The Effect of Program and Model Structure on the Effectiveness of MC/DC Test Adequacy Coverage

GREGORY GAY, University of South Carolina
AJITHA RAJAN, University of Edinburgh
MATT STAATS, Google, Inc.
MICHAEL WHALEN and MATS P.E. HEIMDAHL, University of Minnesota

Test adequacy metrics defined over the structure of a program, such as Modified Condition and Decision Coverage (MC/DC), are used to assess testing efforts. Unfortunately, MC/DC can be "cheated" by restructuring a program to make it easier to achieve the desired coverage. This is concerning, given the importance of MC/DC in assessing the adequacy of test suites for critical systems domains. In this work, we have explored the impact of implementation structure on the efficacy of test suites satisfying the MC/DC criterion using four real world avionics systems.

Our results demonstrate that test suites achieving MC/DC over implementations with structurally complex Boolean expressions are generally larger and more effective than test suites achieving MC/DC over functionally equivalent, but structurally simpler, implementations. Additionally, we found that test suites generated over simpler implementations achieve significantly lower MC/DC and fault finding effectiveness when applied to complex implementations, whereas test suites generated over the complex implementation still achieve high MC/DC and attain high fault finding over the simpler implementation. By measuring MC/DC over simple implementations, we can significantly reduce the cost of testing, but in doing so we also reduce the effectiveness of the testing process. Thus, developers have an economic incentive to "cheat" the MC/DC criterion, but this cheating leads to negative consequences. Accordingly, we recommend organizations require MC/DC over a structurally complex implementation for testing purposes to avoid these consequences.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools*

General Terms: Experimentation, Verification

Additional Key Words and Phrases: Coverage, Fault Finding

ACM Reference Format:
Gregory Gay, Ajitha Rajan, Matt Staats, Michael Whalen, and Mats P.E. Heimdahl. 2014. The Effect of Program and Model Structure on the Effectiveness of MC/DC Test Adequacy Coverage. *ACM Trans. Softw. Eng. Methodol.* 0, 0, Article 0 (2014), 31 pages.
DOI : http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Test adequacy metrics defined over the structure of a program, such as statement coverage, branch coverage, and decision coverage, have been used for decades to assess the adequacy of test suites. Of particular note is Modified Condition and Decision Coverage (MC/DC) criterion [Chilenski and Miller 1994], as it is used as an exit criterion when testing highly critical software in the avionics industry. For certification of such software, a vendor must demonstrate that the test suite provides MC/DC of the source code [RTCA 1992].

This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, NSF grants CNS-0931931 and CNS-1035715, and the L-3 Titan Group.
Author's addresses: G. Gay, Department of Computer Science & Engineering, University of South Carolina. greg@greggay.com; A. Rajan, School of Informatics, University of Edinburgh. ajitha.rajan@gmail.com; M. Staats, Google. Inc. staatsm@gmail.com; M. Whalen and M.P.E. Heimdahl, Department of Computer Science & Engineering, University of Minnesota. [whalen,heimdahl]@cs.umn.edu

DOI : http://dx.doi.org/10.1145/0000000.0000000

Unfortunately, it is well known that structural coverage criteria, including MC/DC, can easily be "cheated" by restructuring a program to make it easier to achieve the desired coverage [Rajan et al. 2008]. This is concerning: a straightforward way to reduce the difficulty of achieving MC/DC over a program is to introduce additional variables to factor complex Boolean expressions into simpler expressions.

We have examined the effect of program structure transformations by comparing test suites necessary to cover programs consisting of simple decisions (consisting of at most one logical operator) versus programs in which more complex expressions are used. We refer to these versions as the non-inlined and inlined programs, respectively.

To understand the effect of program structure on the MC/DC criterion, we have focused on four dimensions—the cost to produce a MC/DC-satisfying test suite, the fault finding effectiveness of the produced suite, the ability of that suite to cover other structural representations of that system, and the effectiveness of that suite when applied to other representations. To that end, we have produced inlined and non-inlined implementations of four real-world avionics systems, generated test cases over those structures, generated hundreds to thousands of mutants, and used those mutants and various test oracles to assess fault finding when those tests are executed.

From our results, we can conclude that program structure has a dramatic effect on not only the coverage achieved, but also the cost and fault finding effectiveness of the testing process. Test suites achieving MC/DC over inlined implementations are generally larger than test suites achieving MC/DC over non-inlined implementations, requiring 125.00%-1,566.66% more tests. While it is clearly more expensive to achieve MC/DC over the inlined system, this effort yields fault finding improvements of up to 4,542.47%. Test suites generated over non-inlined implementations achieve significantly less MC/DC when applied to inlined implementations, attaining 13.08%-67.67% coverage on the more complex implementation. We also found that test suites generated over non-inlined implementations cannot be expected to yield effective fault finding on inlined implementations, finding 17.88-98.83% fewer faults than tests generated and executed on the inlined implementation. On the other hand, tests generated using the inlined implementation generally attained 100% MC/DC on the non-inlined system, and found up to 5,068.49% more faults than tests generated and executed on the non-inlined implementation.

Given that the inlined and non-inlined programs are semantically equivalent, the degree to which this simple transformation influences the effectiveness of the MC/DC criterion is cause for concern. We believe these results are concerning, particularly in the context of certification: we have demonstrated that by measuring MC/DC over simple implementations, we can significantly reduce the cost of testing, but in doing so we also reduce the effectiveness of testing. Thus developers have an economic incentive to "cheat" the MC/DC criterion (and by building tools similar to those used in our study, the means), but this cheating leads to negative consequences.

Based on these results, we strongly recommend that organizations using MC/DC metric in their software development efforts measure MC/DC on an inlined version of the system under test. While developers may choose to create non-inlined versions of the program with numerous intermediate variables for clarity or efficiency, such programs make it significantly easier to achieve MC/DC, with negative implications for testing effectiveness. In the absence of a metric that is robust to structural changes in the system under test, the use of tools, similar to those used in our study, can significantly improve the quality of the testing process.

This work is an expansion of two previous papers in which we studied the effect of program structure on *coverage* of MC/DC obligations [Rajan et al. 2008; Heimdahl et al. 2008]. In the first study, we examined how well a test suite generated to achieve maximum achievable coverage over a non-inlined system achieves coverage over an inlined system [Rajan et al. 2008]. Although this coverage result is important, the utility of a coverage metric is ultimately defined by how well test suites satisfying that metric are able to detect faults in the code. Thus, in the second study, we examined the effectiveness of test suites generated over inlined and non-inlined implementations on a set of seeded mutations [Heimdahl et al. 2008]. This report repeats those experiments using an improved experimental framework and updated definitions of the MC/DC obligations. We have

Version 1: Non-inlined Implementation

```
expr_1 = in_1 or in_2;      //stmt1
out_1 = expr_1 and in_3;    //stmt2
```

Version 2: Inlined Implementation

```
out_1 = (in_1 or in_2) and in_3;
```

Sample Test Sets for (in_1, in_2, in_3):

```
TestSet1 = {(TFF),(FTF),(FFT),(TTT)}
TestSet2 = {(TFT),(FTT),(FFT),(TFF)}
```

Fig. 1. Example of behaviorally equivalent implementations with different structures

conducted more rigorous studies, including: generating many more test suites for each system structure using more advanced test generation techniques, using most—if not all—mutants instead of a small random sampling, considering an additional type of test oracle—the largest common oracle—and performing more thorough statistical analyses. Our expanded and re-executed studies confirm the trends observed in the results of the previous work in a more rigorous fashion, and allow us to discuss the implications of our results in more detail. Additionally, we have added a fourth analysis to our experiment studying the effect of using one structure for test generation and another for test execution.

The remainder of the paper is organized as follows. In the next section, we provide background on MC/DC and problems related to program structure. Section 3 introduces our experimental setup and the case examples used in our investigation. Results and statistical analyses are presented in Section 4. In Section 5, we summarize our findings and their implications for practice. Section 6 discussed threats to validity. Finally, Section 7 discusses related work and we conclude in Section 8.

2. BACKGROUND AND MOTIVATION

A test suite provides MC/DC over the structure of a program or model if every condition within a decision has taken on all possible outcomes at least once, and every condition has been shown to independently affect the decision's outcome (note here that when discussing MC/DC, a decision is defined to be an expression involving any Boolean operator).

Independent effect is defined in terms of *masking*, which means that the condition has no effect on the value of the decision as a whole. As an example, consider the trivial program fragments in Figure 1. The program fragments have different structures but are functionally equivalent. Version 1 is non-inlined with intermediate variable `expr_1`, and Version 2 is inlined with no intermediate variables. Given a decision of the form `in_1 or in_2`, the truth value of `in_1` is irrelevant if `in_2` is true, so we state that `in_1` is masked out. A condition that is not masked out has *independent effect* for the decision.

Based on the definition of MC/DC, `TestSet1` in Figure 1 provides MC/DC over program Version 1 but not over Version 2; the test cases with `in_3 = false` contribute towards MC/DC of the expression `in_1 or in_2` in Version 1 but not over Version 2 since the masking effect of `in_3 = false` is revealed in Version 2.

In contrast, MC/DC over the inlined version requires a test suite to take the masking effect of `in_3` into consideration as seen in `TestSet2`. This disparity in MC/DC over the two versions can have significant ramifications with respect to fault finding of test suites. Suppose the code fragment in Figure 1 is faulty and the correct expression should have been `in_1` **`and`** `in_2` (which was erroneously coded as `in_1` **`or`** `in_2`). `TestSet1` would be incapable of revealing this fault since there would be no change in the observable output, `out_1`. On the other hand, any test set providing MC/DC of the inlined implementation would be able to reveal this fault.

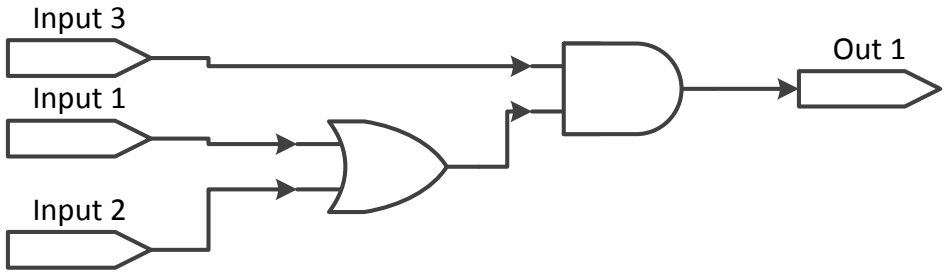

Fig. 2. Simulink model of example in Figure 1

Programs may be structured with significant numbers of intermediate variables for many reasons, for example, for clarity (nice program structure), efficiency (no need to recompute commonly used values), or to make it easier to achieve the desired MC/DC (MC/DC tests are easier to find if the decisions are simple). The programs may be restructured automatically via machine transformation, or they may be structured by the developers to improve program comprehension.

The potential problems with decision structure are not confined only to code. The move towards Model-based Development in the critical systems community makes test-adequacy measurement a crucial issue in the modeling domain. Coverage criteria such as MC/DC are being used in conjunction with modeling tools such as Simulink [Mathworks Inc. 2015] and SCADE [Esterel-Technologies 2004] for testing models. Currently, MC/DC measurement over models in these tools is being done in the weakest possible manner [Mathworks Documentation 2015]. For example, Figure 2 is a Simulink model equivalent to the example in Figure 1. MC/DC of such models is currently defined on a "gate level" (analogous to the MC/DC measurement over Version 1 in Figure 1). Since there are no complex decisions in this definition of MC/DC, MC/DC measured this way is susceptible to the masking problem discussed above, and test suites designed to provide MC/DC over the models may therefore provide poor fault finding capability. Thus, the current approach to measuring MC/DC over such models is cause for concern. For simplicity, in the remainder of this report, we refer to a "model" or "program" as the "implementation" since the concerns discussed here are the same regardless of whether we are discussing a model or a program.

Given the economic incentive to change implementation structure to ease certification, and the natural tendency of developers to simplify system structure, we believe understanding how implementation structure impacts the effectiveness of the MC/DC criterion is worthwhile. In particular, we are concerned that using a simpler implementation structure may be detrimental to the effectiveness of the testing process. In particular, we would like to know (1) how implementation structure impacts the coverage achieved by test suites built to satisfy MC/DC across differing implementation structures, and (2) how constructing test suites to satisfy MC/DC for different implementation structures impacts fault finding effectiveness.

