A Flexible and Non-intrusive Approach for Computing Complex Structural Coverage Metrics

Michael W. Whalen*, Suzette Person†, Neha Rungta‡, Matt Staats§ and Daniela Grijincu¶
*University of Minnesota, Minneapolis, Minnesota, USA Email: whalen@cs.umn.edu
†NASA Langley Research Center, Hampton, Virginia, USA Email: suzette.person@nasa.gov
‡NASA Ames Research Center, Moffett Field, California, USA Email: neha.s.rungta@nasa.gov
§Google Inc., Zurich, Switzerland Email: staatsm@gmail.com
¶University of St. Andrews, Scotland, UK Email: dana.grijincu@gmail.com

***Abstract*—Software analysis tools and techniques often leverage structural code coverage information to reason about the dynamic behavior of software. Existing techniques instrument the code with the required structural obligations and then monitor the execution of the compiled code to report coverage. Instrumentation based approaches often incur considerable runtime overhead for complex structural coverage metrics such as Modified Condition/Decision (MC/DC). Code instrumentation, in general, has to be approached with great care to ensure it does not modify the behavior of the original code. Furthermore, instrumented code cannot be used in conjunction with other analyses that reason about the structure and semantics of the code under test.**

In this work, we introduce a *non-intrusive* preprocessing approach for computing structural coverage information. It uses a static partial evaluation of the decisions in the source code and a source-to-bytecode mapping to generate the information necessary to efficiently track structural coverage metrics during execution. Our technique is *flexible*; the results of the preprocessing can be used by a variety of coverage-driven software analysis tasks, including automated analyses that are not possible for instrumented code. Experimental results in the context of symbolic execution show the efficiency and flexibility of our non-intrusive approach for computing code coverage information.

I. Introduction

Software analyses often leverage code coverage information to reason about the dynamic behavior of software. Coverage information describes structural elements in the source code, such as paths, functions, statements and branches that have been executed (covered). Coverage information is used to assess test adequacy [24], perform test case selection and prioritization [8], [25], and for test suite minimization [11], [26]. It has also been used for other software maintenance tasks such as predicting fault likelihood [9] and fault localization [14].

Although structural code metrics are defined at the source code level, they are measured during execution of the compiled code, e.g., Java bytecode. Existing state-of-the-art techniques compute code coverage by monitoring the execution of the instrumented bytecode to determine which coverage obligations are satisfied. Instrumentation involves insertion of "trace" statements at strategic locations in the bytecode depending on the coverage metric. Then, as bytecode instructions are executed, the instrumentation facilitates tracking the parts of the corresponding source code that are executed (covered).

Instrumentation based techniques can incur considerable overhead by adding a large number of conditional statements to the original code, which in turn, can significantly increase the program's execution space. The problem becomes even worse as the complexity of the coverage criterion increases, both in terms of the number of coverage obligations and the number of operators in each obligation. For metrics such as Modified Condition/Decision Coverage (MC/DC) – a metric widely used in the avionics software domain – the overhead of the instrumentation can render the cost of computing the metric prohibitive for programs of even modest size.

Another drawback of code instrumentation is that it may interfere with other analyses, e.g., a change impact analysis, because the instrumentation adds behaviors to the execution space of the program. Without a clear delineation between the original program behaviors and those arising from instrumentation, it is not possible to combine complementary analyses. Finally, there are often considerable risks associated with code instrumentation and great care must be taken to ensure the instrumentation does not modify the behavior of the original code, e.g., due to extra evaluations of Boolean expressions with side effects.

In this work, we introduce a novel preprocessing approach based on static partial evaluation of the decisions in the source code and a source-to-bytecode mapping procedure, to generate the information necessary to efficiently track various structural coverage metrics during execution without the need for code instrumentation. Tracking of the coverage metrics is accomplished by using any tool capable of monitoring the instruction stream, such as an instrumented JVM or a software verification tool such as Java PathFinder [36]. In this work, we use the Symbolic PathFinder tool [22] to track the coverage metrics, compute the set of covered obligations, and generate test cases that cover the obligations.

Our approach offers substantial benefits for computing coverage metrics such as MC/DC, condition decision, multiple condition, weak mutation, and strong mutation by avoiding the drawbacks of code instrumentation described above. There are benefits of using our approach for metrics such as branch or statement that do not require the same degree of instrumentation as MC/DC, however, the relative improvement is not as dramatic. Our approach is *efficient* in that it does

not increase the amount of code that must be monitored in order to compute the coverage metrics. It is also *flexible*, enabling complementary automated program analyses which do not have a mechanism to distinguish between the original code and the instrumentation to leverage coverage information to compute novel coverage metrics, e.g., structural coverage metrics over code impacted by a set of changes. In this work, we describe how the preprocessed coverage information can be combined with change impact information computed by a software analysis framework, Directed Incremental Symbolic Execution (DiSE) [23], [28] to analyze evolving software.

To demonstrate the efficiency and flexibility of our approach, we perform an empirical evaluation focused on two analyses, standard symbolic execution with instrumentation-based coverage measurement [31] and DiSE [23], [28], as applied to the problem of generating test suites that satisfy MC/DC obligations. We also compute the overhead incurred when using our approach with standard symbolic execution relative to symbolic execution with no measurement. We selected the MC/DC coverage criterion because we and other researchers have found the measurement of MC/DC to be a serious bottleneck when applying these analyses.

The results of our study indicate that the application of our preprocessing approach results in significant speedups in test input generation speed, up to 4x, relative to an instrumentation based approach. Generally low overhead of roughly 8% relative to standard symbolic execution when applied to sufficiently large systems was also observed, and the applicability of the approach to DiSE was confirmed.

This paper makes the following contributions:

- We present a novel idea for leveraging preprocessed information to track coverage obligations during execution of the code under analysis to avoid the issues related to code instrumentation.
- We present algorithms to preprocess the coverage conditions and implement the preprocessing algorithms as an Eclipse plugin to automatically collect the information relevant to various structural coverage metrics.
- We present the algorithms for leveraging the preprocessed coverage information for two MC/DC coverage based applications built on Symbolic PathFinder [22]: (1) symbolic execution for test case generation, and (2) DiSE for regression analysis.
- We empirically evaluate our approach to show that the preprocessed coverage information enables (1) efficient coverage based analyses, and (2) novel analyses that were not be possible with instrumented code.

