Path Exploration based on Symbolic Output

Dawei Qi, Hoang D.T. Nguyen, Abhik Roychoudhury
School of Computing, National University of Singapore
{dawei,nguyend1,abhik}@comp.nus.edu.sg

ABSTRACT

Efficient program path exploration is important for many software engineering activities such as testing, debugging and verification. However, enumerating all paths of a program is prohibitively expensive. In this paper, we develop a partitioning of program paths based on the program output. Two program paths are placed in the same partition if they derive the output similarly, that is, the symbolic expression connecting the output with the inputs is the same in both paths. Our grouping of paths is gradually created by a smart path exploration. Our experiments show the benefits of the proposed path exploration in test-suite construction.

Our path partitioning produces a semantic signature of a program — describing all the different symbolic expressions that the output can assume along different program paths. To reason about changes between program versions, we can therefore analyze their semantic signatures. In particular, we demonstrate the applications of our path partitioning in debugging of software regressions.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools, Symbolic execution*

General Terms

Experimentation, Performance, Reliability

Keywords

Path Exploration, Relevant Slice Condition, Symbolic Execution

1. INTRODUCTION

Programs follow paths. Indeed a program path constitutes a "unit" of program behavior in many software engineering activities, notably in software testing and debugging. Use of program paths to capture underlying program behavior is evidenced in techniques such as Directed Automated Random Testing or DART [8] - which try to achieve path coverage in test-suite construction.

Why do we attempt to cover more paths in software testing? The implicit assumption here is that by covering more paths, we are likely to cover more of the possible behaviors that can be exhibited by a program. However, as is well known, path enumeration is extremely expensive. Hence any method which covers various possible behaviors of a given program while avoiding path enumeration, can be extremely useful for software testing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.
Copyright 2011 ACM 978-1-4503-0443-6/11/09 ...$10.00.

```
1  int x,y,z; // input variables
2  int out; // output variable
3  int a;
4  int b = 2;
5  scanf("%d %d %d",&x,&y,&z);
6  if(x - y > 0) //b1
7      a = x;
8  else
9      a = y;
10 if(x + y > 10) //b2
11     b = a;
12 if(z*z > 3)  //b3
13     printf("square(z) > 3 \n");
14 else
15     printf("square(z) <=  3 \n");
16 out = b; //slicing criteria
```

Figure 1: Sample program

We note that software testing typically involves checking the program output for a given input - whether the observed output is same as the "expected" output. Hence, instead of enumerating individual program paths, we could focus on all the different ways in which the program output is computed from the program inputs. In other words, we can define an output as a symbolic expression in terms of the program inputs. Thus, given a program P, we seek to enumerate all the different possible symbolic expressions which describe how the output will be computed in terms of the inputs. Of course, the symbolic expression defining the output (in terms of the inputs) will be different along different program paths. However, we expect that the number of such symbolic expressions to be substantially lower than the number of program paths. In other words, a large number of paths can be considered "equivalent" since the symbolic expressions describing the output are the same.

To illustrate our observation, let us consider the program in Figure 1. The output variable `out` can be summarized as follows.

- If $x - y > 0$ and $x + y > 10$, then $out == x$
- If $x - y \leq 0$ and $x + y > 10$, then $out == y$
- If $x + y \leq 10$, then $out == 2$

The summary given in the preceding forms a "semantic signature" of the program as far as the output variable `out` is concerned. Note that there are *only three* cases in the semantic signature -

whereas there are *eight paths* in the program. Thus, such a semantic signature can be much more concise than an enumeration of all paths.

In this paper, we develop a method to compute such a semantic signature for a given program. Our semantic signature is computed via dynamic path exploration. While exploring the paths of a program, we establish a natural partitioning of paths *on-the-fly* based on program dependencies - such that only one path in a partition is explored. Thus, for the example program in Figure 1 only three execution traces corresponding to the three cases will be explored. For test-suite construction, we can then construct only three tests corresponding to the three cases in the semantic signature.

How do we partition paths? The answer to this question lies in the computation of the *output* variable. We consider two program paths to be "equivalent" if they have the same relevant slice [9] with respect to the program output. A relevant slice is the transitive closure of dynamic data, control and potential dependencies. Data and control dependencies capture statements which affect the output by getting executed; on the other hand, potential dependencies capture statements which affect the program output by *not* getting executed. In Figure 1, even if line 11 is not executed, the output statement in line 16 is potentially dependent on the branch in line 10. This is to capture the fact that if line 10 is evaluated differently, the assignment in line 11 will be executed leading different values flowing to the output `out`. We base our path partitioning on relevant slices to capture all possible flows into the output variable - whether by the execution of certain statements or their non-execution.

The contributions of this paper can be summarized as follows. We present a mechanism to partition program paths based on the program output. The grouping of paths is done by efficient dynamic path exploration - where paths sharing the same relevant slice naturally get grouped together. We show that our smart path exploration is much more time efficient as opposed to full path exploration via path enumeration. Our efficient path exploration method has immediate benefits in software testing. Since our path exploration naturally groups several paths together - it is much more efficient than the full path exploration (as in Directed Automated Random Testing or DART) as evidenced by experiments. Moreover, since several paths are grouped as "equivalent" in our method (meaning that these paths compute the output similarly), the test-suite generated from our path exploration will also be concise.

Secondly, we show the application of our path partitioning method in reasoning about program versions, in particular, for debugging the root-cause of software regressions. While trying to introduce new features to a program, existing functionality often breaks; this is commonly called as software regression. Given two program versions P, P' and a test t which passes in P while failing in P' — we seek to find a bug report explaining the root cause of the failure of t in P'. In an earlier work [14], we presented the DARWIN approach for root causing software regressions. The DARWIN approach constructs and composes the path conditions of test t in program versions P, P' in trying to come up with a bug report explaining an observed regression. In this work, we show that computing and composing the logical condition over a relevant slice (also called *relevant-slice condition* throughout the paper) produces more pinpointed bug reports in a shorter time — as opposed to computing and composing path conditions. The reason for obtaining shorter bug reports in lesser time comes from the path conditions containing irrelevant information which are filtered out in *relevant-slice conditions*. Hence *relevant-slice conditions* are smaller formulae, which are constructed and solved (via Satisfiability Modulo Theory solvers) more efficiently.

2. OVERVIEW

We begin with a few definitions.

DEFINITION 1 (PATH CONDITION). *Given a program P and a test input t, let π be the execution trace of t in P. The path condition of π, say pc_π is a quantifier free first order logic formula which is satisfied by exactly the set of inputs executing π in program P. Clearly, $t \vDash pc_\pi$.*

The path condition is computed through symbolic execution. During symbolic execution, we interpret each statement and update the symbolic state to represent the effects of the statements (such as assignments) on program variables. At every conditional branch, we compute a branch constraint, which is a formula over the program's input variables which must be satisfied for the branch to be evaluated in the same direction as the concrete execution. The result of symbolic execution is a path condition, which is a conjunction of constraints corresponding to all branches along the path. Any input that satisfies the path condition generated by executing an input t is guaranteed to follow the same path as t. We take the following example to show that the effect of assignments is also considered in path conditions. The path condition for input $\langle x == 0 \rangle$ is $\neg(x - 1 > 0)$, that is, the effect of the assignment in line 4 is considered.

```
1 int x; //input variable
2 int a = 0;
3 scanf("%d",&x);
4 x = x - 1;
5 if(x > 0)
6     a = 1;
7 out = a;
```

Figure 2: Example to show path condition and *relevant-slice condition* computation

We now define slice conditions, which are path conditions computed over slices.

