Directed Incremental Symbolic Execution

GUOWEI YANG, Texas State University
SUZETTE PERSON, NASA Langley Research Center
NEHA RUNGTA, NASA Ames Research Center
SARFRAZ KHURSHID, University of Texas at Austin

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 to real systems remains challenging despite recent algorithmic and technological advances. An effective approach to address scalability is to *reduce* the scope of the analysis. For example, in regression analysis, *differences* between two related program versions are used to guide the analysis. While such an approach is intuitive, finding efficient and precise ways to identify program differences, and characterize their impact on how the program executes has proved challenging in practice.

In this article, we present *Directed Incremental Symbolic Execution* (DiSE), a novel technique for detecting and characterizing the impact of program changes to scale symbolic execution. 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 complements other reduction and bounding techniques for improving 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. An experimental evaluation using 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

Additional Key Words and Phrases: Program differencing, symbolic execution, software evolution

ACM Reference Format:
Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. 2014. Directed incremental symbolic execution. ACM Trans. Softw. Eng. Methodol. 24, 1, Article 3 (September 2014), 42 pages.
DOI: http://dx.doi.org/10.1145/2629536

1. INTRODUCTION

For over three decades, *symbolic execution* [Clarke 1976; King 1976]—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. The results computed by symbolic execution enable various analyses of program behavior, for example, to check conformance of code to rich behavioral specifications using automated test input generation [Khurshid et al. 2003; Deng et al. 2007].

The work of G. Yang and S. Khurshid was supported in part by the NSF under Grant Nos. IIS-0438967, CCF-0845628, CCF-1319688, and CNS-0958231, and AFOSR grant FA9550-09-1-0351.
Authors' addresses: G. Yang, Department of Computer Science, Texas State University, San Marcos, TX 78666; S. Person, NASA Langley Research Center, Hampton, VA 23681; N. Rungta, NASA Ames Research Center, Moffett Field, CA 94035; S. Khurshid, Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712. Corresponding email: GYang@txstate.edu.

DOI: http://dx.doi.org/10.1145/2629536

Initial work on symbolic execution largely focused on checking properties of programs with primitive types, such as integers and booleans [Clarke 1976; King 1976]. Later projects generalized the core ideas of symbolic execution to enable it to handle programs with more general types, including references and arrays [Khurshid et al. 2003; Godefroid et al. 2005; Sen et al. 2005; Cadar and Engler 2005; Deng et al. 2007]. Recent algorithmic techniques based on reduction, abstraction, composition, and parallel analysis have further enhanced the effectiveness of symbolic execution [Anand et al. 2009; Khurshid and Suen 2005; Godefroid 2007; Bush et al. 2000; Staats and Păsăreanu 2010; Siddiqui and Khurshid 2012]. These core algorithmic advances have been complemented by more efficient approaches for constraint solving—the key supporting technology that affects the effectiveness of symbolic execution—for example, by leveraging multiple decision procedures in synergy, that is, as in Satisfiability Modulo Theory (SMT) solvers [de Moura and Bjørner 2008]. Moreover, raw computing power has substantially increased during the last decade, thereby enabling symbolic execution to be applicable to larger programs. Despite these algorithmic and technological advances, scaling symbolic execution remains a key challenge because of the *path explosion problem*—the sheer number of paths to explore can be very large.

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 [Graves et al. 2001; Xu and Rothermel 2009; Taneja et al. 2011]. Analyses based on program differences are attractive and have considerable potential benefits since most software is developed following an evolutionary process. Moreover, 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 *impacted*[1] by the program changes.

In *Directed Incremental Symbolic Execution* (DiSE), our insight is to combine the *efficiencies* of a static impact analysis with the *precision* of symbolic execution to explore only program behaviors that may be *impacted* by the changes. The program behaviors computed by DiSE characterize the differences between two closely related program versions.

The essence of symbolic execution is that it abstracts the semantics of program behaviors by generating constraints on the program inputs. Program behaviors, which are defined by execution paths, are encoded using *path conditions*, where a path condition represents properties of inputs that execute the corresponding path.

Impacted path conditions can be solved to generate values for the inputs which, when used to execute the program, exhibit the *impacted* program behaviors. The results of DiSE can then be used by subsequent program analysis techniques to focus on only the program behaviors that may be impacted by the changes to the program. DiSE enables other program analysis techniques to efficiently perform software evolution tasks such as equivalence checking, 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. DiSE performs a two-phase analysis. The first phase of DiSE uses a static intraprocedural program slicing technique to compute the set of program locations (instructions) that may be impacted by the changes to the source code. In the second phase of DiSE, the information generated by the static analysis is used to *direct* symbolic execution to explore only the parts of the program impacted by the changes, potentially avoiding a large number of unimpacted execution paths.

[1]In this article, we use the terms *affected* and *impacted* interchangeably.

In this work, we develop a conceptual framework for DiSE, implement a prototype of our framework in the Java PathFinder symbolic execution framework [Visser et al. 2003; Păsăreanu et al. 2008; Păsăreanu and Rungta 2010], 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 the DiSE analysis results.

We make the following contributions.

—We present 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.
—We provide a technique for *characterizing* program differences by generating path conditions impacted by the changes.
—We give 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.

Scope and Limitations. This article is a revised version of our earlier paper presented at the Conference on Programming Language Design and Implementation (PLDI) 2011 [Person et al. 2011], which introduced DiSE. This article presents more details about the core algorithms that embody DiSE, new heuristics to guide the state space exploration, and a more comprehensive evaluation. Our focus in this article is on an intraprocedural analysis to compute program differences. Thus, we apply incremental symbolic execution to one method at a time. Moreover, the current implementation of DiSE does not compute the flow of impact through global heap locations. The subjects used in our evaluation operate on inputs with primitive types, and for all the subjects we inlined the methods invoked by the main method under symbolic execution. Some more recent extensions to DiSE, for example, an interprocedural version of the DiSE analysis [Rungta et al. 2012] and application of DiSE to regression verification [Backes et al. 2013b], as well as some ideas for future work are discussed in Section 6.

2. OVERVIEW

In this section, we provide a high-level overview of DiSE. The overall DiSE architecture is shown in Figure 1. We also present in this section, a motivating example and explain how DiSE can be applied to it in order to generate impacted program behaviors.

Inputs to DiSE. The inputs to DiSE are the source code for two related procedures M and M' as shown in Figure 1. A lightweight differential analysis (*diff*) (e.g., source line or abstract syntax tree *diff*) compares the source of M and M' to identify the syntactic differences between the two procedures. The outputs of the *diff* analysis are two *change sets*—sets of locations in the source code that are different between procedures M and M'. The change set of M contains the lines *removed* with respect to M'; while, the change set of M' contains the lines *added* with respect to M. Note that all *changed* lines are treated as *removed* in one version and *added* in the other.

Preproccessing. As a preprocessing step to DiSE, a control flow graph (CFG) is generated for each procedure M and M'. We refer to the CFG of procedure M as CFG_{base} and the CFG of procedure of M' as CFG_{mod}. During the preprocessing step, DiSE maps the change information to the corresponding nodes in each CFG. The CFG for the

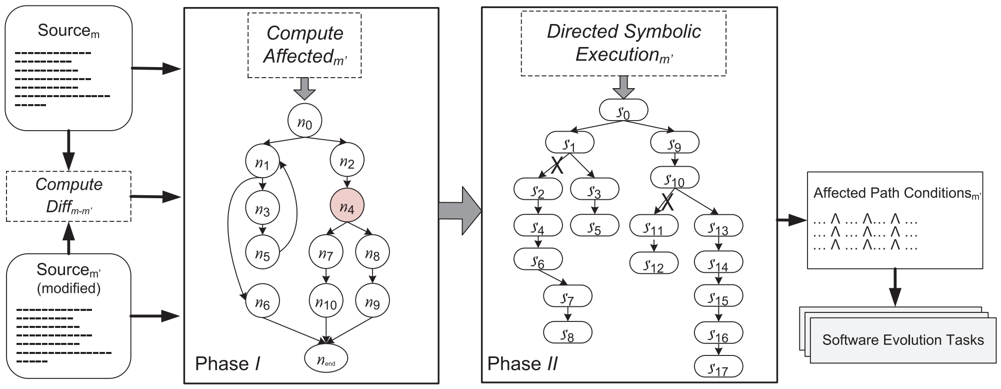


Fig. 1. DiSE architecture.

base version, *CFG*$_{base}$, has nodes marked as *removed* or *unchanged* with respect to the CFG of the modified version, *CFG*$_{mod}$. The nodes in *CFG*$_{mod}$ are marked as *added* or *unchanged* with respect to *CFG*$_{base}$.

Static Impact Analysis. In phase I, shown in Figure 1, DiSE computes a static impact analysis based on the syntactic differences between M and M'. The shaded node, n_4, in Figure 1 is marked as *added* and the analysis computes the set of nodes impacted by node n_4. The static impact analysis used by DiSE is a standard, intra-procedural program slicing analysis (forward and backward), that uses the *added* and *removed* change sets as the slicing criteria. The analysis uses control and data flow information to compute the *impact set*—the set of nodes in *CFG*$_{mod}$ that may be impacted by the *removed* nodes in *CFG*$_{base}$ or by the *added* nodes in *CFG*$_{mod}$. For example, if a node, n, is control dependent on a changed or another impacted node, then n is also marked as impacted. Similarly if n reads a value that was defined at a changed or impacted node then n is marked as impacted. The complete set of rules to compute the impacted nodes is presented in Section 4.1.

Directed Symbolic Execution. In phase II, DiSE uses the impact information to direct symbolic execution of the modified procedure M' as shown in Figure 1. This *incremental* symbolic execution generates the set of path conditions that encode the impacted program behaviors. DiSE leverages the impact sets (of program locations) computed by the static analysis to explore only the parts of the program that may be impacted by the change(s) with respect to the base version of the procedure. The impact sets are used to direct DiSE to explore only (feasible) paths where one or more *impacted* nodes in *CFG*$_{mod}$ are reachable on that path, and that sequence of impacted nodes has not yet been explored. If either of these conditions is not met, then symbolic execution backtracks. By effectively "pruning" paths that only differ in constraints generated at unimpacted statements, DiSE avoids the cost of exploring execution paths in M' that are not impacted by the change(s) to M'. The resulting set of path conditions computed by directed symbolic execution characterizes the set of program execution behaviors in M' that may be impacted by the change(s). The impacted path conditions can be used in various software maintenance tasks.

3. 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.

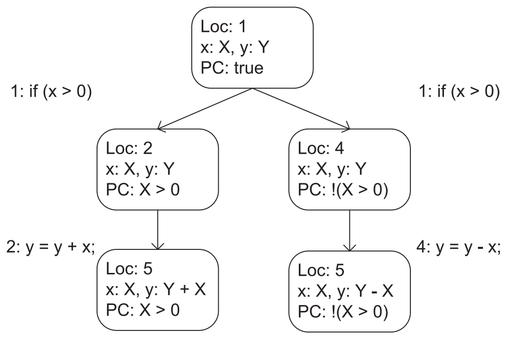


Fig. 2. Symbolic execution tree for `testX()`.

3.1. Symbolic Execution

Symbolic execution is a program analysis technique for systematically exploring a large number of program execution paths [Clarke 1976; King 1976]. 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 $\langle l, V_{sym}, V_{local}, \phi\rangle$ consists of a program location l, the set of symbolic input variables V_{sym} and their corresponding values, the set of local variables V_{local} and their corresponding values, and a path condition ϕ which represents the set of constraints over the symbolic variables in V_{sym} and constants.

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 2. 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 $Y + X$ 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

```
public class WBS {

    int AltPress = 0;
    int Meter = 2;

  public void update(int PedalPos,
   int BSwitch, int PedalCmd) {
        if (PedalPos == 0)
            PedalCmd = PedalCmd + 1;
        else if (PedalPos == 1)
            PedalCmd = PedalCmd + 2;
        else PedalCmd = PedalPos;

        PedalCmd = PedalCmd+1;

        if (BSwitch == 0)
            Meter = 1;
        else if (BSwitch == 1)
            Meter = 2;

        if(PedalCmd == 2)
            AltPress = 0;
        else if (PedalCmd == 3)
            AltPress = 1;
        else AltPress = 2;
    }

  }
```

(a): Method M

```
public class WBS {

    int AltPress = 0;
    int Meter = 2;

  public void update(int PedalPos,
   int BSwitch, int PedalCmd) {
        if (PedalPos <= 0)
            PedalCmd = PedalCmd + 1;
        else if (PedalPos == 1)
            PedalCmd = PedalCmd + 2;
        else PedalCmd = PedalPos;

        PedalCmd = PedalCmd+1;

        if (BSwitch == 0)
            Meter = 1;
        else if (BSwitch == 1)
            Meter = 2;

        if(PedalCmd == 2)
            AltPress = 0;
        else if (PedalCmd == 3)
            AltPress = 1;
        else AltPress = 2;
    }

  }
```

(b): Method M'

```
- if (PedalPos == 0)                         + if (PedalPos <= 0)
```

(c): *diff*(M,M')

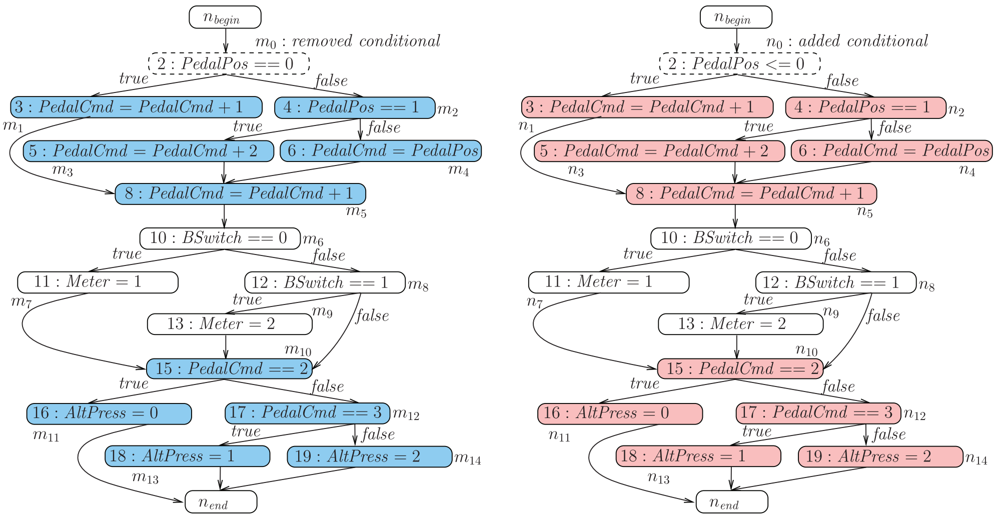
Fig. 3. Two related program versions for a simplified `update` method in a Wheel Brake System. (a) the first conditional checks whether the `PedalPos` is equal to 0, (b) the first conditional is updated to check the `PedalPos` is less than equal to 0, (c) the textual *diff* between the two versions.

execution paths in `testX`. The path conditions in the symbolic summary can be used as input to a subsequent analysis, for example, the *solved* path conditions can be used as regression test case inputs.

3.2. Motivating Example

We use two related program versions in Figure 3 to illustrate how DiSE leverages information about program changes to direct symbolic execution and *only* generate path conditions impacted by the changes. Two versions of an `update` method in a Wheel Brake System are shown in Figures 3(a) and (b), respectively. The `update` method has three input parameters `PedalPos`, `BSwitch`, and `PedalCmd` of type integer. The `update` procedure sets the value of two global variables, `AltPress` and `Meter`, based on the values of the input parameters.

A syntactic textual diff is performed between the two program versions. The output of the diff is shown in Figure 3(c). The conditional check $PedalPos == 0$ in Figure 3(a) is changed to $PedalPos <= 0$ in Figure 3(b). The "-" indicates that the program statement is removed from method *M* while "+" indicates that the program statement is added to



(a) CFG_{base} (b) CFG_{mod}

Fig. 4. Control flow graphs of a simplified version of the `update` method in a Wheel Brake System. (a) CFG for method M and (b) CFG for method M' in Figure 3.

method M'. Note that all changed program statements are marked as *removed* in one version of the program and *added* in the other version of the program.

The control flow graphs (CFGs) for each program version in Figures 3(a) and 3(b) are shown in Figures 4(a) and 4(b), respectively. Each node in the CFG corresponds to a program location in the source code; the node identifier appears in *italics* just outside the node, for example, m_1, m_2, etc. in CFG_{base} and n_1, n_2, etc. in CFG_{mod}. Edges between the nodes represent the possible flow of execution between the nodes. The nodes with a dashed outline represent the changes that were made to the `update` method. Node m_0 which represents the program statement $PedalPos == 0$ in Figure 4(a) is marked as removed while the node n_0 which represents the program statement $PedalPos <= 0$ is marked as added to the program.