II. Background

In software testing, the need to determine the adequacy of test suites has motivated the development of several test coverage criteria [40]. One such class of criteria are structural coverage criteria, which measure test suite adequacy in terms of coverage over the structural elements of the system under test. In the domain of critical systems — particularly in avionics — demonstrating structural coverage is required for certification [27]. In recent years, there has been rapid progress in the creation of tools for automatic directed test generation for structural coverage criteria [20], [30], [31]; as well as tools promising to improve coverage and reduce the cost associated with test creation.

```
if (TestGen.needsTest("MCDC=55_120")) {
  TestGen.printCommentIf(a && (b || c),
    "MCDC=55_120");
  Verify.ignoreIf(true);
}
if (TestGen.needsTest("MCDC=56_120")) {
  TestGen.printCommentIf(b && a,
    "MCDC=56_120");
  Verify.ignoreIf(true);
}
if (TestGen.needsTest("MCDC=57_120")) {
  TestGen.printCommentIf(c && !(b) && a,
    "MCDC=57_120");
  Verify.ignoreIf(true);
}
if (TestGen.needsTest("MCDC=58_120")) {
  TestGen.printCommentIf(!a,
    "MCDC=58_120");
  Verify.ignoreIf(true);
}
if (TestGen.needsTest("MCDC=59_120")) {
  TestGen.printCommentIf(!b && !(c) && a,
    "MCDC=59_120");
  Verify.ignoreIf(true);
}
if (TestGen.needsTest("MCDC=60_120")) {
  TestGen.printCommentIf(!c && !(b) && a,
    "MCDC=60_120");
  Verify.ignoreIf(true);
}
if (a && (b || c)) {
  ...
}
```

Fig. 1: Instrumentation required for: a `and` (b `or` c)

A. Motivating Example

The standard mechanism for generating test suites and/or measuring the adequacy of test suites involves instrumenting the code and monitoring the instrumentation output during execution. However, for complex test metrics, the overhead of measurement can be significant. To instrument for MC/DC, for example, it is necessary to create two test obligations for every condition (basic Boolean expression). To illustrate, we use a small code snippet as a example:

```
if (a && (b || c)) { .. }
```

The instrumentation must track whether each test obligation, i.e., specification of a structural component relevant to the coverage metric, is satisfied. This leads to a substantial amount of instrumentation code; for example, the instrumentation in [31] generates the instrumented code in Figure 1 for the single line of code shown in the snippet above. Our goal in showing the instrumented code in Figure 1 is not for the reader to fully understand mechanics of the instrumentation, it is rather to illustrate how significant the instrumentation can be for a seemingly simple expression. For programs with significant

amounts of Boolean logic, the size of the instrumented code is often several times as large as the original code.

During execution of the instrumented program, the executed annotations record the coverage obligations that have been satisfied. The annotations, however, lead to additional branch points that are not in the original program; this slows down execution in a standard VM and further exacerbates the path explosion problem when applying analyses such as symbolic execution. Furthermore, code instrumentation obscures the original code. In Figure 1, the instrumentation code is the sequence of `if` statements preceding the code under analysis, and would be indistinguishable from the code under analysis without a priori knowledge of the naming conventions and other details regarding how the instrumentation tool modifies the source code.

B. Structural Coverage Metrics

While a wealth of different structural coverage metrics over source code have been proposed, only a handful are commonly used as adequacy criteria. For a test suite, the most common criteria are defined as follows:

Statement Coverage: requires that each statement within the program is executed at least once.

Branch Coverage: requires that each conditional branch within the program (e.g. 'if' statement) evaluates to both true and false at least once.

Decision Coverage: Decision coverage requires that each *Decision* evaluates to both true and false. We follow the RTCA DO178B/C definition [12], in which decision coverage requires that each stand-alone Boolean expression (*decision*) that is not an immediate child of another Boolean expression via a Boolean operator take on values true and false (e.g.: `X := A and B` would require that `A and B` take on both true/false values).

Modified Decision/Condition Coverage (MC/DC): MC/DC requires that (1) each point of entry and exit in the program has been invoked at least once, (2) each condition (a Boolean expression containing no Boolean operators) in a decision in the program has taken on all possible outcomes at least once, and (3) each condition has been shown to independently affect the decision's outcome

Each metric subsumes the metrics listed above it: any test suite satisfying MC/DC satisfies Decision coverage, a suite satisfying Decision coverage satisfies Branch coverage, etc.

In this paper we use the *masking* form of MC/DC [12] to determine the independence of conditions. In masking MC/DC, a basic condition is *masked* if changing its value cannot affect the outcome of a decision. To better illustrate the definition of masking MC/DC, consider the expression `A and B`. To show the independence of `B`, we fix the value of `A` to T and vary the value of `B` to see if the result of the condition also changes with the value of `B`; note that we need to fix the value of `A` otherwise varying `B` will not affect the outcome of the expression. Independence of `A` is shown in a similar manner. The top row left element of Table I shows the test suite required to satisfy MC/DC for the expression `A and B`. When we consider decisions with multiple Boolean operators, we must ensure that the test results for one operator are not masked by the behavior of other operators. For example, given `A and (B or C)` the tests for `B or C` will not affect the outcome of the decision if `A` is F. The top row right element of Table I shows a test suite that would satisfy MC/DC for the expression `A and (B or C)`.

TABLE I: Top row: test suites providing masking MC/DC coverage for $A\ and\ B$ and $A\ or\ (B\ and\ C)$. Bottom row: Short circuit MC/DC obligations for and, or.

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

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

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

A	B	$A\ or\ B$
T	$*$	T
F	T	T
F	F	F

In C/C++, Java, and most other imperative languages, Boolean expressions are evaluated using short circuit evaluation. In short circuit evaluation, the right-side subexpressions of `and` or `or` expressions are not evaluated if the expression is known to be F or T respectively, after evaluation of the left-side subexpression. In such cases, since the right side subexpression is not evaluated, the right-hand side value becomes a 'don't care' (denoted '*'), as shown in the bottom row of Table I. Since we are analyzing imperative code, we assume the short-circuit version of MC/DC in the remainder of the paper.

III. Preprocessing

In this section, we describe our preprocessing approach based on static partial evaluation of the decisions in the source code to enable efficient tracking of MC/DC coverage information at runtime. The preprocessing information gathered for MC/DC can also be used to measure coverage for the other coverage metrics from Section II, as described in Section III-D.

