Efficient Observability-based Test Generation by Dynamic Symbolic Execution

Dongjiang You[1], Sanjai Rayadurgam[1], Michael Whalen[1], Mats P.E. Heimdahl[1], Gregory Gay[2]
[1]Department of Computer Science and Engineering, University of Minnesota, USA
[2]Department of Computer Science and Engineering, University of South Carolina, USA
Email: [djyou, rsanjai, whalen, heimdahl]@cs.umn.edu, greg@greggay.com

***Abstract*—Structural coverage metrics have been widely used to measure test suite adequacy as well as to generate test cases. In previous investigations, we have found that the fault-finding effectiveness of tests satisfying structural coverage criteria is highly dependent on program syntax – even if the faulty code is exercised, its effect may not be observable at the output. To address these problems, observability-based coverage metrics have been defined. Specifically, Observable MC/DC (OMC/DC) is a criterion that appears to be both more effective at detecting faults and more robust to program restructuring than MC/DC. Traditional counterexample-based test generation for OMC/DC, however, can be infeasible on large systems. In this study, we propose an incremental test generation approach that combines the notion of observability with dynamic symbolic execution. We evaluated the efficiency and effectiveness of our approach using seven systems from the avionics and medical device domains. Our results show that the incremental approach requires much lower generation time, while achieving even higher fault finding effectiveness compared with regular OMC/DC generation.**

I. Introduction

Test adequacy criteria defined over the structure of a program, such as branch coverage and modified condition/decision coverage (MC/DC) serve as useful benchmarks for assessing the thoroughness of testing software and for meeting regulatory requirements. Of particular interest to us are criteria used to evaluate testing effort for safety-critical systems, such as MC/DC [1], which is mandated by the U.S. standard DO-178C [2] for testing the most critical avionics software. In previous investigations, we have found that the effectiveness of structural coverage criteria is highly dependent on the structure of the program under test [3]. Simple syntactic transformations, such as inlining of variables, have a dramatic effect on the fault-finding efficacy of test suites designed to satisfy the MC/DC criterion. This effect is not surprising – structural criteria, as the name suggests, require that certain code structures, such as branches or Boolean expressions, be exercised to a certain level of thoroughness. However, the degree to which fault-finding was impacted by simple syntactic changes to the program is bothersome, especially given that such criteria are used to assess the testing effort for safety-critical software. These structural criteria specify that various structural parts of a program must be exercised, they do not, however, mandate that the effect of exercising that part must manifest itself at a subsequent *observable* point in the program; a frequent problem is that the effect of a corrupted variable gets masked out through its use in subsequent operations.

To address this problem, *observability* has been proposed as a desirable attribute of testing in both hardware [4] and software domains [5]. Specifically, Whalen et al. proposed OMC/DC – a combination of traditional MC/DC with the notion of observability [5]. In practical terms, OMC/DC adds an additional path constraint to the coverage obligation. This constraint concretizes the idea that the *effect* of satisfying the MC/DC obligation must propagate to an observable output. The OMC/DC criterion defines propagation (optimistically) by specifying what constitutes *masking* and mandating a non-masking path from the point of exercising the code structure to some monitored output variable. This greatly increases the likelihood of faults triggered during testing to be observed as failures. It was demonstrated that OMC/DC significantly outperforms MC/DC with respect to fault finding and robustness to syntactic transformations on industrial models from the avionics and medical device domains [5], [6].

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

In this paper, we propose an incremental test generation strategy that addresses this scalability problem using *dynamic symbolic execution*, in which test inputs are generated incrementally by combining concrete and symbolic executions of the program under test [7], [8]. Starting with some concrete values for inputs to obtain a concrete execution path, path constraints are then generated for that execution path by treating (some of the) variables as symbolic. These constraints are then systematically modified and solved to force the program to take different (but feasible) execution paths. There

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

has been a large volume of research on dynamic symbolic execution that demonstrates encouraging improvement over classic symbolic execution [9], [10], [11], [12]. Efficiency is gained by modifying the path constraints along a known concrete path, which localizes the search for newer feasible paths. This reduces the high computational cost associated with symbolic search and, in practice, leads to better coverage of the program paths and higher likelihood of revealing faults.

In the test generation approach presented in this paper, we adopt the dynamic symbolic execution idea in a novel approach to generate tests that, first, satisfy a condition based coverage criterion to exercise a particular part of the code (MC/DC) and, second, satisfy a dataflow criterion propagating the possibly corrupted state to a point where it is used by an observable output. To this effect, we employ a tagging semantics similar to the one defined by Whalen et al. [5] where a tag is assigned to each condition and the propagation of tags to outputs approximates observability. Instead of invoking a model checker to generate a complete OMC/DC test, which can be prohibitively expensive due to the test case length, the incremental test generation starts with concrete test inputs that satisfy an obligation and then invokes the model checker at each test step repeatedly to solve path conditions in an attempt to propagate tags through non-masking paths towards outputs. A test case satisfying an obligation incrementally grows and eventually tags associated with the condition of interest (the condition the MC/DC obligation exercised) reach output variables (to form an OMC/DC test).

While this approach introduces some incompleteness – a tag associated with a condition may not be able to propagate to outputs on the particular concrete path being extended – this approach worked remarkably well in our experiments. Our results indicate that the incremental approach consumes significantly less time than regular test generation, while achieving universally better fault finding effectiveness.

II. Background

Coverage metrics can often be used to measure test suite adequacy as well as to generate test cases. Modified condition/decision coverage (MC/DC) is a structural coverage metric used in some safety-critical systems domains. In order to satisfy MC/DC, (1) each condition in a decision has to take on every possible outcome, and (2) each condition in a decision has to be shown to independently affect the outcome of the decision [1]. For example, in order for a to independently affect the outcome of $(a\ and\ b)$, b has to be $true$, otherwise, $(a\ and\ b)$ would evaluate to $false$ no matter what value a has. In this paper, we use the masking form of MC/DC [13].

