosts the efficacy of the generated test suites, generally with the end result that Observable MC/DC produces the most effective test suites. OMC/DC does not always produce the best suites, but it is the safest choice of the studied criteria.

Across the board, efficacy tends to be higher for the maximum oracle strategy, and the gap between observable and non-observable criteria tends to be less. This can be explained by examining the concept of masking. With an output-only oracle strategy, input must trigger a fault, and the effect of a fault must not be masked by expressions on the path to the output. Observability is intended to overcome masking, and clearly does assist—given the results for output-only oracles. However, with a maximum oracle strategy, we already have expression-level observability. Masking along the path to the output does not need to be overcome. The observable criteria generally produce more effective suites even in these cases, but the possibility for improvement is smaller.

In general, however, maximum oracles are prohibitively expensive to employ [3]. A tester would need to specify expected values for all variables, for each test step. This is not usually a realistic goal. Output-only oracles are the most common, and OMC/DC appears to be the most effective criterion when paired with this common oracle strategy.

5.2 Efficacy Impact of the Addition of Observability

In Table 9, we present the average improvement in efficacy when moving from a traditional criterion—such as MC/DC—to its observable counterpart over all models for each dataset. Choosing the test input that would reveal a fault for a given test oracle is an undecidable task. However, the intent of observability is to increase the likelihood of detection by overcoming masking. The results show the merit of this idea—there is generally an increase in efficacy. Regardless of the underlying coverage criterion, observability seems to have a positive impact on the likelihood of detecting faults.

TABLE 9: Average improvement in the likelihood of fault detection, after adding observability constraints

		MX Oracle	OO Oracle
Benchmarks	MC/DC	1.79%	53.38%
	Decision	12.78%	88.03%
	Condition	11.12%	132.79%
	Branch	26.44%	163.10%
Rockwell (Inlined)	MC/DC	0.80%	2.77%
	Decision	-1.80%	14.81%
	Condition	42.92%	32.18%
	Branch	-3.08%	22.13%
Rockwell (Non-inlined)	MC/DC	4.58%	351.12%
	Decision	5.95%	392.44%
	Condition	6.46%	384.93%
	Branch	12.81%	389.99%

This is especially true when an output-only oracle—the most common oracle strategy [3]—is used. When using an output-only oracle, masking is a major problem. Our results show that observability can overcome masking. This can be clearly seen in the Rockwell (non-inlined) models, where the addition of observability improves efficacy up to 392.44%. Results are more subdued for the inlined variants—up to a 32.18% improvement.

We can see from these results that the structure of the system—how code is written—has some impact on the impact of adding observability. The Rockwell examples offer two extremes—either entirely inlined or with the simplest possible expressions. At the later end, there is tremendous improvement from adding observability. If there are a large number of simple expressions, then masking along the path to the output is far more likely than if there are a smaller number of expressions. As a result, observability has a major impact, propagating the effect of a fault to the output variables. On the other hand, if there are a small number of expressions, then the

path to output will be shorter. Therefore, observability will have a smaller impact.

In addition, past work has shown that the test cases generated for heavily inlined systems may be more effective from the start [1], [2]. Criteria such as MC/DC require that an independent impact be shown for each condition within a decision. That is, if MC/DC is fulfilled, then a condition will not be masked within the expression that it appears in. Its impact can be masked on the path to output, but will affect the outcome of the decision that it falls within. If a model is more heavily inlined, then the requirements of standard MC/DC are more strenuous—independent impact must be shown for more complex expressions. At the same time, the path to output is shorter, limiting further opportunities for masking. Therefore, the test cases may be more effective from the start, and the further impact of observability may be more limited.

We can see some evidence from this that observability helps bridge the gap from output-only oracle to maximum oracle—without adding additional human oracle cost [56], and the gap from non-inlined to inlined program structure. The Benchmarks examples are varied in terms of structure. As a result, the impact of adding observability falls between the two extremes of the Rockwell models—with improvements of up to 163.10% for the output-only oracle strategy.

As noted earlier, improvements tend to be smaller when employing a maximum oracle strategy. For non-inlined implementation of Rockwell models, we see average improvements of up to 12.81%. For the inlined variants, we see up to a 42.92% average improvement, and even see small performance downgrades of up to 3.08%. For the Benchmarks dataset, we see average improvements of up to 26.44%.

> Adding observability improves efficacy over satisfaction of traditional criteria, with average improvements of 11.94% with a maximum oracle and 125.98% with the output-only oracle (with per-model improvements of up to 1760.52%).

We can establish evidence by performing statistical analysis, employing the same test used previously. We formulate our hypotheses as follow:

H_2: For each system and oracle, the observable version of a criterion produces test suites with a higher likelihood of fault detection than the traditional variant.

The paired null hypothesis is:

H_2: For each system in our study—with any oracle strategy—the observable version of a criterion produces test suites with a likelihood of fault detection drawn from the same distribution as the traditional variant.

The statistical results are presented in Table 10, where we list the percent of cases where we can reject H_2—we can provide evidence that the observable criterion produces more effective test suites—along with the percentage of cases where we can state with significance that the reverse is true—that the traditional criterion is more effective. For example, for the Benchmark models—with a maximum oracle strategy—suites satisfying the OMC/DC criterion outperform MC/DC-satisfying suites with significance in 45% of cases, while the reverse is true for only 15% of cases. For the remaining 40% of the models, neither outperforms the other with significance.

Almost universally, the observable variant outperforms the traditional variant—with significance—in more cases. The only two situations where this is reversed are for Decision Coverage and Branch Coverage on the inlined Rockwell models, paired with a maximum oracle strategy. As highlighted above, this is the exact situation where we would expect the least benefit from the addition of observability. However, with the more realistic output-only oracle strategy, the observable variant of the criterion produces more effective suites in the vast majority of cases.

5.3 Factors That Influence Efficacy

Observability tends to improve the efficacy of test suites. However, it does not do so in all cases, and the gains in efficacy are not consistent across all models. Therefore, it is worth examining the factors that influence the efficacy of observability. We can illustrate some of these factors by looking at situations when observability had a minimal—or, worse, a negative—impact on efficacy (listed in Table 11).

