Directed Incremental Symbolic Execution

Suzette Person
NASA Langley Research Center
suzette.person@nasa.gov

Guowei Yang
University of Texas at Austin
gyang@ece.utexas.edu

Neha Rungta
NASA Ames Research Center
neha.s.rungta@nasa.gov

Sarfraz Khurshid
University of Texas at Austin
khurshid@ece.utexas.edu

Abstract

The last few years have seen a resurgence of interest in the use of symbolic execution – a program analysis technique developed more than three decades ago to analyze program execution paths. Scaling symbolic execution and other path-sensitive analysis techniques to large systems remains challenging despite recent algorithmic and technological advances. An alternative to solving the problem of scalability is to *reduce* the scope of the analysis. One approach that is widely studied in the context of regression analysis is to analyze the *differences* between two related program versions. While such an approach is intuitive in theory, finding efficient and precise ways to identify program differences, and characterize their effects on how the program executes has proved challenging in practice.

In this paper, we present *Directed Incremental Symbolic Execution* (DiSE), a novel technique for detecting and characterizing the effects of program changes. The novelty of DiSE is to combine the *efficiencies* of static analysis techniques to compute program difference information with the *precision* of symbolic execution to explore program execution paths and generate path conditions affected by the differences. DiSE is a complementary technique to other reduction or bounding techniques developed to improve symbolic execution. Furthermore, DiSE does not require analysis results to be carried forward as the software evolves—only the source code for two related program versions is required. A case-study of our implementation of DiSE illustrates its effectiveness at detecting and characterizing the effects of program changes.

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

General Terms Verification, Algorithms

Keywords Program Differencing, Symbolic Execution, Software Evolution

PLDI '11 June 4–8, San Jose, California, USA

1. Introduction

Over the last three decades, *symbolic execution* [7, 22]—a program analysis technique for systematic exploration of program execution paths using symbolic input values—has provided a basis for various software testing and verification techniques. For each path it explores, symbolic execution builds a *path condition*, i.e., constraints on the symbolic inputs, that characterizes the program execution path. During symbolic execution, satisfiability checks are performed each time a constraint is added to the path condition to determine the feasibility of the path; if a path condition becomes unsatisfiable, it represents an infeasible path and symbolic execution does not consider any other paths that contain the known infeasible path as its prefix. The set of path conditions computed by symbolic execution can be used to perform various analyses of program behavior, for example, to check conformance of code to rich behavioral specifications using automated test input generation [10, 19].

Initial work on symbolic execution largely focused on checking properties of programs with primitive types, such as integers and booleans [7, 22]. Several recent projects generalized the core ideas of symbolic execution which enabled it to be applied to programs with more general types, including references and arrays [4, 10, 12, 19, 32]. These generalizations have been complimented by recent advances in constraint solving, which is the key supporting technology that affects the effectiveness of symbolic execution—checking path conditions for satisfiability relies heavily on the capabilities of the underlying constraint solvers. During the last few years, not only has raw computing power increased, enabling more efficient constraint solving, but basic constraint solving technology has also advanced considerably, specifically in leveraging multiple decision procedures in synergy, for example, as in Satisfiability Modulo Theory (SMT) solvers [9].

Scaling symbolic execution, as with other path-sensitive analyses, remains challenging because of the large number of execution paths generated during the symbolic analysis. This is known as the *path explosion problem*, and despite recent advances in reduction and abstraction techniques, remains a fundamental challenge to path-sensitive analysis techniques. A large body of work has been dedicated to developing techniques that try to achieve better scalability [1, 3, 11, 20]. One alternative approach to solving the problem of scalability is to reduce the scope of the analysis to only certain parts of the program.

Regression analysis is a well known example where the *differences* between program versions serve as the basis to reduce the scope of the analysis [14, 35, 38]. Analyses based on program *differences* are attractive and have considerable potential benefits since most software is developed following an evolutionary process. And, with the recent push toward agile development, differences between two program versions tend to be small and localized. The challenge, however, lies in determining precisely which program execution behaviors are *affected* by the program changes. Techniques to identify modified parts of the program can be broadly classified into two categories: *syntactic* and *semantic*. Syntactic techniques consider differences in the program source code or some other static representation of the source and can be computed efficiently. Semantic techniques are typically more expensive, but also compute generally more precise differences by considering the differences in the execution behaviors of the program.

In *Directed Incremental Symbolic Execution* (DiSE), our insight is to combine the *efficiencies* of static analysis techniques that compute program difference information with the *precision* of symbolic execution. The path conditions computed by DiSE then characterize the differences between two related program versions. The essence of symbolic execution is that it abstracts the semantics of program behaviors by generating constraints on the program inputs. The constraints for a given program execution behavior (path), referred to as a *path condition*, encode the input values that will cause execution to follow that path. The goal of this work is to direct symbolic execution on a modified program version to explore path conditions that may be *affected* by the changes. Consider a set of concrete (actual) input values that generate paths p and p' in the original and modified versions of the program respectively. If symbolic execution of p and p' result in different path conditions, then the path condition generated along p' is termed as affected.

Affected path conditions can be solved to generate values for the variables which, when used to execute the program, exhibit the *affected* program behaviors. The results of DiSE can then be used by subsequent program analysis techniques to focus on only the program behaviors that are affected by the changes to the program. DiSE enables other program analysis techniques to efficiently perform software evolution tasks such as program documentation, regression testing, fault localization and program summarization.

The novelty of DiSE is to leverage the state-of-the-art in symbolic execution and apply static analyses in synergy to enable more efficient symbolic execution of programs as they evolve. Static analysis and symbolic execution form the two phases in DiSE. Program instructions whose execution may lead to the generation of affected path conditions are termed as affected locations or affected instructions. The goal of the first phase is to generate the set of affected program instructions. The static analysis techniques in the first phase are based on intra-procedural data and control flow dependences. The dependence analyses are used to identify instructions in the source code that define program variables relevant to *changes* in the program. Conditional branch instructions that use those variables, or are themselves affected by the changes, are also identified as affected. In the second phase, the information generated by the static analysis is used to *direct* symbolic execution to explore only the parts of the programs affected by the changes, potentially avoiding a large number of unaffected execution paths. DiSE generates, as output, path conditions only on conditional branch instructions that use variables affected by the change or are otherwise affected by the changes.

In this work, we develop a conceptual framework for DiSE, implement a prototype of our framework in the Java PathFinder symbolic execution framework [26, 28, 36], present a case-study to demonstrate the effectiveness of our approach, and demonstrate, as a proof of concept, how the framework enables incremental program analysis to perform software evolution related tasks. For the examples used in our case-study, DiSE consistently explores fewer states and takes less time to generate fewer path conditions compared to standard symbolic execution when the changes affect only a subset of the program execution paths. This demonstrates the effectiveness of DiSE in terms of reducing the cost of symbolic execution of evolving software. Furthermore, we apply the results of our analysis to test case selection and augmentation to demonstrate the utility of such an analysis.

We make the following contributions:

- *A novel incremental analysis* that leverages the state-of-the-art in symbolic execution and applies a static analysis in synergy to enable efficient symbolic execution of programs as they undergo changes.
- A technique for *characterizing* program differences by generating path conditions affected by the changes.
- *A case-study* that demonstrates the effectiveness of DiSE in reducing the cost of performing symbolic execution and illustrates how DiSE results can be used to support software evolution tasks.

2. Background and Motivation

We begin with a brief explanation of symbolic execution, the underlying algorithm used in DiSE. Next, we present an example to demonstrate the motivation for the development of DiSE.

2.1 Symbolic Execution

Symbolic execution is a program analysis technique for systematically exploring a large number of program execution paths [7, 22]. It uses symbolic values in place of concrete (actual) values as program inputs. The resulting output values are computed as expressions defined over constants and symbolic input values, using a specified set of operators.

A symbolic execution tree characterizes all execution paths explored during symbolic execution. Each node in the tree represents a symbolic program state, and each edge represents a transition between two states. A symbolic program state contains a unique program location identifier (Loc), symbolic expressions for the symbolic input variables, and a path condition (PC). During symbolic execution, the path condition is used to collect constraints on the program expressions, and describes the current path through the symbolic execution tree. Path conditions are checked for satisfiability during symbolic execution; when a path condition is infeasible, symbolic execution stops exploration of that path and backtracks. In programs with loops and recursion, infinitely long execution paths may be generated. In order to guarantee termination of the execution in such cases, a user-specified depth bound is provided as input to symbolic execution.