It has been shown that structural coverage metrics are sensitive to program structures [3]. Simple syntactic transformations can have a dramatic effect on the generated test suite as well as its fault finding effectiveness. Furthermore, even if the faulty code is exercised, the corrupted program state may not be propagated to observable outputs. Observability-based coverage metrics have been proposed to address these problems [14], [5]. Observable modified condition/decision coverage (OMC/DC) is an improved criterion over MC/DC, in which a tag is associated with each condition and the propagation of tags is used to approximate observability.

TABLE I
ENHANCED TAGGING SEMANTICS

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

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

Although OMC/DC outperforms MC/DC in terms of fault finding effectiveness and sensitivity to program structure, it suffers from scalability issues. This is not surprising – with the notion of observability, the generated test case is much longer than the ones generated to satisfy MC/DC, and the time cost of generation increases exponentially. Furthermore, the original tagging semantics has optimistic inaccuracy. In the following code fragment, the condition c is reported to be observable, but c cannot affect the outcome of this code fragment.

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

In this paper, we extended the tagging semantics defined by Whalen et al. by removing the optimistic inaccuracy in *if-then-else* expressions. The extended tagging semantics is shown in Table I. The rules with gray backgrounds are enhancements of the original definitions, where $=_v$ represents value equality of two expressions.

A. The Dataflow Language Lustre

Dataflow languages, which assign values to a set of equations in response to periodic inputs, are popular in model-based development using tools such as Simulink and SCADE. The dataflow language we are using – Lustre – is a synchronous dataflow language for programming reactive systems [15]. It is typically used as an intermediate representation between behavioral models and source code. Lustre programs can be automatically translated from models in notations such as Simulink, and can be automatically translated further to implementations in languages such as C/C++ and also as input models to verification tools such as model checkers.

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

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

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

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

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

TABLE II
LUSTRE PROGRAM EVALUATION

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

B. Dynamic Symbolic Execution

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

Dynamic symbolic execution provides improved scalability over classic symbolic execution by combining concrete and symbolic executions. By using concrete values, dynamic symbolic execution is able to simplify constraints, which helps it generate test inputs for execution paths that classic symbolic execution is infeasible to analyze.

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

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

III. INCREMENTAL TEST GENERATION

We define *observation points* as the set of variables where (internal) program state can be observed, i.e., an effect propagated to any of the observation points is *observable*. In an observability-based coverage criterion, observation points are usually the set of output variables. We use $\mathcal{O}$ to represent *output observation points*. In our incremental test generation approach, we also define *delayed observation points* (represented by $\mathcal{D}$) as the set of delay variables, i.e., an effect propagated to delay variables is stored and may be further propagated. Therefore, when a tag is propagated to some output observation point, we have a complete test. Alternatively, when a tag is propagated to some delayed observation point, it will be further propagated in the next step.

A. Non-Masking Path Conditions

A condition/variable is *observable* if it is not masked along some path as described in the tagging semantics in Table I. We call the path a *non-masking path* and the constraints for the program to take the path as *non-masking path conditions*. An *immediate* non-masking path is a dataflow path from a variable to an observation point that is entirely within one computational step (i.e., no delays) where the value of the variable is not masked. Immediate non-masking paths can be defined inductively by examining define-use relationships among variables. In our incremental test generation approach, all non-masking paths are immediate. Therefore, we use the term *non-masking paths* in the remaining of the paper.

Suppose y is one of the variables that uses x, then tags associated with x can be propagated if x is not masked in the definition of y (i.e., x affects y) and y is observable. We track these relationships by instrumenting the original program with additional path condition variables. Specifically, we use x_AFFECT_y to indicate the path condition that x is not masked in the definition of y (i.e., tags with x can be propagated to y). We use $var\_observed$ to represent the constraint of a non-masking path that the variable var can be propagated to (one or more) observation points. We also create additional input variables $var\_observable$ for all delay variables to represent whether a delay variable var is one of the observation points. Output variables are always observable.

Suppose we have the same code fragment from Section II, we can then generate the following non-masking path conditions.

```
in2_AFFECT_v2 = v1;
v1_AFFECT_v2 = in2;

in3_AFFECT_v4 = (v2 <> v3);
v2_AFFECT_v4 = in3;
v3_AFFECT_v4 = (not in3);

in1_observed = in1_observable;
in2_observed = (in2_AFFECT_v2 and v2_observed);
in3_observed = (in3_AFFECT_v4 and v4_observed);
v1_observed = (v1_AFFECT_v2 and v2_observed);
v2_observed = ((v2_AFFECT_v4 and v4_observed)
              or v2_observable);
v3_observed = (v3_AFFECT_v4 and v4_observed);
v4_observed = v4_observable;
out_observed = true;
```

In Whalen et al.'s work, tags associated with the condition $in3$ can always be propagated to the variable $v4$, which creates optimistic inaccuracy: when $v2 = v3$, $in3$ does not play any role in determining the value of $v4$. In the present work we remove this inaccuracy by strengthening the path condition for tag propagation in such cases. In the above example, in order to propagate tags from $in3$ to $v4$, we also require the inequality of $v2$ and $v3$. More precisely, we require the value inequality of expressions from two branches.

The variable $v2$ is used in two equations and therefore has two non-masking paths to observability: one through $v4$ and the other through itself. Each of the other variables is used once and thus has one non-masking path.

B. Test Generation

We then define *propagation* as a structure that contains the following fields:

1) tag labels a condition and its Boolean value.
2) $state$ represents the current program state.
3) $locations$ represents the set of variables that tag has propagated to. Initially, for example, tag_c is at the location of condition c.
4) $visited$ represents the set of visited observation points.
5) $test$ represents an incrementally generated test case.

