THE NATIONAL UNIVERSITY
of SINGAPORE

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

TRC3/11

Path Exploration based on Symbolic Output

Dawei Qi, Hoang D. T. Nguyen and Abhik Roychoudhury

March 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

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.

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 [6] - 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.

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.


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

Figure 1: Sample program

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

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

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 [7] 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 10 is not executed, the output statement in line 15 is potentially dependent on the branch in line 9. This is to capture the fact that if line 9 is evaluated differently, the assignment in line 10 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 [9], 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 3 is considered.

```
1 int x; //input variable
2 int a = 0;
3 x = x - 1;
4 if(x > 0){
5     a = 1;}
6 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 $\pi \mid_C$ 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 $\pi \mid_C$ 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 $\pi \mid_C$.*

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 criteria C in π. Note that given a branch condition bc in $dsc_{(\pi,C)}$, the symbolic values of the variables used in bc must be the same in $dsc_{(\pi,C)}$ and pc_π. This is because assignments are considered in symbolic execution as shown in Figure 2 and each assignment instance in dynamic slice is contained in the full execution trace. Since path condition pc_π contains each branch condition in the execution trace π, we have $pc_\pi \Rightarrow dsc_{(\pi,C)}$. 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, 7].

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

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

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, 7]. 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 be 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 6. The *relevant-slice condition* for input $\langle x == 0\rangle$ is $\neg(x - 1 > 0)$, that is, the effect of the assignment in line 3 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 15. 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 $\langle x == 6, y == 2, z == 2\rangle$ as an example 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 15 is {4,15} - 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 9 can potentially affect the value of out in the slicing criteria. This is because if the branch in line 9 is evaluated differently (to true), the variable b is re-defined (in line 10) which affects the output variable out. Hence the relevant slice contains line 9. The entire relevant slice is {1,4, 9,15}, 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.

Just like the DART approach [6] 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 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 on line 3 is generated by negating the second branch condition of the *relevant-slice condition* in line 2 in Table 1. That is, when we solve $(x - y > 0) \wedge \neg(x + y > 10)$ the *relevant-slice condition* of the new input 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, path condition based path exploration 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 prove that our path exploration algorithm is complete, as far as *relevant-slice condition* coverage is concerned. 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 $b(\psi)$ to denote the branch instance of a branch condition ψ. We use $bc(b)$ to denote the branch condition generated by b. Given a *relevant-slice condition* or reordered *relevant-*

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

slice condition θ and a branch condition ψ, we use $\theta\backslash\psi$ to denote the result of removing ψ from θ. Recall that θ is a conjunction of branch conditions. If ψ is contained in θ, ψ is deleted from the conjunction to get $\theta\backslash\psi$. Otherwise, $\theta\backslash\psi$ is the same as θ.

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. 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 3. Note that our reorder 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 3 is on branch instances for simplicity. On the left of Figure 3, the dependencies among all the branch instances are provided. If there is an arrow from bj to bi, then bj is transitively dependent on bi. 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 condition*. The main algorithm keeps on processing the formulae in the stack when it is not empty. In each

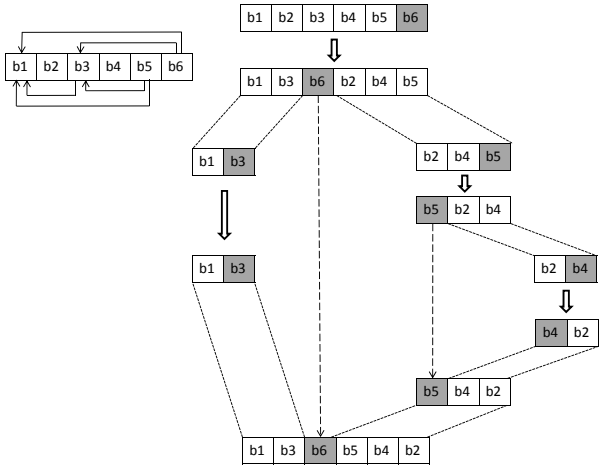

Figure 3: *Reorder* algorithm in action

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* $\varphi_i = \psi'_1 \wedge \psi'_2 \wedge \ldots \wedge \psi'_{i-1} \wedge \neg\psi'_i$, $1 \le i \le n \le m$, 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.

Algorithm 1 Path exploration

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 $b(\psi_i)$ is in relevant slice of $b(\psi_k)$ **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

Algorithm 2 Augmented reorder

procedure $reorder(seq, p)$
  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 $b(\psi_i)$ is in relevant slice of $b(\psi_k)$ **then**
      seq_1 = $seq_1 \wedge \psi_i$
    else
      seq_2 = $seq_2 \wedge \psi_i$
    end if
  end for
  assign the priority of $b(\psi_k)$ as $p@[b(\psi_k)]$
  seq'_1 = $reorder(seq_1, p@[b(\psi_k)])$
  seq'_2 = $reorder(seq_2, p)$
  return $seq'_1 \wedge \psi_k \wedge seq'_2$
end procedure

3.2 Proofs

Assumptions. We assume that the program input space is bounded. 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.

Execution Index. In the following proofs, we need to align/-compare different paths. Hence, it is critical to determine whether two statement instances from different paths are the same. We use the concept of execution index [15], two statement instances in different paths are the same iff. they have exactly the same "execution index". Given two paths π_1 and π_2 and statement instance s in π_1, we say s also appears in π_2 iff. in π_2 there is a statement instance s' such that the execution index of s' in π_2 is same as the execution index of s in π_1. In its simplest form, we can use the path from root to s in the Dynamic Control Dependence Graph of π_1 as the execution index of s in π_1.

Additional Notations Used in Proofs. Over and above the notations introduced earlier, we use the following notations in our proofs. The immediate post-dominator of a branch b is denoted as $postdom(b)$. We use $\rightarrow_d$ to denote dynamic data dependence, and $\rightarrow_c$ to denote dynamic control dependence. We use $\rightarrow_p$ to denote potential dependence. We use $\rightsquigarrow_d$ to denote transitive data dependence and $\rightsquigarrow_c$ to denote transitive control dependence. When no subscript is specified, we use $\rightarrow$ to denote any type of direct dependence and $\rightsquigarrow$ to denote the transitive closure of $\rightarrow$.

We use $\rightsquigarrow_s$ to denote a *special* kind of transitive dependence. Let u be a statement instance and b be a branch instance in path π, then $u \rightsquigarrow_s b$, iff. (i) there exist a variable v used at u, (ii) there is no definition of u between $postdom(b)$ and u and (iii) there is at least one static definition of v that is statically transitively control dependent on the static branch of b. There could potentially many static definitions of v that are statically transitively control dependent on static branch of b. Depending on whether these definitions of v are executed, there are two different scenarios when $u \rightsquigarrow_s b$. If all the definitions of v are not executed, then u is potentially dependent on b. Otherwise, u is data dependent on the last definition of v (say it is d), and d is control dependent on b. In both cases, there is a dependence chain from u to b.

Transformations. For the ease of our proofs, we statically transform any program in the following way. Note that the transformations do not affect the program semantics, and does not impact the generality of our proofs. (i) We add a dummy statement nop at the start of the program. The statement nop means no operation. (ii) If the slicing criteria (output statements(s)) is not a branch, we add a dummy branch that contains a dummy use for each variable appearing in the output statement(s). We use this branch as the slicing criterion C.

3.2.1 Priority sequence and shortened priority sequence

Recall that we need to prove the following property for *relevant-*

slice condition: if $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \psi_i$ is a prefix of a reordered $reordered\_rsc(C,t)$, the reordered *relevant-slice condition* of any input t' 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. There are two important facts to prove:(i) Each $b(\psi_k)$, $1 \leq k \leq i$, is included in the relevant slice in path $\pi(t')$. (ii) The relative order of branch conditions in $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ is not changed. To prove these two facts, we need to find out what is not changed between $reordered\_rsc(C,t')$ and $reordered\_rsc(C,t)$. In the following, we define a shortened priority sequence $sp(b)$ for each branch instance b. The shortened priority sequence has the following two properties:

1. Let t and t' be two inputs. Suppose $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$ is a prefix of $reordered\_rsc(C,\pi(t))$. If $t' \vDash \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1}$, then $sp(b(\psi_i))$ is the same in $\pi(t)$ and $\pi(t')$.
2. Let b_x and b_y be two branch instances in path $\pi(t)$. If $bc(b_x)$ is reordered before $bc(b_y)$ by the reorder algorithm in Algorithm 2, then $sp(b_x) > sp(b_y)$.

The first property means that the shortened priority sequence for the corresponding branch instance of each branch condition in $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$ is not changed between $reordered\_rsc(C,t')$ and $reordered\_rsc(C,t)$. The second property means that the shortened priority sequence essentially defines the order of branch conditions in a reordered *relevant-slice condition*. We explain how the shortened priority sequence is computed in the following.

