THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Computing 1, 13 Computing Drive, Singapore 117417

TR11/11

Comprehensive Test Suite Augmentation

Marcel Böhme and Abhik Roychoudhury

November 2011

T e c h n i c a l R e p o r t

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

OOI Beng Chin
Dean of School

Comprehensive Test Suite Augmentation

Marcel Böhme
School of Computing
National University of Singapore
marcel.boehme@nus.edu.sg

Abhik Roychoudhury
School of Computing
National University of Singapore
abhik@comp.nus.edu.sg

***Abstract*—Test Suite Augmentation (TSA) is based on the assumption that an existing test suite T already determines whether a program P works properly. When P evolves to P', a comprehensive set of test cases that witness the changed behavior shall be generated and augmented to T. We observe that a syntactic change may be propagated in different ways, thereby manifesting itself as different semantic changes (in terms of observable program behavior). We present a path exploration technique that seeks to generate one change revealing test case for every semantic change, rather than generate multiple test cases which execute the same change and propagate it in the same fashion to the program output.**

Our change-based path exploration technique 1) finds all paths that execute at least one changed statement, 2) finds all paths along which the impact of a syntactic change is propagated to at least one output statement and 3) generates for both programs P and P' a comprehensive number of test cases, each of which exposes a different semantic change. If time is bounded during test generation, the semantic difference is computed over those change partitions that have been explored. Our method is not restricted by the number of changes across the two program versions being analyzed - there can be multiple changes across the program versions.

The results from our experiments suggest that by covering the syntactic changes, or even by considering possible ways of propagating the effects of these syntactic changes - we cannot hope to uncover all possible differences in behavior across two program versions. In other words, analysis of both program versions is needed for generating a comprehensive new set of tests which can augment the test suite of the old program version. For our subject programs, our method of change execution, change propagation and semantic differencing, yields 6 times as many change-revealing test cases as opposed to tests generated by covering syntactic changes.

***Keywords*-Symbolic Execution, Testing, Evolving Programs**

I. Introduction

Test Suite Augmentation is based on the assumption that a program P is sufficiently covered by an existing test suite. When P is changed to P', only those test cases are generated and augmented that witness the change in behavior. For transformational programs, the semantic difference across two program versions can be precisely determined if for both program versions we know exactly how every possible program input relates to the computed program output - the transformation function underlying the program version. We can then find the semantic difference across two program versions by comparing their transformation functions and finding inputs for which the transformation functions yield different outputs in the two versions.

How can we compute the transformation function for both program versions? Exploring all program paths [1]–[3], or determining all possible/feasible symbolic values of the program output [4] does not scale. Many paths through a changed program are not affected by the changed statements at all.

In fact, it has been observed that a semantic change originates in a syntactic change [5]–[9]. TSA approaches that are based on the Propagate-Infect-Execute principle (PIE [10]), seek to generate test cases that *execute* a changed statement, *infect* the program state, and *propagate* the impact to the output. Santelices et al. employ static symbolic execution to propagate the impact of a single change[1] up to a certain distance [8], [11]. Qi et al. [7] uses dynamic symbolic execution to find a program path that satisfies the PIE requirements for a *single change* - multiple changes across program versions are not allowed unless the changes are completely independent or non-correlated.

This paper extends previous work along both axis. Our Change-based Path Exploration (CBPE) method finds not one but *all paths that satisfy the PIE* requirements not for a single but *multiple changes*. As a result of the path exploration a concise portion of the transformation function is computed w.r.t. these syntactic changes - the partial semantic signature. Using the partial semantic signatures of the original and the modified version of a program, semantic differencing generates change-revealing test cases as witnesses of semantic changes. Our experiments show that subsequent semantic differencing exposes 1.5 times more changes in behavior than test cases that are generated from exploration of only the modified program P' (e.g., [9], [12], [13]).

Our overall approach for test suite augmentation proceeds as follows. We seek to determine the differences between the transformation function of the two program versions being analyzed. However, this is achieved in a scalable fashion by grouping paths based on whether they reach the syntactic changes, whether they propagate the syntactic changes, and

[1]Multiple changes are supported if there are program paths in which both changes are not executed.

how the propagation determines the output value (defined as a function of the input values). Moreover, after computing a summary of the transformation function of the two program versions via path exploration, we compute the difference between these two computed transformation functions to find the "semantic changes". For each of these semantic changes formulae, we can generate a test case via constraint solving. These are the augmenting test cases.

The main contributions of this paper can be summarized as follows – (i) we present a test suite augmentation method which works in the presence of multiple changes across program versions, and (ii) the comprehensive nature of our test suite augmentation method seeks to uncover and group all changes in observable behavior across two program versions.

II. All Change-Executing Paths

The first step to Change-based Path Exploration is finding all paths that execute a change. But how to find an adjacent path that executes a change, when an initial input does not?

Dynamic path exploration is the means of finding all feasible paths in a program via a combination of concrete and symbolic execution (also called concolic execution) [1], [2]. When a program is executed, the evaluation of the executed branch instances (e.g., `if`, `goto`) determines on which path the program execution proceeds. After this concolic program execution, a *path condition* is generated. The path condition is a quantifier free first order logic formula on program inputs. Any test input satisfying the path condition of a path π is guaranteed to exercise exactly the path π. In order to find "adjacent" paths, an exploration algorithm generates new conditions by negating the branch conditions in the path condition one by one. A Satisfiability Modulo Theory (SMT) solver uses these conditions to generate new program input. This procedure repeats until there is no new path to be explored.

Many branches are not relevant for reaching a given statement c (and do not need to be negated during path exploration). If some program input executes c, then obviously those branches upon which the execution of c (control-) depends are relevant. If some program input does not execute c, then those branches that *upon negation* lead the subsequent exploration towards the execution of c are relevant. In summary, all branches b_i in the execution trace are relevant upon which c control-depends and those branches that contribute in computing the value of b_i.

A. Definitions

The *relevant slice condition* rsc w.r.t. a statement instance c_i ensures that the variables used in c_i have the same symbolic values for all input satisfying rsc [4]. If c_i is not executed in π, then the relevant slice condition is unsatisfiable.

Definition 1 (*Relevant Slice Condition [4]*)

Given an execution trace π and a statement instance c_i in π, the relevant slice condition $rsc(c_i, \pi)$ in π w.r.t. c_i is the path condition computed over the statement instances of π which are included in the transitive closure of dynamic data, control and potential dependence of π.

