Observable Modified Condition/Decision Coverage

Michael Whalen, Gregory Gay, Dongjiang You
Mats P.E. Heimdahl
Department of Computer Science and Engineering
University of Minnesota, USA

Matt Staats
Division of Web Sciences & Technology
Korea Advanced Institute of Science & Technology, South Korea

***Abstract*—In many critical systems domains, test suite adequacy is currently measured using structural coverage metrics over the source code. Of particular interest is the modified condition/decision coverage (MC/DC) criterion required for, e.g., critical avionics systems. In previous investigations we have found that the efficacy of such test suites is highly dependent on the structure of the program under test and the choice of variables monitored by the oracle. MC/DC adequate tests would frequently exercise faulty code, but the effects of the faults would not propagate to the monitored oracle variables.**

In this report, we combine the MC/DC coverage metric with a notion of *observability* that helps ensure that the result of a fault encountered when covering a structural obligation propagates to a monitored variable; we term this new coverage criterion *Observable MC/DC (OMC/DC)*. We hypothesize this path requirement will make structural coverage metrics 1.) more effective at revealing faults, 2.) more robust to changes in program structure, and 3.) more robust to the choice of variables monitored. We assess the efficacy and sensitivity to program structure of OMC/DC as compared to masking MC/DC using four subsystems from the civil avionics domain and the control logic of a microwave. We have found that test suites satisfying OMC/DC are significantly more effective than test suites satisfying MC/DC, revealing up to 88% more faults, and are less sensitive to program structure and the choice of monitored variables.

I. Introduction

Test adequacy metrics defined over the structure of a program, such as branch coverage and modified condition/decision coverage (MC/DC) have been used for decades to assess the adequacy of test suites. Such criteria can be useful tools when evaluating a testing effort. Nevertheless, these criteria are quite sensitive to the *structure* of the program under test, e.g., the complexity of Boolean expressions [18].

In our work we have been particularly interested in the coverage criterion Modified Condition/Decision Coverage (MC/DC) [4] since it is used as an exit criterion when testing software for critical software in the avionics domain. For certification of such software, a vendor must demonstrate that the test suite provides MC/DC coverage of the source code [21]. In previous investigations, we have found that the effectiveness of MC/DC is highly dependent on the syntactic structure of the code under test. A simple syntactic transformation, such as *inlining* variables— eliminating intermediate Boolean values to create more complex decisions— can dramatically improve the effectiveness of the MC/DC criterion with increases in fault detection of up to 89% [29].

When examining the discrepancy in fault finding between test suites for non-inlined and inlined programs, we often found that the test case encountered a fault in the code, e.g., an erroneous Boolean operator, leading to a corrupted internal state, but this state was masked out in a subsequent condition and did not propagate to an output. This effect was far more prevalent in programs with many small Boolean expressions whose results were stored in intermediate values (a non-inlined implementation). Furthermore, in both non-inlined and inlined programs, it was common that a test case encountered a fault leading to a corrupted internal state, but the test case was too short to allow the corrupted state to propagate to an output; the test case terminated before the corrupted state became visible in a variable monitored by the test oracle.

The underlying issue is that structural coverage criteria such as MC/DC require only that each syntactic element—in the case of MC/DC, a particular truth assignment of a decision—is covered. Nevertheless, covering an element does not ensure faults found will be observed by a test oracle. In the case of MC/DC, the effects of masking and test length can be overcome if the test oracle monitors all variables in the program under test, i.e., all internal state variables as well as all outputs [25], but this is often prohibitively expensive. Instead, we would prefer to use a coverage criterion requiring that the result of the covered syntactic structure, e.g., a condition, be likely to propagate to the test oracle variables.

To address this issue, we have defined Observable Modified Condition/Decision Coverage (OMC/DC). OMC/DC combines the coverage of decisions required by MC/DC with a path condition that increases the likelihood that a fault encountered when executing the decision will propagate to a monitored variable. Unlike previous extensions to MC/DC [27], this path condition does not increase the number of test obligations over MC/DC; instead, it makes the existing obligations more difficult to satisfy, since the possibility of propagating a fault revealed by the MC/DC obligation must also be demonstrated. We hypothesize that this additional observability obligation will improve the effectiveness of the MC/DC criterion, particularly when used as a test generation target for automated tools, paired with output-based test oracles.

The idea of observability has been explored in hardware testing [7], [9], but our ideas extend this work in several ways. First, we provide a straightforward semantic definition of observability to ground the discussion of the metric. Second, the hardware work is *pessimistically inaccurate*; it states that

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

certain observable obligations, using our semantic definition, are not observable, making it unsuitable as a coverage target for critical software. Finally, we describe the close connection between the notion of observability and MC/DC.

In this paper, we present experiments conducted on four subsystems from the civil avionics domain and one example of the logic control of a microwave. Our results indicate that test suites generated to satisfy 100% achievable OMC/DC over non-inlined systems achieve between 2% and 88% better fault finding than test suites providing 100% achievable MC/DC when using an oracle observing the output variables only. When using an oracle observing all state variables, the advantage of OMC/DC diminishes, but it is still significant (1% to 14% better fault finding than MC/DC, with a median improvement of 4.5%). We also observed that OMC/DC is dramatically less sensitive to the structure of the program under test than MC/DC—a highly desirable trait of a structural test adequacy metric.

Based on these results, OMC/DC is—for the systems studied—a far more effective test adequacy coverage criterion, both in terms of fault finding and robustness to changes in program structure and variables monitored by the test oracle.

II. Background and Problem Statement

Modified Condition/Decision Coverage (MC/DC) is a white-box structural coverage metric developed as a compromise between the benefits of multiple-condition coverage and the lower number of test cases required by condition/decision coverage [4]. 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 (the condition of interest cannot be masked out by the other conditions in the decision). Note that when discussing MC/DC, a decision is defined to be any Boolean expression and a condition is an atomic Boolean expression with no connectives such as `and` or `or`.