As the percentage of fulfilled obligations decreases, the efficacy of the resulting suite decreases as well. The observable versions of criteria impose much more difficult obligations to satisfy, so some drop is not surprising—either from provably infeasible obligations or obligations that the test generator is unable to address. However, if the loss in obligation satisfaction is major, then we will observe some loss in performance. The worst case of this is the Docking_Approach example, where—for OCondition Coverage—we lose 89% satisfaction of the obligations. This resulted in a 31% loss in efficacy.

At first glance, the structure of the model—the level of inlining—appears to have some impact. With the Rockwell models, we never see a downgrade in performance for the non-inlined variants. We do see occasional efficacy losses for inlined models (including some of the "worst" examples from the Benchmarks set). However, we do not believe that this is due to inlining alone—for instance, Docking_Approach is not inlined—but because inlining is a factor informative of *model complexity*.

Inlined models tend to see less improvement from observability because they have more complex expressions. Regardless of the length of the path to output, complex expressions suffer more from masking, making it harder to guarantee a clear path to output. In turn, this potentially leads to lower levels of overall satisfaction for the observable variants. However, even though Docking_Approach is not inlined, it does have a deep

TABLE 10: Cases where observable criterion produces suites outperforming non-observable variant with significance, and when the non-observable variant is more effective.

		MX Oracle		OO Oracle	
		Observable	**Traditional**	**Observable**	**Traditional**
Benchmarks	MC/DC	45.00%	15.00%	55.00%	5.00%
	Decision	57.50%	5.00%	67.50%	2.50%
	Condition	65.00%	7.50%	67.50%	7.50%
	Branch	60.53%	5.26%	71.05%	2.63%
Rockwell (Inlined)	MC/DC	40.00%	20.00%	80.00%	20.00%
	Decision	20.00%	40.00%	60.00%	20.00%
	Condition	60.00%	20.00%	80.00%	20.00%
	Branch	0.00%	40.00%	80.00%	20.00%
Rockwell (Non-inlined)	MC/DC	80.00%	0.00%	100.00%	0.00%
	Decision	100.00%	0.00%	100.00%	0.00%
	Condition	100.00%	0.00%	100.00%	0.00%
	Branch	60.00%	0.00%	100.00%	0.00%

TABLE 11: Downgrade or lowest upgrade in efficacy when transitioning from traditional to observable criteria.

		MX Oracle	**OO Oracle**
Benchmarks	MC/DC	-11.7%, PetersonAll	-47.41%, Car_All
	Decision	-9.35%, MoesiAll	0.00%, 6counter/Metros1/ProductionCell/ Stalmark/Switch/Switch2/Ticket3iAll/ Tramway/TwistedCounters/UMS
	Condition	-31.62%, DockingApproach	-22.75%, DockingApproach
	Branch	-3.64%, Rtp_All	-8.17%, Rtp_All
Rockwell (Inlined)	MC/DC	-1.42%, DWM1	-0.91%, DWM1
	Decision	-6.43%, DWM1	-0.71%, DWM2
	Condition	-7.67%, DWM1	-0.46%, DWM1
	Branch	-11.65%, DWM1	-6.53%, DWM2
Rockwell (Non-inlined)	MC/DC	0.04%, Latctl_Batch	12.42%, DWM2
	Decision	0.92%, DWM2	27.77%, DWM2
	Condition	0.67%, DWM2	18.47%, DWM2
	Branch	0.00%, Latctl_Batch/Vertmax	40.69%, Latctl_Batch

state space—a series of gated conditions—which results in a longer path to establish to ensure observability. Therefore, we get lower satisfaction of the obligations for the observable variant than the original, which must simply satisfy obligations on individual expressions. The problem, then—inlined or not—is establishing a masking-free path from the expression to the output.

Non-inlined models can offer complex observability requirements because the path length is long—a failure must propagate through a long series of expressions to impact the output. Each individual expression is simple, but there are a large number of them to pass through unmasked. Therefore, the *path length* can be informative of the difficulty of achieving observability—impacting both obligation satisfaction and efficacy. In non-inlined models, the individual statements are simple. This results in trivial satisfaction of traditional coverage criteria, and weaker tests. Even if tests trigger a fault, they tend to be masked on the path to output. As a result, there tends to be a greater performance boost from observability.

Inlined models tend to have a shorter path-to-output, but each expression is more complex. Therefore, at each expression from activation to output, a failure could be easily masked. *Statement complexity*—which can be judged by the level of inlining—impacts obligation satisfaction as well as the efficacy gap between observable and traditional variants. Suites satisfying the traditional criterion must satisfy much more difficult obligations, and there are fewer opportunities for masking on the path to output. Therefore, the efficacy of the suites satisfying the traditional criterion tend to be relatively effective even without observability. Observability can boost efficacy, but the difficulty of finding a path through the more complex expressions can also cause issues.

The above only discussed combinatorial paths—from expression to output in a single computation cycle. Complexity must also be considered over multiple computation cycles, as observability can be established *after delays*. One additional factor impacting the path to output are the *number of delay expressions*. Failures can be propagated across computation cycles. However, the use of such expressions introduces an additional source of complexity to a model, and test obligations that require a delay observable path can be harder to satisfy.

In cases where the loss in performance—or gain—are small, one factor that may contribute is the test suite reduction process. Tests are chosen randomly for the reduced suite, based on their ability to cover obligations. In general, efficacy may be essentially identical between the observable and non-observable suites, and poor test cases choices push the average slightly lower—but not in a statistically significant manner. This would explain most of the inlined Rockwell scenarios, as well as Branch-satisfying suites on the Rtp_All system from the Benchmarks dataset. This is a case where Branch and OBranch attain generally the same results—the `if` statements in the model are easily observable—but the average for OBranch is slightly lower due to poor test selection during suite reduction.

Factors that can harm efficacy—generally resulting in a reduction in the number of fulfilled obligations—include expression complexity, the

TABLE 12: Average change in generation cost, test suite size, and percentage of satisfied obligations when moving from a criterion to the observable version.