The static analysis phase computes the potential impact of (a) the nodes removed in M and (b) the nodes added in M'. Standard control and data dependence information is used to compute the set of impacted program statements in M and M'. For example, node n_1 is control-dependent on n_0, and node n_5 reads the value written at n_1. The set of impacted nodes (shown as shaded nodes in Figure 4) in M and M' have a one-to-one correspondence because only the comparison operator is different between program statements in m_0 and n_0. The set of nodes marked as impacted in M' at the end of the static analysis are: n_0, n_1, n_2, n_3, n_4, n_5, n_{10}, n_{11}, n_{12}, n_{13}, and n_{14}.

The impacted program statements may affect certain path conditions in M'. To illustrate how DiSE uses the set of impacted locations to reduce the scope of symbolic execution, 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. 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`. Each path condition generated by DiSE characterizes a program execution path that is impacted by the change to `update`.

Using full symbolic execution (traditional symbolic execution with no pruning) 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 *impacted* and *unimpacted* program paths. And, as a result, any validation technique which uses these results may unnecessarily analyze unimpacted program behaviors. For a small example such as this, a full analysis is feasible; however, for larger methods or when complex constraints are involved, a full analysis may be computationally expensive and possibly infeasible.

4. THE DISE ALGORITHM

There are two main phases in the DiSE algorithm. The first phase is a static impact analysis that marks the set of CFG nodes that may be impacted by the changes. The second phase directs symbolic execution to only generate program behaviors along the impacted nodes.

4.1. Static Impact Analysis

The impact sets of affected program locations are computed using standard program slicing techniques, with the initial change sets computed by the syntactic *diff* analysis as the slicing criteria. The DiSE analysis presented here is an intraprocedural analysis and does not generate impacted path conditions arising from changes for which the impact flows between methods. 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 impact sets of affected program locations.

Definition 4.1. 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 transfer of control flow between the nodes in the CFG. Each CFG has a single *begin*, n_{begin}, and *end*, n_{end}, node. All of the nodes in the CFG are reachable from the n_{begin} node, and the n_{end} node is reachable from all nodes in the CFG.

Note that for simplifying the presentation, in our notation, we map each program location to a corresponding CFG node. In the evaluation, however, a CFG node maps to a basic block.

Definition 4.2. (*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 $0 \le k \le |\pi| - 1$ and $n_0 = n_i$, $n_{|\pi|-1} = n_j$; otherwise, it returns false.

Definition 4.3. *Vars* is the set of variable names that are either read or written to in a procedure.

Definition 4.4. (*Variable Definitions*) is a map `Def` : $N \mapsto \mathit{Vars} \cup \{\bot\}$ that returns a variable $v \in \mathit{Vars}$ if the variable, v, is defined at node $n \in N$; otherwise, returns $\bot$.

Definition 4.5. (*Variable Uses*) is a map `Use` : $N \mapsto 2^{\mathit{Vars}} \cup \{\bot\}$ that returns a set of variables $V \subseteq \mathit{Vars}$ where $v \in V$ is a variable read at node n; otherwise, returns $\bot$.

Definition 4.6. (*Post Dominance*) is a map `postDom` : $N \times N \mapsto \{T, F\}$ that returns true for a pair of nodes (n_i, n_j) if, for each CFG path from n_i to n_{end}, $\pi := \langle n_i, \ldots, n_{end}\rangle$,

$$\begin{aligned}
&\textbf{if } n_i \in \mathit{Imp} \wedge \texttt{controlD}(n_i, n_j) \\
&\quad \textbf{then } \mathit{Imp} := \mathit{Imp} \cup \{n_j\} && (1) \\
&\textbf{if } n_i \in \mathit{Imp} \wedge \texttt{Def}(n_i) \in \texttt{Use}(n_j) \wedge \texttt{Def}(n_i) \neq \bot \wedge \texttt{IsCFGPath}(n_i, n_j) \\
&\quad \textbf{then } \mathit{Imp} := \mathit{Imp} \cup \{n_j\} && (2) \\
\hline
&\textbf{if } n_j \in \mathit{Imp} \wedge \texttt{controlD}(n_i, n_j) \\
&\quad \textbf{then } \mathit{Imp} := \mathit{Imp} \cup \{n_i\} && (3) \\
&\textbf{if } n_j \in \mathit{Imp} \wedge \texttt{Def}(n_i) \in \texttt{Use}(n_j) \wedge \texttt{Def}(n_i) \neq \bot \wedge \texttt{IsCFGPath}(n_i, n_j) \\
&\quad \textbf{then } \mathit{Imp} := \mathit{Imp} \cup \{n_i\} && (4)
\end{aligned}$$

Fig. 5. Updating impact sets based on control and data dependence.

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

The set of *Vars* contains variables *AltPress*, *PedalPos*, *PedalCmd*, *BSwitch*, and *Meter* for the example in Figure 4. $\texttt{Def}(n_9)$ returns the variable *Meter*. Similarly, the map $\texttt{Uses}(n_{10})$ returns the variable read—*PedalCmd*. The map $\texttt{postDom}(n_0, n_5)$ returns true because all paths from node n_0 to n_{end} go through n_5. Finally, node n_1 is control dependent on node n_0. Node n_0 has two successors n_1 and n_2, where $\texttt{postDom}(n_1, n_1)$ is true and $\texttt{postDom}(n_1, n_2)$ is false.

The static impact analysis first computes the impact of nodes (a) added to M' and (b) removed from M.

4.1.1. Computing Impact of Added Nodes in M′. The impact set for CFG_{mod}, *Imp*, is initialized to the change set for CFG_{mod}. The change set contains nodes that are marked as *added* by the source line *diff* analysis. The rules for updating the impact set are specified in Figure 5. Nodes n_i, n_j, and n_k used in Figure 5 refer to nodes in CFG_{mod}.

Rule (1) and Rule (2) in Figure 5 compute the impact of the changes based on forward control- and data-flow dependence. The impact set is updated by iteratively applying Rule (1) and Rule (2) until a fixed-point is reached. This allows the analysis to compute the transitive closure on the forward control- and data-flow dependence between the nodes. The analysis is guaranteed to terminate even in the presence of loops because the *Imp* set contains CFG nodes; even the nodes that are part of a loop are added at most once to the impact set. Rule (1) states that if there exists a node, n_i in *Imp*, such that another node n_j is control dependent on n_i, then n_j is added to the set *Imp*. Rule (2) states that if there exists a node, n_j, that uses a variable defined at an impacted node, n_i, and there is a CFG path (Definition 4.2.) from n_i to n_j, then n_j is added to the impact set.

Rule (3) in Figure 5 computes the impact of the changes based on backward control-flow dependence. The rule in Rule (3) is applied iteratively to update *Imp*, marking a node n_i as impacted, if another impacted node n_j is control dependent on n_i.

Finally, the reaching definitions rule, Rule (4) in Figure 5, computes the impact of the changes based on backward data-flow dependence. The rule in Rule (4) is applied iteratively to update the impact set until a fix-point is reached. Again, the analysis is guaranteed to terminate even in the presence of loops because the fix-point computation is performed on CFG nodes. Rule (4) marks a node, n_i, as impacted if n_i defines a variable which is used in an impacted node n_j and there exists a CFG path from n_i to n_j.

The rules in Figure 5 essentially compute a program slice using the change set as the slicing criterion. The program slice computed with respect to the change set enables the DiSE analysis to reason about the set of program statements, that when executed, result in the generation of path conditions that may be impacted by the changes.

4.1.2. Computing the Impact of Removed Nodes in M. DiSE also computes the impact of the removed nodes in M and uses these results in the analysis of M'. DiSE initializes the impact set, Imp_{base}, for CFG_{base} to the set of removed nodes. When the rules in Figure 5 are applied to compute the impact of removed statements, the nodes n_i, n_j, and n_k represent nodes in CFG_{base}. First, Rule (1) and Rule (2) are applied iteratively to Imp_{base} until a fix-point is reached. Second, Rule (3) is applied iteratively to compute the backward control dependences. And finally, the reaching definitions rule (Rule (4)) is iteratively applied to Imp_{base}. Note that the order in which the rules are applied is the same as when computing the impact of added nodes.

4.1.3. Map Nodes from M to M′. The final step of the static analysis is to apply the impact of the statements removed from M to the CFG for M'. To do this, nodes that are impacted by removed statements in M are mapped onto the corresponding CFG nodes in M'. By construction, we know that for each impacted node, m_i, in M that is not removed in M', there exists a corresponding node n_i in M'. An Abstract Syntax Tree (AST) mapping algorithm is used to find corresponding nodes between M and M'.

$$Map = \{(m_i, n_i) | m_i \in CFG_{base} \wedge n_i \in CFG_{mod} \wedge m_i, n_i \notin diff(M, M') \wedge ASTClone(m_i, n_i)\}.$$

The *ASTClone* method takes as input two CFG nodes m_i and n_i. If the AST nodes corresponding to m_i and n_i have identical AST node types, identical labels, and their positions with respect to the unchanged parent nodes are the same, then the method returns true. Note that the *ASTClone* function can be replaced with more sophisticated algorithms to handle refactorings, such as variable renaming and re-ordering of statements. The information in the *Map* is combined with the impact set of M in order to compute the impact of the removed statements on M'. The impact set of M' is updated as follows:

$$Imp_{mod} = Imp_{mod} \cup \{n_j | (n_i, n_j) \in Map \wedge n_i \in Imp_{base}\}.$$

After mapping the impacted nodes from M to M', the set of impacted nodes in M' includes the impact of added and removed program statements.

4.1.4. Example. We describe how the impact sets are generated for the two program versions in Figure 3. Recall that the conditional branch at node n_0 in Figure 4 has the predicate $PedalPos == 0$ in the original version of the `update` procedure, M, which is modified to $PedalPos <= 0$ in the modified version of `update`, M'.

The impact set of M', Imp_{mod}, is initialized to the single *added* node, n_0, in the CFG shown in Figure 4(b). The impact set is updated based on the rules in Figure 5 and each update step is shown in Figure 6. Nodes n_1 and n_2 in Figure 4 are control dependent on n_0 causing n_1 and n_2 to be added to Imp_{mod}. Next, nodes n_3, and n_4 are added to Imp_{mod} since they are control dependent on node n_2. Nodes n_5, n_{10}, and n_{12} are added to *Imp* because they use the variable *PedalCmd* that is defined at impacted nodes n_1, n_3, and n_4 and they are on a CFG path with the impacted nodes. Node n_{11} is control dependent on n_{10}, while nodes n_{13} and n_{14} are control dependent on n_{12}; hence nodes n_{11}, n_{13}, and n_{14} are added to *Imp*. Note that rules Rule (3) and Rule (4) do not get used in generating the impact set for this small example.

The impact set of M, Imp_{base}, is initialized to the single element m_0 (the removed node). The same control and data dependence rules are applied iteratively to generate the set of shaded nodes in Figure 4(a). All the shaded nodes in Figure 4(a) correspond to

Imp_{mod}	n_i	n_j	Rule
$\{n_0\}$	-	-	-
$\{n_0, n_2\}$	n_0	n_2	Rule (1)
$\{n_0, n_1, n_2\}$	n_0	n_1	Rule (1)
$\{n_0, n_1, n_2, n_3\}$	n_2	n_3	Rule (1)
$\{n_0, n_1, n_2, n_3, n_4\}$	n_2	n_4	Rule (1)
$\{n_0, n_1, n_2, n_3, n_4, n_{10}\}$	n_1	n_{10}	Rule (2)
$\{n_0, n_1, n_2, n_3, n_4, n_5, n_{10}\}$	n_1	n_5	Rule (2)
$\{n_0, n_1, n_2, n_3, n_4, n_5, n_{10}, n_{11}\}$	n_{10}	n_{11}	Rule (1)
$\{n_0, n_1, n_2, n_3, n_4, n_5, n_{10}, n_{11}, n_{12}\}$	n_1	n_{12}	Rule (2)
$\{n_0, n_1, n_2, n_3, n_4, n_5, n_{10}, n_{11}, n_{12}, n_{13}\}$	n_{12}	n_{13}	Rule (1)
$\{n_0, n_1, n_2, n_3, n_4, n_5, n_{10}, n_{11}, n_{12}, n_{13}, n_{14}\}$	n_{12}	n_{14}	Rule (1)

Fig. 6. Demonstration of the computation of the impact set for M' on the CFG shown in Figure 4(b).

nodes that are already marked as impacted in Figure 4(b), hence, no additional nodes need to be marked as impacted due to the program statement removed in M.

In the next section, we discuss how the impact set is used to direct symbolic execution in the next phase of DiSE.

4.2. Directed Symbolic Execution

The directed symbolic execution phase of DiSE explores feasible paths that reach impacted nodes (statements) identified by the static impact analysis. The symbolic execution analysis generates path conditions that contain constraints related to the modifications to the program. 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 analysis generates constraints along a path, that is, both impacted and unimpacted constraints, but avoids generating the potentially many path conditions arising from sequences of unimpacted nodes in the program. This enables directed symbolic execution to not only explore all branches in the symbolic execution tree impacted by the affected nodes but, also to prune the paths related to the unaffected parts of the code.

4.2.1. Symbolic Exploration. Recall that a symbolic program state $\langle l, V_{sym}, V_{local}, \phi\rangle$ consists of a program location l, the set of symbolic input variables V_{sym} and their corresponding values, the set of local variables V_{local} and their corresponding values, and a path condition ϕ which represents the set of constraints over the symbolic variables in V_{sym} and constants. Again to simplify the presentation, in our notation, there exists a symbolic program state for every program location. Most symbolic execution engines, however, generate program states at conditional branch statements—we do the same as well in our implementation. The symbolic execution environment provides a set of functions to access information in a symbolic program state s and its possible execution.

—`GetInitState(`M'`)` returns the initial symbolic program state.
—`GetCFGNode(s)` returns the CFG node corresponding to the program location l in the symbolic program state s.
—`GetSuccessors(s)` generates successor states for s using the following rules.
  (1) If the program location l in symbolic program state s is a conditional branch that uses symbolic primitive data types in its branch predicate, p, then there are two possible successor states generated: (a) the true branch of the conditional statement represented by $\phi \wedge p$, and (b) the false branch of the conditional statement represented by the path condition $\phi \wedge \neg p$. A decision procedure is

```
function init($M'$, $Imp$, $bound$)
1: $Explored := \emptyset$;
2: DiSE(GetInitState($M'$))
3:
function DiSE($s$)
4: if GetDepth($s$) $> bound \vee$ error($s$) then return
5: $n :=$ GetCFGNode($s$)
6: if $n \in Imp$ then $Explored := Explored \cup \{n\}$
7: for each $s_i \in$ GetSuccessors($s$) do
8:    if $\neg$prune(GetCFGNode($s_i$)) then DiSE($s_i$)
9: return
10:
function prune($n_i$)
11: $noImpact := true$
12: for each $n_j \in Imp \setminus Explored$ do
13:    if $\neg$IsCFGPath($n_i, n_j$) then continue
14:    $noImpact := false$
15:    for each $n_k \in Explored \wedge$ IsCFGPath($n_j, n_k$) do
16:       $Explored := Explored \setminus \{n_k\}$
17: return $noImpact$
```

Fig. 7. Pseudocode for the directed symbolic execution algorithm using the impact set.

used to check the satisfiability of the updated path condition. This rule accounts for non-determinism arising from primitive data input.

(2) If the program location l in the symbolic program state s accesses an uninitialized symbolic complex data structure of type T, then the execution environment generates multiple possible successor states where the object is initialized to: (a) null, (b) new instance of type T, and (c) aliases to objects of type T that were previously initialized. This rule accounts for non-determinism arising from complex data structures. (Note: we assume data structures are initialized *lazily* [Khurshid et al. 2003].)

(3) If neither rule (1) nor (2) are satisfied, then the execution environment generates a single successor state obtained by executing location l and updating the values of the variables accordingly.

In the initial symbolic state, the path condition ϕ is initialized to *true*. The program locations are initialized to the start location of the program, while the variables in V_{sym} are assigned a symbolic value $v_\perp$ which represents an uninitialized and unconstrained variable. Symbolic execution engines such as SPF provide support for richer program constructs such as switch statements and virtual method invocations. To generate successor states for these program constructs our technique leverages the underlying analysis engine and can look up values in the virtual table or the switch-case table. In the case that the outcome of the switch statement depends on a symbolic value we generate successors for each feasible switch case value. Whereas if the virtual method is invoked on a symbolic complex object we generate successors for all the subtypes of the symbolic object; this allows us to handle complex data structures in the presence of polymorphism.

4.2.2. Incremental Symbolic Execution. The algorithm for directed symbolic execution is shown in Figure 7. The inputs to the algorithm are (a) the modified program being analyzed M', (b) the impact set *Imp* of M', and (c) a user-specified depth *bound* for the symbolic execution.