We illustrate symbolic execution with the following example:

```
int y;
...
  int testX(int x){
1:  if (x > 0)
2:    y = y + x;
3:  else
4:    y = y - x;
5:  }
```

This code fragment introduces two symbolic variables: Y, the symbolic representation of the integer field y, and X, the symbolic representation of the integer argument x to procedure testX. For this example, symbolic execution explores the two feasible behaviors shown in the symbolic execution tree in Figure 1. When program execution begins, the path condition is set to true. When $X > 0$ evaluates to TRUE at line 1 in the source code, the expression $Y + X$

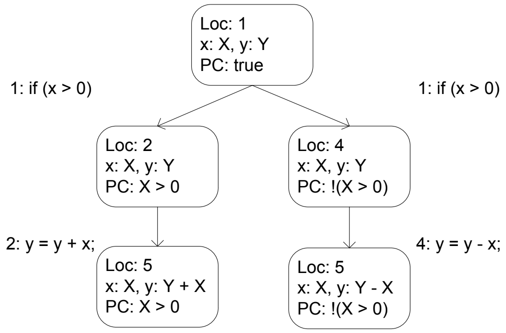


Figure 1. Symbolic execution tree for `testX()`

is computed and stored as the value of `y`. When $!(X > 0)$, the expression $Y - X$ is computed and stored as the value of `y`. A *symbolic summary* for procedure `testX` is made up of path conditions that represent the feasible execution paths in `testX`. The path conditions in the symbolic summary can be used as input to a subsequent analysis, e.g., the *solved* path conditions can be used as regression test case inputs.

2.2 Motivating Example

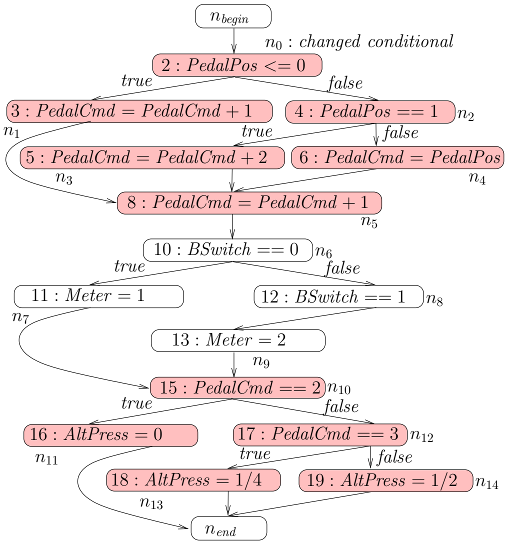
We use the example in Fig. 2 to illustrate how DiSE leverages information about program changes to direct symbolic execution and generate path conditions affected by the changes. Consider the source code for method `update(int PedalPos, int BSwitch, int PedalCmd)` shown in Fig. 2(a). The `update` procedure sets the value of two global variables, `AltPress` and `Meter`, based on the input values of its arguments. Assume a change was made to the `update` method at line 2 in Fig. 2(a) where the comparison operator is changed from $==$ to $<=$, as shown in the modified version of `update` presented in Fig. 2(a). Using full symbolic execution to validate this change results in 21 path conditions, each of which represents a program execution path of the modified version of `update`. As expected, the results of full symbolic execution include *all* execution paths for `update`, and no distinction is made between *affected* and *unaffected* program paths. And, as a result, any validation technique which uses these results may unnecessarily analyze unaffected program behaviors. For a small example such as this, a full analysis may be feasible; however, for larger methods or when complex constraints are involved, a full analysis may be infeasible.

To characterize the effects of the change to `update` using DiSE, we first compute the set of affected program locations. Consider the CFG computed for the modified version of `update` shown in Fig. 2(b). Each node corresponds to a program location in the source code; we use the node label to identify the corresponding code statement(s), and a node identifier appears in *italics* just outside the node, e.g., n_1, n_2, etc. Edges between the nodes represent possible flow of execution between the nodes. Nodes in the CFG that correspond to affected and changed program instructions are termed as affected and changed nodes respectively.

DiSE uses the results of a lightweight source file differential analysis to identify n_0 as having a direct correspondence to the change at line 2. The results of the static analysis computing affected locations indicate that nodes n_1, n_3, n_4, n_5, n_{11}, n_{13}, and n_{14} are *affected* write nodes–nodes that because of the change at line 2, may affect subsequent execution of a control node. Nodes n_0, n_2, n_{10}, and n_{12} are *affected* control nodes–nodes that may be affected by the change and that may affect the path condition. The information about affected locations is used to direct symbolic execution of the modified version of `update` to explore only the program behaviors affected by the change.

```
1:  int $AltPress := 0$; int $Meter := 2$ /* Global Vars */
procedure update(int $PedalPos$, int $BSwitch$, int $PedalCmd$)
2:  if $PedalPos <= 0$ /* changed conditional */ then
3:      $PedalCmd = PedalCmd + 1$
4:  else if $PedalPos == 1$ then
5:      $PedalCmd = PedalCmd + 2$
6:  else $PedalCmd = PedalPos$
7:
8:  $PedalCmd = PedalCmd + 1$
9:
10: if $BSwitch == 0$ then
11:     $Meter = 1$
12: else if $BSwitch == 1$ then
13:     $Meter = 2$
14:
15: if $PedalCmd == 2$ then
16:     $AltPress = 0$
17: else if $PedalCmd == 3$ then
18:     $AltPress = 1/4$
19: else $AltPress = 1/2$
```

(a)



(b)

Figure 2. A simplified version of a Wheel Brake System. (a) Example program. (b) Control flow graph.

Pruning. To illustrate how DiSE prunes symbolic execution using the set of affected locations, consider a feasible execution path, $p_0 := \langle n_0, n_1, n_5, n_6, n_7, n_{10}, n_{11} \rangle$, generated during directed symbolic execution. The path p_0 contains the sequence of affected nodes, $\langle n_0, n_1, n_5, n_{10}, n_{11} \rangle$, and the sequence of unaffected nodes, $\langle n_6, n_7 \rangle$. However, another feasible path, $p_1 := \langle n_0, n_1, n_5, n_6, n_8, n_9, n_{10}, n_{11} \rangle$, is *pruned* during symbolic execution because the sequence of *affected* nodes is already covered by p_0. The only difference between p_0 and p_1 is the sequence of *unaffected* nodes—p_1 contains $\langle n_6, n_8, n_9 \rangle$ as the sequence of unaffected nodes. DiSE applies the same pruning technique throughout symbolic execution to generate a total of seven path conditions for `update`, versus the 21 path conditions generated by full symbolic execution. Each path condition generated by DiSE characterizes a program execution path that is affected by the change to `update`.

3. Directed Incremental Symbolic Execution

In this section we first provide a high level overview of our DiSE technique. We then present the two main algorithms of DiSE that compute the set of affected path conditions.

3.1 Overview

Inputs to DiSE. The inputs to DiSE are the control flow graphs (CFGs) for two versions of a procedure, *base* and *mod*, and the results of a lightweight differential (*diff*) analysis (e.g., source line or abstract syntax tree *diff*) comparing the two versions of the procedure. The results of the *diff* analysis identify the locations of the differences in the source code between *base* and *mod*. During a pre-processing step, DiSE maps the change information to the corresponding nodes in each CFG. The CFG for the base version, CFG_{base}, has nodes marked as *removed*, *changed*, or *unchanged* with respect to the CFG of the modified version, CFG_{mod}. The nodes in CFG_{mod} are marked as *added*, *changed*, or *unchanged* with respect to CFG_{base}. The results of the *diff* analysis are also used to compute a map $(diffMap)$ that stores information relating nodes in CFG_{base} to their corresponding nodes in CFG_{mod}.

Computing Affected Locations. DiSE begins by performing a conservative, intra-procedural analysis to compute the set of nodes in CFG_{mod} that may be affected by the *removed* nodes in CFG_{base} or by the *changed* and *added* nodes in CFG_{mod}. This static analysis uses control dependence and data flow information to generate the set of affected CFG nodes in CFG_{mod}. These nodes correspond to conditional branch statements and to write statements in the modified version of the program that either influence, or may be influenced by the modifications made to the procedure, and as a result, may affect the path conditions computed by symbolic execution. Affected conditional branch nodes directly lead to the generation of affected path conditions. Affected write nodes indirectly lead to the generation of affected path condition–they either define a variable that may be subsequently read at an affected branch node, or the reachability of the affected write nodes is control dependent on an affected conditional branch node.

Directed Symbolic Execution. During this step, DiSE performs a form of *incremental* symbolic execution on the modified version of the procedure, leveraging the information computed during the previous step to explore only the parts of the program that are changed with respect to the base version of the procedure. The affected nodes information *directs* DiSE to explore only (feasible) paths where one or more *affected* nodes in CFG_{mod} are reachable on that path, and that sequence of affected nodes has not yet been explored. If either of these conditions is not met, then symbolic execution backtracks. By effectively "pruning" paths that generate unaffected path conditions, DiSE avoids the cost of exploring execution paths in the modified program version that are not affected by the change(s) to the program. The resulting set of path conditions computed by DiSE then characterizes the set of program execution behaviors in the modified version of the procedure that are affected by the change(s). Each (directly or indirectly) affected conditional branch node on the path is represented by a constraint in the path condition; conditional branch nodes that are not affected by the changes are represented by a constraint that represents a feasible path through the unaffected parts of the program.

3.2 Computing Affected Locations

The set of affected program locations is computed using conservative static analyses based on control and data dependences. The analysis is performed at an intra-procedural (per-method) level. The corresponding ramification is that DiSE does not generate affected path conditions arising from changes at the inter-procedural (sequence of methods) level. Extending the technique for inter-procedural system level analysis is part of our future work. We first present background definitions related to control flow graphs, control dependence, and data flow in order to define the rules DiSE uses to generate the affected locations sets.