A. Example Demonstrating Independence and Masking

In Java, evaluation occurs in the left-to-right order of the constituent conditions, so determining the independence of a condition depends on the location of the condition within the decision. Conditions on the right side of Boolean operators may mask out the effect of left-side conditions but not vice-versa. On the other hand, conditions on the left side of Boolean operators may cause the evaluation of right side conditions to be skipped entirely (short-circuited). For example, consider the complex Boolean decision at the top of Figure 2. The decision tree shown below the decision contains the corresponding Java byte code and preprocessing annotations for each condition (described in Section III-C). If `a` evaluates to false, then the evaluation of `(b || c)` is skipped entirely. If `c` is evaluated, then it must be the case that `b` evaluated to false and that `a`

evaluated to true. If `c` evaluates to false, then `(b || c)` must evaluate to false, and the effect of `a` is masked out.

B. Marking Functions

Before we present the preprocessing algorithm, we first provide intuition about the information needed by marking functions during runtime to motivate the design of the static partial evaluation. A *marking function* is required at runtime to measure MC/DC coverage. In essence, a marking function tracks whether each condition within a decision independently affects the decision when the condition is assigned both true and false values. A concrete example of a marking function is shown in Section IV.

To represent the obligations that need to be covered, we use triples: $\langle d_i, c_j, v \rangle$, to record that a condition c_j when assigned the value v independently affects the decision d_i. There are two possible values of v: true and false. We assign each decision a unique index: d_i, and each condition within a decision an index relative to the decision: c_j (in Figure 2, the `Decision` and `Condition` markings). We call the set of triples assigned by a test suite the *independence set*.

A challenge is that when a condition is evaluated at runtime, we do not necessarily know that it has an independent effect on the decision, because its effect may be masked out by a condition evaluated subsequently within the same decision. Similarly, the evaluation of the current condition may mask out the effect of earlier conditions within the same decision. During evaluation of a decision at runtime, we need to maintain a temporary set (*temp*) of triples to record conditions that are relevant but not yet known to have an independent effect. The marking function performs three operations whenever a condition is evaluated:

1) it *adds* the $\langle d_i, c_j, v \rangle$ pair associated with the condition and outcome to $temp$,
2) if the outcome of the condition leads to masking, it *removes* elements associated with previous conditions from $temp$, and
3) if the outcome of the condition causes the outcome of the decision to be known, it *unions* $temp$ into the independence set.

In order to create a marking function, we need to know for each condition (1) the index of the condition within the decision to add to the temporary set, (2) the indices of all conditions that the condition *masks* when it evaluates to true or false to remove them from $temp$, and (3) whether the condition *terminates* the decision, to know when to add the contents of $temp$ to the independence set. In Figure 2, these are stored in the (1) `Condition` and `Decision` markings, (2) the `masks` markings, and (3) the `terminates` markings, respectively. We next present pseudocode for computing the masking and termination information necessary for marking functions.

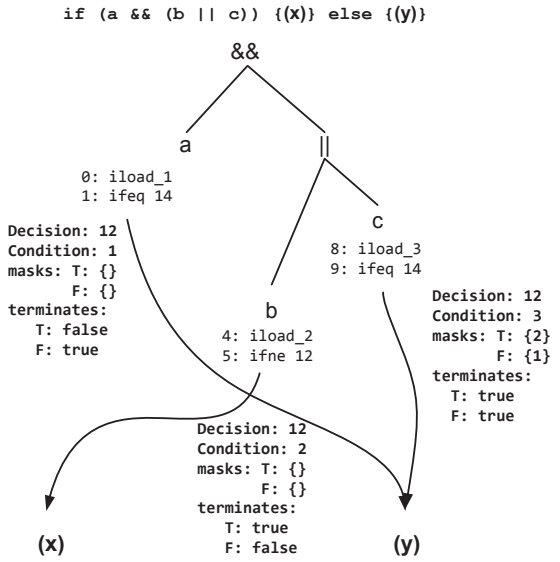

Fig. 2: Code snippet with complex decision and its translation into bytecode

C. Static Partial Evaluation

The pseudocode shown in Figure 3 computes the masking and termination information needed by the marking function in order to measure MC/DC coverage. This pseudocode assumes the existence of a simple abstract syntax tree (AST) for decisions containing AndExpr, OrExpr, NotExpr, and ConditionExpr (basic Boolean expression) classes with a handful of helper methods: (1) $hasParent$, which returns true/false depending on whether the expression is the child of another expression within a decision, (2) $getParent$, which returns the parent expression (if it exists), (3) $side$, which, assuming that the expression has a parent, returns whether this expression is the LEFT or RIGHT child of a binary expression, or the UNARY child of a unary expression, (4) $left$ and $right$, which return the left and right children of a binary expression, respectively, and (5) $getIndexes$, which returns the set of condition indices for the tree rooted at the expression.

The preprocessing algorithm in Figure 3 consists of three functions $partialEval$, $setMask$, and $terminates$. The $setMask$ method computes which preceding conditions in a decision are masked out by a given condition. The $terminates$ method marks if a given condition terminates a decision for a corresponding truth value. These methods use the $partialEval$ method to compute the required information. In order to generate the preprocessed coverage obligations, we run the $setMask$ and $terminates$ functions on each truth value (true and false) of all leaf-level conditions in every decision within the program. The masking and termination information computed for our simple example is shown in Figure 2.

```
procedure partialEval(E, v)
 1: if E.side() = RIGHT then
 2:    return v
 3: else if (E.getParent() IsA AndExpr) and v = FF then
 4:    return FF
 5: else if (E.getParent() IsA OrExpr) and v = TT then
 6:    return TT
 7: else if (E.getParent() IsA NotExpr) and v = TT then
 8:    return FF
 9: else if (E.getParent() IsA NotExpr) and v = FF then
10:    return TT
11: else
12:    return UNKNOWN
13:
procedure setMask(E, mask, v)
14: if v = UNKNOWN then
15:    return mask
16: else
17:    if E.hasParent() then
18:       if E.side() = RIGHT and
             ((E IsA AndExpr and v = FF) or
              (E IsA OrExpr and v = TT)) then
19:          mask = mask ∪ E.getParent().left().allIndexes()
20:       return setMask(E.getParent(), mask, partialEval(E, v))
21:    else
22:       return mask
23:
procedure terminates(E, v)
24: if v = UNKNOWN then
25:    return F
26: else if not E.hasParent() then
27:    return T
28: else
29:    return terminates(E.getParent(), partialEval(E, v))
```

Fig. 3: Pseudocode for masking and terminates functions

The *partialEval* function in Figure 3 is a short-circuit partial evaluator, which evaluates a Boolean expression given the value v of one of its subexpressions E. Partial evaluation is used to determine whether or not the decision is guaranteed to complete given a valuation of one of its leaf-level expressions, and also, as we will see, to determine the scope of masking. The function is three-valued, returning true or false (TT or FF) if the expression's value can be definitely determined, or UNKNOWN if it cannot.