The `DiSE` procedure shown in Figure 7 illustrates the basic depth-first search strategy used during symbolic execution. The `DiSE` procedure is invoked with the initial symbolic state of the program. A global set *Explored* tracks which of the impacted nodes have been visited during symbolic execution. Hence, it is initialized to the empty set at line 1 in Figure 7. At line 4, if the current state is at a depth bound greater than the user-specified depth *bound* or the state is an error state, then the search returns (backtracks) to explore an alternate path; otherwise exploration continues along the same path. Depth refers to the length of the execution path (number of executed transitions); the error state is either a violation of a user-specified property in the form of an assertion violation or an unhandled exception.

The `getCFGNode` function invoked at line 5 in Figure 7 takes as input the symbolic state, s, and returns the corresponding CFG node, n for the current program location at s. If the CFG node, n, is in the impact set, it is marked as explored—by adding it to the *Explored* set at line 6. For each successor state of s that the function `prune` returns false, indicating execution should continue along the current path, `DiSE` is invoked with the corresponding successor state at line 8. The function `GetSuccessors` returns an ordered list of the successors of state s.

The function `prune` at lines 11-17 in Figure 7 returns false when the exploration should continue along the current path containing s_i; else, it returns true to indicate that the path containing s_i does not contain any impacted nodes (statements) resulting from the changes and can be safely pruned. If the CFG node, n_i, corresponding to the input symbolic state, s_i, can reach an impacted node, n_j, on the CFG (Definition 4.2) and n_j has not yet been explored, then the return value of the `prune` function is set to false. Finally, in the `prune` function, all of the impacted nodes, n_k, that are both reachable from n_j and have been explored, are removed from the *Explored* set. This enables directed symbolic execution to explore all possible sequences of impacted nodes that lie along feasible execution paths.

4.2.3. Search Strategies. At line 7 in the `DiSE` procedure, the function `GetSuccessors` returns a list of successor states that are ordered according to a specified exploration strategy for the depth-first search. In this article, we consider three exploration strategies with respect to a depth-first search: default, random, and greedy.

In the *default* strategy, the list of successor symbolic states is ordered using the default implementation choices of the underlying exploration engine. For example, the true branch may always be executed before the false branch of a conditional statement or vice-versa. In the *random* strategy, the successor states are randomly ordered.

In the *greedy* strategy, the successor states are ordered based on a distance estimate to an unexplored impacted location on the CFG path. For each successor state s_i, we compute a distance estimate to each unexplored impacted node. The resulting list of successor states is arranged in ascending order on the distance estimates. The successor state with the shortest distance estimate is explored first. The distance estimate is a lower bound on the number of CFG branches from a node n_i (corresponding to s_i) to each node, n_j, that is both impacted and unexplored:

$$\forall n_i . n_j \in Imp \backslash Explored : d_i := min(branches(n_i, n_j)),$$

$$DistanceEstimates = DistanceEstimates \cup \{d_i\}.$$

The lower bound on the number of CFG branches is computed using the all-pairs shortest path algorithm. This algorithm is cubic in the number of branches in the procedure. Finally, the greedy strategy returns the minimum value in the set, $min(DistanceEstimates)$, as the final distance estimate for the successor state, s_i. In the greedy strategy, metrics other than number of CFG nodes can also be used as a distance estimate, for example, the number of bytecodes.

Table I. Part of the Directed Symbolic Execution Performed on the Example in Figure 3

	CFG Node for symbolic states	*Explored*	*Unexplored* := *Imp* \ *Explored*
1	$\langle\rangle$	$\{\}$	$\{n_0, n_1, n_2, n_3, n_4, n_5, n_{10}, n_{11}, n_{12}, n_{13}, n_{14}\}$
2	$\langle n_0\rangle$	$\{n_0\}$	$\{n_1, n_2, n_3, n_4, n_5, n_{10}, n_{11}, n_{12}, n_{13}, n_{14}\}$
3	$\langle n_0, n_1\rangle$	$\{n_0, n_1\}$	$\{n_2, n_3, n_4, n_5, n_{10}, n_{11}, n_{12}, n_{13}, n_{14}\}$
4	$\langle n_0, n_1, n_5\rangle$	$\{n_0, n_1, n_5\}$	$\{n_2, n_3, n_4, n_{10}, n_{11}, n_{12}, n_{13}, n_{14}\}$
5	$\langle n_0, n_1, n_5, n_6, n_7, n_{10}\rangle$	$\{n_0, n_1, n_5, n_{10}\}$	$\{n_2, n_3, n_4, n_{11}, n_{12}, n_{13}, n_{14}\}$
6	$\langle n_0, n_1, n_5, n_6, n_7, n_{10}, n_{11}\rangle$	$\{n_0, n_1, n_5, n_{10}, n_{11}\}$	$\{n_2, n_3, n_4, n_{12}, n_{13}, n_{14}\}$
7	$\langle n_0, n_1, n_5, n_6, n_7, n_{10}, n_{12}\rangle$	$\{n_0, n_1, n_5, n_{10}, n_{11}, n_{12}\}$	$\{n_2, n_3, n_4, n_{13}, n_{14}\}$
8	$\langle n_0, n_1, n_5, n_6, n_7, n_{10}, n_{12}, n_{13}\rangle$	$\{n_0, n_1, n_5, n_{10}, n_{11}, n_{12}, n_{13}\}$	$\{n_2, n_3, n_4, n_{14}\}$
9	$\langle n_0, n_1, n_5, n_6, n_7, n_{10}, n_{12}, n_{14}\rangle$	$\{n_0, n_1, n_5, n_{10}, n_{11}, n_{12}, n_{13}, n_{14}\}$	$\{n_2, n_3, n_4\}$
10	$\langle n_0, n_1, n_5, n_6, n_8(no\ \ path)\rangle$	$\{n_0, n_1, n_5, n_{10}, n_{11}, n_{12}, n_{13}, n_{14}\}$	$\{n_2, n_3, n_4\}$
11	$\langle n_0, n_2\rangle$	$\{n_0, n_1, n_2\}$	$\{n_3, n_4, n_5, n_{10}, n_{11}, n_{12}, n_{13}, n_{14}\}$

4.2.4. Example. In Table I, we show part of the directed symbolic execution analysis performed for the example in Figure 3. For brevity, we refer to the CFG nodes in Figure 4 corresponding to symbolic states when describing the execution and sequence of states. For example, the second column in Table I shows the sequence of CFG nodes corresponding to the sequence of symbolic states explored during symbolic execution. At the start of directed symbolic execution, the *Explored* set is initialized to empty and the *Unexplored* set is initialized to all of the impacted CFG nodes. When symbolic execution reaches the state corresponding to node n_0, the impacted node, n_0, is added to the *Explored* set as shown at line 2 in Table I; similar updates occur at lines 3, 4, 5, and 6. Consider line 4, in Table I, an unexplored impacted node n_{10} is reachable from n_5 at the end of the sequence $\langle n_0, n_1, n_5\rangle$; hence, symbolic execution continues. However, when symbolic execution reaches node n_8 at line 10 of Table I, there are no unexplored impacted nodes reachable from node n_8, so DiSE backtracks, pruning the current execution path. In Table I, at line 11, nodes n_5, n_{10}, n_{11}, n_{12}, n_{13}, and n_{14} are moved from the *Explored* set to the *Unexplored* set when the state corresponding to node n_2 is explored after backtracking. This allows DiSE to explore all possible permutations of the impacted nodes and generate corresponding feasible execution paths (when possible).

THEOREM 4.8. *For any sequence of impacted nodes that lie on some feasible execution path within the specified depth bound, DiSE explores one execution path containing that sequence of nodes.*

Correctness Argument. We argue the correctness by contradiction. There are two cases to consider: (I) There exists a feasible path (within the specified depth bound) that contains a sequence of impacted nodes, which DiSE does not explore, and (II) DiSE explores more than one feasible execution path for some sequence of impacted nodes (within the specified depth bound).

Case I. Let $q := \langle n_1, \ldots, n_k\rangle$ be a sequence of impacted 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 impacted node. Assume n_i is the first node in q such that DiSE explores a feasible path, p, that contains the subsequence $\langle n_1, \ldots, n_{i-1}\rangle$ but does not explore an execution path that contains the subsequence $\langle n_1, \ldots, n_i\rangle$. 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 impacted node, n_i will not be found in the intersection of *Imp* and *Explored* (line 12 in Figure 7). Hence, DiSE will explore a path that contains the subsequence $\langle n_1, \ldots, n_i\rangle$. Contradiction.

Case II. Assume for a sequence of impacted nodes that lie on path p explored by DiSE, it explores another path p' containing the same sequence of impacted nodes. Let n be the last affected node on path p such that the p and p' have the exact same subsequence of impacted and unimpacted nodes up to and including n. Let $q := \langle n, n_1 \cdots n_k, m\rangle$ be the subsequence of nodes on p such that each n_i is an unimpacted node and m is an impacted node. Let $q' := \langle n, n'_1 \cdots n'_j, m\rangle$ be the corresponding subsequence of nodes on p'. By the construction of the algorithm in Figure 7, when DiSE considers the impacted node n, it only explores one path and prunes the others by controlling the *Explored* set in Figure 7 until the next impacted node, which in this case is m. Hence, q and q' are identical. Contradiction.

In recent work, we show that exploring distinct sequences of impacted statements is sufficient to prove bounded functional equivalence of related program versions [Backes et al. 2013b].

5. EVALUATION

We first evaluate the cost and effectiveness of DiSE relative to full symbolic execution by considering two 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 impacted path conditions generated by DiSE compare with the number of path conditions generated by full symbolic execution?

We then evaluate the impact of two factors on the DiSE algorithm by considering two additional research questions.

RQ3. How does the exploration order of the depth-first search impact DiSE results?

RQ4. How are the characteristics of the program changes related to the effectiveness of DiSE?

5.1. Tool Support

We implemented DiSE in Symbolic PathFinder (SPF) [Păsăreanu et al. 2008; Păsăreanu and Rungta 2010], a symbolic execution extension to the Java PathFinder framework [Visser et al. 2003]. DiSE extends SPF by using a customized listener to (1) load the results of the abstract syntax tree (AST) diff (the change sets), (2) invoke the static data and control dependence analyses to compute the set of impacted program locations, and (3) direct symbolic execution in SPF using the impact set computed by the static analyses.

5.2. Artifacts

To evaluate DiSE, we chose four Java applications. The first program, the Altitude Switch (ASW) application, is a synchronous reactive component from the avionics domain. It was developed as a Simulink model, and was automatically translated to Java using tools developed at Vanderbilt University [Sztipanovits and Karsai 2002]. We inlined all methods involved in symbolic execution into one method, which has 346 source lines of code, to evaluate DiSE.

The second program, the Wheel Brake System (WBS), is a synchronous reactive component derived from the WBS case example found in ARP 4761 [SAE-ARP4761 1996; Joshi and Heimdahl 2005]. The WBS is used to provide safe breaking of the aircraft during taxi, landing, and in the event of a rejected take-off. We use the `update(int PedalPos, boolean AutoBrake, boolean Skid)` method in WBS to evaluate DiSE. 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.

The third program, Traffic Anti-Collision Avoidance System (TCAS), is a system to avoid air collisions available from the Software-artifact Infrastructure Repository

(SIR).[2] We manually converted the C program versions to Java and fully in-lined the methods. The Java version has approximately 150 SLOC.

The fourth program, the Apollo Lunar Autopilot, is a Simulink model that was automatically translated to Java. The translated Java code has 2.6 KLOC in 54 classes. The model is available from MathWorks6. It contains both Simulink blocks and Stateflow diagrams and makes use of complex Math functions (e.g., Math.sqrt). All methods involved in symbolic execution were inlined into one method for the purpose of applying DiSE. All of the method inputs for the artifacts studied have primitive types.

To evaluate DiSE, we require multiple versions for each artifact. There are several versions of the `TCAS` example available in the SIR repository. We created mutants manually and using a mutation tool for the other three artifacts. The mutants generated were of the base version (v0) of the method under analysis.

When creating mutants, we considered a broad range of changes including location, type and number of changes. We also considered the control structures in the code, and made changes at various depths in nested control structures. Each mutant has one, two or three changed Java statements, resulting in up to 36 changed CFG nodes as shown in Table II. Versions with multiple changes were created by combining the individual mutations made to versions with a single change. Each change involved the addition, removal or modification of a statement. Control statements were modified by mutating the comparison operator, for example, from $<$ to $<=$, or the operand, for example, mutating the program variables involved in the comparison. Noncontrol statements were modified by changing the value assigned to a program variable. We omit results for mutants (1) with no changed CFG nodes, for example, where the compiler optimizes out the change or the AST diff is able to determine the versions are equivalent, (2) that contain features unsupported by SPF, and (3) that contain changes to only global data (since DiSE as presented here only computes the impact of changes to methods).

We apply muJava [Ma et al. 2005] to `ASW` and `WBS` in order to automatically generate mutants. We applied all 15 method-level mutation operators provided by muJava. The number of mutants generated for each mutation operator varies considerably and some mutation operators did not result in corresponding mutants. For `ASW`, muJava generated 607 mutants. For each mutation operator that generated mutants, we randomly selected three mutants. Thus, we selected 24 mutants in total. For `WBS`, muJava generated 568 mutants, of which we similarly selected 24 mutants in total.

None of the mutants generated either by hand or by applying muJava in our study have unbounded loops.

5.3. Variables, Measures, and Other Factors

Independent Variables. The *independent* variable in our study is the symbolic execution algorithm used in our empirical study. To study RQ1 and RQ2, we use the DiSE algorithm and compare it with full (traditional) symbolic execution as implemented in the SPF framework. To study RQ3, we change the exploration order of successor states in the depth-first search. We compare the default, random, and greedy exploration orders and their impact on the efficiency of DiSE.

Dependent Variables. For the study of RQ1 and RQ2, we selected three dependent variables: (1) *time*, (2) *states explored*, (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/or performing extra analysis for heuristics, and the time spent performing symbolic execution. The States explored variable is a count of the number of symbolic states generated during symbolic execution. The

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

Table II. Results of DiSE and Symbolic Execution on Hand-Coded Mutants

	CFG Nodes			Time (ss)		States Explored			Path Conditions		
Ver.	Total	Changed	Affected	DiSE	Full	DiSE	Full	Ratio	DiSE	Full	Ratio
v1	126	0* (0.0%)	0 (0.0%)	4	<1	13	2299	176.9	1	576	576.0
v2	126	1 (0.8%)	2 (1.6%)	4	<1	91	2299	25.3	8	576	72.0
v3	126	1 (0.8%)	17 (13.5%)	4	<1	18	2299	127.7	2	576	288.0
v4	126	4 (3.2%)	18 (14.3%)	4	<1	20	2299	115.0	2	576	288.0
v5	126	4 (3.2%)	18 (14.3%)	4	<1	20	2299	115.0	2	576	288.0
v6	126	1 (0.8%)	27 (21.4%)	5	<1	2299	2299	1.0	576	576	1.0
v7	126	5 (4.0%)	12 (9.5%)	4	<1	24	2299	95.8	3	576	192.0
v8	126	4 (3.2%)	12 (9.5%)	4	<1	30	2299	76.6	4	576	144.0
v9	126	4 (3.2%)	3 (2.4%)	5	<1	13	2299	176.9	1	576	576.0
v10	123	1 (0.8%)	13 (10.6%)	7	<1	1726	2299	1.3	289	576	2.0
v11	126	0* (0.0%)	0 (0.0%)	4	<1	13	2299	176.9	1	576	576.0
v12	129	4 (3.1%)	3 (2.3%)	5	<1	13	2299	176.9	1	576	576.0
v13	126	0* (0.0%)	0 (0.0%)	4	<1	13	2299	176.9	1	576	576.0
v14	126	9 (7.1%)	18 (14.3%)	4	<1	26	2299	88.4	3	576	192.0
v15	126	5 (4.0%)	31 (24.6%)	5	<1	2299	2299	1.0	576	576	1.0