While MC/DC ensures that a condition will not be masked out in a decision, it is still possible that the condition will ultimately be masked out within some sequence of statements in a program. As an example, consider the trivial program fragment in Table I. Based on the definition of MC/DC, `TestSet1` in Table I provides MC/DC over the program fragment; the test cases with `in_3 = false` (bold faced) contribute towards MC/DC of `in_1 or in_2` in stmt1. Nevertheless, if our oracle monitors the output variable `out_1`, the effect of `in_1` and `in_2` cannot be observed in the output since it will be masked out by `in_3 = false`. Thus, `TestSet1` gives us MC/DC coverage of the program fragment, but a fault on the first line will never propagate to the output. On the other hand, `TestSet2` will also give MC/DC coverage of the program, but since `in_3 = true` in the first two test cases, faults in the first statement can propagate to an output.

This masking problem can be addressed by monitoring all internal state variables, but the use of such a strong test oracle is often cost-prohibitive (or outright infeasible). An alternative approach is to strengthen the coverage criterion to include a notion of *observability* of expressions in the variables monitored by the test oracle. To this effect, in this paper we propose a new test-adequacy coverage criterion—Observable MC/DC (OMC/DC). Informally, OMC/DC establishes observability of decisions by requiring that the variable whose assignment contains a particular Boolean decision remains unmasked through a path to a variable monitored by the test oracle (commonly an output variable).

TABLE I
Sample program susceptible to masking

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

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

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

Observability is a measure of how well internal states of a system can be inferred, usually through the values of its external outputs [25]. We state that an expression in a program is *observable* in a test case if we can modify its value, leaving the rest of the program intact, and observe changes in the output of the system. If we cannot find such a value, then the expression is *not observable* for that test case.

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

This formulation is a generalization of the semantic idea behind masking MC/DC [3], lifted from decisions to programs. For masking MC/DC, the main obligation[1] is that given decision D, for each condition c in D, we want a pair of test cases t_i and t_j that ensure c is observable for both *true* and *false* values: $(D(t_i) \neq D[true/c_n](t_i)) \wedge (D(t_j) \neq D[false/c_n](t_j))$. Given this definition, one can directly lift MC/DC obligations to observable MC/DC obligations by moving the observability obligation from the decision to the program output. Given test suite T, the OMC/DC obligations are:

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

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

III. Tagged Semantics

Unfortunately, the semantic definition for observability is unwieldy both for test generation and especially for test

[1]The other obligations being that each decision evaluates to both true and false and that each entry and exit point has been invoked; these can be added to the observable MCDC criteria with no difficulty.

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

In order to define an observability constraint that more efficiently supports monitoring and test generation, we approximate semantic observability using a tagging semantics similar to [9]. We assign each condition a tag and then track the observability of these tags through the execution of a program. If a tag reaches the output, we consider the obligation satisfied. More accurately, we track pairs: the first is the condition tag (uniquely assigned for each condition instance in the program syntax), and the second is the Boolean outcome of the condition. The level of coverage for a test suite can be assessed by examining how many of all possible pairs within the program have reached an output in some test.

To demonstrate the generality of the approach, we define semantics both for an imperative command language and a simple dataflow language sharing a common set of expressions, shown in Table II. For presentation, we use a reduction semantics with evaluation contexts (RSEC) [10] which we machine checked for consistency using the K tool suite [20]. The rules operate over *configurations* that contain the syntax being evaluated ($\mathcal{K}$) and a set of labeled configuration parameters. To simplify presentation, elements of the configuration are not shown in rules if not used or modified. The rules operate by applying rewrites at positions in the syntax that are allowed by the *evaluation context* (the $Context$ definition). A context is a program or fragment of a program with a *hole*, where the hole (represented by $\Box$) defines a placeholder where a rewrite can occur. We assume appropriate definitions for maps including lookup $(\sigma\ x)$ and update $\sigma[x \leftarrow v]$ operations, the empty map $\emptyset$, and lists with concatenation $x.y$ and cons $elem :: x$, operators. During rewriting, additional syntax may be introduced; we distinguish this syntax from user-level syntax by formatting it against a gray background.

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

The imperative language semantics describe how tags propagate through commands. The main issue involves conditional statements: tags in conditions should propagate through *all* variables assigned in *either* branch, as a condition may affect the value of the variable by *not* assigning it. We extend the expression configuration to include $\mathcal{C}$: *TS* to store the set of condition tags. Conditional statements add tags to $\mathcal{C}$ that must be removed once the statement has completed. To do so, an `end` statement is appended to reset $\mathcal{C}$ and propagate the conditional tags to all variables assigned in the conditional

TABLE II
SYNTAX AND TAGGING SEMANTICS FOR AN IMPERATIVE AND DATAFLOW LANGUAGE

Expression syntax, context, and semantics:

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

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

Imperative command syntax, context and semantics:

$$
\begin{array}{rcl}
S & ::= & \texttt{skip} \mid S; S \mid \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \mid \\
 & & \mathit{Id} := E \mid \texttt{while}\ E\ \texttt{do}\ S \mid \texttt{end}\ (\textbf{List}\ \mathit{Id}, \mathit{TS}) \\
\mathit{Context} & ::= & \ldots \mid \mathit{Id} := \mathit{Context} \mid \texttt{if}\ \mathit{Context}\ \texttt{then}\ S\ \texttt{else}\ S \mid \\
 & & \mathit{Context}; S \mid \langle \mathcal{K} : \mathit{Context}, \mathcal{E} : \mathit{Env}, \mathcal{C} : \mathit{TS} \rangle
\end{array}
$$