Definition 3.1. *A* Control flow graph *(CFG) of a procedure in the program is a directed-graph represented formally by a tuple $\langle N, E\rangle$. N is the set of nodes, where each node is labeled with a unique program location identifier. The edges, $E \subseteq N \times N$, represent possible flow of execution between the nodes in the CFG. Each CFG has a single* begin*, n_{begin}, and* end*, n_{end}, node. All the nodes in the CFG are reachable from the n_{begin} and the n_{end} node is reachable from all nodes in the CFG.*

Definition 3.2. *(**Check for a CFG Path**) is a map* `IsCFGPath` *: $N \times N \mapsto \{T, F\}$ that returns true for a pair of nodes (n_i, n_j) if there exists a sequence of nodes $\pi := \langle n_0, n_1, \ldots\rangle$ such that $(n_k, n_{k+1}) \in E$ for $0 \leq k \leq |\pi| - 1$ and $n_0 = n_i$, $n_{|\pi|-1} = n_j$; otherwise it returns false.*

Definition 3.3. ***Vars** is the set of variable names that are either read or written to in a procedure.*

Definition 3.4. ***Cond** is the set of nodes $Cond \subseteq N$ where $n \in Cond$ is a conditional branch instruction.*

Definition 3.5. ***Write** is the set of nodes $Write \subseteq N$ where $n \in Write$ is a write instruction.*

Definition 3.6. *(**Variable Definitions**) is a map* `Def` *: $N \mapsto Vars \cup \{\perp\}$ that returns a variable $v \in Vars$ if the variable, v, is defined at node $n \in N$; otherwise returns $\perp$.*

Definition 3.7. *(**Variable Uses**) is a map* `Use` *: $N \mapsto 2^{Vars} \cup \{\perp\}$ that returns a set of variables $V \subseteq Vars$ where $v \in V$ is a variable being read at node n; otherwise returns $\perp$.*

Definition 3.8. *(**Post Dominance**) is a map* `postDom` *: $N \times N \mapsto \{T, F\}$ that returns true for an input pair of nodes (n_i, n_j) if, for each CFG path from n_i to n_{end}, $\pi := \langle n_i, \ldots, n_{end}\rangle$, there exists a k such that $n_j = n_k$ where $i \leq k \leq |\pi| - 1$ (n_j post dominates n_i); otherwise it returns false.*

Definition 3.9. *(**Control Dependence**) is a map* `controlD` *: $N \times N \mapsto \{T, F\}$ that returns true for an input pair of nodes (n_i, n_j) if node n_i has two successors n_k and n_l such that $(n_i, n_k), (n_i, n_l) \in E$, $n_k \neq n_l$,* `postDom`*$(n_k, n_j) == T$, and* `postDom`*$(n_l, n_j) == F$; otherwise it returns false.*

We provide intuitive descriptions for some definitions above using the example in Fig. 2. The set of $Vars$ contains variables $AltPress$, $PedalPos$, $PedalCmd$, $BSwitch$, and $Meter$. `Def`(n_9) returns the variable $Meter$ which is defined at line 13. Similarly the map `Uses`(n_{10}) returns $PedalCmd$, the variable being used at line 15. The map `postDom`(n_0, n_5) returns true because all paths from node n_0 to n_{end} have to go through n_5; an example path is $\langle n_0, n_1, n_3, n_5, \ldots, n_{end}\rangle$. Finally, node n_1 is control dependent n_0. The node n_0 has two successors n_1 and n_2, where `postDom`(n_1, n_1) is true and `postDom`(n_1, n_2) is false.

Approach. Two sets of affected nodes are computed and used by DiSE– the set of affected conditional (branch) nodes, ACN, and the set of affected write nodes, AWN. These sets contain nodes in CFG_{mod} that are marked as *changed* or *added* by the source line *diff* analysis. (Handling changes arising from remove nodes in CFG_{base} is described later in this section.) The ACN and AWN sets are updated by applying the rules specified in Fig. 3 until the sets reach a fixed-point. The analysis is guaranteed to terminate since the ACN and AWN sets contain nodes in a CFG – even nodes that are part of a loop are added at most once to these sets.

The rules in Fig. 3 specify the conditions under which DiSE adds nodes from CFG_{mod} to the affected sets based on the con-

$$\textbf{if } n_i \in ACN \wedge n_j \in \mathit{Cond} \wedge \texttt{controlD}(n_i, n_j)$$
$$\textbf{then } ACN := ACN \cup \{n_j\} \qquad (1)$$
$$\textbf{if } n_i \in ACN \wedge n_j \in \mathit{Write} \wedge \texttt{controlD}(n_i, n_j)$$
$$\textbf{then } AWN := AWN \cup \{n_j\} \qquad (2)$$
$$\textbf{if } n_i \in AWN \wedge n_j \in \mathit{Cond} \wedge \texttt{Def}(n_i) \in \texttt{Use}(n_j)$$
$$\wedge\, \texttt{Def}(n_i) \neq \perp \wedge\; \texttt{IsCFGPath}(n_i, n_j)$$
$$\textbf{then } ACN := ACN \cup \{n_j\} \qquad (3)$$

Figure 3. Updating affected sets based on control and data flow dependence.

$$\textbf{if } n_i \in \mathit{Write} \wedge (n_j \in AWN \vee n_j \in ACN)$$
$$\wedge\, \texttt{Def}(n_i) \in \texttt{Use}(n_j) \wedge\; \texttt{Def}(n_i) \neq \perp$$
$$\wedge\, \texttt{IsCFGPath}(n_i, n_j) \textbf{ then } AWN := AWN \cup \{n_i\} \qquad (4)$$

Figure 4. Updating AWN set based on reaching definitions.

trol dependence relation and the resulting data flow information. The first two rules Eq. (1) and Eq. (2) in Fig. 3 specify that conditional branches and write statements that are control dependent on a *changed* or *added* node in the CFG_{mod}, are added to the affected set. Other nodes are added to the affected sets by repeatedly applying the rules that, in essence compute the transitive closure on the control dependences between the nodes. Eq. (1) states that if there exists a node n_i in ACN, and a conditional node n_j such that n_j is control dependent on n_i, then n_j is added to the set ACN. Similarly in Eq. (2) if n_j is a write statement that is control dependent on n_i, an element in ACN, then n_j is added to AWN. Eq. (3) intuitively specifies that conditional branches that use variables defined in the affected write set are added to the affected conditional set. If the definition of a variable in a write statement, n_i, in AWN is used in a conditional branch statement, n_j, and there is a path in the CFG from n_i to n_j, then n_j is added to ACN. The rules are iteratively applied until the sizes of ACN and AWN do not change.

Eq. (4) in Fig. 4 is applied to update AWN based on the definitions of variables that reach the nodes in the affected sets, after the analysis in Fig. 3 reaches a fixed-point. Eq. (4) specifies that if a variable is defined at a write statement n_i that is used at node n_j such that n_j is in one of the affected sets, and there exists a CFG path from n_i to n_j then n_i is added to the AWN set. The rule specified in Eq. (4) is also guaranteed to reach a fixed-point and terminate since the rule is applied to the nodes in the CFG.

Handling Removed Instructions. When there are nodes marked as *removed* in CFG_{base}, it indicates that the corresponding node does not exist in CFG_{mod}. The removed node may, however, affect the execution of other nodes in CFG_{mod}. The algorithm to compute the effects of the removed nodes is shown in Fig. 5(a).

The removeNodes algorithm in Fig. 5 computes the control and data dependence flow information on CFG_{base}. The affected sets are initialized to *removed* nodes, and after applying the rules in Fig. 3 and Fig. 4, the affected sets contain all the nodes in CFG_{base} that are influenced by the remove nodes (lines 1-3 in Fig. 5(a)). Next, for each of the CFG_{base} nodes in the affected sets the corresponding node mapping to CFG_{mod} found in the *diffMap* is used to replace the elements in the affected sets (lines 4-9 in Fig. 5(a)). Recall that the source line differential analysis provides *diffMap* that maps *unchanged* and *changed* nodes corresponding from CFG_{base} to CFG_{mod}. For the nodes marked as *removed* in CFG_{base} the get method on *diffMap* returns the empty set.

```
procedure removeNodes($CFG_{base}$, $diffMap$)
1: $ACN$ := getRemovedConditionalNodes($CFG_{base}$)
2: $AWN$ := getRemovedWriteNodes($CFG_{base}$)
3: Apply rules in Fig. 3 and Fig. 4
4: $ACN$ := updateSets($ACN$, $diffMap$)
5: $AWN$ := updateSets($AWN$, $diffMap$)
procedure updateSets($AN$, $diffMap$)
6: $ModN := \{\}$
7: for each $n \in AN$ do
8:     $ModN := ModN \cup \{diffMap.\mathrm{get}(n)\}$
9: return $ModN$
```

(a)