3. STUDY OBJECTIVES AND DESIGN

To investigate how MC/DC is affected by the structure of an implementation, we explored three key research questions:

— **Question 1 (Q1):** How does the implementation structure impact the cost, as measured by the number of test inputs required, of achieving MC/DC?
— **Question 2 (Q2):** How does the implementation structure impact the effectiveness, as measured by the number of faults detected, of test suites achieving MC/DC?
— **Question 3 (Q3):** How well do test suites generated over a structurally simple implementation satisfy MC/DC for a semantically equivalent, but structurally complex implementation? Similarly, how well do test suites generated over the structurally complex implementation satisfy MC/DC for the simpler non-inlined implementation?

— **Question 4 (Q4):** How effective are test suites generated over a structurally simple implementation at detecting faults in a semantically equivalent, but structurally complex implementation? Similarly, how effective are test suites generated over the structurally complex implementation at detecting faults for the simpler non-inlined implementation?

The first two questions address how varying implementation structure impacts two key questions related to a test coverage criterion: *how much does it cost?*, and *how effective are tests suites that satisfy it?* Such questions are relevant when assessing the practical differences of cost and effectiveness that occur when varying the implementation structure. The third question is relevant in the context of MC/DC's role in certification of software. We would like to know if measuring coverage over a structurally simple implementation (e.g., as represented in Simulink) instead of more structurally complex implementation (e.g., as written by developers in C code) results in different measurements, as lower coverage results may imply inadequate test data. A related question is whether the source of the test suites impacts the effectiveness of those tests when those tests are executed against a different structure. If tests are generated using a structurally simple implementation, will they be effective at detecting faults in the structurally complex implementation?

Note that while the answers to these questions are likely to be related, as demonstrated by work exploring the impact of test suite size and structural coverage on fault finding effectiveness [Inozemtseva and Holmes 2014; Namin and Andrews 2009], a practically significant relationship is not guaranteed. It is trivial to construct small examples in which changing the structure has a strong impact on fault finding effectiveness, the number of test inputs required, etc. However, it is possible that—in practice—substantial differences in MC/DC correspond to only insignificant differences in fault finding. It is, therefore, important to quantify the degree to which implementation structure can impact the cost and effectiveness of test suites satisfying MC/DC in practice, as it is this information that should drive decision making.

3.1. Choice of MC/DC Criterion

Chilenski investigated three different notions of MC/DC [Chilenski 2001], namely: Unique-Cause MC/DC, Unique-Cause + Masking MC/DC, and Masking MC/DC. In this report we use *Masking* MC/DC [Hayhurst et al. 2001] to determine the independence of conditions within a Boolean expression. In Masking MC/DC, a basic condition is *masked* if varying its value cannot affect the outcome of a decision due to the structure of the decision and the value of other conditions. To satisfy masking MC/DC for a basic condition, we must have test states in which the condition is not masked and takes on both $true$ and $false$ values.

Unique-Cause MC/DC requires a unique cause—when a single condition is flipped, the result of the expression changes—to independent impact. This means that, when faced with strongly-coupled conditions in complex decision statements, there may be situations where no test can establish a unique cause. Unique Cause + Masking MC/DC relaxes the unique cause requirement to all uncoupled conditions, and Masking MC/DC allows masking in all cases. Masking MC/DC is the easiest of the three forms of MC/DC to satisfy since it allows for more independence pairs per condition and more coverage test sets per expression than the other forms. In contrast, Chilenski's analysis showed that even though Masking MC/DC could allow fewer tests than Unique-Cause MC/DC, its performance in terms of probability of error detection was nearly identical to the other forms of MC/DC. This led Chilenski to conclude in [Chilenski 2001] that Masking MC/DC should be the preferred form of MC/DC.

To better illustrate the definition of masking MC/DC, consider the expression $A\ and\ B$. To show the independence of B, we must hold the value of A to $true$; otherwise varying B will not affect the outcome of the expression. Independence of A is shown in a similar manner. Table I shows the test suite required to satisfy MC/DC for the expression $A\ and\ B$. When we consider decisions with multiple Boolean operators, we must ensure that the test results for one operator are not masked out by the behavior of other operators. For example, given $A\ or\ (B\ and\ C)$ the tests for $B\ and\ C$ will not affect the outcome of the decision if A is $true$. Table II gives one of the test suites that would

Table I. Example of a test suite that provides Masking MC/DC over $A\ and\ B$

A	B	$A\ and\ B$
T	T	T
T	F	F
F	T	F

satisfy masking MC/DC for the expression $A\ or\ (B\ and\ C)$. Note that in Table II, test cases {2,4} or {3,4} will demonstrate independence of condition A under Masking MC/DC. However, these test cases will not be sufficient to show independence of A under Unique Cause MC/DC since the values for conditions B and C are not fixed between the test cases.

Table II. Example of a test suite that provides Masking MC/DC over $A\ or\ (B\ and\ C)$

A	B	C	$A\ or\ (B\ and\ C)$
F	T	T	T
F	T	F	F
F	F	T	F
T	F	F	T

3.2. Experimental Design

In our study, we performed the following steps for each case example:

— **Generated inlined and non-inlined implementation versions:** We transformed the implementation structure, creating one version with a high degree of expression complexity (inlined version) and one version with a low degree of expression complexity (non-inlined version). This is detailed in Section 3.4. Note that unless specified, the remaining steps are applied to both the inlined and non-inlined version separately (e.g., separate mutant sets are generated for the inlined and non-inlined versions).
— **Generated mutants:** We generated the full pool of mutants, each containing a single fault, for each system. We then removed functionally equivalent mutants. (Section 3.5.)
— **Generated tests satisfying the MC/DC criterion:** We generated a set of tests satisfying the MC/DC criterion. The JKind model checker [Hagen 2008; Gacek 2015] was used to generate each set of tests, resulting in a test set with one test case for each obligation. (Section 3.6.)
— **Generated reduced test suites:** We generated 100 reduced test suites using a greedy reduction algorithm. Each reduced test suite maintains the coverage provided by the full test suite. (Section 3.6.)
— **Selected oracles:** We used two test oracles in our study: an output-only oracle considering all outputs, and a maximum oracle considering all internal variables and all outputs. (Section 3.7.)
— **Ran tests on mutants:** We ran each mutant and the original case example using every test generated, and collected the internal state and output variable values produced at every step. This yields raw data used for assessing fault finding in our study. (Section 3.8.)
— **Assessed fault finding ability of each oracle and test suite combination:** We determined how many mutants were detected by every oracle and a reduced test suite combination. (Section 3.8.)

We generated test obligations, extracted concrete test cases from JKind counter-examples, and executed the tests on the mutants using an in-house suite of tools designed for testing models written in the Lustre synchronous language. This framework is open-source and freely available from https://github.com/djyou/lustre.

3.3. Case Examples

We used four industrial synchronous reactive systems developed by Rockwell Collins Inc. in our experiment. Information related to these systems is provided in Table III. We list the number of Simulink subsystems, which are analogous to functions, and the number of blocks, analogous to operators. We also list the number of calculated expressions in the Lustre implementations.

Table III. Case example information. NI = non-inlined, I = inlined

	# Simulink Subsystems	# Blocks	Lustre Expressions NI	Lustre Expressions I
DWM_1	3,109	11,439	576	28
DWM_2	128	429	121	19
Vertmax_Batch	396	1,453	417	31
Latctl_Batch	120	718	129	20

All four systems were modeled using the Simulink notation from Mathworks Inc. [Mathworks Inc. 2015] and were translated to the Lustre synchronous programming language [Halbwachs 1993] to take advantage of existing automation. This is analogous to the automated code generation done from Simulink using Real Time Workshop from Mathworks. 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.

3.3.1. Flight Guidance System. A Flight Guidance System (FGS) in an aircraft compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generates pitch and roll-guidance commands to minimize the difference between those two states. The FGS consists of the mode logic, which determines what lateral and vertical modes of operation are active at any given time, and the flight control laws that are used to compute the pitch and roll guidance commands. The two FGS systems used in this work, *Vertmax_Batch* and *Latctl_Batch*, describe the vertical and lateral mode logic for the Rockwell Collins FCS 5000 flight guidance system family.

3.3.2. Display Window Manager. The Display Window Manager (DWM) models, *DWM_1* and *DWM_2*, represent two of the five major subsystems of the DWM of the Rockwell Collins ADGS-2100, an air transport-level commercial display system. The DWM acts as a "switchboard" for the system, routing information to the aircraft displays and managing the location of two cursors that can be used to control applications by the pilot and copilot. The DWM must update the applications being displayed in response to user selections and must handle reversion in case of hardware application failures. The DWM systems decide what information is most critical and moves this information to the remaining displays.

3.4. Implementation Structure

For each case example, we generate two versions that are semantically equivalent, but syntactically different. We term these the *inlined* and *non-inlined* versions of the implementation, described below.

3.4.1. Non-inlined Implementation Structure. In a non-inlined implementation, the structure of the implementation is similar to the structure of the original Simulink model. Each signal from the Simulink model has been preserved, resulting in a very large number of internal state variables, with each internal state variable corresponding to a relatively simple expression. A small example of a non-inlined implementation is given in Figure 3.

Unlike our previous studies exploring program structure [Rajan et al. 2008; Heimdahl et al. 2008], in this study, the structure is flattened such that the implementation has only one Lustre node. In Lustre, a node is a subprogram—a single callable block of code. If the model consists of multiple nodes, all of the nodes are combined into a single node (i.e., the code of external functions is imported into the calling block). This node flattening is conducted for compatibility purposes with experimental infrastructure and does not impact the results of the study.

```
node exampleProgramNode(input_1: bool;
      input_2: bool;
      input_3: int;
      input_4: bool)
   returns (output_1: bool);
var
   internal_1: bool;
   internal_2: bool;
   internal_3: bool;
   internal_4: bool;
   internal_5: bool;
   internal_6: bool;
let
   internal_1 = input_1 AND input_2;
   internal_2 = input_3 > 100;
   internal_3 = internal_1 OR input_4
   internal_4 = IF (internal_3) THEN internal_2 ELSE input_1;
   internal_5 = input_3 > 50;
   internal_6 = IF (internal_5) THEN internal_4 ELSE input_2;
   output_1 = internal_6;
tel;
```

Fig. 3. Example non-inlined implementation.

3.4.2. Inlined Implementation Structure. As with the non-inlined implementation, in the inlined implementation, the structure is first flattened such that the implementation has only one Lustre node. Once this has been completed, we then inline most, but not all, of the intermediate variables into the model. When *inlining* a variable, we substitute the expression corresponding to the variable wherever the variable is referenced, and then remove the variable from the implementation (as it is no longer referenced). This has the effect of (1) reducing the number of internal state variables, as inlined variables are removed from the model, (2) increasing the complexity of expressions, as inlined variables are substituted wherever they are referenced, and (3) increasing the number of nested `if-then-else` expressions, as such expressions are often substituted into `then` and `else` branches.

MC/DC is defined exclusively over traditional imperative structures (such as those found in C, Java, etc.); we therefore restrict our inlining to prevent syntactic constructs impossible in imperative implementations from arising. In particular, we do not place `if-then-else` expressions inside `if` conditions. In Lustre, a statement of the form `result = if (if (internal_1 and internal_2) then internal_3 else internal_4) then true else false)` is a valid expression. In an imperative language, the `if` condition would need to be contained in a separately evaluated expression, or through the use of the ternary operator. MC/DC, as traditionally defined, does not have a definition for the ternary operator [Hayhurst et al. 2001]. While such a definition could be added, we have chosen to stay within the MC/DC definitions commonly used for imperative languages. Therefore, our use of `if-then-else` expressions in Lustre corresponds directly with MC/DC-compliant `if-then-else` statements in C.