asgn	$\langle \mathcal{E} : \sigma \rangle[x := (n, l)] \Rightarrow \langle \mathcal{E} : \sigma[x \leftarrow (n, l)] \rangle[\texttt{skip}]$
seq	$\texttt{skip};\ s_2 \Rightarrow s_2$
cond$_1$	$\langle \mathcal{C} : c \rangle[\texttt{if}\ (\texttt{tt}, l)\ \texttt{then}\ s_1\ \texttt{else}\ s_2] \Rightarrow$ $\langle \mathcal{C} : c \cup l \rangle[s_1;\ \texttt{end}(V, c)]$ where $V = (\mathit{Assigned}\ s_1).(\mathit{Assigned}\ s_2)$
cond$_2$	$\langle \mathcal{C} : c \rangle[\texttt{if}\ (\texttt{ff}, l)\ \texttt{then}\ s_1\ \texttt{else}\ s_2] \Rightarrow$ $\langle \mathcal{C} : c \cup l \rangle[s_2;\ \texttt{end}(V, c)]$ where $V = (\mathit{Assigned}\ s_1).(\mathit{Assigned}\ s_2)$
while	$\texttt{while}(e)\ s \Rightarrow \texttt{if}\ (e)\ \texttt{then}\ (s; \texttt{while}(e)\ s)\ \texttt{else skip}$
endcond$_1$	$\langle \mathcal{C} : c' \rangle[\texttt{end}\ (\texttt{nil}, c)] \Rightarrow \langle \mathcal{C} : c \rangle[\texttt{skip}]$
endcond$_2$	$\langle \mathcal{E} : \sigma, \mathcal{C} : c' \rangle[\texttt{end}\ (x :: V, c)] \Rightarrow \langle \mathcal{E} : \sigma', \mathcal{C} : c' \rangle[\texttt{end}\ (V, c)]$ where $(\sigma\ x) = (n, l)$ and $\sigma' = \sigma[x \leftarrow (n, l \cup c')]$
prog	$s \Rightarrow \langle \mathcal{K} : s, \mathcal{E} : \emptyset, \mathcal{C} : \emptyset \rangle$

Dataflow program syntax, context, and semantics:

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

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

body (using (*Assigned* s), a helper function that returns the list of variables assigned by a statement s). Given a context containing input variable values, these rules determine the set of tags that propagate to outputs.

Dataflow languages, such as Simulink and SCADE, are popular for model-based development, and assign values to a set of equations in response to periodic inputs. To store system state, state variables ($\frac{1}{z}$ blocks in Simulink) are used. Our dataflow language consists of assignments to combinatorial and state variables, and the semantics are defined over lists (traces) of input variable values. The expression configuration is extended to contain an input trace I, output trace O, and state environments S. Evaluation proceeds by cycles: at the beginning of a cycle, the **cycle** rule constructs the initial evaluation environment. During a cycle, variable values are recorded using the **comb** and **state** rules. Note that the context does not force an ordering on evaluation of equations; instead, an equation can evaluate as soon as all variables it uses have been stored in the environment. When all equations have been computed, the **write** rule appends the environment to the output list. The **prog** rule, given an input list, an initial state environment, and a list of equations, initializes the configuration for the **cycle** rule. Coverage can be determined by examining the tags stored in the output environment list.

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

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

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

IV. Test Generation for Dataflow Programs

In the previous section, we presented an extended semantics that accounts for tags in imperative and dataflow programs. In order to generate tests, we would like to instead annotate the program and describe *trap properties* to track the tags. We will generate test obligations such that each obligation is suitable for tracking a *single* tag and determining whether it propagates to an output. This is accomplished by conjoining an MC/DC coverage obligation over a single variable (as described in [18]) with a path condition representing the variable's observability at one of the monitored variables. We describe this annotation for the dataflow language Lustre [13] in order to measure test coverage of industrial Simulink models in Section V.

A. Immediate Non-Masking Paths

A variable x is observable if it is not masked along some computation path (as described in the tagged semantics) to a monitored variable. If the path is entirely within one computational step (i.e., it does not go through any delays), we call it an *immediate non-masking path* and the variable is *immediately observable*. This can be defined inductively by examining the variables that use x in their definition: if one of these variables y is immediately observable, and x is not masked in the definition of y, then x is immediately observable. We track these notions by defining additional variables to track this information: *x_IMM_USED_BY_y* which is true if x is not masked in the definition of y, and *x_IMM_OBSERVED* if x has an immediate non-masking path. Suppose we had the following equations, where `out1` is an observed variable:

```
out1 = v1 and v2 ;
v1 = true ;
v2 = if (input1) then (v3) else (v1) ;
v3 = true ;
```

In this case, we could generate additional definitions to track the observability of the variables as follows:

```
v1_IMM_USED_BY_out1 = v2 ;
v2_IMM_USED_BY_out1 = v1 ;
input1_IMM_USED_BY_v2 = true ;
v3_IMM_USED_BY_v2 = input1 ;
v1_IMM_USED_BY_v2 = (not input1) ;

out1_IMM_OBSERVED = true ;
v1_IMM_OBSERVED =
  (v1_IMM_USED_BY_out1 and out1_IMM_OBSERVED) or
  (v1_IMM_USED_BY_v2 and v2_IMM_OBSERVED) ;
v2_IMM_OBSERVED =
    v2_IMM_USED_BY_out1 and out1_IMM_OBSERVED ;
input1_IMM_OBSERVED =
    input1_IMM_USED_BY_v2 and v2_IMM_OBSERVED ;
v3_IMM_OBSERVED =
    v3_IMM_USED_BY_v2 and v2_IMM_OBSERVED ;
```

$v1$ is used in two equations and therefore has two immediate paths to observability: one through $v2$ and another directly through $out1$, while the other variables are each used once, so have one immediate path.