Statement instance s_i *potentially depends* on conditional statement instance b_i in path π iff. there exists a variable v used in s_i such that (1) v is not defined between b_i and s_i in π but there exists another path σ from b_i to s_i along which v is defined, and (2) evaluating b_i differently may cause this untraversed path σ to be executed [14], [15]. Note that c_i does *not* simply denote the i-th execution of statement c. Instead, c_i is an instance of c that has a unique *execution index* as defined in [16]. This allows to derive the corresponding statement instance in another execution trace.

While the relevant slice condition is concerned with program elements that contribute in *computing* a statement instance, the reachability condition is concerned with elements that contribute in *reaching* a given statement.

Definition 2 (*Reachability Condition*)

The reachability condition of trace π w.r.t. statement c is satisfied by every program input that satisfies the relevant slice conditions of all instances s_i in π of every statement s that c transitively statically control-depends on:

$$reach(c, \pi) \stackrel{def}{=} \bigwedge_{\delta(s_i, s, \pi) \wedge c \rightsquigarrow_c s} rsc(s_i, \pi)$$

If an input t_0 executes statement instance c_i of c, then all inputs t_1 satisfying the reachability condition of the trace of t_0 w.r.t. c execute c_i. In fact, then all instances of c executed by t_0 are also executed by t_1 and vice versa (cp. Theorem 1 on page 5). The predicate $\delta(s_i, s, \pi)$ holds if s_i is an instance of statement s executed in the trace π. The predicate $c \rightsquigarrow_c s$ holds if statement c transitively statically control-depends on statement s. That is, the execution of c statically depends on the evaluation of branch instance s_i, which by the properties of relevant slice conditions are always evaluated in the same direction - in favor of execution of c or not - for input satisfying the same reachability condition.

B. Example

The code example in Figure 1 shall depict the underlying concepts of the reachability condition computed w.r.t. the changed statement in line 9. Note that it shows source code while our implementation analyzes byte code. The program takes two variables `i` and `j` as input. Depending on these values, the change in line 9 is executed or not. The procedure call in line 10 shall contain an unknown number of branches.

Executing the program with [i=0,j=1] yields the execution trace $\pi = [1_0, 2_0, 3_0, 5_0, 6_0, 5_1, 7_0, 8_0, 10_0, .., 10_1, 11_0]$. The changed statement c in line 9 is not executed. Path

```
1  input(i,j);
2  a = 0; b = 0;
3  if(i>0)
4    a=1;
5  for(c=0; c < j; c++)
6    b += c;
7  if(j>0){
8    if(a>0)
9      o=2; //change: o=5
10 b+=procedure(i,j,o);
11 output(o);
```

Figure 1. Reachability of a change

exploration based on the reachability condition of c finds all paths that execute c. To this end, the reachability condition contains branches that *upon negation* might contribute in reaching c. In this example the the reachability condition of π w.r.t. changed statement c contains the branch conditions

1) in line 8 because it contributes in *not* reaching c (static control-dependence)
2) in lines 3 and 7 because they contribute in evaluating the branch in line 8 in the direction that does not favor execution of c (in relevant slice of branch in line 8)

The generated reachability condition for trace π is $(a_0 = 0) \wedge (i_0 \leq 0) \wedge (j_0 > 1) \wedge (a_0 \leq 0)$. Input satisfying this reachability condition never executes the changed statement c. The branch in line 3 is always evaluated to false and the branch in line 7 to true. It is not relevant how often the loop in line 5 is executed or how many branches are executed in the procedure called in line 10. One of the branch conditions negated during path exploration is that of the branch in line 3, therefore leading the exploration towards the execution of c.

III. All Change-Propagating Paths

The properties of reachability conditions can be used during path exploration to find all paths that execute at least one change. For the impact of a change to be observable at the output, our *Change-Based Path Exploration* (CBPE) finds all paths that execute at least one change, at least one output statement and that compute the output differently. This leads to the notion of *change condition*, defined formally in the following. Our Change-based Path Exploration (CBPE) method proceeds by computing change conditions.

Definition 3 (*Change-Condition*)

The change condition of execution trace π w.r.t. a set of changed statements C and a set of output statements O is i) a conjunction of the reachability conditions of π w.r.t. every $c \in C$ if **no** *instance c_i of $c \in C$ is executed in π, or ii) a conjunction of the reachability conditions of of π w.r.t. every $c \in C$ and $o \in O$, and the relevant slice conditions of π w.r.t. every instance o_i of every $o \in O$ if* **at least one** *instance c_i of $c \in C$ is executed in π:*

$$change_{(C,O,\pi)} \stackrel{def}{=} \begin{cases} \bigwedge_{c \in C} reach(c,\pi) & \text{if } \neg\delta(c_i,c,\pi) \\ \left(\begin{array}{l} \bigwedge_{c \in C} reach(c,\pi) \wedge \\ \bigwedge_{o \in O} reach(o,\pi) \wedge \\ \bigwedge_{\delta(o_i,o,\pi)} rsc(o_i,\pi) \end{array}\right) & \text{if } \delta(c_i,c,\pi) \end{cases}$$

If for a given input there is no change executed, branches that are concerned with reaching an output statement can be ignored. Then, the change condition is the conjunction of the reachability conditions of all syntactic changes. Otherwise, the change condition is the conjunction of the reachability conditions of all changes, all output statements, and the relevant slice conditions of all executed output statement instances. The change condition ch captures the unique symbolic values of every output statement instance when executing those changes, that are also executed in π, for every input satisfying ch (cp. Theorem 2 on page 5).

In fact, the CBPE algorithm matches the algorithm of Path Exploration based on Symbolic Output [4] (PESO) only that instead of the relevant slice condition, the change condition computed over the execution traces is used during exploration. CBPE, like PESO, requires the reordering of branch conditions in terms of the dependencies of their branch instances before negating the last branch condition in order to be complete. The proof, that CBPE explores the complete input domain is similar to the proof of completeness of PESO (cp. Theorem 2 in [4]) but left out due to the lack of space.

A. Change Conditions partition Input Space

A path condition imposes constraints onto the program input, so that every input satisfying a path condition exercises the same program path [1]. During path exploration based on path conditions the complete input domain is divided into disjoint input partitions [17] in terms of all feasible program paths that an input exercises.

Path exploration based on the reachability condition w.r.t. a given statement s partitions the input domain in terms of the execution of s. If an input in such a partition executes s, then every input in that partition executes s. That is, input that reaches s on a "similar" path is in the same partition. In this context, path exploration is the attempt to find a witness for each adjacent partition until the complete input domain is covered or some (time/coverage) budget is exhausted.