In Figure 4, we present an inlined version of the implementation from Figure 3. As we can see, the inlined version has been reduced from five internal state variables to zero, with the set of expressions condensed to one, considerably more complex, expression. This expression illustrates how inlining can increase complexity both in terms of Boolean/relational expressions and nesting of `if-then-else` statements. For example, we see that the condition formerly represented by `internal_3` has been inlined, and the `if-then-else` expression formerly represented by `internal_4` has been nested inside another `if-then-else` statement. As we will see shortly,

```
node exampleProgramNode(input_1: bool;
      input_2: bool;
      input_3: int;
      input_4: bool)
   returns (output_1: bool);
let
   output_1 =
       IF (input_3 > 50)
       THEN
           IF ((input_1 AND input_2) OR input_4)
           THEN input_3 > 100
           ELSE input_1 ELSE input_2;
tel;
```

Fig. 4. Example inlined implementation.

while these transformations result in semantically equivalent implementations, their impact on testing is significant.

3.5. Mutant Generation

We have created *mutations* (faulty implementations) of each case example by automatically introducing a single fault into the correct implementation. Each fault was seeded by either inserting a new operator into the system or by replacing an existing operator or variable with a different operator or variable. The type of faults used to create mutants may impact the effectiveness of the selected oracle data when used to test the actual 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 [Andrews et al. 2006]—and similar to those used by Just et al., who found a significant correlation between mutant detection and real fault detection [Just et al. 2014].

We seed the following classes of faults:

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

For each system, we generated all mutants that could possibly be created using the selected mutation operators. The number and types of mutants are listed in Table IV. One risk of mutation testing is *functionally equivalent* mutants, in which faults exist but these faults cannot cause a *failure*, which is an externally visible deviation from correct behavior. For our study, we used model checking to detect and remove functionally equivalent mutants. This is made possible due to our use of synchronous reactive systems as case examples—each system is finite, and thus determining equivalence is decidable. (Equivalence checking is fairly routine on the hardware side of the synchronous reactive system community; Van Eijk provides a good introduction [Van Eijk 2002].) Thus for every mutant used in our study, there exists at least one trace that can lead to a user-visible failure, and all fault finding measurements indeed measure actual faults detected.

In order to make fair fault finding comparisons between implementations, we used *min(noninlined,inlined)* mutants in our experiments. For the implementation where we did not use

Table IV. Mutant information for each case example. NI = non-inlined, I = inlined

		DWM_1		DWM_2		Vertmax_Batch		Latctl_Batch	
		NI	I	NI	I	NI	I	NI	I
	Total Mutants	8,489	13,303	931	1,858	4,411	6,712	1,036	954
	Equivalent Mutants	344	919	15	61	152	182	40	42
	Total Used	8,145	8,145	916	916	4,259	4,259	912	912
Arithmetic	**Total**	8	277	0	0	0	0	0	0
	Equivalent	0	19	0	0	0	0	0	0
	Used	8	170	0	0	0	0	0	0
Relational	**Total**	691	792	14	73	15	231	5	36
	Equivalent	83	99	0	1	1	30	1	16
	Used	608	458	14	37	14	131	4	20
Boolean	**Total**	11	111	44	189	452	861	64	85
	Equivalent	0	25	0	9	99	95	4	6
	Used	11	57	44	92	353	500	55	79
Negation	**Total**	1,041	2,004	290	574	1,443	2,187	333	303
	Equivalent	18	81	7	18	17	20	14	9
	Used	1,023	1,270	283	283	1,426	1413	292	294
Delay	**Total**	3,466	5,081	337	671	1,511	2,432	373	352
	Equivalent	21	128	0	22	1	15	8	3
	Used	3,445	3,270	337	331	1,510	1576	334	349
Constant	**Total**	1,522	3,167	54	112	84	300	41	84
	Equivalent	149	395	7	0	16	17	7	7
	Used	1,373	1,830	47	57	68	185	31	77
Replacement	**Total**	1,750	1,824	192	239	906	701	220	94
	Equivalent	73	172	1	11	18	5	6	1
	Used	1,677	1,091	191	116	888	454	196	93

the full pool of mutants, we selected mutants so that the *fault ratio* for each fault class was approximately uniform. That is, assume for some example, there are R possible Relational faults and B possible Boolean faults. For a uniform fault ratio, we would seed x relational faults and y Boolean faults in the implementation so that $x/R = y/B$. For example, if there are 88 possible Boolean faults and 12 possible relational faults, and we wanted to select 25 mutants, we could select 0.25 as our fault ratio and our resulting set would contain 22 Boolean faults and 3 relational faults. This uniform ratio means that—despite setting a controlled number of mutants—we do not bias the distribution of fault types used in our experiments. The number of mutants for each type of fault reflects the distribution if all possible mutants were used.

3.6. Coverage Directed Test Input Generation

There exist several methods of generating tests to satisfy coverage criteria. We adopt counterexample-based test generation approaches based on existing model checking approaches to generate tests satisfying the MC/DC criterion [Gargantini and Heitmeyer 1999; Rayadurgam and Heimdahl 2001]. We have used the JKind model checker in our experiments [Hagen 2008; Gacek 2015].

We performed the following steps separately using the inlined and non-inlined case examples:

(1) We processed the Lustre source code to generate *coverage obligations* for MC/DC. Each coverage obligation represents a property that should be satisfied by some test (e.g., some branch covered, some condition evaluates to true).
(2) These obligations were inserted into the Lustre source code as *trap properties*—that is, the negation of the properties [Gargantini and Heitmeyer 1999]. Essentially, by asserting that these obligations can *never* be made true, the model checker can produce a counterexample showing how the property can be met.
(3) We ran the model and properties through JKind. This produces a list of counterexamples, each of which corresponds to the satisfaction of some coverage obligation.
(4) Each counterexample is translated into a test input.

Table V. Unreduced test suite measurements. NI = non-inlined, I = inlined.

	NI # of MC/DC Obligations	% NI MC/DC Obligations Achievable	I# of MC/DC Obligations	% I MC/DC Obligations Achievable
DWM_1	2038	100.00%	3806	98.74%
DWM_2	530	98.68%	2192	100.00%
Vertmax_Batch	1732	100.00%	1858	96.99%
Latctl_Batch	380	100.00%	260	99.62%

Note that some coverage obligations are unsatisfiable, i.e., there does not exist a test that satisfies the coverage obligation. (This can occur, for example, if infeasible combinations of conditions are required by some coverage obligation.) Nevertheless, by using this approach, we ensure that the maximum number of satisfiable obligations are satisfied. A listing of the number of obligations and the percent of achievable coverage can be found in Table V.

This approach generates a separate test for each coverage obligation. While a simple method of generating tests, in practice, this results in a large amount of redundancy in the tests generated, as each test likely covers several coverage obligations. For example, in branch coverage, a test satisfying an obligation for a nested branch will also satisfy obligations for outer branches. Consequently, the test suite generated for each coverage criterion is generally much larger than is required to satisfy the coverage criterion. Given the correlation between test suite size and fault finding effectiveness [Namin and Andrews 2009], this has the potential to yield misleading results—unnecessarily large test suites may lead us to conclude that a coverage criterion yields an effective test suite, when, in reality, it is the size of the test suite that is responsible for the effectiveness.

To avoid this, we reduce each naive test suite generated while maintaining the coverage achieved. This reduction is done using a simple greedy algorithm following these steps:

(1) Each test is executed, and the coverage obligations satisfied by each test are recorded. This provides the coverage information used in subsequent steps.
(2) We initialize two empty sets: obs contains satisfied obligations and $reduced$ contains our reduced test suite. We initialize a set $tests$ containing all tests from our naive test suite.
(3) We randomly select and remove a test t from $tests$. If the test satisfies an obligation not currently in obs, we add it to $reduced$, and add the obligations satisfied by t to obs. Otherwise we discard the test.
(4) We repeat the previous step until $tests$ is empty or all obligations have been satisfied.

Due to the randomization, the size and contents of reduced suites may vary quite a bit between suite generations. For example, due to the random test selection, this reduction approach may choose ten tests that each cover a single new obligation when there may exist a single test that covers all ten. Thus, while this reduction approach results in smaller test suites, it is not guaranteed to produce the smallest possible test suites. To prevent us from selecting a test suite that happens to be exceptionally good or exceptionally poor relative to the set of possible reduced test suites, we produce 100 randomly reduced test suites using this process.

3.7. Test Oracles

In our study, we use what are known as *expected value oracles* as our test oracles [Staats et al. 2012a]. 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 types of oracles: an *output-only oracle* that defines expected values for all outputs, and a *maximum oracle* that defines expected values for all outputs and all internal state variables. The output-only oracle represents the oracle most likely to be used in practice, whereas the

maximum oracle represents an idealistic scenario where we can monitor each expression assignment in the system. In practice, the maximum oracle is often prohibitively expensive to specify. However, it offers the ability to assess the effect of oracle selection on fault finding [Gay et al. 2015a; Staats et al. 2012a].

Table VI. Oracle sizes for each case example and implementation

	Implementation	# Output Variables	Total # Variables
DWM_1	Inlined	7	28
	Non-inlined	7	576
DWM_2	Inlined	9	19
	Non-inlined	9	124
Vertmax_Batch	Inlined	2	32
	Non-inlined	2	417
Latctl_Batch	Inlined	1	20
	Non-inlined	1	129

The number of variables in each oracle for the inlined and non-inlined implementations can be seen in Table VI. Regardless of implementation, the number of variables in the output-only oracle will remain the same. However, as a result of the inlining process, the number of expressions in the inlined implementation will be far smaller than in the simple non-inlined implementation. Thus, the size of the maximum oracle will be far smaller for the inlined implementation. We will explore the implications of oracle size in the following section.

3.8. Data Collection

After generating the full test suites and mutant set for a given case example, we ran each test suite against every mutant and the original case example. For each run of the test suite, we recorded the value of every internal variable and output at each step of every test using the Lustre interpreter in our test generation framework. This process yielded a complete set of values produced by running the test suite against every mutant and the correct implementation for each case example.

To determine the fault finding of a test suite t and oracle o for a case example we simply compare the values produced by the original case example against every mutant using (1) the subset of the full test suite corresponding to the test suite t and (2) the subset of variables corresponding to the oracle data for oracle o. The fault finding effectiveness of the test suite and oracle pair is computed as the number of mutants detected (or "killed") divided by the total number of non-equivalent mutants used (see Table IV). We perform this analysis for each oracle and test suite for every case example yielding a very large number of measurements per case example. We use the information produced by this analysis in later sections to explore our research questions.

4. RESULTS

For each case example, we began with a large test suite that achieved the maximum achievable fault finding for the criterion. We then subsequently used a randomized test suite reduction algorithm to generate 100 reduced test suites. Finally, we computed the size and fault finding of each resulting test suite on our pool of non-equivalent mutants using both test oracles.

Using these numbers, we measured the following:

— **Original Number of Obligations:** The number of MC/DC obligations. This is usually slightly larger than the size of the original unreduced test suite, as unachievable obligations have no corresponding test input.
— **Percent Achievable Obligations:** The percent of MC/DC obligations that can be covered. This coverage is maintained during test suite reduction.
— **Median Fault Finding:** Median fault finding across reduced test suites.
— **Median Test Suite Size:** Median test suite size across reduced test suites.

Table VII. Median (μ) reduced test suite size across case example.