		Size of Test Suites	Obligation Satisfaction	Generation Cost
Benchmark	MC/DC	23.59%	-22.37%	346.75%
	Decision	63.66%	-12.38%	173.43%
	Condition	52.61%	-20.02%	247.35%
	Branch	69.68%	-8.02%	129.39%
Rockwell (Inlined)	MC/DC	25.51%	-4.20%	2422.91%
	Decision	23.02%	-5.72%	422.04%
	Condition	49.97%	-4.17%	251.08%
	Branch	7.73%	-3.29%	551.48%
Rockwell (Non-inlined)	MC/DC	307.88%	-6.49%	996.48%
	Decision	392.46%	-8.11%	1285.85%
	Condition	376.25%	-6.30%	1081.58%
	Branch	343.54%	-4.82%	116.99%

TABLE 13: Median time (in seconds) required to generate test suites for each criterion and its observable version.

		Observable	Traditional
Benchmarks	MC/DC	24.81	5.55
	Decision	8.16	2.98
	Condition	13.78	3.97
	Branch	5.55	2.42
Rockwell (Inlined)	MC/DC	880.96	34.92
	Decision	38.26	7.32
	Condition	98.97	28.19
	Branch	37.27	5.72
Rockwell (Non-inlined)	MC/DC	195.49	17.83
	Decision	92.64	6.68
	Condition	104.48	8.84
	Branch	9.72	4.48

length of the combinatorial path from expression to output, and the length of the delayed path from expression to output.

5.4 Impact of Observability on Generation Cost, Test Suite Size, and Obligation Satisfaction

Each test case can satisfy more than one test obligation, depending on the input applied and the execution path taken by applying such input. Observable criteria require the same number of test obligations as their host criterion, but impose more difficult requirements for fulfillment. Rather than simply examining the number of test obligations required by a criterion, we can look at the size of the minimal test suites required to satisfy such criteria and the time cost to generate such test suites as an indicator of the difficulty of meeting testing goals.

In Table 12, we present the average change in generation cost, size of test suites, and percentage of fulfilled obligations when observability is required for each coverage criterion. Table 13 presents the median time required to generate test suites for each criterion. From these two tables, we can see that an observable criterion tends to require significantly more time to generate test suites than when the corresponding host criterion is used on its own. Such criteria yield more complex obligations, resulting in a more complex generation process where the model checker must spend more time identifying a solution that yields a non-masking path.

However, from Table 13, we can also see that test generation can still be completed in a reasonable time frame for observable criteria. For the Benchmarks dataset, the median generation time still tops under thirty seconds. The worst median—for Observable MC/DC over the inlined Rockwell models—is still under fifteen minutes. Compared to the cost of manual test case creation, the addition of observability is minor.

The addition of observability requires a longer test generation process, with average increases ranging from 129.39%-2422.91%.

As we can see from Table 12, regardless of the underlying coverage criterion, we see an increase in the number of test cases required for the observable version of the test suite. Fundamentally, observable criteria require more test cases to fulfill their obligations than the traditional variants. Because of the highly specific path to output required for each obligation, there is less overlap between test cases in terms of the obligations satisfied.

Program structure seems to have an impact on the magnitude of the size increase. Moving to an observable criterion results in a massive increase in test suite size for the non-inlined Rockwell models—307.87-392.46%—while there is only a modest increase of 7.73-49.97% for the inlined models. On a per-model basis for the Benchmarks examples, many of the models with smaller increases in suite size tend to also be heavily inlined.

This observation makes sense given the discussion above. The obligations for non-observable criteria are formed over individual expressions. If those expressions are simple, the obligations too will be simple. As a result, each test case may cover a variety of obligations with ease. If the model is more heavily inlined, then each obligation will be more complex, and more specialized. There will be less overlap in coverage between test cases [1]. The more heavily inlined the model, the larger the test suite tends to be.

Therefore, model structure has a major impact on the size of the test suite for suites satisfying the *non-observable* criteria. Inlined models start with larger test suites. Then, regardless of the model structure, the addition of observability, increases the size further.

The primary factor influencing the size increase from observability is the length of the path. Each expression encountered along the path imposes additional conditions on maintaining a non-masking path. Therefore, the longer the path length, the more complex the requirements are on the test case. As a result, we see a similar effect to changing the program structure. The individual test cases are more specialized, and there is less of a chance of overlap in covered obligations. This further explains the larger increase in size for non-inlined models. Non-inlined models have simpler expressions, but much more of them. As a result, the satisfaction complexity is in satisfying path constraints rather than in fulfilling the original expression-level obligations.

The addition of observability results in an increase in the size of test suites. The magnitude of that increase depends on the length of the path from each expression to the output.

Previous work has shown that observability imposes an additional complexity burden on the test case generator, generally resulting in some loss in obligation satisfaction [10], [11]. The results of this study further confirm this. Table 12 shows that—on average—there is a loss in obligation satisfaction regardless of the criterion. For the inlined Rockwell models, this average loss ranges from 3.29-5.72%. For the non-inlined versions, this ranges from 4.82-8.11%. Then, for the Benchmarks dataset, the average loss ranges from 8.02-22.37%.

As discussed earlier, a loss in obligation satisfaction is to be expected—the test obligations requires to ensure observability are far more complex than the equivalent obligations when observability is not required. This loss can occur for two reasons. First, if there is no masking-free path, then the obligation will be unfulfillable. This means that observability cannot be established, and thus, a fault in that statement *cannot* influence the output. Generally, this indicates dead code—code that, intentionally or not, cannot affect program output. Occasionally, this is a byproduct of either code reuse—where existing code is reused wholesale—or defensive programming.

However, in some cases, obligations may be too complex for the test generator to fulfill. In such cases, the generator will eventually return an "unknown" verdict. This is an indication that the generator was unable to meet the obligation, and was unable to conclusively determine that it could not be met (the case above). If the obligations are too complex, then the test generator can return weaker test suites because it eventually gives up on finding solutions that fulfill these obligations.