ACN	AWN	n_i	n_j	Rule
$\{n_0\}$	$\{\}$			
$\{n_0, n_2\}$	$\{\}$	n_0	n_2	Eq. (1)
$\{n_0, n_2\}$	$\{n_1\}$	n_0	n_1	Eq. (2)
$\{n_0, n_2\}$	$\{n_1, n_3\}$	n_2	n_3	Eq. (2)
$\{n_0, n_2\}$	$\{n_1, n_3, n_4\}$	n_2	n_4	Eq. (2)
$\{n_0, n_2, n_{10}\}$	$\{n_1, n_3, n_4\}$	n_1	n_{10}	Eq. (3)
$\{n_0, n_2, n_{10}\}$	$\{n_1, n_3, n_2, n_{11}\}$	n_{10}	n_{11}	Eq. (2)
$\{n_0, n_2, n_{10}, n_{12}\}$	$\{n_1, n_3, n_4, n_{11}\}$	n_1	n_{12}	Eq. (3)
$\{n_0, n_2, n_{10}, n_{12}\}$	$\{n_1, n_3, n_4, n_{11}, n_{13}\}$	n_{12}	n_{13}	Eq. (2)
$\{n_0, n_2, n_{10}, n_{12}\}$	$\{n_1, n_3, n_4, n_{11}, n_{13}, n_{14}\}$	n_{12}	n_{14}	Eq. (2)
$\{n_0, n_2, n_{10}, n_{12}\}$	$\{n_1, n_3, n_4, n_5, n_{11}, n_{13}, n_{14}\}$	n_5	n_{10}	Eq. (4)

(b)

Figure 5. Computing affected nodes. (a) Algorithm to compute affected nodes resulting from removing certain program instructions. (b) Demonstration of the computation of affected node set for the example in Fig. 2.

The removeNodes algorithm in Fig. 5(a) generates all nodes affected by program instructions *removed* from the base version. Finally, the affected set is updated with any other *changed* or *added* nodes present in CFG_{mod} and the computing affected set algorithm as presented earlier is applied to generate the final affected sets.

Example. To illustrate how affected sets are generated, consider the example in Fig. 2. Suppose, the conditional branch at line 2 has the predicate $PedalPos == 0$ in the base program version that is modified to $PedalPos <= 0$ as shown in Fig. 2. The CFG node corresponding to line 2 in Fig. 2(a) is n_0 in Fig. 2(b). The ACN set is initialized to the lone element n_0 (a *changed* node) and the AWN set is initialized as empty. The sets are updated based on the rules in Fig. 3 as shown in Fig. 5(b).

Node n_2 in Fig. 2(b) is a conditional branch statement that is control dependent on n_0 causing n_2 to be added to ACN. The write statements at nodes n_1, n_3, and n_4 are added since they are control dependent on nodes n_0 or n_2. Nodes n_{10} and n_{12} are added to ACN since they use the variable $PedalCmd$ that is defined in nodes n_1, n_3, and n_4 contained in AWN. The write statement at node n_{11} is control dependent on n_{10}, while the write statements at nodes n_{13} and n_{14} are control dependent on n_{12}; hence nodes n_{11}, n_{13}, and n_{14} are added to AWN. Finally, when the sets ACN and AWN reach a fixed-point after applying the rules in Fig. 3, the Eq. (4) in Fig. 4 is applied. Applying the Eq. (4) rule adds the write statement at node n_5 that defines the variable $PedalCmd$ and is also used at nodes n_{10} and n_{12}.

The affected sets are used to direct the symbolic execution in the next phase of DiSE.

3.3 Directed Symbolic Execution

The directed symbolic execution technique executes the feasible paths that contain sequences of affected nodes generated in the static analysis. The analysis generates path conditions that contain constraints related to the modifications in the program. All feasible path conditions related to the affected nodes are generated during the analysis. Each path condition contains a feasible instance of the conditions generated from the unchanged parts of the code. As a result, the directed symbolic execution process generates fully formed path conditions, while at the same time avoids generating many path conditions arising from sequences of unaffected nodes in the program. This enables directed symbolic execution to not only explore all branches in the symbolic execution tree influenced by the affected nodes but, also to prune the paths related to the unaffected parts of the code.

The algorithm for directed symbolic execution is shown in Fig. 6. The inputs to the algorithm are the affected sets ACN and AWN, and a user-specified depth *bound* for the symbolic execution. The DiSE procedure is invoked with the initial symbolic state of the program. Recall that the symbolic state contains a current program location, a symbolic representation of the variables, and a path condition. There are four global sets initialized on lines 3 and 4 in Fig. 6. The sets $ExCond$ and $ExWrite$ track which of the affected nodes have been "explored" during symbolic execution while the sets $UnExCond$ and $UnExWrite$ track those nodes that are "unexplored" and still need to be explored. Hence, $ExCond$ and $ExWrite$ are initialized as empty whereas $UnExCond$ and $UnExWrite$ are initialized to the ACN and AWN sets respectively.

The DiSE procedure in Fig. 6 provides the basic search strategy. At line 5, if the current state is at a depth bound greater than the user-specified depth *bound* or the state is an error, then the search returns to explore an alternate path; otherwise exploration continues along the same path. The getCFGNode method on line 6 takes as input the symbolic state, s, and returns the corresponding CFG node, n. Note that the current program location of the symbolic state is used to map a symbolic state to its CFG node. The procedure UpdateExploredSet is invoked with CFG node n. For each successor state of s that the procedure AffectedLocIsReachable returns true the DiSE procedure is invoked with the corresponding successor state, exploring the relevant states in a depth-first manner.

The procedure UpdateExploredSet checks whether the input node n is contained in one of the unexplored sets ($UnExWrite$ or $UnExCond$) in order to track the affected nodes explored during symbolic execution (lines 30–35). If the node n is contained in the unexplored write set, $UnExWrite$, then n is removed from $UnExWrite$ and added to the explored write set, $ExWrite$. Similarly, a corresponding update is performed on $UnExCond$ and $ExCond$ when n is contained within $UnExCond$. The procedure ResetUnExploredSet, shown at lines 37–42, does the reverse; removes the input node from the explored sets and adds it to the respective unexplored set.

The AffectedLocIsReachable returns true if the CFG node, n_i, corresponding to the input symbolic state, s_i, can reach a node in the set of unexplored affected nodes to indicate that exploration should continue along the path that contains s_i; else, it returns false indicating that the path containing s_i is not influenced by the modifications to the program and should be pruned. The CheckLoops procedure is invoked to handle generating sequences of affected nodes through loops at line 15. A temporary set $UnExplored$ is initialized with the nodes in $UnExWrite$ and $UnExCond$, while $Explored$ is initialized with nodes from $ExWrite$ and $ExCond$ at lines 16 and 17. Next, when an affected node in $UnExplored$, n_j, is reachable from n_i; first, the variable *isReachable* is set to true, and second the nodes in the explored sets that are reachable from unexplored sets along the execution path containing n_i are moved back to the unexplored set (by invoking the ResetUnExploredSet procedure). Resetting the unexplored sets enables directed symbolic execution to explore all sequences of affected nodes that lie along feasible execution paths.

```
procedure init()/* $Input := ACN;\ AWN;\ bound$ */
 1: DiSE(getInitState())
 2:
 3: /* $ExCond := \emptyset;\ ExWrite := \emptyset$ */
 4: /* $UnExCond := ACN;\ UnExWrite := AWN$ */
procedure DiSE($s$)
 5: if GetDepth($s$) $> bound \lor$ error($s$) then return
 6: $n :=$ GetCFGNode($s$)
 7: UpdateExploredSet($n$)
 8: for each $s_i \in$ GetSuccessors($s$) do
 9:     if AffectedLocIsReachable($s_i$) then
10:         DiSE($s_i$)
11: return
12:
procedure AffectedLocIsReachable($s_i$)
13: $IsReachable := false$
14: $n_i :=$ getCFGNode($s_i$)
15: CheckLoops($n_i$)
16: $UnExplored := UnExWrite \cup UnExCond$
17: $Explored := ExWrite \cup ExCond$
18: for each $n_j \in UnExplored$ do
19:     if $\neg$IsCFGPath($n_i, n_j$) then continue
20:     $IsReachable := true$
21:     for each $n_k \in Explored$ do
22:         if $\neg$IsCFGPath($n_j, n_k$) then continue
23:         ResetUnExploredSet($n_k$)
24: return $IsReachable$
25:
procedure CheckLoops($n$)
26: if IsLoopEntryNode($n_i$) then
27:     for each $n' \in$ GetSCC($n$) do
28:         ResetUnExploredSet($n'$)
29:
procedure UpdateExploredSet($n$)
30: if $n \in UnExWrite$ then
31:     $ExWrite := ExWrite \cup \{n\}$
32:     $UnExWrite := UnExWrite \setminus \{n\}$
33: if $n \in UnExCond$ then
34:     $ExCond := ExCond \cup \{n\}$
35:     $UnExCond := UnExCond \setminus \{n\}$
36:
procedure ResetUnExploredSet($n$)
37: if $n \in ExWrite$ then
38:     $UnExWrite := UnExWrite \cup \{n\}$
39:     $ExWrite := ExWrite \setminus \{n\}$
40: if $n \in ExCond$ then
41:     $UnExCond := UnExCond \cup \{n\}$
42:     $ExCond := ExCond \setminus \{n\}$
```

Figure 6. Pseudocode for the directed symbolic execution algorithm using affected sets.