DEFINITION 2 (DYNAMIC SLICE CONDITION). *Given a program P, a test input t and a slicing criteria C — let π be the execution trace of t in P. Let $\pi\,|_C$ denote the projection of π w.r.t. the dynamic slice of C in π. In other words, a statement instance s in π is included in the projection $\pi\,|_C$ if and only if s is in the backward dynamic slice of C on π. The dynamic slice condition of C in π is the path condition computed over the projected trace $\pi\,|_C$.*

Slice conditions are weaker than path conditions, that is, $pc_\pi \Rightarrow dsc_{(\pi,C)}$ where $dsc_{(\pi,C)}$ is the dynamic slice condition of any slicing criteria C in π (see our technical report [13] for a simple proof of this claim). We now refine dynamic slice condition to *relevant-slice condition* - the central concept behind our path partitioning. But first, let us recall the notion of potential dependencies and relevant slices [1, 9].

DEFINITION 3 (POTENTIAL DEPENDENCE [1]). *Given an execution trace π, let s be a statement instance and br be a branch instance that is before s in π. We say that s is potentially dependent on br iff. there exists a variable v used in s such that (i) v is not defined between br and s in trace π but there exists another path σ from br to s along which v is defined, and (ii) evaluating br differently may cause this untraversed path σ to be executed.*

An example of potential for the program in Figure 1 is shown in Figure 3.

We now introduce the notion of a relevant slice, and *relevant-slice condition*, a logical formula computed over a relevant slice.

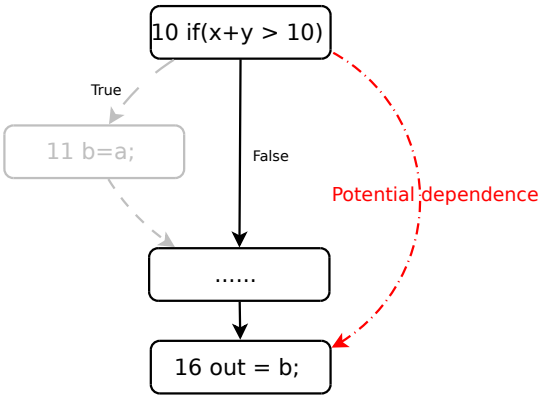


Figure 3: Example of potential dependence. The solid arrows denote the execution path. According to Definition 3, (i) the variable b is not defined between line 10 and line 16 but there exists a path (though line 11) along which b is defined, and (ii) evaluating the branch at line 10 differently may cause the path through line 11 to be executed. Therefore, line 16 is potentially dependent on line 10.

DEFINITION 4 (RELEVANT SLICE). *Given an execution trace π and a slicing criteria C in π, the relevant slice in π w.r.t. C contains a statement instance s in π iff. $C \rightsquigarrow s$ where $\rightsquigarrow$ denotes the transitive closure of dynamic data, control and potential dependence.*

Note that our definition of relevant slice is slightly different from the standard definition of relevant slice [1, 9]. In standard relevant slicing algorithm, if a statement instance A is included only by potential dependence, the statement instances that are only control dependent by A are not included in the relevant slice. We have removed this restriction to simplify the definition of relevant slice, it is simply the transitive closure of three kinds of program dependencies — dynamic data dependencies, dynamic control dependencies and potential dependencies. In the rest of the paper, all appearances of relevant slice and *relevant-slice condition* refer to this simplified definition of relevant slice.

DEFINITION 5 (RELEVANT SLICE CONDITION). *Given an execution trace π and a slicing criteria C in π, the relevant slice condition in π w.r.t. criterion C is the path condition computed over the statement instances of π which are included in the relevant slice of C in π.*

We take the example program in Figure 2 to show that the effect of assignments is also considered in *relevant-slice condition* computation (just as assignments are considered in path condition computation). Let the slicing criteria be the value of `out` in line 7. The relevant slice for input $\langle x == 0\rangle$ is {2,3,4,5,7} and the corresponding *relevant-slice condition* is $\neg(x - 1 > 0)$. That is, the effect of the assignment in line 4 is considered.

We use the simple program in Figure 1 to illustrate the advantage of using *relevant-slice condition* in dynamic path exploration. The slicing criteria is the variable `out` at line 16. Since each statement is executed once, we do not distinguish between different execution instances of the same statement in this example.

We use the executed branch sequence annotated with directions to represent an execution trace. For example, the trace for input $\langle x == 6, y == 2, z == 2\rangle$ of the program in Figure 1 is denoted as $[b1^t, b2^f, b3^t]$. Let us take the input $\langle x == 6, y == 2, z == 2\rangle$ to see the differences between path condition, dynamic slice condition and *relevant-slice condition*. Given the trace $[b1^t, b2^f, b3^t]$ corresponding to input $\langle x == 6, y == 2, z == 2\rangle$, the path condition along this execution is $(x - y > 0) \wedge \neg(x + y > 10) \wedge (z * z > 3)$.

For the execution path of $\langle x == 6, y == 2, z == 2\rangle$, the dynamic backward slice result w.r.t. the slicing criteria at line 16 is {4,16} - it contains no branches. The path condition computed over the statements in the dynamic slice (or the dynamic slice condition) is simply the formula $true$.

Different from dynamic backward slicing, relevant slicing also includes the statement instances that could potentially affect the slicing criteria. For example, if evaluating a branch differently could affect the slicing criteria — such a branch is included in the relevant slice, even though it is not contained in the dynamic backward slice. In the example program, the branch at line 10 can potentially affect the value of out in the slicing criteria. This is because if the branch in line 10 is evaluated differently (to true), the variable b is re-defined (in line 11) which affects the output variable out. Hence the relevant slice contains line 10. The entire relevant slice is {4,5,10,16}, and the *relevant-slice condition* on it is $\neg(x + y > 10)$. Any input t satisfying the *relevant-slice condition* $\neg(x + y > 10)$ has the same symbolic expression for the output `out`, which in this case turns out be the constant value 2.

As mentioned earlier, program paths can be partitioned based on the input-output relation. *Relevant-slice condition* perfectly serves this purpose. If two paths have the same relevant slice with output being the slicing criteria, then they have the same input-output relation. The path partitions of the program in Figure 1 are shown in Figure 4. The grey nodes in Figure 4 are the statements that are contained in the relevant slice w.r.t. to the unique slicing criteria at line 16 in Figure 1. As we can see from Figure 4, based on the relevant slice, we can group the eight program paths into three path partitions.

Just like the DART approach [8] uses path conditions to dynamically explore paths in a program, *relevant-slice condition* can be used to explore the possible symbolic expressions that the program output can be assigned to. How would such an exploration proceed? Suppose we simply use *relevant-slice condition* to replace path condition in DART's path exploration. Given a *relevant-slice condition* $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$ — we construct k sub-formulae of the form of $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$, where $1 \leq i \leq k$. The path exploration is done by solving these formulae to get new inputs and iteratively applying this process to the new inputs. Note that each sub-formula shares a common prefix with the *relevant-slice condition*. Now, we examine the effectiveness of this simple solution on the program in Figure 1. Depth-first exploration strategy is used, and path exploration terminates when no new sub-formulae are generated. Let the initial input be $\langle x == 6, y == 2, z == 2\rangle$, the path for this input is $[b1^t, b2^f, b3^t]$. The entire path exploration process is shown in Table 1. The "from" column of Table 1 can be understood as follows. If the "from" column contains $\alpha.\beta$, it means that the current input is generated by negating the βth branch constraint of the *relevant-slice condition* in the αth row.