To compute masking, recall that in short circuit MC/DC, masking occurs when the right side of an `and` expression evaluates to false or the right hand side of an `or` expression evaluates to true (line 18). In these cases, we want to mask out the affected left-side expressions (line 19). Note that in a complex decision, a condition may be on both the left side of one operator and the right side of another (for example, the condition `b` in Figure 2). In this case, even if the condition is on the left-hand side of its immediate parent, it may mask out conditions further up the decision tree structure. We recursively call the *setMask* function (line 20) until the value of the current operator is UNKNOWN.

For termination, we check to see whether we can completely determine the outcome of the decision via partial evaluation starting from the assignment of a 'leaf level' condition. If the outcome of partial evaluation is UNKNOWN, then the decision cannot be determined and we return false (line $25-26$). Alternately, if we complete evaluation of the decision with a T or F value, we return true (line $26-27$). Otherwise, we continue partial evaluation of the decision.

After the masking and termination information is computed, it is mapped to the bytecode generated by the Java compiler. As demonstrated in Figure 2, the structure of the original decision is translated into a block of bytecode with conditional branch instructions for each condition, ordered sequentially matching the left-to-right order of the conditions. We create a mapping between information generated by static partial evaluation to the conditional branch instructions in the Java bytecode.

The pseudocode presented in Figure 3 is suitable for a subset of Java. The preprocessing of full Java requires handling conditional operator expressions (ternary operators); nested decisions; and boolean relational operators such as boolean equality, inequality and XOR. Note that our implemented algorithm handles all these categories. For space considerations, we only present a subset of the expressions in the paper.

D. Decision, Branch, and Statement Coverage

In this work we focus only on MC/DC coverage; however, the preprocessing information computed can also be used to measure decision coverage, and branch. While statement coverage can be measured with minimal effort. For decision coverage, rather than maintain a set of conditions, one instead maintains a set of decisions; the masking information can be ignored. Instead, we simply examine the value (v) of conditions that terminate the decision: if true, a $\langle d_i, v\rangle$ value is added to the set of decisions. To measure branch coverage requires only a small additional bit of information (which we have in our implemented preprocessing tool): we need to know whether the decision is a 'control' decision that determines the value of an IF, WHILE, or FOR statement; if not, we do not record coverage information for it. Statement coverage can be measured by measuring branch coverage, then post-processing the control flow graph to determine which statements are covered. The marking functions can also be adapted appropriately for the other structural criteria.

IV. APPLICATIONS

Once the coverage information is computed by the preprocessor, it can be used by any tool capable of monitoring the instruction stream, such as a JVM that is instrumented, or by utilizing processor debug instructions to trap the instructions in which a condition is evaluated. Software verification frameworks that interpret bytecode, e.g., a model checker or symbolic execution framework, can also be used to efficiently compute the set of covered obligations on-the-fly.

A. Test case generation using Symbolic Execution

The algorithm in Figure 4 is an instance of a marking function which illustrates how the preprocessed coverage

```
procedure initialize
 1: ExploredConds := ∅, buffer := ∅, T := ∅
 2: s_0 :=getInitState()
 3: depthFirstSearch(⟨s_0, true⟩)
 4:
procedure depthFirstSearch(⟨s, π⟩)
 5: if error(s) or depth(s) or getSucc(s) = ∅ then
 6:    T := T ∪ generateTestInput(π); return
 7: for each ⟨s_i, π_i⟩ ∈ getSucc(s) do
 8:    updateObligations(s_i)
 9:    searchTree(⟨s_i, π_i⟩)
10:
procedure updateObligations(s)
11: if isConditionalStmt(s) then
12:    ⟨d_i, c_j⟩ := getDecisionCondition(s)
13:    v :=getConditionValue(s) /* v ∈ {T, F} */
14:    buffer := buffer ∪ {⟨d_i, c_j, v⟩}
15:    buffer := buffer\ getMaskedConds(⟨d_i, c_j, v⟩)
16:    if ⟨c_j, v⟩ ∈ terminates(d_i) then
17:       ExploredConds := ExploredConds ∪ buffer
18:       buffer := ∅
```

Fig. 4: Tracking MC/DC obligations in symbolic execution

```
procedure initialize(P', ImpConds)
 1: ExploredConds := ∅, buffer := ∅, T := ∅
 2: s_0 :=getInitState()
 3: DiSE(⟨s_0, true⟩)
 4:
procedure DiSE(⟨s, π⟩)
 5: if error(s) or depth(s) or getSucc(s) = ∅ then
 6:    T := T ∪ generateTestInput(π); return
 7: for each ⟨s_i, π_i⟩ ∈ getSucc(s) do
 8:    updateObligations(s_i)
 9:    if not prune(s) then
10:       DiSE(⟨s_i, π_i⟩)
11:
procedure prune(s)
12: if not isConditionalStmt(s) then return true
13: ⟨d_i, c_j⟩ := getDecisionCondition(s)
14: v :=getConditionValue(s)
15: for each ⟨d', c', v'⟩ ∈ ImpConds' \ ExploredConds do
16:    if isReachable(⟨d_i, c_j, v⟩, ⟨d', c', v'⟩) then
17:       return true
18: return false
```

Fig. 5: Pruning the search in DiSE based on coverage of impacted MC/DC obligations in evolving programs

information can be used during symbolic execution to generate test case inputs that cover MC/DC obligations. In symbolic execution, symbolic values are used in lieu of concrete values for program variables. A program state, s, consists of the unique program location identifier and values for the program variables, including heap locations. In symbolic execution the program state also includes a path condition π, that contains the constraints on the symbolic program variables in the program. As constraints are added to the path condition during execution, it is checked for satisfiability. A satisfiable path condition represents a feasible execution path whereas an unsatisfiable path condition represents an infeasible path.

In Figure 4, the *initialize* method initializes the set of observed complex conditions, $ExploredConds$, the temporary buffer $buffer$, and the set of test inputs T to empty. The $depthFirstSearch$ method is then invoked with the initial program state s_0 and the initial path condition ($true$).

