# Path Exploration based on Symbolic Output

DAWEI QI, HOANG D.T. NGUYEN, ABHIK ROYCHOUDHURY
National University of Singapore

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 testing and debugging of software regressions.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Symbolic execution*; D.3.4 [**Programming Languages**]: Processors—*Debuggers*

General Terms: Experimentation, Reliability

Additional Key Words and Phrases: Software Testing, Symbolic Execution, Software Evolution

## 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 [Godefroid et al. 2005] - 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

An initial version of this paper was published as [Qi et al. 2011a] in ESEC-FSE 2011, held in Szeged, Hungary. The conference paper is provided with this submission. Authors' e-mails: {dawei,hoangdtn,abhik}@comp.nus.edu.sg
Address of the authors: School of Computing, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417. *This work was partially supported by Singapore's Ministry of Education research grant MOE2010-T2-2-073.*

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

```
1  int foo(int x, int y, int 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     System.out.println("square(z) > 3");
13   else
14     System.out.println("square(z) <= 3");
15   out = b;
16   return out; //slicing criteria
17 }
```

Fig. 1: Sample program

along different program paths. However, we expect that the number of such symbolic expressions to be substantially lower than the number of program paths. In other words, a large number of paths can be considered "equivalent" since the symbolic expressions describing the output are the same.

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

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

The summary given in the preceding forms a "semantic signature" of the program as far as the output variable `out` is concerned. Note that there are *only three* cases in the semantic signature - whereas there are *eight paths* in the program. Thus, such a semantic signature can be much more concise than an enumeration of all paths.

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

How do we partition paths? The answer to this question lies in the computation of the *output* variable. We consider two program paths to be "equivalent" if they have the same relevant slice [Gyimóthy et al. 1999] 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 assignment to `out` 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 variable `out` in line 15. 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 [Qi et al. 2009], 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 pin-pointed 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.

Finally, we show two applications of the "semantic signature" produced by our path partitioning method. We first apply the "semantic signature" on test-suite augmentation, which is to augment the existing test-suite after a program changes. We compare the semantic signatures of the previous and current program versions. Differences in the signatures lead to test cases that have different outputs in the two versions. These test cases are used to augment the existing test-suite. The "semantic signature" can also help uncover Finite State Automata (FSA) from real programs. For programs implementing FSA, uncovering the implemented FSA provides great help in understanding these programs. Applying our method, by focusing on the FSA state, the computed "semantic signatures" concisely represent the transitions of the FSA state, from which we can easily construct the FSA.

## 2. OVERVIEW

We begin with a few definitions.

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

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 example in Figure 2 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.

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 $\pi$ be the execution trace of $t$ in $P$. Let $\pi \mid_C$ denote the projection of $\pi$ w.r.t. the dynamic slice of $C$ in $\pi$. In other words, a statement instance $s$ in $\pi$ is included in the projection $\pi \mid_C$ if and only if $s$ is in the backward dynamic slice of $C$ on $\pi$. The dynamic slice condition of $C$ in $\pi$ is the path condition computed over the projected trace $\pi \mid_C$.*

```
int foo(int x){ //input variable
  int a = 0;
  x = x - 1;
  if(x > 0)
    a = 1;
  out = a;
  return out;
}
```

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

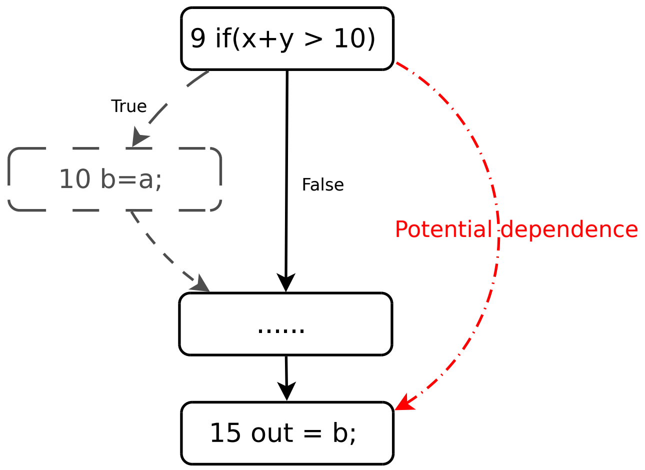


Fig. 3: Example of potential dependence. The solid arrows denote the execution path. According to Definition 3, (i) the variable $b$ is not defined between line 9 and line 15 but there exists a path (though line 10) along which $b$ is defined, and (ii) evaluating the branch at line 9 differently may cause the path through line 10 to be executed. Therefore, line 15 is potentially dependent on line 9.

Slice conditions are weaker than path conditions, that is, $pc_\pi \Rightarrow dsc_{(\pi,C)}$ where $dsc_{(\pi,C)}$ is the dynamic slice condition of any slicing criteria $C$ in $\pi$ (see our technical report [Qi et al. 2011b] for a simple proof of this claim). We now refine dynamic slice condition to *relevant-slice condition* - the central concept behind our path partitioning. But first, let us recall the notion of potential dependencies and relevant slices [Agrawal et al. 1993; Gyimóthy et al. 1999].

DEFINITION 3 (POTENTIAL DEPENDENCE [AGRAWAL ET AL. 1993]). *Given an execution trace $\pi$, let $s$ be a statement instance and $br$ be a branch instance that is before $s$ in $\pi$. We say that $s$ is potentially dependent on $br$ iff. there exists a variable $v$ used in $s$ such that (i) $v$ is not defined between $br$ and $s$ in trace $\pi$ but there exists another path $\sigma$ from $br$ to $s$ along which $v$ is defined, and (ii) evaluating $br$ differently may cause this untraversed path $\sigma$ to be executed.*

An example of potential dependence for the program in Figure 1 is shown in Figure 3.
We now introduce the notion of a relevant slice, and *relevant-slice condition*, a logical formula computed over a relevant slice.

DEFINITION 4 (RELEVANT SLICE). *Given an execution trace $\pi$ and a slicing criteria $C$ in $\pi$, the relevant slice in $\pi$ w.r.t. $C$ contains a statement instance $s$ in $\pi$ 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 [Agrawal et al. 1993; Gyimóthy et al. 1999]. In standard relevant slicing algorithm, if a statement instance $s$ is included only by potential dependence, the statement instances that are only control dependent by $s$ are not included in the relevant slice. We have removed this restriction to simplify the definition of relevant slice, it is simply the transitive closure of three kinds of program dependencies — dynamic data dependencies, dynamic control dependencies and potential dependencies. In the rest of the paper, all appearances of relevant slice and *relevant-slice condition* refer to this simplified definition of relevant slice.

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

We take the example program in Figure 2 to show that the effect of assignments is also considered in *relevant-slice condition* computation (just as assignments are considered in path condition computation). Let the slicing criteria be the value of `out` in line 7. The relevant slice for input $\langle x == 0\rangle$ is [2,3,4,6,7] and the corresponding *relevant-slice condition* 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 16. Since each statement is executed once, we do not distinguish between different execution instances of the same statement in this example.

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

For the execution path of $\langle x == 6, y == 2, z == 2\rangle$, the dynamic backward slice result w.r.t. the slicing criteria at line 16 is [4,15,16] - it contains no branches. The path condition computed over the statements in the dynamic slice (or the dynamic slice condition) is simply the formula $true$.

Different from dynamic backward slicing, relevant slicing also includes the statement instances that could potentially affect the slicing criteria. For example, if evaluating a branch differently could affect the slicing criteria — such a branch is included in the relevant slice, even though it is not contained in the dynamic backward slice. In the example program, the branch at line 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 [4,9,15,16], and the *relevant-slice condition* on it is $\neg(x + y > 10)$. Any input $t$ satisfying the *relevant-slice condition* $\neg(x + y > 10)$ has the same symbolic expression for the output `out`, which in this case turns out be the constant value 2.