Recall from Section 1 that we expect the following three symbolic expressions for out to be explored.

- $x - y > 0 \wedge x + y > 10$: `out` == x
- $\neg(x - y > 0) \wedge x + y > 10$: `out` == y
- $\neg(x + y > 10)$: `out` == 2

As we can see from Table 1, no path having *relevant-slice condition* $\neg(x - y > 0) \wedge (x + y > 10)$ is explored. Therefore, this feasible *relevant-slice condition* is missed by the exploration process. In addition, the *relevant-slice condition* $\neg(x + y > 10)$ is explored

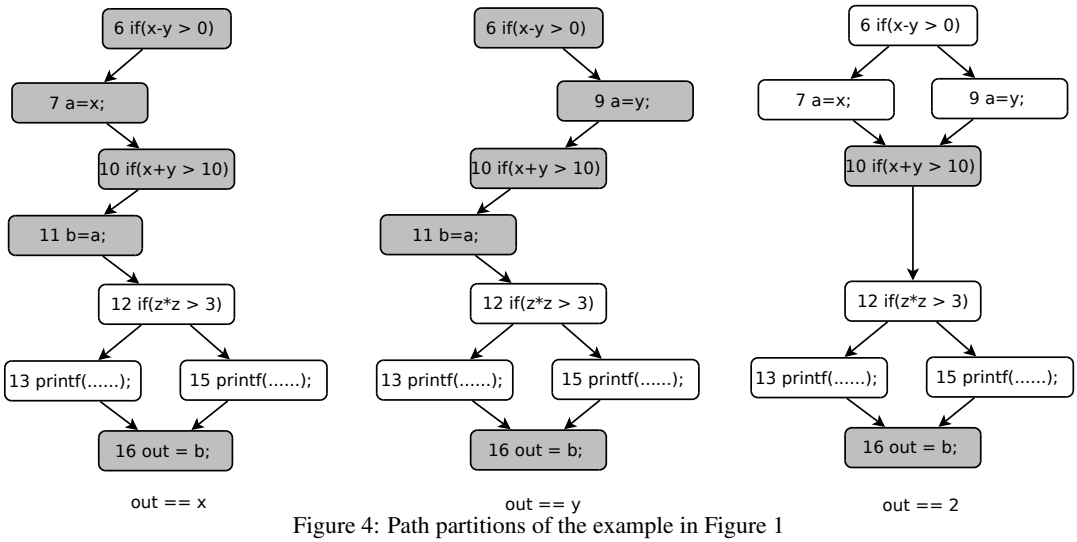


Figure 4: Path partitions of the example in Figure 1

several times. Thus, we cannot simply replace path condition with *relevant-slice condition* in DART's path exploration.

Let us examine closely what went wrong in the path exploration of Table 1. In particular, the input in the third row is generated by negating the second branch condition of the *relevant-slice condition* in second row in Table 1. That is, when we solve $(x - y > 0) \wedge \neg(x + y > 10)$, we get an input $\langle x == 6, y == 2, z == 2\rangle$ whose *relevant-slice condition* is $\neg(x + y > 10)$. The branch condition $(x - y > 0)$ disappears in the new *relevant-slice condition* because the corresponding branch is not contained in the relevant slice anymore. In contrast, DART follows certain path-prefixing properties — if $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \psi_i$ is the prefix of a path condition (for some program input), the path condition of any input satisfying $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ will have $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ as a prefix. Such a property does not hold for *relevant-slice condition*. Hence, simply replacing path condition with *relevant-slice condition* in DART not only causes redundant path exploration but also makes the exploration incomplete (in terms of possible symbolic expressions that the output variable may assume).

We have developed a path exploration method which avoids the aforementioned problems. While exploring (groups of) paths based on *relevant-slice condition*, our method re-orders the constraints in the *relevant-slice condition*. The path exploration is based on reordered *relevant-slice condition*. A reordered *relevant-slice condition* satisfies the following property (which also holds for path conditions): if $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \psi_i$ is a prefix of a reordered *relevant-slice condition*, the reordered *relevant-slice condition* of any input satisfying $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ has $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ as a prefix.

3. OUR APPROACH

In this section, we give our path exploration algorithm based on *relevant-slice condition*. We then give theorems on the completeness of our path exploration algorithm. Throughout the paper, we assume that the slicing criteria is in a basic block that post-dominates the entry of the program.

First we introduce the following notations.

Notations. We use C to denote the unique slicing criteria. When used in a dynamic context, C refers to the last executed instance of the slicing criteria. Given a test case t, we use $\pi(t)$ to denote the execution path of t. We use $rs(sc, \pi)$ to denote the relevant slice on path π w.r.t. slicing criteria sc. We use $rsc(sc, \pi)$ to denote the relevant slice condition on path π w.r.t. slicing criteria sc. We use $reordered\_rsc(sc, \pi)$ to denote the reordered sequence of $rsc(sc, \pi)$. We use $br(\psi)$ to denote the branch instance of a branch condition ψ.

3.1 Path exploration algorithm

We now present our path exploration method which operates on a given program P. All relevant slices and *relevant-slice conditions* are calculated on the same program P with respect to a slicing criteria C (which refers to the program output).

We group paths based on *relevant-slice condition*. As explained in the last section, a DART-like search based on *relevant-slice conditions* is incomplete, that is, not all possible symbolic expressions that the output may assume will be covered. For this reason, we reorder the *relevant-slice conditions*.

Our path exploration algorithm is shown in Algorithm 1. The core of the algorithm is the *reorder* procedure, which reorders the *relevant-slice conditions*. When we compute the *relevant-slice condition*, we get a sequence of branch conditions – ordered according to the sequence in which they are traversed. We use the $reorder$ function to reorder the branch conditions, after which the path exploration will be performed based on the reordered sequence of branch conditions.

The $reorder$ procedure is given in Algorithm 1. The reordering works in a quick-sort-like fashion. In each call to *reorder*, we split the to-be-reordered sequence into two sub-sequences. Suppose the last branch condition in the sequence is from branch instance b_k. Then b_k is used as the "pivot" in the splitting process. If a branch instance b is in the backward relevant slice of b_k, then the branch condition of b is placed before the branch condition of b_k. Otherwise, the branch condition of b is placed after the branch condition of b_k. Then we recursively call the *reorder* procedure to reorder the two sub-sequences.

We show the *reorder* procedure in action in Figure 5. Note that our reordering is done on branch conditions in a *relevant-slice condition*. Since there is a unique branch condition for each branch instance in the execution trace, the example in Figure 5 is on branch instances for simplicity. On the left of Figure 5, the dependencies among all the branch instances are provided. If there is an arrow

No.	from	input	path	RSC
1		$\langle 6,2,2\rangle$	$[b1^t, b2^f, b3^t]$	$\neg(x+y>10)$
2	1.1	$\langle 6,5,2\rangle$	$[b1^t, b2^t, b3^t]$	$(x-y>0)\wedge(x+y>10)$
3	2.2	$\langle 6,2,2\rangle$	$[b1^t, b2^f, b3^t]$	$\neg(x+y>10)$
4	2.1	$\langle 2,6,2\rangle$	$[b1^f, b2^f, b3^t]$	$\neg(x+y>10)$

Table 1: Path exploration based on *relevant-slice conditions* for example in Figure 1