The CheckLoops procedure allows DiSE to explore sequences of affected nodes that are contained within loops. If the input node, n to CheckLoops is the entry node to a loop (line 26), then for all the nodes that are part of loop (strongly connected component containing n—GetSCC(n)) are added to the unexplored set if they have been previously explored by invoking the ResetUnExploredSet. A strongly connected component is a set of nodes where each node can be reached from all of the other nodes, and a strongly connected component has a single entry and a single exit point.

Example. In Table 1 we show part of the directed symbolic execution performed for the example in Fig. 2. For brevity, we refer to

	$\mathit{CFG\ Node\ for\ symbolic\ states}$	$\mathit{ExWrite}$	ExCond	$\mathit{UnExWrite}$	$\mathit{UnExCond}$
1	$\langle\rangle$	$\{\}$	$\{\}$	$\{n_1, n_3, n_4, n_5, n_{11}, n_{13}, n_{14}\}$	$\{n_0, n_2, n_{10}, n_{12}\}$
2	$\langle n_0\rangle$	$\{\}$	$\{n_0\}$	$\{n_1, n_3, n_4, n_5, n_{11}, n_{13}, n_{14}\}$	$\{n_2, n_{10}, n_{12}\}$
3	$\langle n_0, n_1\rangle$	$\{n_1\}$	$\{n_0\}$	$\{n_3, n_4, n_5, n_{11}, n_{13}, n_{14}\}$	$\{n_2, n_{10}, n_{12}\}$
4	$\langle n_0, n_1, n_5\rangle$	$\{n_1, n_5\}$	$\{n_0\}$	$\{n_3, n_4, , n_{11}, n_{13}, n_{14}\}$	$\{n_2, n_{10}, n_{12}\}$
5	$\langle n_0, n_1, n_5, n_6, n_7, n_{10}\rangle$	$\{n_1, n_5\}$	$\{n_0, n_{10}\}$	$\{n_3, n_4, n_{11}, n_{13}, n_{14}\}$	$\{n_2, n_{12}\}$
6	$\langle n_0, n_1, n_5, n_6, n_7, n_{10}, n_{11}\rangle$	$\{n_1, n_5, n_{11}\}$	$\{n_0, n_{10}\}$	$\{n_3, n_4, n_{13}, n_{14}\}$	$\{n_2, n_{12}\}$
7	$\langle n_0, n_1, n_5, n_6, n_7, n_{10}, n_{12}\rangle$	$\{n_1, n_5, n_{11}\}$	$\{n_0, n_{10}, n_{12}\}$	$\{n_3, n_4\}$	$\{n_2\}$
8	$\langle n_0, n_1, n_5, n_6, n_7, n_{10}, n_{12}, n_{13}\rangle$	$\{n_1, n_5, n_{11}, n_{13}\}$	$\{n_0, n_{10}, n_{12}\}$	$\{n_3, n_4, n_{14}\}$	$\{n_2\}$
9	$\langle n_0, n_1, n_5, n_6, n_7, n_{10}, n_{12}, n_{14}\rangle$	$\{n_1, n_5, n_{11}, n_{13}, n_{14}\}$	$\{n_0, n_{10}, n_{12}\}$	$\{n_3, n_4\}$	$\{n_2\}$
10	$\langle n_0, n_1, n_5, n_6, n_8(\mathit{no\ path})\rangle$	$\{n_1, n_5, n_{11}, n_{13}, n_{14}\}$	$\{n_0, n_{10}, n_{12}\}$	$\{n_3, n_4\}$	$\{n_2\}$
11	$\langle n_0, n_2\rangle$	$\{n_1\}$	$\{n_0, n_2\}$	$\{n_3, n_4, n_5, n_{11}, n_{13}, n_{14}\}$	$\{n_{10}, n_{12}\}$

Table 1. Part of the directed symbolic execution performed on the example in Fig. 2.

CFG nodes corresponding to symbolic states when describing the execution and sequence of states. For example the second column in Table 1 shows a sequence of CFG nodes corresponding to the sequence of symbolic states generated during symbolic execution. The $\mathit{ExWrite}$ and ExCond are initialized to the empty set, while the $\mathit{UnExWrite}$ and $\mathit{UnExCond}$ are initialized to the respective affected sets. At n_0, an affected conditional node is moved from $\mathit{UnExCond}$ to ExCond at line 2 in Table 1; similar updates occur at lines 3, 4, 5, and 7. At lines 4 and 5 in Table 1, execution continues along the successor states since a node in an unexplored set is reachable from the last node in the sequence. For example, at line 4, node n_{10} in $\mathit{UnExCond}$ is reachable from n_5 at the end of the sequence $\langle n_0, n_1, n_5\rangle$. There is no path to any nodes in the unexplored sets at line 10 in Table 1 from n_8.

In Table 1, at line 11, nodes are moved from explored to the unexplored set. The explored conditional branches n_{10} and n_{12} are reachable from node n_2, and the explored write instructions n_5, n_{11}, n_{13}, and n_{14} are reachable from n_2 as seen in Fig. 2(b); as node n_2 is added to explored set, nodes n_5, n_{10}, n_{11}, n_{12}, n_{13}, and n_{14} are added back to the unexplored sets in order for DiSE to explore different sequences of affected nodes and generate corresponding feasible execution paths (if possible).

Theorem 3.10. *For any sequence of affected nodes that lie on some feasible execution path within the specified depth bound, DiSE explores one execution path containing that sequence of nodes.*

Proof Sketch. By contradiction. There are two cases to consider: I) There exists a feasible path (within the specified depth bound) that contains a sequence of affected nodes, which DiSE does not explore and II) DiSE explores more than one feasible execution path for some sequence of affected nodes (within the specified depth bound).

Case I Let $q := \langle n_1, \ldots, n_k\rangle$ be a sequence of affected nodes, which is not explored by DiSE but is contained in a feasible execution path. By construction, DiSE must explore n_1, since it is an affected node that is added to $\mathit{UnExWrite}$ or $\mathit{UnExCond}$ during initialization. Assume n_i is the first node in q such that DiSE explores a feasible path, p, that contains the sub-sequence $\langle n_1, \ldots, n_{i-1}\rangle$ but does not explore an execution path that contains the sub-sequence $\langle n_1, \ldots, n_i\rangle$. Consider DiSE's exploration of p when it processes node n_{i-1}. Since n_i is reachable from n_{i-1} and is an affected node, n_i must be contained in $\mathit{UnExWrite}$ or $\mathit{UnExCond}$ (line 23). Hence DiSE will explore a path that contains the sub-sequence $\langle n_1, \ldots, n_i\rangle$. Contradiction.

Case II Assume for a sequence of affected nodes that lie on path p explored by DiSE, it explores another path p' containing the same sequence of affected nodes. Let n be the last affected node on path p such that the p and p' have the exact same sub-sequence of affected and unaffected nodes up to and including n. Let $q := \langle n, n_1 \ldots n_k, m\rangle$ be the sub-sequence of nodes on p such that each n_i is an unaffected node and m is an affected node. Let $q' := \langle n, n'_1 \ldots n'_j, m\rangle$ be the corresponding sub-sequence of nodes on p'. By the construction of the algorithm in Fig. 6, when DiSE considers the affected node n, it only explores one path and prunes the others by controlling the unexplored and unexplored sets in the `UpdateExploredSets` and `ResetUnExploredSet` until the next affected node, which in this case is m. Hence, q and q' are identical. Contradiction.

4. Evaluation

In this section we evaluate the effectiveness of DiSE at generating affected path conditions relative to the path conditions generated by full symbolic execution of the changed methods in evolving software. To perform our evaluation, we implemented DiSE and performed a case-study on three Java applications, comparing the results of DiSE with full, traditional symbolic execution of the changed versions of the method. We begin this section with a description of our implementation of DiSE.

4.1 Tool Support

We implemented DiSE in Symbolic PathFinder (SPF) [26, 28], a symbolic execution extension to the Java PathFinder model checker–a Java bytecode analysis framework [36]. SPF is an open source execution engine that symbolically executes Java bytecode. SPF supports a variety of constraint solvers/decision procedures for solving path conditions; in this work we use the Choco constraint solver [6]. In general, state matching is undecidable when states represent path conditions on unbounded input data. Hence, SPF does not perform any state matching and explores the symbolic execution tree using a stateless search. Furthermore, if the solver is unable to determine the satisfiability of the path condition within a certain time bound, SPF treats the path condition as unsatisfiable. While this situation does not occur for any of the artifacts in our study, this limitation of constraint solvers could affect DiSE, causing it to miss generating affected path conditions in the modified program. Loops and recursion can be bounded by placing a depth limit on the search depth in SPF or by limiting the number of constraints encoded for any given path; SPF indicates when one of these bounds has been reached during symbolic execution. There are no loops or recursive calls in the artifacts used in our empirical study, hence, we do not specify a depth bound.