	Non-inlined μ Size	Inlined μ Size	% Size Increase
DWM_1	27.00	450.00	1,566.66%
DWM_2	69.00	250.00	262.32%
Vertmax_Batch	68.00	334.00	391.18%
Latctl_Batch	28.00	63.00	125.00%

Table VIII. Median (μ) fault finding measurements across case example. FF = fault finding, OO = output-only oracle, MX = maximum oracle

	Non-inlined μ FF (OO)	Inlined μ FF (OO)	% FF Increase (OO)
DWM_1	1.46%	67.78%	4,542.47%
DWM_2	85.04%	88.81%	4.43%
Vertmax_Batch	51.22%	85.60%	67.12%
Latctl_Batch	61.02%	74.94%	22.81%
	Non-inlined μ FF (MX)	**Inlined μ FF (MX)**	**% FF Increase (MX)**
DWM_1	72.28%	68.01%	-6.28%
DWM_2	94.54%	88.97%	-5.89%
Vertmax_Batch	92.32%	85.68%	-7.19%
Latctl_Batch	92.33%	78.07%	-15.44%

— **Relative Change in Median Test Suite Size:** We measure the relative increase (as a percentage) in median test suite size when generating tests using the inlined implementation as compared to when generating tests using the non-inlined implementation. Positive percentages indicate that the median test suite size is larger for the inlined implementation.
— **Relative Change in Median Fault Finding:** We measure the relative increase (as a percentage) in median test suite fault finding effectiveness when generating tests using the inlined implementation as compared to when generating tests using the non-inlined implementation. Positive percentages indicate that the median effectiveness is larger when generating test suites using the inlined implementation.
— **Median Coverage of Non-inlined Suites over Inlined Implementation:** We measure the median MC/DC achieved when using test suites generated from the non-inlined system over the inlined implementation. 100% indicates no change; the test suite achieves 100% of the achievable MC/DC over both systems.
— **Median Coverage of Inlined Suites over Non-inlined Implementation:** We measure the median MC/DC achieved when using test suites generated from the inlined system over the non-inlined implementation.
— **Median Fault Finding of Non-inlined Suites over Inlined Implementation:** We measure the median fault finding across reduced test suites generated from the non-inlined implementation over the inlined implementation.
— **Median Fault Finding of Inlined Suites over Non-inlined Implementation:** We measure the median fault finding across reduced test suites generated from the inlined implementation over the non-inlined implementation.

The measurements for unreduced test suites are given in Table V, and the measurements for reduced test suites are given in Tables VII (test suite size), VIII and IX (fault finding), X (coverage across implementation types), and XI (fault finding across implementation types).

We can see four patterns. First, per *Q1* (see Table VII), the number of test inputs required to satisfy MC/DC over the inlined implementation is often significantly higher than the number of test inputs required to satisfy MC/DC over the non-inlined implementation, with increases of 125.00%-1,566.66%. Thus, the cost of satisfying MC/DC, in terms of the number of test inputs we must create and execute, tends to increase as the implementation structure's complexity increases.

Second, per *Q2* (see Table VIII), the effectiveness of test suites achieving maximum MC/DC also varies with implementation structure. When making use of the common output-only oracle, test suites satisfying MC/DC over the inlined structure outperform those satisfying MC/DC over

Table IX. Measurements across case example for the largest oracle common to both implementations. μ = median, FF = fault finding, LCO = largest common oracle

	Non-inlined μ FF (LCO)	Inlined μ FF (LCO)	% FF Increase (LCO)
DWM_1	5.27%	68.01%	1,190.51%
DWM_2	85.26%	88.97%	4.35%
Vertmax_Batch	53.48%	85.68%	60.21%
Latctl_Batch	67.00%	78.07%	16.52%

the non-inlined structure, with relative differences ranging from a slight increase (4.43%) to a large increase (4,542.47%) in median fault finding.

Fault finding results when making use of the maximum oracle demonstrate the opposite effect—test suites that satisfy MC/DC over the non-inlined structure slightly outperform those that satisfy MC/DC over the inlined structure, finding between 5.89 and 15.44% *fewer* mutants. However, this can easily be explained by the size of the maximum oracles for each implementation, as noted in Table VI. Because of the inlining process, the maximum oracles for the inlined systems are far smaller than those for the non-inlined systems. The inlined systems calculate and store the results of a smaller number of expressions. These additional points of observation give the non-inlined implementation a clear advantage in observing faults when using an oracle that checks the behavior of all possible observation points.

In practice, however, using such a large oracle would be prohibitively expensive, as the tester would have to explicitly specify the expected behavior for each variable. Even if they did that, the monitoring overhead from observing all of those variables may negatively impact the behavior of the system. If testing software on an embedded system, a tester may not even be able to observe all of the internal variables. Thus, although the maximum oracle for non-inlined systems does yield effective fault finding potential, its use may not be practical.

As the non-inlined implementation always contains more variables than the inlined implementation, the non-inlined implementation can always have an oracle that checks the behavior at more points of observation than the inlined implementation. A more equal comparison would be to compare the fault finding effectiveness of tests generated from both implementations with the largest possible fixed-sized oracle with variables that appear in both implementations (dubbed the **largest common oracle**). If we use the maximum oracle for the inlined implementation to compare fault finding effectiveness, we can increase the number of observation points while assessing the fault finding capability of the generated tests on a more equivalent basis. The results for the largest common oracle can be seen in Table IX.

The effectiveness of test suites achieving maximum MC/DC paired with the largest common oracle is similar to that using the output-only oracle, with relative differences ranging from a 4.35 to 1,190.51% improvement in median fault finding when using tests generated over the inlined implementation. While the improvement in fault finding is more subdued than when using the output-only oracle, we can still conclude that differences in fault finding roughly correspond to the previously noted differences in test suite cost, with larger—but more effective—test suites resulting from the use of more complex decisions in the implementations.

Third, per *Q3* (see Table X), we can see that test suites achieving 100% achievable MC/DC of the non-inlined implementations always achieve less than 100% achievable coverage over the inlined implementations, with coverage ranging from 13.08% to 67.67%. The reverse is not true, test suites that achieve 100% achievable MC/DC of the inlined implementation generally achieve close to, if not, 100% MC/DC of the non-inlined implementation. This highlights that not only does the implementation structure impact the cost and effectiveness of test suites satisfying the MC/DC criterion, it also can have a strong impact on the coverage achieved. We cannot expect to satisfy MC/DC on a structurally simple implementation, and still achieve 100% coverage (or even similar coverage levels) on a semantically equivalent—but more structurally complex—implementation.

Finally, to address *Q4*, we have examined the effect of implementation structure on test effectiveness in more detail. We have executed tests generated over the non-inlined implementation on the

Table X. Median (μ) coverage results across implementation type

	μ % Achievable MC/DC of Non-inlined Suites over Inlined Implementation
DWM_1	13.08%
DWM_2	53.25%
Vertmax_Batch	31.95%
Latctl_Batch	67.67%
	μ % Achievable MC/DC of Inlined Suites over Non-inlined Implementation
DWM_1	100.00%
DWM_2	100.00%
Vertmax_Batch	100.00%
Latctl_Batch	99.31%

Table XI. Median (μ) fault finding measurements across implementation type. I-on-NI means that tests are generated using the inlined implementation and executed on the non-inlined implementation. NI-on-I means that tests were generated using the non-inlined implementation and executed on the inlined implementation. FF = fault finding, OO = output-only oracle, MX = maximum oracle, LCO = largest common oracle, Incr. = increase.

	I-on-NI			NI-on-I		
	I-on-NI μ FF (OO)	**% FF Incr. NI-on-NI**	**% FF Incr. I-on-I**	**NI-on-I μ FF (OO)**	**% FF Incr. I-on-I**	**% FF Incr. NI-on-NI**
DWM_1	75.46%	5,068.49%	11.33%	0.79%	-98.83%	-45.89%
DWM_2	90.50%	6.40%	1.90%	72.93%	-17.88%	-14.24%
Vertmax_Batch	90.23%	76.16%	5.41%	29.37%	-65.69%	-42.66%
Latctl_Batch	82.57%	35.48%	10.18%	48.58%	-25.17%	-20.39%
	I-on-NI μ FF (MX)	**% FF Incr. NI-on-NI**	**% FF Incr. I-on-I**	**NI-on-I μ FF (MX)**	**% FF Incr. I-on-I**	**% FF Incr. NI-on-NI**
DWM_1	97.29%	34.60%	43.05%	2.74%	-95.97%	-96.21%
DWM_2	92.80%	-1.80%	4.30%	72.93%	-18.03%	-22.86%
Vertmax_Batch	96.90%	4.96%	13.10%	100.00%	16.71%	8.32%
Latctl_Batch	98.47%	6.65%	26.13%	55.37%	-29.07%	-40.03%
	I-on-NI μ FF (LCO)	**% FF Incr. NI-on-NI**	**% FF Incr. I-on-I**			
DWM_1	75.75%	1,217.39%	11.38%			
DWM_2	90.50%	6.15%	1.72%			
Vertmax_Batch	90.40%	69.04%	5.51%			
Latctl_Batch	85.64%	27.82%	9.70%			

more complex inlined implementation, and executed tests generated over the inlined implementation against the non-inlined implementation. In Table XI, we list the median fault finding effectiveness of tests generated over the inlined implementation when executed against a non-inlined implementation. We then note the increase in fault finding over executing tests generated from the non-inlined implementation against the same implementation and executing those test generated over the inlined implementation against the same implementation. We then compare tests generated over the non-inlined implementation and executed on the inlined structure against executing those same tests on the non-inlined structure and executing the tests based on inlined structure against the inlined structure.

From these results, we can see two key trends. First, we can see that not only are tests generated over the non-inlined implementation inadequate at covering the structure of the structurally complex implementation, but they also generally achieve worse fault finding over the complex inlined implementation. Tests generated over the non-inlined implementation attain between 17.88 to 98.83% fewer faults over the inlined implementation than than tests generated using the complex implementation to begin with (see Table XI). The exception to this is for the *Vertmax_Batch* system, paired with the maximum oracle, where fault finding improves by 16.71%.

The opposite is true when tests generated over the inlined system are executed against the simpler non-inlined implementation. With the exception of *DWM_2* with the impractically-large maximum

Table XII. Summary of statistical results

Hypothesis	Results
1	$H0_1$ rejected for all systems.
2	$H0_2$ rejected for all systems.
3	$H0_3$ rejected for all systems.
4	$H0_4$ rejected for all systems.
5	$H0_5$ rejected for *Latctl_Batch*. We fail to reject $H0_5$ for *DWM_1*, *DWM_2*, and *Vertmax_Batch*.
6	$H0_6$ rejected for *DWM_1*, *DWM_2*, *Latctl_Batch*, and *Vertmax_Batch* (OO/LCO oracle). We fail to reject $H0_6$ for *Vertmax_Batch* with the maximum oracle.
7	$H0_7$ rejected for all systems.
8	$H0_8$ rejected for *DWM_1*, *DWM_2*, *Latctl_Batch*, and *Vertmax_Batch* (OO oracle). We fail to reject $H0_8$ for *Vertmax_Batch* with the maximum oracle.
9	$H0_9$ rejected for *DWM_1*, *Vertmax_Batch*, *Latctl_Batch*, and *DWM_2* (OO/LCO oracle). We fail to reject $H0_6$ for *DWM_2* with the maximum oracle.

oracle, tests generated over the complex inlined implementation achieved fault finding gains ranging from a modest 4.96% to a massive 5,068.49% over executing the tests generated using the simpler non-inlined implementation. This bolsters the results of Q3—we cannot expect tests generated from the non-inlined implementation to be effective at identifying faults in other implementation structures. However, we *can* expect tests generated over more complex implementations to be effective against other implementations.

Second, these results add an interesting extension to Q2. Not only are tests generated from the inlined implementation effective when executed against the inlined implementation—but they may be *more effective* when executed on the simpler version of the system structure. Tests generated from the inlined structure are 1.72-43.05% more effective when executed over the non-inlined implementation than when executed over the inlined implementation. This result reinforces the idea that we should use the more structurally complex version of the implementation when producing test cases, regardless of the final form that the system takes. It further indicates that we may be able to find more faults by simplifying the implementation when we execute those test cases.

The results to these four analyses raise a number of questions and concerns. In particular, we can see that the implementation structure potentially has a strong impact on both the cost and effectiveness of an MC/DC-driven testing process. This indicates that our choice of implementation structure, like our choice of coverage criterion, is an important practical aspect of the software testing process. This is further reflected in our results for *Q3*, which highlight the potential impact on a certification process using MC/DC simply achieving MC/DC over *some* implementation does not imply 100% coverage, or even high coverage, over all possible implementations. Similarly, our results for *Q4* reinforce that the choice of implementation is important when producing test cases. Tests generated using a simple implementation cannot be expected to be effective when executed on a complex implementation. The choice of implementation is important at *both* generation and execution time. We discuss these issues in detail in Section 5.