To define the shortened priority sequence, we define the priority sequence first. In Algorithm 2, we have an augmented reorder algorithm. When $reorder$ is invoked from $Execute$, the value for the second parameter of the augmented reorder procedure is an empty list. The @ symbol in Algorithm 2 means list concatenation. Given the same parameters, the augmented reorder algorithm computes the same reordered sequence as the one in Algorithm 1. In the augmented reorder procedure, a priority sequence is computed for each branch instance along with the reorder process. Recall that the reorder process is done in a quick-sort-like fashion. When we divide the input sequence of the reorder procedure using ψ_k as the "pivot", if $b(\psi_i)$ is in the relevant slice of $b(\psi_k)$, then $b(\psi_k)$ is added to the end of the priority sequence of $b(\psi_i)$.

Let t be an input and b_x be a branch instance in path $\pi(t)$. Let the priority number for b_x in $\pi(t)$ be $p(b_x) = [\hat{b}_x^1, \hat{b}_x^2, \ldots, \hat{b}_x^\sigma]$. From this priority sequence, we form a new shortened priority sequence $sp(b_x)$ by selecting only the branches $\hat{b}_x^i$ such that $\hat{b}_x^i$ satisfies: there does not exists any $\hat{b}_x^j$ in $p(b_x)$ such that $\hat{b}_x^i \rightsquigarrow_c \hat{b}_x^j$. We denote the new shortened priority sequence as $sp(b_x) = [b_x^1, \ldots, b_x^\alpha]$. Note that the last branch instance in both $p(b_x)$ and $sp(b_x)$ is always b_x itself. Because of our transformation, if b_k is in $rs(C,\pi(t))$, then the first branch instance in both $p(b_x)$ and $sp(b_x)$ is from the slicing criteria C.

Let b_x and b_y be two branch instances in path $\pi(t)$. If $bc(b_x)$ is reordered before $bc(b_y)$, then one of the following two cases must be true: (i) There is a branch instance b in $p(b_x)$, where b is after b_y in time order and $b \nrightsquigarrow b_y$. (ii) $b_y \rightsquigarrow b_x$. In the first case, when b is used as the "pivot" in reorder algorithm, $bc(b_x)$ is reordered before $bc(b_y)$. In the second case, since $b_y \rightsquigarrow b_x$, then b_x should always be before b_y in the entire reorder process.

Suppose $sp(b_x) = [b_x^1, b_x^2, \ldots, b_x^\alpha]$ and $sp(b_y) = [b_y^1, b_y^2, \ldots, b_y^\beta]$. Let k be the maximal number that satisfies: for each i, $i \leq k$, $b_x^i == b_y^i$. Then we say $sp(b_x) > sp(b_y)$ if either (i) $k == min(\alpha,\beta)$ and $b_y \rightsquigarrow b_x$ or (ii) $k < min(\alpha,\beta)$ and $b_y^{k+1} \rightsquigarrow_c b_x^{k+1}$ or (iii) $k < min(\alpha,\beta)$, $b_x^{k+1} \nrightsquigarrow_c b_y^{k+1} \wedge b_y^{k+1} \nrightsquigarrow_c b_x^{k+1}$ and b_x^{k+1} is after b_y^{k+1} in time order. By this definition, it is impossible to have both $sp(b_x) > sp(b_y)$ and $sp(b_y) > sp(b_x)$.

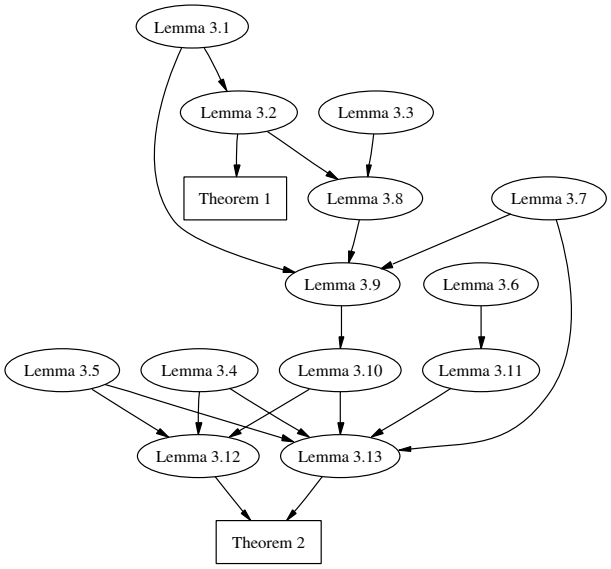

Figure 4: Structure of the proofs

3.2.2 *Proof structure*

We prove two theorems in this paper about *relevant-slice condition* and our path exploration algorithm based on *relevant-slice condition*. The proof structure is shown in figure 4. For the ease of understanding, we give the outline of our proofs and the relations between lemmas and theorems in the following.

In Theorem 3.1, we show that a *relevant-slice condition* could guarantee the unique symbolic values of the variables used in the slicing criteria. Symbolic value can be computed by dynamic symbolic execution. Each symbolic value is a 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 a 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. To prove Theorem 3.1, we actually prove the stronger Lemma 3.2. Let t and t' be two inputs and s be a statement instance in $\pi(t)$. In Lemma 3.2, we show that if $t' \vDash rsc(s,\pi(t))$, then the relevant slice w.r.t. s in $\pi(t')$ would be exactly the same as that in $\pi(t)$. Theorem 3.1 could be easily derived from Lemma 3.2.

In Theorem 3.2, given any feasible path π, we show that our path exploration algorithm would explore a path π' that share the same *relevant-slice condition* with π. This is concretized by showing that the algorithm gradually gets a sequence of *relevant-slice conditions* with each one closer to the *relevant-slice condition* of π than the previous one. Recall that the path exploration process is by iteratively negating a branch condition in a *relevant-slice condition*. Suppose we solve φ to get a new input t', we need to prove that the *relevant-slice condition* on $\pi(t')$ still contains φ as a prefix. Otherwise, the path exploration process would be out of order. Although this is obviously true for path condition, it is not obvious for *relevant-slice condition*. According to the result of relevant slice, some branch constraints do not appear in *relevant-slice condition* even they are in path condition. This important property of reordered *relevant-slice condition* is proved in Lemma 3.13. Let t and t' be two inputs. In Lemma 3.13, we prove that if $t' \vDash \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$, where $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$ is a prefix of $reodered\_rsc(C,\pi(t))$, then $reordered\_rsc(C,\pi(t'))$ must contain $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ as a prefix. Let the target

reordered *relevant-slice condition* be $g = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{n-1} \wedge \varphi_n$ and $reodered\_rsc(C, \pi(t))$ be f. If the first different branch condition between f and g is at location k, we prove that $\psi_k == \neg\varphi_k$ in Lemma 3.12. Combining with 3.13, we show that we could indeed get closer to π(having longer common prefix with $reordered\_rsc(C, \pi)$ by negating the kth branch condition in f. All the lemmas from Lemma 3.3 to 3.11 are used to gradually prove Lemma 3.12 and Lemma 3.13.

3.2.3 Full proofs

LEMMA 3.1. *Let t and t' be two inputs and s be a statement instance in $\pi(t)$. Suppose s is not in $\pi(t')$. Let b_s be the last branch instance in $\pi(t)$ that satisfies: $s \rightsquigarrow_c b_s$ and b_s is in both $\pi(t)$ and $\pi(t')$. Then b_s is evaluated differently in $\pi(t)$ and $\pi(t')$.*

PROOF. Let the control dependence chain from s to b_s in $\pi(t)$ be $s \rightsquigarrow_c b \rightarrow_c b_s$, where $b \rightarrow_c b_s$ is the last link in $s \rightsquigarrow_c b_s$. Note that b could be same as s. Assume to the contrary that b_s is evaluated to the same direction in $\pi(t)$ and $\pi(t')$. Then b must also be executed in $\pi(t')$. Therefore, b also satisfies: $s \rightsquigarrow_c b$ and b is in both $\pi(t)$ and $\pi(t')$. Since b is after b_s in time order, this contradicts that b_s is the last branch instance that satisfy this condition. Therefore, b_s is evaluated to different directions in $\pi(t)$ and $\pi(t')$. □

LEMMA 3.2. *Let t and t' be two inputs and s be a statement instance in $\pi(t)$. If $t' \vDash rsc(s, \pi(t))$, then s will be executed in $\pi(t')$, the variables used in s in $\pi(t')$ will have the same symbolic values as in $\pi(t)$, $rs(s, \pi(t'))$ is exactly the same as $rs(s, \pi(t))$ and each branch instance in $rs(s, \pi(t))$ is evaluated to the same direction in $\pi(t)$ and $\pi(t')$.*

PROOF. We prove this lemma by induction. Given the path $\pi(t)$, suppose it is a sequence $[s_0, s_1, \ldots, s_{n-1}, s_n]$.

Initial Step: According to our transformation, the statement instance s_0 must be from nop. Then $rsc(s_0, \pi(t))$ is $true$. It is obvious that s_0 satisfies Lemma 3.2.

Inductive Step: The induction hypothesis is: for each statement s_j, $j < i$, s_j satisfies Lemma 3.2. We need to prove that s_i also satisfies Lemma 3.2.