B. Delayed Non-Masking Paths

Although many variables can be immediately observed, often, the effect of a variable on an output can only be observed after several steps. In each of these intermediate steps, its tag is stored in a delay, until it eventually propagates to an output. We call this a *delayed non-masking path* and the variable is *delay observable*. This situation can be broken into immediate observations: the first from a variable to a latch, the next from the latch to another latch, etc., until an output is reached. Suppose we had the following Lustre program[2]:

```
delay1 = 0 -> pre(v1) ;
v1 = v2 and delay2 ;
v2 = in1 ;
delay2 = 0 -> pre(in1) ;
```

In the same way that we modeled immediate observability, we could talk about immediate use by delay equations:

[2]In Lustre, latches are represented slightly differently than in the language in Section II: the Lustre equation `var = init -> pre(expr)` contains both the initial value of the latch $init$ and the latch expression $pre(expr)$, whereas our semantics imports the initial value of latches through an initial latch environment s.

```
v2_IMM_USED_BY_v1 = delay2 ;
delay2_IMM_USED_BY_v1 = v2 ;
v1_DEL_USED_BY_delay1 = true ;
v2_DEL_USED_BY_delay1 = v2_IMM_USED_BY_v1
                        and v1_DEL_USED_BY_delay1 ;
delay2_DEL_USED_BY_delay1 = delay2_IMM_USED_BY_v1
                        and v1_DEL_USED_BY_delay1 ;
```

We now have a mechanism that defines immediate paths to latches. What is necessary is some means to knit these paths together to define a sequential path through (possibly) several delays to an output. We accomplish this using a *token* variable that describes the current delay location. Once the token is initialized to a delay variable X, it can non-deterministically move to any other delay location (as long as X is `DEL_USED_BY` that location) or to a special `COMPLETE` state (if X is immediately observed). If the token cannot move, because it is not observable at another delay or the output, the token moves to an `ERROR` state and stays there.

C. Test Obligations

An OMC/DC coverage obligation can be represented as an MC/DC obligation over a single variable conjoined with a path condition describing the observability of that variable at one of the monitored variables. For delayed paths, we have to describe the instant in which the expression was immediately observable at a delay (called *capture*). We then want to latch this fact for the rest of the execution, hoping that the token will propagate to an output. So, the test consists of the original MC/DC obligation (`v2_AT_v1_TRUE`) below and a path condition, which can be satisfied by either be an *immediate* or *delayed* path, as described in the following Lustre code:

```
v2_AT_v1_TRUE = in1 and delay2 ;
v2_AT_v1_TRUE_CAPTURE = (v2_AT_v1_TRUE and
    (v1_DEL_USED_BY_delay1 and token=delay1) ;
v2_AT_v1_TRUE_CAPTURED = v2_AT_v1_TRUE_CAPTURE ->
          (v2_AT_v1_TRUE_CAPTURE or
          pre(v2_AT_v1_TRUE_CAPTURED))
v2_true_ob = ((v2_AT_v1_TRUE and v1_IMM_OBSERVED)
   or (v2_AT_v1_TRUE_CAPTURED and token=TOK_COMPLETE))
```

V. Evaluation & Experiment

We wish assess the *quality* in terms of fault finding of the test suites generated to satisfy OMC/DC as compared to masking MC/DC [3]. We also want to evaluate the effect of program structure on the effectiveness of test suites generated to provide OMC/DC. Thus, we address the following questions:

1) Are test suites generated to provide OMC/DC more effective at revealing faults than test suites generated to satisfy masking MC/DC?
2) How robust is the OMC/DC criterion to the structure of the program under test? Will the effectiveness of OMC/DC change as program structure changes?

Additionally, we are interested in the nature of the tests generated to satisfy the OMC/DC and MC/DC coverage criteria:

3) How do the length of the individual tests, the size of test suites, and the percentage of achievable coverage compare between OMC/DC and MC/DC for the systems included in our experiment?

A. Experimental Setup Overview

In this research, we have used four industrial systems developed by Rockwell Collins engineers. Two of these systems, *DWM1* and *DWM2*, represent distinct portions of a Display Window Manager (DWM) for a cockpit display system. The other two systems, *Vertmax* and *Latctl*, describe the vertical and lateral mode logic for a Flight Guidance System. In addition, we have used a Microwave System — control software for a generic microwave oven developed as a non-proprietary teaching aid at Rockwell Collins.

Each of the systems under test represent sizable, realistic industrial systems. The size of each system and number of variables are listed in Table III.

TABLE III
Case Example Information

	Subsysts.	Blocks	Outputs	Internal Vars.
DWM1	3109	11,439	7	569
DWM2	128	429	9	115
Vertmax	396	1,453	2	415
Latctl	120	718	1	128
Microwave	22	101	4	162

For each case example, we generated inlined and non-inlined versions of each system (detailed in Section V-B below). Then, for each implementation of each system, we:

1) Generated 10 test input suites each for OMC/DC and MC/DC (Section V-C).
2) Generated 250 mutants of each system (Section V-D).
3) Ran test suites on mutants with output-only and maximum test oracles (Section V-E).
4) Assessed fault finding of each test suite and oracle combination. (Section V-E).

B. Inlined and Non-Inlined Implementations

Our subject systems are modeled using the Simulink notation from Mathworks Inc. [17] and were automatically translated into the Lustre synchronous programming language [13] in order to take advantage of existing automation. This is analogous to the automated code generation from Simulink offered by Mathworks' Real Time Workshop. Lustre can be automatically translated to C code.

When translating these systems from Simulink to Lustre, a number of options exist on how to structure the generated code. For example, one can factor complex boolean expressions through the introduction of additional variables or one can inline expressions to reduce the number of variables while increasing the complexity of the boolean expressions. As we know the structure of the program under test influences the effectiveness of the MC/DC criterion [18], we have generated both inlined (complex boolean conditions) and non-inlined systems (intermediate variables used to factor expressions).