The $depthFirstSearch$ method in Figure 4 explores the symbolic execution space until either (a) an error is encountered, (b) a user-defined depth bound is reached, or (c) the end of the path is reached and there are no more successors to the current state. Note that for programs with recursive methods and loops that operate on symbolic variables, a user-specified depth-bound is required for search termination.

The $getSucc$ method takes as input a symbolic program state and the current path condition to generate the set of successor states. Here, we assume that the path conditions are checked for satisfiability within the $getSucc$ method. For each successor state the $updateObligations$ method is invoked and then the search is recursively called on the successor state s_i.

The $updateObligations$ method in Figure 4 updates the set of observed test obligations. At line 11, there is a check to determine whether the current program location corresponds to a conditional branch statement in the object code. Recall that the static partial evaluation maps each conditional branch statement in the object code to a unique condition within a decision in the program source. At line 12, the $getDecisionCondition$ method returns a tuple containing the decision d_i and condition c_j for a given conditional branch statement, and at line 13, we get the value of the conditional branch statement: T indicates that the branch will be taken and F indicates that the branch will not be taken.

The $buffer$ data structure in Figure 4 contains all the unmasked conditions for the current decision being evaluated. At line 14, the condition c_j is added to the buffer, then at line 15 each condition in the buffer masked by c_j is removed from the buffer along with its corresponding value. At line 16, a check is performed to determine if the search has reached a condition whose value terminates a decision, and if so, adds all of the conditions in the buffer to the $ExploredCond$ set and clears the buffer. Note that when the next decision in the program is encountered, the buffer is empty.

At the end of symbolic execution, the set of $ExploredConds$ contains all of the MC/DC obligations covered during the analysis, and the set T contains the set of test inputs whose execution guarantees coverage of the obligations in the $ExploredCond$ set.

Similar algorithms could also be used with other search techniques such as model checking and dynamic symbolic execution, and for other analyses such as measuring test adequacy.

B. Regression Analysis

There are several automated program analysis techniques that leverage the structure of the code as well as the semantics of the system under test to reason about it. These techniques, however, cannot operate on instrumented code because of the changes to the the structure and semantics of the program caused by the instrumentation. One such analysis framework is Directed Incremental Symbolic Execution (DiSE) platform

for analyzing evolving software programs [23], [28]. DiSE leverages the differences between two related program versions to detect and characterize the differences in program behaviors between the two versions. DiSE supports various software maintenance tasks such as regression testing, regression verification, equivalence checking, and delta debugging, among others.

The inputs to DiSE are two related program versions P and P'. A source-level Abstract Syntax Tree differencing algorithm computes the set of syntactic changes to P resulting in P'. The changes are treated as slicing criteria, and standard control- and data- dependence analyses are used in the slicing algorithm to compute the set of program statements that may be impacted by the changes. DiSE uses the set of impacted locations to direct a symbolic search of the system under analysis to generate path conditions that encode impacted program behaviors.

In this work, we extend the DiSE framework to compute a novel analysis which combines the existing change impact analysis results with the preprocessed coverage information to compute a set of test inputs that satisfy complex structural coverage obligations, e.g., MC/DC, for program behaviors that may be impacted by the differences between P and P'.

At a high level, the modified DiSE algorithm, shown in Figure 5, uses information from the static analysis to determine for each impacted conditional branch statement at the bytecode-level, the corresponding condition and decision. The set $ImpConds$ represent the set of tuples containing condition, decision, and condition values for each impacted conditional branch statement. The $ImpConds$ set is provided as input, along with the new program P' in Figure 5.

The search strategy and the helper method definitions in Figure 5 are similar to those in Figure 4. The $isReachable$ function takes as input, two tuples of decision, condition, and condition values, and checks whether the corresponding conditional branch statement at the object-code level is reachable. In order to check reachability, DiSE performs a conservative check to determine if a path exists in the interprocedural control flow graph from one program location to another. When there are no unexplored decision condition tuples reachable from the current program location, the analysis prunes the search and backtracks. Note that it is possible to prune entire sub-trees reachable from the current program location when the current path will not lead to any impacted coverage obligations that have not already been covered.

V. Evaluation

We empirically evaluate our non-intrusive approach for computing with complex structural coverage metrics. Our evaluation addresses the following three research questions:

RQ1: How does our approach for preprocessingMC/DC coverage conditions improve the efficiency of test input generation in Symbolic PathFinder when compared with an existing state-of-the-art instrumentation based test generation technique [31]?

RQ2: How much overhead does symbolic execution incur when leveraging the coverage condition information computed by our preprocessor?

RQ3: Does a regression analysis technique based on our non-intrusive approach outperform a monolithic analysis technique?

A. Tool Support

We use a version of the Eclipse plugin from our previous work [31] to instrument the Java source code. The original implementation of the plugin did not include a check to determine if an obligation was previously covered, thus computing redundant instrumentation and exacerbating the issues outlined in Section II. To address this issue, we have modified the plugin to check if an obligation was previously covered before attempting to cover it again. This helps reduce the path explosion problem in the instrumentation algorithm, and improves the algorithm performance relative to the original plugin implementation. This modification was done to avoid bias in the evaluation towards our non-intrusive approach.

We implement the preprocessor for computing condition information as an Eclipse plugin. The plugin automatically analyzes the AST generated by the Eclipse Java compiler to compute the set of conditions and their respective locations in the source code that are relevant for tracking coverage obligations. The plugin results are saved to an XML file that can then be used during execution of the code under analysis to track coverage obligations.

We implement the test case generation application as an extension to the Symbolic PathFinder (SPF) [21], [22] engine for analyzing Java bytecode. For a given system under test, our extension reads in the XML and sets up data structures to map information about the uncovered decision, condition, and value triples in the source code to the Java bytecode. A listener in the extension then monitors the execution of the bytecode and updates coverage information based on the algorithm presented in Section IV. Finally, we use the regression analysis implemented in the DiSE framework [23], [28], another extension to SPF. The preprocessor plugin and extension to SPF that uses the preprocessed information can be downloaded from: https://github.com/spftest

B. Artifacts

We evaluated our technique on six Java artifacts. The first two artifacts are Java implementations of two container classes used in [37]. `FibHeap` is an implementation of a Fibonacci heap consisting of 286 SLOC. `TreeMap` is an implementation of a red-black tree extracted from `java.util.TreeMap` and consisting of 580 SLOC.