Path exploration based on the change conditions (i.e., CBPE) employs the properties of change conditions to partition the complete input domain in terms of the reachability of changed statements and how their effects are propagated to one or more output statements. Because branches that are not relevant w.r.t. s are pruned, the reachability conditions as well as change conditions in general represent groups of "similar" paths, as opposed to a single path. Thus, they are weaker than the individual path conditions in the program.

B. Partial Semantic Program Signature

For semantic differencing, that portion of the transformation function of a program is required that covers the impact of a syntactic change on the output. To this end, our CBPE finds all input partitions, for which an input executes at least one change, at least one output statement, and computes the values used in an output statement instance differently. Input partitions that do not execute a change or an output statement are not relevant for computing the semantic difference between two program versions. The relevant portion of the transformation function of a program P is called *partial semantic signature* of P.

The partial semantic signature can be infinite, so that its precise computation is incomplete in finite time [18]. Particularly, this is the case in the presence of loops and recursive method invocations that transitively depend on the program input. For instance, during path exploration a loop is unrolled once in the first execution, twice in the second execution and so on. That yields possibly infinitely many change partitions. The completeness of a program signature may be achieved by abstraction (e.g., abstract summary [18] / domain [19]) but at the cost of conciseness. If the execution of a loop body is relevant for change execution or propagation, then there is no reason not to explore all of its (relevant) iterations.

Every execution of the subject program during path exploration adds knowledge about the transformation function in this program. The more change-partitions are explored the more information exists in the partial semantic signature. After a certain time budget is exhausted or input partition coverage is satisfied, semantic differencing is executed on the partial semantic signature that is computed until then. The task of exposing many semantic changes quickly in this possibly infinite space of change partitions can be reduced to a search problem - which of the adjacent, unexplored change partitions is to be explored next? In this context, a contribution of this paper is the significant reduction of the search space by grouping "similar" program paths in terms of their relevance for executing and propagating changes.

IV. Semantic Difference

A test case that yields different output for programs P and P' is a witness of the change in behavior when P evolves to P'. The previous sections discuss how to find all paths that execute a changed statement and propagate the impact to an output statement. After CBPE of P and P', for every input that executes a change the symbolic values of the output are known - the partial semantic signature. Intuitively, a semantic difference can be observed when for overlapping change-partitions the output is different.

A. Example

Figure 2 shows two programs P and P' on the left side, the comment in line 2 being the *syntactic change* that if applied leads to P'. The right side shows the respective partitions found by our change-based path exploration algorithm. An example for a test case that reveals changed behavior would be $[i = 0]$. Executed upon P the output is set to 0 while executed upon P' the output is set to 6.

```
1 input(i, j);
2 i=i; //i=i+1;
3 if(i>0)
4   o = i+5;
5 else
6   o = 0;
7 output(o);
```

(a) Programs P and P'

	Input	Output
P	$i \leq 0$	$o = 0$
	$i > 0$	$o = i + 5$
P'	$i + 1 \leq 0$	$o = 0$
	$i + 1 > 0$	$o = i + 1 + 5$

(b) Change Partions

Figure 2. Semantic Difference

B. Quantifying Semantic Change

For the example in Figure 2, there are infinitely many change-revealing test cases, $i \geq 0$. Is there a finite number that measures the change in behavior equally precise?

We now formally define the notion of a *semantic change* with respect to the computed semantic signatures of two versions P and P' of a program. Figure 2.b) shows the computed semantic signatures of the two program versions in Figure 2.a). Given two such semantic signatures, we can compute the set of semantic changes as follows. Let the partial semantic signature of program P be:

$$\varphi_1 \mapsto Out_1, \ldots, \varphi_m \mapsto Out_m$$

This means when φ_x (the change-condition computed using the program input variables for input that executes at least one change) holds, a set of output statement instances are executed and assigned symbolic values (again an expression over the program input variables) - Out_x. Similarly, let the partial signature of program P' be:

$$\psi_1 \mapsto Out'_1, \ldots, \psi_n \mapsto Out_n$$

The set of semantic changes between P and P' are then computed by computing the following $m{*}n$ formulae (where $1 \leq i \leq m$, $1 \leq j \leq n$)

$$\varphi_i \wedge \psi_j \wedge Out_i \neq Out'_j$$

and finding those which are satisfiable. Each satisfiable formula corresponds to one *semantic change*, and it can be solved to find an augmenting test. Note that our computation of semantic changes is based on the computation of semantic signatures of the program versions. Thus, if the computed semantic signatures are incomplete w.r.t. the effects of all syntactic changes (for scalability reasons), our computation of sematic changes (via semantic differencing) will also be incomplete. Also note, that $\varphi_1, .., \varphi_m$ and $\psi_1, .., \psi_n$ are change-conditions and capture the knowledge of which syntactic changes have been executed in P and P' before the

corresponding semantic change is observable at the output. That can help the software tester to pinpoint the root cause of a semantic change.

Intuitively, to compute the semantic changes, we find overlapping change partitions in which the outputs differ. Taking the partitions in Figure 2.b) as example, we test the satisfyability of the following (simplified) formulas:

1) $(i \leq 0) \wedge (i+1 \leq 0) \wedge (0 \neq 0)$
2) $(i > 0) \wedge (i+1 \leq 0) \wedge (i+5 \neq 0)$
3) $(i \leq 0) \wedge (i+1 > 0) \wedge (0 \neq i+1+5)$
4) $(i > 0) \wedge (i+1 > 0) \wedge (i+5 \neq i+1+5)$

A solver generates two satisfying witnesses of behavioral difference: $t_3 : [i = 0]$, and $t_4 : [i = 1]$. Note that infinitely many test cases ($i \geq 0$) would expose not more than these two semantic changes. However, our method will report only two test cases revealing the two semantic changes.

V. Proofs

The proofs of both theorems are based on the properties of relevant slice conditions. If the relevant-slice conditions of two paths π_0 and π_1 w.r.t. statement instance c_i are the same, then the variables used in c_i have the same symbolic values in π_0 and π_1. The proof of this theorem in [4] is based on the proof of the stronger Lemma 3.2 in [20]. For easy reference, the lemma shall be repeated here.