C. Test Suite Generation

We use a counterexample-based test generation approach to generate tests satisfying masking MC/DC and OMC/DC [11][19]. This approach is guaranteed to generate a

test suite that achieves the maximum possible coverage of the system under test. We have used the Kind model checker [12] in our experiments.

The obligations generated to satisfy OMC/DC will differ depending on the set of monitored variables. In this study, we generate OMC/DC with respect to the output variables of each system, as an output-only oracle is most likely to be used in practice. Note when using the maximum oracle, all variables are observable, and so the OMC/DC obligations are equivalent to MC/DC obligations.

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

In these experiments, we have produced 10 different test suites for each case example and program structure to eliminate the possibility that we by accident create a very good (or very poor) test suite in the test suite reduction step.

D. Mutant Generation

Mutation testing is the practice of automatically generating *faulty* implementations of a system for the purpose of empirically examining the fault-finding potential of a test suite [6]. During mutation testing, clones of the system under test are created by introducing a single fault into the program. This method is designed such that all mutants produced are both syntactically and semantically valid. That is, the mutants will compile, and no mutant will "crash" the system under test.

The mutation operators used in this experiment are similar to those used by other researchers, for example, arithmetic, relational, and boolean operator replacement, boolean variable negation, constant replacement, and delay introduction (that is, use the stored value of the variable from the previous computational cycle rather than the newly computed value). A detailed description is available in [18].

For each case example, we created 250 mutants. We then remove *functionally equivalent* mutants from each evaluation set using the Kind model checker. This is possible due to the nature of the systems examined in this research. Each system is finite; therefore, determining equivalence is decidable (and, in practice, fast)[3].

[3]Equivalence checking is common in the hardware domain; Van Eijk provides a nice introduction [26].

E. Test Oracles and Data Collection

For the inlined and non-inlined implementations of each case example, we ran the reduced test suites against each mutant and the original version of the system. For each test suite, we recorded the value of every internal variable and output at every step of the execution of every test case using an in-house Lustre interpreter.

To determine the fault finding effectiveness of the generated test suites, we paired each suite with two different *expected value test oracles* [25]. When using an expected value oracle, for each test input, concrete values are specified that the system is expected to produce for one or more variables monitored by the oracle (internal states and/or outputs). We have found that this form of test oracle is commonly used by our industrial partners in the testing of critical software systems. In this study, we have chosen two expected value test oracles, (1) *output-only*, an oracle that compares expected and actual values for each of the system's output variables, and (2) *maximum*, an oracle that compares values for all internal and output variables.

We compute the fault finding of an oracle/test suite pairing as the percentage of mutants killed. We perform this analysis for each oracle and test suite for every case example.

VI. Results & Discussion

In this section, we address our research questions and discuss the implications of our results. We begin by presenting the median percent of seeded faults revealed by each combination of test suite and oracle type in Table IV (plotted in Figure 1).

A. RQ1: Fault Finding Effectiveness

We would first like to determine whether OMC/DC performs better than MC/DC with respect to fault finding. To address this question we formulated the following hypothesis:

H_1: For a given oracle and program structure, the test suite satisfying OMC/DC reveals more faults than the test suite satisfying MC/DC.

This is paired with the appropriate null hypothesis:

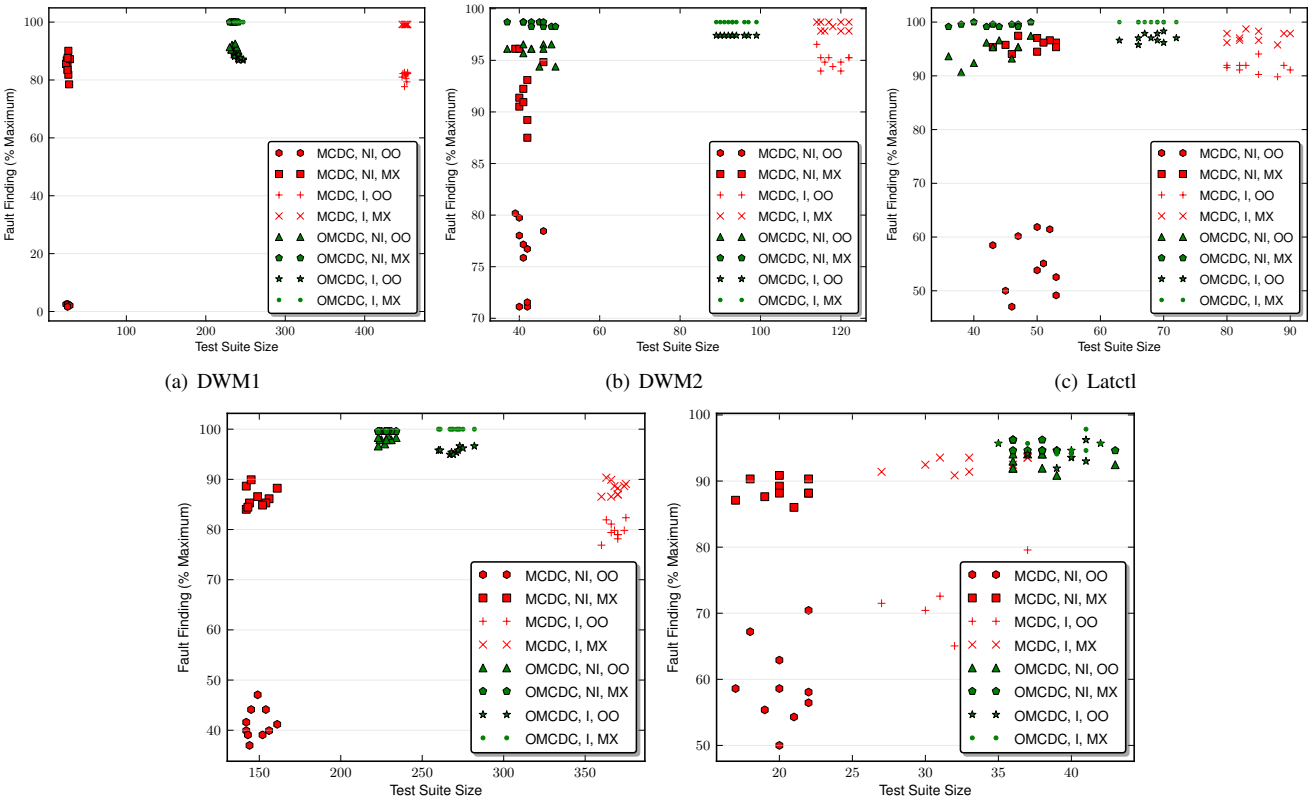
H_0: For a given oracle and program structure, the fault finding results for the test suite satisfying OMC/DC are drawn from the same distribution as the fault finding results for the test suite satisfying MC/DC.