The third program, Traffic Anti-Collision Avoidance System (TCAS), is a Java implementation of a system to avoid air collisions. It is available from the Software-artifact Infrastructure Repository (SIR)[1] which consists of 150 SLOC. The fourth artifact, the Wheel Brake System (WBS), is a synchronous

[1]SIR Repository. http://sir.unl.edu.

reactive component derived from the WBS case example found in ARP 4761 [15], [29]. The WBS is used to provide safe breaking of the aircraft during taxi, landing, and in the event of an aborted take-off. The Simulink model was translated to C using tools developed at Rockwell Collins and manually translated to Java. It consists of 231 SLOC. The Altitude Switch (ASW) and FGS applications are asynchronous reactive components from the avionics domain. They were developed as a Simulink models, and were automatically translated to Java using tools developed at Vanderbilt University [33].

C. Experimental Setup

We explore one independent variable in our study: the method for computing achieved coverage. Two methods are explored: the non-intrusive approach presented in this work and a state-of the-art instrumentation based technique.

The two dependent variables in our evaluation are (1) the coverage achieved (against time) for *RQ1*, and (2) the overall wall clock time required to completely explore all paths (*RQ2*). Note that for *RQ1* it was not always feasible to run until the entire system was explored as the time required is sometimes too long. For both research questions, time measurements do not include the time required to perform the preprocessing step. For the artifacts used in this study, the preprocessing time to instrument the code or to preprocess the coverage information was a small fraction of the analysis time (less than one second).

In this study, we control two factors: the coverage criterion used, and the randomization of symbolic execution's exploration during test generation. We chose to explore the effectiveness of our approach in the context of MC/DC coverage—a complex structural coverage criteria mandated for use in critical systems domains such as avionics and automotive systems. MC/DC is also a prime example of a structural coverage criteria that quickly becomes unwieldy when computed using source code instrumentation. To avoid any bias associated with this search order, we randomize which branch is first explored, and run each approach/artifact combination ten times, thus producing ten different sets of results for each combination of artifact/approach.

D. Results and Analysis

For each artifact, we compute the set of decision, condition, and value triples for each conditional branch statement in the Java bytecode using the preprocessor (un-instrumented version). We also create an instrumented version of the program to enable tracking of MC/DC obligations. We randomize the search during symbolic execution for both approaches which entails randomly selecting the next state from the set of possible successors. The search is bound to one hour. We run ten trials of each experiment.

Figure 6 presents a comparison of the time taken to successfully generate test cases that cover MC/DC obligations for each of the six artifacts. We plot a point on the graph each time an MC/DC obligation is successfully covered by the preprocessing approach or the instrumented approach. The time taken for the coverage of each obligation is plotted on the X-axis while the total number of covered obligations (normalized) is plotted on the Y-axis. The maximum number of obligations covered by the two techniques for a given time period was treated as the total number of obligations in order to normalize the results on the Y-axis. Three artifacts, TCAS, Treemap, and WBS, completed full symbolic execution within the time bound of one hour, whereas for the ASW, Fibheap, and FGS examples, the search is terminated when the time bound of one hour was reached.

TABLE II: Overhead for computing MC/DC obligations.

Artifacts	Preprocessed	Std. Sym Exe	Increase	p-value
WBS	15.7	7.0	124.29	< 0.01
TreeMap	27.8	16.5	68.48	< 0.01
TCAS	95.6	88.6	7.9	< 0.01

RQ1 The results in Figure 6 demonstrate that symbolic execution using preprocessed information can quickly discover a large number of coverage obligations compared to the instrumented approach. For the ASW, TCAS, Fibheap, and FGS artifacts, our approach incurs less runtime overhead compared to that of the instrumented approach. In the WBS example, however, the instrumented approach begins covering MC/DC obligations a fraction of a second before our approach. This is due to the fact that symbolic execution using the preprocessed information has a constant overhead cost of approximately half a second to read in the preprocessed data and set up the data structures. In the WBS example, this becomes noticeable since the entire analysis is performed within a few seconds.

In the FGS artifact, the highest number of MC/DC obligations covered by the preprocessed approach among the ten runs before reaching the time bound is 552, whereas the maximum number of covered MC/DC obligations by the instrumented approach across ten trials is 407. For the ASW artifact, the preprocessed approach successfully explores a maximum of 79 MC/DC obligations prior to reaching the time bound, whereas the instrumented approach maximally covers 58 obligations. Overall, for the artifacts analyzed in this study, the preprocessed approach improves the efficiency of the test input generation in SPF when compared to the state-of-the-art instrumented based test generation technique.

RQ2 We measure and report the overhead incurred during symbolic execution when computing the covered MC/DC obligations (Preprocessed) as compared to symbolic execution on an un-instrumented program (Std. Sym Exe) in Table II. We use the three artifacts that complete execution within the given time bound. For each artifact we present the total wall clock time taken by each approach in seconds, the increase in runtime in percentage, and the p-value as computed using a bootstrap permutation test. The WBS and TreeMap artifacts have a significant overhead of 124% and 68% respectively. Note, however, that the WBS and TreeMap artifacts finish generation of the tree in less than 20 seconds. In these examples, the constant time to read in the preprocessed data at the start tends to dominate the total time taken. Whereas the TCAS has