DiSE extends SPF by implementing custom data and control dependence analyses that compute a conservative approximation of the the affected locations for a changed method. DiSE performs these analyses when the the modified method is invoked during symbolic execution, and then uses the information about the affected locations to direct symbolic execution within SPF.

4.2 Case-Study

The goal of our technique is to direct symbolic execution on a modified program version to explore only program conditions that are *affected* by the changes to the source code. We evaluate the cost and effectiveness of DiSE relative to full symbolic execution on the changed methods by considering the following research questions:

RQ1: How does the cost of applying DiSE compare to full symbolic execution on the changed method?

RQ2: How does the number of affected path conditions generated by DiSE compare with the number of path conditions generated by full symbolic execution?

4.2.1 Artifacts

To evaluate DiSE we compared method versions from three Java artifacts. The first program, the Wheel Brake System (WBS), is a synchronous reactive component from the automotive domain. The Java model is based on a Simulink model derived from the WBS case example found in ARP 4761 [17, 30]. We use the `update(int PedalPos, boolean AutoBrake, boolean Skid)` method in WBS to evaluate DiSE. This method determines how much braking pressure to apply based on the environment. The Simulink model was translated to C using tools developed at Rockwell Collins and manually translated to Java. It consists of one class and 231 source lines of code.

Our second artifact is a version of the Java program used to model NASA's On-board Abort Executive (OAE) . The OAE models the Crew Exploration Vehicle's prototype ascent abort handling software, receiving its inputs from sensors and other software components. The inputs are analyzed to determine the status of the ascent flight rules and to evaluate which ascent abort modes are feasible. Once a flight rule is broken and an abort mode is selected, it is relayed to the rest of the system for initiation. The version of OAE evaluated for this work consists of five classes. The method under analysis consists of approximately 150 source lines of code.

The third artifact we used to evaluate DiSE is the Altitude Switch (ASW) application. This program is a synchronous reactive component from the avionics domain. It turns power on to a Device Of Interest (DOI) when the aircraft descends below a threshold altitude above ground level (AGL). It was developed as a Simulink model, and was automatically translated to Java using tools developed at Vanderbilt University [34].

To evaluate DiSE in an empirical study we required multiple versions of each method being analyzed. Because multiple versions of these artifacts were not available, we generated versions by manually creating mutants of the base version (v0) of the method under analysis. When creating mutants, we considered a broad range of changes that can be applied to the code: change location, change type and number of changes. We introduced changes at the beginning, middle and end of each method. We also considered the control structures in the code, and make changes at various depths in nested control structures. Each mutant has one, two or three changed Java statements, resulting in up to nine changed nodes in the CFG for the changed version of a method as shown in Table 4.2.1.

In the WBS example, versions 1–6 contain a single changed Java source statement, versions 7–11 contain two changed statements and versions 12–16 contain three changed statements. For the ASW example, we made a single change to versions 1–11, and two changes to versions 12–15. For the OAE example, a single change was made to versions 1–6, two changes are made to versions 7 and 8, and version 9 contains three changed Java source statements. The versions that contain multiple changes were created by combining the individual mutations made to versions with a single change. For example, version 12 of the WBS example contains the same individual changes as versions 1, 4, and 5. Each change involved the addition, removal or modification of a statement. Control statements were modified by mutating the comparison operator, e.g., from $<$ to $<=$, or the operand, e.g., mutating the program variables involved in the comparison. Non-control statements were modified by changing the value assigned to a program variable.

4.2.2 Variables and Measures

The *independent* variable in our study is the symbolic execution algorithm used in our empirical study. We use the DiSE algorithm, and as a control, we use full (traditional) symbolic execution as implemented in the SPF framework. For our study, we selected three dependent variables and measures: 1) *time*, 2) *states explored*, and 3) *number of path conditions generated*. *Time* is measured as the total elapsed time reported by SPF. It includes the time spent computing the affected program locations and the time spent performing symbolic execution. *States explored* provides a count of the number of symbolic states generated during symbolic execution. *Number of path conditions generated* provides a count of the number of program execution paths generated by a given technique. *Time* and *states explored* are used to relate the cost of DiSE to the cost of full symbolic execution of the changed method (RQ1), while *number of path conditions generated* is used to judge the effectiveness of DiSE relative to full symbolic execution (RQ2).

4.2.3 Experiment Setup

To perform our study, we compiled all of our artifacts using Java version 1.6.0_22. We used a custom Java application to perform a lightweight *diff* analysis comparing the abstract syntax tree (AST) for each mutant with the AST for the base version of the program. We then analyzed each mutant version with DiSE, using the results of the AST diff. We also performed symbolic execution on each mutant using standard symbolic execution in SPF (JPF v6). The study was performed on a MacBook Pro running at 2.26 GHz with 2 GB of memory and running Mac OS X version 10.5.8.

4.2.4 Threats to Validity

The primary threats to *external validity* for our study are (1) the use of SPF within which we implemented our technique, (2) the use of Choco as the underlying constraint solver, (3) the selection of artifacts used to evaluate DiSE, and (4) the changes applied to create the mutants. Implementing DiSE in another framework or using another constraint solver/decision procedure could produce different results; however, replicated studies with other tool frameworks would address this threat. The artifacts selected for our study are control applications that are amenable to symbolic execution. The artifacts are comparable in structure and complexity to other artifacts that we are aware of that are used to evaluate symbolic execution techniques. The mutant versions we used to perform our study were created manually, and may or may not reflect actual program changes; however, the mutations were developed in a systematic way that considered program location, change type, and number of changes. Further evaluation of DiSE on a broader range of program types and on programs with actual version histories would address this threat.

The primary threats to *internal validity* are the potential faults in the implementation of our algorithms and in SPF. We controlled for this threat by testing our algorithms on examples that we can manually verify. With respect to threats to *construct validity*, the metrics we selected to evaluate the cost of DiSE are commonly used to measure the cost of symbolic execution.

Version	CFG Nodes		Time (mm:ss)		States Explored		Path Conditions	
	Changed	Affected	DiSE	Full Symbc	DiSE	Full Symbc	DiSE	Full Symbc
v1	0	0	00:19	17:19	23	3,948,476	0	1,728
v2	1	1	00:22	16:49	564	3,948,476	0	1,728
v3	1	16	00:21	17:03	30	3,948,476	1	1,728
v4	4	18	00:20	18:07	34	3,948,476	1	1,728
v5	4	18	00:20	17:56	34	3,948,476	1	1,728
v6	1	26	00:32	16:49	24,036	3,948,476	381	1,728
v7	5	12	00:29	16:37	37	3,957,692	2	1,728
v8	4	12	00:30	16:33	44	3,948,476	3	1,728
v9	4	3	00:32	16:22	23	3,948,476	0	1,728
v10	1	13	00:28	20:15	38	3,948,476	3	1,728
v11	1	0	00:31	22:14	23	3,948,476	0	1,728
v12	4	3	00:32	20:21	23	3,948,476	0	1,728
v13	1	0	00:31	17:13	23	3,948,476	0	1,728
v14	9	18	00:30	18:50	41	3,957,692	2	1,728
v15	5	30	00:47	18:36	24,148	3,948,476	383	1,728

(a) ASW Example

Version	CFG Nodes		Time (mm:ss)		States Explored		Path Conditions	
	Changed	Affected	DiSE	Full Symbc	DiSE	Full Symbc	DiSE	Full Symbc
v1	1	39	03:19	02:30	677,976	677,976	24	24
v2	1	7	00:08	02:22	93	677,976	17	24
v3	1	3	00:27	02:41	65,976	677,976	12	24
v4	1	0	00:08	02:44	17	677,976	1	24
v5	7	56	00:23	03:44	59,610	1,317,048	14	24
v6	1	1	00:08	02:44	17	677,976	1	24
v7	1	39	03:07	02:51	677,976	677,976	24	24
v8	8	57	00:29	03:45	59,610	1,317,048	14	24
v9	2	4	00:33	02:41	65,976	677,976	12	24
v10	2	39	03:40	02:51	677,976	677,976	24	24
v11	7	56	00:28	03:43	59,610	1,317,048	14	24
v12	8	65	00:31	03:54	70,129	1,317,048	6	24
v13	9	57	00:29	03:44	59,610	1,317,048	14	24
v14	3	39	03:39	02:51	677,976	677,976	24	24
v15	3	42	03:37	02:51	677,976	677,976	24	24
v16	8	56	00:28	03:43	59,610	1,317,048	14	24

(b) WBS Example

Version	CFG Nodes		Time (mm:ss)		States Explored		Path Conditions	
	Changed	Affected	DiSE	Full Symbc	DiSE	Full Symbc	DiSE	Full Symbc
v1	1	41	00:30	08:34	81,737	264,885	13,082	130,820
v2	1	2	00:04	08:34	23	264,885	2	130,820
v3	2	2	00:15	08:41	33,341	264,885	3,926	130,820
v4	2	2	00:04	08:39	56	264,882	3	130,820
v5	2	2	00:03	08:41	36	264,885	2	130,820
v6	3	7	00:57	10:27	134,065	264,885	26,164	130,820
v7	2	43	00:53	09:32	107,901	264,885	26,164	130,820
v8	3	4	00:17	10:32	41,193	264,885	7,852	130,820
v9	4	43	00:53	10:31	107,901	264,885	26,164	130,820

(c) OAE Example

Table 2. DiSE and Symbolic Execution Results

4.2.5 Results and Analysis

In this section, we present the results of our case-study, and analyze the results with respect to our two research questions. In Tables 4.2.1(a)–(c) we list the results of running DiSE and full symbolic execution on each version of each Java artifact. For each mutant version of the method, we list the number of CFG nodes changed (*Changed*), the number of CFG nodes affected by the changes (*Affected*), and the metrics described in Section 4.2.2 – the time to perform DiSE and the time to perform traditional symbolic execution of the mutant version as reported by SPF, the number of states explored during execution of each technique, and the number of path conditions generated by each technique in the resulting method summary. The results for DiSE are listed under the subheading *DiSE* and the results for traditional symbolic execution are listed under the subheading *Full Symbc*.