No.	from	input	path	RSC	reordered RSC
1		$\langle 6,2,2\rangle$	$[b1^t, b2^f, b3^t]$	$\neg(x+y>10)$	$\neg(x+y>10)$
2	1.1	$\langle 6,5,2\rangle$	$[b1^t, b2^t, b3^t]$	$(x-y>0)\wedge(x+y>10)$	$(x+y>10)\wedge(x-y>0)$
3	2.2	$\langle 5,6,2\rangle$	$[b1^f, b2^t, b3^t]$	$\neg(x-y>0)\wedge(x+y>10)$	$(x+y>10)\wedge\neg(x-y>0)$

Table 2: Path exploration with reordered *relevant-slice conditions* for example in Figure 1

Algorithm 1 Path exploration using *relevant-slice condition*

Input:
P : The program to test
t : An initial test case for P
C : A slicing criterion
Output:
T: A test-suite for P

$Stack$ = $null$ // The stack of partial rsc to be explored
$Execute(t, 0)$
while $Stack$ is not empty **do**
  let $\langle f, j\rangle$ = $pop(Stack)$
  if f is satisfiable **then**
    let μ be one input that satisfies f
    put μ into T
    $Execute(\mu, j)$
  end if
end while
return T

procedure $Execute(t, n)$
  execute t in P and compute *relevant-slice condition* rsc w.r.t. C
  let rsc = $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{m-1} \wedge \psi_m$
  let rsc' = $reorder(rsc)$
  suppose rsc' = $\psi'_1 \wedge \psi'_2 \wedge \ldots \wedge \psi'_{m-1} \wedge \psi'_m$
  for all i from n+1 to m **do**
    let h = $(\psi'_1 \wedge \psi'_2 \wedge \ldots \wedge \psi'_{i-1} \wedge \neg\psi'_i)$
    push $\langle h, i\rangle$ into $Stack$
  end for
  return
end procedure

procedure $reorder(seq)$
  if $|seq|$ == 0 **then**
    return seq
  end if
  let seq be $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$
  seq_1 = $true$, seq_2 = $true$
  for all i from 1 to k-1 **do**
    if $br(\psi_i)$ is in relevant slice of $br(\psi_k)$ **then**
      seq_1 = $seq_1 \wedge \psi_i$
    else
      seq_2 = $seq_2 \wedge \psi_i$
    end if
  end for
  return $reorder(seq_1) \wedge \psi_k \wedge reorder(seq_2)$
end procedure

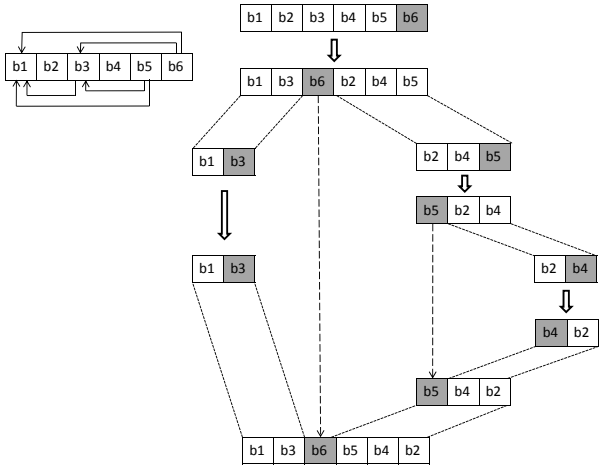


Figure 5: *Reorder* algorithm in action

from bj to bi, then bi is in the relevant slice of bj. The "pivot" in each reorder step is marked in dark; the other branches are reordered w.r.t to the "pivot". For example, initially `b6` is the pivot and we reorder `b1`, ... `b5` depending on whether they are in the relevant slice of `b6`.

In Algorithm 1, we use a stack to maintain the to-be-explored partial *relevant-slice conditions*. The main algorithm keeps on processing the formulae in the stack when it is not empty. In each iteration, the algorithm pops out one partial *relevant-slice condition* from the stack, and checks whether it is satisfiable or not. If it is satisfiable, we get a new input μ by solving the formula. The new input μ could lead to some unexplored *relevant-slice condition*. The *relevant-slice condition* for the execution trace of input μ is then explored, as shown by the procedure *Execute* in Algorithm 1. Given the execution trace of μ, the *relevant-slice condition* over this trace w.r.t. the slicing criteria C is first computed. The *relevant-slice condition* is reordered using the *reorder* procedure, and the to-be-explored partial *relevant-slice conditions* are pushed into the stack.

The second parameter of $Execute$ is used to avoid redundancy in path search. When $Execute$ is called with parameters t and n, let the reordered *relevant-slice condition* $reordered\_rsc(C, \pi(t))$ be $\psi'_1 \wedge \psi'_2 \wedge \ldots \wedge \psi'_{m-1} \wedge \psi'_m$. For any partial *relevant-slice condition* φ_i = $\psi'_1 \wedge \psi'_2 \wedge \ldots \wedge \psi'_{i-1} \wedge \neg\psi'_i$, $1 \leq i \leq n \leq m$, we know that φ_i has been pushed into the stack a-priori. So the for-loop in the *Execute* procedure starts from $n + 1$ to avoid these explored partial *relevant-slice conditions*.

The path exploration of Algorithm 1 when employed on the program in Figure 1 leads to the *relevant-slice conditions* shown in Table 2. If the "from" column of Table 2 contains $\alpha.\beta$, it means that the current input is generated by negating the βth branch constraint of the reordered *relevant-slice condition* in the αth row. The path exploration based the reordered *relevant-slice condition* explores all possible *relevant-slice conditions* of the program.

3.2 Theorems

Due to space limit, the proofs of the theorems are omitted in this paper. The readers can refer to our technical report [13] for the proofs.

Assumptions. We also assume that the SMT solver used to solve *relevant-slice conditions* is sound and complete. As mentioned earlier, we assume that the slicing criteria is in a basic block that post-dominates the entry of the program — this is the location of the program output. If the program contains *multiple outputs*, the slicing criteria can simply be a set of of primitive criteria of the form

$$\langle output\ variable, output\ location \rangle$$

Note that slicing can be performed on such a criteria (which is a set) without any change to our method.

THEOREM 3.1. *If the relevant-slice conditions of two paths π_1 and π_2 w.r.t. C are the same, then the variables used in the slicing criteria C have the same symbolic values in π_1 and π_2.*

In Theorem 3.1, we show that the *relevant-slice condition* determines the symbolic values of variables used in the slicing criteria — if the *relevant-slice conditions* of two paths are same, the variables in the slicing criteria have the same symbolic values. Symbolic value can be computed by dynamic symbolic execution. Each symbolic value is an expression in terms of the program inputs. Let s be a statement instance in the path of input t, and v be a variable used in s. The symbolic value of v in s is an expression in terms of input variables. If the symbolic value of v is concretized with t, it must be the same as the value of v in s when the program is run concretely with input t.

THEOREM 3.2. *Given a program P and an execution trace $\pi(t)$ for input t in P, Algorithm 1 must explore an execution trace $\pi(t')$ for some input t' such that $\pi(t)$ and $\pi(t')$ share the same relevant-slice condition (irrespective of the initial test input with which Algorithm 1 is started) — provided the total number of relevant-slice conditions in P is bounded.*

In Theorem 3.2, given any feasible path π, we show that our path exploration algorithm is guaranteed to explore a path π' that shares the same *relevant-slice condition* with π. This establishes the completeness of our path search.

4. IMPLEMENTATION

In this section, we discuss our combined infra-structure for symbolic execution and dependency analysis of Java programs.

Our implementation is based on JSlice [17][1]. JSlice is an open-source dynamic slicing tool on Java bytecode. We have extended JSlice to compute *relevant-slice conditions*. The architecture of our extended JSlice is shown in Figure 6.