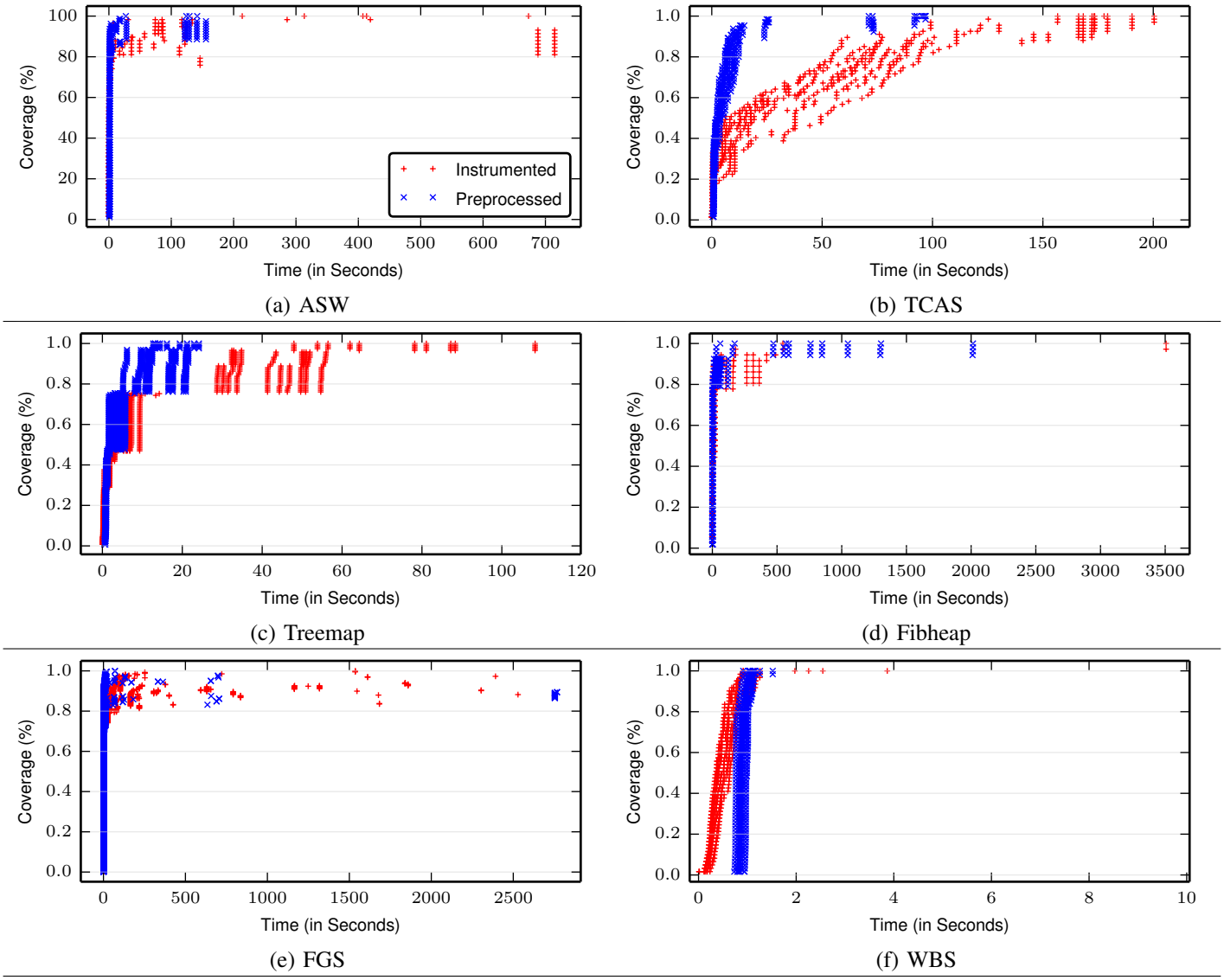

Fig. 6: Comparison of the time taken to observe MC/DC coverage obligation using the instrumentation approach versus the preprocessing based approach

TABLE III: States explored and time taken to compute impacted coverage obligations.

Version	DiSE States	DiSE Time	SE States	SE Time
V1	42	4	695	47
V2	166	12	679	44
V3	325	24	837	55
V4	52	3	743	51
V5	197	15	763	53

longer total runtime compared to WBS and TreeMap examples and here we observe an acceptable overhead of 7.9%.

RQ3 To evaluate the efficiency of our preprocessing approach in the context of a novel analysis, we compare the number of states generated and total time taken by DiSE to cover impacted MC/DC obligations compared to standard symbolic execution. The results for five versions of the TCAS artifact found in the SIR repository are presented in Table III. In Table III we present the total number of states explored and the total wall clock time is measured in seconds. DiSE explores fewer states and takes less overall time compared to standard symbolic execution (SE). Using the preprocessed information, we can leverage DiSE to efficiently compute and cover the set of impacted MC/DC obligations.

E. Threats to Validity

External: Our study is limited to six Java programs. Although the results may not generalize to other artifacts, we attempted to mitigate this threat by analyzing artifacts from two distinct classes of objects (1) data structure examples, and (2) Java applications representing embedded systems. Both classes represent considerable challenge for automatic test case generation. Furthermore, all of the artifacts used in the

evaluation of our approach have been used in previous studies of symbolic execution based techniques.

Internal: The primary threats to internal validity are the potential faults in the implementation of our algorithms to compute structural coverage metrics. We controlled for this threat by testing the tools and implementation of the algorithms on examples that could be manually verified. It is also possible that another implementation of an instrumentation-based approach exists which does not incur the overhead observed in our current implementation. To mitigate this threat we improved the current state-of-the-art published instrumentation-based approach.

Construct: We measured the efficiency of each approach based on coverage achieved over time. Naturally, the goal of testing is fault detection, with coverage serving as a proxy. We recognize (and have indeed demonstrated in our previous work) that the relationship between coverage and fault detection is not always straightforward, and that work strengthening this relationship is necessary [13].

VI. Related Work

Many code coverage analysis tools have been developed to assess the quality of software testing. These tools help developers ensure that all or most of the coverage obligations, e.g., statement, branch, are met during testing and identify the parts of the code that were not covered by the test suite. A number of commercial tools that perform code coverage analysis have been reviewed in a recent survey [39], including JCover [4], IBM's Rational PurifyPlus [5], and Clover [2] for Java. These tools are able to measure different levels of code coverage, e.g., statement/line/block, branch/decision, method/class, by monitoring the execution of the program and recording coverage information. Program execution monitoring represents an important challenge for code coverage tools as it is based on program instrumentation which can inflict considerable overhead on the testing process. Most of the tools reviewed in [39] use source code instrumentation, while a smaller number use byte code instrumentation (including JCover and PurifyPlus) or dynamic (runtime) instrumentation [1]. Regardless of the instrumentation approach, all of the coverage tools described in [39] have a reported instrumentation overhead of more than 30% [17].

Tikir and Hollingsworth propose a freely available tool which takes advantage of dynamic instrumentation and periodical garbage collection to remove instrumentation when it does not provide additional coverage in order to reduce the runtime overhead of code coverage [34]. Although they report reduced runtime overhead by 38-90% compared to the PureCoverage commercial tool [3], dynamic deletion of instrumentation code can introduce considerable risk and complexity into the instrumentation process. Another similar coverage tool that uses dynamic instrumentation to perform code coverage is proposed by Misurda at al. [19]. Compared to Tikir and Hollingsworth's tool, the prototype tool in [19] can remove instrumentation code immediately, rather than periodically, which gives it slightly better performance at branch coverage (average slowdown of 1.03 compared with Tikir and Hollingsworth). Experiments in [19] also reveal an average of 1.6 speed up compared with static instrumentation for branch coverage. Pin [18] provides a dynamic instrumentation API which uses a customized just-in-time (JIT) compiler to instrument code before it is translated for execution. This makes it portable and more efficient due to the optimizations performed by the JIT, however, it has been reported to increase the instruction count over the execution of native applications up to 60% on average [35].