Lemma 1

Let t_0 and t_1 be two inputs, π_0 the trace of t_0, and π_1 the trace of t_1. Let c_i be a statement instance in π_0. If t_1 satisfies the relevant slice condition of π_0 w.r.t. c_i, then i) c_i is also executed in π_1, ii) the variables used in c_i in π_1 have the same symbolic values as in π_0, and iii) the relevant slice conditions of π_0 and π_1 w.r.t. c_i are exactly the same.

Theorem 1 states that if an input t_0 executes an instance c_i of statement c, then all inputs t_1 satisfying the reachability condition computed over the trace of t_0 w.r.t. c, execute c_i and t_1 and t_0 have the same reachability condition w.r.t. c.

Theorem 1

Let t_0 and t_1 be two inputs, π_0 the trace of t_0, and π_1 the trace of t_1. Let c be a statement. If t_1 satisfies the reachability condition of π_0 w.r.t. c, then i) all instances c_i of c executed in π_0 are also executed in π_1 and vice versa, and ii) the reachability conditions of π_2 and π_1 w.r.t. c are exactly the same.

Proof: Per Definition 2, $reach(c, \pi_0)$ is satisfied by every program input that satisfies $rsc(s_i, \pi_0)$ of all instances s_i in π_0 of every statement s that c transitively statically control-depends on. (A_1): Because t_1 satisfies $reach(c, \pi_0)$, t_1 also satisfies every $rsc(s_i, \pi_0)$, if $\delta(s_i, s, \pi_0)$ and $c \rightsquigarrow_c s$.

We prove i) all instances c_i of c executed in π_0 are also executed in π_1. i.a) If $(c \not\rightsquigarrow_c s)$, i.e., c does not statically control-depend on any statement s, then every instance of c, including c_i, is in all paths, including π_1. i.b) If $(c \rightsquigarrow_c s)$ and $\delta(s, s_i, \pi_0)$, i.e., the execution of c depends on the evaluation of the branch condition of instance s_i, then an instance c_i dynamically control-depends on s_i. By assumption and (A_1), $t_1 \vDash rsc(s_i, \pi_0)$. By Lemma 1, $t_1 \vDash rsc(s_i, \pi_1)$. That is, the variables used in s_i have the same symbolic values in π_0 and π_1. Thus, if c_i is executed in π_0, it is also in π_1. i.c) The case that $(c \rightsquigarrow_c s)$ and $\neg\delta(s, s_i, \pi_0)$, i.e., for every s that satisfies $(c \rightsquigarrow_c s)$ there exists no instance s_i of s in π_0, is impossible to satisfy by the theorems of transitive static control-dependence. This can easily be shown. Assuming $(c \rightsquigarrow_c s)$ and $\neg\delta(s, s_i, \pi_0)$, then the (non-)execution of s transitively depends on the evaluation of another statement instance r_i in π_0 that is exactly evaluated in the direction that does not favor the execution of s. This is a contradiction because s transitively control-depends on r_i. Concluding cases i.a), i.b), and i.c), all instances c_i of c executed in π_0 are also executed in π_1. ii) The application of Definition 2, Lemma 1 and i) onto $reach(c, \pi_0)$ derives $reach(c, \pi_1)$ and vice versa. ■

Theorem 2 states that if an input t_0 executes an instance c_i of changed statement $c \in C$, an instance o_i of an output statement $o \in O$, and computes values v for all variables used in o_i, then all inputs t_1 satisfying the change-partition of the trace computed over t_0 execute c_i and o_i and compute values v for the variables used in o_i.

Theorem 2

Let t_0 and t_1 be two inputs, π_0 the trace of t_0, and π_1 the trace of t_1. Let C be the set of changed statements and O the set of output statements. If t_1 satisfies the change condition of π_0 w.r.t. C and O, then i) all instances c_i of $c \in C$ executed in π_0 are also executed in π_1 and vice versa, ii) the change conditions of π_0 and π_1 w.r.t. C and O are exactly the same, and if there also exists an instance c_j of $c \in C$ in π_0, then iii) all instances o_i of $o \in O$ executed in π_0 are also executed in π_1 and vice versa and iv) the variables used in o_i have the same value in π_0 and π_1.

Proof: By Definition 3, $change(C, O, \pi_0)$ is 1) the conjunction of $reach(c, \pi_0)$ for every $c \in C$ if $\neg\delta(c_i, c, \pi_0)$, i.e., there is no instance c_i of $c \in C$ executed in π_0, or ii) the conjunction of $reach(c, \pi_0)$, $reach(o, \pi_0)$, and $rsc(o_i, \pi_0)$ for every $c \in C$ and o_i of $o \in O$ if $\delta(c_i, c, \pi_0)$. Thus, if $t_1 \vDash change(C, O, \pi_0)$ it also satisfies all its reach- and relevant slice conditions. i) By Theorem 1, all instances c_i of $c \in C$ executed in π_0 are also executed in π_1 and vice versa. ii) The application of Definition 3, Lemma 1, and Theorem 1 onto $change(C, O, \pi_0)$ derives $change(C, O, \pi_1)$, and vice versa. iii) By Theorem 1, if there exists an instance c_j of $c \in C$ in π_0, then all instances o_i of $o \in O$ executed in π_0 are also executed in π_1 and vice versa. iv) By Lemma 1, if there exists an instance c_j of $c \in C$ in π_0, then the variables used in o_i have the same value in π_0 and π_1. ■

VI. Implementation

A. Change-based Path Exploration

The computation of the *change condition* for an input executed on a program has been implemented into JSlice [21] - an open source dynamic backward slicing tool for Java bytecode. Previously, JSlice was extended to incorporate dynamic symbolic execution [4]. By augmenting the dynamic symbolic execution capability of JSlice, it is now also possible to compute reachability and change conditions. The user specifies the set of changed statements, a set of output statements and the initial input for a given program.

The dynamic symbolic execution of the program with the given input in JSlice generates a compressed trace upon which the change-condition is computed. In order to find adjacent change-partitions, every branch in the change condition is negated one by one. The order in which branch conditions are negated is derived from the branch dependencies computed by JSlice. The generated formulae are used by the Satisfiability Modulo Theory solver Z3 [22], to compute the next program input that witnesses an adjacent change-partition until all change-partitions are explored.

In order to compute the reachability condition w.r.t. a statement c, it is checked for every conditional construct instance in the trace whether there is a path to c along the edges of the control-flow graph (CFG). This matches the procedure for computing potential dependencies and is therefore only slightly more expensive than computing the relevant slice condition. If the conditional construct b_i is found to have a path to s in the CFG, then JSlice computes the relevant slice condition w.r.t. b_i.