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

Just like the DART approach [Godefroid et al. 2005] 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

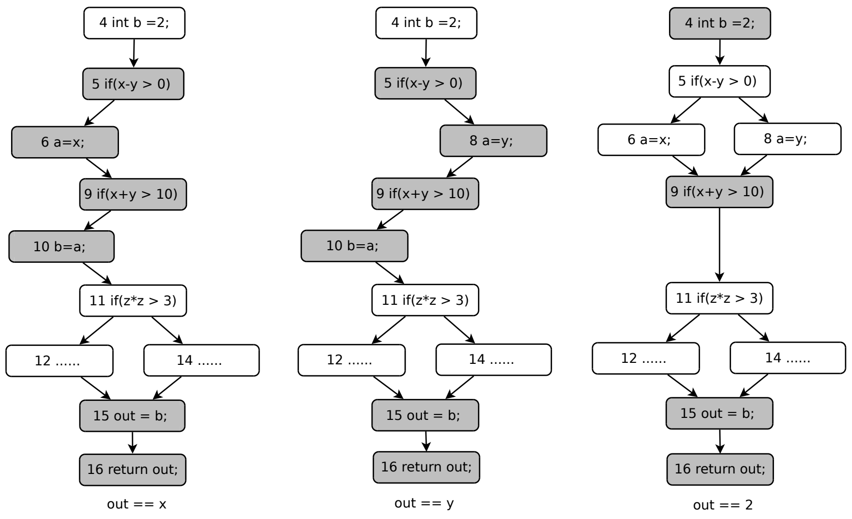


Fig. 4: Path partitions of the example in Figure 1

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

We have developed a path exploration method which avoids the aforementioned problems. While exploring (groups of) paths based on *relevant-slice condition*, our method re-orders the constraints

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

| No. | From | Input | Path | RSC | Path condition |
|---|---|---|---|---|---|
| 1 | | $\langle 6,2,2\rangle$ | $[b1^t, b2^f, b3^t]$ | $\neg(x+y>10)$ | $(x-y>0)\wedge\neg(x+y>10)\wedge(z*z>3)$ |
| 2 | 1.1 | $\langle 6,5,2\rangle$ | $[b1^t, b2^t, b3^t]$ | $(x-y>0)\wedge(x+y>10)$ | $(x-y>0)\wedge(x+y>10)\wedge(z*z>3)$ |
| 3 | 2.2 | $\langle 6,2,2\rangle$ | $[b1^t, b2^f, b3^t]$ | $\neg(x+y>10)$ | $(x-y>0)\wedge\neg(x+y>10)\wedge(z*z>3)$ |
| 4 | 2.1 | $\langle 2,6,2\rangle$ | $[b1^f, b2^f, b3^t]$ | $\neg(x+y>10)$ | $\neg(x-y>0)\wedge\neg(x+y>10)\wedge(z*z>3)$ |

Table II: Path exploration with reordered *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 | $\langle 5,6,2\rangle$ | $[b1^f, b2^t, b3^t]$ | $\neg(x-y>0)\wedge(x+y>10)$ | $(x+y>10)\wedge\neg(x-y>0)$ |

in the *relevant-slice condition*. The path exploration is based on reordered *relevant-slice condition*. A reordered *relevant-slice condition* satisfies the following property (which also holds for path conditions): if $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \psi_i$ is a prefix of a reordered *relevant-slice condition*, the reordered *relevant-slice condition* of any input satisfying $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ has $\psi_1 \wedge \psi_2 \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ as a prefix.

## 3. OUR APPROACH

In this section, we give our path exploration algorithm based on *relevant-slice condition*. We then give theorems on the completeness of our path exploration algorithm. Throughout the paper, we assume that the slicing criteria is in a basic block that post-dominates the entry of the program. More discussion of this assumption is provided at the beginning of Section 3.2.

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 $\pi$ w.r.t. slicing criteria $sc$. We use $rsc(sc, \pi)$ to denote the relevant slice condition on path $\pi$ w.r.t. slicing criteria $sc$. We use $reordered\_rsc(sc, \pi)$ to denote the reordered sequence of $rsc(sc, \pi)$. We use $br(\psi)$ to denote the branch instance of a branch condition $\psi$. We use $bc(b)$ to denote the branch condition generated by $b$. Given a *relevant-slice condition* or reordered *relevant-slice condition* $\theta$ and a branch condition $\psi$, we use $\theta\backslash\psi$ to denote the result of removing $\psi$ from $\theta$. Recall that $\theta$ is a conjunction of branch conditions. If $\psi$ is contained in $\theta$, $\psi$ is deleted from the conjunction to get $\theta\backslash\psi$. Otherwise, $\theta\backslash\psi$ is the same as $\theta$.

### 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 RSCExplore 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.

**Algorithm 1** RSCExplore:path exploration using *relevant-slice condition*

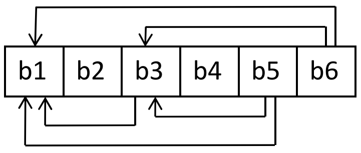
```
Input:
$P$ : The program to test
$t$ : An initial test case for $P$
$C$ : A slicing criterion
Output:
$T$: A test-suite for $P$

$Stack = null$ // The stack of partial rsc to be explored
$Execute(t, 0)$
while $Stack$ is not empty do
    let $\langle f, j \rangle = pop(Stack)$
    if $f$ is satisfiable then
        let $\mu$ be one input that satisfies $f$
        put $\mu$ 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 $|seq| \leq 1$ then
        return $seq$
    end if
    let $seq$ be $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$
    $seq_1 = true$, $seq_2 = true$
    for all i from 1 to k-1 do
        if $br(\psi_i)$ is in relevant slice of $br(\psi_k)$ then
            $seq_1 = seq_1 \wedge \psi_i$
        else
            $seq_2 = seq_2 \wedge \psi_i$
        end if
    end for
    return $reorder(seq_1) \wedge \psi_k \wedge reorder(seq_2)$
end procedure
```

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

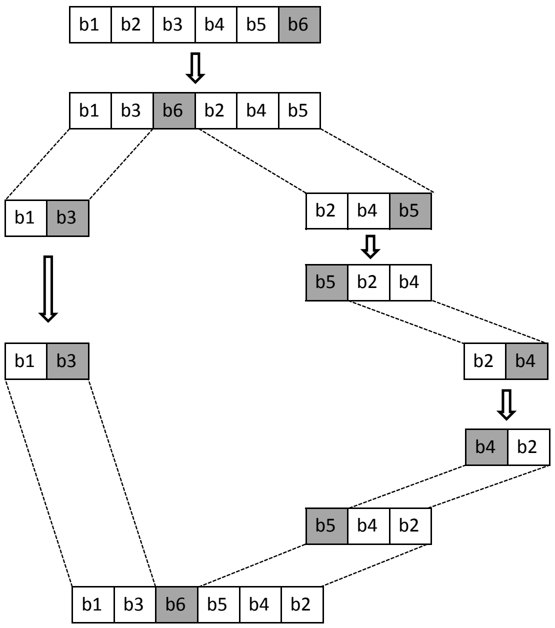
We show the *reorder* procedure in action in Figure 5. Note that our reordering is done on branch conditions in a *relevant-slice condition*. Since there is a unique branch condition for each branch instance in the execution trace, the example in Figure 5 is on branch instances for simplicity. On the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

**Algorithm 2** Augmented reorder

```
1:  procedure $reorder(seq, p)$
2:      let $seq$ be $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1} \wedge \psi_k$
3:      if $|seq| == 1$ then
4:          assign the priority of $b(\psi_1)$ as $p@[b(\psi_1)]$
5:      end if
6:      if $|seq| \leq 1$ then
7:          return $seq$
8:      end if
9:      $seq_1 = true, seq_2 = true$
10:     for all i from 1 to k-1 do
11:         if $b(\psi_i)$ is in relevant slice of $b(\psi_k)$ then
12:             $seq_1 = seq_1 \wedge \psi_i$
13:         else
14:             $seq_2 = seq_2 \wedge \psi_i$
15:         end if
16:     end for
17:     assign the priority of $b(\psi_k)$ as $p@[b(\psi_k)]$
18:     $seq_1' = reorder(seq_1, p@[b(\psi_k)])$
19:     $seq_2' = reorder(seq_2, p)$
20:     return $seq_1' \wedge \psi_k \wedge seq_2'$
21: end procedure
```



(a) Dependencies among branch instances (Arrows denote both direct and indirect dependencies).



(b) Reorder process. Each arrow represents one single reorder step. The "pivot" nodes are highlighted in grey.