(a) ASW Example

	CFG Nodes			Time (ss)		States Explored			Path Conditions		
Ver.	Total	Changed	Affected	DiSE	Full	DiSE	Full	Ratio	DiSE	Full	Ratio
v1	100	1 (1.0%)	43 (43.0%)	3	<1	47	47	1.0	24	24	1.0
v2	100	1 (1.0%)	12 (12.0%)	4	<1	39	47	1.2	18	24	1.3
v3	100	1 (1.0%)	4 (4.0%)	2	<1	35	47	1.3	12	24	2.0
v4	100	0 (0.0%)	0 (0.0%)	2	<1	4	47	11.8	1	24	24.0
v5	100	7 (7.0%)	63 (63.0%)	2	<1	47	47	1.0	24	24	1.0
v6	100	1 (1.0%)	2 (2.0%)	2	<1	8	47	5.9	2	24	12.0
v7	100	1 (1.0%)	39 (39.0%)	2	<1	47	47	1.0	24	24	1.0
v8	100	8 (8.0%)	64 (64.0%)	2	<1	47	47	1.0	24	24	1.0
v9	100	2 (2.0%)	6 (6.0%)	2	<1	35	47	1.3	12	24	2.0
v10	100	1 (1.0%)	39 (39.0%)	2	<1	47	47	1.0	24	24	1.0
v11	100	7 (7.0%)	63 (63.0%)	2	<1	47	47	1.0	24	24	1.0
v12	100	8 (8.0%)	68 (68.0%)	2	<1	47	47	1.0	24	24	1.0
v13	100	9 (9.0%)	64 (64.0%)	2	<1	47	47	1.0	24	24	1.0
v14	100	3 (3.0%)	39 (39.0%)	2	<1	47	47	1.0	24	24	1.0
v15	100	3 (3.0%)	42 (42.0%)	3	<1	47	47	1.0	24	24	1.0
v16	100	8 (8.0%)	63 (63.0%)	2	<1	47	47	1.0	24	24	1.0

(b) WBS Example

	CFG Nodes			Time (ss)		States Explored			Path Conditions		
Ver.	Total	Changed	Affected	DiSE	Full	DiSE	Full	Ratio	DiSE	Full	Ratio
v1	73	6 (8.2%)	14 (19.2%)	38	126	330	695	2.1	23	76	3.3
v2	73	14 (19.2%)	32 (43.8%)	120	125	679	679	1.0	68	68	1.0
v3	73	5 (6.9%)	35 (47.9%)	154	157	837	837	1.0	79	79	1.0
v4	73	6 (8.2%)	14 (19.2%)	37	137	353	743	2.1	22	84	3.8
v5	73	5 (6.9%)	38 (52.8%)	148	150	763	763	1.0	94	94	1.0
v6	73	17 (23.3%)	23 (31.5%)	124	127	679	679	1.0	68	68	1.0
v9	73	7 (9.6%)	20 (27.4%)	33	142	329	823	2.5	23	108	4.7

(Continued)

Table II. Continued

	CFG Nodes					Time (ss)		States Explored			Path Conditions		
Ver.	Total	Changed		Affected		DiSE	Full	DiSE	Full	Ratio	DiSE	Full	Ratio
v10	73	29	(39.7%)	29	(39.7%)	135	129	743	743	1.0	84	84	1.0
v11	70	36	(51.4%)	28	(40.0%)	129	130	743	743	1.0	84	84	1.0
v12	73	6	(8.2%)	39	(53.4%)	265	275	1407	1407	1.0	144	144	1.0
v20	73	7	(9.6%)	20	(27.4%)	34	144	329	823	2.5	23	108	4.7
v21	70	4	(5.7%)	19	(27.1%)	26	123	225	615	2.7	24	88	3.7
v22	70	4	(5.7%)	19	(27.1%)	24	108	245	591	2.4	23	88	3.8
v23	70	4	(5.7%)	18	(25.7%)	119	119	631	631	1.0	88	88	1.0
v24	70	4	(5.7%)	18	(25.7%)	131	129	655	655	1.0	88	88	1.0
v25	73	6	(8.2%)	14	(19.2%)	127	129	679	679	1.0	76	76	1.0
v26	72	5	(6.9%)	38	(52.8%)	119	125	667	667	1.0	62	62	1.0
v27	72	5	(6.9%)	38	(52.8%)	148	151	763	763	1.0	94	94	1.0
v28	73	7	(9.6%)	20	(27.4%)	36	123	363	799	2.2	23	108	4.7
v29	70	4	(5.7%)	19	(27.1%)	25	107	245	591	2.4	23	88	3.8
v30	70	4	(5.7%)	19	(27.1%)	26	120	225	615	2.7	24	88	3.7
v31	69	1	(1.4%)	10	(14.5%)	6	105	55	631	11.5	5	68	13.6
v32	69	1	(1.4%)	10	(14.5%)	16	105	91	631	6.9	8	68	8.5
v34	73	4	(5.5%)	36	(49.3%)	458	457	2317	2317	1.0	239	239	1.0
v35	73	7	(9.6%)	20	(27.4%)	34	127	349	799	2.3	22	108	4.9
v37	73	24	(32.9%)	29	(39.7%)	121	118	679	679	1.0	68	68	1.0
v39	73	6	(8.2%)	14	(19.2%)	124	121	679	679	1.0	76	76	1.0
v40	68	6	(8.8%)	12	(17.6%)	6	97	54	583	10.8	5	76	15.2
v41	72	5	(6.9%)	13	(18.1%)	3	112	309	695	2.3	22	84	3.8

(c) TCAS Example

number of path conditions generated provides a count of the number of program execution paths generated by a given technique. The *time* and *states explored* variables 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).

For the study of RQ3, we selected four additional dependent variables and measures: (1) *prunings*, (2) *infeasible paths*, (3) *terms*, and (4) *decision procedure calls*. The prunings variable counts the number of times DiSE is able to prune a path during symbolic execution. The infeasible paths variable contains a count of the number of times an infeasible path was encountered during the search.

The terms variable records the total number of terms (constraints) in the path conditions. It includes two submeasures: (1) *total terms*—the sum of all the terms that appear on all of the feasible paths explored, and (2) *average term*—the arithmetic mean of *total terms*. The Decision procedure (DP) calls variable provides a count of the number of decision procedure calls made during symbolic execution. It includes four submeasures: *minimum decision procedure calls*, *maximum decision procedure calls*, *total decision procedure calls*, and *average decision procedure calls*. Note that total decision procedure calls is the sum of the decision procedure calls made across all the paths generated during symbolic execution. All four variables are used to evaluate the effectiveness of DiSE using different search strategies (RQ3).

Other Factors. To study RQ4, we check if there is a possible co-relation between the following factors: *CFG nodes* and *nature of the changes*. The CFG nodes factor includes two submeasures: *changed CFG nodes* and *affected CFG nodes*. The Nature of the changes factor provides the characteristics of the actual change to the source code.

It includes the type of statement modified and the location of the change. For nested conditionals and nested loops, changes are also categorized as outermost changes (*om*) and innermost changes (*im*).

5.4. Experiment Setup

Our study was performed on a Dell Desktop running at 2.8 GHz Intel Core i7 CPU with 8 GB of memory and running Windows 7 Professional. The artifacts are compiled with Java version 1.6. We used a custom Java application[3] based on an AST comparison to compute the initial change set for each mutant of the base version of the program. We then used the change set as input to DiSE to analyze each mutant version. We also performed standard symbolic execution on each mutant using SPF. As none of the artifacts used in our study have unbounded loops, there is no need to specify a depth bound for symbolic execution of these artifacts.

When performing the DiSE analysis using random exploration order, we ran DiSE 10 times on each program version using different seeds, and computed the arithmetic mean of the results from the 10 runs.

5.5. Threats to Validity

The primary threats to *external validity* for our study are (1) the use of SPF to implement our technique, (2) the use of Choco and Coral for solving linear and nonlinear constraints, (3) the selection of artifacts used to evaluate DiSE, and (4) the changes applied to create the mutants. To mitigate this threat, we have implemented the DiSE algorithm on the LLVM platform to analyze evolving C programs. This analysis framework, Proteus, implements the static impact analysis as an LLVM optimization pass, the incremental symbolic execution as an extension to the KLEE symbolic execution engine, and uses STP as the constraint solver [Backes et al. 2013a]. The results generated by Proteus when analyzing C programs with DiSE are similar to the ones presented at http://bit.ly/MIsCal.

The artifacts selected for our study are control applications that are amenable to symbolic execution. The object programs have previously been used to evaluate symbolic execution techniques [Staats and Păsăreanu 2010; Souza et al. 2011; Person et al. 2011; Yang et al. 2012]. Some of the mutant versions used in our study are 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. We also controlled for this threat by using existing program versions and mutants automatically created by a mutation tool, muJava. 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 could 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.

5.6. Results and Analysis

In this section, we present the results of our experiments, and analyze the results with respect to our four research questions.

[3]The AST diff tool was developed at the University of Nebraska-Lincoln.

RQ1: How does the cost of applying DiSE compare to full symbolic execution on the changed method? In Tables II(a)–(c), we list the results of running DiSE and full symbolic execution on each hand-coded version of the three Java artifacts ASW, WBS, and TCAS. For each mutant version, we list the number of CFG nodes (basic blocks at the bytecode level) changed (*Changed*), the number of CFG nodes affected by the changes (*Affected*), and the metrics described in Section 5.3, which are the time to perform DiSE and the time to perform full symbolic execution as reported by SPF. We also list the ratios of full symbolic execution results to DiSE results for states explored and path conditions. A ratio value of 1.0, for example, v6 in Table II(a), indicates that DiSE and symbolic execution explored the same space. A ratio greater than 1.0, for example, v1 in Table II(a), indicate that DiSE explores many few states or generates many fewer path conditions compared to full symbolic execution.

In Tables II(a)–(c), we can see that DiSE takes between four and seven seconds to perform the static analysis and symbolic execution on the ASW mutant versions, and between two and four seconds for the WBS mutant versions. The total time taken by DiSE is greater than full symbolic execution for the ASW and WBS artifacts, however, the DiSE analysis time for the TCAS artifact is either less or very close to the analysis time for full symbolic execution. In many versions, the reduction achieved by DiSE is considerable. For example, in v41, the reduction is more than an order of magnitude. In cases where DiSE explores the same number of states as full symbolic execution, the total time taken by DiSE is a little more than symbolic execution, for example, versions v10 and v24. This extra execution time accounts for the overhead of computing the affected locations and supporting data structures.

There is considerable variation in the number of states explored by DiSE for all the mutant versions in Tables II(a)–(c). In eight out of 15 versions of ASW, DiSE explores 100*X fewer states* compared to full symbolic execution (2299). Only for two versions, DiSE and full symbolic execution generate the same number of states. In contrast, for the majority of the WBS and TCAS mutant versions, DiSE and full symbolic execution explore the same number of states. The full state space for the WBS mutants is 47 states, and for the TCAS mutants the full state space ranges from 591 to 2317 states.

In Tables III(a)–(b), we list the results of running DiSE and full symbolic execution on the auto-generated mutants for ASW and WBS. The column labels for these tables are the same as in Tables II(a)–(c). For both artifacts, the results in Tables III(a)–(b) and in Tables II(a)–(c) are similar with respect to the cost of DiSE versus full symbolic execution.

RQ2: How does the number of affected path conditions generated by DiSE compare with the number of path conditions generated by full symbolic execution? The number of path conditions computed by a given technique is a measure of its efficiency; fewer path conditions means less work for the subsequent client analysis that uses the DiSE results. In Tables II(a)–(c), we see that ratio of path conditions computed by DiSE varies considerably between versions for all three examples. The number of path conditions computed by symbolic execution is up to 576 times more than the number of path conditions computed by DiSE. For only two versions of the ASW artifact, DiSE computes the same number of the path conditions as full symbolic execution (*Ratio* $= 1.0$). For these examples, DiSE cannot safely prune the execution space and must explore the same paths as full symbolic execution. However, for the WBS and TCAS artifacts, there are many more versions for which DiSE and full symbolic execution compute the same number of path conditions. In other words, the reductions in the number of path conditions in the ASW model is much more pronounced compared to the TCAS and WBS artifacts.

The results for automatically generated mutants (Tables III(a)–(b)) show a similar trend to the results for manually generated mutants. DiSE achieved significant

Table III. Results of DiSE and Symbolic Execution on Mutants Automatically Generated

	CFG Nodes					Time (ss)		States Explored			Path Conditions		
Ver.	Total	Changed		Affected		DiSE	Full	DiSE	Full	Ratio	DiSE	Full	Ratio
v16	126	1	(0.8%)	25	(19.8%)	5	<1	1726	2299	1.3	289	576	1.99
v18	126	1	(0.8%)	12	(9.5%)	6	<1	43	43	1	8	8	1
v19	126	1	(0.8%)	16	(12.7%)	4	<1	18	2299	127.7	2	576	288.0
v20	126	1	(0.8%)	16	(12.7%)	4	<1	18	2299	127.7	2	576	288.0
v21	126	1	(0.8%)	16	(12.7%)	4	<1	18	2299	127.7	2	576	288.0
v22	126	1	(0.8%)	16	(12.7%)	5	<1	18	2299	127.7	2	576	288.0
v23	126	1	(0.8%)	25	(19.8%)	4	<1	2299	2299	1	576	576	1.0
v24	126	4	(3.2%)	14	(11.1%)	4	<1	18	2299	127.7	2	576	288.0
v25	126	4	(3.2%)	18	(14.3%)	4	<1	20	2299	115.0	2	576	288.0
v26	125	0	(0.0%)	0	(0.0%)	4	<1	13	2299	176.9	1	576	576.0
v27	126	4	(3.2%)	3	(2.4%)	4	<1	13	2299	176.9	1	576	576.0
v28	122	1	(0.8%)	10	(8.2%)	4	<1	18	2299	127.7	2	576	288.0
v29	122	1	(0.8%)	3	(2.5%)	4	<1	13	2299	176.9	1	576	576.0
v30	126	5	(4.0%)	4	(3.2%)	5	<1	2299	2299	1	576	576	1.0
v31	123	1	(0.8%)	4	(3.3%)	4	<1	13	2299	176.9	1	576	576.0
v32	123	1	(0.8%)	3	(2.4%)	4	<1	16	2299	143.7	1	576	576.0
v33	123	1	(0.8%)	4	(3.3%)	4	<1	15	2299	153.3	2	576	288.0
v34	126	4	(3.2%)	14	(11.1%)	5	<1	715	2299	3.2	144	576	4.0
v35	129	4	(3.2%)	8	(6.2%)	5	<1	22	2299	104.5	3	576	192.0
v36	129	4	(3.2%)	13	(10.1%)	6	<1	21	2299	109.5	3	576	192.0
v37	126	4	(3.2%)	18	(14.3%)	4	<1	20	2299	115.0	2	576	288.0
v38	126	4	(3.2%)	27	(21.4%)	5	<1	2299	2299	1	576	576	1.0
v39	126	1	(0.8%)	2	(1.6%)	4	<1	13	2299	176.9	1	576	576.0

(a) ASW Example

	CFG Nodes					Time (ss)		States Explored			Path Conditions		
Ver.	Total	Changed		Affected		DiSE	Full	DiSE	Full	Ratio	DiSE	Full	Ratio
v17	100	1	(1.0%)	2	(2.0%)	2	<1	35	47	1.3	12	24	2.0
v18	100	1	(1.0%)	2	(2.0%)	2	<1	35	47	1.3	12	24	2.0
v19	100	1	(1.0%)	2	(2.0%)	2	<1	12	47	3.9	5	24	4.8
v20	100	1	(1.0%)	63	(63.0%)	2	<1	47	47	1	24	24	1.0
v21	100	1	(1.0%)	2	(2.0%)	2	<1	28	47	1.7	13	24	1.9
v22	100	1	(1.0%)	1	(1.0%)	2	<1	4	47	11.8	1	24	24.0
v23	100	1	(1.0%)	2	(2.0%)	2	<1	35	47	1.3	12	24	2.0
v24	100	2	(2.0%)	35	(35.0%)	3	<1	47	47	1	24	24	1.0
v25	100	1	(1.0%)	2	(2.0%)	2	<1	28	47	1.7	13	24	1.9
v26	100	1	(1.0%)	40	(40.0%)	2	<1	47	47	1	24	24	1.0
v27	100	1	(1.0%)	9	(9.0%)	2	<1	35	47	1.3	12	24	2.0
v28	100	1	(1.0%)	14	(14.0%)	2	<1	31	47	1.5	12	24	2.0
v29	100	9	(9.0%)	63	(63.0%)	2	<1	47	47	1	24	24	1.0
v30	105	7	(7.0%)	38	(38.0%)	2	<1	47	47	1	24	24	1.0
v31	100	2	(2.0%)	31	(31.0%)	2	<1	47	47	1	24	24	1.0
v32	100	7	(7.0%)	63	(63.0%)	2	<1	47	47	1	24	24	1.0
v33	100	7	(7.0%)	63	(63.0%)	2	<1	95	95	1	24	24	1.0
v34	100	3	(3.0%)	52	(52.0%)	2	<1	47	47	1	24	24	1.0
v35	100	1	(1.0%)	10	(10.0%)	2	<1	35	47	1.3	12	24	2.0
v36	100	1	(1.0%)	7	(7.0%)	2	<1	35	47	1.3	12	24	2.0
v37	100	2	(2.0%)	33	(33.0%)	2	<1	47	47	1	24	24	1.0
v38	100	1	(1.0%)	2	(2.0%)	2	<1	4	47	11.8	1	24	24.0
v39	100	2	(2.0%)	31	(31.0%)	2	<1	47	47	1	24	24	1.0
v40	100	1	(1.0%)	13	(13.0%)	2	<1	35	47	1.3	15	24	1.6

(b) WBS Example

reductions for most versions of ASW; for WBS, it achieved small reductions on about half of the mutant versions.