RQ1 (Cost). In Tables 4.2.1(a)–(c), we can see that for all versions of the ASW and OAE examples, and for the majority of versions in the WBS example, DiSE takes considerably less time than full symbolic execution. In many cases, the differences in time is several orders of magnitude. For all of the examples, when the changes to the program do not affect all path conditions (program paths), DiSE takes at most 20% of the time taken by full symbolic execution. For the five versions of the WBS example (v1, v7, v10, v14, and v15) where DiSE explores the same number of states as full symbolic execution, the time taken by DiSE is 9%–30% longer than symbolic execution. This extra execution time accounts for the overhead of computing the affected locations and supporting data structures.

For all of our examples, there is considerable variation in the number of states explored by DiSE; our intuition is that other factors beyond the number of changes, e.g., location and nature

Version	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	v11	v12	v13	v14	v15
# Changes	1	1	1	1	1	1	1	1	1	1	1	2	2	2	2
Selected	0	0	0	1	0	59	2	2	0	3	0	0	0	2	60
Added	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
Total Tests	0	0	1	1	1	59	2	2	0	3	0	0	0	2	60

(a) ASW Example

Version	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	v11	v12	v13	v14	v15	v16
# Changes	1	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3
Selected	20	17	12	1	14	1	20	14	12	20	14	5	14	20	20	14
Added	4	0	0	0	0	0	4	0	0	4	0	1	0	4	4	0
Total Tests	24	17	12	1	14	1	24	14	12	24	14	6	14	24	24	14

(b) WBS Example

Version	v1	v2	v3	v4	v5	v6	v7	v8	v9
# Changes	1	1	1	1	1	1	2	2	3
Selected	1	0	89	2	2	2	178	2	584
Added	291	2	0	0	0	582	0	582	0
Total Tests	292	2	89	2	2	584	178	584	584

(c) OAE Example

Table 3. Regression Testing Results

of the change, also have a considerable effect on the reductions that can be achieved by DiSE (or any other technique that can characterize the effects of program changes). We also conjecture that program structure, particularly with regard to the number and complexity of the constraints generated during symbolic execution contributes to the differences in execution time for each technique.

RQ2 (Effectiveness). The number of path conditions computed by DiSE varies greatly between versions for all three examples as shown in Tables 4.2.1(a)–(c). For the 15 versions analyzed in the ASW example, DiSE computes between zero and three path conditions for 13 versions. For the other two versions, DiSE computes approximately 22% of the path conditions generated by full symbolic execution (381 and 383 path conditions). In v1 of the ASW example, no CFG nodes are changed due to a compiler optimization that effectively masks the change, and for the remaining versions where no path conditions are generated, one or more CFG nodes was affected, however, the changes do not affect the path conditions generated by DiSE.

While none of the changes require DiSE to explore all paths in the ASW or OAE examples, in five of the 16 versions of the WBS example, DiSE generate the same number of path conditions as full symbolic execution. For these five versions, the number of *changed* CFG nodes ranges from one to three (out of 100 CFG nodes), and the number of *affected* CFG nodes ranges from 39 to 42. On the other hand, in v12 of the WBS example where only six path conditions are generated by DiSE (compared with 24 by full symbolic execution), eight CFG nodes are changed and 65 nodes are affected. In the OAE example, the change made to v6 affects seven CFG nodes (10%) and generates 26,164 path conditions – 20% of the path conditions computed by full symbolic execution. In v7, the changes affect 43 CFG nodes (62%) and yet the same number of of path conditions is generated. Based on our analysis of the mutants created for these examples, there does not appear to be any correlation between the number or percentage of affected nodes and the number of affected path conditions.

Overall, our study demonstrates that for the examples used in this study, DiSE has potential application for detecting and characterizing affected program behaviors in evolving software. In the artifacts used in our evaluation, DiSE was able to correctly identify and characterize the subset of path conditions computed by full symbolic execution as *affected*. In some instances, the change affected only a small percentage of path conditions, and in others, the change(s) had a much greater impact. When only a subset of the path conditions were affected by the changes, DiSE was able to consistently compute the affected path conditions in less time – often several orders of magnitude – than full symbolic execution; when all of the path conditions were affected by the changes, the overhead incurred by DiSE was between nine and 30% for these examples.

5. Discussion

The goal of DiSE is to enable more efficient symbolic execution by focusing it on generating affected path conditions. DiSE uses a conservative analysis to identify affected program locations. Thus, in principle, DiSE may generate some path conditions that represent unchanged paths. However, as experimental results demonstrate, DiSE is effective at focusing symbolic execution on affected program behaviors and enables efficient incremental symbolic execution.

5.1 Bug finding using DiSE

DiSE can handle programs with assertions when the "assert" statements are de-sugared into "if" and "throw" statements. In Java programs, this de-sugaring takes place when assert statements in Java source are compiled into Java byte code. Since DiSE performs symbolic execution on Java byte code, DiSE will not miss detecting a failed assertion violation caused by a program change. Thus, DiSE supports finding bugs when assertions are present and assertion failures characterize bugs. If the assertions were written in a language other than the underlying programming language (e.g., Alloy [16] assertions in Java programs), our technique would work with the assertions translated to Java (e.g., from Alloy).

5.2 Software evolution

The results of DiSE can be used to support various software evolution tasks. We present one such application—regression testing as it relates to test case selection and augmentation. We note that our goal is not to demonstrate the effectiveness of test case selection and augmentation, but, rather to demonstrate one application of DiSE results to support software evolution tasks.

SPF outputs values that can be used for the method arguments (test inputs) based on the generated path conditions. The test inputs are produced by solving the constraints in the path condition and using the resulting values to generate a call to the method under analysis. The results are output in string format. Our implementation of test case selection and augmentation is trivial in its approach – it simply performs a string comparison of the test cases generated for the original version (by full symbolic execution) with the tests generated by DiSE. Tests generated for the original version of the method represent an existing test suite. Tests generated by DiSE that are also found in the tests generated for the original version are marked as *selected*, while the other tests are considered tests to be *added* to augment the test suite.

The results of using DiSE to perform test case selection and test case augmentation for our three examples are shown in Tables 3(a)–(c). The combination of *selected* and *added* tests execute all of the branches in the program that are in some way affected by the changes made to the method under analysis. Full symbolic execution on the original version of ASW generates 256 tests, while full symbolic execution on the WBS and OAE examples generates 24 tests and 2,920 tests respectively. Note that the number of test cases generated by symbolic execution and by DiSE differs from the number of path conditions generated by each technique; this is due to the fact that the current implementation of the test generation tool computes input values only for the method arguments, i.e., a partial state. As a result, when fields are represented by symbolic values, multiple path conditions generated by a given technique may map to a single set of concrete inputs to the method under analysis.

For all of the artifacts in our study, the results of DiSE can be used to identify test cases that can be re-used, and the test cases that exercise affected path conditions for which test cases must be generated. For the ASW and WBS examples, DiSE results show that only a small number of test cases is necessary to augment the test suite for the new version of the program. For the OAE example, the results of DiSE indicate that for some changes, e.g., v3, it is unnecessary to generate any new test cases. For versions v6 and v8, however, only two of the existing test cases are valid for the new version of the program and 584 test cases are necessary to test the program behaviors identified as affected by DiSE. When only a subset of the path conditions is affected by the changes to a program, using DiSE results for test case selection and augmentation of the artifacts in our study would result in executing between 20% and 70% of the test cases that would be run in a re-test all approach, reducing the time necessary for regression testing of the new version of the program.

The requirements for soundness and completeness of an analysis are driven by the needs of the specific evolution task that uses the results of the analysis. For example, the test selection and augmentation application considered here covers all the branches in the method that are affected by the change. However, it does not necessarily provide full (bounded) path coverage. To achieve full path coverage, additional information could potentially be recorded from a previous run, and combined with DiSE results. As future work, we plan to investigate the use of DiSE for other software evolution tasks.