Algorithm 1 shows the main algorithm of our incremental test generation approach. The MAIN procedure takes a condition c and returns a test case that can propagate the effect of c to some output observation points, or *UNSAT*. During each iteration of the *while* loop, the algorithm first attempts to generate a test that can propagate tag_c to $\mathcal{O}$ (line 7 – 10). If *SAT*, the newly generated test is concatenated with existing ones and we have a complete test. Otherwise, the algorithm attempts to generate a test that can propagate tag_c to $\mathcal{D}$ (line 11 – 14). If *UNSAT*, there is no feasible path that can propagate tag_c further and thus *UNSAT* is returned. Otherwise, tag_c is propagated to delayed observation points and will be further propagated in the next iteration.

Algorithm 1 Incremental Test Generation

```
1:  O ← output observation points
2:  D ← delayed observation points
3:
4:  procedure MAIN(c)
5:      p ← (tag_c, ∅, {c}, ∅, ∅)
6:      while true do
7:          test ← GENERATE(p, O)
8:          if test = SAT then
9:              p.test ← p.test ++ test                ▷ Concatenate
10:             return p.test
11:         test ← GENERATE(p, D)
12:         if test = UNSAT then
13:             return UNSAT
14:         p ← PROPAGATE(p, test)
```

Algorithm 2 shows the sub-procedure GENERATE. The GENERATE procedure takes a propagation structure p and a given set of variables as observation points op, and returns a test that can propagate $p.tag$ to any variable(s) in op, or *UNSAT*. The *path constraint* is first constructed by a disjunction of path conditions from the set of variables in $p.locations$ (line 2 – 5). The disjunction ensures that the tag can be propagated when *any* of the variables can be observable. Then the constraint on observation points is added (line 6 – 10). For each delay variable, if it is in the given set of observation points and not visited, it will be a candidate observation point that $p.tag$ can propagate to. Finally, the program state $p.state$ is added such that the generation starts from a concrete state (line 11).

Algorithm 2 Test Generation

```
1:  procedure GENERATE(p, op)
2:      pc ← false                                     ▷ Path constraint
3:      for each var in p.locations do
4:          pc ← pc ∨ var_observed
5:      pc ← (pc)
6:      for each var in D do
7:          if var ∈ (op \ p.visited) then
8:              pc ← pc ∧ (var_observable)
9:          else
10:             pc ← pc ∧ (not var_observable)
11:     pc ← pc ∧ p.state
12:     return SOLVE(pc)
```

Essentially, in an observability-based coverage for dataflow languages, tags are always located at delay variables, unless they are propagated to output variables. In the incremental test generation approach, however, it is possible that a tag is repeatedly propagated to the same set of delay variables, forming a cyclic propagation. Cyclic propagation is a result

of concretizing program state and generating new tests locally. From a concrete program state, test generation (and thus tag propagation) is deterministic. The locally optimal path found by the model checker may be in a subspace that will repeatedly propagate the tag inside the subspace – the particular path chosen need not necessarily be the best candidate for eventually reaching an output – making test generation not terminate.

In our present approach, we favor *incrementality*. To account for the problem of cyclic propagation, we include a simple heuristic (i.e., recording all visited variables in $p.visited$ and not propagating $p.tag$ to them) that avoids repeatedly visiting the same delayed observation point. This heuristic forces the test case to take a different path – which intuitively would increase fault finding effectiveness – and guarantees that the generation process will terminate. Otherwise, as in the traditional generation, a model checker typically generates the shortest possible path that satisfies an obligation. This may not be desirable from a testing perspective: for it may lead to fewer program states being visited which can negatively impact fault finding [6]. Given all the benefits, however, the heuristic may potentially miss some feasible test cases that necessarily require passing through cyclic propagation to reach some observable output. These test cases could have been found (if feasible) by attempting to generate a complete test case to satisfy the OMC/DC obligation.

Algorithm 3 shows the sub-procedure PROPAGATE. The PROPAGATE procedure takes a propagation structure p and a newly generated $test$, and returns the updated p.

Algorithm 3 Tag Propagation

```
1:  procedure PROPAGATE(p, test)
2:      p.state ← EXECUTE(p.state, test)
3:      locations ← ∅
4:      while p.locations ≠ ∅ do
5:          Select and remove var from p.locations
6:          useSet ← DFG retrieval variables that use var
7:          for each use in useSet do
8:              if var_AFFECT_use then
9:                  if use ∈ (D \ p.visited) then
10:                     Add use to locations
11:                 else
12:                     Add use to p.locations
13:     p.locations ← locations
14:     p.visited ← p.visited ∪ p.locations
15:     p.test ← p.test ++ test                    ▷ Concatenate
16:     return p
```

The newly generated test is first executed from the current program state to obtain an updated program state (line 2). Then $p.tag$ is propagated from current $p.locations$ to new $locations$ (line 3 – 13). Specifically, we first select and remove a location var from $p.locations$ and retrieve the set of variables $useSet$ that use var from the dataflow graph (line 5 – 6). For each variable use in $useSet$, the path condition var_AFFECT_use is checked (line 8). If var affects use, $p.tag$ is propagated from var to use. Then if use is an unvisited observation point, it is added to new $locations$; otherwise, it is added to $p.locations$ for further propagation. Eventually, new locations are recorded as visited (line 14) and the new test is concatenated with existing ones (line 15).

C. Test Generation Example

Suppose our goal is to cover the MC/DC obligation of $in2$ with $true$ value (as MC/DC requires, the condition $in2$ has to evaluate to both $true$ and $false$ values, and be shown to independently affect the outcome of the decision) and propagate its effect not only to $v2$ (as MC/DC requires), but also to out (as OMC/DC requires). In the following illustration, the program state is not expanded in order to simplify the representation. We will only show feasible path constraints and generated tests, as well as the propagation structure before and after each iteration.