First we prove that s_i will be executed in $\pi(t')$. Let s_j be the statement prior to s_i such that $s_i \rightarrow_c s_j$. Then each statement in $rs(s_j, \pi(t))$ is also in $rs(s_i, \pi(t))$. So we have $rsc(s_i, \pi(t)) \Rightarrow rsc(s_j, \pi(t))$. Since $t' \vDash rsc(s_i, \pi(t))$, $t' \vDash rsc(s_j, \pi(t))$. By the induction hypothesis , s_j will be executed to the same direction in $\pi(t)$ and $\pi(t')$. This implies that s_i will be executed in $\pi(t')$.

The core of the inductive step is to prove that $rs(s_i, \pi(t'))$ is exactly the same as $rs(s_i, \pi(t))$. This is proved in two directions. (i) If $s_i \rightsquigarrow s'$ in $\pi(t')$, then $s_i \rightsquigarrow s'$ in $\pi(t)$. (ii) If $s_i \rightsquigarrow s'$ in $\pi(t)$, then $s_i \rightsquigarrow s'$ in $\pi(t')$.

Now, we prove that given any statement instance s' in $\pi(t')$, if $s_i \rightsquigarrow s'$ in $\pi(t')$, then it must also be $s_i \rightsquigarrow s'$ in $\pi(t)$. Suppose in $\pi(t')$, $s_i \rightarrow s_k \rightsquigarrow s'$ where s_k is another statement instance in $\pi(t')$. We first prove that $s_i \rightarrow s_k$ in $\pi(t)$ in two steps: (i) s_k appears in $\pi(t)$. (ii) $s_i \rightarrow s_k$ in $\pi(t)$.

We first prove that s_k appears in $\pi(t)$. We prove this by contradiction. Assume to the contrary that s_k does not appear in $\pi(t)$. We find the last control dependence ancestor of s_k that is in both $\pi(t)$ and $\pi(t')$. Let this statement be s_u. This means that s_u is the last statement in both $\pi(t)$ and $\pi(t')$ such that s_k is transitively control dependent on s_u in both the execution traces $\pi(t), \pi(t')$.

According to Lemma 3.1, the branch in s_u must be evaluated differently in $\pi(t)$ and $\pi(t')$. According to the type of $s_i \rightarrow s_k$ in $\pi(t')$ we have the following cases.

- (a) $s_i \rightarrow_c s_k$. The existence of s_i in $\pi(t)$ contradicts that s_k is not in $\pi(t)$.

- (b) $s_i \rightarrow_d s_k$ or $s_i \rightarrow_p s_k$. In this case, the existence of s_i in both paths $\pi(t), \pi(t')$ indicates that $s_i \not\rightsquigarrow_c s_u$ in $\pi(t')$ since s_u is evaluated differently in $\pi(t)$ and $\pi(t')$. Similarly, $s_i \not\rightsquigarrow_c s_u$ in $\pi(t)$. This means that s_i appears after $postdom(s_u)$ in both execution traces $\pi(t), \pi(t')$.

  Suppose $s_i \rightarrow_d s_k$ or $s_i \rightarrow_p s_k$ in $\pi(t')$ is caused by the use of variable v at s_i (in case of multiple such variables, we choose one randomly). There should be no definition of v between $postdom(s_u)$ and s_i in $\pi(t)$. Otherwise the definition would also appear in $\pi(t')$, making $s_i \rightarrow_d s_k$ or $s_i \rightarrow_p s_k$ impossible in $\pi(t')$. In $\pi(t)$, according to the definition of $\rightsquigarrow_s$, $s_i \rightsquigarrow_s s_u$ in $\pi(t)$. Therefore, s_u is in the relevant slice of s_i in $\pi(t)$. By the induction hypothesis , s_u is then evaluated to the same direction in $\pi(t)$ and $\pi(t')$, contradicting our original assumption that s_u is evaluated differently in $\pi(t)$ and $\pi(t')$.

Therefore, in both cases, we achieve a contradiction - thereby establishing that s_k must appear in $\pi(t)$.

Given that s_k is in $\pi(t)$, we prove that $s_i \rightarrow s_k$ in $\pi(t)$. According to the type of $s_i \rightarrow s_k$ in $\pi(t')$ we have the following cases. (a) $s_i \rightarrow_c s_k$ in $\pi(t')$. — The existence of s_i and s_k in $\pi(t)$ already shows that $s_i \rightarrow_c s_k$. (b) $s_i \rightarrow_d s_k$ or $s_i \rightarrow_p s_k$ in $\pi(t')$. — Suppose the dependence between s_i and s_k is caused by the use of variable v (in case of multiple such variables, we choose one randomly) at s_i in $\pi(t')$. Then $s_i \not\rightarrow s_k$ could only happen in $\pi(t)$ because v is redefined by another statement instance between s_k and s_i in $\pi(t)$. Suppose the last definition of v before s_i in $\pi(t)$ is at statement instance s_n. So we have $s_i \rightarrow s_n$ in $\pi(t)$. By the induction hypothesis, s_n will be executed in $\pi(t')$. The variable v will still be redefined by s_n in $\pi(t')$, which contradicts that $s_i \rightarrow_d s_k$ or $s_i \rightarrow_p s_k$ in $\pi(t')$. Therefore, we have proved that in both case, $s_i \rightarrow s_k$ in $\pi(t)$.

We have now proved that $s_i \rightarrow s_k$ in $\pi(t)$. According to induction hypothesis, we have the relevant slice of s_k is the same in $\pi(t)$ and $\pi(t')$, that is, $rs(s_k, \pi(t)) == rs(s_k, \pi(t'))$. Thus, for any statement instance s' such that $s_k \rightsquigarrow s'$ in $\pi(t')$ — we must have $s_k \rightsquigarrow s'$ in $\pi(t)$. Therefore, $s_i \rightarrow s_k \rightsquigarrow s'$ in $\pi(t)$. Thus, we have proved that given any statement instance s' in $\pi(t')$, if $s_i \rightsquigarrow s'$ in $\pi(t')$, then it must also be $s_i \rightsquigarrow s'$ in $\pi(t)$.

Next, we prove that given a statement instance s' in $\pi(t)$, if $s_i \rightsquigarrow s'$ in $\pi(t)$, then it must also be $s_i \rightsquigarrow s'$ in $\pi(t')$. Suppose $s_i \rightarrow s_j \rightsquigarrow s'$ in $\pi(t)$. According to induction hypothesis, s_j appears in $\pi(t')$ and $s_j \rightsquigarrow s'$ in $\pi(t')$. So we only need to prove that $s_i \rightarrow s_j$ in $\pi(t')$. According to the dependence type of $s_i \rightarrow s_j$, we have the following two cases.

- (a) $s_i \rightarrow_c s_j$ in $\pi(t)$. The existence of both s_i and s_j already implies that $s_i \rightarrow_c s_j$ in $\pi(t')$.

- (b) $s_i \rightarrow_d s_j$ or $s_i \rightarrow_p s_j$ in $\pi(t)$. We need to prove that the dependence between s_i and s_j still appears in $\pi(t')$. Suppose $s_i \rightarrow_d s_j$ or $s_i \rightarrow_p s_j$ in $\pi(t)$ is caused by the use of variable v at s_i. We prove $s_i \rightarrow_d s_j$ or $s_i \rightarrow_p s_j$ in $\pi(t')$ by contradiction. Assume to the contrary that this is not the case in $\pi(t')$. This could only happen if v is redefined between s_j and s_i in $\pi(t')$. Suppose the last definition of v before s_i in $\pi(t')$ is statement instance s_n, so $s_i \rightarrow_d s_n$ in $\pi(t')$. We have already established that for any statement instance s' in $\pi(t')$, if $s_i \rightsquigarrow s'$ in $\pi(t')$, then it must also be $s_i \rightsquigarrow s'$ in $\pi(t)$. Thus, s_n must also appear in $\pi(t)$, and $s_i \rightarrow_d s_n$

in $\pi(t)$. This contradicts that $s_i \rightarrow_d s_j$ or $s_i \rightarrow_p s_j$ in $\pi(t)$ (simply by the definition of dynamic data dependencies and potential dependencies). So if $s_i \rightarrow_d s_j$ or $s_i \rightarrow_p s_j$ is in $\pi(t)$, $s_i \rightarrow_d s_j$ or $s_i \rightarrow_p s_j$ is also in $\pi(t')$,

Therefore, we have proved by induction that $rs(s_i, \pi(t'))$ is exactly the same as $rs(s_i, \pi(t))$ (inductive step).