Fig. 5: *Reorder* process for relevant slice condition from relevant slice [b1,b2,b3,b4,b5,b6]

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

In Algorithm 1, we use a stack to maintain the to-be-explored partial *relevant-slice conditions*. The main algorithm keeps on processing the formulae in the stack when it is not empty. In each iteration, the algorithm pops out one partial *relevant-slice condition* from the stack, and checks whether it is satisfiable or not. If it is satisfiable, we get a new input $\mu$ by solving the formula. The new input $\mu$ could lead to some unexplored *relevant-slice condition*. The *relevant-slice condition* for the execution trace of input $\mu$ is then explored, as shown by the procedure *Execute* in Algorithm 1. Given the execution trace of $\mu$, 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* $\varphi_i = \psi'_1 \wedge \psi'_2 \wedge \ldots \wedge \psi'_{i-1} \wedge \neg\psi'_i$, $1 \leq i \leq n \leq m$, we know that $\varphi_i$ 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 II. If the "from" column of Table II contains $\alpha.\beta$, it means that the current input is generated by negating the $\beta th$ branch constraint of the reordered *relevant-slice condition* in the $\alpha th$ row. The path exploration based the reordered *relevant-slice condition* explores all possible *relevant-slice conditions* of the program.

We now use the same example program in Figure 1 to explain that our technique is different from employing path condition based path exploration on the static slice of the program. Given a slicing criteria, we could first perform static slicing on the program with respect to the slicing criteria. Since the static slicing result is also a complete program, we can enumerate all paths of the static slice. Applying this approach on the program in Figure 1, the static slice result contains all lines except for lines 11-14. As there are two branches in the static slice, path exploration based on path condition explores all four feasible paths. In contrast, our technique generates only three *relevant-slice conditions* as shown in Table II.

## 3.2. Proofs

*Assumptions.* We assume that the SMT solver used to solve *relevant-slice conditions* is sound and complete (more discussion on this assumption is given in Section 6). 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. This assumption makes sure that the slicing criteria is executed for any program inputs. If a program does not satisfy this assumption, a simple program transformation can produce an equivalent program that meets this assumption. We give one example of transformation in Figure 6. If the program contains *multiple outputs*, the slicing criteria can simply be a set 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 as well as our implementation, 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 [Xin et al. 2008], which is defined as:

DEFINITION 6 (EXECUTION INDEX [XIN ET AL. 2008]). *Given a program $P$, the index of an execution trace $\pi$ of $P$, denoted as $EI^{\pi}$, is a function of execution points in $\pi$, that satisfies: $\forall$ two execution points $x \neq y$, $EI^{\pi}(x) \neq EI^{\pi}(y)$.*

```
int foo(int x){
  if(x > 0){
    return x+1;//slicing criteria
  else
    return x+2;//slicing criteria
}
```

(a) Program before transformation

```
int foo(int x){
  int ret;
  if(x > 0){
    ret = x+1;
  else
    ret = x+2;
  return ret;//slicing criteria
}
```

(b) Program after transformation

Fig. 6: Example of program transformation that makes the slicing criteria post-dominates the program entry

Two statement instances in different paths are the same iff. they have exactly the same "execution index". Given two paths $\pi$ and $\sigma$ and a statement instance $s$ in $\pi$, we say $s$ also appears in $\sigma$ iff. in $\sigma$ there is a statement instance $s'$ such that $EI^{\pi}(s) == EI^{\sigma}(s')$. In its simplest form, we use the path from root to $s$ in the Dynamic Control Dependence Graph of $\pi$ as the execution index of $s$ in $\pi$.

*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 $\pi$, then $u \rightsquigarrow_s b$, iff. (i) there exist a variable $v$ used at $u$, (ii) there is no definition of $v$ 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 do 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 statement(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.* 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 $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1}$, 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 $\psi_k$ 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 $b \not\rightsquigarrow b_y$. (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) $k < min(\alpha, \beta)$, $b_x^{k+1} \not\rightsquigarrow_c b_y^{k+1} \wedge b_y^{k+1} \not\rightsquigarrow_c b_x^{k+1}$ and $b_x^{k+1}$ is after $b_y^{k+1}$ in time order. By this definition, it is impossible to have both $sp(b_x) > sp(b_y)$ and $sp(b_y) > sp(b_x)$.

*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 7. For the ease of understanding, we give the outline of our proofs and the relations between lemmas and theorems in the following.

In Theorem 3.3, 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 an expression in terms of the program inputs. Let $s$ be a statement instance in the path of input $t$, and $v$ be a variable used in $s$. The symbolic value of $v$ in $s$ is 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 3.3, 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 $t' \models rsc(s, \pi(t))$, then the relevant slice w.r.t. $s$ in $\pi(t')$ would be exactly the same as that in $\pi(t)$. Theorem 3.3 could be easily derived from

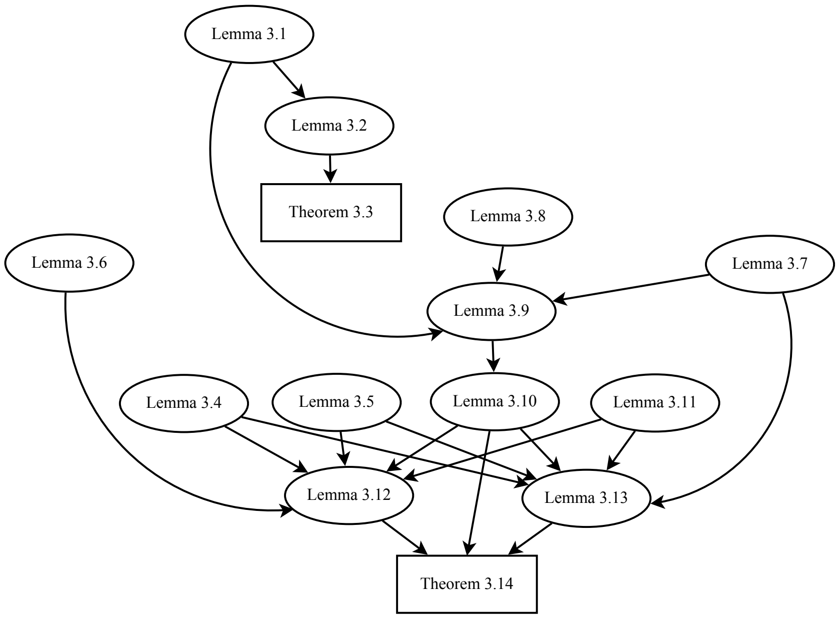


Fig. 7: Structure of the proofs

Lemma 3.2.

In Theorem 3.14, given any feasible path $\pi$, we show that our path exploration algorithm would explore a path $\pi'$ that shares the same *relevant-slice condition* with $\pi$. 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 $\pi$ 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 $\varphi$ to get a new input $t'$, we need to prove that the *relevant-slice condition* on $\pi(t')$ still contains $\varphi$ 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 $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$, 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 $\pi$ (having longer common prefix with $reordered\_rsc(C, \pi)$ ) by negating the $k^{th}$ branch condition in $f$. All the lemmas from Lemma 3.4 to Lemma 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 $\pi(t')$. $\square$

LEMMA 3.2. *Let $t$ and $t'$ be two inputs and $s$ be a statement instance in $\pi(t)$. If $t' \models rsc(s, \pi(t))$, 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 $t' \models rsc(s_i, \pi(t))$, $t' \models rsc(s_j, \pi(t))$. 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 $s_i \not\rightarrow_c s_u$ in $\pi(t')$ since $s_u$ is evaluated differently in $\pi(t)$ and $\pi(t')$. Similarly, $s_i \not\rightarrow_c s_u$ 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 $s_i \to s_k$ in $\pi(t)$. According to induction hypothesis, 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, $s_i \to s_k \rightsquigarrow s'$ 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 $s_i \to s_j \rightsquigarrow s'$ 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 $s_i \to s_j$ in $\pi(t')$. According to the dependence type of $s_i \to s_j$, we have the following two cases.