1) Iteration 1: Propagate tag_{in2_true} from $\{in2\}$.

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

Before: p = (tag_{in2_true}, $state$, $\{in2\}$, $\emptyset$, $\emptyset$)
pc = ($in2$ and $in2\_AFFECT\_v2$ and $v2\_observed$)
and
($v2\_observable$ and $v4\_observable$)
After: p = (tag_{in2_true}, $state$, $\{v2\}$, $\{v2\}$,
{(T, F, F), (F, T, F)})

2) Iteration 2: Propagate tag_{in2_true} from $\{v3\}$.

At the second iteration, tag_{in2_true} automatically propagates from $v2$ to $v3$, since $v3$ uses the delay variable $v2$. Propagating tag_{in2_true} to output observation points is still infeasible, so it will be propagated to $v4$ as shown in the following. Note that since $v2$ has been visited, it is shown as *not observable* in the path constraint.

Before: p = (tag_{in2_true}, $state$, $\{v3\}$, $\{v2\}$,
{(T, F, F), (F, T, F)})
pc = ($v3\_observed$)
and
(not $v2\_observable$ and $v4\_observable$)
and
($p.state$)
After: p = (tag_{in2_true}, $state$, $\{v4\}$, $\{v2, v4\}$,
{(T, F, F), (F, T, F), (F, F, F)})

3) Iteration 3: Propagate tag_{in2_true} from $\{out\}$.

At the third iteration, tag_{in2_true} automatically propagates from $v4$ to out, since out uses the delay variable $v4$. Propagating tag_{in2_true} to output observation points is feasible, so a complete test is generated and the incremental test generation algorithm terminates. Note that since both $v2$ and $v4$ have been visited, they are shown as *not observable* in the path constraint.

Before: p = (tag_{in2_true}, $state$, $\{out\}$, $\{v2, v4\}$,
{(T, F, F), (F, T, F), (F, F, F)})
pc = ($out\_observed$)
and
(not $v2\_observable$ and not $v4\_observable$)
and
($p.state$)
After: p = (tag_{in2_true}, $state$, $\{out\}$, $\{v2, v4, out\}$,
{(T, F, F), (F, T, F), (F, F, F), (F, F, F)})

Table III shows a summary of the generated test case as well as the locations of tag_{in2_true} at each step.

TABLE III
TEST CASE EXAMPLE

	Locations of tag_{in2_true}	in1	in2	in3
Step 1	$in2$	$True$	$False$	$False$
Step 2	$v2$	$False$	$True$	$False$
Step 3	$v4$	$False$	$False$	$False$
Step 4	out	$False$	$False$	$False$

IV. EVALUATION

The quality in terms of fault finding of the test suites generated to satisfy OMC/DC and MC/DC has been evaluated [5]. In this paper, we are interested in comparing the two approaches – incremental and regular test generation satisfying OMC/DC – in terms of efficiency and effectiveness. Therefore, we have the following research questions:

1) Is incremental test generation more efficient than regular test generation? What is the percentage of time needed to generate a test suite in the incremental approach compared with regular test generation?
2) What is the generated test suite size (i.e., the number of satisfied obligations) and how does it affect test suite effectiveness in the two approaches?
3) How well does the test suite generated by the incremental approach perform in terms of fault finding effectiveness compared with regular test generation?

A. Case Example Systems

In this study, we used four industrial systems (i.e., *DWM1*, *DWM2*, *Vertmax*, and *Latctl*) developed by Rockwell Collins Inc., two subsystems (i.e., *Alarm* and *Infusion Manager*) of a Generic Patient Controlled Analgesia (GPCA) infusion system [16], and a last system (i.e., *Docking Approach*) created as a case example at NASA.

The Rockwell Collins systems were modeled using Simulink [17]. Two of the systems, *DWM1* and *DWM2*, represent distinct portions of a Display Window Manager for a commercial display system. The other two systems, *Vertmax* and *Latctl*, represent the vertical and lateral mode logic for a Flight Guidance System.

The remaining three systems are modeled using Stateflow [18]. Two of the systems, *Alarm* and *Infusion Manager*, represent the alarm-induced behavior and the prescription management of an infusion pump device. The NASA system, *Docking Approach*, describes the behavior of a space shuttle as it docks with the International Space Station.

All the systems were translated to the Lustre programming language [15] to take advantages of existing automation. Lustre code generation from Simulink/Stateflow models is analogous to the automated code generation offered by Mathworks Simulink Coder [19]. In practice, Lustre programs will be automatically translated to C code. Therefore, results of this study would be identical, if applied to their C implementations.

Table IV shows basic information of the systems measured on the original Simulink and Stateflow models.

TABLE IV
SIMULINK AND STATEFLOW CASE EXAMPLE INFORMATION

(a) Simulink models

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

(b) Stateflow models

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

B. Test Suite Generation

We used the counterexample-based approach to generate test cases satisfying obligations/path conditions [20], [21], specifically, we used the JKind model checker [22] in our experiments. This approach is guaranteed to generate a test suite that achieves the maximum possible coverage of the system under test. Note that, however, in regular test generation, this is guaranteed as long as the model checker is given enough computing resource to terminate; in incremental test generation, the maximum propagation is only guaranteed at each step, which can have both advantages and disadvantages and will be discussed in detail in Section V.

To account for variance in the generation time, we generated 10 test suites for each of the two approaches (i.e., incremental and regular test generation) for each of the seven systems under test. We measured time using Linux *time* utility. Specifically, we measured *user time* in order to account for the variance of parallel computing inside the model checker in a multi-core environment.