Since the entire slice of s_i is exactly the same in two paths, the symbolic values of the variables used in s_i must be exactly the same in $\pi(t)$ and $\pi(t')$. Suppose s_i uses α variables $v_1, v_2, \ldots, v_{\alpha-1}, v_\alpha$ to define variable $\bar{s}_i$. Let the corresponding definition of these variables be at $s_1^i, s_2^i, \ldots, s_{\alpha-1}^i, s_\alpha^i$. Note that each definition s_x^i, $1 \leq x \leq \alpha$, is same in both $\pi(t')$ and $\pi(t)$ since s_x^i is in the relevant slice of s_i. According the induction hypothesis, the symbolic value of each v_x is the same in $\pi(t)$ and $\pi(t')$, where $1 \leq x \leq \alpha$. Moreover, the definition of $\bar{s}_i$ is computed in exactly the same way (using the same operations) from $v_1, v_2, \ldots, v_{\alpha-1}, v_\alpha$ in $\pi(t)$ and $\pi(t')$. Therefore, the symbolic value of $\bar{s}_i$ in s_i is the same in $\pi(t)$ and $\pi(t')$. We know that $t' \vDash rsc(s_i, \pi(t))$. Therefore, t' should satisfy the branch constraints corresponding to the branches appearing in the relevant slice $rs(s_i, \pi(t)) == rs(s_i \pi(t'))$. Therefore each branch instance in $rs(s_i, \pi(t))$ is evaluated to the same direction in $\pi(t)$ and $\pi(t')$. This completes the proof. □

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.*

PROOF. Let t_1 and t_2 be two test inputs whose execution traces are π_1 and π_2 respectively. According to the theorem statement, we have $rsc(C, \pi_1) == rsc(C, \pi_2)$. Since $t_2 \vDash rsc(C, \pi_2)$, $t2 \vDash rsc(C, \pi_1)$. According to Lemma 3.2, the symbolic values of the variables used in C are exactly the same in $\pi(t_2)$ and $\pi(t_1)$, where $\pi(t_2) == \pi_2$ and $\pi(t_1) == \pi_1$. Therefore, the variables used in the slicing criteria C have the same symbolic values in π_1 and π_2. This completes the proof. □

LEMMA 3.3. *Let t be an input. Suppose the reordered relevant-slice condition in $\pi(t)$ is $reordered\_rsc(C, \pi(t)) = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$. Then for any i, $1 \leq i \leq k$, $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i \Rightarrow rsc(b(\psi_i), \pi(t))$.*

LEMMA 3.4. *Let t and t' be two inputs, given a branch instance b in $\pi(t)$, if $t' \vDash rsc(b, \pi(t)) \backslash bc(b)$, b will be executed in $\pi(t')$ and the variables used in b in $\pi(t')$ would have the same symbolic values as in $\pi(t)$.*

LEMMA 3.5. *Let t be an input. Let $reordered\_rsc(C, \pi(t))$ be $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$ in path $\pi(t)$. Then for any i, $1 \leq i \leq k$, $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \Rightarrow rsc(b(\psi_i), \pi(t)) \backslash \psi_i$.*

LEMMA 3.6. *Let b_x and b_y be two branch instances in path $\pi(t)$. If $bc(b_x)$ is reordered before $bc(b_y)$ by the reorder algorithm in Algorithm 2, then $sp(b_x) > sp(b_y)$.*

PROOF. Suppose $sp(b_x)$ is $[b_x^1, b_x^2, \ldots, b_x^{\alpha-1}, b_x^\alpha]$ and $sp(b_y)$ is $[b_y^1, b_y^2, \ldots, b_y^{\beta-1}, b_y^\beta]$. When $sp(b_x) \neq sp(b_y)$, let k be the maximal number that satisfies: for each i, $i \leq k$, $b_x^i == b_y^i$.

If $k == min(\alpha, \beta)$, it must be either $k == \alpha$ or $k == \beta$. If $k == \alpha$, then we have $b_x == b_x^\alpha == b_x^k == b_y^k$ and $b_y^k \rightsquigarrow b_y$. Therefore, we have $b_x \rightsquigarrow b_y$, which could not be possible when $bc(b_x)$ is reordered before $bc(b_y)$. Therefore, when $k == min(\alpha, \beta)$, it must be $k == \beta$ and $b_y \rightsquigarrow b_x$.

If $k < min(\alpha, \beta)$, we enumerate the relation between b_x^{k+1} and b_y^{k+1}.

1. $b_y^{k+1} \rightsquigarrow_c b_x^{k+1}$. This is possible.
2. $b_x^{k+1} \rightsquigarrow_c b_y^{k+1}$. We prove that it is not possible to have $bc(b_x)$ being reordered before $bc(b_y)$. This is proved through (i) $b_y^{k+1} \not\rightsquigarrow b_x$ (ii) There is no branch b after b_y^{k+1} such that b is in $p(b_x)$, but $b \not\rightsquigarrow b_y$. We first prove $b_y^{k+1} \not\rightsquigarrow b_x$ by contradiction. Assume to the contrary that $b_y^{k+1} \rightsquigarrow b_x$. According to the process of computing $p(b_x)$, if $b_y^{k+1} \rightsquigarrow b_x$ and $b_x^{k+1} \rightsquigarrow_c b_y^{k+1}$, then b_y^{k+1} is in $p(b_x)$. Since b_y^{k+1} is in $sp(b_x)$, b_y^{k+1} is also in $p(b_x)$. However, according to the definition of $sp(b_x)$, if $b_x^{k+1} \rightsquigarrow_c b_y^{k+1}$ and b_y^{k+1} is in $p(b_x)$, then b_x^{k+1} is not in $sp(b_x)$. This contradicts that b_x^{k+1} is in $sp(b_x)$. Next, we prove that there is no branch b after b_y^{k+1} such that b is in $p(b_x)$, but $b \not\rightsquigarrow b_y$. This is obvious since for each b after b_y^{k+1} and b is in $p(b_x)$, we have $b \rightsquigarrow b_x^{k+1} \rightsquigarrow b_y^{k+1} \rightsquigarrow b_y$, contradicting $b \not\rightsquigarrow b_y$. Therefore, we know that there is no branch b after b_y^{k+1} such that b_y is reordered after b_x by using b as the "pivot". So when $bc(b_y^{k+1})$ is used as the "pivot" in the reorder algorithm, either $bc(b_x)$ is already after $bc(b_y^{k+1})$ hence after $bc(b_y)$, or $bc(b_x)$ is still before $bc(b_y^{k+1})$. If $bc(b_x)$ is still before $bc(b_y^{k+1})$, given the "pivot" $bc(b_y^{k+1})$ and the fact that $b_y^{k+1} \not\rightsquigarrow b_x$, $bc(b_x)$ will be reordered after $bc(b_y^{k+1})$ hence after $bc(b_y)$. This contradicts that $bc(b_x)$ is before $bc(b_y)$.
3. $b_x^{k+1} \not\rightsquigarrow_c b_y^{k+1} \wedge b_y^{k+1} \not\rightsquigarrow_c b_x^{k+1}$. We prove that b_x^{k+1} is after b_y^{k+1} in time order using contradiction. Assume to the contrary that b_x^{k+1} is before b_y^{k+1} in time order. According to whether b_y^{k+1} is transitively dependent on b_x^{k+1}, we have the following two cases: (i) $b_y^{k+1} \rightsquigarrow b_x^{k+1}$, then the $(k+1)th$ element in $sp(b_x)$ should be either be b_y^{k+1} or some branch that b_y^{k+1} is transitively control dependent on. Neither case is true for b_x^{k+1}. (ii) $b_y^{k+1} \not\rightsquigarrow b_x^{k+1}$. We first show $b_y^{k+1} \not\rightsquigarrow b_x$. Assume to the contrary that $b_y^{k+1} \rightsquigarrow b_x$. Then when b_y^{k+1} is used as the "pivot" in the reorder process, b_x is before b_y^{k+1} and b_x^{k+1} is after b_y^{k+1}. Therefore, b_x and b_x^{k+1} are in two different sub-sequences, making it impossible to have b_x^{k+1} in the shortened priority sequence of b_x. This contradicts that b_x^{k+1} is in $sp(b_x)$. Given $b_y^{k+1} \not\rightsquigarrow b_x$, when b_y^{k+1} is used as the "pivot" in the reorder process, either $bc(b_x)$ is already after $bc(b_y^{k+1})$ hence after $bc(b_y)$, or $bc(b_x)$ is still before $bc(b_y^{k+1})$. If $bc(b_x)$ is still before $bc(b_y^{k+1})$, given the "pivot" $bc(b_y^{k+1})$, $bc(b_x)$ will be reordered after $bc(b_y^{k+1})$ hence after $bc(b_y)$. This contradicts that $bc(b_x)$ is before $bc(b_y)$. So it is impossible to have b_x^{k+1} before b_y^{k+1} in either case.

So we have either (i) $k == min(\alpha, \beta)$ and $b_y \rightsquigarrow b_x$ or (ii) $k < min(\alpha, \beta)$ and $b_y^{k+1} \rightsquigarrow_c b_x^{k+1}$. or (iii) $k < min(\alpha, \beta)$ and $b_x^{k+1} \not\rightsquigarrow_c b_y^{k+1} \wedge b_y^{k+1} \not\rightsquigarrow_c b_x^{k+1}$ and b_x^{k+1} is after b_y^{k+1} in time order. This is exactly the definition of $sp(b_x) > sp(b_y)$. □

LEMMA 3.7. *Let t be an input and b and b_k be two branch instances in $\pi(t)$. Suppose in $\pi(t)$, $sp(b_k)$ is $[b_k^1, b_k^2, \ldots, b_k^i]$. If $b_k^j \rightsquigarrow b$, where $1 \leq j < i$, and $postdom(b)$ is after $postdom(b_k^{j+1})$, then $bc(b)$ is before $bc(b_k)$ in $reordered\_rsc(C, \pi(t))$.*

PROOF. We first prove that b is not in $p(b_k)$. Based on the possible position of b in $\pi(t)$, we have the following two cases:(i) b is after $postdom(b_k^{j+1})$. Since $b_k^j \rightsquigarrow b$, b could only be between b_k^j and $postdom(b_k^{j+1})$. According to the process of computing the shortened priority sequence, any branch instances between b_k^j and $postdom(b_k^{j+1})$ cannot be in $p(b_k)$. (ii) b is before

$postdom(b_k^{j+1})$. Since $postdom(b)$ is after $postdom(b_k^{j+1})$, it must be $b_k^{j+1} \rightsquigarrow_c b$. Therefore, we have $b_k^{j+1} \rightsquigarrow_c b$ and b is in $p(b_k)$. This cannot happen given b_k^{j+1} is in $sp(b_k)$.