4.1. Demonstration of Statistical Significance

To ensure that the results for *Q1-Q4* are not due to chance, we propose and evaluate the following hypotheses:

— **Hypothesis** (H_1): A test suite satisfying MC/DC over the non-inlined implementation will require fewer tests relative to a test suite satisfying MC/DC over the inlined implementation.
— **Hypothesis** (H_2): When making use of the output-only oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve a lower level of fault finding relative to a test suite satisfying MC/DC over the inlined implementation.
— **Hypothesis** (H_3): When making use of the largest common oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve a lower level of fault finding relative to a test suite satisfying MC/DC over the inlined implementation.

— **Hypothesis (H_4)**: A test suite satisfying MC/DC over a non-inlined implementation will achieve less than 100% MC/DC over an inlined implementation.
— **Hypothesis (H_5)**: A test suite satisfying MC/DC over an inlined implementation will achieve less than 100% MC/DC over a non-inlined implementation.
— **Hypothesis (H_6)**: When using any oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve a lower level of fault finding when executed against the inlined implementation than on the non-inlined implementation.
— **Hypothesis (H_7)**: When using any oracle, a test suite satisfying MC/DC over the inlined implementation will achieve a lower level of fault finding when executed against the inlined implementation than on the non-inlined implementation.
— **Hypothesis (H_8)**: When using any oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve a lower level of fault finding when executed against the inlined implementation than a test suite satisfying MC/DC on the inlined implementation.
— **Hypothesis (H_9)**: When using any oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve a lower level of fault finding when executed against the non-inlined implementation than a test suite satisfying MC/DC on the inlined implementation.

To evaluate our hypotheses, we first formed null hypotheses as follows:

— $H0_1$: A test suite satisfying MC/DC over the non-inlined implementation will contain the same number of tests as a test suite satisfying MC/DC over the inlined implementation.
— $H0_2$: When making use of the output-only oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve the same level of fault finding relative to a test suite satisfying MC/DC over the inlined implementation.
— $H0_3$: When making use of the largest common oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve the same level of fault finding relative to a test suite satisfying MC/DC over the inlined implementation.
— $H0_4$: A test suite satisfying MC/DC over a non-inlined implementation will achieve 100% coverage over an inlined implementation.
— $H0_5$: A test suite satisfying MC/DC over an inlined implementation will achieve 100% coverage over a non-inlined implementation.
— $H0_6$: When using any oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve the same level of fault finding on both implementations.
— $H0_7$: When using any oracle, a test suite satisfying MC/DC over the inlined implementation will achieve the same level of fault finding on both implementations.
— $H0_8$: When using any oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve the same level of fault finding on the inlined implementation as a test suite satisfying MC/DC over the inlined implementation.
— $H0_9$: When using any oracle, a test suite satisfying MC/DC over the non-inlined implementation will achieve the same level of fault finding on the non-inlined implementation as a test suite satisfying MC/DC over the inlined implementation.

To accept H_1-H_9, we must reject $H0_1$-$H0_9$. Our observations are drawn from an unknown distribution; therefore, we cannot fit our data to a theoretical probability distribution. To evaluate our hypotheses without any assumptions on the distribution of our data, we use a one-sided (strictly lower) Mann-Whitney-Wilcoxon rank-sum test [Wilcoxon 1945], a non-parametric hypothesis test for determining if one set of observations is drawn from a different distribution than another set of observations. As we cannot generalize across non-randomly selected case examples, we apply the statistical test for each pairing of case example and oracle type with $\alpha = 0.05$[1].

[1]Note that we do not generalize across case examples or oracles as the needed statistical assumption, random selection from the population of case examples or oracles, is not met. The statistical tests are used to only demonstrate that observed differences are unlikely to have occurred by chance.

For H_1 through H_3, there exist two sets of data: one containing measurements for the 100 reduced test suites satisfying 100% achievable coverage for non-inlined implementation, and one containing measurements for the 100 reduced test suites satisfying 100% achievable coverage for inlined implementation. For H_1, these measurements are test suite sizes. For H_2 and H_3, these measurements are fault finding effectiveness measurements. The application of the permutation test is therefore straightforward. For H_6-H_9, a similar process is applied to that used for H_1-H_3. The sole difference is that, for H_1-H_3, we compared tests generated and executed on the same implementation to tests generated and executed on the other implementation. For H_6-H_9, the generation and execution source may differ, as indicated by the individual hypotheses.

For H_4 and H_5, we have only one set of data: the percentage of MC/DC achieved by each test suite generated from one implementation and executed over the other implementation. When performing the permutation test, we therefore use a set of equal size (100 records) consisting only of 100% coverage—that is, we compare against test suites achieving the maximum achievable coverage.

We perform this statistical test using each case example. For H_1-H_4, our resulting p-values are each very small—less than 0.0001. Given a traditional $\alpha = 0.05$, we reject our null hypotheses for all instances. We see that the results support our hypotheses, and in these scenarios we accept H_1-H_4 for each case example.

For one of the case examples—*Latctl_Batch*—we reject the null hypothesis $H0_5$. For the other three systems—*DWM_1*, *DWM_2*, and *Vertmax_Batch*—we fail to reject $H0_5$. In the latter cases, tests generated to satisfy MC/DC over the inlined implementation almost always achieve 100% of the achievable MC/DC obligations over the non-inlined implementation. In the case of *Latctl_Batch*, the tests generated over the inlined implementation achieve very high MC/DC, but fail to achieve 100% of the achievable coverage on average.

A natural question to ask is—why would tests created to achieve MC/DC over the inlined system fail to achieve 100% coverage on the non-inlined system? Intuitively, the reverse is easy to imagine. MC/DC on the inlined system will require specific combinations of input that will not be required to achieve coverage of the non-inlined system. However, in general, the reverse can only happen when every instance of the condition in the non-inlined decision, once inlined, does not independently affect the outcome of any decision into which it is embedded. In this case, those conditions will show up as ”unachievable” in the inlined model—no test can be produced that satisfies the test obligation. In satisfying the obligation for the inlined system, we would be required to produce a test that demonstrates this condition in its context. This test, by the nature of inlined code, would typically be stronger than necessary to demonstrate it outside of its context (i.e., on the non-inlined version of the system). These missing tests actually demonstrate that those conditions are, in fact, dead code, once considered in the context in which they are used.

As indicated in Table X, the only system where tests created for the inlined implementation fail to achieve 100% coverage on the non-inlined implementation is *Latctl_Batch*. There are two obligations that the inlined version of the suite fails to cover. One of those is explained by the above. The equivalent obligation for the inlined version is impossible to satisfy (see Table V), therefore, no test exists in the suite produced for the inlined version that covers the equivalent condition in the non-inlined implementation. That expression is, in fact, dead code.

The second can be explained as a result of the inlining process. *Latctl_Batch* runs in a dual-redundant configuration. There is an input that turns this model into a passive controller that just feeds through the outputs from the other side. If you allow this input to be true, you cannot prove anything about the model—the inputs from the other side are arbitrary. Therefore, this input was set to a constant value of false. The non-inlined system contains an expression used to set the system to the passive mode. However, as this expression, an `and` statement, would always evaluate to false given the environmental settings, the automated inlining process removed the expression completely. In practice, where code transformations would likely be done by hand, that expression would have remained in the system and an equivalent test for the inlined system would have been produced.

Regarding H_6 and H_8—in both cases, we can reject the null hypothesis for three of the four systems for all oracles, but we fail to reject the null hypothesis for *Vertmax_Batch* when paired with the maximum oracle. For the former systems, tests generated over the non-inlined system and executed on the inlined system fail to outperform tests generated and executed on the non-inlined system and tests generated and executed on the inlined system. For the latter case, the tests generated on the non-inlined version of *Vertmax_Batch* find all non-equivalent mutants in the inlined system when the expensive maximum oracle is used. In that case, the oracle provides the necessary observability to identify those faults. When the more common output-only oracle is used, we can reject the null hypotheses for *Vertmax_Batch*.

We can reject $H0_9$ for three of the four systems for all oracles, but we fail to reject the null hypothesis for *DWM_2* when paired with the maximum oracle. For the former systems, tests generated over the non-inlined system and executed on the non-inlined system fail to outperform tests generated on the inlined system and executed on the non-inlined system. In the latter case, tests generated on the inlined system and executed on the non-inlined system perform slightly worse (-1.80%) on average than those generated and executed on the non-inlined system, when the maximum oracle is used. Again, when the more common output-only or the largest common oracle is used, we can reject the null hypotheses for *DWM_2*.

Based on these results, we conclude that a clear, statistically significant pattern exists in the four case examples used in our study: by varying implementation structure, we can impact both the number of test inputs required to satisfy MC/DC as well as the fault finding effectiveness of test suites achieving 100% achievable MC/DC.

5. DISCUSSION

For each research question in our study, we have found that the structure of the implementation has a strong, consistent impact on the testing process when using the MC/DC criterion.

Our results for *Q1* indicate that, by varying the structure of the implementation, we can influence—positively or negatively—the cost of generating test inputs to satisfy MC/DC. This observation is potentially useful in the context of MC/DC's role in the certification process for critical avionics systems, as satisfying MC/DC during testing can be extremely expensive. By applying syntactic transformations to avionics systems, developers can generate systems for which MC/DC is significantly easier and cheaper to achieve. For our examples, inlined implementations required test suites as least twice as large as the size of the test suites generated over the semantically equivalent non-inlined systems—meaning that there is a potentially dramatic savings in testing costs when using a non-inlined system.

Unfortunately, our results for *Q2* indicate that test suites generated to satisfy structural coverage criteria over the inlined implementation generally outperformed those test suites generated to satisfy structural coverage criteria over the non-inlined implementation, with relative improvements of up to 4,542.47% when using the fixed-sized test oracles. Thus, we see a potential tradeoff—we *can* restructure our implementation to reduce the cost of satisfying coverage criteria, but we incur the risk of reducing the effectiveness of the testing process.

Both of these implications point at a deeper issue: the sensitivity of the MC/DC criterion to the structure of the implementation. Our results for *Q3* indicate that while implementation structure impacts the cost and effectiveness of the MC/DC criterion, it also impacts the measurement of the criterion. Tests generated using a simpler implementation cannot be expected to attain high coverage of the structurally complex implementation, while tests generated over the structurally complex implementation *can* be expected to cover the test obligations of the structurally simple implementation. Similarly, the results of *Q4* indicate a similar idea. Tests generated using the complex implementation are not only effective at finding faults in the same implementation, but are effective when that implementation is varied. Tests generated using the simple implementation are less effective regardless of the implementation used during test execution.

In the avionics domain, the role of the MC/DC criterion is to determine if our test inputs are adequate, as implied in the synonymous term *test adequacy metric*. It is therefore worth questioning if

this metric is *itself* adequate. Given that the effectiveness of MC/DC can vary considerably depending on the structure, ranging from very high to—in the case of the *DWM_1* system—very low, when asking the question: *is this set of test inputs adequate?* it is difficult to fully trust the results of the criterion. Therefore, we believe that either care must be taken when using MC/DC as an adequacy metric, or that another method of measuring test adequacy be constructed. By specifying a specific implementation structure for measuring and producing MC/DC-satisfying test suites, we can ensure that the potential benefits of using MC/DC as an adequacy criterion are more likely to be gained through its use. In the remainder of this section, we discuss observations and concerns raised from these results.

5.1. Cost and Effectiveness Increase with Complexity

Citing Tables VII and VIII, we again note that both size and fault finding effectiveness increase when using a more complex implementation structure, sometimes dramatically. An extreme example of this is the *DWM_1* system: on average, 1,566.66% more tests are required on average when satisfying MC/DC on the inlined system versus the non-inlined system, resulting in 4,542.47% more faults detected on average.

This increase in tests can be attributed to the increased complexity of the coverage obligations generated when using the inlined implementation as opposed to the non-inlined implementation. Recall that when transforming the non-inlined implementation into the inlined implementation, the complexity of conditional expressions increases, as intermediate variables formerly used as atomic Boolean conditions are instead replaced with more complex subexpressions. We illustrate an example of this in Figure 5.