JSlice keeps the collected trace in a compressed form to achieve scalability. The compression is online — as the trace is generated it is simultaneously compressed and then slicing is done on the compressed trace. The slicing algorithm works directly on the compressed trace. We design our extension of JSlice to retain this feature (of analyzing compressed traces without decompression).

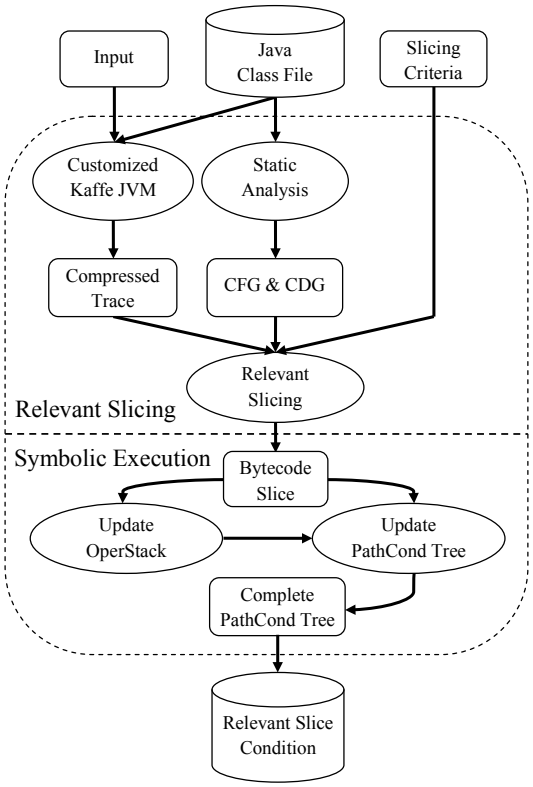


Figure 6: Architecture of *relevant-slice condition* computation

In Figure 6, relevant slicing and symbolic execution are separated for ease of understanding. However, we do not need the entire relevant slicing result to start computing *relevant-slice condition* in the implementation. The process of constructing the *relevant-slice condition* is done along with the backward relevant slicing to achieve efficiency. Since the relevant slicing process is backward, we also compute the relevant slice condition via a backward symbolic execution which starts from the slicing criteria and stops at the beginning of the trace.

For backward symbolic execution, we keep a set of symbolic values which need to be explained. The symbolic value of a variable v is explained by either an assignment to v or by program input to v. Let us take the sample program in Figure 1 to show our backward symbolic execution on a relevant slice. Note that although we show this example at the source code level, our implementation is at the Java bytecode level. Suppose the input is $\langle x == 6, y == 5, z == 2 \rangle$. The relevant slice for the execution trace of this input is $[5, 6, 7, 10, 11, 16]$. Backward symbolic execution along this trace is shown in Table 3.

To construct the *relevant-slice conditions*, we need to precisely represent the semantics of each bytecode type in the generated formula. There are more than 200 different bytecode types in the Java Virtual Machine instruction set, and all of them are handled in our implementation. Our implementation also handles native method calls (more details in the next paragraph). However, due to the JSlice version that our implementation is based on, currently we cannot handle programs with multi-threading and reflection.

In the original implementation of JSlice, the concrete operand values of most executed instructions are not stored in the compressed trace as they are not needed in the slicing process. How-

[1] http://jslice.sourceforge.net/

Relevant slice	Symbolic values	To be explained variables	Relevant slice condition
`16 out = b;`	$\{ out \rightarrow b \}$	$\{ b \}$	true
`11 b = a;`	$\{out \rightarrow a, b \rightarrow a \}$	$\{ a \}$	true
`10 if(x+y > 10)`	$\{out \rightarrow a, b \rightarrow a \}$	$\{ x, y \}$	$x + y > 10$
`7 a = x;`	$\{out \rightarrow x, b \rightarrow x, a \rightarrow x \}$	$\{ x, y \}$	$x + y > 10$
`6 if(x-y >0)`	$\{out \rightarrow x, b \rightarrow x, a \rightarrow x \}$	$\{ x, y \}$	$x - y > 0 \wedge x + y > 10$
`5 scanf("%d %d %d",&x,&y,&z);`	$\{out \rightarrow x, b \rightarrow x, a \rightarrow x \}$	$\{ x, y \}$	$x - y > 0 \wedge x + y > 10$

Table 3: Backward symbolic execution example

ever, these values are needed when the semantics of some operations cannot be precisely modelled. In such cases, we have to under-approximate the generated path condition/*relevant-slice condition* by concretizing certain symbolic values in the *relevant-slice condition*. For example, Java allows a program to use libraries written in other languages through native method call. Since the native calls cannot be traced in Java Virtual Machine, the symbolic return values from native calls cannot be precisely modelled. In this case, we simply concretize the symbolic return value from a native call using the concrete return value of the native call (therefore, the concrete return value of native calls are traced in our implementation).

In our implementation, we use the concept of execution index [20] to uniquely identify a statement instance across different paths. Two statement instances in different paths are the same iff. they have exactly the same "execution index". In its simplest form, we can use the path from root to a statement instance s in the Dynamic Control Dependence Graph of path π as the execution index of statement instance s in path π.

As mentioned in Section 3, we need to reorder the branch conditions in a *relevant-slice condition* in our path exploration process. Let $rs(C, \pi)$ be the relevant slice on trace π w.r.t. the slicing criteria C. Let $rsc(C, \pi)$ be the *relevant-slice condition* computed on $rs(C, \pi)$. To reorder the branch conditions in $rsc(C, \pi)$ using the *reorder* procedure shown in Algorithm 1, we need to compute a relevant slice using each branch instance in $rs(C, \pi)$ as the slicing criteria. Suppose there are m branch instances in $rs(C, \pi)$, our implementation traverses the trace π for m times to compute the m relevant slices. In future, we plan to speed up this process, by computing all m relevant slices at the same time of computing $rs(C, \pi)$. We also observe that there are a lot similarities among the slices w.r.t. different branch instances (used as slicing criteria) in the same trace. For example, if a branch instance b_i is in the relevant slice of branch instance b_j, then the relevant slice w.r.t. b_i is a subset of the relevant slice w.r.t. b_j. In future, we could exploit the similarities among these slices to further reduce the cost of our *reorder* procedure.

Our execution engine is a combined infra-structure for dynamic dependency analysis and dynamic symbolic execution. Thus, apart from computing *relevant-slice conditions*, we can simply disable the dependency analysis in our engine to compute path conditions. The path conditions and *relevant-slice conditions* generated from our tool are in the format of SMT2[2], which can be solved by various Satisfiability Modulo Theory or SMT solvers. In our implementation, we choose Z3 [4][3] as the SMT solver for our tool.

5. EXPERIMENTS

In the following, we first compare our *relevant-slice condition* based path exploration method with Directed Automated Random Testing (DART). We then present an application of *relevant-slice conditions* in the debugging of evolving programs.

5.1 Path exploration

We compare our path exploration algorithm with DART. The subject programs shown in Table 4 are from SIR [5] repository. The lines of code (LOC) in each program are also shown.