RQ3: How does the symbolic execution search strategy impact DiSE results? We performed DiSE using three exploration orders for choosing the successor state during symbolic execution. We evaluated both the hand-coded and auto generated mutants for all three Java artifacts ASW, WBS, and TCAS. Table IV, Table V, and Table VI show the results of eight representative versions for each artifact; the complete set of results is available in Appendix A. In each table, we list the exploration strategy applied (*Strategy*), and the cost in terms of time (*Time*) and number of states explored (*States*). The efficiency of each approach is measured in terms of number of path conditions generated (*PC*) and number of prunings performed by DiSE (*Prunings*). We also list the number of infeasible paths encountered during the search (*Infeasible*). For each feasible path explored, we list the number of terms in the corresponding path condition, and the number of decision procedure calls made during exploration of a path. For terms, we list the total (Total Terms) and the average (Avg. Terms). For decision procedure calls, we list the minimum (Min DP), maximum (Max DP), total (Total DP), and average (Avg. DP).

Table IV shows the results for ASW. We find that for all versions except v6, the time cost for DiSE using the default exploration strategy is the same as DiSE using a greedy exploration strategy. Moreover, the time cost for the random exploration strategy is the same or slightly more than the default exploration strategy. For two versions, v6 and v38, the number of states generated and the number of path conditions generated is the same for all exploration strategies. For the other versions, the default and greedy exploration strategies explore the same number of states and generate the same number of path conditions; however, the random exploration often computes the fewest number of path conditions although it may explore the same, fewer, or more states.

It is interesting to note that DiSE with a random exploration strategy generated only 0.1 path conditions on average, while the other two exploration strategies generated exactly one path condition. The reason for this is that our implementation of DiSE prunes paths during backtracking. Thus, even when no CFG nodes are marked as affected, a single path condition will be generated if the selected path is feasible, otherwise, no path condition is generated when the first path is infeasible. For example in v1, DiSE with the default and greedy exploration strategies explored the first path in the search which is feasible, while in some runs of DiSE using the random exploration strategy, the first path explored was infeasible and thus no path condition was generated.

Table V shows the results for WBS. For all metrics except Time and States, the cost and efficiency of DiSE using the greedy and default exploration strategies is the same, and Time and States vary just slightly for only two versions, v2 and v6. The results for DiSE with random exploration strategy are mostly similar to the other exploration strategies; however, DiSE with random exploration performed slightly better for some versions, for example, v2 and v21 in terms of path conditions (and total terms) generated, and slightly worse for other versions, for example, v19.

Table VI shows the results for the TCAS artifact. Similar to the other artifacts, the default and greedy exploration strategies tend to be most similar; however, in version v31 the greedy and random exploration strategy results are more similar. Overall, the TCAS results show more variability in general, across exploration strategies, although we see for versions v2 and v41, the results are nearly the same for all exploration strategies.

We note that for all versions of the ASW and WBS artifacts, DiSE using the default and greedy exploration strategies achieves the same results except for the time metric.

Table IV. Results of DiSE on ASW using Different Search Strategies

Ver.	Strategy	Time (ss)	States	PC	Prunings	Infeasible	Total Terms	Avg. Terms	Min DP	Max DP	Total DP	Avg. DP
v1	default	4	13	1	0	12	7	7.0	12	12	12	12.0
v1	greedy	4	13	1	0	12	7	7.0	12	12	12	12.0
v1	random	4	6.4	0.1	0	5.4	0.7	0.7	1.2	1.2	1.2	1.2
v2	default	4	91	8	14	48	56	7.0	12	12	96	12.0
v2	greedy	4	91	8	14	48	56	7.0	12	12	96	12.0
v2	random	4	91	4.2	17.8	48	29.4	7.0	12	12	50.4	12.0
v3	default	4	18	2	0	16	14	7.0	12	12	24	12.0
v3	greedy	4	18	2	0	16	14	7.0	12	12	24	12.0
v3	random	4	27.4	2.2	2.8	21.6	15.4	6.3	10.8	10.8	26.4	10.8
v6	default	5	2299	576	590	0	4032	7.0	12	12	6912	12.0
v6	greedy	7	2299	576	590	0	4032	7.0	12	12	6912	12.0
v6	random	5.4	2299	576	590	0	4032	7.0	12	12	6912	12.0
v8	default	4	30	4	0	26	28	7.0	12	12	48	12.0
v8	greedy	4	30	4	0	26	28	7.0	12	12	48	12.0
v8	random	4	36.6	2	4.7	28	14	6.3	10.8	10.8	24	10.8
v28	default	4	18	2	0	16	14	7.0	12	12	24	12.0
v28	greedy	4	18	2	0	16	14	7.0	12	12	24	12.0
v28	random	4	21.3	1.3	2.2	16.8	9.1	5.6	9.6	9.6	15.6	9.6
v33	default	4	15	2	0	12	14	7.0	12	12	24	12.0
v33	greedy	4	15	2	0	12	14	7.0	12	12	24	12.0
v33	random	4.2	17.1	1.4	2.7	12	9.8	7.0	12	12	16.8	12.0
v35	default	5	22	3	0	18	21	7.0	12	12	36	12.0
v35	greedy	5	22	3	0	18	21	7.0	12	12	36	12.0
v35	random	5.1	18.9	0.9	1.5	14.8	6.3	4.2	7.2	7.2	10.8	7.2
v38	default	5	2299	576	590	0	4032	7.0	12	12	6912	12.0
v38	greedy	5	2299	576	590	0	4032	7.0	12	12	6912	12.0
v38	random	5.2	2299	576	590	0	4032	7.0	12	12	6912	12.0

And, for TCAS, the differences in the results of using these two exploration strategies are small. To understand this observation, we examined some of the versions, and found that for many cases, the default exploration strategy was the same as the greedy exploration strategy. Moreover, even when the exploration strategy is altered, the greedy algorithm does not necessarily render fewer states or fewer path conditions since the distance in our implementation is measured by the number of CFG edges, and it is unknown which edges correspond to symbolic branches and which correspond to concrete branches.

Although the performance of DiSE varies for many versions when different exploration strategies are applied, the differences are not substantial, and overall the performance of DiSE does not appear to be sensitive to the symbolic execution exploration strategy used.

RQ4: How are the characteristics of the program changes and the effectiveness of DiSE related? To answer this research question we first compare the number of *changed* and *affected* CFG nodes with the number of path conditions computed by DiSE. On one hand, we see in Table II(a) that DiSE explores all paths for only two versions (v6 and v15) of the ASW example. Yet, for version v14 which has more changed CFG nodes than any other version (9), DiSE is able to compute many fewer path conditions (3) versus full symbolic execution (576) for a ratio of 192.0. On the other hand, the changes made to 11 of the 16 versions of the WBS example, shown in Table II(b), require DiSE to

Table V. Results of DiSE on WBS using Different Search Strategies

Ver.	Strategy	Time (ss)	States	PC	Prunings	Infeasible	Total Terms	Avg. Terms	Min DP	Max DP	Total DP	Avg. DP
v1	default	3	47	24	0	0	72	3.0	3	7	128	5.3
v1	greedy	3	47	24	0	0	72	3.0	3	7	128	5.3
v1	random	2.2	47	24	0	0	72	3.0	3	7	128	5.3
v2	default	4	39	18	0	4	54	3.0	3	7	86	4.8
v2	greedy	3	39	18	0	4	54	3.0	3	7	86	4.8
v2	random	2.2	33	13.5	0	7	40.5	3.0	3	7	69.8	5.2
v4	default	2	4	1	0	3	3	3.0	3	3	3	3.0
v4	greedy	2	4	1	0	3	3	3.0	3	3	3	3.0
v4	random	2	4.5	1	0	3.5	3	3.0	3.5	3.5	3.5	3.5
v6	default	2	8	2	0	5	6	3.0	3	4	7	3.5
v6	greedy	4	8	2	0	5	6	3.0	3	4	7	3.5
v6	random	2.1	9.1	2	0	6.1	6	3.0	3	5.1	8.1	4.1
v9	default	2	35	12	0	12	36	3.0	3	7	64	5.3
v9	greedy	2	35	12	0	12	36	3.0	3	7	64	5.3
v9	random	2.1	35	12	0	12	36	3.0	3	7	64	5.3
v19	default	2	12	5	0	4	15	3.0	3	4	16	3.2
v19	greedy	2	12	5	0	4	15	3.0	3	4	16	3.2
v19	random	2.5	27.8	13	0	4	39	3.0	3.4	5.8	71.2	4.9
v21	default	2	28	13	0	6	39	3.0	3	6	54	4.2
v21	greedy	2	28	13	0	6	39	3.0	3	6	54	4.2
v21	random	2.2	24.4	10.6	0	6	31.8	3.0	4.1	6.6	59.2	5.4
v33	default	2	95	24	24	0	72	3.0	4	8	152	6.3
v33	greedy	2	95	24	24	0	72	3.0	4	8	152	6.3
v33	random	2.1	95	24	24	0	72	3.0	4	8	152	6.3

explore the same paths as full symbolic execution. For these 11 versions, the number of changed CFG nodes ranges from one to nine (out of 100 CFG nodes), and the number of affected CFG nodes ranges from 39 to 68. Furthermore, in Table II(c), we see that for the TCAS example, versions v25 and v40 have similar numbers of changed and affected CFG nodes, yet the DiSE analysis of V25 generates as many path conditions as full symbolic execution, while full symbolic execution on v40 generates 15.20 times as many path conditions as DiSE. Based on our analysis of the hand-coded mutants for these examples, there does not appear to be any correlation between the number or percentage of affected nodes and the number of impacted path conditions.

In Figure 8, we show the relationship between the percentage of changed CFG nodes (out of the total number of CFG nodes) and the percentage of path conditions generated by DiSE (relative to the number of path conditions generated by full symbolic execution) for the results of all hand-coded and auto-generated mutants, shown in Tables II(a)–(c) and Tables III(a)–(b). From this data, we can see there does not appear to be any significant correlation between the number of changed nodes and the number of path conditions generated. For example, when the percentage of changed CFG nodes is small (0% to 5%), the percentage of impacted path conditions ranges from 0% to 100%.

We also compare the percentage of affected CFG nodes (out of the total number of CFG nodes) with the percentage of path conditions generated by DiSE (relative to the number of path conditions generated by full symbolic execution) in Figure 9. Again, we see a similar pattern and find no significant correlation between these factors.

To better understand how code changes can impact the effectiveness of DiSE, we conducted another case study, on the Apollo example. For this study, we made changes to different types of program statements as well as at different program locations.

Table VI. Results of DiSE on TCAS using Different Search Strategies

Ver.	Strategy	Time (ss)	States	PC	Prunings	Infeasible	Total Terms	Avg. Terms	Min DP	Max DP	Total DP	Avg. DP
v2	default	120	679	68	272	0	611	9.0	3	18	835	12.3
v2	greedy	127	679	68	272	0	611	9.0	3	18	835	12.3
v2	random	122.6	679	68	272	0	611	9.0	3	18	835	12.3
v4	default	37	353	22	100	111	117	5.3	3	17	129	5.9
v4	greedy	37	349	22	100	109	117	5.3	3	17	125	5.7
v4	random	40.6	343.4	25.7	96	102.9	155.3	6.0	3	16	178.3	6.9
v10	default	135	743	84	288	0	783	9.3	3	18	1079	12.9
v10	greedy	128	743	84	288	0	783	9.3	3	18	1079	12.9
v10	random	128.4	743	84	288	0	783	9.3	3	18	1079	12.9
v20	default	34	329	23	93	99	128	5.6	3	17	145	6.3
v20	greedy	34	330	24	94	98	139	5.8	3	17	158	6.6
v20	random	40.6	348.5	28.6	98.9	96.5	187.5	6.5	3	17.6	224.2	7.8
v25	default	127	679	76	264	0	697	9.2	3	18	969	12.8
v25	greedy	123	679	76	264	0	697	9.2	3	18	969	12.8
v25	random	110.2	611.1	68.4	237.6	0	627.3	8.3	2.7	16.2	872.1	11.5
v30	default	26	225	24	52	75	139	5.8	3	16	158	6.6
v30	greedy	27	226	25	53	74	150	6.0	3	16	170	6.8
v30	random	29.9	239.2	29.3	56.3	70.8	194.2	6.6	3	16.1	225.3	7.7
v31	default	6	55	5	14	25	50	10.0	6	17	72	14.4
v31	greedy	11	86	18	20	26	110	6.1	3	17	130	7.2
v31	random	7.6	83.9	17.7	17.6	27.6	111	6.4	3.3	17	131.4	7.6
v41	default	33	309	22	84	99	117	5.3	3	17	129	5.9
v41	greedy	31	305	22	84	97	117	5.3	3	17	125	5.7
v41	random	31	305	22	84	97	117	5.3	3	17	125	5.7

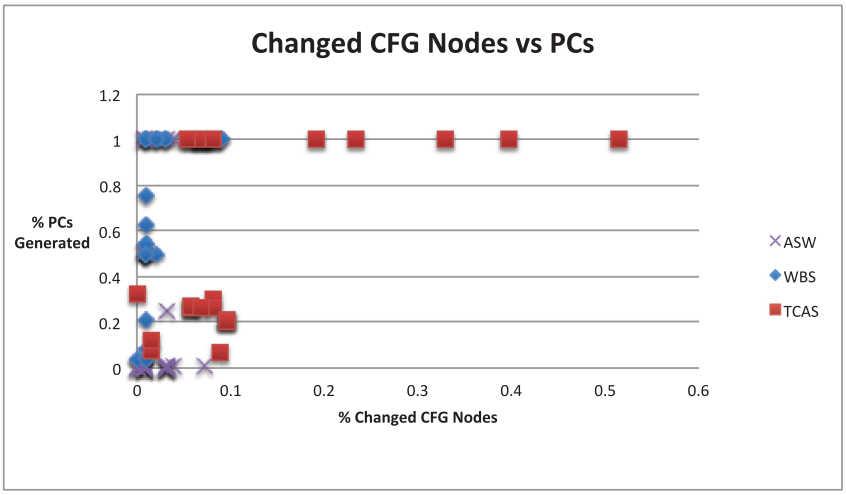


Fig. 8. Correlation between changed CFG nodes and PCs generated.

The results are shown in Table VII. The Change column shows the change type for each mutant version. The changes were made at regular statements (reg), nested conditionals (nc), and nested loops (nl). They were also made in varying locations: at the top (t), bottom (b), and middle (m) of the method. For nested conditionals and

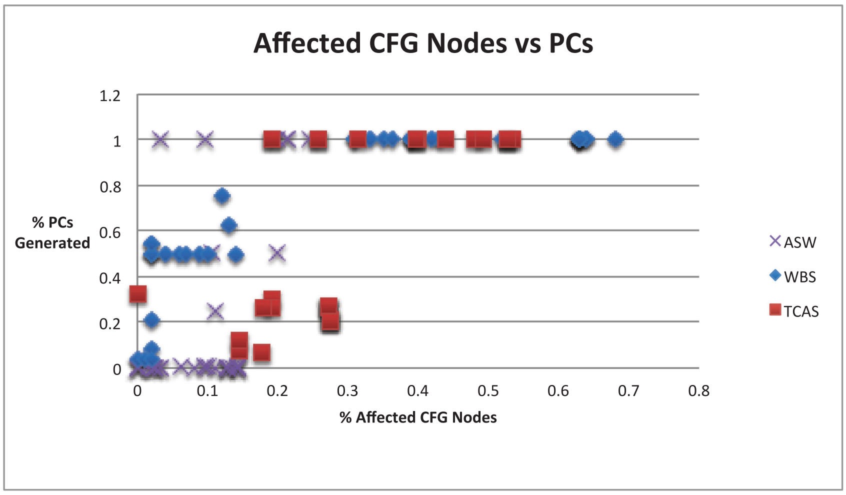


Fig. 9. Correlation between affected CFG nodes and PCs generated.

nested loops, changes are also classified as outermost changes (om) and innermost changes (im).

According to the results in Table VII, there does not appear to be any significant correlation between the change location in the method (top, bottom, or middle) and DiSE results. For example, for versions v1, v2, and v3, where a change is made to a regular statement, DiSE achieved more reductions in the number of states and the number of path conditions when the change was at a bottom location versus a middle location; however, for versions v4, v5, and v6, where a change is made to a conditional statement, it is quite the opposite.