6. Related Work

Recent years have seen a significant growth in research projects based on symbolic execution, first introduced in the 1970's by Clarke [7] and King [22]. These projects have pursued three primary research directions to enhance traditional symbolic execution: (1) to improve its effectiveness [4, 10, 12, 19, 32]; (2) to improve its efficiency [1, 3, 5, 11, 20, 31, 37]; and (3) to improve its applicability [8, 18, 27, 33]. The novelty of DiSE is to leverage state-of-the-art symbolic execution techniques and apply a static analysis in synergy to enable symbolic execution to perform more efficiently as a program undergoes changes.

The projects to enhance the effectiveness of symbolic execution have focused on two areas. First, is to enable symbolic execution to handle programs written in commonly used languages, such as Java and C/C++, by providing support for symbolic execution over the core types used in these languages [4, 10, 12, 19, 32]. The second area of focus is to enable symbolic execution to work around the traditional limitation of undecidability of path conditions through the use of mixed symbolic/concrete execution to attempt to prevent the path conditions from becoming too complex [12, 32].

Research to enhance the efficiency of symbolic execution has followed three basic directions. One, to use abstraction with symbolic execution to reduce the space of exploration [1, 20]. Two, to perform *compositional* symbolic execution, where summaries of parts of code are used in place of an actual implementation to enable the underlying constraint solvers to perform more efficiently [3, 11]. Three, to enable symbolic execution to find bugs faster through the use of heuristics, e.g., genetic algorithms [37], that directly control the symbolic exploration and focus it on parts that are more likely to contain bugs.

Static analysis has also been used effectively for guiding symbolic execution. Chang's recent doctoral dissertation [5] uses a def-use analysis based on user-provided control points of interest, and applies a program transformation that incorporates boundary conditions on program inputs into the program logic to enable more efficient bug finding. Santelices and Harrold [31] use control and data dependencies to symbolically execute groups of paths, rather than individual paths to enable scalability. The key difference between DiSE and previous work is the ability of DiSE to utilize information about program differences for efficient symbolic execution as code undergoes changes.

While several projects have made significant advances in applying symbolic execution to test input generation and program verification – two traditional applications of symbolic execution – recent projects have used it as an enabling technology for various novel applications, including program differencing [27], data structure repair [18], dynamic discovery of invariants [8], as well as estimation of energy consumption on hardware devices with limited battery capacity [33].

An application of symbolic execution with a focus on program differences is regression testing [14, 15, 25]. Several recent projects use symbolic execution as a basis of test case selection and augmentation [29, 35, 38]. DiSE differs from these projects in its focus on the core symbolic execution technique to enable a variety of software evolution tasks – not only regression testing.

Program differencing, in general, is a well-studied research area with several techniques for computing differences [2, 21, 27] as well as leveraging them to enable various software evolution tasks [23]. Research in utilizing differences to speed-up symbolic execution by focusing it on code changes has only recently begun. Godefroid et al. [13] consider the problem of statically validating symbolic test summaries against changes, specifically for compositional dynamic test generation. Our approach is complementary since it uses change impact information to explore only the paths of the symbolic execution tree that are affected by the change, thereby reducing the cost of recomputing symbolic summaries.

The Java PathFinder model checker [36] has previously been used for incremental checking of programs that undergo changes [24, 39]. Incremental state-space exploration (ISSE) [24] focuses on evolving programs and stores the explored state space graph to use it for checking a subsequent version of the program, and reduces the time necessary for state-space exploration by avoiding the execution of some transitions and related computations that are not necessary. Regression model checking (RMC) [39] presents a complementary approach to ISSE and uses the difference between two versions to drive the pruning of the state space when model checking the new version. RMC computes reachable program coverage elements, e.g., basic blocks, for each program state during a recording mode run of RMC on the original version. Impact analysis is then used to calculate dangerous elements whose behavior may now differ because of changes. This is done by comparing the bytecodes and control-flow graphs for the two program versions. The dangerous elements information is then combined with the reachable elements information to prune safe sub-state spaces during a pruning mode run of RMC on the modified version of the program. DiSE takes inspiration from RMC and supports incremental symbolic execution, which is not addressed by RMC and ISSE. Moreover, a key difference between DiSE and previous work is that to analyze the current program version DiSE does not require the availability of the internal states of the previous analysis run – ISSE requires the state-space graph and RMC requires the dynamic reachability information for program coverage elements.

7. Conclusions and Future Work

In this paper, we introduced Directed Incremental Symbolic Execution (DiSE), a novel technique that leverages program differences to guide symbolic execution to explore and characterize the effects of program changes. We implemented DiSE in the symbolic execution extension of the Java PathFinder verification framework, and evaluated its cost and effectiveness on methods from three Java applications. The results of our case-study demonstrate that DiSE efficiently generates the set of path conditions affected by the change(s) to a program. We demonstrate the utility of our technique by using DiSE results to perform test case selection and test input generation for the examples in our study.

DiSE is an intra-procedural, incremental analysis technique that generates and characterizes method-level differences. DiSE does not generate affected path conditions arising from the control and data flow between various methods. As a result, DiSE does not consider the effects of the return value of a changed method flowing back to its calling context, nor does DiSE capture the effects of changes to the global state if whatever change is causing the effect does not also affect the path condition. We plan to extend DiSE to use an inter-procedural analysis to generate affected path conditions over the entire system as part of our future work.

We observed from our study that the number of changes is only one factor affecting which (and how many) path conditions are changed; program structure and the nature of the change also appear to have a considerable effect. We expect these factors to also affect the number of path conditions that are changed when we evaluate DiSE on a more diverse set of programs. In future work, we plan to conduct a more comprehensive evaluation of our DiSE technique.

Acknowledgments

The authors gratefully acknowledge the contributions of Matt Dwyer and Gregg Rothermel to early work on DiSE. The authors also thank Eric Mercer for the helpful comments to improve the paper. The work of Yang and Khurshid was supported in part by the NSF under Grant Nos. IIS-0438967 and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

References

[1] S. Anand, C. S. Pasareanu, and W. Visser. Symbolic execution with abstraction. *International Journal on Software Tools for Technology Transfer (STTT)*, 11:53–67, January 2009.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14(1):3–36, 2007.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[4] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, pages 2–23, 2005.

[5] W. C. Chang. *Improving Dynamic Analysis with Data Flow Analysis*. PhD thesis, University of Texas at Austin, 2010.

[6] Choco. Main–page Choco. `http://www.emn.fr/z-info/choco-solver/`, 2010.

[7] L. A. Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, ACM '76, pages 488–491, 1976.

[8] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *ICSE*, pages 281–290, 2008.

[9] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[10] X. Deng, Robby, and J. Hatcliff. Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. In *TAICPART-MUTATION*, pages 3–12, 2007.

[11] P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.

[12] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, pages 213–223, 2005.

[13] P. Godefroid, S. K. Lahiri, and C. Rubio-Gonzalez. Incremental compositional dynamic test generation. Technical Report MSR-TR-2010-11, Microsoft Research, 2010.

[14] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Transactions Software Engineering and Methodology*, 10(2):184–208, 2001.

[15] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *OOPSLA*, pages 312–326, 2001.

[16] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, MA, 2006.

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

[18] S. Khurshid, I. García, and Y. L. Suen. Repairing structurally complex data. In *SPIN*, pages 123–138, 2005.

[19] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.

[20] S. Khurshid and Y. L. Suen. Generalizing symbolic execution to library classes. In *PASTE*, pages 103–110, 2005.

[21] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE*, pages 333–343, 2007.

[22] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[23] S. K. Lahiri, K. Vaswani, and T. Hoare. Differential static analysis: Opportunities, applications, and challenges. In *FoSER*, pages 201–204, 2010.

[24] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan. Incremental state-space exploration for programs with dynamically allocated data. In *ICSE*, pages 291–300, 2008.

[25] H. Leung and L. White. Insights into regression testing. In *ICSM*, pages 60–69, 1989.

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

[27] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *FSE*, pages 226–237, 2008.

[28] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, pages 15–25, 2008.

[29] D. Qi, A. Roychoudhury, and Z. Liang. Test generation to expose changes in evolving programs. In *ASE*, pages 397–406, 2010.

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

[31] R. Santelices and M. J. Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *ISSTA*, pages 195–206, 2010.

[32] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for c. In *ESEC/FSE*, pages 263–272, 2005.

[33] C. Seo, S. Malek, and N. Medvidovic. An energy consumption framework for distributed Java-based software systems. Technical Report USC-CSE-2006-604, University of Southern California, 2006.

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

[35] K. Taneja, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Guided path exploration for regression test generation. In *ICSE, New Ideas and Emerging Results*, pages 311–314, 2009.

[36] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[37] Z. Xu, M. B. Cohen, and G. Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *GECCO*, pages 1365–1372, 2010.

[38] Z. Xu and G. Rothermel. Directed test suite augmentation. In *APSEC*, pages 406–413, 2009.

[39] G. Yang, M. B. Dwyer, and G. Rothermel. Regression model checking. In *ICSM*, pages 115–124, 2009.