Path exploration can be started in two modes. Newly generated conditions are put on top of a stack. In the default mode the solver takes the condition from the bottom of the stack, solves the condition and the input is executed next (First In, First Out; *FIFO*). In the second mode, the solver takes from the top of the stack (Last In, First Out; *LIFO*). In exhaustive exploration, obviously, there is no difference in the number of partitions found within the complete input domain. Within a time bound however, the order in which partitions are explored matters. The LIFO-mode resembles a *depth-first search*. For example, if the the initial input executes n branch instances and the negation of the $n-th$ branch executes a path that has $n+m$ branch instances, then LIFO proceeds with the negation of the additional m branch instances before negating the rest of the n branch instances from the initial input. It takes a while before the partition is explored that corresponds to the negation of the first of n branch instances of the initial input. The default FIFO mode of CBPE resembles *breadth-first search*. In the above example, the partitions corresponding to the negation of the n branch instances from the initial input are explored first. After that the partitions corresponding to the negation of m branches are explored in the same manner.

There are more than 200 different bytecode types in the Java Virtual Machine instruction set, and all of them are handled in our implementation. However, due to the virtual machine, the current version of JSlice is based on, we cannot handle programs with multi-threading and reflection. As is the nature of dynamic symbolic execution [1], deterministic but unknown values are discretized. Particularly, this applies to library calls and native calls. The use of concrete values in these cases yields a stronger computed change condition as defined. Also, the current implementation of JSlice fails to compute a trace from program executions that do not terminate gracefully (i.e., program chrashes). For instance, the bytecode $getfield$, used to access a variable in an object, might throw a `NullpointerException`. The output statement instance o_i control-depends on statement $getfield$ because the evaluation of $getfield$ decides whether o_i is executed or not. Because JSlice does not consider program chrashes during the computation of control-dependence, the computed change-condition may be weaker as defined.

B. Semantic Difference

Change-based Path Exploration is executed on both versions of the program - P and P'. This JSlice-extension has been modified to highlight every instance of an output statement in the change-condition formula file and give it an execution index [16]. If two formulae of P and P' reference a different number of output instances, then the conjunction of both change-condition formula is generated and solved by the SMT-solver Z3. If the number of output instances matches for both formula and at least one output instance is referenced, then the conjunction of both formulae is generated with the additional condition that the values of at least one output instance-pair in P and P' have to be different.

VII. Experiments

A. Infrastructure

Three exploration techniques are executed upon 32 different program versions. The exploration of the modified versions of the subjects yields a set of change-revealing test cases, which need to be consolidated w.r.t. the number of exposed semantic changes. The increase in number of exposed changes is also shown, when the behavior of the original version is taken into account - by semantic differencing.

1) Subjects: The subjects **TCas** and **Printtokens2** are C-programs provided in the Software-artifact Infrastructure Repository (SIR) [23]. Both subjects have mostly a single syntactic change between the original and a modified version of the source code. TCas is an aircraft collision avoidance system with 173 Lines of Code (LoC), while Printtokens2 is a lexical analyzer of 570 LoC. For our experiments, we use the first 20 versions of TCas and all 10 versions of Printtokens2.

NanoXML is an XML reader and writer with about 8,000 LoC provided in the SIR. It has six original versions, each of which has about six seeded syntactic changes. We use version `v1` as P and activate three changes - `F_SP_HD_1`, `F_SP_HD_2`, and `F_SP_HD_3` - to derive P'.

(i) Barcode4J v1.12 (ii) Barcode4J v1.17

Figure 3. DataMatrix generated by two versions of Barcode4J for the same input "witness{"

Barcode4J[2] is a flexible generator for barcodes that has 135 classes and about 17,000 LoC. The comments in the project repository show that version v12 of the class `DataMatrixHighlevelEncoder` has several bugs that are corrected until version v17 of the same class. In version v12, in some cases the character "{" is encoded wrongly, so that the generated data matrix reads "Z" instead. In other cases "{" is encoded correctly[3]. A working example is shown in Figure 3. Furthermore, in some cases the character "´ " might completely abort the generation of the corresponding data matrix. In version v17 the generation works as expected.

Our subjects are refactored in order to prepare them for analysis by JSlice, the tool that implements our approach. Particularly, the two C-programs TCas and Printtokens2 are transformed into Java-programs, `switch`-statements are transformed into `if`-statements, a simple version of the classes String, StringBuffer, Vector and Map are used instead of the native implementations and exception handling is modified because JSlice cannot handle program crashes, yet. Instead, the program terminates in an error state. When the refactoring is finished, the complete source code is copied and the given syntactic changes are applied to derive versions P' from the original program P, thereby ensuring that changes in behaviour only originate in explicitly marked syntactic changes.

2) Path Exploration Techniques: In our experiments we seek to expose the maximal number of exposed semantic changes overall or within limited time. In order to compare, we choose three path exploration techniques. *Path Exploration based on Symbolic Output* (PESO) is an exploration technique to find all symbolic values a post-dominating output statement can have [4]. Therefore, if input is bounded and the program always terminates, PESO finds the complete semantic signature of a program. *Change-based Path Exploration* (CBPE) is the approach presented in this paper and finds all possible values of an output statement when executing at least one change. REACH is a derivation of CBPE that finds all paths that execute a change and does *not* proceed to find paths that propagate the impact of that change to the output.

Technically, PESO is based on the relevant slice condition w.r.t. the output statement instances, CBPE is based on the change condition w.r.t. changes and output statements, and REACH is based on the reachability condition w.r.t. the changes. All path exploration approaches start with the same initial input.

3) Computation of Semantic Changes: The exploration techniques are executed upon the *modified versions* P' of the subjects. The output of an exploration is a set of test cases each of which witnesses an explored partition of the input domain in P'. Executing such a test case upon the original version P and P' reveals whether there is an observable change in output or not. Therefore, for every generated test case in the test suites computed by PESO, REACH and CBPE as exploration of P', the change conditions ch_P and $ch_{P'}$ are computed for its execution in P and P'. A semantic change is exposed if for overlapping change partitions, the output is different. Consequently, if the change conditions of two change-revealing test cases t_0 and t_1 are the same, i.e., $t_0 \vDash ch_P$, $t_0 \vDash ch_{P'}$, $t_1 \vDash ch_P$, and $t_1 \vDash ch_{P'}$, then t_0 and t_1 expose the same semantic change. The *consolidated test suite* contains a single test case for every semantic change.