In this study, we used the full generated test suite to perform fault finding analysis, even if there can be redundancy in the generated test cases [5], [3]. There are two reasons for this setting: (1) when we compare the generation time of a test suite, the comparison is based on the full test suite rather than a reduced test suite; (2) it is debatable whether test suite reduction will decrease fault finding effectiveness [23], [24], and the choice of reduction algorithms may also have an effect on fault finding and reduced test suite size. Performing coverage measurement and test suite reduction have a cost (e.g., at least we need to execute all tests to measure how they satisfy obligations) and the cost of measuring coverage of the full test suite would be much more than just executing the test suite on the original program. Because of the above reasons, we decided to use the full generated test suite in this study for the purpose of fault finding analysis.

C. Mutant Generation

Mutation testing has been shown to be an adequate proxy for real faults for the purpose of investigating fault finding effectiveness [25]. In our experiment, mutant generation is done by automatically introducing a single fault into the correct implementation. The mutation operators used in this study are typical and were discussed in detail in our previous paper [26]. The generated set of mutants has roughly the

same distribution of fault types across the system occurring naturally.

For each of our systems, we created 750 mutants[1] and randomly grouped them into 10 sets, each of which contains 75 mutants. We removed mutants that cause compile-time or run-time errors (e.g., divide-by-zero), but we left possibly equivalent mutants. Although checking and removing equivalent mutants can be feasible on small systems [5], the cost of checking equivalent mutants for our last three systems is prohibitive. Therefore, we did not attempt to remove equivalent mutants in our experiments in order to keep our experimental configuration consistent across different case example systems.

D. Test Oracles

For the purpose of this study, we used output-only expected value oracles – the ones people would most likely use in practice [27]. When using an output-only expected value test oracle, for each test input, concrete values are specified that the system is expected to produce for output variables.

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

To determine the fault finding effectiveness of the generated test suites, we simply compared output values produced by a faulty version against expected output values produced by the original version. The fault finding effectiveness of a test suite is computed as the percentage of mutants killed.

Although we generated 10 test suites for each test generation approach to account for the variance in generation time, for the purpose of evaluating fault finding effectiveness, we used one of them, since both the incremental and regular test generation approaches are deterministic in producing test cases.

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

V. Results & Discussion

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

A. RQ1: Test Generation Time

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

[1]750 is slightly smaller than the maximum number of possible mutants in our smallest case example system Latctl.

TABLE V
Test generation time (in seconds) for each system, average over 10 test suites

System	Incremental Generation	Regular Generation	Time Ratio	p-value
DWM1	141.2	107.4	131.5%	< 0.01
DWM2	30.4	39.8	76.4%	< 0.01
Vertmax	119.8	279.8	42.8%	< 0.01
Latctl	19.1	28.4	67.3%	< 0.01
Alarm	363.3	2958.9	12.3%	< 0.01
Infusion	278.5	3623.6	7.7%	< 0.01
Docking	40413.2	176226.3*	22.9%	< 0.01

Table V shows the average test generation time for each case example system and each test generation approach. The *Time Ratio* column is the percentage of time needed to incrementally generate a test suite as opposed to the regular generation of the suite. *p-value* is computed using a two-sided permutation test using *mean* as the test statistic, under the null hypothesis that test generation time from incremental and regular approaches are drawn from the same distribution. Note that regular test generation is infeasible on *Docking*. We set a timeout of 48 hours for the model checker to terminate and measured user time (as starred in Table V). It is both time and memory prohibitive to obtain the actual time from regular test generation on *Docking*. Figure 1 further illustrates the distribution of test generation time for each system.

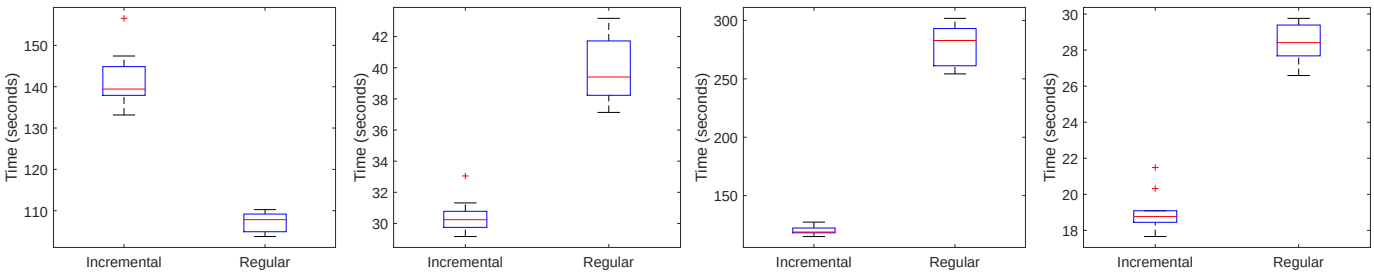
The incremental approach is significantly more efficient than regular test generation for all systems except *DWM1*. We investigated this outlier and found that *DWM1* is purely combinatorial, i.e., there is no state maintained in the system and thus no delay operations. Therefore, all computation is within one step and finding an immediate non-masking path is generally cheap since the tag propagation is trivial, while the incremental approach adds more overhead to produce path constraints and track tag propagations. On all other systems, the incremental approach requires much less time as opposed to regular test generation with a trend that the larger/more complex the system is, the more is the improvement achieved. For smaller systems, the overhead associated with the incremental approach dominates the generation time. Additionally, our approach calls the model checker multiple times and JVM overhead can be significant on small systems with an execution time of tens of seconds.

The observed improvement matches our expectation. As we generate new test inputs at each step, we concretize the inputs we have already computed and solve a much simpler path condition to further propagate tags based on the execution results of existing test inputs.