Our observations are drawn from an unknown distribution. Therefore, we use a two-sided Mann-Whitney-Wilcoxon rank-sum test [30], 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, program structure, and oracle type with $\alpha = 0.05$.

Our results indicate that the null hypotheses can be rejected for all combinations of case examples, program structures, and oracle types, with $p < 0.001$. Furthermore, we can see in Table IV that test suites satisfying OMC/DC outperform those satisfying MC/DC in all cases, with improvements in fault

TABLE IV
PERCENT OF MUTANTS KILLED FOR EACH CASE EXAMPLE, MEDIAN OVER 10 REDUCED TEST SUITES

		DWM1	DWM2	Latctl	Vertmax	Microwave
Non-Inlined	**OMC/DC Output-Only**	91%	96%	95%	98%	93%
	MC/DC Output-Only	3%	77%	55%	41%	59%
	OMC/DC Maximum	100%	99%	99%	99%	95%
	MC/DC Maximum	86%	92%	96%	86%	89%
Inlined	**OMC/DC Output-Only**	88%	97%	97%	96%	95%
	MC/DC Output-Only	82%	95%	92%	80%	73%
	OMC/DC Maximum	100%	99%	100%	100%	95%
	MC/DC Maximum	99%	98%	97%	89%	93%



(a) DWM1 (b) DWM2 (c) Latctl

(d) Vertmax (e) Microwave

Fig. 1. Percent of mutants killed, plotted against reduced test suite size, for each implementation and oracle pairing for each system.

finding of 19-88% for non-inlined systems using an output-only oracle (3-14% when using a maximum oracle) and 2-22% for inlined systems when using an output-only oracle (1-11% when using a maximum oracle).

We therefore *accept* H_1—for all combinations of system, program structure, and oracle type—and conclude that test suites satisfying OMC/DC provide a statistically significant and practical improvement over those satisfying MC/DC.

B. RQ2: Sensitivity to Program Structure

Previous research indicates that MC/DC is sensitive to program structure [18]; inlining a program generally makes it more difficult to achieve MC/DC over a program, but yields a large improvement in fault finding for test suites satisfying MC/DC, with one study demonstrating increases of 9-89% [29]. As seen in Table IV and summarized in Table V, we see similar increases of 14-79% for output-only oracles.

This sensitivity is undesirable for any criterion, as the value of satisfying the criterion depends heavily on how the program is written; indeed, the sensitivity of MC/DC to program structure, and its negative impact on automated test generation, is one of the motivations for the creation of OMC/DC. The cause of MC/DC's sensitivity relates to (1) errors not propagating to observed variables due to masking, coupled with (2) inlining imposing additional constraints when computing MC/DC obligations over complex expressions, which limits opportunities for masking.

In the studied examples, OMC/DC is far less sensitive to changes in program structure. When paired with an output-only oracle, we see a median improvement of 1% from inlining (and, in two cases, actually see a -2 to -3% decrease in fault finding). When using a maximum oracle, the effect is negligible. Thus, while inlining has been proposed as a solution to the masking and propagation-related deficiencies

of MC/DC [29], the addition of the path condition required by OMC/DC, which explicitly addresses issues with masking, is also an effective, and—in our opinion—cleaner and more straightforward solution.

TABLE V
MEDIAN IMPROVEMENT FROM INLINING

Case Example	Oracle	OMC/DC	MC/DC
DWM1	Output-Only	-3%	79%
	Maximum	0%	13%
DWM2	Output-Only	1%	18%
	Maximum	0%	6%
Latctl	Output-Only	2%	37%
	Maximum	1%	1%
Vertmax	Output-Only	-2%	39%
	Maximum	1%	3%
Microwave	Output-Only	2%	14%
	Maximum	0%	4%

On a related note: the use of a maximum oracle has also been proposed as a method of addressing issues with propagation. As we can see in Table IV, generally test suites satisfying OMC/DC with an output-only oracle perform the same or better than suites satisfying MC/DC with a maximum oracle (though, gains are low to negligible). Only in one case—*DWM1* when inlined—do suites satisfying MC/DC achieve significantly higher median fault finding (99% versus 88%).

C. RQ3: Test Suites and Coverage Obligations

Our final question is concerned with the cost and challenge of satisfying OMC/DC. In particular, does satisfying OMC/DC require more test inputs, must the test inputs be longer, and what percentage of obligations are uncoverable, i.e., cannot be achieved by any test input? Note that as OMC/DC requires the same obligations as MC/DC (with an additional path condition), the total number of obligations required to cover a system will be the same for both criteria.

From the information in Table VI, we can see that (1) there is no definitive pattern for test suite size; sometimes reduced suites for OMC/DC are larger than those for MC/DC and vice-versa, and (2), OMC/DC obligations are more likely to be uncoverable, with as low as 57.5% of obligations coverable.