Change-based path exploration with subsequent semantic differencing (*CBPE_SD*) is the only technique in our experiments that employs knowledge about the original program P to generate test cases each of which witnesses a different semantic change.

B. Results for Exposed Semantic Changes

The experiments are conducted on the subjects to determine which path exploration technique is more likely to expose a semantic difference overall or in bounded time.

1) Exhaustive TSA: The chart in Figure 4 compares different path exploration techniques in terms of exposed semantic changes without any time constraints. The consolidated test cases generated for the syntactic changes in the 20 modified versions of TCas are executed on both the original and modified version of TCas to derive the number of exposed semantic changes.

The results suggest that even test cases that witness all different symbolic values of the output in the modified program, are insufficient in exposing the maximal number of semantic changes. PESO exposes on average 4.2 semantic changes while CBPE_SD reveals 6.5. The graphs of PESO and CBPE do overlap. That agrees with our assumption that only paths that execute a syntactic change can witness a semantic change. It is also not sufficient to find all paths only that execute a change. REACH reveals only one semantic

[2]Details at: http://barcode4j.sourceforge.net.

[3]E.g., the string "nowitness{" is encoded correctly while "witness{" is not.

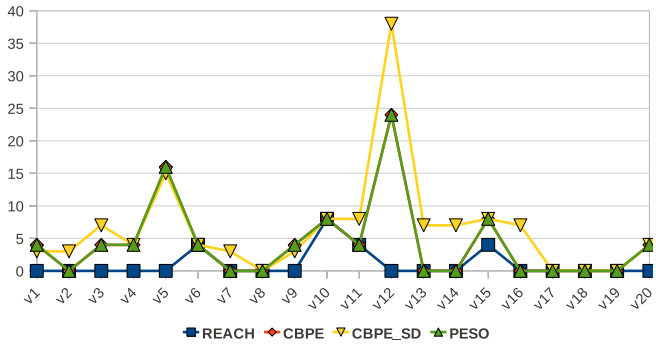

Figure 4. Exhaustive Path-Exploration - TCas

change on average. Thus, it is four times more likely to observe a semantic change, when the impact of the syntactic change is also propagated to the output. It is even six times more likely, when the semantic impact of the changed statements onto the output is differenced, so that for the same input the output is different for both versions P and P' of the program.

2) TSA in Bounded Time: There might be a given time budget a software tester can spend when testing the program. The vision of continuous testing is to support the developer with automatic testing procedures at the same time as he or she is writing the program. While it does not need to be exhaustive, any approach shall expose as many semantic changes within the given time constraints as possible.

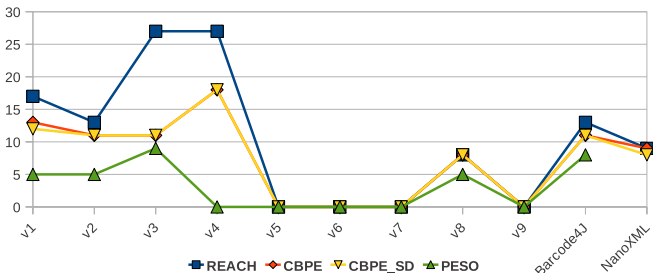

Figure 5. Path-Exploration in Bounded Time

The chart in Figure 5 compares different path exploration approaches in terms of exposed semantic changes within a given time frame. Four different path exploration algorithms are executed on the nine modified versions of Printtokens2 in 10 minutes, NanoXML in 30 minutes, and Barcode4J in 120 minutes in FIFO-mode. The subjects are shown on the X-axis while the number of exposed semantic changes is shown on the Y-axis. Like with unbounded time, the consolidated test cases generated for the modified version P' are executed on P and P' to derive the number of exposed semantic changes. For NanoXML we chose an output that is not executed on every path. Hence, there is no result for PESO which needs the output to be executed on every program path.

For the tested subjects, REACH exposes on average 10.4 semantic changes within a given time bound. That is significantly more than CBPE and CBPE_SD with an average of 7.4 and 7.2, respectively. That means, for these subjects it is more important to find different paths that execute a change than finding paths that propagate an executed change to the output. PESO is the path exploration technique that does not take the syntactic change into account and performs worst with 3.2 exposed semantic changes on average.

Furthermore, the chart suggests that subsequent semantic differencing does not expose any more semantic changes than CBPE alone if the time is bounded. That might have different reasons. For one, the syntactic changes might be of a certain category that already yields different output when executed, as for instance in Barcode4J. It might also be that the SMT-solver generates the input necessary to observe the semantic change at the output merely as coincidence, even though only P' is analyzed.

The results are shown for the experiments of path exploration techniques in FIFO-mode - path exploration as breadth-first search. The results for the execution in LIFO mode - path exploration as depth-first search - are similar. REACH exposes on average 13.5 semantic changes, CBPE and CBPE_SD follow with 9.8 exposed semantic changes on average, and PESO performs worst with about 5.4 exposed semantic changes. The average number of exposed semantic changes using FIFO strategy are higher because, one subject, Barcode4J, exposes between 35 and 50 semantic changes in LIFO- and only between 8 and 13 in the default FIFO-mode.

3) Conclusion: For exhaustive test suite augmentation, change-based path exploration with subsequent semantic differencing performs significantly better than any other approach in terms of number of exposed semantic changes. That means, even though test cases are generated that represent all possible symbolic value the output of modified program P' can have, they do not witness the maximum number of semantic changes. Only analyzing the modified program P' is not sufficient. In our experiments, subsequent semantic differencing exposes 1.5 times more semantic changes. Change-based path exploration (CBPE) exposes as many semantic changes as path exploration based on symbolic output (PESO), even though CBPE takes fewer runs. This is because only paths that execute a change are considered during CBPE exploration, while PESO considers all paths that contribute in computing the output.

For path exploration-based test suite augmentation within a specified time budget, it is significantly more important to find different paths that execute a change, than finding paths that propagate the effects of a change differently. Finding test cases that represent all symbolic values the output can have, without considering the syntactic changes, exposes less semantic changes in the same time frame, than if the syntactic changes are considered during path exploration.

VIII. Related Work

Certainly the simplest and most scalable approach to test suite augmentation is to use a random input generator, execute that input on both program versions and add such input to the augmenting test suite that yields different output. Orso and Xie discuss this approach in [24]. However, this technique is only exhaustive if every possible program input is generated eventually. Also, if a syntactic change is made in a path that has a low probability of being executed, the probability to expose a semantic change is at least as low.