Recall that our path search is complete as supported by Theorem 3.2. However, the completeness is difficult to achieve in practice for several reasons. Two of the main reasons are the limited power of current SMT solvers and imprecise modelling of program semantics. Because of these reasons, our technique may miss a certain *relevant-slice condition* rsc_i when DART can explore a path whose *relevant-slice condition* is rsc_i. The example below explains how imprecise modelling of array can causes our implementation to be not as complete as DART in terms of *relevant-slice condition* coverage.

```
1  int x, y; //input
2  int out; //output
3  int a[2] = {0,1};
4  scanf("%d %d", &x, &y);
5  if(x > 0)
6      printf("x is greater than zero\n");
7  if(a[x]>0){
8      if(y > 0)
9          out = 1;
10     else
11         out = -1;
12 }else{
13     out = 0;
14 }
15 printf("%d\n", out);//slicing criteria
```

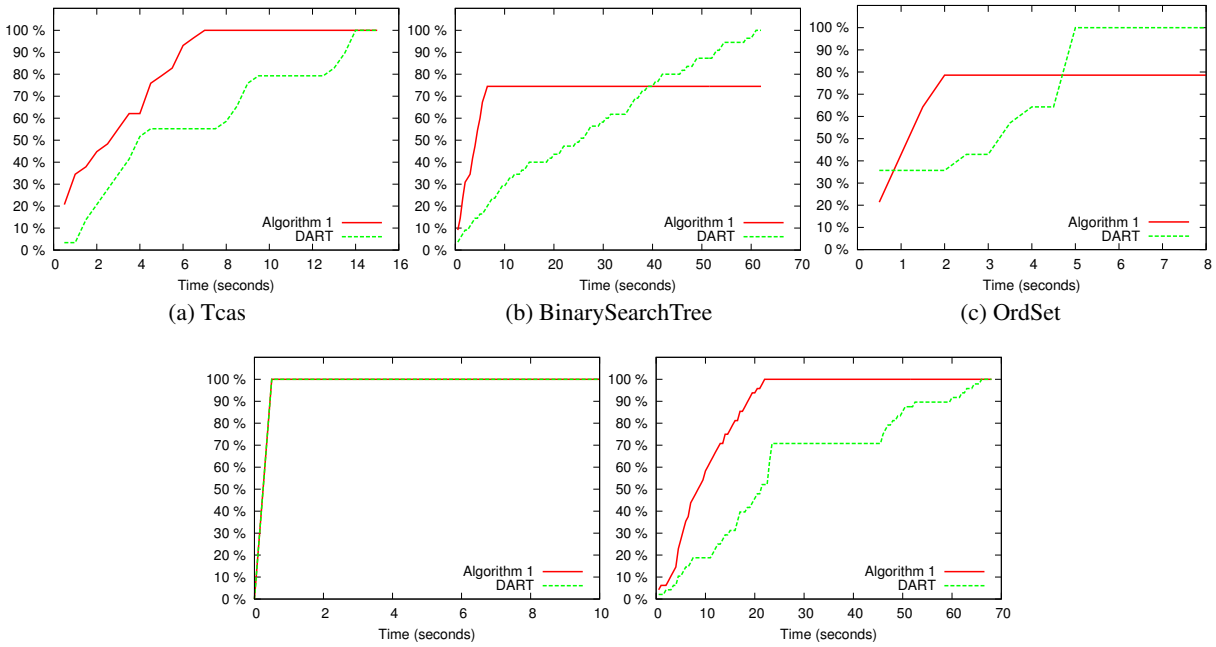
In our current implementation, we concretize symbolic array index using the value observed at execution time. Suppose the initial input for our method and DART are both $\langle x == 0, y == 0 \rangle$. Due to the concretization of symbolic array index, the branch at line 7 cannot contribute a branch condition to either path condition or *relevant-slice condition*. The path condition and *relevant-slice condition* for $\langle x == 0, y == 0 \rangle$ are $\neg(x > 0)$ and $true$ respectively. Since the *relevant-slice condition* for the initial input $\langle x == 0, y == 0 \rangle$ is $true$ (containing no branch condition), our technique terminates. However, some *relevant-slice conditions* are missed by our technique. In particular, the *relevant-slice conditions* of paths that evaluate branch at line 7 to false are missed. In contrast, DART could explore all feasible paths (hence all *relevant-slice conditions*) of the above program. Although the branch at line 5 cannot directly affect the computation of *out*, it can help DART to negate the branch at line 7 due to the correlation between the two branches. If arrays are modelled precisely, this problem will disappear.

[2] http://combination.cs.uiowa.edu/smtlib/

[3] http://research.microsoft.com/en-us/um/redmond/projects/z3/

Subject prog.	Size (LOC)	Completeness	Time		#Testcases		Avg. formula size		#Solver calls	
			RSC	DART	RSC	DART	RSC	DART	RSC	DART
Tcas	113	100%	6.3s	13.1s	29	88	5744	64810	412	939
BinarySearchTree	175	75%	6.1s	58.6s	64	453	3836	49266	163	3188
OrdSet	211	79%	2.1s	7.4s	12	59	6444	55461	96	293
Schedule	257	100%	0.3s	15.4s	3	75	1808	13728	13	932
DisjointSet	102	100%	20.8s	64.8s	69	278	7643	170533	1192	3855

Table 4: Experiments in full program exploration



(a) Tcas (b) BinarySearchTree (c) OrdSet

(d) Schedule (e) DisjointSet

Figure 7: Relevant-slice condition coverage comparison

The "Completeness" column in Table 4 measures how much incompleteness in *relevant-slice condition* coverage is introduced by the imprecise modelling of program semantics in our implementation. The numbers in the "Completeness" column are computed as follows. Let the program being explored be P. We employ DART on P to explore program paths and construct a test-suite $T_{DART-ALL}$ which contains the set of all paths in P covered by DART. For each test case t in $T_{DART-ALL}$, we compute the *relevant-slice condition* on the execution trace of t and put this *relevant-slice condition* into a set $S_{DART-ALL}$. Similarly, we generate a test-suite T_{RSC} for program P using our path exploration method. For each test case t in T_{RSC}, we compute the *relevant-slice condition* on the execution trace of t and put this *relevant-slice condition* into a set S_{RSC}. Then the "Completeness" column in Table 4 is $\frac{|S_{RSC}|}{|S_{DART-ALL}|}$. As shown in Table 4, our method cannot always achieve 100 percent relevant slice coverage as compared to DART due to the imprecise modelling of program semantics in our implementation. Note that this does affect the validity of the completeness claim in Theorem 3.2, the incompleteness is only in the implementation.

In columns 4-11 of Table 4, we compare the time, number of generated test cases, formula size and number of solver calls between our method and DART. The formula size is measured by the number of bytes in the SMT2 formula file. For getting these numbers, both our method (RSC) and the DART method are *run to completion*, and the running time is recorded. Note that the time reported in Table 4 includes the time taken in every steps of our method and DART. For example, the time taken by our method includes the time for program execution, relevant slicing, *relevant-slice condition* computation, branch condition reordering, formula solving, etc. As shown in Table 4, our technique takes much less time than DART. The efficiency comes from several sources. First, since we use *relevant-slice condition* instead of path condition, the formula size of our approach is much smaller than that of DART. This reduces the time taken by the solver. Second, the number of different *relevant-slice conditions* is considerably smaller than the number of path conditions. This reduces both the number of executions and the number of solver calls.

Figure 7 compares the *relevant-slice condition* coverage of our Algorithm 1 with Directed Automated Random Testing (DART) *under the same time limit*. Note that DART intends to achieve path coverage. However, as we have observed - several paths may have the same input-output relationship, and testing is always done by checking outputs. We check the number of *relevant-slice conditions* that are covered by the paths explored in DART search. As shown in Figure 7, our technique gets higher *relevant-slice condition* coverage then DART when the given time is short.