With respect to size, we can see that for non-inlined systems, OMC/DC generally requires more—sometimes many more—test inputs than MC/DC. This reflects the strength of OMC/DC: the observability requirements ensure test cases are more diverse with respect to which paths they must take; thus, it is less likely that several test cases will cover the same obligation. Naturally, more test cases are needed to satisfy 100% achievable OMC/DC. Also, we observed that OMC/DC test case lengths (not shown for space reasons) are slightly longer than that of MC/DC, as test cases must take additional steps through delays to propagate certain variables to outputs.

Nevertheless, while we naturally do not wish to increase the cost of testing, we believe ensuring propagation during testing is necessary; otherwise, why bother to exercise a condition at all? Furthermore, we believe the substantial improvements in testing effectiveness justify the cost, particularly in the domain of critical systems.

The increase in uncoverable obligations is more concerning. For MC/DC, a coverage obligation is uncoverable if it is impossible to demonstrate that a condition can independently affect the outcome of a decision. This situation can occur through interrelationships between conditions in a decision making certain truth assignments impossible. For OMC/DC, this can also cause uncoverable obligations, as can the inability to propagate values to monitored variables. The path constraints accounts for the increase in uncoverable obligations.

Uncovered obligations for structural coverage criteria may reflect problems with the code (conditions that are not properly influencing decisions) or simply eccentricities in the code (e.g., dead code or interrelationships between conditions in a decision). In our case, since we are using a verification tool for test generation, the tool will simply produce maximum achievable coverage with a proof that the uncovered obligations are truly uncoverable. However, if OMC/DC was adopted as a coverage criterion in conjunction with other test generation techniques (e.g., manual or based on heuristic search), it may become necessary—as it currently is with MC/DC in the context of critical avionics systems—to demonstrate that all necessary test have been found, i.e., to demonstrate that all uncovered obligations are indeed uncoverable. This is already a challenging task for MC/DC, and would be made more challenging with the addition of path constraints; how to address this problem is a topic for future research.

VII. RELATED WORK

Lustre and Function Block Diagram (FBD) are data-flow languages that describe how inputs are transformed into outputs instead of describing the control flow of the program. Researchers studying coverage metrics for Lustre [15] and FBD [14] implicitly investigated observability by examining variable propagation from the inputs to the outputs.

Structural coverage metrics for Lustre are based on *activation conditions* that are defined as the condition upon which a data flow is transferred from the input to the output of a path. When the activation condition of a path is true, any change in input causes modification of the output within a finite number of steps [15]. Coverage metrics for FBD are based on a *d-path condition* that is similar to activation conditions in Lustre [14].

Coverage metrics in Lustre and FBD are different from OMC/DC in several respects. First, these metrics check if specific inputs affect the outputs and measure the coverage of variable propagation on all possible paths. OMC/DC, on the other hand, checks if each atomic condition in a Boolean expressions affects the monitored variables, and determines if a path exists which propagates the effect of the condition. Second, OMC/DC (as well as MC/DC) is stronger in terms of how a decision must be exercised.

Observability has been studied in testing of hardware logic circuits. Observability-based code coverage metric (OCCOM) is a technique where tags are attached to internal states in a circuit and the propagation of tags is used to predict the actual propagation of errors (corrupted state) [7], [9]. A variable is tagged when there is a possible change in the value of the

TABLE VI
NUMBER OF ACHIEVABLE TEST OBLIGATIONS AND MEDIAN REDUCED TEST SUITE SIZE

Case Example	Structure	Total Obligations	OMC/DC Achievable	MC/DC Achievable	OMC/DC Reduced Size	MC/DC Reduced Size
DWM1	Non-Inlined	2038	2036 (99.9%)	2038 (100%)	236	26
	Inlined	2894	1987 (68.7%)	2838 (98.1%)	244	450
DWM2	Non-Inlined	382	343 (89.8%)	364 (95.3%)	43	40
	Inlined	830	477 (57.5%)	538 (64.8%)	92	118
Latctl	Non-Inlined	380	355 (93.4%)	380 (100%)	47	51
	Inlined	260	241 (92.7%)	259 (99.6%)	70	80
Vertmax	Non-Inlined	1732	1700 (98.2%)	1732 (100%)	228	152
	Inlined	1232	1188 (96.4%)	1221 (99.1%)	272	360
Microwave	Non-Inlined	472	325 (68.9%)	467 (98.9%)	36	22
	Inlined	468	338 (72.2%)	441 (94.2%)	40	32

variable due to an fault. The observability coverage can be used to determine whether erroneous effects that are activated by the inputs can be observed at the outputs.

The key differences between OMC/DC and OCCOM are twofold: (1) OMC/DC investigates variable value propagation, while OCCOM investigates fault propagation and (2) OCCOM has pessimistic inaccuracy because of tag cancelation. When both positive and negative tags exist in the same assignment (e.g., different tags in an ADDER or the same tags in a COMPARATOR cancel each other out), no tag is assigned [7] or an unknown tag "?"[9] is used. Variables without tags or with unknown tags are not considered to carry an observable error. In OMC/DC, since we do not make a distinction between positive and negative tags, we do not have tag cancelation or the corresponding pessimistic inaccuracy. Extended work in [8] may fix pessimistic inaccuracy by producing test vectors with specific values, but is highly infeasible.

Dynamic taint analysis, or dynamic information flow analysis, marks and tracks data in a program at runtime, similar to our tagging semantics. This technique has been used in security as well as software testing and debugging [16], [5]. Taint propagation occurs in both explicit information flow (i.e., data dependencies) and implicit information flow (control dependencies). Although the way in which markings are combined varies based on the application, the default behavior is to union them [5]. Thus, dynamic taint analysis is conservative and does not consider masking. More accurate techniques for information flow modeling, such as [28], define path conditions quite similar to those used in this paper to prove *non-interference*, that is, the non-observability of a variable or expression on a particular output.