According to the reorder algorithm, if $bc(b_k)$ is reordered before $bc(b)$ and b is not in $p(b_k)$, then there must be a branch instance $\hat{b}_k^u$ in $p(b_k)$ such that $\hat{b}_k^u$ is after b and $\hat{b}_k^u \nrightsquigarrow b$. We will prove that such a $\hat{b}_k^u$ cannot exist. Such a $\hat{b}_k^u$ should be after $postdom(b)$, otherwise $\hat{b}_k^u \rightsquigarrow_c b$. Since $postdom(b)$ is after $postdom(b_k^{j+1})$, $\hat{b}_k^u$ is after $postdom(b_k^{j+1})$. However if $\hat{b}_k^u$ in $p(b_k)$ is after $postdom(b_k^{j+1})$, $\hat{b}_k^u \rightsquigarrow b_k^j \rightsquigarrow b$. So it is not possible to have any $\hat{b}_k^u$ in $p(b_k)$ such that $\hat{b}_k^u$ is after b and $\hat{b}_k^u \nrightsquigarrow b$. This means that $bc(b_k)$ cannot be reordered before $bc(b)$. Therefore, $bc(b)$ is before $bc(b_k)$ in $reordered\_rsc(C, \pi(t))$. □

LEMMA 3.8. *Let t and t' be two inputs. Let the reordered relevant-slice condition in $\pi(t)$ be $reordered\_rsc(C, \pi(t)) = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$. Then if $t' \vDash \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k$, $k \leq i$, for any j, $1 \leq j \leq k$, $b(\psi_j)$ is evaluated to the same direction in $\pi(t)$ and $\pi(t')$.*

LEMMA 3.9. *Let t and t' be two inputs. Suppose $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$ is a prefix of $reordered\_rsc(C, \pi(t))$. Let $b(\psi_k)$ be b_k. Suppose the shortened priority sequence for b_k in $\pi(t)$ is $sp(b_k) = [b_k^1, b_k^2, \ldots, b_k^i]$, where $b_k^i == b_k$. If $t' \vDash \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, then (i) For any j, $1 \leq j \leq k$, b_k^j also appears in $\pi(t')$. (ii) For any j, $1 \leq j < k$, each statement that is between b_k^j and $postdom(b_k^{j+1})$ in $rs(b_k^j, \pi(t'))$ is also in $rs(b_k^j, \pi(t))$. (iii) For any j, $1 \leq j < k$, each statement that is between b_k^j and $postdom(b_k^{j+1})$ in $rs(b_k^j, \pi(t))$ is also in $rs(b_k^j, \pi(t'))$.*

PROOF. We prove the claims in the lemma one by one.

For any j, $1 \leq j \leq k$, b_k^j also appears in $\pi(t')$. Suppose $b_k^j \rightarrow_c b_c$. Since $b_k^j \rightarrow_c b_c$, we have $b_k^j \rightsquigarrow b_c$ and $postdom(b_c)$ after b_k^j hence after $postdom(b_k^{j+1})$. According to Lemma 3.7, $bc(b)$ is reordered before $bc(b_k)$(same as ψ_k) in $reordered\_rsc(C, \pi(t))$. This means that the branch condition of b is actually one of the ψ_m, where $m \leq k-1$. Since $t' \vDash \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, according to Lemma 3.8, b will be evaluated to the same direction in $\pi(t)$ and $\pi(t')$. So b_k^j is also executed in $\pi(t')$.

For any j, $1 \leq j < k$, each statement that is between b_k^j and $postdom(b_k^{j+1})$ in $rs(b_k^j, \pi(t'))$ is also in $rs(b_k^j, \pi(t))$. We prove this by contradiction. Assume to the contrary that there is an s in $\pi(t')$, where $s \in rs(b_k^j)$ and s is between b_k^j and $postdom(b_k^{j+1})$, but s is not in $rs(b_k^j, \pi(t))$. There must exist two nodes $s1$ and $s2$ in $b_k^j \rightsquigarrow s$ in $\pi(t')$ such that $b_k^j \rightsquigarrow s1$ in $\pi(t)$, but $s1 \nrightsquigarrow s2$ in $\pi(t)$. If it is not the case, then in $\pi(t)$ we have $b_k^j \rightsquigarrow s$. We prove $s1 \rightarrow s2$ to draw the contradiction in two steps: (i) $s2$ appears in $\pi(t)$. (ii) $s1 \rightarrow s2$.

We first prove that $s2$ appears in $\pi(t)$. According to the dependence type from $s1$ to $s2$ in $\pi(t')$, we have the following two cases:(i) $s1 \rightarrow_c s2$. the existence of $s1$ in $\pi(t)$ shows that $s2$ also exists in $\pi(t)$. (ii) $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$. Assume to the contrary that $s2$ does not appear in $\pi(t)$. We find the last control ancestor $s3$ of $s2$ that is in both $\pi(t)$ and $\pi(t')$. According to Lemma 3.1, $s3$ is evaluated to different directions in $\pi(t)$ and $\pi(t')$. Suppose $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$ is caused by the use of variable v at $s1$(in case of multiple such variables, we choose one randomly). There should be no definition of v between $postdom(s3)$ and $s1$ in $\pi(t)$. Otherwise, that definition would be kept in $\pi(t')$, making $s1 \rightarrow s2$ impossible in $\pi(t')$. According to the definition of $\rightsquigarrow_s$, $s1 \rightsquigarrow_s s3$ in $\pi(t)$. Therefore, we have $b_k^j \rightsquigarrow s1 \rightsquigarrow s3$ in $\pi(t)$. Because $s2 \rightsquigarrow_c s3$, $postdom(s3)$ is after $s2$ and hence after $postdom(b_k^{j+1})$ in $\pi(t')$. Since the relative time order of any two statement instances does not change across different paths, the existence of both $s3$ and b_k^{j+1} in $\pi(t)$ indicates that $postdom(s_3)$ is after $postdom(b_k^{j+1})$ in $\pi(t)$. According to Lemma 3.7, $bc(s3)$ is reordered before $bc(b_k)$(same as ψ_k) in $reordered\_rsc(C, \pi(t))$. Since $t' \vDash \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, according to Lemma 3.8, $s3$ is evaluated to the same direction in $\pi(t)$ and $\pi(t')$. However, this contradicts that $s3$ is evaluated to different directions in $\pi(t)$ and $\pi(t')$.

Then, we show $s1 \rightarrow s2$ in $\pi(t)$. According to the dependence type from $s1$ to $s2$ in $\pi(t')$, we have the following two cases: (i) $s1 \rightarrow_c s2$. The existence of $s1$ and $s2$ already shows that $s1 \rightarrow_c s2$ in $\pi(t)$. (ii) $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$. Assume to the contrary that $s1 \nrightarrow s2$ in $\pi(t)$. Suppose $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$ is caused by the use of variable v at $s1$(in case of multiple such variables, we just choose one randomly). Therefore, $s1 \nrightarrow s2$ in $\pi(t)$ could only be v is redefined by some statement instance between $s1$ and $s2$ in $\pi(t)$. We denote this statement instance as $s4$. Suppose $s4$ is control dependent on $s5$ in $\pi(t)$, we have $b_k^j \rightsquigarrow s4 \rightsquigarrow s5$ and $postdom(s5)$ after $s4$ hence after $postdom(b_k^{j+1})$. According to Lemma 3.7, $bc(s5)$ is reordered before $bc(b_k)$ in $reordered\_rsc(C, \pi(t))$. Since $t' \vDash \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, according to Lemma 3.8, $s5$ is evaluated to the same direction in $\pi(t)$ and $\pi(t')$. Therefore, $s4$ will be executed in $\pi(t')$. This contradicts that $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$ in $\pi(t')$.

For any j, $1 \leq j < k$, each statement that is between b_k^j and $postdom(b_k^{j+1})$ in $rs(b_k^j, \pi(t))$ is also in $rs(b_k^j, \pi(t'))$. For a statement s that is in $rs(b_k^j)$ and is between b_k^j and $postdom(b_k^{j+1})$, We first prove that s exists in $\pi(t')$. Suppose $s \rightarrow_c b_c$. Therefore, we have $b_k^j \rightsquigarrow s \rightsquigarrow b_c$ and $postdom(b_c)$ after s hence after $postdom(b_k^{j+1})$. According to Lemma 3.7, $bc(b_c)$ is reordered before $bc(b_k)$(same as ψ_k) in $reordered\_rsc(C, \pi(t))$. Since $t' \vDash \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, according to Lemma 3.8, b_c is evaluated to the same direction in $\pi(t)$ and $\pi(t')$. Therefore, s will be executed in $\pi(t')$.