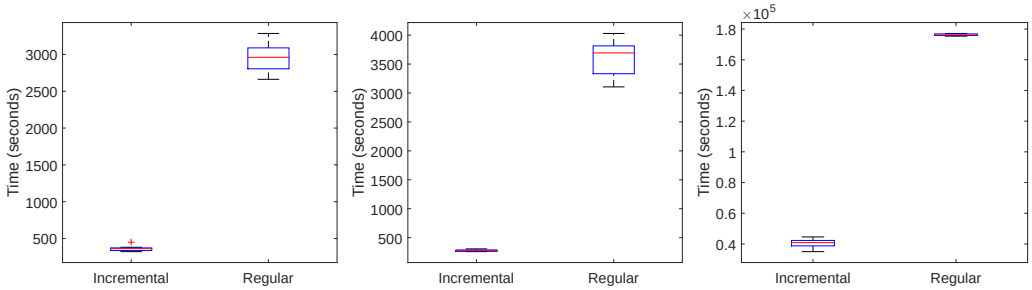
B. RQ2: Test Suite Size

Since we used the full test suite generated from each approach, the test suite size indicates the number of satisfied obligations. Table VI shows the total number of obligations and generated test suite sizes for each approach for each case example system.

Given the enhanced tagging semantics used in our incremental approach, the test suites generated incrementally will be of equal or smaller size than test suites generated regularly,



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



(e) Alarm (f) Infusion (g) Docking

Fig. 1. Test generation time

TABLE VI
TEST SUITE SIZE FOR EACH SYSTEM

System	Total Obligations	Incremental Generation	Regular Generation
DWM1	2038	1833	2036
DWM2	530	511	511
Vertmax	1732	1705	1728
Latctl	380	371	371
Alarm	1406	991	1098
Infusion	1666	719	934
Docking	4900	961	1954

since more obligations would become unsatisfiable and tests that in fact cannot propagate tags (i.e., optimistic inaccuracy in the original OMC/DC) would be eliminated. In addition, since the incremental approach concretizes all input values at each step, it may fail to propagate tags in later steps, because future paths may have been rendered infeasible based on the previous concretization. As a result, incrementally generated test suites are expected to have fewer tests than test suites generated regularly.

Previous investigations on test suite size and fault finding effectiveness have shown that there is a moderate to high correlation between them [28], [29]. In this study, the incrementally generated test suite is, however, smaller than the test suite generated regularly. This could be a concern since we do not wish to achieve better efficiency by losing fault finding effectiveness. This leads to our third research question on comparing fault finding effectiveness between incremental and regular test generation approaches.

C. RQ3: Fault Finding Effectiveness

In the work presented in this paper we have a stronger tagging semantics than the one defined by Whalen et al. [5]. Given this more restrictive propagation requirement, it is harder (or impossible) to satisfy propagation obligations

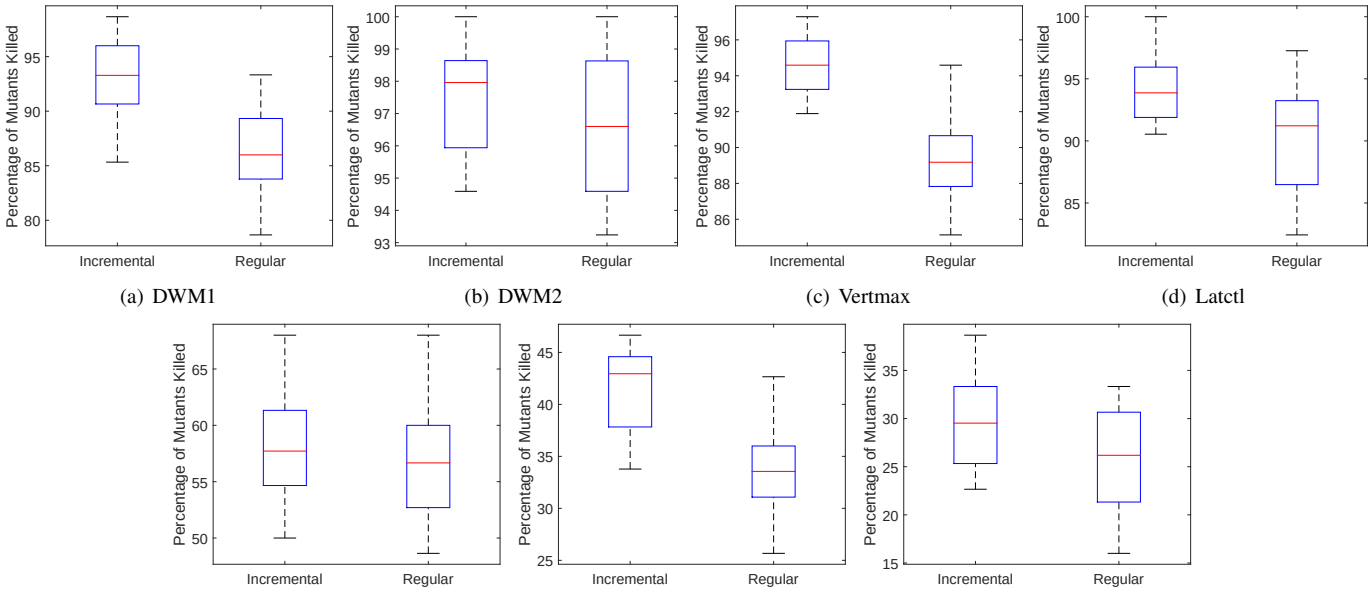
TABLE VII
PERCENTAGE OF MUTANTS KILLED FOR EACH SYSTEM, AVERAGE OVER 10 SETS OF MUTANTS

System	Incremental Generation	Regular Generation	% Change	p-value
DWM1	93.1%	86.1%	7.0%	< 0.01
DWM2	97.5%	96.6%	0.9%	0.26
Vertmax	94.3%	89.3%	5.0%	< 0.01
Latctl	94.4%	89.9%	4.5%	0.02
Alarm	58.0%	56.5%	1.5%	0.56
Infusion	41.3%	33.7%	7.6%	< 0.01
Docking	29.8%	25.7%	4.1%	0.14

that would have been satisfiable. Consequently, we run the risk of generating fewer test cases and potentially having worse fault finding effectiveness. On the other hand, stronger path conditions can lead to better tests – with guaranteed propagation – that may increase fault finding.