To better understand the reasons we lose coverage, we have listed the average percent of obligations fulfilled and the percent of obligations that resulted in "unknown" verdicts—where the test generator gave up on finding a solution—for each dataset in Table 14. First, we can see again that the observable variants see a lower rate of obligation fulfillment than the traditional criteria. Again, this is expected. In the case of the Rockwell models, we see that there are no situations where the test generator returned an unknown verdict. This means that any loss in such situations is due to *provably* unfulfillable obligations—dead code. This reduction in fulfillment is acceptable, as such obligations can never be fulfilled.

However, for the complex Benchmarks models, we do see some loss due to the test generator. On average, 4.11% (OBranch), 8.46% (OCondition), 3.90% (ODecision), and 7.41% (MC/DC) of obligations return "unknown" verdicts during test generation. We wish to avoid such situations, as they are situations where we cannot prove that the obligation cannot be fulfilled—the test generator just did not find a solution in time. Some of these obligations may have test cases meeting them. Many will not, but we lack proof in either case.

In the Benchmarks dataset, even the traditional criteria have obligations that result in unknown verdicts—on average, 0.09% (Branch), 0.33% (Condition), 0.09% (Decision), and 0.66% (MC/DC) of the obligations time out. However, these percentages are far lower than for the observable variants. This speaks to the complexity of establishing observability, which is often far beyond that of covering the obligations of the host criterion.

The two driving factors in these unknown verdicts are the length of the combinatorial path from expression and output and the number of delay expressions—the length of the delayed path—between the expression and the output. Both increase the complexity of finding a masking-free path between the expression that is the source of the base obligation and an output variable. If the path is more complex, the generator will have a harder time satisfying the test obligations.

Although paths are shorter in inlined models, the individual expressions are more complex than in non-inlined models. Although expressions are simple in non-inlined models, the paths are longer than in inlined models. As a result, the level of inlining does not play a major role in the loss in obligation satisfaction. The level of correlation between inlining and loss in satisfaction is relatively low. The length of the path—whether delayed or immediate—is of far more importance.

The addition of observability results in an decrease in the number of fulfilled obligations. This loss is due to either the discovery of dead code that cannot influence the output or obligations that are too complex for the test generator to solve.

5.5 The Effect of Observability

For our final research question, we wish to take a look at the effect of observability *itself*. Regardless of the underlying host criterion, does observability have a consistent impact on suite efficacy, oracle sensitivity, structure sensitivity, and obligation fulfillment?

5.5.1 The Choice of Host Criterion

The choice of coverage criterion is often made based on the perceived strength of that criterion. MC/DC is more strenuous to fulfill than Branch Coverage, and therefore, suites satisfying it *should* be more effective. While there are exceptions, this generally bears out in practice. In our study, MC/DC satisfaction results in stronger test suites than Branch Coverage satisfaction.

However, one question we are curious about is—when observability is required, *does the choice of host criterion matter*? Does observability consistently improve results, and is there still reasonable differentiation in the final results to see an impact from the choice of host criterion.

TABLE 14: Average % of obligations fulfilled, and the average % of the unfulfilled obligations that were due to an "unknown" verdict being returned by the test generation for each dataset.

	Branch		OBranch		Condition		OCondition	
	% Fulfilled	% Unknown	% Fulfilled	% Unknown	% Fulfilled	% Unknown	% Fulfilled	% Unknown
Benchmark	94.77%	0.09%	87.88%	4.11%	96.22%	0.33%	75.78%	8.46%
Rockwell (inlined)	97.88%	0.00%	94.94%	0.00%	98.88%	0.00%	94.92%	0.00%
Rockwell (non-inlined)	100.00%	0.00%	95.18%	0.00%	99.83%	0.00%	93.58%	0.00%
	Decision		ODecision		MC/DC		OMC/DC	
	% Fulfilled	% Unknown	% Fulfilled	% Unknown	% Fulfilled	% Unknown	% Fulfilled	% Unknown
Benchmark	94.60%	0.09%	81.17%	3.90%	86.84%	0.66%	67.08%	7.41%
Rockwell (inlined)	98.93%	0.00%	93.50%	0.00%	96.66%	0.00%	92.70%	0.00%
Rockwell (non-inlined)	99.95%	0.00%	91.85%	0.00%	99.48%	0.00%	93.07%	0.00%

From the results in Tables 7 and 9, we can still see that the choice of criterion matters. Observability generally results in better test suites, but there is no real consistency in the magnitude of that impact across criteria, oracles, and system structures. The choice of criteria does impact the end result. OMC/DC satisfaction does tend to result in better test suites than OBranch satisfaction. The gap between criteria is often narrower for the observable variants than their traditional variants, but there is still a gap. Therefore, we can conclude that the choice of host criterion still influences the final result.

With traditional coverage criteria, weaker criteria may be used because they offer *enough* benefit, but are less expensive to fulfill. This is particularly true when test cases are written by human developers. Branch Coverage is easier to understand and explain than MC/DC, and proving that your test cases meet the more strenuous requirements of MC/DC requires more time and effort. If satisfaction of Branch Coverage can be achieved within the time period alloted to testing and offers benefits to the testing process, it may be better to make use of it than to spend the same amount on time attaining partial coverage of MC/DC. Even in the case of automated generation, it may be reasonable to choose to maximize Branch Coverage over attaining partial coverage of MC/DC. If the test generator is unable to satisfy the requirements of MC/DC, then attaining a higher level of Branch Coverage could lead to better efficacy.

However, this same trade-off does not necessarily function in an equivalent manner once observability is required. As we can see from the discussion in Section 5.4, the added complexity of observability vastly outweighs the complexity added by the use of a criterion such as MC/DC over Branch Coverage. If the test generation framework employed in this study can satisfy Branch Coverage for a model, it can usually attain similar levels of MC/DC. There is a far more perceptible drop when moving to any of the observable criteria. A gap still exists between Observable Branch and Observable MC/DC, but the leap from non-observable to observable is much greater.

It follows then that—rather than asking which criterion to employ—the more important questions is whether to require observability. In the context of manual test creation, employing observability without tool support is likely to be too expensive to consider in any situation except when safety is absolutely crucial. In the case of automated generation, observability is—at least for the studied programs—reasonable to require. Although there are situations where the loss in coverage due to unknown verdicts is unacceptably high, for most of the studied programs there were clear benefits.