— (a) $s_i \to_c s_j$ in $\pi(t)$. The existence of both $s_i$ and $s_j$ already implies that $s_i \to_c s_j$ in $\pi(t')$.
— (b) $s_i \to_d s_j$ or $s_i \to_p s_j$ in $\pi(t)$. We need to prove that the dependence between $s_i$ and $s_j$ still appears in $\pi(t')$. Suppose $s_i \to_d s_j$ or $s_i \to_p s_j$ in $\pi(t)$ is caused by the use of variable $v$ at $s_i$. We prove $s_i \to_d s_j$ or $s_i \to_p s_j$ 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 $s_i \to_d s_n$ 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 $s_i \to_d s_n$ in $\pi(t)$. This contradicts that $s_i \to_d s_j$ or $s_i \to_p s_j$ in $\pi(t)$ (simply by the definition of dynamic data dependencies and potential dependencies). So if $s_i \to_d s_j$ or $s_i \to_p s_j$ is in $\pi(t)$, $s_i \to_d s_j$ or $s_i \to_p s_j$ 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 $\alpha$ 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 $s_x^i$, $1 \le x \le \alpha$, 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 $1 \le x \le \alpha$. 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 $t' \models rsc(s_i, \pi(t))$. 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 3.3. *If the relevant-slice conditions of two paths $\pi_1$ and $\pi_2$ w.r.t. $C$ are the same, then the variables used in the slicing criteria $C$ have the same symbolic values in $\pi_1$ and $\pi_2$.*

PROOF. Let $t_1$ and $t_2$ be two test inputs whose execution traces are $\pi_1$ and $\pi_2$ respectively. According to the theorem statement, we have $rsc(C, \pi_1) == rsc(C, \pi_2)$. Since $t_2 \models rsc(C, \pi_2)$, $t_2 \models rsc(C, \pi_1)$. 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 $\pi_1$ and $\pi_2$. This completes the proof. □

LEMMA 3.4. *Let $t$ and $t'$ be two inputs, given a branch instance $b$ in $\pi(t)$, if $t' \models rsc(b, \pi(t)) \backslash bc(b)$, $b$ 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 $i$, $1 \le i \le k$, $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \Rightarrow rsc(b(\psi_i), \pi(t)) \backslash \psi_i$.*

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 the shortened priority sequences $sp(b_x) > sp(b_y)$.*

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 $p(b_x)$. 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 there is at least one branch $b$ that is between $postdom(b_x^{k+1})$ and $b_x^k$ such that $b$ is in $sp(b_x)$. this contradicts that the $(k+1)th$ element in $sp(b_x)$ is $b_x^{k+1}$. (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 $\hat{b}_k^u \not\rightsquigarrow b$. 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 $\hat{b}_k^u \not\rightsquigarrow b$. 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 $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_k$, $k \leq i$, 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 $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, then (i) For any $j$, $1 \leq j \leq i$, $b_k^j$ also appears in $\pi(t')$. (ii) For any $j$, $1 \leq j < i$, 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 $j$, $1 \leq j < i$, 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 $j$, $1 \leq j \leq i$, $b_k^j$ also appears in $\pi(t')$.** Suppose $b_k^j \rightarrow_c b$. Since $b_k^j \rightarrow_c b$, we have $b_k^j \rightsquigarrow b$ 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 $\psi_k$) in $reordered\_rsc(C, \pi(t))$. This means that the branch condition of $b$ is actually one of the $\psi_m$, where $m \leq k-1$. Since $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, 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 $j$, $1 \leq j < i$, 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 $s \in rs(b_k^j, \pi(t'))$ 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 $s1 \not\rightarrow s2$ 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 $\psi_k$) in $reordered\_rsc(C, \pi(t))$. Since $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, 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 $s1 \not\rightarrow s2$ 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, $s1 \not\rightarrow s2$ 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 $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, 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 $j$, $1 \leq j < i$, 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 $\pi(t')$. For any branch instance $b_c$ such that $s \rightsquigarrow_c b_c$, 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 $\psi_k$) in $reordered\_rsc(C, \pi(t))$. Since $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, 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 $s1 \not\rightarrow s2$ 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, $s1 \not\rightarrow s2$ 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 $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{k-1}$, 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 also $[b_k^1, b_k^2, \ldots, b_k^i]$ in $\pi(t')$ in the following steps:(i) We prove that $b_k^j \rightsquigarrow b_k^{j+1}$ for each $j$, $1 \leq j < i$, 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 control 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 $1 \leq j < i$. Since $b_k^j \rightsquigarrow b_k^{j+1}$ and $b_k^j \not\rightsquigarrow_c b_k^{j+1}$ 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 in Lemma 3.9, between $postdom(b_k^{j+1})$ and $b_k^j$, $rs(b_k^j, \pi(t'))$ is exactly the same as $rs(b_k^j, \pi(t))$. 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 $\psi_k$ 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" $b$ (including $b_k^j$) between $b_k^{j-1}$ and $b_k^j$, where $b_k^{j-1} \rightsquigarrow b$ and $b \not\rightsquigarrow b_k^j$ and $b \rightsquigarrow b_k$ in $\pi(t')$. 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 $b \not\rightsquigarrow b_k^j$. (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 $b \not\rightsquigarrow b_k^j$ in $\pi(t)$, $b \not\rightsquigarrow b_k^j$ in $\pi(t')$ either. This contradicts that $b \rightsquigarrow b_k^j$ in $\pi(t')$. In each case, we get 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 any given branch instance in $p(b_k)$ in $\pi(t')$, either this branch instance is transitively control 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 $postdom(b_k^j)$, $1 < j \leq i$, 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 $[b_k^1, b_k^2, \ldots, b_k^i]$. $\square$

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 $\pi_1$ and $\pi_2$ 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 $\pi_1$ 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 $\varphi_k$) 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 $\pi_1$. Similarly we have $sp(b_y) > sp(b_x)$ from $\pi_2$. 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 $\pi_1$ and $\pi_2$. So $\varphi_k$ could only be different from $\psi_k$ if the branch $b(\varphi_k)$ and $b(\psi_k)$ are evaluated to different directions in $\pi_1$ and $\pi_2$. So we have $\varphi_k == \neg\psi_k$. □

LEMMA 3.13. *Let t and t′ be two inputs. If* $t' \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$, *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 $\pi(t')$.
— The first $i$ branch conditions of $reodered\_rsc(C, \pi(t))$ must be from the branch instances $\{b(\psi_k) | 1 \leq k \leq i\}$.

**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 $\psi_j$ and $\psi_k$ 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 $\psi_j$ is before $\psi_k$, 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 each branch instance is restricted by the corresponding branch condition in $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$.

**The first $i$ branch conditions of $reodered\_rsc(C, \pi(t))$ must be from the branch instances $\{b(\psi_k) | 1 \leq k \leq i\}$.** Assume to the contrary that this is not the case. Let $\varphi$ be the first branch condition whose branch instance is not in $\{b(\psi_k) | 1 \leq k \leq i\}$ and $\varphi$ is one of the first $i$ branch conditions in $reordered\_rsc(C, \pi(t'))$. According to the above proof, all the branch conditions before $\varphi$ are satisfied by $t$. Therefore, according to Lemma 3.10, $b(\varphi)$ appears in $\pi(t)$. Since $sp(\varphi) > sp(\psi_i)$, this contradicts that $b(\varphi)$ does not appear in $\{b(\psi_k) | 1 \leq k \leq i\}$.

According to the above proved properties, $reordered\_rsc(C, \pi(t'))$ must contain $\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i$ as a prefix. □

We now prove the completeness of our path search method.

THEOREM 3.14. *Given a program $P$ and an execution trace $\pi(t)$ for input $t$ in $P$, Algorithm 1 must explore an execution trace $\pi(t')$ for some input $t'$ such that $\pi(t)$ and $\pi(t')$ share the same relevant-slice condition (irrespective of the initial test input with which Algorithm 1 is started) — provided the total number of relevant-slice conditions in $P$ is bounded.*

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 first show that when $k == min(i,j)$, it must be that $f == g$. Without losing generality, let us assume to the contrary that $f \neq g$ and $k == j$, which means that $i > j$. If an input $t_f$ satisfies $f$, then $t_f \models \varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_{j-1} \wedge \varphi_j$, which is the same as $t_f \models \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{j-1} \wedge \psi_j$. According to the proof of Lemma 3.10, $b(\psi_{j+1})$ must appear in the trace $\pi(t_f)$, which contradicts that the first $k$ branch conditions in $f$ and $g$ are the same and the length of $f$ is only $k$.

We now define the distance on reordered *relevant-slice conditions*. Given two reordered *relevant-slice conditions* $f$ and $g$, 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 *relevant-slice conditions* is bounded in program $P$.