Then, we prove that $b_k^j \rightsquigarrow s$ in $\pi(t')$. Assume to the contrary that this is not the case. There must exist two nodes $s1$ and $s2$ in $b_k^j \rightsquigarrow s$ in $\pi(t)$ such that $b_k^j \rightsquigarrow s1$ in $\pi(t)$, but $s1 \nrightsquigarrow s2$ in $\pi(t')$. If it is not the case, then in $\pi(t')$ we have $b_k^j \rightsquigarrow s$. According the proof in the last paragraph, $s1$ and $s2$ are both executed in $\pi(t')$. According to the dependence type from $s1$ to $s2$ in $\pi(t)$, we have the following two cases:(i) $s1 \rightarrow_c s2$. The existence of $s1$ and $s2$ already shows that $s1 \rightarrow_c s2$ in $\pi(t')$. (ii) $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$. Suppose $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$ in $\pi(t)$ is caused by the use of variable v at $s1$(in case of multiple such variables, we just choose one randomly). Therefore, $s1 \nrightarrow s2$ in $\pi(t')$ could only be v is redefined by some statement instance between $s1$ and $s2$ in $\pi(t')$. We denote this statement instance as $s4$. According to the above proof, $s4$ also exists in $\pi(t)$. Therefore, v should also be redefined by $s4$ in $\pi(t)$, contradicting that $s1 \rightarrow_d s2$ or $s1 \rightarrow_p s2$ in $\pi(t)$. □

LEMMA 3.10. *Let t and t' be two inputs. Suppose $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$ is a prefix of $reordered\_rsc(C, \pi(t))$. If $t' \vDash \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, then $sp(b(\psi_k))$ is the same in $\pi(t)$ and $\pi(t')$.*

PROOF. Let $b(\psi_k)$ be b_k. Suppose the shortened priority sequence for b_k in $\pi(t)$ is $sp(b_k) = [b_k^1, b_k^2, \ldots, b_k^i]$, where $b_k^i == b_k$.

We prove $sp(b(\psi_k))$ is the same as $[b_k^1, b_k^2, \ldots, b_k^i]$ in the following steps:(i) We prove that $b_k^j \rightsquigarrow b_k^{j+1}$ for each j, $1 \leq j < k$, in $\pi(t')$. (ii) We prove that each b_k^j is in $p(b_k)$ in $\pi(t')$. (iii) We prove that for any branch instance in $p(b_k)$ in $\pi(t')$, either it is

transitively dependent on some b_k^j, where b_k^j is in $sp(b_k)$ in $\pi(t)$ or it is some b_k^j. (iv) We show that for each b_k^j in $\pi(t')$, if $b_k^j \rightsquigarrow_c b_c$ then b_c is not contained in $p(b_k)$ in $\pi(t')$.

We first show that $b_k^j \rightsquigarrow b_k^{j+1}$ in $\pi(t')$, where $1 \leq j < k$. Since $b_k^j \rightsquigarrow b_k^{j+1}$ and $b_k^j \nrightsquigarrow_c b_k^{j+1}$ in $\pi(t)$, there must be a statement instance s between $postdom(b_k^{j+1})$ and b_k^j in $\pi(t)$ such that $b_k^j \rightsquigarrow s$ and $s \rightsquigarrow_s b_k^{j+1}$. Note that such an s could be the same as b_k^j. Suppose $s \rightsquigarrow_s b_k^{j+1}$ is caused by the use of variable v at s(in case of multiple such variables, we choose one randomly). As proved above, between $postdom(b_k^{j+1})$ and b_k^j, $rs(b_k^j, \pi(t'))$ is exactly the same as $rs(b_k^j, \pi(t))$. Therefore, $b_k^j \rightsquigarrow s$ in $\pi(t')$ and there is not definition of v between $postdom(b_k^{j+1})$ and s in $\pi(t')$. According to the definition of $\rightsquigarrow_s$, $s \rightsquigarrow_s b_k^{j+1}$ is irrespective of the direction of b_k^{j+1}. So in $\pi(t')$, we also have $s \rightsquigarrow_s b_k^{j+1}$. So we have $b_k^j \rightsquigarrow b_k^{j+1}$ in $\pi(t')$.

Next, we prove that each b_k^j is in $p(b_k)$ in $\pi(t')$, where $1 \leq j \leq k$. Assume to the contrary that this is not the case. Since ψ_k is in $reorderd\_rsc(C, \pi(t))$, $b(\psi_k)$(same as b_k) is in $rs(C, \pi(t))$. Then the first element in $sp(b_k)$ must be from C. According to the proof in last paragraph, $b_k^1 \rightsquigarrow b_k$ in $\pi(t')$. Therefore, the first element of $p(b_k)$ in $\pi(t')$ would still be from C, which is b_k^1. So b_k^1 is in $p(b_k)$ in $\pi(t')$. For $j > 1$, suppose b_k^j is the first one in $sp(b_k)$ in $\pi(t)$ that is not in $p(b_k)$ in $\pi(t')$. Therefore, we have b_k^{j-1} is in $p(b_k)$ in $\pi(t')$. According to the process of computing $p(b_k)$, there must be some "pivot" b between b_k^{j-1} and b_k^j(including b_k^j), where $b_k^{j-1} \rightsquigarrow b$ and $b \nrightsquigarrow b_k^j$ and $b \rightsquigarrow b_k$. According to the proof in last paragraph, we have $b_k^{j-1} \rightsquigarrow b_k^j$ and $b_k^j \rightsquigarrow b_k$ in $\pi(t')$. Therefore, b could not be the same as b_k^j, meaning b could only be after b_k^j and before b_k^{j-1}. According to the possible locations of b, we have the following two cases: (i) b is between b_k^j and $postdom(b_k^j)$. If b is between b_k^j and $postdom(b_k^j)$, then $b \rightsquigarrow_c b_k^j$, contradicting that $b \nrightsquigarrow b_k^j$. (ii) b is between $postdom(b_k^j)$ and b_k^{j-1}. As shown in Lemma 3.9, $rs(b_k^{j-1}, \pi(t'))$ is exactly the same as $rs(b_k^{j-1}, \pi(t))$ between b_k^{j-1} and b_k^j. Since we have $b \nrightsquigarrow b_k^j$ in $\pi(t)$, $b \nrightsquigarrow b_k^j$ in $\pi(t')$ either. This contradicts that $b \rightsquigarrow b_k^j$ in $\pi(t')$. In each case, we got a contradiction showing that the assumption is wrong. So we have b_k^j is in $p(b_k)$ in $\pi(t')$.

Then, we prove that for given any branch instance in $p(b_k)$ in $\pi(t')$, either this branch instance is transitively dependent on some b_k^j, where b_k^j is in $sp(b_k)$ in $\pi(t)$, or it is some b_k^j. This is the same as: for any branch b, if b is not between any pair of b_k^j and $postdom(b_k^j)$, $1 < j \leq k$, then b cannot be in $p(b_k)$ in $\pi(t')$. Note that $j > 1$ is because the range between b_k^1 and $postdom(b_k^1)$ is after the slicing criteria C(same as b_k^1) in time order. Assume to the contrary that such a b exists, then b must be between some b_k^j and $postdom(b_k^{j+1})$. According to Lemma 3.9, between $postdom(b_k^{j+1})$ and b_k^j, $rs(b_k^j, \pi(t'))$ is exactly the same as $rs(b_k^j, \pi(t))$, then such b is also in $\pi(t)$. According to the process of computing $p(b_k)$, b is contained in $p(b_k)$ in $\pi(t)$, contradicting that $p(b_k)$ does not contain any branches that are between $postdom(b_k^{j+1})$ and b_k^j. Therefore, we have proved that such b could not exist.

Finally, we show that for each b_k^j in $\pi(t')$, if $b_k^j \rightsquigarrow_c b_c$ then b_c is not contained in $p(b_k)$ in $\pi(t')$. Assume to the contrary that there exists a b_c, $b_k^j \rightsquigarrow_c b_c$ and b_c is contained in $p(b_k)$ in $\pi(t')$. According to proof in the last paragraph b_c must be either transitively dependent on some b_k^i or b_c is the same as b_k^i. In either case, we have $b_k^j \rightsquigarrow_c b_k^i$ in $\pi(t')$. Recall that control dependence between two statement instances are preserved across paths as long as the two statement instances both exist. Since b_k^j and b_k^i are also in $\pi(t)$, we have $b_k^j \rightsquigarrow_c b_k^i$ in $\pi(t)$. Therefore b_k^j can not be in $sp(b_k)$ in $\pi(t)$, contradicting that b_k^j is in $sp(b_k)$ in $\pi(t)$.