Table XIII. Impact of implementation structure on coverage obligations

Non-inlined MC/DC Obligations		Inlined MC/DC Obligations	
(1)	$(x \wedge y)$	(1)	$(x \wedge y) \wedge \neg(z \wedge w)$
(2)	$(\neg x \wedge y)$	(2)	$\neg(x \wedge y) \wedge (z \wedge w)$
(3)	$(x \wedge \neg y)$	(3)	$(\neg x \wedge y) \wedge \neg(z \wedge w)$
(4)	$(z \wedge w)$	(4)	$(x \wedge \neg y) \wedge \neg(z \wedge w)$
(5)	$(\neg z \wedge w)$	(5)	$\neg(x \wedge y) \wedge (\neg z \wedge w)$
(6)	$(z \wedge \neg w)$	(6)	$\neg(x \wedge y) \wedge (z \wedge \neg w)$
(7)	$(x \wedge y) \wedge \neg(z \wedge w)$		
(8)	$\neg(x \wedge y) \wedge (z \wedge w)$		
(9)	$\neg(x \wedge y) \wedge \neg(z \wedge w)$		

In the case of MC/DC the complexity of the obligations we must meet to satisfy MC/DC is related to the complexity of the expressions. To understand why, consider Figure 5. Here we see the number of atomic conditions on line 4 grows from two to four when line 1 and 2 are inlined. Consider the obligations we must satisfy to achieve MC/DC over these expressions: we must demonstrate each condition can *positively* and *negatively* affect the outcome of the expression. In other words, we must show each condition can cause the expression to be true and false. Consequently, as the number of

```
[1] internal42 = x AND y
[2] internal13 = z AND w
[3]
[4] output1 = (internal42 OR internal13)
```

Non-inlined Implementation

```
[1] output1 =  (x AND y) OR (z AND w)
```

Inlined Implementation

Fig. 5. Increasing complexity of boolean decisions during inlining transformation

conditions grows, the number of MC/DC obligations for the expression also grows, and with it the number of test inputs we must generate also generally grows.

The MC/DC obligations to satisfy can be schematically derived, resulting in the obligations in Table XIII [Whalen et al. 2006]. We can see that, while the non-inlined implementation actually has more obligations (nine rather than six)—due to the larger number of expressions (three small expressions rather than one large one)—the inlined implementation's obligations are generally more complex and require more specific inputs. As a result, opportunities to generate test inputs satisfying multiple obligations—allowing a reduction in the total number of inputs required—are more limited for the obligations generated from the inlined implementation.

For example, consider the obligations 7-9 for the non-inlined system. These obligations are logically disjoint, and thus require three test cases. However, by carefully selecting the test cases, we can also satisfy all the obligations for lines 1-6, as shown in Table XIV.

Table XIV. Impact of implementation structure on coverage obligations

Inputs				Non-inlined Obligations									Inlined Obligations					
x	**y**	**z**	**w**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**1**	**2**	**3**	**4**	**5**	**6**
T	F	T	T	F	F	T	T	F	F	F	T	F	F	T	F	F	F	F
T	T	T	F	T	F	F	F	F	T	T	F	F	T	F	F	F	F	F
F	T	F	T	F	T	F	F	T	F	F	F	T	F	F	T	F	T	F

We have less freedom in selecting test inputs for the inlined implementation's more complex obligations, and thus, this scenario is not possible. The test inputs above only satisfy inlined obligations 1, 2, 3, and 5, leaving the remaining obligations to be covered by other test inputs. Therefore, while fewer obligations are generated, more test inputs are required.

These increases in test suite size result in generally improved fault finding, for two apparent reasons. First, and perhaps most obvious, is the increase in test suite size. Recent work on modeling the effectiveness of testing has indicated that effectiveness is highly dependent on test suite size [Namin and Andrews 2009; Gligoric et al. 2013; Inozemtseva and Holmes 2014]. By simply increasing the number of test inputs used, we can improve the effectiveness of our testing process.

Second, by changing the implementation structure, we have strengthened the constraints on our coverage obligations, resulting in not only more test inputs, but test inputs that are differentiated—exploring combinations of conditions that would not otherwise be explored. For example, consider a situation where the implementations in Figure 5 are incorrect—where the final implementation line should be `((x AND NOT y) OR (z AND NOT w))`. The test suite depicted in Table XIV, while sufficient to satisfy MC/DC over the non-inlined implementation, would not pick up on this fault. However, if we created a test suite to satisfy MC/DC over the inlined implementation, any test sufficient to satisfy inlined obligation 4 would detect this particular fault.

5.2. Coverage Measurements Vary with Complexity

In the previous section, we discussed the factors contributing to the results observed for *Q1* and *Q2*, demonstrating how changing implementation structure increases the number of test inputs required and the effectiveness of said test inputs. The same factors contribute to the results observed for *Q3*, in which we found that test inputs generated to satisfy 100% achievable MC/DC over a non-inlined implementation result in less than 100% achievable coverage over an inlined implementation.

The low average levels of MC/DC–as low as 13.08%—achieved over the inlined implementation by the non-inlined test suites were somewhat surprising. One reason for the reduction in coverage is likely to be the reduction in the number of test inputs. As seen in Table VII, the average size of a test suite generated over the non-inlined system is—at largest—less than half of the average size of the test suite for the inlined version of the same system.

However, we believe that the dramatic drop in achieved coverage is not only a result of test suite size, but also of the test generation method. By using counterexample-based test generation, we ensure that each coverage obligation is satisfied. However, each generated test input is specifically

targeted at satisfying a single coverage obligation. Thus, each input tends to perform the minimum number of actions needed to cover that obligation. This is because, in the context of model checking, generated counterexamples are ideally short and simple, so as to be better understood by the user. In particular, default input values (e.g., *0 or false*) are used whenever possible, and generally only the variables that must be manipulated to satisfy the coverage obligation are changed. Thus, when generating test inputs to satisfy MC/DC over the non-inlined implementation, the resulting test inputs may not be sufficiently complex to satisfy coverage obligations over the inlined implementation; the quantity of inputs is not only low, but each individual test input is also less complex.

To demonstrate this, for each case example, we used the inlined test suites to generate reduced test suites of size equal to the reduced test suites from the non-inlined implementation. In other words, for each reduced test suite generated using the non-inlined implementation, we generated a reduced test suite of equal size using the inlined implementation[2]. We then computed the average coverage achieved over the inlined implementation.

We present the results in Table XV, along with the sizes and original average coverages for the non-inlined test suites (also found previously in Table VII). As shown, we see that reduced test suites of size equal to those drawn from the non-inlined implementation can provide much higher coverage levels, provided sufficiently effective test inputs are used to construct the test suites. In our case examples, increases in coverage of 41.24% to 643.27% are observed.

Table XV. Average coverage of reduced suites. NI = Non-inlined, I = Inlined

	DWM_1	**DWM_2**	**Latctl_Batch**	**Vertmax_Batch**
Median Size	27.00	69.00	28.00	68.00
NI Suite Coverage Over I	13.08%	53.25%	67.67%	31.95%
Paired I Suite Coverage Over I	97.22%	90.60%	95.58%	84.04%
% Increase	643.27%	70.14%	41.24%	163.04%

One key observation that can be seen in Table XV is that a small number of tests, created for the inlined implementation, are often sufficient to cover a large number of test obligations. When formulating test obligations, some will be simple and some will require complex combinations of input to satisfy. When working with an inlined implementation—by the nature of inlining—more obligations will be of the latter format. Thus, when designing tests to satisfy those obligations, the tests will cover a large variety of the possible input combinations. Some of those inputs will be so specific that they only satisfy a small number of obligations, but a more common result is that a number of tests will be required to execute the system for enough execution cycles or method calls to trigger the exact required combination of conditions. These tests are highly likely to cover several additional obligations in the process of covering the one specific obligation they were designed to cover.

This is another piece of evidence in favor of designing tests over the inlined implementation—tests are likely to exercise the system more thoroughly. Tests may cover a larger portion of the state space or execute the system for a longer period of time, exploring a wider variety of combinations of input as they execute, and satisfying a large number of obligations in the process. As a result, even a small number of tests, created for the inlined version of the system, may achieve higher coverage and fault detection capability than tests created for the non-inlined implementation.

5.3. Effectiveness Varies With Complexity At Both Generation and Execution Time

Given the ease of satisfying MC/DC over a non-inlined implementation, one could imagine a scenario where test generation is performed on a structurally simple implementation such as a Simulink model or an uninlined version of the source code, then such tests are later executed against a different implementation (such as the real code, if the Simulink model was used during generation).

[2]The randomized greedy algorithm was again used, but the stopping point was no longer 100% coverage, but instead a test suite size.

Table XVI. Median (μ) Fault finding measurements when tests are generated using the inlined implementation, the suites are reduced to match the size of suites generated using the non-inlined implementation, and then the suites are executed on the non-inlined implementation. FF = fault finding, OO = output-only oracle, MX = maximum oracle, LCO = largest common oracle.

	Reduced I-on-NI μ FF(OO)	**% FF Increase Over NI-on-NI**
DWM_1	25.02%	1,613.70%
DWM_2	88.32%	3.86%
Vertmax_Batch	74.38%	45.22%
Latctl_Batch	72.15%	18.24%
	Reduced I-on-NI μ FF(MX)	**% FF Increase Over NI-on-NI**
DWM_1	83.68%	15.77%
DWM_2	91.05%	-3.69%
Vertmax_Batch	86.17%	-6.66%
Latctl_Batch	95.29%	3.21%
	Reduced I-on-NI μ FF(LCO)	**% FF Increase Over NI-on-NI**
DWM_1	27.10%	414.23%
DWM_2	88.43%	3.48%
Vertmax_Batch	75.09%	40.41%
Latctl_Batch	76.75%	14.55%

The results of *Q3* indicate that this would not be advisable, as tests generated over a simple implementation cannot be expected to attain coverage over a complex implementation. The results of *Q4* further make this point—tests generated over the non-inlined implementation were less effective at finding faults in the inlined implementation than tests generated and executed over the inlined implementation. In the reverse case, tests generated over the inlined implementation are *more effective* at finding faults in the non-inlined implementation than tests generated and executed over the non-inlined implementation.

As the results of *Q2* indicated, the size of the test suite is a factor in the effectiveness of that suite. Thus, these results are not entirely surprising—by generating on the non-inlined system, we end up with a smaller number of tests to execute on the inlined system than when we generated using the inlined system. By generating on the inlined system, we end up with substantially more tests that can be executed on the non-inlined system than when we generated using the non-inlined system. To better understand the effect of test suite size on these results, we took test suites generated on the inlined system and reduced them to the size of suites generated using the non-inlined system (the same suites used in Section 5.2). We then computed the median fault finding of these reduced test suites when executed against the non-inlined implementation.

The results of this experiment can be seen in Table XVI. We can see that, while there was a drop in fault finding effectiveness when the suite size was reduced, the tests generated using the inlined implementation were still more effective at fault finding in nearly all situations. When using the output-only or largest common oracle, the suites generated using the inlined implementation found 3.48-1,613.70% more faults in the non-inlined implementation than the tests generated using that implementation. This suggests that the larger number of tests required to satisfy MC/DC over the inlined implementation are not the sole source of the improved effectiveness of those suites. Indeed, just as with the coverage analysis in Section 5.2, it seems that the complex combinations of input required to satisfy MC/DC over complex implementations are also likely to reveal faults. It is not enough to simply satisfy the letter of the law—to attain MC/DC in any manner possible. Whether a test obligation is covered is less important than *how it was covered*.

Regardless of the structure used during test execution, these results provide more evidence in favor of using the structurally complex implementation to generate test cases. Although the cost of producing the tests will be higher than when a simple structure is used, the resulting tests will be effective at attaining coverage *and* finding faults. By more thoroughly exercising the system under test, even a small number of tests created using the inlined implementation may be effective at finding faults in the source code.

The results of *Q4* in Table XI offered one additional observation of interest. Test suites generated using the inlined implementation were not only more effective at finding faults in the non-inlined implementation than tests generated on that implementation, but they attained *higher* fault finding on the non-inlined implementation than on the inlined implementation. This hints that there may, in fact, be a use for a simple implementation—as a version of the system to *execute* tests on. If a tool can be used to automatically inline or uninline code, then the code could be maximally inlined for test generation purposes—offering the benefits of MC/DC satisfaction on a complex implementation. The code could then be maximally uninlined during test execution, potentially making it easier to observe the effect of program faults triggered by those test cases.