TABLE 15: Average improvement in mutation detection when changing from OO to MX oracle strategy.

	Benchmarks	Rockwell (Inlined)	Rockwell (Non-inlined)
OMC/DC	96.77%	10.73%	16.31%
ODecision	93.33%	49.94%	19.38%
OCondition	96.81%	34.53%	16.83%
OBranch	102.69%	59.89%	40.63%
MC/DC	208.76%	13.03%	352.92%
Decision	233.13%	78.20%	384.62%
Condition	297.46%	31.45%	375.41%
Branch	278.98%	97.67%	412.78%

These results also show that—as long as the test generator can handle the complexity of observability at all—the additional loss from choosing a more complex host criterion is minor. Therefore, we would recommend the use of stronger criteria such as MC/DC over weaker ones when observability can be handled by the test generator. That said, if the combined complexity of observability and criterion is too much for generation to handle, then a tester could first change the host criterion—then drop the observability requirement.

> The choice of host criterion influences the final efficacy, but the largest increase in complexity comes from the addition of observability itself. Varying both dimensions—criterion and observability—may allow testers to find an optimal level of efficacy and complexity.

5.5.2 Oracle Sensitivity

In normal situations, the results of testing are sensitive to the choice of variables monitored as part of the test oracle. We can see this in comparing the results of the maximum and output-only oracle strategies for suites satisfying the traditional non-observable criteria. When results are checked with the maximum oracle strategy, efficacy tends to be much high. This is because masking can prevent program elements from influencing other variables. With any oracle strategy other than the maximum oracle strategy, suite efficacy depends on the selection of variables monitored by the oracle [3]. This complicates the testing process, as it is not obvious *which* variables should be monitored, and coming up with

expected values for any variables other than the output variables can be very difficult.

In theory, observability should be a powerful tool in overcoming oracle sensitivity. By requiring a masking-free path from any targeted expression to the output, we should be able to increase the efficacy of using an output-only oracle strategy. In Table 15, we present the average improvement in fault finding when moving from an output-only oracle strategy to the maximum oracle strategy for each coverage criterion, and for each of the three datasets.

From these results, we can see that for the non-inlined Rockwell systems, oracle sensitivity is *greatly* reduced when we require observability—for instance, Branch-satisfying suites improve by 412.78% when changing oracle strategies, but OBranch-satisfying suites only improve by 40.63%. As discussed earlier, non-inlined systems tend to have a large number of simple expressions and long paths to output. These results make sense. The maximum oracle strategy monitors every single expression in the program. Therefore, the size of the maximum oracle is much larger than the output-only oracle, as it is much easier to detect faults. When paired with an output-only oracle strategy, suites satisfying traditional criteria will suffer greatly from masking. Observability overcomes this masking by requiring that each expression be able to influence the output.

We do not see the same magnitude of effect for the aggressively inlined versions of the Rockwell models. Except in the case of Condition Coverage, there is a reduction in oracle sensitivity, but the impact is less. Again, however, these results make intuitive sense. An inlined implementation has fewer expressions. Therefore, the maximum oracle is also smaller—with fewer points of observation. The observable versions of criteria still produce suites that are less sensitive to the choice of oracle strategy, but there is also potentially less oracle sensitivity to overcome in the first place.

The Benchmark models again fall between the two extremes. The suites satisfying the observable criteria are less sensitive to the choice of oracle strategy than suites satisfying traditional counterparts. Suites satisfying traditional Branch Coverage improve by 273.10% from the shift in oracle strategy, while suites satisfying OBranch Coverage only improve by 102.51%.

> Observability reduces sensitivity to the choice of oracle strategy by ensuring a masking-free path from expression to monitored variables.

5.5.3 Structural Sensitivity

Traditional coverage criteria—particularly MC/DC—are known to be sensitive to program structure [1], [2]. With an output-only oracle strategy, suites generated using the inlined version of the program will be far more effective at finding faults than suites generates using the non-inlined version of the program. Because individual expressions are more complex in the inlined program, their test obligations are more complex. There are also, often, fewer opportunities for masking on the path to output, as there are fewer expressions along that path. Observability should help overcome that sensitivity to structure. Although the individual expressions are simpler, overcoming masking along the path should result in a more robust test suite.

TABLE 16: Average change in efficacy when switching from non-inlined to inlined versions of Rockwell models.

	Max Oracle	OO Oracle
OMC/DC	-3.32%	1.50%
ODecision	-15.23%	-25.41%
OCondition	-10.71%	-18.56%
OBranch	-16.70%	-20.47%
MC/DC	0.18%	354.62%
Decision	-7.83%	344.09%
Condition	-22.54%	332.22%
Branch	-5.83%	332.24%

In our experiments, this seems to be the case. Table 16 lists the average change in efficacy when switching from non-inlined to inlined version of the Rockwell models. On average, we see that—for the traditional suites—there is a major improvement in efficacy when we change program structures. For suites satisfying the observable variants, we actually see a slight downgrade in performance.

For the traditional criteria, if we use a maximum oracle strategy, we see a downgrade in performance when changing program structure instead of the upgrade we saw with an output-only oracle strategy. This is because, with a non-inlined program, the size of the maximum oracle is very large. Each of the many simple expressions is monitored and checked. With an inlined program, the maximum oracle is much smaller—there are fewer expressions. With a maximum oracle strategy, changing to an inlined program structure is somewhat detrimental to performance. With traditional criteria—as the maximum oracle is generally prohibitively expensive—we would recommend inlining code to improve the performance of the output-only oracle strategy.

The above results make sense then as, with observability, we essentially see the same effect. There is no benefit from changing program structure, as the increased complexity of individual statements is replicated in the masking-free path to output required to attain observability. Instead, there is a slight downgrade in performance because the individual statements are more complex. When observability is required, a simpler program structure may be slightly preferable.

> Observability reduces sensitivity to the program structure by capturing the complexity benefits of inlining in the path from expression to output.

6 Threats to Validity

External Validity: Our study has focused on a small number of systems but, nevertheless, we believe the systems are representative of the critical systems domain, and our results are generalizable to that domain.