According to the process of computing shortened priority sequence, $sp(b_k)$ in $\pi(t')$ would be $[b_k^1, b_k^2, \ldots, b_k^i]$. □

LEMMA 3.11. *Let t be an input and b_x be a branch instance in $rs(C, \pi(t))$. If the shortened priority sequence of b_x in $\pi(t)$ is $sp(b_x) = [b_x^1, \ldots, b_x^\alpha]$, then for any i, $1 \leq i < \alpha$, $b_x^i \rightsquigarrow b_x^{i+1}$. This essentially means that there is a dependence chain from slicing criteria to b_x, which means b_x will be included in $rs(C, \pi(t))$.*

LEMMA 3.12. *Let π_1 and π_2 be two paths. Let f and g be $reordered\_rsc(C, \pi_1)$ $reordered\_rsc(C, \pi_2)$ respectively. Suppose f is $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{j-1} \wedge \varphi_j$ and g is $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$. If the first different branch condition between f and g is at location k, then $\varphi_k == \neg\psi_k$.*

PROOF. We first show that $b(\varphi_k)$ and $b(\psi_k)$ must be the same. We prove this by contradiction. Assume to the contrary that $b(\varphi_k)$ and $b(\psi_k)$ are different. Let $b(\varphi_k)$ be b_x and $b(\psi_k)$ be b_y. Since the first different branch condition between f and g is at location k, $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{k-1}$ (same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$) is satisfied by the input of both paths. , Since the input of π_1 satisfy $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{k-1}$, b_y is contained in $rs(C, \pi_1)$ according to Lemma 3.11. The branch condition $bc(b_y)$ should not be in $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, which is the same as $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{k-1}$. So $bc(b_y)$ could only be after $bc(b_x)$(same as φ_k) in $reordered\_rsc(C, \pi_1)$. Similarly, $bc(b_x)$ could only be after $bc(b_y)$ in $reordered\_rsc(C, \pi_2)$. According to Lemma 3.6, we have $sp(b_x) > sp(b_y)$ from π_1. Similarly we have $sp(b_y) > sp(b_x)$ from π_2. This contradicts that the shortened priority sequences are the same in both paths by Lemma 3.10. So $b(\varphi_k)$ and $b(\psi_k)$ must be the same.

According to Lemma 3.4 and 3.5, $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{k-1}$ can guarantee that the symbolic values of the variables used at $b(\varphi_k)$(same as $b(\psi_k)$) are the same in π_1 and π_2. So φ_k could only be different from ψ_k if the branch $b(\varphi_k)$ and $b(\psi_k)$ are evaluated to different directions in π_1 and π_2. So we have $\varphi_k == \neg\psi_k$. □

LEMMA 3.13. *Let t and t' be two inputs. If $t' \vDash \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$, where $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$ is a prefix of $reodered\_rsc(C, \pi(t))$, then $reordered\_rsc(C, \pi(t'))$ must contain $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ as a prefix.*

PROOF. We will prove the following properties of $\pi(t')$:

- Each $b(\psi_k)$, $1 \leq k \leq i$, in $\pi(t)$ is executed in $\pi(t')$ and the variables used in each $b(\psi_k)$ have the same symbolic values in $\pi(t)$ and $\pi(t')$.
- Each $b(\psi_k)$, $1 \leq k \leq i$, in $\pi(t)$ is contained in $rs(\pi(t'))$.
- The order of the branch conditions is the same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ in $\pi(t')$.

Each $b(\psi_k)$, $1 \leq k \leq i$, in $\pi(t)$ is executed in $\pi(t')$ and the variables used in each $b(\psi_k)$ have the same symbolic values in $\pi(t)$ and $\pi(t')$. Since $k \leq i$, so $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i \Rightarrow \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$. According to Lemma 3.4 and 3.5, each $b(\psi_k)$ in $\pi(t)$ is executed and the variables used in each $b(\psi_k)$ have the same symbolic values in $\pi(t)$ and $\pi(t')$.

Each $b(\psi_k)$, $1 \leq k \leq i$, in $\pi(t)$ is contained in $rs(\pi(t'))$. Since $k \leq i$, so $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i \Rightarrow \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$. According to Lemma 3.11, each $b(\psi_k)$ in $\pi(t)$ is contained in $rs(\pi(t'))$.

The order of the branch conditions is the same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ in $\pi(t')$. Let ψ_j and ψ_k be any two branch conditions in $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$, where $1 \leq j < k \leq i$. According to

Lemma 3.7, if ψ_j is before ψ_k, then $sp(b(\psi_j)) > sp(b(\psi_k))$. According to Lemma 3.10, the priority sequence of $b(\psi_j)$ and $b(\psi_k)$ in $\pi(t')$ are the same as those in $\pi(t)$ respectively. Therefore in $\pi(t')$, we also have $sp(b(\psi_j)) > sp(b(\psi_k))$. This shows that the relative order of any two branch conditions in $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ are the same $\pi(t)$ and $\pi(t')$. Therefore, the order of the branch conditions is the same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$. in $\pi(t')$.

The direction of b_j is restricted by the corresponding branch condition in $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$. According to the first two properties above, each branch condition in $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ still appears in $reordered\_rsc(C, \pi(t'))$. And the order of these branch conditions in $reordered\_rsc(C, \pi(t'))$ is the same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$. So $reordered\_rsc(C, \pi(t'))$ contains $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ as a prefix. □

We now prove the completeness of our path search method.

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 input space (total number of possible input values) of P is bounded.*

PROOF. Consider *any* input t in program P, its execution trace $\pi(t)$ and the associated reordered *relevant-slice condition* g. We use $dist(f, g)$ to denote the distance from f to g where f is also a reordered *relevant-slice condition* of some path. Suppose $f = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{j-1} \wedge \varphi_j$ and $g = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$. Let k be the number such that (i) for all $m \leq k$ we have $\varphi_m == \psi_m$, and (ii) either $k == min(i, j)$ or $\varphi_{k+1} \neq \psi_{k+1}$. We define $dist(f, g) \equiv 1 - \frac{k}{i}$. When $dist(f, g) == 0$, f and g are the same. The definition of $dist$ is asymmetric, that is, $dist(f, g) \neq dist(g, f)$ is possible.

In Algorithm 1, we maintain a $f_{current}$ which has the closest distance to g among all the explored *relevant-slice conditions*. Suppose $f_{current} = \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{j-1} \wedge \varphi_j$ and $g = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \psi_i$. Suppose the first different branch condition between $f_{current}$ and g is at location $k + 1$. When $f_{current}$ is explored, the partial *relevant-slice condition* $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi_{k+1}$ is pushed into the stack. This formula will be eventually processed by our path search algorithm, provided the total number of program paths is finite (which is the case since the input space of the program is finite, and each input traces exactly one path).

According to Lemma 3.12, $\neg\varphi_{k+1} == \psi_{k+1}$. It is clear that $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi_{k+1}$ is the same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k \wedge \psi_{k+1}$. Note that $g = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_i$ is satisfiable, as g is the *relevant-slice condition* of a feasible path $\pi(t)$. Since $k < i$ ($f_{current}$ and g are same up to the first k conjuncts), $g \Rightarrow \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k \wedge \psi_{k+1}$. Since g is satisfiable, $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi_{k+1}$ (same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k \wedge \psi_{k+1}$) is also satisfiable. Let t_0 be an input which satisfies $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi_{k+1}$, that is $t_0 \models \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi_{k+1}$. Using Lemma 3.13 we get that $reordered\_rsc(C, \pi(t_0))$ contains $\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_k \wedge \neg\varphi_{k+1}$ (which is same as $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k \wedge \psi_{k+1}$) as a prefix. By the definition of distance $dist$, the distance from $reordered\_rsc(C, \pi(t_0))$ to g should be

$$dist(reordered\_rsc(C, \pi(t_0)), g) \leq 1 - \frac{k+1}{i} < 1 - \frac{k}{i}$$

Replacing $f_{current}$ with $reordered\_rsc(C, \pi(t_0))$ will therefore decrease $dist(f_{current}, g)$. Thus, from $f_{current}$ our path search algorithm moves to the execution trace for input t_0 in one step. Since g contains only i conjuncts, we need at most i such steps to make $dist(f_{current}, g)$ to be 0. When $dist(f_{current}, g) == 0$, we have a path $\pi(t')$ that has the same reordered *relevant-slice condition* with g (such a t' can be found since in each step of replacing $f_{current}$ we obtain a feasible execution trace which is executed by at least one program input). Since the reordered *relevant-slice conditions* of $\pi(t)$ and $\pi(t')$ are identical, therefore the *relevant-slice conditions* of $\pi(t)$ and $\pi(t')$ must be identical. □

4. IMPLEMENTATION

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

4.1 Engine

Our implementation is based on JSlice [12][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 5.

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 without fully uncompressing it. We design our extension of JSlice to retain this feature (of analyzing compressed traces with decompression).

In Figure 5, 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 $[1, 2, 5, 6, 9, 10, 15]$. 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 almost all of them are handled in our implementation engine. In particular, bytecode types related to multi-threading and exception are not handled. Moreover, precisely modeling the semantics of certain bytecode types may be difficult. In the "Threats to Validity" section (Section **??**), we discuss the bytecode types which are not precisely modeled in our current implementation.