Table VII shows the average percentage of mutants killed for each case example system and each test generation approach. The *% Change* column indicates the improvement in fault finding effectiveness from generating test suites incrementally compared with generating test suites regularly. *p-value* is computed using a two-sided permutation test using *mean* as the test statistic, under the null hypothesis that fault finding effectiveness from incremental and regular approaches are drawn from the same distribution. Figure 2 further illustrates the distribution of the percentage of mutants killed for each system.

Fault finding effectiveness of incrementally generated tests is for our case example systems universally better than (or equal to) regularly generated tests. Several reasons contribute to this result. Given the stronger tag propagation requirement, the optimistic inaccuracy presented in the original definition of OMC/DC is eliminated. Furthermore, JKind is an SMT-based model checker and will always generate the shortest



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

(e) Alarm (f) Infusion (g) Docking

Fig. 2. Fault finding effectiveness

test cases possible to satisfy an obligation or a path condition. In our experience, such tests technically satisfy the coverage criterion, but they are (1) susceptible to the masking effect discussed earlier and (2) likely to only vary the few input variables needed to cover the obligation and leave all other input variables at the model checker's default values (typically zero and false) [6]. Both issues are mitigated in our incremental approach. First, the incremental search for a propagation path allows the possibility to pursue a path longer than the shortest propagation path. Second, since we are restarting the search for a path in each incremental step, there is an opportunity for more variables to change values (i.e., we are no longer searching for the shortest path with the fewest variable changes). From our experimental results, we see that a combination of these two aspects of the incremental approach leads to longer and more diverse tests with better fault finding effectiveness.

On a related note, as introduced in Section IV, we did not remove equivalent mutants in this study in order to keep our experimental configuration consistent across different case example systems. As a result, fault finding effectiveness for the first four systems would be in general slightly lower than our previous results [5].

VI. Threats to Validity

External Validity: We have used seven synchronous critical systems of different sizes from different domains (i.e., industrial avionics systems and medical device systems). We believe that these system are representative of avionics and medical device domains. Thus, our results are generalizable to other systems in these domains.

We have used the dataflow language Lustre [15] as the implementation language. Although Lustre is a less common language than C/C++ or Java, these Lustre programs are similar in structure to systems written in C/C++ in these domains. In practice, Lustre programs will be automatically translated to C code. Therefore, we believe that our results are applicable to systems written in more traditional imperative languages.

We have generated approximately 750 mutants for each case example system. These mutants were further divided into 10 sets. The total number of mutants and the number of mutants in each set were chosen to provide enough data points and yield a reasonable cost for evaluating fault finding effectiveness. Based on past experience, results using these numbers of mutants are representative [27]. The mutants were generated using the same tool and configuration as in previous work [5], in which OMC/DC was shown to achieve much better fault finding effectiveness than MC/DC.

Internal Validity: We have used the model checker JKind [22] to generate test cases. The counterexample-based approach provides the shortest test cases possible to satisfy an obligation or a path condition. These automatically generated test cases may behave differently from test cases obtained by other means [6], and thus, it is possible that test cases derived by hand or random generation, may provide different results.

Construct Validity: The fault finding effectiveness is measured over mutants, i.e., 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. showed that the use of mutants leads to conclusions similar to those obtained using real faults [25].

VII. Related Work

A. Observability-based Coverage Criteria

The study of observability is not new in testing of hardware logic circuits [4]. Observability-based code coverage metric (OCCOM) is a criterion in which tags are attached to internal

program states and the propagation of tags is used to predict the actual propagation of errors [14]. A variable is tagged when there is a change in the value of the variable due to an error. The observability-based coverage can be used to determine whether erroneous effects that are triggered by the inputs can be observed at the outputs.

Observable modified condition/decision coverage was recently defined as an improvement over MC/DC [5]. OMC/DC also uses a counterexample-based test generation approach and is complete in theory. That is, it is guaranteed to find non-masking paths if they exist, as long as enough time and memory are given to the model checker. In practice, however, such test generation can be infeasible on large systems and thus lose completeness. On the other hand, OMC/DC has optimistic inaccuracy in certain program structures (i.e., if-then-else statements). As a result, it may generate tests to propagate a condition to outputs, but the condition is actually not observable, leading to wasted testing effort.

Both OCCOM and OMC/DC are observability-based coverage metrics, as well as using tag propagation to approximate observability. But both approaches can be infeasible on large systems when generating tests.

In this study, we extended the tagging semantics defined by Whalen et al. [5] to generate path conditions. The differences between our work and the original OMC/DC are: (1) Our approach is a test generation approach rather than a test adequacy measurement approach defining new coverage metrics. (2) We removed optimistic inaccuracy in the original definition of OMC/DC by requiring value inequality of expressions from two branches when propagating if conditions. This added constraint is feasible due to our improvement over efficiency. (3) Our approach will explicitly terminate when there is no feasible paths, while for the regular test generation, a timeout is usually estimated and manually set in order to terminate the generation process.

A similar tagging approach, dynamic taint analysis [30], has been used in security as well as software testing and debugging. However, taint analysis is usually defined over define-use relations and masking is generally not considered. Thus, taint analysis is far more optimistically inaccurate than an observability-based approach.

B. Dynamic Symbolic Execution

The notion of dynamic symbolic execution has been used in many applications such as CUTE and jCUTE [8], [31], DART [7], KLEE [9], and Pathcrawler [10], as well as combined with fuzz testing to detect exploitable security issues [12].