There also does not appear to be any correlation between the location of a change in the program structure (innermost or outermost) and DiSE results. For instance, there is no difference in states and path conditions between DiSE for outermost and innermost changes for versions v10 to v15. We also find that the kinds of changes and DiSE results have no correlation. For example, DiSE on v2 generated many fewer path conditions than for v5, but it generated many more path conditions for v3 than for v6.

Summary. Overall, our study demonstrates that for the examples used, DiSE was often able to guide symbolic execution to explore a smaller number of paths by using the results of the static analysis to direct symbolic execution to explore paths that may be impacted by the change(s) to the code. In the artifacts used, 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.

For TCAS, when only a subset of the path conditions was affected by the changes, DiSE was able to consistently and efficiently 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 small—up to 5%. For ASW and WBS, DiSE always incurred more time than full symbolic execution because of the static analysis overhead.

Although the reduction in time for DiSE compared with full symbolic execution is small and even negative for some changes and artifacts, the cost savings of DiSE, we believe, is realized by the client analysis that uses the DiSE results, for example, regression testing. When DiSE produces fewer path conditions than full symbolic

Table VII. Results of DiSE and Symbolic Execution on Hand-Coded Mutants of Apollo Example

Ver.	Change	CFG Nodes Total	Changed		Affected		Time (ss) DiSE	Full	States DiSE	Full	PC DiSE	Full
v1	reg-t	431	1	(0.2%)	9	(2.1%)	210	219	31	350	1	81
v2	reg-b	431	1	(0.2%)	1	(0.2%)	214	214	41	349	2	81
v3	reg-m	431	7	(1.6%)	10	(2.3%)	264	221	222	350	25	81
v4	nc-om-t	431	1	(0.2%)	3	(0.7%)	210	222	32	350	1	81
v5	nc-om-b	431	1	(0.2%)	61	(14.2%)	327	214	350	350	81	81
v6	nc-om-m	431	1	(0.2%)	3	(0.7%)	210	213	32	350	1	81
v7	nc-im-t	431	1	(0.2%)	2	(0.5%)	220	215	32	350	1	81
v8	nc-im-b	431	1	(0.2%)	1	(0.2%)	326	215	350	350	81	81
v9	nc-im-m	431	1	(0.2%)	2	(0.5%)	234	217	108	350	1	81
v10	nl-om-t	431	1	(0.2%)	9	(2.1%)	210	213	31	349	1	81
v11	nl-om-b	431	1	(0.2%)	7	(1.6%)	324	216	350	350	81	81
v12	nl-om-m	431	1	(0.2%)	7	(1.6%)	214	117	112	314	26	85
v13	nl-im-t	431	1	(0.2%)	7	(1.6%)	210	214	31	349	1	81
v14	nl-im-b	431	1	(0.2%)	7	(1.6%)	324	217	350	350	81	81
v15	nl-im-m	431	1	(0.9%)	7	(1.6%)	214	116	112	314	26	85

Table VIII. Summary of DiSE results using Different Search Strategies

Techniques Pair	Time <	>	=	States <	>	=	PC <	>	=
default vs. greedy	25	15	70	5	9	96	9	5	96
default vs. random	64	21	25	41	24	45	22	35	53
greedy vs. random	62	23	25	41	24	45	22	38	50

execution, the client analysis can be more efficient because the scope of the analysis can focus on the impacted parts of the program, and avoid the parts of the program that are not impacted by the change(s).

In Table VIII, we summarize the results of our exploration strategy evaluation of DiSE. For each pair of exploration strategies, we count the number of versions where each technique takes less (<), more (>), or the same (=) amount of time. We perform a similar comparison with the number of States Explored (*States*) and Path Conditions (PC). For instance, for the technique pair "default vs shortest", there are 25 versions, where it takes less time for DiSE using the default exploration strategy than DiSE using a greedy exploration strategy. However, overall the performance of DiSE does not appear to be particularly sensitive to the exploration strategy used.

In terms of the impact of changes on the effectiveness of DiSE, our intuition is that other factors beyond the number of changes, for example, location and nature of the change, also have a considerable impact on the reductions that can be achieved by DiSE (or any other technique that can characterize the impact 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.

6. DISE AND BEYOND

The goal of DiSE is to enable efficient analysis of program behaviors impacted by changes to the code. DiSE uses a conservative analysis to identify impacted program locations, which are then used to focus symbolic execution on path conditions impacted by program differences. Because the static analysis is conservative, DiSE may, in prici-ple, generate path conditions representing behaviors that are not impacted. However, as experimental results demonstrate, DiSE is generally able to focus symbolic execu-tion on affected program behaviors and enable efficient incremental symbolic execution.

Moreover, the reductions achieved by DiSE benefit the subsequent client analysis, enabling it to also focus on a reduced set of program execution behaviors. The results also show that when DiSE cannot prune the symbolic state space, the overhead is small in comparison with traditional symbolic execution.

6.1. Analysis of Interprocedural Programs

In this work, we present the core DiSE algorithms; however, most realistic programs consist of multiple methods passing data from one method to another following call chains. Information in interprocedural programs flows forward from the "calling" method to the "called" method through the use of arguments. Information flows back to the calling method through values returned by the callee. Hence, when a change is made to a method, the impact of the change can flow to other methods through arguments and return values. To illustrate, consider two methods, A and B.

```
int A (int x) {x = x+1; return B(x);}
int B (int x) {if (x>0) return 1;
               else return 0;}
```

Method `A` contains an assignment to an input variable x and then passes x as an argument to method `B`. Method `B` returns either 0 or 1 based on the value of x. Suppose, a change is made to the assignment statement in method `A`, which is now $x = x - 1$. Then, the impact of this change would flow forward to method `B`. Similarly, when a change is made that affects a variable whose value is returned to the calling method, the impact of the change would flow back to the calling method.

The work in Rungta et al. [2012] extends DiSE to account for the flow of impact between the methods in a program. In the interprocedural version of DiSE (iDiSE), the static impact analysis generates impact sets for each method in P' and estimates the flow of impact between the methods. A call graph is first constructed to capture the possible call sequences between methods in the program. For each method in the call graph, an impact set is generated for the method itself. The potential flow of impact information between methods is stored by annotating the edges in the call graph and generating additional impact sets for the formal parameter of each method.

During directed symbolic execution, paths are pruned based on the reachability of impacted program locations from the current state. Before checking reachability, iDiSE dynamically refines the impact sets for a given method based on which caller invoked the method and the annotation on the corresponding edge in the call graph. This approach efficiently generates precise change impact information between methods.

6.2. Bug Finding Using DiSE

DiSE can be used to analyze programs with assertions when the `assert` statements are desugared into `if` and `throw` statements. In Java programs, this desugaring takes place when `assert` statements in Java source are compiled into Java bytecode. Since DiSE performs symbolic execution on Java bytecode, DiSE does not treat the assertion specially, nor will DiSE 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 are written in a language other than the underlying programming language (e.g., Alloy [Jackson 2006] assertions in Java programs), our technique would work if the assertions are translated to Java, for example, from Alloy. In Yang et al. [2014] we show how DiSE can be used in conjunction with a property differencing technique to optimize common regression scenarios. In that work, we show how the property differencing technique combined with DiSE enable verification of properties that could not be verified in a non-incremental manner.

Table IX. Regression Testing for ASW Example

Ver.	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	1	1	1	1	1	2	1	1	1	2	1	1	1	1	2
Added	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total Tests	1	1	1	1	1	2	1	1	1	2	1	1	1	1	2

6.3. Using DiSE Results to Support Software Evolution Tasks

In other work, we show how the results of DiSE can be used to support various software evolution tasks, for example, regression verification [Backes et al. 2013b], debugging [Rungta et al. 2012]. In this work, we present another 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.

The SPF framework outputs values that can be used as 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. Note that the number of test cases generated by symbolic execution and by DiSE may differ 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, that is, a partial state. As a result, when fields are represented by symbolic values, and constraints on these fields are present on the path condition, multiple path conditions will be generated, however, only a single test case (representing the set of path conditions which differ only in the constraints on the program fields) will be output.

Test cases 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 generated by DiSE are considered tests to be added to augment the test suite.

The results of using DiSE to perform test case selection for `ASW` are shown in Table IX. For each version of `ASW`, Table IX shows the number of changes to the method under test, the number of test cases selected from the original test suite, the number of test cases added by DiSE, and the total number of tests in the new test suite (*selected* + *added*). The combination of selected and added tests will execute all of the branches in the program that are in some way impacted by the changes made to the method under analysis. In the results shown in Table IX, one or two test cases are re-used from the existing test suite, depending on the impact of the change(s). In this example, DiSE does not augment the test suite because the changes do not introduce new behaviors; they only impact existing program behaviors. The re-test all impacted approach using DiSE results can reduce the time spent on regression testing the new version of the program, and reuse of test cases is especially beneficial when existing test cases include oracles.

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 of the branches in the method that are affected by the change. DiSE can be configured to produce test cases that satisfy different types of impacted coverage criteria. The possible configurations of DiSE are described in Rungta et al. [2012]. Furthermore, we also show in another recent work that the results of the DiSE analysis can be used for sound and complete (up to a bound) functional equivalence checking [Backes et al. 2013b].

6.4. C Analysis Framework

DiSE has also been implemented in an another framework, *Proteus*, to analyze C programs [Backes et al. 2013b]. The inputs to Proteus are two related C programs. Proteus first performs a source-level textual diff between the two program versions, then runs the static impact analysis as an LLVM optimization pass and annotates the LLVM bitcode (using LLVM metadata constructs) with the impact information. Incremental symbolic execution on the annotated program is performed using a custom extension to the KLEE symbolic execution engine (KLEE-Inc). Similar reductions have been observed when analyzing evolving C programs [Backes et al. 2013b] as those presented in this work when analyzing evolving Java programs.

6.5. Limitations of DiSE

The DiSE analysis summarizes the impact of program changes for a method under test. Summaries take the form of a set of path conditions which represent the set of impacted program behaviors. The effectiveness of DiSE depends in part on the accuracy of the static impact analysis that computes the impact set (of locations). While the DiSE approach supports using any such analysis, the implementation described in this work uses only a basic analysis that does not compute the flow of impact through global heap locations. The core algorithms discussed in this article are based on an intra-procedural analysis to compute program differences. The work in Rungta et al. [2012] extends DiSE to account for the flow of impact between the methods in a program. We are working on adding support for computing dependencies related to heap structures. Incorporating more sophisticated analyses, such as points-to analyses, can compute more precise impact sets. However, there is potentially a larger computation cost cost as well; investigating this cost/benefit trade-off is a topic for future work.

7. RELATED WORK

Recent years have seen a significant growth in research projects based on symbolic execution, first introduced in the 1970's by Clarke [1976] and King [1976]. These projects have pursued three primary research directions to enhance traditional symbolic execution: (1) to improve its effectiveness [Khurshid et al. 2003; Godefroid et al. 2005; Sen et al. 2005; Cadar and Engler 2005; Deng et al. 2007; Păsăreanu et al. 2011]; (2) to improve its efficiency [Anand et al. 2009; Khurshid and Suen 2005; Godefroid 2007; Inkumsah and Xie 2008; Chang 2010; Santelices and Harrold 2010; Visser et al. 2012]; and (3) to improve its applicability [Person et al. 2008; Khurshid et al. 2005; Csallner et al. 2008; Seo et al. 2006; Geldenhuys et al. 2012; Ma et al. 2011]. The novelty of DiSE is to leverage state-of-the-art symbolic execution techniques and apply a static analysis in synergy to efficiently analyze programs as they undergo changes.

Static analysis has been used effectively to guilde symbolic execution. Chang's dissertation [Chang 2010] 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 [2010] use control and data dependencies to symbolically execute groups of paths, rather than individual paths to enable scalability. In recent work by Qi et al. [2011], program slicing based on program outputs is used with backward symbolic execution to partition paths when they derive the output similarly, generating smaller summaries of program paths. 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.

7.1. Effectiveness of Symbolic Execution

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 [Khurshid et al. 2003; Godefroid et al. 2005; Sen et al. 2005; Cadar and Engler 2005; Deng et al. 2007]. 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 [Godefroid et al. 2005; Sen et al. 2005; Păsăreanu et al. 2011].

7.2. Efficiency of Symbolic Execution

Research to enhance the efficiency of symbolic execution has followed four basic directions: (1) to use abstraction with symbolic execution to reduce the space of exploration [Anand et al. 2009; Khurshid and Suen 2005], (2) to use the underlying constraint solvers more efficiently by performing *compositional* symbolic execution [Godefroid 2007; Bush et al. 2000] and by reusing the constraint solving results [Cadar et al. 2008; Yang et al. 2012; Visser et al. 2012], (3) to enable symbolic execution to find bugs faster through the use of heuristics, such as genetic algorithms [Inkumsah and Xie 2008] that directly control symbolic exploration and focus it on parts that are more likely to contain bugs, and (4) to distribute the problem of symbolic execution into subproblems of lesser complexity and solve them in parallel [Staats and Păsăreanu 2010; Siddiqui and Khurshid 2010, 2012]. DiSE improves the efficiency of symbolic execution by using the results of static analyses to direct symbolic execution to explore program execution paths that may be impacted by changes to the program. Techniques that cache or reuse constraints to speed up performance, for example, Green [Visser et al. 2012], are orthogonal to DiSE and can be used with DiSE to improve the efficiency of the analysis.

7.3. Applicability of Symbolic Execution Results

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 regression analysis [Backes et al. 2013b; Godefroid et al. 2011; Person et al. 2008; Ramos and Engler 2011; Yang et al. 2012], data structure repair [Khurshid et al. 2005], dynamic discovery of invariants [Csallner et al. 2008], program debugging [Ma et al. 2011], and estimation of energy consumption on hardware devices with limited battery capacity [Seo et al. 2006].

Several recent projects use symbolic execution as a basis of test case selection and augmentation [Xu and Rothermel 2009; Taneja et al. 2011; Qi et al. 2010]. 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. Godefroid et al. [2011] 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. DiSE can be configured for several client analysis, among which is regression verification: checking equivalence of related program versions [Backes et al. 2013b]. The reductions computed by DiSE are sound and complete modulo a depth-bound and decidability of SMT-theories.

In previous work, we have developed Memoized Symbolic Execution (Memoise) [Yang et al. 2012]. Memoise enables regression analysis by only re-executing the paths impacted by the program change. While Memoise generates a trie which represents all paths, DiSE only generates affected path conditions. Moreover, while Memoise is dynamic, based on the trie previously collected, DiSE is based on static analysis, using control- and data-flow analyses on the CFG, and thus does not require the results from symbolic execution performed on the previous version.

Differential Symbolic Execution (DSE) [Person et al. 2008] utilizes symbolic execution to characterize the effects of program changes; however, DSE does not use program slicing techniques to improve the efficiency of symbolic execution. It instead relies on abstract summaries of unchanged code blocks to reduce the cost of analysis. During symbolic execution, DSE uses uninterpreted functions to encode the blocks of code that are unchanged between two methods. UC-KLEE [Ramos and Engler 2011], built on top of the KLEE Symbolic Vitual Machine [Cadar et al. 2008] analyzes two arbitrary versions of a C function using symbolic execution, checking that they produce identical outputs when run on the same input values. As an optimization, UC-KLEE is able to skip unchanged instructions. However, unlike DiSE, UC-KLEE neither leverages nor produces impacted behavior information.

DiSE takes inspiration from regression model checking (RMC) [Yang et al. 2009], which uses the differences between two program versions to drive the pruning of the state space when model checking the new version of a program. RMC computes reachable program coverage elements, for example, 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. The dangerous elements information is then combined with the reachable elements information to prune safe substate spaces during a pruning mode run of RMC on the modified version of the program. A key difference between DiSE and RMC is that DiSE does not require the availability of the internal states of the previous analysis to analyze the current program version.

8. CONCLUSIONS

In this article, we describe 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 four Java applications. The results of our case-study demonstrate that DiSE can efficiently generate 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.

The evaluation presented in this article demonstrates the potential of a technique such as DiSE to efficiently generate impacted program behaviors. In this article, we present how the output of DiSE can be used for regression testing, specifically test case selection and addition. In other recent work, we show how we can use the generated path conditions for improving regression verification, improving delta debugging, obtaining better coverage of impacted constructs, and visualizing the output for presentation to end-users [Backes et al. 2013b; Rungta et al. 2012; Mercer et al. 2012].