We have used one method of test generation—counterexample-based generation. There are many methods of generating tests and these methods may yield different results. Counterexample-based testing is used to produce coverage-directed test cases because it is a method used widely in testing safety-critical systems.

For each model and criteria, we have built 50 reduced test suites reduced using a simple greedy algorithm. It is possible that larger sample sizes may yield different results. However, in previous studies, smaller numbers of reduced test suites have produced consistent results [2].

Construct Validity: In our study, we primarily measure fault finding over seeded faults, rather than real faults encountered during development. However, Andrews et al. showed that seeded faults lead to similar conclusions to those obtained using real faults [27] for the purpose of measuring test effectiveness and Just et al. found a positive correlation between mutant detection and fault detection [26]. We have assumed these conclusions hold true in our domain/language, where examples of real faults are rare.

To control experiment costs, we limited the number of mutants used per model to 500. When more than 500 mutants exist, a random selection was used to avoid bias in mutant selection. While the selection of specific mutants is randomized, the distribution is matched to the full distribution of possible mutants in the model. In our experience, mutants sets greater than 100 result in similar fault finding; we generated up to 500 to further increase our confidence that no bias was introduced.

Lau and Yu [57] and Kaminski et al. [58] have defined fault hierarchies for Boolean expressions, outlining cases where detection of one fault could guarantee detection of other, redundant faults. The mutation operators we have employed could produce redundant mutations, but we have not removed those in our experiments. We do not know which faults are actually redundant in our evaluation. However, we have performed a worst-case analysis, and found that there is generally a low correlation between the percent of redundant faults and the percent of detected faults. Therefore, we do not believe that redundancies have had a significant impact on the results of our study.

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions beyond these analyses are met, and have favored non-parametric methods. In cases in which the base assumptions are clearly not met, we have avoided using statistical methods. Notably, we have avoided statistical inference across case examples.

7 Related Work

In this section, we will discuss our prior work on observability, the role of adequacy criteria in test case generation, other notions of observability, and other topics related to this work.

7.1 Prior Work on Observability

This work is an extension of our prior work defining and exploring the concept of observability [10]–[12]. We first proposed the concept of observability as an extension of the MC/DC coverage criterion [10]. An extended study found that that OMC/DC was more effective—and overcame many of the weaknesses of—traditional coverage criteria [11].

In a recent study, we extended the original tagging semantics of MC/DC in order to generate path conditions as part of Dynamic Symbolic Execution [12]. This work used OMC/DC purely as a test generation target rather than a general adequacy measurement approach. A source of optimistic inaccuracy in the original definition of OMC/DC was addressed by requiring value inequality of expressions from two branches when propagating `if` conditions. This approach was also able to explicitly terminate when there is no feasible paths. In the regular model-based test generation approach used in this and the other past work, a timeout is usually estimated and manually set in order to terminate the generation process. The DSE-based approach, as a result, could complete generation in a more efficient manner.

This work extends previous efforts by decoupling the notion of observability from MC/DC and exploring its application as a generic addition to any coverage criterion. While we found that MC/DC was still the most effective host criterion in many applications, this was not a universal case. This decoupling allows us to explore the impact of choosing a host criterion and to explore the efficacy of observability as a general construct of adequacy criteria. Our experimental work also considers a far wider range of programs than previously explored in order to better understand the general efficacy of observability-based coverage criteria.

7.2 Adequacy Criteria Efficacy in Test Generation

Automated test generation relies on the selection of a measurable test goal. Adequacy criteria, such as the coverage criteria that are the focus of this study are commonly used for this purpose. However, coverage is merely an approximation of a harder to quantify goal—"finding faults". The need to rely on approximations leads to two questions that researchers have examined multiple times. First, *do such proxies produce effective tests?* If so, *which criteria should be used to generate tests?*

Answers to these two questions are—to date—inconclusive. Some studies have noted positive correlation between coverage level and fault detection [21], [54], [59], while other work paints a negative portrait

of coverage [60]. Our prior work in search-based test generation for Java programs has found that coverage level is more strongly indicative of efficacy than factors such as suite size [21]. However, in our prior studies of model-based generation, tests generated specifically to achieve coverage were often outperformed by randomly-generated tests [11], [47], [48].

Results to date are promising, given the complexity of some of the faults detected [21], [61]–[63]. However, automated generation does not yet produce human competitive results [64]. Ultimately, if automated generation is to have an impact on testing practice, it must produce results that match—or, ideally, outperform—manual testing efforts. The efficacy of suites generated for many coverage criteria is limited by issues such as masking. Choices about how code is written [1], [2] and the selection of test oracle [3], [5], [40] impact the efficacy of some criteria. The notion of observability was designed to address both issues.

7.3 Coverage Criteria in Lustre and Function Block Diagram

Researchers studying coverage criteria for Lustre [65] and Function Block Diagrams (FBD) [66] implicitly investigated observability by examining variable propagation from the inputs to the outputs.

Some of the structural coverage criteria proposed specifically 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 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 [65]. Coverage metrics for FBD are based on a *d-path condition* that is similar to activation conditions in Lustre [66].

These coverage criteria in Lustre and FBD are different from the notion of observability in several respects. First, these metrics check if specific inputs affect the outputs and measure the coverage of variable propagation on all possible paths. Observability, on the other hand, checks if each test obligation from the host criterion affects the monitored variables, and determines if a path exists which propagates the effect of the obligation. Second, observability requires a stronger notion of how a decision must be exercised.

7.4 Observability in Hardware Testing

Observability has been studied in testing of hardware logic circuits. Observability-based code coverage metric (OCCOM) is a technique where tags are attached to internal states in a circuit and the propagation of tags is used to predict the actual propagation of errors (corrupted state) [42], [67]. A variable is tagged when there is a possible change in the value of the variable due to an fault. The observability coverage can be used to determine whether erroneous effects that are activated by the inputs can be observed at the outputs.

The key differences between our notion of observability and OCCOM are twofold: (1) our notion of observability investigates variable value propagation, while OCCOM investigates fault propagation and (2) OCCOM has pessimistic inaccuracy because of tag cancellation. 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 [67] or an unknown tag "?" [42] is used. Variables without tags or with unknown tags are not considered to carry an observable error.