5.2 Debugging of evolving programs

The obvious application of *relevant-slice conditions* is in software testing - it groups program paths and can be used to efficiently

Subject prog.	Stable version	Buggy version	Diff	Time		Debugging results	
				PC	RSC	PC	RSC
JLex	1.2.1 (7290 LOC)	1.1.1 (6984 LOC)	518 LOC	543 min	15 min	50 LOC	3 LOC
JTopas	0.8 (4514 LOC)	0.7 (5754 LOC)	2489 LOC	81 min	5 min	4 LOC	4 LOC

Table 5: DARWIN debugging results (LOC stands for Lines of Code)

generate a concise test-suite. We now show another application of *relevant-slice conditions* namely in the debugging of evolving programs. As a program evolves, functionality which worked earlier breaks. This is commonly known as software regressions. For any large scale software development, debugging the root-case of regressions is an extremely time consuming activity.

We applied our *relevant-slice conditions* on the DARWIN method for debugging evolving programs [14]. Given two program versions P and P', and a test case t which passes in P but fails in P', the work in [14] tries to find the root cause of the failure of t in P'. The debugging proceeds by computing and composing the path conditions of t in P and P', as follows.

First, the path conditions f and f' of t in P and P' are computed. We then compute the formula $f \wedge \neg f'$ as follows. Suppose f' is $f' = (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_m)$ where ψ_i are primitive constraints. The following m formulae $\{\varphi_i \mid 0 \leq i < m\}$ are then solved where $\varphi_i \stackrel{def}{=} f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$. We invoke a Satisfiability Modulo Theory or SMT solver to solve the m formulae $\{\varphi_i \mid 0 \leq i < m\}$. Finally, for every φ_i which is satisfiable, we can find a single line in the source code which is a potential error root cause — the branch corresponding to ψ_{i+1} (which is negated in φ_i).

We observe that the path conditions f and f' in the above method can be replaced by *relevant-slice conditions*. Path conditions are calculated by a forward computation along an execution trace. Thus, a path condition is not "goal-directed" — it contains the constraints of branches which are not "related" to the observable error. In particular, a path condition will typically contain constraints for branches which are not in the dynamic or relevant slice of the observable error. Consider the following example program

```
1  ... // input inp1, inp2
2  if (inp1 > 0)
3    x = inp1 + 1;
4  else
5    x = inp1 - 1;
6  if (inp2 > 0)
7    y = inp2 + 1
8  else
9    y = inp2 - 1;
10 ... // output x, y
```

Suppose the observed value of `x` is unexpected for `inp1 == inp2 == 0` because of a "bug" in line 2 (say, the condition should be `inp1 >= 0`). The path condition is $\neg(inp1 > 0) \wedge \neg(inp2 > 0)$. Clearly, the constraint $\neg(inp2 > 0)$ corresponding the the branch in line 6 is unrelated to the observable error (unexpected value of `x`). Indeed, line 6 is not in the dynamic slice or relevant slice of the slicing criterion corresponding to the output value of `x` in line 10.

Thus, due to the inherent parallelism in sequential programs, path conditions contain constraints for branches which are not in the slice of the observed error. Composing these path conditions for debugging then allows for such "unrelated" branches to be incorporated into the bug report (which is output by the debugging method). Indeed including these "unrelated" branch constraints increases the burden on the SMT solvers invoked by the DARWIN method, both in terms of the size of the formulae and the number of the formulae to solve. In addition, these "unrelated" branch constraints also introduce some false positives into the bug report produced by the DARWIN method.

Replacing path condition with *relevant-slice condition* in the DARWIN method resolves these issues. Thus, given a test case t that passes in the old version program P but fails in the new version program P' — we now compute g and g', the *relevant-slice condition* of t in P and P' respectively. We then solve $g \wedge \neg g'$ in a manner similar to the solving of $f \wedge \neg f'$ in DARWIN (where f, f' were the path conditions of t in programs P, P').

We compare the debugging result of DARWIN using *relevant-slice conditions* with the original DARWIN method (which uses path conditions) in Table 5.

Both methods are *fully* automated. We did not use the same SIR programs as used in Section 5.1 because debugging regression errors for SIR programs are usually trivial. This is because the difference between two SIR program versions is usually small. The first subject program being used is `JLex`[4]. `JLex` is a lexical analyzer generator written in Java. We use version 1.2.1 of `JLex` as the stable version, and version 1.1.1 as the buggy version. There are 6984 and 7290 lines of code in version 1.1.1 and version 1.2.2 respectively. The changes across version 1.1.1 and version 1.2.1 consist of 518 lines of code. In particular, the version 1.1.1 of `JLex` cannot recognize '\r' as the newline symbol, while in version 1.2.1 this bug is fixed. We use an input file manifesting this bug.

The experimental results from DARWIN using *relevant-slice conditions* vs. the original DARWIN method appears in Table 5. The original DARWIN method, which uses path conditions, takes 543 minutes (or 9 hours) to perform the debugging. The result of DARWIN is a bug report containing 50 lines of code, which are highlighted to the programmer as potential root-causes of the observable error. In contrast, DARWIN using *relevant-slice condition* takes only 15 minutes. The result is a bug report containing only 3 lines of code — potential root causes of the observed error. Indeed, the actual error root-cause lies in one of these three lines of code. Thus, by using *relevant-slice conditions* inside our DARWIN debugging method - we could avoid 47 false positives among the potential error causes which are reported to the programmer. Moreover, there is a huge savings in the debugging time (15 minutes vs 9 hours) which comes from the *relevant-slice conditions* being much smaller than path conditions.

We also conducted experiments using `JTopas` [5]as the subject program. `JTopas` is a Java library for parsing arbitrary text data. We use version 0.8 of `JTopas` as the stable version, and version 0.7 as the buggy version. There are 5754 and 4514 lines of code in version 0.7 and version 0.8 respectively. `JTopas` allows users to customize whitespace characters (i.e. characters that are considered as whitespace characters) by using function *setWhitespaces*. `JTopas` also uses a boolean field *_defaultWhitespaces* to control whether the default whitespace characters are used or the user-customized whitespace characters are used. To use the customized whitespace characters, *_defaultWhitespaces* has to be set to *false*. Unfortunately, the buggy `JTopas-0.7` does not reset the member *_defaultWhitespaces* leading to the default whitespace

[4] `http://www.cs.princeton.edu/~appel/modern/java/JLex/`

[5] `http://jtopas.sourceforge.net/jtopas/index.html`

characters still being used instead of the customized ones although the user has specified the custom whitespace characters. In our experiment, we customize whitespace characters to {' ', '\r','\t'} ({' ', '\n', '\r','\t'} by default) and use an input file manifesting the aforementioned bug. The debugging results of DARWIN using path condition and DARWIN using *relevant-slice condition* are shown in Table 5. The results from the original DARWIN method (using path condition) and DARWIN using *relevant-slice condition* are both four lines of code. They both contain the location where '\n' is treated differently between the two versions. The pinpointed location shows that the stable version does not consider '\n' as a whitespace. In contrast, the buggy version still treats '\n' as a whitespace because *_defaultWhitespaces* is *true* (even though whitespace characters have already been customized). From this clue, the programmer could easily infer that the member *_defaultWhitespaces* was not assigned to the correct value. Although using *relevant-slice condition* does not eliminate any false positives in the debugging result, it does reduce the time taken by DARWIN from 81 minutes to 5 minutes.