Concolic testing [8] is an approach that incrementally generates test inputs by combining concrete and symbolic executions. Specifically, it starts with a random input, symbolic constraints are generated in the execution of a concrete input. These constraints are then modified and solved to force the program to take different (but feasible) execution paths. Feedback-directed random test generation [32] is another approach that also builds test inputs incrementally. That is, as soon as a new test input is created, it is executed. The execution results are used to guide further generation. New test inputs would extend previous ones, such that it can avoid redundant or illegal test inputs and towards the execution of new program states.

We refer the reader to recent surveys [33], [34] for more details on symbolic execution systems.

Most of these approaches, however, are applied for a program to take more and different paths, as well as to cover simple coverage metrics such as branch coverage. Our incremental test generation approach, together with the extended tagging semantics for observability, explicitly propagates faulty program state to observable outputs, and thus, fault finding effectiveness can be improved dramatically.

C. Test Concatenation

Concatenating test cases have been investigated in dataflow languages [35], [36]. In a common approach to generating tests satisfying a coverage metric, each obligation is solved at the initial program state. In Hamon et al.'s approach [35], after creating a test case for an obligation, the final state of the test case serves as the initial state for the next test case. That is, the next test case can be interpreted as an extension of the previous test case. Fraser et al. [36] further investigated how test case length affects fault finding effectiveness by concatenating test cases to create test suites with different test case length, while achieving similar coverage.

VIII. Conclusions

In this paper, we described an incremental test generation approach that combines the notion of observability and dynamic symbolic execution. This approach is proposed to address optimistic inaccuracy and feasibility issues in regular counterexample-based test generation satisfying observability-based coverage criteria. Our empirical study on seven case example systems from avionics and medical device domains indicates that compared with traditional test generation approach: (1) the incremental approach is far more efficient and feasible on large systems; (2) the generated test suite size is generally smaller; (3) fault finding effectiveness of test suites generated by the incremental approach is generally better.

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

(1) Reducing pessimistic inaccuracy. An obligation that is satisfiable may not have a feasible path to outputs by extending certain concrete test inputs. To address this issue, we may leave certain variables in previous tests symbolic and have a backtracking mechanism, which also have to be carefully tuned, since having symbolic values will increase overhead and frequent backtracking may also slow down incremental test generation.

(2) Comparison to other coverage metrics. We have specifically compared incremental and regular test generation satisfying OMC/DC. It would be interesting to see how OMC/DC tests perform against tests generated from other means, such as test suites satisfying requirements-based coverage, which may also address the propagation from inputs to outputs.

REFERENCES

[1] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

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

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

[4] 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*, 1997, pp. 418–425.

[5] M. Whalen, G. Gay, D. You, M. P. Heimdahl, and M. Staats, "Observable modified condition/decision coverage," in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 102–111.

[6] G. Gay, M. Staats, M. Whalen, and M. Heimdahl, "The risks of coverage-directed test case generation," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 803–819, 2015.

[7] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.

[8] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th International Symposium on Foundations of Software Engineering*, 2005, pp. 263–272.

[9] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, vol. 8, 2008, pp. 209–224.

[10] N. Williams, B. Marre, P. Mouy, and M. Roger, "Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis," in *Proceedings of the 5th European Dependable Computing Conference*, 2005, pp. 281–292.

[11] N. Tillmann and J. De Halleux, "Pex–white box test generation for .NET," in *Tests and Proofs*, 2008, pp. 134–153.

[12] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *Proceedings of the Network and Distributed System Security Symposium*, vol. 8, 2008, pp. 151–166.

[13] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, *A practical tutorial on modified condition/decision coverage*. NASA Langley Research Center, 2001.

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

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[16] A. Murugesan, S. Rayadurgam, and M. P. Heimdahl, "Modes, features, and state-based modeling for clarity and flexibility," in *Proceedings of the 5th International Workshop on Modeling in Software Engineering*, 2013, pp. 13–17.

[17] "MathWorks Simulink," http://www.mathworks.com/products/simulink, August 2015.

[18] "MathWorks Stateflow," http://www.mathworks.com/products/stateflow, August 2015.

[19] "MathWorks Matlab Coder," http://www.mathworks.com/products/matlab-coder, August 2015.

[20] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," in *Proceedings of 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1999, pp. 146–162.

[21] S. Rayadurgam and M. P. E. Heimdahl, "Coverage based test-case generation using model checkers," in *Proceedings of the Eighth Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, 2001, pp. 83–91.

[22] "JKind," https://github.com/agacek/jkind, August 2015.

[23] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Proceedings of the 17th International Conference on Software Engineering*, 1995, pp. 41–41.

[24] G. Rothermel, M. J. Harrold, J. Von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.

[25] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[26] A. Rajan, M. Whalen, M. Staats, and M. P. Heimdahl, "Requirements coverage as an adequacy measure for conformance testing," in *Formal Methods and Software Engineering*, 2008, pp. 86–104.

[27] G. Gay, M. Staats, M. Whalen, and M. Heimdahl, "Automated oracle data selection support," *IEEE Transactions on Software Engineering*, 2015.

[28] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness." in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 435–445.

[29] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proceedings of the eighteenth International Symposium on Software Testing and Analysis*, 2009, pp. 57–68.

[30] 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*, 2007, pp. 196–206.

[31] K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," in *Computer Aided Verification*, 2006, pp. 419–423.

[32] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 75–84.

[33] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 4, pp. 339–353, 2009.

[34] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[35] G. Hamon, L. De Moura, and J. Rushby, "Generating efficient test sets with a model checker," in *Proceedings of the Second International Conference on Software Engineering and Formal Methods*, 2004, pp. 261–270.

[36] G. Fraser and A. Gargantini, "Experiments on the test case length in specification based test case generation," in *Proceedings of the 2009 ICSE Workshop on Automation of Software Test*, 2009, pp. 18–26.