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. However, these values are needed when the semantics of some operations cannot be precisely modeled. 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 modeled. In this case, we

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

Relevant slice	Symbolic values	To be explained variables	Relevant slice condition
`15 out = b;`	$\{ out \rightarrow b \}$	$\{ b \}$	true
`10 b = a;`	$\{out \rightarrow a, b \rightarrow a \}$	$\{ a \}$	true
`9 if(x+y > 10)`	$\{out \rightarrow a, b \rightarrow a \}$	$\{ x, y \}$	$x + y > 10$
`6 a = x;`	$\{out \rightarrow x, b \rightarrow x, a \rightarrow x \}$	$\{ x, y \}$	$x + y > 10$
`5 if(x-y >0)`	$\{out \rightarrow x, b \rightarrow x, a \rightarrow x \}$	$\{ x, y \}$	$x - y > 0 \wedge x + y > 10$
`2 int out;`	$\{out \rightarrow x, b \rightarrow x, a \rightarrow x \}$	$\{ x, y \}$	$x - y > 0 \wedge x + y > 10$
`1 int x, y, z; //input`	$\{out \rightarrow x, b \rightarrow x, a \rightarrow x \}$	$\{ \}$	$x - y > 0 \wedge x + y > 10$

Table 3: Backward symbolic execution example

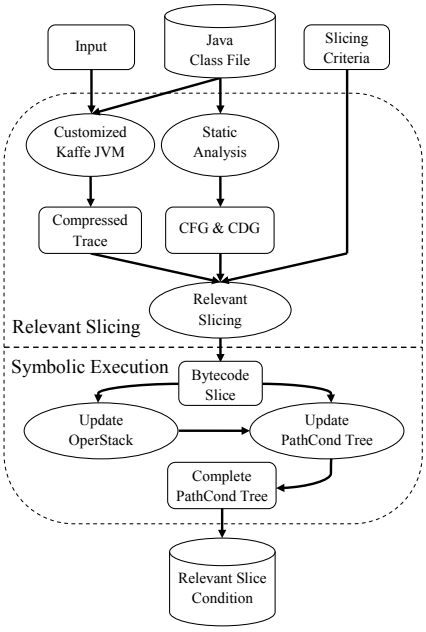

Figure 5: Architecture of *relevant-slice condition* computation

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).

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[3] as the SMT solver for our tool.

4.2 Approximation of Path Search

The core component of our path search algorithm (Algorithm 1) is the reorder procedure for reordering *relevant-slice conditions*. Although reordering of *relevant-slice conditions* can be performed efficiently, in our JSlice setup reordering of *relevant-slice conditions* turn out to be very expensive. This is because reordering involves many relevant slice computations. Such repeated relevant slicing is inefficient using JSlice since a lot of the time in each slicing computation goes into the computation of the static control/data dependencies (which are needed to compute the dynamic control dependencies and potential dependencies respectively).

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

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

In order to alleviate this inefficiency , we have approximated the reorder procedure in our path search implementation. Thus, our implementation of Algorithm 1 in JSlice is approximate and not exact. This approximation introduces incompleteness into our path search *implementation*. However, the incompleteness is only in the JSlice based implementation, the path search method is complete.

To understand the approximation we introduce into the reordering of *relevant-slice conditions*, take the sample program in Figure 1 as an example. Recall that the simple but problematic DART-like path exploration using *relevant-slice condition* is shown in Table 1. In particular, when we negate the second branch condition of the *relevant-slice condition* in row 2, solving $(x-y > 0) \wedge \neg(x+y > 10)$ generates a new input whose *relevant-slice condition* is $\neg(x + y > 10)$. The branch condition $(x - y > 0)$ disappears in the new *relevant-slice condition*. As we have mentioned in Section 2, this is because whether the branch $b1$ (`if (x-y >0)`) appears in relevant slice depends on which direction b2 (`if (x +y >10)`) is evaluated to. Our approximation method observes this relationship between $b1$ and $b2$ as follows — when $b2$ is negated, branch $b1$ disappears from the relevant slice. After we detect this relationship between $b1$ and $b2$, it is used to reorder the *relevant-slice conditions*. By reordering the branch condition of $b2$ before the branch condition of $b1$, the path exploration process is shown in Table 2.

Let the reordered *relevant-slice condition* of execution trace $\pi(t)$ from input t be $\psi_1 \wedge \ldots \psi_{n-1} \wedge \psi_n$. Suppose we solve $\psi_1 \wedge \ldots \psi_{m-1} \wedge \neg\psi_m$, where $m \leq n$, to generate a new input t'. If the relevant slice of execution trace $\pi(t')$ does not contain b_i (the branch corresponding to ψ_i) where $i < m \leq n$, then we can detect whether the appearance of b_i appears in the relevant slice depends on the direction of evaluation of b_m (the branch corresponding to ψ_m). If we detect that the appearance of b_i in relevant slice is dependent on the direction of b_m, we will keep track of the pair (b_i, b_m). We detect and maintain a set of such branch pairs during the execution of our path search algorithm. This set is used in the reordering the *relevant-slice conditions*. In particular, if we know (b_i, b_m) is in the set of branch pairs, while reordering a *relevant-slice condition* of the form $(\ldots \wedge \ldots \wedge \psi_i \wedge \ldots \wedge \psi_m \ldots)$ the branch condition of b_m is placed before that of b_i in the reordered *relevant-slice condition*.

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 [3] repository. The lines of code (LOC) in each program are also shown.

Recall that our path search is complete, but an approximation

Subject prog.	Size (LOC)	Coverage (RSC)	Coverage (DART)	Time		#Testcases		Avg. formula size		#Solver calls	
				RSC	DART	RSC	DART	RSC	DART	RSC	DART
tcas	113	100%	54%	6.6s	14.7s	37	120	5632	61663	452	987
BinarySearchTree	175	76%	29%	14.6s	61.9s	175	527	3174	38310	472	3188
OrdSet	211	64%	28%	4.1s	10.8s	20	59	5513	53932	150	293
Schedule	257	100%	100%	0.4s	40.1s	3	186	1775	59061	11	932
Disjoint Set	102	90%	25%	12.1s	64.5s	60	278	7297	167678	523	3855

Table 4: Experiments in full program exploration

in the reordering of *relevant-slice conditions* introduces some degree of incompleteness. The "Coverage (RSC)" column in Table 4 measures how much incompleteness is introduced by the approximated reorder method described in section 4.2. The numbers in the "Coverage (RSC)" column are computed as follows. Let the program being explored be P. We employ path enumeration on P to explore all the feasible paths and construct a test-suite T_{all} which covers the set of all feasible paths in P. For each test case t in T_{all}, we compute the *relevant-slice condition* on the execution trace of t and put this *relevant-slice condition* into a set S_{all}. Similarly, we generate a test-suite T_{RSC} for program P using our path exploration method (with the approximation in re-ordering). 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 "Coverage (RSC)" column in Table 4 is $\frac{|S_{RSC}|}{|S_{all}|}$. As shown in Table 4, our method cannot always achieve 100 percent relevant slice coverage due to the approximation in reordering. Note that this does affect the validity of the completeness claim in Theorem 3.2, the approximation/incompleteness is only in the implementation.

We now compare the coverage in our *relevant-slice condition* based search with the coverage of Directed Automated Random Testing (DART). 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 covered in DART search. The time budget given to the DART search is exactly the same as the time taken by our *relevant-slice condition* based search to finish. The "Coverage (DART)" column in Table 4 is computed as follows. We set the time limit for DART as the time taken by our approximate implementation of *relevant-slice condition* based path search. Suppose during this given time, DART generates a test-suite T_{DART}. For each test case t in T_{DART}, we compute the *relevant-slice condition* on the execution trace of t and put this *relevant-slice condition* into a set S_{DART}. Then we report the ratio $\frac{|S_{DART}|}{|S_{all}|}$. As shown in columns 3-4 of Table 4, *given the same amount of time* DART achieves less relevant slice coverage than our method.

In columns 5-12 of Table 4, we compare the time, number of generated test cases, formula size and number of solver calls between our method and DART. For getting these numbers, both our method (RSC) and the DART method are *run to completion*, and the running times are recorded. 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.

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 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 [9]. Given two program versions P and P', and a test case t which passes in P but fails in P', the work in [9] 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

Subject prog.	Stable version	Buggy version	Time from PC	Time from RSC	Results from PC	Results from RSC
JLex	1.2.1 (7290 LOC)	1.1.1 (6984 LOC)	543 min	15 min	50 LOC	3 LOC

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

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). Both methods are *fully* automated. Our subject program to be debugged 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 conditions* 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.

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. hence the possible null pointer deference is not spotted by the generated test-suite. Realistically, our test-suite construction could be supplemented by techniques to statically detect possible program crashes, such as memory error detection [14].

The completeness proof of Algorithm 1 in Section 3.2.3 is under the assumption that the semantics of the different program statement executed in a trace is precisely modeled in the computed *relevant-slice condition* of that execution trace. However, in our implementation, certain program features are not precisely modeled, which causes our path exploration to be incomplete. In particular, polymorphism and arrays are not precisely modeled in our current implementation.

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

7. RELATED WORK

The technique proposed in this paper is based on dynamic path exploration [6, 11] and relevant slicing [1, 7, 12]. 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 [4], function summaries are generated and exploited. In [5], 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 [10], 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 [10] also compute a "path family condition" for each path family that could characterize that path family. The main difference between our work and [10] lies in the static vs. dynamic nature of the two techniques. [10] 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 [10] 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.

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 [8, 13]. 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 of [2] 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. 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 which 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.

We have proved that 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.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.