More research is needed to confirm this hypothesis, but these results make it clear that implementation structure is important not only during test generation, but when those tests are executed as well. If tests are generated using a simple implementation, they cannot be expected to be effective when executed on a complex implementation. If tests are generated using a complex implementation, they are more likely to be effective no matter what implementation is used at execution time. Further, additional improvement in effectiveness may be possible by generating over a complex implementation, then executing over a simple implementation.

5.4. Less Than 100% Fault Finding

Fault finding was often significantly lower than 100% over the case examples—regardless of the test oracle employed, and particularly for non-inlined test suites. Although it is well known that testing is an incomplete form of verification, we were initially surprised by the poor fault finding of test suites that meet the rigorous MC/DC criterion. As we studied the models closely, we identified several possible reasons for the poor absolute fault finding, including:

Faults in uncovered portions of the model: As seen from our results in Table V, in several instances, the achievable MC/DC over the implementation is less than 100%. This implies that there are portions of the implementation for which there exists no test case that can provide MC/DC. We term these as the "uncovered" portion in the implementation. Note here that the test suites we use in our experiment provide maximum achievable MC/DC over their respective implementation. When seeding faults in an implementation, we may seed faults in the uncovered portion. Since no MC/DC test case could be constructed to cover this portion, the test suite will likely (though not necessarily) miss such faults. In our experiment, we did not attempt to identify the faults that are seeded in the uncovered portion of the implementation. Such a task would be time consuming and difficult since it would require manually examining the implementation, test suites, and mutations.

Delay expressions: All of the studied case examples execute on some form of execution cycle; they sample the environment, execute to completion, and then wait until it is time for a new execution cycle. A *delay expression* is one that uses the value of an expression from a previous execution cycle. This delay expression is explicit in modeling languages such as SCADE and Simulink; in programming languages like C the same effect is achieved using state variables that are assigned during one cycle and used by statements earlier in the loop during the next cycle. Delay faults are not a classic mutation to perform; however, this kind of fault occurs with regularity in control software, which runs as a periodic polling loop. Errors occur when the "previous" value is used when the "current" value is expected or vice-versa; these kind of errors affect edge-detection, input smoothing, integration, and other state-requiring operations.

The key here is that delay expressions (or state variables) are assigned one or more cycles before use. Thus, it is possible—especially when using a model checker that intentionally generates short tests—that a test can terminate too early (i.e., after covering the structure that is assigned to a state variable, but before the variable can propagate to an output)[3].

To illustrate this problem in a C-like implementation language, consider the program fragment in Figure 6. In this code, execution steps are represented as loop iterations. Variables used in future

[3]Note that this problem is different from semantically equivalent mutants since it is *possible* to reveal the mutation, but only with a test case longer than what is necessary to achieve MC/DC.

```
void Delay_Expr()
{
    bool pre_var;
    bool var_a = false;

    while(1)
    {
        in1 = sample(in1); // Update value
        in2 = sample(in2); // Update value
        pre_var = var_a; // Store previous value
        var_a = in1 or in2; // Calculate new value
        print pre_var; // Display new output
    }
}
```

MC/DC Test Set for (in1, in2):
`TestSet1 = {(TF),(FT),(FF)}`

Fig. 6. Sample program fragment that uses variable values from previous step

execution steps are stored away as intermediate variables for use in later loop iterations. Because of this delayed usage mechanism, these intermediate variables cannot be inlined. `TestSet1` with one step test cases will provide MC/DC of this program fragment, since we only need to exercise the *or* Boolean expression up to MC/DC. If we were to erroneously replace the *or* operator with *and*, or any other Boolean operator, `TestSet1` providing MC/DC would not be able to reveal the fault, since the test cases are too short to affect the output—the test cases only consist of one input step, they would need to be at least 2 steps long for failures to propagate to the output. Many systems in the domains of vehicle or plant control (such as the avionics domain) are designed to use variable values from previous steps. Thus, test cases generated to provide MC/DC over such systems will often be shorter than needed to allow erroneous state information to propagate to the outputs. Based on this observation and our results in Table VIII, we believe that delay expressions represent a serious concern related to the effectiveness of MC/DC test suites.

Intermediate variable masking: As mentioned previously, generated test suites do not ensure that intermediate variables affect the output. While we expect such masking to occur with non-inlined implementations, masking is also possible with inlined versions of implementations as they are not completely inlined (this is discussed in Section 3.4 and is also seen in the delay expression discussion). We may, therefore, still have some intermediate variables in the inlined implementations that present opportunities for masking. Figure 7 shows a sample C like program with an intermediate variable *no_alarm* that cannot be inlined and can, therefore, potentially be masked out in a test case.

`TestSet1` in Figure 7 provides MC/DC over the *compute* function; the test cases with `in3 = false` (bold faced) contribute towards MC/DC of the `in1 or in2` condition in the if-then-else statement. However, these test cases mask out the effect of the intermediate variable `no_alarm` in the *and* expression since `in3 = false`. Suppose the code fragment in Figure 7 was faulty, the correct expression should have been `in1` **`and`** `in2` (which was erroneously coded as `in1` **`or`** `in2`). `TestSet1` providing MC/DC would be incapable of revealing this fault, since there would be no change in the observable output. Thus, seeded faults like the one mentioned here cannot be revealed by a test suite achieving MC/DC because of intermediate variable masking. This observation serves as a reminder that masking is a crucial consideration for generating test suites that are effective in fault finding.

One potential solution for this is to use stronger test oracles which consider the value of internal state. Such test oracles, when used in conjunction with test inputs satisfying MC/DC, have been empirically and theoretically shown to provide increased fault finding capability in this domain and others [Gay et al. 2015a; Staats et al. 2012a; Staats et al. 2011b]. Additionally, Whalen et al. have proposed a stronger variant of MC/DC (dubbed Observable MC/DC) that addresses mask-

```
bool Compute(bool in1,in2,in3)
{
    bool no_alarm;

    if (in1 or in2)
        no_alarm = true
    else
        no_alarm = false;
    return (in3 and no_alarm);
}
```

MC/DC Test Set for (in1, in2, in3):

TestSet1 = {**(TFF)**, **(FTF)**, (FFT), (TTT)}

Fig. 7. Sample C like inlined program fragment

ing by requiring a propagation path from a decision statement to a variable observed by the test oracle [Whalen et al. 2013]. Observable MC/DC demonstrates higher fault finding effectiveness than masking MC/DC—particularly for non-inlined systems. Our experimental results highlight the importance of such work.

6. THREATS TO VALIDITY

External Validity: We have conducted our study on four synchronous reactive critical systems. We believe these systems are representative of the class of systems in which we are interested, and our results are thus generalizable to other systems in the domain.

We have used implementations expressed in Lustre rather than a more common language such as C or C++. Nevertheless, systems written in the Lustre language are similar in style to traditional imperative code produced by code generators used in embedded systems development. We therefore believe that testing Lustre code is sufficiently similar to testing reactive systems written in traditional imperative languages.

We have generated our test inputs using a model checker. Other possible options would be to use tests manually created by testers, tests generated through other automated processes, or randomly generated tests. It is possible these other options would yield results that are significantly different. However, in our experience, tests generated using a model checker are relatively *less effective* than other options; we, therefore, are effectively studying worst-case (or at least not exceptionally positive) behavior of the MC/DC criterion. One reason for this, as discussed in Section 5.2, is that the tests produced lack *diversity*—the model checker prefers to leave inputs at default values when not required to satisfy a particular property. Another reason, as discussed in Section 5.4, is that the tests are often not long enough to ensure fault propagation to output (however, this is more of a problem with the coverage criterion than the generation tool—the tool does exactly what it is asked to do and no more). Given that we are evaluating the impact of implementation structure on the effectiveness of this criterion, both of these traits allow us to clearly explain the risks involved in changing the implementation structure.

We have examined 100 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 been seen to produce consistent results [Rajan et al. 2008; Staats et al. 2010].

We have varied implementation structure using an in-house tool for transforming implementations. It is possible that one or both structures used do not correspond to an implementation structure we are likely to see in actual implementations. Nevertheless, the non-inlined implementation structure is very similar to the original structure of the system before it is translated from Simulink, and we thus believe it is representative of systems likely to be considered. The inlined implementation structure is similar to the code generated from tools such as Real-Time Workbench.

Internal Validity: Our study makes extensive use of automation. It is possible that some effects observed are due to errors in the automation of the experiment. However, much of this automation is part of an industrial framework (courtesy of Rockwell-Collins), particularly the more complex automation for implementation transformation, and has been extensively verified [Miller et al. 2010]. Other automation for running tests, producing oracles, etc., developed at the University of Minnesota has also undergone verification, and is relatively simple. We therefore believe it is highly unlikely that errors that could lead to erroneous conclusions exist in the automation.

Construct Validity: In our study, we measure fault finding over seeded faults, rather than real faults encountered during development of the software. It is possible using real faults would lead to different results. However, Andrews et al. have shown the use of seeded faults leads to conclusions similar to those obtained using real faults in fault finding experiments [Andrews et al. 2006]. Additionally, recent work from Just et al. suggests a significant correlation between mutant detection and real fault detection [Just et al. 2014].

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions for these analyses are met, and thus have favored non-parametric methods. In cases where the base assumptions are clearly not met, we have avoided using statistical methods. Notably, we have avoided statistical inference across case examples, as we have not randomly sampled these examples from the larger population.

7. RELATED WORK

Work related to this study can be divided into roughly two groups: work related to structural coverage criteria, and work related to the MC/DC criterion specifically.

7.1. Structural Coverage

The current favored proxy for measuring the adequacy of testing efforts is the coverage of structural elements of the source code of the software, such as individual statements, branches of the softwares control flow, and complex boolean conditional statements [Kit and Finzi 1995; Perry 2006; Pezzé and Young 2006]. The idea is simple, but compelling—unless code is executed, faults will never be found. Using coverage as a measure for the adequacy of testing gives developers a number that can be quickly calculated and used as a target, as a measure of what has been accomplished, as the basis for planning future efforts, and—importantly—as the means by which we can enable to automated generation of test cases.

According to Chilenski and Miller [Chilenski and Miller 1994], structural coverage criteria are divided into two types: data flow and control flow. Data flow criteria measure the flow of data between variable assignments and references to the variables. Data flow metrics, such as all-definitions and all-uses [Beizer 1990], involve analysis of the paths (or subpaths) between the definition of a variable and its subsequent use. The structural coverage criteria in many standards, including DO-178C [RTCA/DO-178C], are often control flow criteria. Control flow criteria measure the flow of control between statements and sequences of statements. Pezzé and Young [Pezzé and Young 2006] discuss most of the well known control flow metrics used in structural testing, including, statement coverage, branch coverage, condition coverage, MC/DC, path coverage, and call coverage.

Automated test generation methods can use coverage criteria as optimization targets, producing test cases that cover a large portion of code with minimal human effort [McMinn 2004]. If coverage is, in fact, correlated with fault finding effectiveness, then automated test generation represents a powerful direction in lowering the cost and human effort associated with software testing.

That said, recent work has endeavored to revise and clarify the exact role of structural coverage in testing, and to quantify the actual power of coverage to predict for faults [Groce et al. 2014]. Studies on the effectiveness of test suites created with the goal of satisfying common coverage metrics have yielded inconclusive results, some noting positive correlation between coverage level and fault detection [Namin and Andrews 2009; Mockus et al. 2009], while more recent work paints a negative portrait of the effect of coverage on improving the detection of real faults [Inozemtseva and Holmes 2014]—particularly when tests have been automatically generated [Fraser et al. 2013]. Previous

work from the authors of this publication echoes the latter results—tests generated with the goal of maximizing coverage over the code, including MC/DC-satisfying tests, were often outperformed by test cases generated at random, and even for systems where the generated tests outperformed the random tests, the generated tests often found fewer than 50% of the seeded faults [Gay et al. 2015b; 2014; Staats et al. 2012b]. These results match recent experiments performed using state-of-the-art test generation techniques, where—despite noting positive correlation between fault finding and coverage—the authors found that automated testing performed poorly *overall* (collectively detecting only 55.7% of the faults) [Shamshiri et al. 2015].