Since we do not make a distinction between positive and negative tags, we do not have tag cancellation or the corresponding pessimistic inaccuracy. Extended work in [68] may fix pessimistic inaccuracy by producing test vectors with specific values, but is highly infeasible.

7.5 Mutation Coverage

As discussed in Section 3.2 there are close connections between observability and strong mutation coverage [25]. In the general case, strong mutation coverage is very difficult to achieve and expensive to measure [28], though recent efforts have made it somewhat more efficient [69]. Therefore, weak mutation coverage is often used instead, as a high level of weak mutation coverage can be more easily reached. Observability, as proposed in this work, offers a means to increase strong mutation coverage of faults in Boolean decisions.

7.6 Dynamic Taint Analysis

Our ideas for examining observability via tag propagation were derived from dynamic taint analysis (also called dynamic information flow analysis). In this approach, data is marked and tracked in a program at runtime, similar to our tagging semantics. This technique has been used in security as well as software testing and debugging [70], [71]. Taint propagation occurs in both explicit information flow (i.e., data dependencies) and implicit information flow (control dependencies). Although the way in which markings are combined varies based on the application, the default behavior is to union them [71]. Thus, dynamic taint analysis is conservative and does not consider masking. More accurate techniques for information flow modeling define path conditions quite similar to those used in this paper to prove *non-interference*, that is, the non-observability of a variable or expression on a particular output [72].

7.7 Dynamic Program Slicing

Dynamic program slicing [73] 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. However, similarly to dynamic taint analysis, it does not consider masking. Checked coverage uses dynamic slicing to assess oracle quality, where

oracles are program assertions [74]. Given a test suite, it yields 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. This work is designed to assess the *oracle*, not the test suite.

7.8 Checked Coverage

Recent work presents a stronger notion of coverage, checked coverage, which counts only statements whose execution contributes to an outcome checked by an oracle [75], [76]. The ideas of checked coverage and observability are conceptually very similar. With observability, one is trying to see whether a condition—or other code structure—propagates to a point of observation. With checked coverage, one restricts measured code coverage to program statements that are found in a backwards dynamic slice. Thus, in observability, metrics markings are *forward propagated*, while in checked coverage, markings are *back propagated*. Checked coverage is computed over program statements rather than conditions, so is not as precise as our work on observability. Also, it assumes a priori that one has a test from which to perform a backwards dynamic slice, so it is not suitable in its current form for test case generation.

8 Conclusion and Future Work

Many test adequacy criteria are highly sensitive to how statements are structured or the choice of test oracle. This sensitivity is caused by the fact that the obligations for structural coverage criteria are only posed over specific syntactic elements—statements, branches, conditions. Such obligations ensure that execution *reaches* the element of interest, and exercises it in the prescribed manner. However, no constraints are imposed on the execution path *after* this point. We are not guaranteed to observe a failure just because a fault is triggered. To address this issue, we have proposed the concept of *observability*—an extension to coverage criteria based on Boolean expressions that has the potential to eliminate masking. Observable coverage criteria combine the test obligations of their host criterion with an additional path condition that increases the likelihood that a fault encountered when executing the element of interest will propagate to a variable monitored by the test oracle. We hypothesize that this additional observability constraint will improve the effectiveness of the host criterion—no matter which criterion is chosen—particularly when used as a test generation target, paired with a common output-based test oracle strategy.

Our study has revealed that test suites satisfying Observable MC/DC are generally the most effective criterion. Overall, we found that adding observability tends to improve efficacy over satisfaction of the traditional criteria, with average improvements of up to 392.44% in mutation detection and per-model improvements of up to 1654.38%. Some of the factors that can harm efficacy include expression complexity, the length of the combinatorial path from expression to output, and the length of the delayed path from expression to output. The addition of observability results in an increase in the size of test suites and a decrease in the number of fulfilled obligations. The choice of host criterion influences the final efficacy, but the largest increase in complexity comes from the addition of observability itself. Varying both dimensions—criterion and observability—may allow testers to find an optimal level of efficacy and complexity. Finally, our hypothesis has proven accurate—observability reduces sensitivity to the choice of oracle and to the program structure.

Based on our results, *observability* is a valuable extension—regardless of the chosen host criterion. The addition of observability increases test efficacy and produces test suites that are robust to changes in the structure of program or the variables under monitored by test oracle. While our results are encouraging, there are areas open for exploration in future research:

Extension to other coverage criteria: A variety of coverage criteria have been proposed for logical expressions, some potentially more effective than MC/DC [77]. We will explore the effect of extending such criteria to offer observability.

Oracle data selection: We used two types of oracles representing different extremes. Maximum oracles monitor all internal and output variables, and output-only oracles monitor only the output variables. However, we have found that some level of oracle sensitivity could be overcome with intelligently constructed oracles [3]. We intend to further consider whether such oracles could be more effective in situations where observability constraints are too difficult for the test generator.

Selection of solver used for test generation: While conducting out study, we found that the model checker had difficulties with satisfying the observability constraints for some models. Further, we witnessed varying efficacy performance between the underlying solvers powering out employed test generation approach. We will extend our work in the future to quantify and further explore the choice of solver and its effect on suite efficacy.

Method of test generation: In this work, we have used model-based test generation. In past work, we also used Dynamic Symbolic Execution to generate test suites satisfying Observable MC/DC [12]. In the future, we would like to explore other methods of generating tests for observable criteria, such as search-based generation.

References

[1] G. Gay, A. Rajan, M. Staats, M. Whalen, and M. P. E. Heimdahl, "The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, pp. 25:1–25:34, July 2016.

[2] A. Rajan, M. Whalen, and M. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 161–170, ACM, 2008.

[3] G. Gay, M. Staats, M. Whalen, and M. Heimdahl, "Automated oracle data selection support," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[4] M. Staats, G. Gay, and M. Heimdahl, "Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing," in *Proceedings of the 2012 Int'l Conf. on Software Engineering*, pp. 870–880, IEEE Press, 2012.