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.

Our implementation is based on JSlice[Wang and Roychoudhury 2008][1]. JSlice is an open-source dynamic slicing tool working on Java bytecodes. We have extended JSlice to compute *relevant-slice*

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

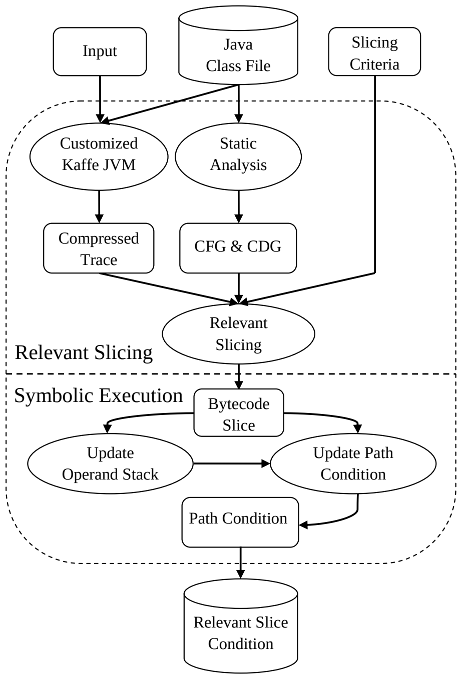


Fig. 8: Architecture of *relevant-slice condition* computation

*conditions*. The architecture of our extended JSlice is shown in Figure 8. The "Operand Stack" in Figure 8 stands for the stack of operands in a method activation frame. The recorded execution trace of JSlice does not contain operand stack information. This is a design choice for keeping the trace compact. During slicing, we need to recover the execution stack from a recorded trace to derive operation on stack variables. For example, given an `iload` instruction (loading integer to the top of current stack), we need the operand stack to know which stack variable is modified. Different from stack variables, operations of heap variables are directly recorded in execution trace.

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

In Figure 8, 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 whose definitions have not been encountered and need to be explained later in the backward symbolic execution process. The symbolic value of a variable $v$ is explained by either an assignment to $v$ or by program input to $v$. Let us take the sample program in Figure 1 to show our backward symbolic execution on a relevant slice. Note that although we show this example at the source code level, our implementation is at the Java bytecode level. Suppose the input is $\langle x == 6, y == 5, z == 2\rangle$. The relevant slice for the execution trace of this input is $[5, 6, 9, 10, 15, 16]$. Backward symbolic execution along this relevant slice is shown in Table III. The set of to-be-explained variables are shown in the third column of

Table III: Backward symbolic execution example

| Relevant slice | Symbolic values | To be explained variables | Relevant slice condition |
|---|---|---|---|
| `16 return out;` | $\{\}$ | $\{ out \}$ | true |
| `15 out = b;` | $\{ out \to b \}$ | $\{ b \}$ | true |
| `10 b = a;` | $\{out \to a, b \to a \}$ | $\{ a \}$ | true |
| `9 if(x+y > 10)` | $\{out \to a, b \to a \}$ | $\{a, x, y \}$ | $x + y > 10$ |
| `6 a = x;` | $\{out \to x, b \to x, a \to x \}$ | $\{ x, y \}$ | $x + y > 10$ |
| `5 if(x-y >0)` | $\{out \to x, b \to x, a \to x \}$ | $\{ x, y \}$ | $x - y > 0 \wedge x + y > 10$ |

Table III.

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

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

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

Our execution engine is a combined infra-structure for dynamic dependency analysis and dynamic symbolic execution. Thus, apart from computing *relevant-slice conditions*, we can simply disable the dependency analysis in our engine to compute path conditions. The path conditions and *relevant-slice conditions* generated from our tool are in the SMT-LIB format[2], which can be solved by various Satisfiability Modulo Theory or SMT solvers. In our implementation, we choose Z3 [De Moura and Bjørner 2008][3] as the SMT solver for our tool.

## 5. EXPERIMENTS

In the following, we first compare our *relevant-slice condition* based path exploration method with *path condition* based path exploration. We then present three applications of *relevant-slice condi-*

[2]`http://www.smt-lib.org/`

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

Table IV: Experiments in full program exploration

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

*tions* in: i) the debugging of evolving programs and ii) test-suite augmentation and iii) mining finite state automata from programs.

**Algorithm 3** PCExplore:path exploration using path condition

```
1: Input:
2: $P$ : The program to test
3: $t$ : An initial test case for $P$
4: Output:
5: $T$: A test-suite for $P$
6:
7: $Stack = null$ // The stack of partial PC to be explored
8: $Execute(t, 0)$
9: while $Stack$ is not empty do
10:     let $\langle f, j \rangle = pop(Stack)$
11:     if $f$ is satisfiable then
12:         let $\mu$ be one input that satisfies $f$
13:         put $\mu$ into $T$
14:         $Execute(\mu, j)$
15:     end if
16: end while
17: return $T$
18:
19: procedure $Execute(t, n)$
20:     execute $t$ in $P$ and compute path condition $pc$
21:     let $pc = \psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{m-1} \wedge \psi_m$
22:     for all i from n+1 to m do
23:         let $h = (\psi_1 \wedge \psi_2 \wedge \ldots \wedge \psi_{i-1} \wedge \neg\psi_i)$
24:         push $\langle h, i \rangle$ into $Stack$
25:     end for
26:     return
27: end procedure
```

### 5.1. Path exploration

We compare our path exploration algorithm RSCExplore in Algorithm 1 with the PCExplore shown in Algorithm 3. The PCExplore algorithm closely resembles our RSCExplore algorithm in Algorithm 1. The main difference is that PCExplore uses path condition instead of *relevant-slice condition*. Because of using path condition, neither slicing nor reordering takes place in PCExplore.

The subject programs shown in Table IV are from SIR [Do et al. 2005] repository. Tcas and Schedule are originally written in C language, we manually translate them into Java language. For Tcas and Schedule, the slicing criteria are set as the final program outputs. For the other three data-structure programs, the slicing criteria are set as the outputs of the test drivers. The lines of code (LOC) in each program are also shown in Table IV.

The completeness of exploration is difficult to achieve in practice for several reasons. Two of the main reasons are (i) the limited power of current SMT solvers and (ii) imprecise modeling of program semantics for symbolic execution. Because of these two reasons, our technique may miss

```
1  int foo(int x, int y){ //input
2    int out; //output
3    int a[2] = {0,1};
4    if(x > 0)
5      System.out.println("x is greater than zero");
6    if(a[x]>0){
7      if(y > 0)
8        out = 1;
9     else
10       out = -1;
11   }else{
12     out = 0;
13   }
14   return out;//slicing criteria
15 }
```

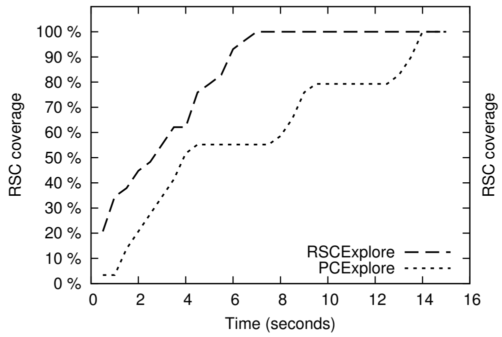
Fig. 9: Example of imprecise array modelling

a certain *relevant-slice condition* $rsc_i$ when PCExplore can explore a path whose *relevant-slice condition* is $rsc_i$. More discussion of the incompleteness of SMT solvers is provided in Section 6. In the next paragraph, we explain how imprecise modeling of array can cause our implementation to be not as complete as PCExplore in terms of *relevant-slice condition* coverage.