Automated coverage driven testing techniques has been an active area of research for the past several decades [16]. Techniques based on symbolic execution [31], random testing [32], and search-based techniques [6], [10] have been developed for complex coverage criteria such as MC/DC. One recent approach based on Dynamic Symbolic Execution (DSE) proposes a new testing criterion *label coverage* which is intended to be both expressive and efficient in the context of DSE [7]. Their approach is capable of computing several coverage criteria including decision, condition, decision-condition and multiple-condition coverage. and it achieves scalability by performing tight instrumentation and iterative label deletion. On the other hand, as noted in [7], the approach is not capable of computing MC/DC coverage as it involves both path conditions and, in the case of xor and $=, \neq$ expressions involving Boolean arguments, choices as to the required test cases. Also, it requires modification of the source code to ensure all conditional expressions are *side-effect free*; our approach does not require any source code modifications.

The inspiration for this work comes from previous work in which we propose a hardware-supported monitoring framework and an efficient algorithm for tracking MC/DC based on the framework [38]. While the work presented in [38] addresses the issue of instrumentation overhead, it proposes a very different solution leveraging multicore processor architectures to create a non-intrusive general purpose monitoring framework, while this work proposes a technique for avoiding instrumentation by pre-computing the coverage information.

VII. Conclusion

In this paper, we introduced a non-intrusive and flexible approach for pre-computing structural coverage information. Our approach is based on a static partial evaluation of the decisions in the source code and a source-to-object code mapping procedure. The key novelty of our approach is that it uses pre-computed coverage information to enable applications to efficiently compute coverage obligations without the need for code instrumentation. Moreover, our approach enables new coverage driven analyses that rely on the structure of the code and are therefore not compatible with instrumentation-based techniques. Although the focus of this paper is on MC/DC, and symbolic execution based applications, the initial evaluation of our approach indicates it can support diverse structural coverage metrics and enable a variety of applications to efficiently compute coverage obligations.

REFERENCES

[1] Agitar Code Coverage. http://www.agitar.com/.

[2] Clover Java Code Coverage. http://www.cenqua.com/clover.

[3] IBM Rational Pure Coverage . http://www.rational.com/products/purecoverage/.

[4] Jcover Java code coverage analyzer. http://www.mmsindia.com/JCover.html.

[5] Purify Plus run-time analysis tools for application reliability and performance. http://www-03.ibm.com/software/products/en/purifyplus.

[6] Z. Awedikian, K. Ayari, and G. Antoniol. MC/DC automatic test input data generation. In *GECCO*, pages 1657–1664, 2009.

[7] S. Bardin, N. Kosmatov, and F. Cheynier. Efficient leveraging of symbolic execution to advanced coverage criteria. In *ICST*, pages 173–182, 2014.

[8] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *ISSTA*, pages 102–112, 2000.

[9] S. G. Elbaum and J. C. Munson. Evaluating regression test suites based on their fault exposure capability. *Journal of Software Maintenance: Research and Practice*, 12(3):171–184, 2000.

[10] K. Ghani and J. A. Clark. Automatic test data generation for multiple condition and mc/dc coverage. In *ICSEA*, pages 152–157, 2009.

[11] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.

[12] K. Hayhurst, D. Veerhusen, and L. Rierson. A practical tutorial on modified condition/decision coverage. Technical Report TM-2001-210876, NASA, 2001.

[13] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.

[14] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477, 2002.

[15] A. Joshi and M. P. E. Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In *SAFECOMP*, volume 3688 of *LNCS*, pages 122–135, September 2005.

[16] K. Lakhotia, P. McMinn, and M. Harman. Automated test data generation for coverage: Haven't we solved this problem yet? In *TAIC-PART*, pages 95–104, 2009.

[17] J. J. Li, D. M. Weiss, and H. Yee. An automatically-generated run-time instrumenter to reduce coverage testing overhead. In *AST*, pages 49–56, 2008.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[19] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *ICSE*, pages 156–165, 2005.

[20] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. ICSE, pages 75–84, 2007.

[21] C. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE*, pages 179–180, 2010.

[22] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic Pathfinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, pages 1–35.

[23] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI*, pages 504–515, 2011.

[24] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Engineering*, SE-11(4):367–375, Apr. 1985.

[25] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, Apr. 1997.

[26] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *ICSM*, pages 34–43, 1998.

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

[28] N. Rungta, S. Person, and J. Branchaud. A change impact analysis to characterize evolving program behaviors. In *ICSM*, pages 109–118, 2012.

[29] SAE-ARP4761. *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. SAE International, December 1996.

[30] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006. (Tool Paper).

[31] M. Staats. Towards a framework for generating tests to satisfy complex code coverage in java pathfinder. In *NFM*, pages 116–120, 2009.

[32] M. Staats, G. Gay, M. W. Whalen, and M. P. Heimdahl. On the danger of coverage directed test case generation. In *15th Int'l Conf. on Fundamental Approaches to Software Engineering (FASE)*, April 2012.

[33] J. Sztipanovits and G. Karsai. Generative programming for embedded systems. In *GPCE*, pages 32–49, 2002.

[34] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. *SIGSOFT Softw. Eng. Notes*, 27(4):86–96, July 2002.

[35] G.-R. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari. Analyzing dynamic binary instrumentation overhead. In *Workshop on Binary Instrumentation and Application*, 2007.

[36] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE*, Grenoble, France, 2000.

[37] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for Java containers using state matching. In *ISSTA*, pages 37–48, 2006.

[38] M. W. Whalen, M. P. Heimdahl, and I. J. D. Silva. Efficient test coverage measurement for mc/dc. Technical Report 13-019, University of Minnesota Twin Cities, 2013.

[39] Q. Yang, J. J. Li, and D. Weiss. A survey of coverage based testing tools. In *AST*, pages 99–103, 2006.

[40] H. Zhu and P. Hall. Test data adequacy measurement. *Software Engineering Journal*, 8(1):21–29, 1993.