6. THREATS TO VALIDITY

Our path exploration does not try to cover all paths. Instead, we try to group paths based on symbolic outputs. This is done with the goal of test-suite construction, where testing will expose possible failures in the program. However, failure of a test case does not only come from unexpected outputs - it can also come from program crashes. Thus, for the paths which we do not explore if they contain program crashes - these will not be exposed by the test-suite computed by our technique. Realistically, our test-suite construction could be supplemented by techniques to statically detect possible program crashes, such as memory error detection [19].

Due to the conservative nature of static analysis used in computing relevant slice, our technique may under-approximate sometimes. In that case, we may explore more than one paths that have the same *relevant-slice condition*. Consider the following program

```
101 if(x > 0){
102     p.num = 0;
103 }
104 out = q.num;
```

Suppose `p` and `q` never alias to each other. If the static analysis cannot determine the non-alias between `p` and `q`, line 104 is potential dependent on line 101 when the branch at line 101 is evaluated to false. Therefore, the branch at line 101 is included in relevant slice and our technique will try to explore both directions of the branch at line 101, which is unnecessary. Note that this under-approximation of *relevant-slice condition* only causes duplicated exploration of some *relevant-slice conditions*, it does not affect the completeness claim of our technique.

Although our technique considerably improves the efficiency of the path exploration, the path explosion problem still exist. In the worst case, the number of *relevant-slice conditions* grows exponentially with the size of the program.

7. RELATED WORK

The technique proposed in this paper is based on dynamic path exploration [8, 16] and relevant slicing [1, 9, 17]. Our technique improves existing dynamic path exploration techniques by grouping several paths together using *relevant-slice condition*. Existing dynamic path exploration tries to achieve path coverage. In contrast, our technique only selects one path from each *relevant-slice condition* to explore.

There are several works which focus on improving the efficiency of dynamic path exploration. In [6], function summaries are generated and exploited. In [7], the grammar of the input is used to avoid generating large percentage of invalid inputs. Our approach is orthogonal to these approaches, therefore, our approach can be combined together with any of these approaches to further improve the efficiency of the path search.

In [15], a program is statically decomposed into several path families, where each path family contains several paths that share similar behavior. Instead of analyzing each path individually, a program can be analyzed at the granularity of path family. The authors of [15] also compute a "path family condition" for each path family, which could characterize that path family. The path partition based on *relevant-slice condition* is different from the notion of "path family" in [15] in the sense that "path family" is more general. For example, all program paths in Figure 1 can be grouped into one path family, but they are grouped into three partitions by our technique. At the same time, "path family condition" does not the same input-output relationship, whereas *relevant-slice condition* does. The main difference between our work and [15] lies in the static vs. dynamic nature of the two techniques. The work in [15] statically computes their path family conditions, while we dynamically explore the *relevant-slice conditions*. Because of the dynamic nature of our method, we can under-approximate the *relevant-slice conditions* , while [15] over-approximates their "path family conditions" if needed. Clearly, the dynamic nature of our method makes it more suitable for test generation. Note that the effect of program statements written in real-life programming languages are hard to precisely model as symbolic formulae. In such a situation, under-approximation is a practical simplification, since it amounts to concretizing parts of the formula.

Other researches have also used the notion of "path equivalence" to alleviate the path explosion problem. However, what paths are considered equivalent are different between our work and earlier works. The difference in the definition of path equivalence originates from the different goals of our work and earlier works. In [2], the goal is to explore all possible program states. Based on this goal, two paths are equivalent if the symbolic states of all live variables are the same. In contrast, we only consider the variables that can affect the output – two paths are equivalent if they have the same symbolic expression for the output. In [11], the goal is to reach some critical locations in a program. Therefore, two paths are equivalent if they cannot reach any critical locations for the same reason (blocked by the same condition).

Apart from the application of *relevant-slice condition* in debugging mentioned in Section 5, there are many other path condition based techniques that could benefit from *relevant-slice condition*.

Our *relevant-slice condition* can be used to minimize an existing test-suite [10, 18]. If a test-suite contains two test cases that have the same *relevant-slice condition*, these two test cases compute the output in the same way. Therefore, we can choose to eliminate one of them to make the test-suite smaller.

The work in [3] explores paths to generate program invariants. For each path explored, the path condition serves as a pre-condition and the symbolic program output is treated as a post-condition. Thus, each explored path produces a program invariant which is defined as such a (pre-condition, post-condition) pair. Similar approaches are used in [6, 12] to generate method summaries. Instead of using path condition, we can generate such program invariants using *relevant-slice conditions* — the *relevant-slice condition* is the pre-condition and for each *relevant-slice condition* explored, there is a unique symbolic output which serves as the post-condition. Moreover, the invariants generated using *relevant-slice*

conditions will be simpler (as *relevant-slice conditions* are smaller than path conditions) and fewer (since a single *relevant-slice condition* groups more paths).

8. DISCUSSION

In this paper, we have presented a novel path exploration method based on symbolic program outputs. Our path exploration dynamically groups paths on-the-fly, where two paths that have the same symbolic output are grouped together. Given such a path partitioning, we can generate a test case from each partition. This enables us to efficiently obtain a concise test-suite which stresses all possible input-output relationships in the program.

Our path exploration method is complete, that is, it covers all possible symbolic outputs in a given program. We also experimentally compare the efficiency and coverage of our method with respect to Directed Automated Random Testing, another path search method based on symbolic execution.

Apart from testing, the path partitioning computed by our method can be exploited in other software engineering activities. We have shown its use in the debugging of errors introduced by program changes, that is, in root-causing observable software regressions. By comparing the path partitioning in two program versions, we infer the semantic differences across the versions, leading to precise root cause identification.

Acknowledgements. This work was partially supported by a Ministry of Education research grant MOE2010-T2-2-073 (R-252-000-456-112 and R-252-100-456-112).

9. REFERENCES

[1] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. London. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*, ICSM '93, pages 348–357, Washington, DC, USA, 1993. IEEE Computer Society.

[2] P. Boonstoppel, C. Cadar, and D. Engler. Rwset: Attacking path explosion in constraint-based test generation. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–366, 2008.

[3] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 281–290, New York, NY, USA, 2008. ACM.

[4] L. De Moura and N. Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[5] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4), 2005.

[6] P. Godefroid. Compositional dynamic test generation. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 47–54, New York, NY, USA, 2007. ACM.

[7] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM.

[8] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[9] T. Gyimóthy, A. Beszédes, and I. Forgács. An efficient relevant slicing method for debugging. In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-7, pages 303–321, London, UK, 1999. Springer-Verlag.

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

[11] K. McMillan. Lazy annotation for program testing and verification. In *Computer Aided Verification*, pages 104–118. Springer, 2010.

[12] S. Person, M. Dwyer, S. Elbaum, and C. Pasareanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237. ACM, 2008.

[13] D. Qi, H. D. T. Nguyen, and A. Roychoudhury. Path exploration based on soymbolic output. Technical report, `http://dl.comp.nus.edu.sg/dspace/handle/1900.100/3347`, March 2011.

[14] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. DARWIN: an approach for debugging evolving programs. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 33–42, New York, NY, USA, 2009. ACM.

[15] R. Santelices and M. J. Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 195–206, New York, NY, USA, 2010. ACM.

[16] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[17] T. Wang and A. Roychoudhury. Dynamic slicing on Java bytecode traces. *ACM Trans. Program. Lang. Syst.*, 30:10:1–10:49, March 2008.

[18] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 41–50, New York, NY, USA, 1995. ACM.

[19] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 327–336, New York, NY, USA, 2003. ACM.

[20] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 238–248, New York, NY, USA, 2008. ACM.