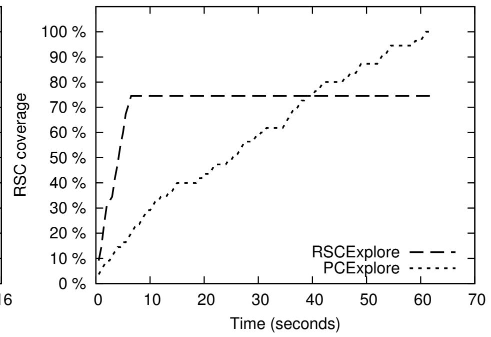
Consider the example program in Figure 9. The branch in line 6 uses `a[x]` in its branch condition. In our current implementation, we concretize symbolic array index using the value observed at execution time. Suppose the initial input for the program in Figure 9 in both our method RSCExplore and PCExplore is $\langle x == 0, y == 0\rangle$. When computing the branch condition of line 6 for input $\langle x == 0, y == 0\rangle$, we concretize the symbolic value of `x` using 0, which is the value of `x` in line 6 when executing the program with input $\langle x == 0, y == 0\rangle$. After concretization, the branch condition of line 6 is $\neg(a[0] > 0)$, which is reduced to $true$. Thus, due to the concretization of symbolic array index, the branch at line 6 cannot contribute a branch condition to either path condition or *relevant-slice condition*. The path condition and *relevant-slice condition* for $\langle x == 0, y == 0\rangle$ are $\neg(x > 0)$ and $true$ respectively. Since the *relevant-slice condition* for the initial input $\langle x == 0, y == 0\rangle$ is $true$ (containing no branch condition), our technique terminates. However, some *relevant-slice conditions* are missed by our technique. In particular, the *relevant-slice conditions* of paths that evaluate branch at line 6 to false are missed. In contrast, PCExplore could explore all feasible paths (hence all *relevant-slice conditions*) of the program in Figure 9. If arrays are modelled precisely, this problem will disappear.

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

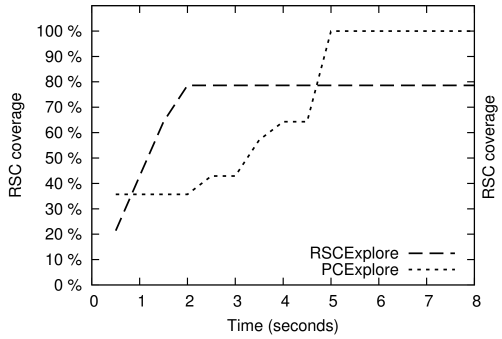
In columns 4-11 of Table IV, we compare the time, number of generated test cases, formula size and number of solver calls between our method RSCExplore and PCExplore. The formula size is measured by the number of bytes in the SMT-LIB formula file. For getting these numbers, both our method (RSC) and the PCExplore method are *run to completion*, and the running time is recorded. Note that the time reported in Table IV includes the time taken in every steps of our method and



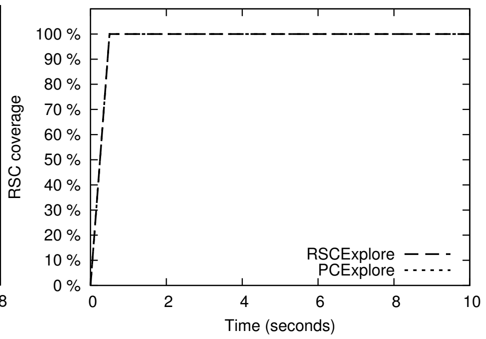
(a) Tcas



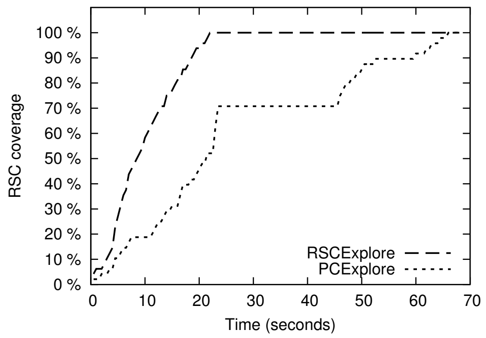
(b) BinarySearchTree



(c) OrdSet



(d) Schedule



(e) DisjointSet

Fig. 10: Relevant-slice condition coverage comparison

PCExplore. For example, the time taken by our method includes the time for program execution, relevant slicing, *relevant-slice condition* computation, branch condition reordering, formula solving, etc. As shown in Table IV, our technique takes much less time than PCExplore. 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 PCExplore. 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.

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

| Subject prog. | Stable version | Buggy version | Diff | Time | | Debugging results | |
|---|---|---|---|---|---|---|---|
| | | | | PC | RSC | PC | RSC |
| JLex | 1.2.1 (7290 LOC) | 1.1.1 (6984 LOC) | 518 LOC | 543 min | 15 min | 50 LOC | 3 LOC |
| JTopas | 0.8 (4514 LOC) | 0.7 (5754 LOC) | 2489 LOC | 81 min | 5 min | 4 LOC | 4 LOC |
| NanoXML | 2.1(4947 LOC) | 2.2 (5244 LOC) | 2496 LOC | 2m56s | 43s | 8 LOC | 6 LOC |

Figure 10 compares the *relevant-slice condition* coverage of our Algorithm 1 with PCExplore *under the same time limit*. Note that PCExplore intends to achieve path coverage. However, as we have observed - several paths may have the same input-output relationship, and testing is always done by checking outputs. We check the number of *relevant-slice conditions* that are covered by the paths explored in PCExplore. As shown in Figure 10, our technique gets higher *relevant-slice condition* coverage then PCExplore when the given time is short.

## 5.2. Debugging of evolving programs

The obvious application of *relevant-slice conditions* is in software testing - it groups program paths and can be used to efficiently 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 [Qi et al. 2009]. Given two program versions $P$ and $P'$, and a test case $t$ which passes in $P$ but fails in $P'$, the work in [Qi et al. 2009] 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 $\psi_i$ are primitive constraints. The following $m$ formulae $\{\varphi_i \mid 0 \leq i < m\}$ are then solved where $\varphi_i \overset{def}{=} f \wedge \psi_1 \wedge \ldots \psi_i \wedge \neg\psi_{i+1}$. We invoke a Satisfiability Modulo Theory or SMT solver to solve the $m$ formulae $\{\varphi_i \mid 0 \leq i < m\}$. Finally, for every $\varphi_i$ which is satisfiable, we can find a single line in the source code which is a potential error root cause — the branch corresponding to $\psi_{i+1}$ (which is negated in $\varphi_i$).

We observe that the path conditions $f$ and $f'$ in the above method can be replaced by *relevant-slice conditions*. 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 to the branch in line 6 is unrelated to the observable error (unexpected value of `x`). Indeed, line 6 is not in the dynamic slice or relevant slice of the slicing criterion corresponding to the output value of `x` in line 10.

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

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

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

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

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

We also conducted experiments using `JTopas` [5] as the subject program. `JTopas` is a Java library for parsing arbitrary text data. We use version 0.8 of `JTopas` as the stable version, and version 0.7 as the buggy version. There are 5754 and 4514 lines of code in version 0.7 and version 0.8 respectively. `JTopas` allows users to customize whitespace characters (i.e. characters that are considered as whitespace characters) by using function *setWhitespaces*. `JTopas` also uses a boolean field *_defaultWhitespaces* to control whether the default whitespace characters are used or the user-customized whitespace characters are used. To use the customized whitespace characters, *_defaultWhitespaces* has to be set to *false*. Unfortunately, the buggy `JTopas-0.7` does not reset the member *_defaultWhitespaces* leading to the default whitespace characters still being used instead of the customized ones although the user has specified the custom whitespace characters. In our experiment, we customize whitespace characters to {' ', '\r','\t'} ({' ', '\n', '\r','\t'} by default) and use an input file manifesting the aforementioned bug. The debugging results of DARWIN using path condition and DARWIN using *relevant-slice condition* are shown in Table V. The results from the original DARWIN method (using path condition) and DARWIN using *relevant-slice condition* are both four lines of code. They both contain the location where '\n' is treated differently between the two versions. The pinpointed location shows that the stable version does

[4]`http://www.cs.princeton.edu/~appel/modern/java/JLex/`
[5]`http://jtopas.sourceforge.net/jtopas/index.html`

```
public int getProperty(String key,
                       int defaultValue)
{
  String val=(String)attributes.get(key);
  if (val == null) {
    return defaultValue;
  } else {
    try {
      return parseInt(val);
    } catch (NumberFormatException e) {
       throw invalidValue(key, val);
    }
  }
}
```

(a) getProperty in NanoXML-2.1

```
public int getProperty(String name,
                       int defaultValue)
{
  return getIntAttribute(name, defaultValue);
}
public int getIntAttribute(String name,
                       int defaultValue)
{
  String value=(String)attributes.get(name);
  if (value == null) {
    return defaultValue;
  } else {
    try {
      return parseInt(name); //bug
    } catch (NumberFormatException e) {
      throw invalidValue(name, value);
    }
  }
}
```

(b) getProperty in NanoXML-2.2

Fig. 11: Regression bug in NanoXML-2.2

not consider '\n' as a whitespace. In contrast, the buggy version still treats '\n' as a whitespace because *_defaultWhitespaces* is *true* (even though whitespace characters have already been customized). From this clue, the programmer could easily infer that the member *_defaultWhitespaces* was not assigned to the correct value. Although using *relevant-slice condition* does not eliminate any false positives in the debugging result, it does reduce the time taken by DARWIN from 81 minutes to 5 minutes.