APPENDIX A. COMPLETE RESULTS

Table X. Results of DiSE using Different Search Strategies on ASW Mutants

Ver.	Strategy	Time (ss)	States	PC	Prunings	Infeasible	Total Terms	Avg. Terms	Min DP	Max DP	Total DP	Avg. DP
v1	default	4	13	1	0	12	7	7.0	12	12	12	12.0
v1	greedy	4	13	1	0	12	7	7.0	12	12	12	12.0
v1	random	4	6.4	0.1	0	5.4	0.7	0.7	1.2	1.2	1.2	1.2
v2	default	4	91	8	14	48	56	7.0	12	12	96	12.0
v2	greedy	4	91	8	14	48	56	7.0	12	12	96	12
v2	random	4	91	4.2	17.8	48	29.4	7.0	12	12	50.4	12.0
v3	default	4	18	2	0	16	14	7.0	12	12	24	12.0
v3	greedy	4	18	2	0	16	14	7.0	12	12	24	12.0
v3	random	4	27.4	2.2	2.8	21.6	15.4	6.3	10.8	10.8	26.4	10.8
v4	default	4	20	2	1	16	14	7.0	12	12	24	12.0
v4	greedy	4	20	2	1	16	14	7.0	12	12	24	12.0
v4	random	4.1	37.1	2.4	4.4	27.8	16.8	6.3	10.8	10.8	28.8	10.8
v5	default	4	20	2	1	16	14	7.0	12	12	24	12.0
v5	greedy	4	20	2	1	16	14	7.0	12	12	24	12.0
v5	random	4	33.8	2.7	4.3	25	18.9	7.0	12	12	32.4	12.0
v6	default	5	2299	576	590	0	4032	7.0	12	12	6912	12.0
v6	greedy	7	2299	576	590	0	4032	7.0	12	12	6912	12.0
v6	random	5.4	2299	576	590	0	4032	7.0	12	12	6912	12.0
v7	default	4	24	3	0	21	21	7.0	12	12	36	12.0
v7	greedy	4	24	3	0	21	21	7.0	12	12	36	12.0
v7	random	4	25.4	1.9	2.1	20.6	13.3	5.6	9.6	9.6	22.8	9.6
v8	default	4	30	4	0	26	28	7.0	12	12	48	12.0
v8	greedy	4	30	4	0	26	28	7.0	12	12	48	12.0
v8	random	4	36.6	2	4.7	28	14	6.3	10.8	10.8	24	10.8
v9	default	5	13	1	0	12	7	7.0	12	12	12	12.0
v9	greedy	5	13	1	0	12	7	7.0	12	12	12	12.0
v9	random	5.1	14.3	0.2	0.4	12	1.4	1.4	2.4	2.4	2.4	2.4
v10	default	7	1726	289	304	574	2023	7.0	12	12	3468	12.0
v10	greedy	5	1726	289	304	574	2023	7.0	12	12	3468	12.0
v10	random	5.2	1724.4	292.2	299.2	575	2045.4	7.0	12	12	3506.4	12.0
v11	default	4	13	1	0	12	7	7.0	12	12	12	12.0
v11	greedy	4	13	1	0	12	7	7.0	12	12	12	12.0
v11	random	4.2	6.2	0.2	0	5.2	1.4	1.4	2.4	2.4	2.4	2.4
v12	default	5	13	1	0	12	7	7.0	12	12	12	12.0
v12	greedy	5	13	1	0	12	7	7.0	12	12	12	12.0
v12	random	5.2	14.4	0.5	0.7	12	3.5	3.5	6	6	6	6.0
v13	default	4	13	1	0	12	7	7.0	12	12	12	12.0
v13	greedy	4	13	1	0	12	7	7.0	12	12	12	12.0
v13	random	4.1	3.4	0	0	2.4	0	0.0	0	0	0	0.0
v14	default	4	26	3	1	21	21	7.0	12	12	36	12.0
v14	greedy	4	26	3	1	21	21	7.0	12	12	36	12.0
v14	random	4	34.7	2.8	3.5	25.8	19.6	6.3	10.8	10.8	33.6	10.8
v15	default	5	2299	576	590	0	4032	7.0	12	12	6912	12.0
v15	greedy	5	2299	576	590	0	4032	7.0	12	12	6912	12.0
v15	random	5.3	2299	576	590	0	4032	7.0	12	12	6912	12.0

(Continued)

Table X. Continued

Ver.	Strategy	Time (ss)	States	PC	Prunings	Infeasible	Total Terms	Avg. Terms	Min DP	Max DP	Total DP	Avg. DP
v16	default	5	1726	289	304	574	2023	7.0	12	12	3468	12.0
v16	greedy	5	1726	289	304	574	2023	7.0	12	12	3468	12.0
v16	random	5.1	1724.2	286.6	304.6	575.2	2006.2	7.0	12	12	3439.2	12.0
v17	default	5	1726	289	304	574	2023	7.0	12	12	3468	12.0
v17	greedy	5	1726	289	304	574	2023	7.0	12	12	3468	12.0
v17	random	0	0	0	0	0	0	0.0	0	0	0	0.0
v18	default	4	43	8	14	0	24	3.0	7	7	56	7.0
v18	greedy	4	43	8	14	0	24	3.0	7	7	56	7.0
v18	random	4.2	43	8	14	0	24	3.0	7	7	56	7.0
v19	default	6	18	2	0	16	14	7.0	12	12	24	12.0
v19	greedy	4	18	2	0	16	14	7.0	12	12	24	12.0
v19	random	4.3	37.7	2.9	4.6	28.6	20.3	7.0	12	12	34.8	12.0
v20	default	4	18	2	0	16	14	7.0	12	12	24	12.0
v20	greedy	4	18	2	0	16	14	7.0	12	12	24	12.0
v20	random	4	32.8	2.4	3	25.8	16.8	6.3	10.8	10.8	28.8	10.8
v21	default	4	18	2	0	16	14	7.0	12	12	24	12.0
v21	greedy	4	18	2	0	16	14	7.0	12	12	24	12.0
v21	random	4.1	32	2.9	3.5	24.6	20.3	7.0	12	12	34.8	12.0
v22	default	4	18	2	0	16	14	7.0	12	12	24	12.0
v22	greedy	4	18	2	0	16	14	7.0	12	12	24	12.0
v22	random	4.2	31.6	2.4	3.4	24.8	16.8	6.3	10.8	10.8	28.8	10.8
v23	default	5	2299	576	590	0	4032	7.0	12	12	6912	12.0
v23	greedy	5	2299	576	590	0	4032	7.0	12	12	6912	12.0
v23	random	5.3	2299	576	590	0	4032	7.0	12	12	6912	12.0
v24	default	4	18	2	0	16	14	7.0	12	12	24	12.0
v24	greedy	4	18	2	0	16	14	7.0	12	12	24	12.0
v24	random	4.3	29	2	3.7	21.8	14	6.3	10.8	10.8	24	10.8
v25	default	4	20	2	1	16	14	7.0	12	12	24	12.0
v25	greedy	4	20	2	1	16	14	7.0	12	12	24	12.0
v25	random	4	36.9	3	3.9	28.6	21	7.0	12	12	36	12.0
v26	default	4	13	1	0	12	7	7.0	12	12	12	12.0
v26	greedy	4	13	1	0	12	7	7.0	12	12	12	12.0
v26	random	4	4.6	0	0	3.6	0	0.0	0	0	0	0.0
v27	default	4	13	1	0	12	7	7.0	12	12	12	12.0
v27	greedy	4	13	1	0	12	7	7.0	12	12	12	12.0
v27	random	4	14.3	0.6	0.6	12	4.2	4.2	7.2	7.2	7.2	7.2
v28	default	4	18	2	0	16	14	7.0	12	12	24	12.0
v28	greedy	4	18	2	0	16	14	7.0	12	12	24	12.0
v28	random	4	21.3	1.3	2.2	16.8	9.1	5.6	9.6	9.6	15.6	9.6
v29	default	4	13	1	0	12	7	7.0	12	12	12	12.0
v29	greedy	4	13	1	0	12	7	7.0	12	12	12	12.0
v29	random	4.2	15.4	1	2.4	12	7.0	7.0	12	12	12	12.0
v30	default	5	2299	576	590	0	4032	7.0	12	12	6912	12.0
v30	greedy	5	2299	576	590	0	4032	7.0	12	12	6912	12.0
v30	random	5.7	2299	576	590	0	4032	7.0	12	12	6912	12.0
v31	default	4	13	1	0	12	7	7.0	12	12	12	12.0
v31	greedy	4	13	1	0	12	7	7.0	12	12	12	12.0
v31	random	4	27.2	1.7	3.1	20.4	11.9	4.9	8.4	8.4	20.4	8.4

(Continued)

Table X. Continued

Ver.	Strategy	Time (ss)	States	PC	Prunings	Infeasible	Total Terms	Avg. Terms	Min DP	Max DP	Total DP	Avg. DP
v32	default	4	16	1	1	13	7	7.0	12	12	12	12.0
v32	greedy	4	16	1	1	13	7	7.0	12	12	12	12.0
v32	random	4.3	17.1	0.7	1.2	13	4.9	4.2	7.2	7.2	8.4	7.2
v33	default	4	15	2	0	12	14	7.0	12	12	24	12.0
v33	greedy	4	15	2	0	12	14	7.0	12	12	24	12.0
v33	random	4.2	17.1	1.4	2.7	12	9.8	7.0	12	12	16.8	12.0
v34	default	5	715	144	14	432	1008	7.0	12	12	1728	12.0
v34	greedy	5	715	144	14	432	1008	7.0	12	12	1728	12.0
v34	random	5	716.8	72.2	86.4	433.2	505.4	7.0	12	12	866.4	12.0
v35	default	5	22	3	0	18	21	7.0	12	12	36	12.0
v35	greedy	5	22	3	0	18	21	7.0	12	12	36	12.0
v35	random	5.1	18.9	0.9	1.5	14.8	6.3	4.2	7.2	7.2	10.8	7.2
v36	default	6	21	3	3	12	21	7.0	12	12	36	12.0
v36	greedy	5	21	3	3	12	21	7.0	12	12	36	12.0
v36	random	5.1	23.2	3.2	4.4	12	22.4	7.0	12	12	38.4	12.0
v37	default	4	20	2	1	16	14	7.0	12	12	24	12.0
v37	greedy	4	20	2	1	16	14	7.0	12	12	24	12.0
v37	random	4.2	39.7	3	4	29.6	21	5.6	9.6	9.6	36	9.6
v38	default	5	2299	576	590	0	4032	7.0	12	12	6912	12.0
v38	greedy	5	2299	576	590	0	4032	7.0	12	12	6912	12.0
v38	random	5.2	2299	576	590	0	4032	7.0	12	12	6912	12.0
v39	default	4	13	1	0	12	7	7.0	12	12	12	12.0
v39	greedy	4	13	1	0	12	7	7.0	12	12	12	12.0
v39	random	4	14.5	0.6	0.8	12	4.2	4.2	7.2	7.2	7.2	7.2

Table XI. Results of DiSE using Different Search Strategies on WBS Mutants

Ver.	Strategy	Time (ss)	States	PC	Prunings	Infeasible	Total Terms	Avg. Terms	Min DP	Max DP	Total DP	Avg. DP
v1	default	3	47	24	0	0	72	3.0	3	7	128	5.3
v1	greedy	3	47	24	0	0	72	3.0	3	7	128	5.3
v1	random	2.2	47	24	0	0	72	3.0	3	7	128	5.3
v2	default	4	39	18	0	4	54	3.0	3	7	86	4.8
v2	greedy	3	39	18	0	4	54	3.0	3	7	86	4.8
v2	random	2.2	33	13.5	0	7	40.5	3.0	3	7	69.8	5.2
v3	default	2	35	12	0	12	36	3.0	3	7	64	5.3
v3	greedy	2	35	12	0	12	36	3.0	3	7	64	5.3
v3	random	2.2	35	12	0	12	36	3.0	3	7	64	5.3
v4	default	2	4	1	0	3	3	3.0	3	3	3	3.0
v4	greedy	2	4	1	0	3	3	3.0	3	3	3	3.0
v4	random	2	4.5	1	0	3.5	3	3.0	3.5	3.5	3.5	3.5
v5	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v5	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v5	random	2.2	47	24	0	0	72	3.0	3	7	128	5.3
v6	default	2	8	2	0	5	6	3.0	3	4	7	3.5
v6	greedy	4	8	2	0	5	6	3.0	3	4	7	3.5
v6	random	2.1	9.1	2	0	6.1	6	3.0	3	5.1	8.1	4.1

(Continued)

Table XI. Continued

Ver.	Strategy	Time (ss)	States	PC	Prunings	Infeasible	Total Terms	Avg. Terms	Min DP	Max DP	Total DP	Avg. DP
v7	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v7	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v7	random	2.4	47	24	0	0	72	3.0	3	7	128	5.3
v8	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v8	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v8	random	2	47	24	0	0	72	3.0	3	7	128	5.3
v9	default	2	35	12	0	12	36	3.0	3	7	64	5.3
v9	greedy	2	35	12	0	12	36	3.0	3	7	64	5.3
v9	random	2.1	35	12	0	12	36	3.0	3	7	64	5.3
v10	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v10	greedy	4	47	24	0	0	72	3.0	3	7	128	5.3
v10	random	2.2	47	24	0	0	72	3.0	3	7	128	5.3
v11	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v11	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v11	random	2.4	47	24	0	0	72	3.0	3	7	128	5.3
v12	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v12	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v12	random	2.3	47	24	0	0	72	3.0	3	7	128	5.3
v13	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v13	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v13	random	2.1	47	24	0	0	72	3.0	3	7	128	5.3
v14	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v14	greedy	3	47	24	0	0	72	3.0	3	7	128	5.3
v14	random	2	47	24	0	0	72	3.0	3	7	128	5.3
v15	default	3	47	24	0	0	72	3.0	3	7	128	5.3
v15	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v15	random	2.4	47	24	0	0	72	3.0	3	7	128	5.3
v16	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v16	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v16	random	2.2	47	24	0	0	72	3.0	3	7	128	5.3
v17	default	2	35	12	0	12	36	3.0	3	7	64	5.3
v17	greedy	2	35	12	0	12	36	3.0	3	7	64	5.3
v17	random	2.1	35	12	0	12	36	3.0	3	7	64	5.3
v18	default	2	35	12	0	12	36	3.0	3	7	64	5.3
v18	greedy	2	35	12	0	12	36	3.0	3	7	64	5.3
v18	random	2.1	35	12	0	12	36	3.0	3	7	64	5.3
v19	default	2	12	5	0	4	15	3.0	3	4	16	3.2
v19	greedy	2	12	5	0	4	15	3.0	3	4	16	3.2
v19	random	2.5	27.8	13	0	4	39	3.0	3.4	5.8	71.2	4.9
v20	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v20	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v20	random	2.2	47	24	0	0	72	3.0	3	7	128	5.3
v21	default	2	28	13	0	6	39	3.0	3	6	54	4.2
v21	greedy	2	28	13	0	6	39	3.0	3	6	54	4.2
v21	random	2.2	24.4	10.6	0	6	31.8	3.0	4.1	6.6	59.2	5.4
v22	default	2	4	1	0	3	3	3.0	3	3	3	3.0
v22	greedy	2	4	1	0	3	3	3.0	3	3	3	3.0
v22	random	2.2	5.5	1	0	4.5	3	3.0	4.5	4.5	4.5	4.5

(Continued)

Table XI. Continued

Ver.	Strategy	Time (ss)	States	PC	Prunings	Infeasible	Total Terms	Avg. Terms	Min DP	Max DP	Total DP	Avg. DP
v23	default	2	35	12	0	12	36	3.0	3	7	64	5.3
v23	greedy	4	35	12	0	12	36	3.0	3	7	64	5.3
v23	random	2.6	35	12	0	12	36	3.0	3	7	64	5.3
v24	default	3	47	24	0	0	72	3.0	3	7	128	5.3
v24	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v24	random	2	47	24	0	0	72	3.0	3	7	128	5.3
v25	default	2	28	13	0	6	39	3.0	3	6	54	4.2
v25	greedy	2	28	13	0	6	39	3.0	3	6	54	4.2
v25	random	2.1	25.9	11.4	0	6	34.2	3.0	3.8	6.7	63.6	5.4
v26	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v26	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v26	random	2	47	24	0	0	72	3.0	3	7	128	5.3
v27	default	2	35	12	0	12	36	3.0	3	7	64	5.3
v27	greedy	2	35	12	0	12	36	3.0	3	7	64	5.3
v27	random	2.7	35	12	0	12	36	3.0	3	7	64	5.3
v28	default	2	31	12	0	8	36	3.0	3	7	53	4.4
v28	greedy	2	31	12	0	8	36	3.0	3	7	53	4.4
v28	random	2	29	10.5	0	9	31.5	3.0	3	7	49.7	4.8
v29	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v29	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v29	random	2.1	47	24	0	0	72	3.0	3	7	128	5.3
v30	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v30	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v30	random	2	47	24	0	0	72	3.0	3	7	128	5.3
v31	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v31	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v31	random	2.3	47	24	0	0	72	3.0	3	7	128	5.3
v32	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v32	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v32	random	2.2	47	24	0	0	72	3.0	3	7	128	5.3
v33	default	2	95	24	24	0	72	3.0	4	8	152	6.3
v33	greedy	2	95	24	24	0	72	3.0	4	8	152	6.3
v33	random	2.1	95	24	24	0	72	3.0	4	8	152	6.3
v34	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v34	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v34	random	2	47	24	0	0	72	3.0	3	7	128	5.3
v35	default	2	35	12	0	12	36	3.0	3	7	64	5.3
v35	greedy	4	35	12	0	12	36	3.0	3	7	64	5.3
v35	random	2	35	12	0	12	36	3.0	3	7	64	5.3
v36	default	2	35	12	0	12	36	3.0	3	7	64	5.3
v36	greedy	2	35	12	0	12	36	3.0	3	7	64	5.3
v36	random	2.4	35	12	0	12	36	3.0	3	7	64	5.3
v37	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v37	greedy	2	47	24	0	0	72	3.0	3	7	128	5.3
v37	random	2	47	24	0	0	72	3.0	3	7	128	5.3
v38	default	2	4	1	0	3	3	3.0	3	3	3	3.0
v38	greedy	2	4	1	0	3	3	3.0	3	3	3	3.0
v38	random	2	4.2	1	0	3.2	3	3.0	3.2	3.2	3.2	3.2