[5] M. Staats, M. Whalen, and M. Heimdahl, "Better testing through oracle selection (nier track)," in *Proceedings of the 33rd Int'l Conf. on Software Engineering*, pp. 892–895, 2011.

[6] J. J. Chilenski and S. P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing," *Software Engineering Journal*, pp. 193–200, September 1994.

[7] RTCA, *DO-178B: Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.

[8] S. Vilkomir and J. Bowen, "Reinforced condition/decision coverage (RC/DC): A new criterion for software testing," *Lecture Notes in Computer Science*, vol. 2272, pp. 291–308, 2002.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Dataflow Programming Language Lustre," *Proceedings of the IEEE*, vol. 79, pp. 1305–1320, September 1991.

[10] M. Whalen, G. Gay, D. You, M. Heimdahl, and M. Staats, "Observable modified condition/decision coverage," in *Proceedings of the 2013 Int'l Conf. on Software Engineering*, ACM, May 2013.

[11] G. Gay, M. Staats, M. Whalen, and M. Heimdahl, "The risks of coverage-directed test case generation," *Software Engineering, IEEE Transactions on*, vol. PP, no. 99, 2015.

[12] D. You, S. Rayadurgam, M. Whalen, M. P. E. Heimdahl, and G. Gay, "Efficient observability-based test generation by dynamic symbolic execution," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 228–238, Nov 2015.

[13] E. Kit and S. Finzi, *Software Testing in the Real World: Improving the Process*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995.

[14] W. Perry, *Effective Methods for Software Testing, Third Edition*. New York, NY, USA: John Wiley & Sons, Inc., 2006.

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

[16] A. Groce, M. A. Alipour, and R. Gopinath, "Coverage and its discontents," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward!'14, (New York, NY, USA), pp. 255–268, ACM, 2014.

[17] S. Rayadurgam and M. Heimdahl, "Coverage based test-case generation using model checkers," in *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*, pp. 83–91, IEEE Computer Society, April 2001.

[18] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, pp. 105–156, 2004.

[19] N. Juristo, A. Moreno, and S. Vegas, "Reviewing 25 years of testing technique experiments," *Empirical Software Engineering*, vol. 9, no. 1, pp. 7–44, 2004.

[20] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," *Software Testing, Verification and Reliability*, vol. 23, no. 2, pp. 119–147, 2013.

[21] G. Gay, "The fitness function for the job: Search-based generation of test suites that detect real faults," in *Proceedings of the International Conference on Software Testing*, ICST 2017, IEEE, 2017.

[22] G. Fraser and A. Arcuri, "Whole test suite generation," *Software Engineering, IEEE Transactions on*, vol. 39, pp. 276–291, Feb 2013.

[23] RTCA/DO-178C, "Software considerations in airborne systems and equipment certification."

[24] J. Chilenski, "An investigation of three forms of the modified condition decision coverage (MCDC) criterion," Tech. Rep. DOT/FAA/AR-01/18, Office of Aviation Research, Washington, D.C., April 2001.

[25] P. Ammann and J. Offutt, *Introduction to Software Testing*. New York, NY, USA: Cambridge University Press, 2 ed., 2016.

[26] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?," in *FSE 2014, Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, (Hong Kong), pp. 654–665, November 18–20, 2014.

[27] J. Andrews, L. Briand, Y. Labiche, and A. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, pp. 608 –624, aug. 2006.

[28] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, 2014.

[29] R. Just, F. Schweiggert, and G. M. Kapfhammer, "Major: An efficient and extensible tool for mutation analysis in a java compiler," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, (Washington, DC, USA), pp. 612–615, IEEE Computer Society, 2011.

[30] R. A. D. Millo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, pp. 900–910, September 1991.

[31] B. Marick, "The weak mutation hypothesis," in *Proceedings of the Symposium on Testing, Analysis, and Verification*, TAV4, (New York, NY, USA), pp. 190–199, ACM, 1991.

[32] M. R. Woodward and K. Halewood, "From weak to strong, dead or alive? an analysis of some mutation testing issues," in *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, pp. 152–158, Jul 1988.

[33] "Mathworks Inc. Simulink." http://www.mathworks.com/products/simulink, 2015.

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

[35] Esterel-Technologies, "SCADE Suite product description." http://www.esterel-technologies.com/v2/ scadeSuiteForSafety-CriticalSoftwareDevelopment/index.html, 2004.

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

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

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

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

[40] M. Staats, G. Gay, and M. Heimdahl, "Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing," in *Proceedings of the 2012 International Conference on Software Engineering*, pp. 870–880, 2012.

[41] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking," *Journal on Software Tools for Technology Transfer*, vol. 4, February 2003.

[42] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM-efficient computation of observability-based code coverage metrics for functional verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, pp. 1003–1015, 2001.

[43] M. Felleisen and R. Hieb, "The revised report on the syntactic theories of sequential control and state," *Theor. Comput. Sci.*, vol. 103, pp. 235–271, Sept. 1992.

[44] G. Rosu and T. F. Serbanuta, "An overview of the k semantic framework," *J. of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397 – 434, 2010.

[45] J.-L. Colaço, B. Pagano, and M. Pouzet, "Scade 6: A formal language for embedded critical software development," in *International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2017.

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

[47] G. Gay, M. Staats, M. W. Whalen, and M. P. E. Heimdahl, "Moving the goalposts: Coverage satisfaction is not enough," in *Proceedings of the 7th International Workshop on Search-Based Software Testing*, SBST 2014, (New York, NY, USA), pp. 19–22, ACM, 2014.

[48] M. Staats, G. Gay, M. Whalen, and M. Heimdahl, "On the danger of coverage directed test case generation," in *Fundamental Approaches to Software Engineering* (J. de Lara and A. Zisman, eds.), vol. 7212 of *Lecture Notes in Computer Science*, pp. 409–424, Springer Berlin Heidelberg, 2012.

[49] G. Gay, S. Rayadurgam, and M. P. E. Heimdahl, "Automated steering of model-based test oracles to admit real program behaviors," *IEEE Transactions on Software Engineering*, vol. 43, pp. 531–555, June 2017.