Lastly, we applied DARWIN technique on a regression bug in NanoXML [6]. NanoXML is a simple XML file parser. A regression bug happened when NanoXML was changed from version 2.1 to version 2.2. The simplified source code of the bug is shown in Figure 11. Given a property name of an XML element as specified in the parameter, the `getProperty` method is used to get the integer-typed value of the property. The implementation of `getProperty` method was changed from version 2.1 to 2.2. In particular, in version 2.2, the code was restructured and `getProperty` method was implemented by simply calling another method `getIntAttribute`. Unfortunately, the implementation of method `getIntAttribute` contains a bug. The bug lies in the second return statement in Figure 11b. Instead of `return parseInt(name)`, it should be `return parseInt(value)`. We applied our DARWIN debugging technique with version 2.1 as the reference version and version 2.2 as the buggy version. Version 2.1 and version 2.2 of NanoXML have 4947 and 5244 lines of code respectively, and there are 2496 different lines of code between these two versions. The original path-condition-based DARWIN technique took 2 minutes and 56 seconds to generate a bug report with 8 lines of source code. In contrast, DARWIN technique using *relevant-slice condition* only took 43 seconds to generate a bug report with 6 lines. Both the debugging time and the debugging result were improved.

## 5.3. Test-suite augmentation

Test-suite needs to be augmented when old test-suite no longer meets the test requirement due to program changes [Qi et al. 2010; Santelices et al. 2008; Xu et al. 2010]. Suppose a program $P$ is changed to program $P'$. We can employ PCExplore to generate a set of change-exposing test cases to augment the existing test suite. More specifically, we can first apply PCExplore to get signatures of program $P$ and $P'$ separately. Suppose the signature for program $P$ is $Sig(P)$, which is a set of path condition and symbolic output value pairs. For each path $\pi$ explored by PCExplore, we add $\langle pc(\pi), symout(\pi)\rangle$ into $Sig(P)$, where $pc(\pi)$ denotes the path condition along

[6] `http://devkix.com/nanoxml.php`

path $\pi$ and $symout(\pi)$ denotes the symbolic value of output computed on $\pi$. Similarly, we compute the signature for program $P'$ as $Sig(P')$. We could try to generate a test case for each formula $f(\pi_i, \pi_j) \overset{\text{def}}{=} (pc(\pi_i) \wedge pc(\pi_j) \wedge symout(\pi_i) \neq symout(\pi_j))$, where $\langle pc(\pi_i), symout(\pi_i)\rangle \in Sig(P)$ and $\langle pc(\pi_j), symout(\pi_j)\rangle \in Sig(P')$. If $f(\pi_i, \pi_j)$ is satisfiable, its solution is guaranteed to have different output in program $P$ and $P'$ according to the definition of $f(\pi_i, \pi_j)$. We then enumerate all possibilities values of $i$ and $j$ which make $f(\pi_i, \pi_j)$ satisfiable. Whenever $f(\pi_i, \pi_j)$ is satisfiable, its solution is put into a set $T_{\text{PCExplore}}$. Through this process, we get a set of test cases in $T_{\text{PCExplore}}$. These test cases in $T_{\text{PCExplore}}$ expose the semantic changes between the two programs.

As we have seen in this paper, we could use the *relevant-slice condition* to efficiently generate a more concise signature for a program. When using *relevant-slice condition* to generate the signature, each element in $Sig(P)$ becomes $\langle rsc(\pi), symout(\pi)\rangle$, where $rsc(\pi)$ is the relevant slice condition along path $\pi$. We also need to change the definition of $f(\pi_i, \pi_j)$ accordingly. For signatures generated using *relevant-slice condition*, we define $f(\pi_i, \pi_j)$ as $f(\pi_i, \pi_j) \overset{\text{def}}{=} (rsc(\pi_i) \wedge rsc(\pi_j) \wedge symout(\pi_i) \neq symout(\pi_j))$. By solving all the possible instances of $f(\pi_i, \pi_j)$, we get a set of test cases $T_{\text{RSCExplore}}$. As path exploration based on *relevant-slice condition* does not lose any precision when generating the signature, $T_{\text{RSCExplore}}$ has the same change-exposing ability than $T_{\text{PCExplore}}$. On the other hand, path exploration based on *relevant-slice condition* is much more efficient than PCExplore, which makes computing $T_{\text{RSCExplore}}$ much less costly than computing $T_{\text{PCExplore}}$.

```
1 int foo(int x, int y, int z){
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    System.out.println("square(z)>3");
13  else
14    System.out.println("square(z)<=3");
15  out = b;
16  return out; //slicing criteria
17}
```

(a) Original Program

```
1 int foo(int x, int y, int z){
2   int out; // output variable
3   int a;
4   int b = 2;
5   if(x - y > 2) //b1, changed
6     a = x;
7   else
8     a = y;
9   if(x + y > 10) //b2
10    b = a;
11  if(z*z > 3)  //b3
12    System.out.println("square(z)>3");
13  else
14    System.out.println("square(z)<=3");
15  out = b;
16  return out; //slicing criteria
17}
```

(b) Changed Program

Fig. 12: Sample program

We now show the process of computing $T_{\text{RSCExplore}}$ in action using the programs in Figure 12 as an example. The program in Figure 12a is the same as the program in Figure 1. Figure 12b contains a changed version of the program in Figure 12a. The branch at line 5 is changed from `if(x-y>0)` to `if(x-y >2)`. As mentioned in Section 1, the program in Figure 12a has the following signature

— $(x - y > 0) \wedge (x + y > 10) \Rightarrow \mathit{out} == x$
— $(x - y \leq 0) \wedge (x + y > 10) \Rightarrow \mathit{out} == y$
— $(x + y \leq 10) \Rightarrow \mathit{out} == 2$

The signature of the changed program in Figure 12b is as follows,

— $(x - y > 2) \wedge (x + y > 10) \Rightarrow \mathit{out} == x$
— $(x - y \leq 2) \wedge (x + y > 10) \Rightarrow \mathit{out} == y$
— $(x + y \leq 10) \Rightarrow \mathit{out} == 2$

Following the aforementioned process of generating $T_{\text{RSCExplore}}$, after removing unsatisfiable formulae, only the following formula is satisfiable

$$((x - y > 0) \wedge (x + y > 10)) \wedge ((x - y \leq 2) \wedge (x + y > 10)) \wedge (x \neq y)$$

By generating a test input from this formula, we get $T_{\text{RSCExplore}}$ as $\{\langle x == 6, y == 5\rangle\}$ (The input variable $z$ is not bounded and can be any integer value). Programs $P$ in Figure 12a and $P'$ in Figure 12b produce different output when given this input.