Dynamic program slicing [1] computes a set of statements that influence the variables used at a program point for a particular execution. This can identify all variables that contribute to a specific program point, including output. However, similarly to dynamic taint analysis, it does not consider masking. Checked coverage uses dynamic slicing to assess oracle quality, where oracles are program assertions [22]. Given a test suite, it yields a percentage of all statements that contribute to the value of any assertion (i.e., are observable at that assertion) vs. the total number of statements covered by the test suite. This work is designed to assess the *oracle*, not the test suite.

VIII. THREATS TO VALIDITY

External Validity: We have chosen to focus on five synchronous reactive critical systems. We believe these systems are representative of the avionics domain and our results are, therefore, generalizable to other systems in this domain.

We have used Lustre [13] as an implementation language rather than a more common language, such as C or C++. As noted in Section V-B, in this domain, systems written in Lustre are similar in structure to systems written in C or C++. Thus, we believe our results are applicable to programs written in more traditional imperative languages.

We have generated approximately 250 mutants for each program structure of each case example. This number was chosen to yield a reasonable cost for the study. It is possible the number of mutants is too low. Based on past experience, however, we have found results using fewer than 250 mutants to be representative [23], [24]. Additional studies have yielded evidence that results plateau when using over 100 mutants.

Internal Validity: We have used a model checker (Kind [12]) to generate test cases. This generation approach provides the shortest test cases that provide the desired coverage. It is possible that test cases derived by hand or through some other automated means, e.g., through heuristic search, may provide different results.

Construct Validity: We measure the fault finding over seeded faults, rather than real faults encountered during development; it is possible that using real faults would lead to different results. However, Andrews et al. have shown that the use of seeded faults like ours leads to conclusions similar to those obtained using real faults in similar experiments [2].

IX. CONCLUSIONS & FUTURE WORK

Structural coverage metrics, such as MC/DC, are commonly used to measure the adequacy of test suites. Such criteria require only that certain code structures, such as a particular Boolean assignment of a decision, be exercised, without requiring the resulting value to affect an observable point in the program. As a result, test suites satisfying these criteria can produce corrupted internal state without revealing a fault, resulting in wasted testing effort.

To address this, we have proposed Observable MC/DC, a combination of traditional MC/DC testing with a notion

of observability—an additional path constraint, which helps ensure that faults will be observed through a *non-masking* path from the point the obligation is satisfied to a variable monitored by the test oracle. Our results indicate that test suites generated to satisfy achievable OMC/DC locate a median of 17.5% (and up to 88%) more faults than test suites providing MC/DC coverage when paired with an oracle observing only the output variables. Furthermore, we have also observed that OMC/DC is less sensitive to the structure of the program under test than MC/DC, and also provides the benefits of using a very strong test oracle with MC/DC coverage.

While our results are encouraging, there are a number of areas open to explore in future research, including:

- **Oracle data selection:** OMC/DC test obligations are defined in terms of both the system structure and a test oracle. In this work, we have paired the case examples with an output-only oracle, but an intelligently selected set of internal and output variables (such as [24]) could potentially yield more cost-effective test suites.
- **Comparison to other coverage metrics:** We have directly compared the performance of OMC/DC to MC/DC. However, there exist many other test adequacy metrics. In particular, we would like to compare the effectiveness and cost of generation of OMC/DC to black-box requirements metrics, which have been previously shown to be adept at propagating faults to the output level [23].
- **Applying observability to other metrics:** The current notion of observability is general and could be adapted to orthogonal metrics such as boundary-value coverage.

ACKNOWLEDGMENT

This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A; NSF grants CCF-0916583, CNS-0931931, and CNS-1035715; an NSF graduate fellowship; and the World Class University program under the National Research Foundation of Korea (Project No: R31-30007).

We also thank the Advanced Technology Center at Rockwell Collins Inc. for granting access to industrial case examples.

REFERENCES

[1] H. Agrawal and J. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, 1990.

[2] J. Andrews, L. Briand, Y. Labiche, and A. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608 –624, aug. 2006.

[3] J. Chilenski. 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., April 2001.

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

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

[6] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE computer*, 11(4):34–41, 1978.

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

[8] F. Fallah, P. Ashar, and S. Devadas. Functional vector generation for sequential HDL models under an observability-based code coverage metric. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(6):919–923, 2002.

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

[10] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, Sept. 1992.

[11] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.

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

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

[14] E. Jee, J. Yoo, S. Cha, and D. Bae. A data flow-based structural testing technique for FBD programs. *Information and Software Technology*, 51(7):1131–1139, 2009.

[15] A. Lakehal and I. Parissis. Structural test coverage criteria for Lustre programs. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 35–43, 2005.

[16] W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, pages 198–209, 2004.

[17] Mathworks Inc. Simulink product web site. http://www.mathworks.com/products/simulink.

[18] A. Rajan, M. Whalen, and M. Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *Proc. of the 30th Int'l Conference on Software engineering*, pages 161–170. ACM New York, NY, USA, 2008.

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

[20] G. Rosu and T. F. Serbanuta. An overview of the k semantic framework. *J. of Logic and Algebraic Programming*, 79(6):397 – 434, 2010.

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

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

[23] M. Staats. *The Influence of Multiple Artifacts on the Effectiveness of Software Testing*. PhD thesis, University of Minnesota, 2011.

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

[25] M. Staats, M. Whalen, and M. Heimdahl. Better testing through oracle selection (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 892–895, 2011.

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

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

[28] M. W. Whalen, D. A. Greve, and L. G. Wagner. *Model Checking Information Flow*. Springer-Verlag, Berlin Germany, March 2010.

[29] M. W. Whalen, M. P. Heimdahl, A. Rajan, and M. Staats. On MC/DC and implementation structure: An empirical study. In *Proceedings of the 27th Digital Avionics Systems Conference*, October 2008.

[30] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):pp. 80–83, 1945.