A promising line of work seeks to find test cases that *Execute* a syntactic change, *Infect* the program state, and *Propagate* the effects of the change to the output (PIE [10]). Santelices et al. present an approach that employs static symbolic execution to generate requirements so that a satisfying test case propagates the effects of an executed change up to a maximum distance along the dependency chain [8], [11], [25]. The authors consider only the propagation of changes that have a change-free path to the program entry. Due to the maximum propagation distance, the test case may or may not witness a semantic change. Also, it is not considered, how a change is actually reached. Using dynamic symbolic execution, Qi et al. can generate from random initial input a test case which executes a change and propagates the effects to the output [7]. The presented algorithm is very effective in finding for a single change a single witness that satisfies the PIE-requirements. However, our target is to find *all witnesses* that satisfy the PIE-requirements for *multiple syntactic changes*.

Xu et al. discuss a coverage-driven approach to test suite augmentation [12], [26]. Test cases are generated that stress elements in modified progam P', such as nodes or edges in a control-flow graph, that are not covered by an existing test suite for the original version P. While code coverage is an excellent measure of adequacy for regression test suites, there might be better measures for augmenting test suites - e.g., the number of exposed semantic changes. An augmenting test suite that re-establishes 100% code coverage is insufficient for two reasons. Firstly, the mere execution of all changed elements does not guarantee the propagation of their effects to the output (cp. [25], [27]) and secondly, even if the effects would be observable at the output, the mere analysis of P' exposes only a subset of all semantic changes, as shown by our experiments.

Harman et al. debate TSA in the presence of multiple syntactic changes using concepts and terms of mutation-based test data generation [13]. Multiple syntactic changes that may semantically interfere [28]–[30], are called higher-order mutants. Non-interfering, single changes are first-order mutants. A test case weakly kills a mutant if it executes a change and strongly kills it if it also propagates the effects to the output. In three phases Harman et al. try to find a test case that strongly kills a higher-order mutant. (1) Static Analysis: In the CFG a path is determined that may execute all changes. (2) Weak Kill: Using this information and dynamic path exploration, a feasible path is determined that executes all changes. (3) Strong Kill: A search algorithm is employed to find a path that is more likely to propagate guided by a fitness function. The possibly arbitrary order and number in which changes can be executed to cause even more subtle faults is not considered in [13]. Our CBPE cum differencing finds not a single but *all* possible ways to *strongly kill* first- and higher-order mutants (reachability condition = all weak kills). Furthermore, our approach does not need to be repeated for each possible mutant, for which, as the authors say, "the cost would potentially be prohibitive" [13]. Instead, we consider all mutants at the same time. For example, given 2 changes c_1 and c_2 — if there exists a path that propagates only the effects of c_1, and another path that propagates the effects of c_1 *via* c_2 to the output, then CBPE finds both paths.

Taneja et al. present eXpress as TSA technique based on the PIE-principle that explores all paths in P' except for those that guarantee *not* to execute a change (B_E) and not to propagate the change (B_P) [9], [31]. In fact, eXpress prunes all branches $b \in B_E$ ($B_P \subseteq B_E$) from the exploration space that *guarantee not to execute a change*. In contrast, our CBPE prunes branches that do *not contribute in reaching a change*. What does that mean? Consider the example in Figure 1 on page 3. Our CBPE prunes the branch in line 5 from the exploration space, because it does not contribute in reaching c. However, there is no branch in Figure 1 that guarantees not to reach c that can be pruned by eXpress. Moreover, our CBPE prunes branches that do not contribute in reaching or computing the output (e.g., all branches in the procedure called at line 10). In summary, our CBPE prunes more irrelevant branches. It explores bigger input partitions, thereby needing less executions of the subject to cover the complete input domain. Also, Tuneja et al. consider only the analysis of P', which is insufficient for finding all semantic changes (cp. [27]).

Approaches following the PIE-principle are rather syntactic. An observable difference in program state is propagated from one program element to the next until a maximal distance is reached or the difference is observable in the output [7], [8], [10]. Another stream of work employs *semantic differencing* on the transformation functions of P and P' to determine input for which the output is computed differently. Exploiting the fact that both versions are syntactically largely similar, the behaviour of common code fragments can be summarized as uninterpreted functions yielding *abstract method summaries*. Person et al. introduce Differential Symbolic Execution (DSE) to derive differential method summaries [18]. Here, abstract summaries are an attempt towards the complete account of behavioural difference (i.e., for the complete input domain). Kawaguchi et al. do not seek semantic difference but whether two

program versions behave equivalent for input satisfying a user-provided set of conditions [19]. The behavior of all functions that do not appear along the chain of method calls down to the modified function is summarized by uninterpreted functions. However, we claim the application of abstract method summaries does not meet the needs of comprehensive TSA. Relevant information on finding input that reaches a change or propagates its effects to the output is removed. Consider the example in Figure 1. The code blocks in lines 1 to 6 have not changed from one version to the other. DSE substitutes the behavior for the common code fragment by an uninterpreted function. The computed differential summary captures that the values of i and j possibly influence the value of o in both versions (via defining a and b). The exact symbolic input values ($i \geq 0$) or even their existence, however, are only known in terms of that uninterpreted function. Moreover, while the concepts of conditional equivalence [19] are not suitable for comprehensive TSA, vice versa, CBPE cum differencing can determine the conditions under which two programs behave equivalent, starting with those input partitions that do not execute a single syntactic change.

Qi et al. discuss Path Exploration based on Symbolic Output (PESO) as an approach to compute the *concise* semantic signature of a program [4]. As shown in our experiments, not considering the location of a syntactic change during path exploration reduces the number of exposed semantic changes within the same time frame.

IX. Discussion

A. Summary

In this paper the definitions and properties of reachability and change conditions are discussed and the respective theorems proven. Path exploration based on the reachability condition w.r.t. a statement c finds all disjoint input partitions [17] for which an input executes c. Change-based path exploration finds all disjoint input partitions for which an input executes at least one change and propagates its effects to at least one output statement. When CBPE is executed upon a program, a portion of its transformation function is computed - the partial semantic signature. CBPE scales because it groups relevant and irrelevant paths into partions. That is, CBPE computes (1) only the portion of the transformation function that is affected by syntactic changes, (2) the portion that is possible within a given time frame (3) as concise as necessary so that no semantic change is missed. The main concern of comprehensive TSA is to expose as many semantic changes as possible. However, if time is bounded, the task of exposing many semantic changes quickly can be reduced to a search problem - which of the adjacent, unexplored change partitions is to be explored next? The contribution of CBPE is to reduce the search space significantly without sacrificing conciseness.