In this work, we have focused on the idea that a major factor in determining the power of coverage as a proxy for adequate testing is the structure of the code and the sensitivity of the coverage criterion to that structure. Another factor that has been explored is the size of the test suite, with the majority of work confirming the intuitive idea that larger test suites are more effective at detecting faults [Gligoric et al. 2013; Inozemtseva and Holmes 2014]. Our previous work, while noting the same effect, has found that an even more important factor is the number and selection of variables examined by the test oracle [Staats et al. 2011a; Staats et al. 2012a; Gay et al. 2015a]. Careful selection of which variables to monitor and check for failing values is a major factor in improving the fault finding effectiveness of tests created to satisfy structural coverage criteria.

7.2. MC/DC

Hayhurst et al. first observed the sensitivity of the MC/DC metric, stating that " *if a complex decision statement is decomposed into a set of less complex (but logically equivalent) decision statements, providing MC/DC for the parts is not always equivalent to providing MC/DC for the whole*" [Hayhurst et al. 2001]. Our work attempts to quantify the difference, both in terms of coverage and fault finding effectiveness, between measuring MC/DC on complex decisions versus simple decisions.

Gargantini et al. have also observed the sensitivity of structural coverage metrics to modification of the code structure and have proposed a method of automatically measuring the resilience of a piece of code to modification [Gargantini et al. 2013]. The AURORA tool produces copies of a piece of software where the code has been transformed through user-defined rules. This process—similar to the practice of mutation testing [Andrews et al. 2006]—allows for the calculation of a fragility index measuring the sensitivity of the coverage level to changes in code structure.

Chilenski made the observation that " *If the number of tests M is fixed at N + 1 (N being the number of conditions), the probability of distinguishing between incorrect functions grows exponentially with N, $N > 3$*" [Chilenski 2001]. This observation is based only on the number of tests, not on which tests were run. The results in our experiments support this observation. For our industrial examples, test suites that provide MC/DC over the non-inlined implementation provided poor coverage over the inlined implementations. The results indicate that MC/DC, when measured, should be on the inlined implementation with complex decisions where N is usually much larger than on the non-inlined implementation with simple decisions (with smaller N). Using our results along with Chilenski's observation, we infer that a given test suite would be more effective in revealing incorrect functions in the inlined implementation than in the non-inlined implementation.

Despite the importance of the MC/DC criterion [Chilenski and Miller 1994; RTCA 1992], studies of its effectiveness are few. Yu and Lau study several structural coverage criteria, including MC/DC, and find MC/DC is cost effective relative to other criteria [Yu and Lau 2006]. Kandl and Kirner evaluate MC/DC using an example from the automotive domain, and note less than perfect fault finding [Kandl and Kirner 2011]. Dupuy and Leveson evaluate MC/DC as a complement to functional testing, finding that the use of MC/DC improves the quality of tests [Dupuy and Leveson 2000]. Our previous work found that automatically generating tests with the goal of achieving MC/DC led to poor results, but that the use of MC/DC as a stopping criterion for existing testing efforts yielded some benefit [Staats et al. 2012b; Gay et al. 2015b]. None of these studies, however, explore the impact of program structure on the criterion.

8. CONCLUSION

In this work, we have explored the impact of implementation structure on the efficacy of test suites satisfying the MC/DC criterion using four real world avionics systems. For each system, we automatically constructed two semantically equivalent implementations: a *non-inlined* implementation which uses only simple Boolean expressions, and an *inlined* implementation which uses more complex Boolean expressions.

Our results demonstrate that test suites achieving maximum achievable MC/DC over inlined implementations are generally larger than test suites achieving maximum achievable MC/DC over non-inlined implementations, with increases in size of 125.00%-1,566.66% observed. This increase in test suite size also generally corresponds with an increase in fault finding effectiveness, with improvements up to 4,542.47% observed at fixed oracle sizes. We found that test suites generated over non-inlined implementations achieve significantly less MC/DC when applied to inlined implementations, 13.08%-67.67% in our study. We also found that test suites generated over non-inlined implementations cannot be expected to yield effective fault finding on inlined implementations, finding 17.88-98.83% fewer faults than tests generated and executed on the inlined implementation. Tests generated using the inlined implementation generally attained 100% MC/DC on the non-inlined system, and found up to 5,068.49% more faults than tests generated and executed on the non-inlined implementation.

We believe these results are concerning, particularly in the context of certification: we have demonstrated that by measuring MC/DC over simple implementations, we can significantly reduce the cost of testing, but in doing so we also reduce the effectiveness of testing. Thus developers have an economic incentive to "cheat" the MC/DC criterion (and by building tools similar to those used in our study, the means), but this cheating leads to negative consequences.

Accordingly, we recommend organizations adopt a canonical—at least moderately structurally complex—implementation form for testing purposes to avoid these negative consequences.

REFERENCES

J.H. Andrews, L.C. Briand, Y. Labiche, and A.S. Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *Software Engineering, IEEE Transactions on* 32, 8 (aug. 2006), 608 –624. DOI: http://dx.doi.org/10.1109/TSE.2006.83

Boris Beizer. 1990. *Software Testing Techniques, 2nd Edition*. Van Nostrand Reinhold, New York.

J. Chilenski. 2001. *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*. Technical Report DOT/FAA/AR-01/18. Office of Aviation Research, Washington, D.C.

J. J. Chilenski and S. P. Miller. 1994. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal* (September 1994), 193–200.

A. Dupuy and N. Leveson. 2000. An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software. In *Proc. of the Digital Aviation Systems Conf. (DASC)*. Philadelphia, USA.

Esterel-Technologies. 2004. SCADE Suite Product Description. http://www.esterel-technologies.com/v2/ scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html. (2004).

Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2013. Does Automated White-box Test Generation Really Help Software Testers?. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 291–301. DOI: http://dx.doi.org/10.1145/2483760.2483774

Andrew Gacek. 2015. JKind - a Java implementation of the KIND model checker. https://github.com/agacek. (2015).

Angelo Gargantini, Massimo Guarnieri, and Eros Magri. 2013. AURORA: AUtomatic RObustness coveRage Analysis Tool. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 463–470.

A. Gargantini and C. Heitmeyer. 1999. Using Model Checking to Generate Tests from Requirements Specifications. *Software Engineering Notes* 24, 6 (November 1999), 146–162.

G. Gay, M. Staats, M. Whalen, and M. Heimdahl. 2015a. Automated Oracle Data Selection Support. *Software Engineering, IEEE Transactions on* PP, 99 (2015), 1–1. DOI: http://dx.doi.org/10.1109/TSE.2015.2436920

G. Gay, M. Staats, M. Whalen, and M.P.E. Heimdahl. 2015b. The Risks of Coverage-Directed Test Case Generation. *Software Engineering, IEEE Transactions on* PP, 99 (2015). DOI: http://dx.doi.org/10.1109/TSE.2015.2421011

Gregory Gay, Matt Staats, Michael W. Whalen, and Mats P. E. Heimdahl. 2014. Moving the Goalposts: Coverage Satisfaction is Not Enough. In *Proceedings of the 7th International Workshop on Search-Based Software Testing (SBST 2014)*. ACM, New York, NY, USA, 19–22. DOI: http://dx.doi.org/10.1145/2593833.2593837

Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2013. Comparing Non-adequate Test Suites Using Coverage Criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 302–313. DOI: http://dx.doi.org/10.1145/2483760.2483769

Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. 2014. Coverage and Its Discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 255–268. DOI: http://dx.doi.org/10.1145/2661136.2661157

G. Hagen. 2008. *Verifying safety properties of Lustre programs: an SMT-based approach*. Ph.D. Dissertation. University of Iowa.

N. Halbwachs. 1993. *Synchronous Programming of Reactive Systems*. Klower Academic Press.

K.J. Hayhurst, D.S. Veerhusen, and L.K. Rierson. 2001. *A Practical Tutorial on Modified Condition/Decision Coverage*. Technical Report TM-2001-210876. NASA.

M.P.E. Heimdahl, M.W. Whalen, A. Rajan, and M. Staats. 2008. On MC/DC and implementation structure: An empirical study. In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*. 5.B.3–1–5.B.3–13. DOI: http://dx.doi.org/10.1109/DASC.2008.4702848

Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 435–445. DOI: http://dx.doi.org/10.1145/2568225.2568271

René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. 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, 654–665.

S. Kandl and R. Kirner. 2011. Error Detection Rate of MC/DC for a Case Study From the Automotive Domain. *Software Technologies for Embedded and Ubiquitous Systems* (2011), 131–142.

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

Mathworks Documentation. 2015. Types of Model Coverage. http://www.mathworks.com/help/slvnv/ug/types-of-model-coverage.html. (2015). Accessed: 2015-08-19.

Mathworks Inc. 2015. Mathworks Inc. Simulink. http://www.mathworks.com/products/simulink. (2015).

Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability* 14 (2004), 105–156.

Steven P. Miller, Michael W. Whalen, and Darren D. Cofer. 2010. Software model checking takes off. *Commun. ACM* 53, 2 (2010), 58–64. DOI: http://dx.doi.org/10.1145/1646353.1646372

A. Mockus, N. Nagappan, and T.T. Dinh-Trong. 2009. Test coverage and post-verification defects: A multiple case study. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*. 291–301. DOI: http://dx.doi.org/10.1109/ESEM.2009.5315981

A.S. Namin and J.H. Andrews. 2009. The influence of size and coverage on test suite effectiveness. (2009).

William Perry. 2006. *Effective Methods for Software Testing, Third Edition*. John Wiley & Sons, Inc., New York, NY, USA.

M. Pezzé and M. Young. 2006. *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons.

A. Rajan, M.W. Whalen, and M.P.E. Heimdahl. 2008. The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage. In *Proc. of the 30th Int'l Conf. on Software engineering*. ACM, 161–170.

S. Rayadurgam and M.P.E. Heimdahl. 2001. 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*. IEEE Computer Society, 83–91.

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

RTCA/DO-178C. Software Considerations in Airborne Systems and Equipment Certification. (????).

Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE 2015)*. ACM, New York, NY, USA.

M. Staats, G. Gay, and M.P.E. Heimdahl. 2012a. 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*. IEEE Press, 870–880.

Matt Staats, Gregory Gay, Michael Whalen, and Mats Heimdahl. 2012b. On the Danger of Coverage Directed Test Case Generation. In *Fundamental Approaches to Software Engineering*, Juan de Lara and Andrea Zisman (Eds.). Lecture Notes in Computer Science, Vol. 7212. Springer Berlin Heidelberg, 409–424. DOI: http://dx.doi.org/10.1007/978-3-642-28872-2_28

M. Staats, M.W. Whalen, and M.P.E. Heimdahl. 2011a. Better testing through oracle selection (NIER track). In *Proceedings of the 33rd Int'l Conf. on Software Engineering*. 892–895.

M. Staats, M.W. Whalen, and M.P.E. Heimdahl. 2011b. Programs, Testing, and Oracles: The Foundations of Testing Revisited. In *Proceedings of the 33rd Int'l Conf. on Software Engineering*. 391–400.

Matt Staats, Michael W. Whalen, Ajitha Rajan, and Mats P.E. Heimdahl. 2010. Coverage Metrics for Requirements-Based Testing: Evaluation of Effectiveness. In *Proceedings of the Second NASA Formal Methods Symposium*. NASA.

CAJ Van Eijk. 2002. Sequential equivalence checking based on structural similarities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 19, 7 (2002), 814–819.

M. Whalen, G. Gay, D. You, M.P.E. Heimdahl, and M. Staats. 2013. Observable Modified Condition/Decision Coverage. In *Proceedings of the 2013 Int'l Conf. on Software Engineering*. ACM.

M.W. Whalen, A. Rajan, M.P.E. Heimdahl, and S.P. Miller. 2006. Coverage metrics for requirements-based testing. In *Proceedings of the Int'l Symposium on Software Testing and Analysis*. 25–36.

Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), pp. 80–83. http://www.jstor.org/stable/3001968

Y.T. Yu and M.F. Lau. 2006. A Comparison of MC/DC, MUMCUT and Several Other Coverage Criteria for Logical Decisions. *Journal of Systems and Software* 79, 5 (2006), 577–590.