(Continued)

Table XI. Continued

Ver.	Strategy	Time (ss)	States	PC	Prunings	Infeasible	Total Terms	Avg. Terms	Min DP	Max DP	Total DP	Avg. DP
v39	default	2	47	24	0	0	72	3.0	3	7	128	5.3
v39	greedy	4	47	24	0	0	72	3.0	3	7	128	5.3
v39	random	2.4	47	24	0	0	72	3.0	3	7	128	5.3
v40	default	2	35	15	0	6	45	3.0	3	7	68	4.5
v40	greedy	2	35	15	0	6	45	3.0	3	7	68	4.5
v40	random	2.2	28.6	10.2	0	9.2	30.6	3.0	3	7	49.7	4.9

Table XII. Results of DiSE using Different Search Strategies on TCAS Mutants

Ver.	Strategy	Time (ss)	States	PC	Prunings	Infeasible	Total Terms	Avg. Terms	Min DP	Max DP	Total DP	Avg. DP
v1	default	38	330	23	92	102	128	5.6	3	18	147	6.4
v1	greedy	37	330	24	93	100	139	5.8	3	18	160	6.7
v1	random	41.3	332.5	25.6	96.8	91.4	155.3	6.0	3	17.6	181.3	7.0
v2	default	120	679	68	272	0	611	9.0	3	18	835	12.3
v2	greedy	127	679	68	272	0	611	9.0	3	18	835	12.3
v2	random	122.6	679	68	272	0	611	9.0	3	18	835	12.3
v3	default	154	837	79	340	0	726	9.2	3	18	1006	12.7
v3	greedy	151	837	79	340	0	726	9.2	3	18	1006	12.7
v3	random	151.8	837	79	340	0	726	9.2	3	18	1006	12.7
v4	default	37	353	22	100	111	117	5.3	3	17	129	5.9
v4	greedy	37	349	22	100	109	117	5.3	3	17	125	5.7
v4	random	40.6	343.4	25.7	96	102.9	155.3	6.0	3	16	178.3	6.9
v5	default	148	763	94	288	0	903	9.6	3	17	1207	12.8
v5	greedy	153	763	94	288	0	903	9.6	3	17	1207	12.8
v5	random	146.7	763	94	288	0	903	9.6	3	17	1207	12.8
v6	default	124	679	68	272	0	611	9.0	3	18	835	12.3
v6	greedy	126	679	68	272	0	611	9.0	3	18	835	12.3
v6	random	128.1	679	68	272	0	611	9.0	3	18	835	12.3
v9	default	33	329	23	93	99	128	5.6	3	17	145	6.3
v9	greedy	33	330	24	94	98	139	5.8	3	17	158	6.6
v9	random	39.5	347.1	30.2	95.8	98.8	205.1	6.7	3	17.4	247.6	8.1
v10	default	135	743	84	288	0	783	9.3	3	18	1079	12.9
v10	greedy	128	743	84	288	0	783	9.3	3	18	1079	12.9
v10	random	128.4	743	84	288	0	783	9.3	3	18	1079	12.9
v11	default	129	743	84	288	0	783	9.3	3	18	1079	12.9
v11	greedy	128	743	84	288	0	783	9.3	3	18	1079	12.9
v11	random	127.7	743	84	288	0	783	9.3	3	18	1079	12.9
v12	default	265	1407	144	560	0	1360	9.4	3	18	1888	13.1
v12	greedy	270	1407	144	560	0	1360	9.4	3	18	1888	13.1
v12	random	270	1407	144	560	0	1360	9.4	3	18	1888	13.1
v20	default	34	329	23	93	99	128	5.6	3	17	145	6.3
v20	greedy	34	330	24	94	98	139	5.8	3	17	158	6.6
v20	random	40.6	348.5	28.6	98.9	96.5	187.5	6.5	3	17.6	224.2	7.8
v21	default	26	225	24	52	75	139	5.8	3	16	158	6.6
v21	greedy	26	226	25	53	74	150	6.0	3	16	170	6.8
v21	random	35.4	248.8	31.3	61.9	65.1	216.3	6.9	3	16.5	252.7	8.0

(Continued)

Table XII. Continued

Ver.	Strategy	Time (ss)	States	PC	Prunings	Infeasible	Total Terms	Avg. Terms	Min DP	Max DP	Total DP	Avg. DP
v22	default	24	245	23	51	99	128	5.6	3	16	143	6.2
v22	greedy	24	245	24	52	97	139	5.8	3	16	155	6.5
v22	random	29.7	233.9	29.1	56.1	66.3	193	6.5	3	16.1	220.4	7.4
v23	default	119	631	88	228	0	826	9.4	3	17	1042	11.8
v23	greedy	113	631	88	228	0	826	9.4	3	17	1042	11.8
v23	random	116.3	631	88	228	0	826	9.4	3	17	1042	11.8
v24	default	131	655	88	240	0	826	9.4	3	17	1066	12.1
v24	greedy	126	655	88	240	0	826	9.4	3	17	1066	12.1
v24	random	129.6	655	88	240	0	826	9.4	3	17	1066	12.1
v25	default	127	679	76	264	0	697	9.2	3	18	969	12.8
v25	greedy	123	679	76	264	0	697	9.2	3	18	969	12.8
v25	random	110.2	611.1	68.4	237.6	0	627.3	8.3	2.7	16.2	872.1	11.5
v26	default	119	667	62	272	0	527	8.5	3	17	751	12.1
v26	greedy	120	667	62	272	0	527	8.5	3	17	751	12.1
v26	random	121.5	667	62	272	0	527	8.5	3	17	751	12.1
v27	default	148	763	94	288	0	903	9.6	3	17	1207	12.8
v27	greedy	158	763	94	288	0	903	9.6	3	17	1207	12.8
v27	random	147	763	94	288	0	903	9.6	3	17	1207	12.8
v28	default	36	363	23	94	132	128	5.6	3	17	145	6.3
v28	greedy	36	361	23	96	128	128	5.6	3	17	145	6.3
v28	random	39.2	358.1	32.5	98.2	99.6	228.7	7.0	3	17.5	278.3	8.5
v29	default	25	245	23	51	99	128	5.6	3	16	143	6.2
v29	greedy	25	245	24	52	97	139	5.8	3	16	155	6.5
v29	random	29.1	235	29.9	56	65.9	200.9	6.6	3	16.3	230.2	7.6
v30	default	26	225	24	52	75	139	5.8	3	16	158	6.6
v30	greedy	27	226	25	53	74	150	6.0	3	16	170	6.8
v30	random	29.9	239.2	29.3	56.3	70.8	194.2	6.6	3	16.1	225.3	7.7
v31	default	6	55	5	14	25	50	10.0	6	17	72	14.4
v31	greedy	11	86	18	20	26	110	6.1	3	17	130	7.2
v31	random	7.6	83.9	17.7	17.6	27.6	111	6.4	3.3	17	131.4	7.6
v32	default	16	91	8	30	20	83	10.4	6	17	114	14.3
v32	greedy	20	128	21	40	20	143	6.8	3	17	176	8.4
v32	random	13.9	116	17.3	30.9	29.1	119.9	7.1	3.6	16.6	147.4	8.9
v34	default	458	2317	239	920	0	2405	10.1	3	18	3361	14.1
v34	greedy	449	2317	239	920	0	2405	10.1	3	18	3361	14.1
v34	random	454.9	2317	239	920	0	2405	10.1	3	18	3361	14.1
v35	default	34	349	22	92	123	117	5.3	3	17	129	5.9
v35	greedy	34	349	23	93	121	128	5.6	3	17	142	6.2
v35	random	36.1	338.4	30.9	90.9	97.6	211.8	6.8	3	17.2	253.3	8.2
v36	default	1	18	22	92	17	117	5.3	3	17	129	5.9
v36	greedy	1	18	23	93	17	128	5.6	3	17	142	6.2
v36	random	1.1	4.9	1	0	3.9	3.9	3.9	3.9	3.9	3.9	3.9
v37	default	121	679	68	272	0	611	9.0	3	18	835	12.3
v37	greedy	117	679	68	272	0	611	9.0	3	18	835	12.3
v37	random	123.1	679	68	272	0	611	9.0	3	18	835	12.3
v39	default	124	679	76	264	0	697	9.2	3	18	969	12.8
v39	greedy	121	679	76	264	0	697	9.2	3	18	969	12.8
v39	random	122.6	679	76	264	0	697	9.2	3	18	969	12.8

(Continued)

Table XII. Continued

Ver.	Strategy	Time (ss)	States	PC	Prunings	Infeasible	Total Terms	Avg. Terms	Min DP	Max DP	Total DP	Avg. DP
v40	default	6	54	5	13	25	50	10.0	6	17	65	13.0
v40	greedy	9	71	17	12	23	99	5.8	3	17	111	6.5
v40	random	7.8	85.9	13.4	17.9	32.8	84.6	6.5	3.8	16.6	98.3	7.5
v41	default	33	309	22	84	99	117	5.3	3	17	129	5.9
v41	greedy	31	305	22	84	97	117	5.3	3	17	125	5.7
v41	random	31	305	22	84	97	117	5.3	3	17	125	5.7

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.

REFERENCES

Saswat Anand, Corina S. Păsăreanu, and Willem Visser. 2009. Symbolic execution with abstraction. *Inter. J. Softw. Tools Technol. Transfer* 11, 1 (2009), 53–67.

John Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. 2013a. Proteus: A change impact analysis framework. Tech. Rep.

John Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. 2013b. Regression verification using impact summaries. In *Model Checking Software*, Springer, 99–116.

William R. Bush, Jonathan D. Pincus, and David J. Sielaff. 2000. A static analyzer for finding dynamic programming errors. *Software: Prac. Exper.* 30, 7 (2000), 775–802.

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of OSDI*. 209–224.

Cristian Cadar and Dawson R. Engler. 2005. Execution generated test cases: How to make systems code crash itself. In *Proceedings of SPIN*. 2–23.

Walter Chochen Chang. 2010. Improving dynamic analysis with data flow analysis. Ph.D. Dissertation, University of Texas at Austin.

Lori A. Clarke. 1976. A program testing system. In *Proceedings of the 1976 Annual Conference (ACM'76)*. 488–491.

Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic symbolic execution for invariant inference. In *Proceedings of ICSE*. 281–290.

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of TACAS*. 337–340.

Xianghua Deng, Robby, and John Hatcliff. 2007. Kiasan/KUnit: Automatic test case generation and analysis feedback for open object-oriented systems. In *Proceedings of TAICPART-MUTATION*. 3–12.

Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. In *Proceedings of ISSTA*. 166–176.

Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In *Proceedings of POPL*. 47–54.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of PLDI*. 213–223.

Patrice Godefroid, Shuvendu K. Lahiri, and Cindy Rubio-González. 2011. Statically validating must summaries for incremental compositional dynamic test generation. In *Proceedings of SAS*. 112–128.

Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. 2001. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Meth.* 10, 2, 184–208.

Kobi Inkumsah and Tao Xie. 2008. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of ASE*. 297–306.

Daniel Jackson. 2006. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, MA.

Anjali Joshi and Mats Per Erik Heimdahl. 2005. Model-based safety analysis of simulink models using SCADE design verifier. In *Proceedings of SAFECOMP*. Lecture Notes in Computer Science, vol. 3688, 122–135.

Sarfraz Khurshid, Iván García, and Yuk Lai Suen. 2005. Repairing structurally complex data. In *Proceedings of SPIN*. 123–138.

Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized symbolic execution for model checking and testing. In *Proceedings of TACAS*. 553–568.

Sarfraz Khurshid and Yuk Lai Suen. 2005. Generalizing symbolic execution to library classes. In *Proceedings of PASTE*. 103–110.

James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7, 385–394.

Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed symbolic execution. In *Proceedings of SAS*. 95–111.

Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: An automated class mutation system. *Softw. Test. Verif. Reliab.* 15, 2 (2005), 97–133.

Eric Mercer, Suzette Person, and Neha Rungta. 2012. Computing and visualizing the impact of change with Java PathFinder extensions. *SIGSOFT Softw. Eng. Notes* 37, 6 (2012), 1–5.

Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proceedings of ISSTA*. 15–25.

Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic execution of Java bytecode. In *Proceedings of ASE*. 179–180.

Corina S. Păsăreanu, Neha Rungta, and Willem Visser. 2011. Symbolic execution with mixed concrete-symbolic solving. In *Proceedings of ISSTA*. 34–44.

Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. 2008. Differential symbolic execution. In *Proceedings of FSE*. 226–237.

Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. In *Proceedings of PLDI*. 504–515.

Dawei Qi, Hoang D. T. Nguyen, and Abhik Roychoudhury. 2011. Path exploration based on symbolic output. In *Proceedings of ESEC/FSE*. 278–288.

Dawei Qi, Abhik Roychoudhury, and Zhenkai Liang. 2010. Test generation to expose changes in evolving programs. In *Proceedings of ASE*. 397–406.

David A. Ramos and Dawson R. Engler. 2011. Practical, low-effort equivalence verification of real code. In *Proceedings of CAV*. 669–685.

Neha Rungta, Suzette Person, and Joshua Branchaud. 2012. A change impact analysis to characterize evolving program behaviors. In *Proceedings of ICSM*. 109–118.

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

Raul Santelices and Mary Jean Harrold. 2010. Exploiting program dependencies for scalable multiple-path symbolic execution. In *Proceedings of ISSTA*. 195–206.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. In *Proceedings of ESEC/FSE*. 263–272.

Chiyoung Seo, Sam Malek, and Nenad Medvidovic. 2006. An energy consumption framework for distributed Java-based software systems. Tech. Rep. USC-CSE-2006-604. University of Southern California.

Junaid Haroon Siddiqui and Sarfraz Khurshid. 2010. ParSym: Parallel symbolic execution. In *Proceedings of ICSTE*. V1–405–V1–409.

Junaid Haroon Siddiqui and Sarfraz Khurshid. 2012. Scaling symbolic execution using ranged analysis. In *Proceedings of OOPSLA*. 523–536.

Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Păsăreanu. 2011. CORAL: Solving complex constraints for symbolic PathFinder. In *NASA Formal Methods*. Lecture Notes in Computer Science, vol. 6617, Springer, 359–374.

Matt Staats and Corina S. Păsăreanu. 2010. Parallel symbolic execution for structural test generation. In *Proceedings of ISSTA*. 183–194.

Janos Sztipanovits and Gabor Karsai. 2002. Generative programming for embedded systems. In *Proceedings of GPCE*. 32–49.

Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2011. eXpress: Guided path exploration for efficient regression test generation. In *Proceedings of ISSTA*. 1–11.

Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, reusing and recycling constraints in program analysis. In *Proceedings of FSE*. 58:1–58:11.

Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. 2003. Model checking programs. *Automat. Softw. Eng.* 10, 2 (2003), 203–232.

Zhihong Xu and Gregg Rothermel. 2009. Directed test suite augmentation. In *Proceedings of APSEC*. 406–413.

Guowei Yang, Matthew B. Dwyer, and Gregg Rothermel. 2009. Regression model checking. In *Proceedings of ICSM*. 115–124.

Guowei Yang, Sarfraz Khurshid, Suzette Person, and Neha Rungta. 2014. Property differencing for incremental checking. In *Proceedings of ICSE*. to appear.

Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In *Proceedings of ISSTA*. 144–154.

Received September 2013; revised March 2014; accepted April 2014