CBPE is executed upon an original and modified program version to compute the partial semantic signatures. A semantic change is exposed when for overlapping change-partitions, the output is different. Our definition of semantic change allows to quantify change in behavior. It also preserves the information of which syntactic changes are executed before this semantic change is witnessed. Moreover, we claim that the number of exposed semantic changes are a better measure of adequacy for augmenting test suites than coverage-based criteria.

Our experiments suggest that semantic differencing exposes 1.5 times more semantic changes than test cases that represent all possible symbolic values the output of the modified program P' can have. Also, the mere execution of changed elements does not guarantee the propagation of their effects to the output. Finding paths that propagate the effects of a change to the output (plus differencing) increases the number of exposed semantic changes by a factor of six. However, when time is limited it is more important to find different paths that execute a change, than finding different paths that propagate the effects to the output.

B. Applications

Comprehensive TSA is only one application of path exploration based on a composition of reachability and relevant slice conditions. In Section VIII, we discuss the possibility to find all input partitions for which both program versions behave the same (cp. conditional equivalence [19])), starting with partitions that do not execute a change.

A statement s'' *semantically depends* on another statement s' if s' decides the value (dynamic data-dependence) or the execution (dynamic control-dependence) of s'' (cp. [32]). Semantic dependence can be computed precisely for a given trace or approximated for all traces (via syntactic dependence). Path exploration based on condition $change(s', s'', \pi)$ allows to *precisely compute for all traces* semantic dependence in terms of input partitions.

Another application of our work is in *change comprehension*. Thus, our work can be used to address questions of the following form. Is the output reachable after the execution of change c? Which changes semantically interfere [28]? What impact does the execution of change c'' *after* change c' have on the output o?

To answer the last question using the concepts defined in this paper, we can proceed as follows. Path exploration based on the condition $reach(c', \pi) \wedge change(c'', o)$ finds all input partitions that execute c', c'', and o and compute o differently. To find the combined impact of c' via c'' on the output o, test cases, generated by the path exploration have to be selected that execute the sequence: c', c'', o. Because by the theorems 1 and 2, if a test case that witnesses partition p executes c' first, c'' next, and o last, than every input in p follows this sequence.

ACKNOWLEDGMENT

The authors would like to thank Hoang D. T. Nguyen for his extension of JSlice to incorporate dynamic symbolic execution and the concurrent computation of branch dependencies.

REFERENCES

[1] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, 2005, pp. 213–223.

[2] P. Godefroid, "Compositional dynamic test generation," in *POPL*, 2007, pp. 47–54.

[3] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/SIGSOFT FSE*, 2005, pp. 263–272.

[4] D. Qi, H. D. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," in *ESEC/SIGSOFT FSE*, 2011.

[5] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *ESEC / SIGSOFT FSE*, 1999, pp. 253–267.

[6] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: a tool for change impact analysis of java programs." in *OOPSLA*, 2004, pp. 432–448.

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

[8] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *ASE*, 2008, pp. 218–227.

[9] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux, "express: guided path exploration for efficient regression test generation," in *ISSTA*. ACM, 2011, pp. 1–11.

[10] J. M. Voas, "Pie: A dynamic failure-based technique," *IEEE Transactions on Software Engineering*, vol. 18, pp. 717–727, 1992.

[11] T. Apiwattanapong, R. A. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold, "Matrix: Maintenance-oriented testing requirement identifier and examiner," in *TAIC PART*, August 2006, pp. 137–146.

[12] Z. Xu, "Directed test suite augmentation," in *ICSE*, 2011, pp. 1110–1113.

[13] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *ESEC/SIGSOFT FSE*, 2011, pp. 212–222.

[14] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London, "Incremental regression testing," in *Proceedings of the Conference on Software Maintenance*, ser. ICSM '93, 1993.

[15] T. Gyimóthy, A. Beszédes, and I. Forgács, "An efficient relevant slicing method for debugging," in *ESEC/SIGSOFT FSE*, 1999, pp. 303–321.

[16] B. Xin, W. N. Sumner, and X. Zhang, "Efficient program execution indexing," in *PLDI*, 2008, pp. 238–248.

[17] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. Softw. Eng.*, vol. 17, pp. 703–711, July 1991.

[18] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu, "Differential symbolic execution," in *SIGSOFT FSE*, 2008, pp. 226–237.

[19] M. Kawaguchi, S. K. Lahiri, and H. Rebelo, "Conditional equivalence," Microsoft, MSR-TR-2010-119, Tech. Rep., October 2010.

[20] D. Qi, H. D. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," National University of Singapore, http://dl.comp.nus.edu.sg/dspace/handle/1900.100/3347, Tech. Rep., March 2011.

[21] T. Wang and A. Roychoudhury, "Using compressed bytecode traces for slicing Java programs," in *ICSE*, 2004, pp. 512–521.

[22] L. M. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *TACAS*, 2008, pp. 337–340.

[23] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[24] A. Orso and T. Xie, "Bert: Behavioral regression testing," in *WODA*, 2008, pp. 36–42.

[25] R. Santelices and M. J. Harrold, "Applying aggressive propagation-based strategies for testing changes," in *ICST*, 2011, pp. 11–20.

[26] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, "Directed test suite augmentation: techniques and tradeoffs," in *SIGSOFT FSE*, 2010, pp. 257–266.

[27] M. Fisher, J. Wloka, F. Tip, and B. G. Ryder, "An evaluation of change-based coverage criteria," in *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, 2011.

[28] G. L. Thione and D. E. Perry, "Parallel changes: Detecting semantic interferences," in *COMPSAC*, 2005, pp. 47–56.

[29] W. Le and M. L. Soffa, "Path-based fault correlations," in *SIGSOFT FSE*, 2010, pp. 307–316.

[30] R. Santelices, M. J. Harrold, and A. Orso, "Precisely detecting runtime change interactions for evolving software," *Software Testing, Verification, and Validation, 2008 International Conference on*, vol. 0, pp. 429–438, 2010.

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

[32] A. Podgurski and L. A. Clarke, "A formal model of program dependences and its implications for software testing, debugging, and maintenance," *IEEE Trans. Softw. Eng.*, vol. 16, pp. 965–979, September 1990.