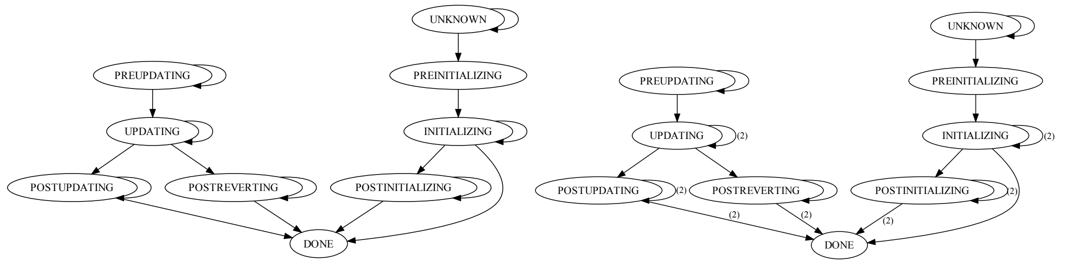
We compared the change-exposing ability of $T_{\text{PCExplore}}$ and $T_{\text{RSCExplore}}$ using `tcas` from SIR as the benchmark. There are 41 versions of `tcas` with seeded bugs. Each one of these 41 versions has only one change from the original program. Among them, two versions always crash with array out of bound exceptions after being translated from C to Java. These two versions are not considered in our experiment. For each version $i$, we use the original program as $P$ and version $i$ as $P'$. Then we compute the two set of test cases $T_{\text{PCExplore}}$ and $T_{\text{RSCExplore}}$ based on $P$ and $P'$. We compare the change-exposing ability of $T_{\text{PCExplore}}$ and $T_{\text{RSCExplore}}$. We observe that whenever the number of change-exposing inputs in $T_{\text{PCExplore}}$ is not zero, the number of change-exposing inputs in $T_{\text{RSCExplore}}$ is not zero. This shows that using *relevant-slice condition* to explore the program does not change the ability of generating change-exposing inputs. On the other hand, redundant path exploration is avoided, which improves the efficiency of the approach. Overall, the time taken to generate $T_{\text{RSCExplore}}$ is 20.8% of the time taken to generate $T_{\text{PCExplore}}$ in the experiment with `tcas`.

## 5.4. Mining finite state automata from programs

In this section, we show an application of our RSCExplore algorithm in mining FSA (Finite-State Automata) from real programs. An FSA contains finite number of states and transitions. At any point of time, an FSA is only in one state. Each transition connects its source state to its destination state. A transition is only activated when the FSA is in its source state and its transition condition is satisfied. FSA is frequently used to model control systems and event-driven systems. Uncovering FSA from real implementation of such systems could help better understand these systems. Programmers can also check whether the mined FSA matches the system specification to make sure that the specification is correctly implemented.

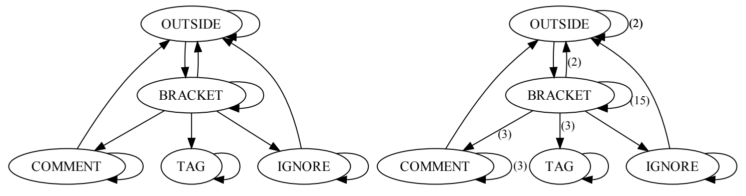
Uncovering FSA from real program implementation amounts to uncovering all states, transitions and transition conditions associated with transitions. We first need to identify the variables that represent the FSA state in the program. Those variables are then set as the slicing criteria in our path exploration algorithm. Recall from Section 5.3 that we can use our path exploration algorithm based on *relevant-slice condition* to produce a signature of the explored program. Given a program $P$, our RSCExplore algorithm generates a signature $Sig(P)$ for $P$. Each entry in the signature $Sig(P)$ is in the form of $\langle rsc(\pi), symout(\pi)\rangle$. Suppose variable $s$ in the program contains the FSA state. By setting $s$ as the slicing criteria, $symout(\pi)$ is the symbolic value of $s$ which tells us how the FSA state $s$ changes. In addition, the *relevant-slice condition* $rsc(\pi)$ represents the transition condition that has to be satisfied for the FSA state to be $symout(\pi)$. In other words, if the program state (including variables besides the FSA state) satisfies $rsc(\pi)$, the FSA state will change to $symout(\pi)$. By setting the slicing criteria as the FSA state, our technique only focuses on the computation that affects the FSA state. Computation unrelated to FSA state is immediately ruled out during slicing. This not only saves effort in path exploration but also helps derive more precise FSA models. In contrast, we can also use path exploration based on path condition to generate a signature $Sig^{\text{PC}}(P)$ for $P$. Each entry in $Sig^{\text{PC}}(P)$ is in the form of $\langle pc(\pi), symout(\pi)\rangle$. The path condition $pc(\pi)$ can be used as the transition condition and $symout(\pi)$ represents how the state changes. However, computation unrelated to the FSA state also takes place in the path condition, making path condition unnecessarily complicated. Moreover, more than one path containing the same state transition is explored, leading to wastage of time and computation power.

We have applied the above technique onto two real-life programs: CTAS (Center TRACON Automation System) and an XML type parser used in SQL Power Architect [Software 2012].



(a) FSA of real implementation (b) FSA uncovered by our method

Fig. 13: FSAs for CTAS Air Traffic Controller



(a) FSA of real implementation (b) FSA uncovered by our method

Fig. 14: FSAs for XML type parser

CTAS is an air traffic control system from NASA whose weather control logic specification is publicly available [cta ]. Each CTAS system consists of one Weather Control Panel(WCP), one Air-Traffic Controller(ATC) and multiple clients. Both WCP and clients only communicate with ATC. The major task of ATC is to get weather information from WCP and update the weather information of all clients. We focus on the FSA used in the ATC. The ATC implements an FSA with 9 states and 17 transitions. The 9 states are `UNKNOWN`, `PREINITIALIZING`, `INITIALIZING`, `POSTINITIALIZING`, `PREUPDATING`, `UPDATING`, `POSTUPDATING`, `POSTREVERTING` and `DONE`. Initially, the ATC is in the `UNKNOWN` state. Initialization of a newly connected client puts the ATC into `PREINITIALIZING` state. The ATC will then send load-weather instructions to the client and transit to `INITIALIZING` state, waiting for response from the client. If the client has successfully loaded the weather, the ATC will get into `POSTINITIALIZING` state, waiting the response from the client of successfully using the weather information. If the response comes, the ATC will go to `DONE` state. The ATC may also update all clients with new weather information. This process starts with the `PREUPDATING` state, in which the ATC sends out the update instructions to all clients and goes to `UPDATING` state. If all clients have successfully loaded the weather information, the ATC goes to `POSTUPDATING` state waiting all clients to use the new weather information. If the ATC fails to get the new weather from the WCP in the weather updating process, the ATC will instruct all clients to use previous weather information and go into `POSTREVERTING` state to wait the responses from the clients. When all clients have updated the new weather or reverted to previous weather, the ATC goes into `DONE` state. The main control logic of ATC is implemented as an FSA with the aforementioned 9 states of ATC and transitions representing the actions of the ATC.

The other subject program we use is an XML type parser taken from SQL Power Architect which is a data modeling and profiling tool. Different from normal XML parser, the XML type parser tries to read as few bytes from the parsed XML file as possible and yet decide the document type of the XML file. It uses an FSA to maintain the parsing state. There are 5 states and 12 transitions. The 5 states are `OUTSIDE`, `BRACKET`, `COMMENT`, `IGNORE` and `TAG`. The state `OUTSIDE` means that parsed byte is outside any brackets. If the parsed byte is inside brackets but not part of a tag, then the state is `BRACKET`. When parsing XML comments, the state becomes `COMMENT`. Finally, if the parsed byte is part of a tag, the parsing state becomes `TAG`.

The FSAs used by the aforementioned programs and the mined FSAs are shown in Figure 13 and Figure 14. Transition conditions are omitted for simplicity. If a transition is met more than once in our technique, the number of times is labeled aside the transition. For the weather control logic in CTAS, our method constructs an FSA with 9 states and 24 transitions in 5.5 seconds. In contrast, path exploration based on path condition does not terminate within 1 hour. For the XML type parser in SQL Power Architect, our method constructs an FSA with 5 states and 37 transitions in 7.6 seconds. In contrast, path exploration based on path condition takes 6 minutes to construct an FSA with 1337 transitions. For both programs, all states and transitions are uncovered using our method. On the other hand, our method constructs FSAs with more transitions than the actual FSAs implemented by the subject programs. This is due to the imprecision of static analysis in computing potential dependence, causing more than one path with the same *relevant-slice condition* to be explored.

## 6. THREATS TO VALIDITY

*Internal threats.* An internal threat to validity comes from potential bugs in our implementation. We note that the slicing functionality of JSlice is thoroughly tested and has been widely used in both academia and industry. To guarantee the correctness of our symbolic execution implementation in JSlice, we have manually checked some of the generated path conditions/*relevant-slice conditions*.

Another source of threat to validity comes from the subject program selection in the evaluation. Studies on more subject programs could help better assess the effectiveness of our technique.

*Program crashes.* Our path exploration does not try to cover all paths. Instead, we try to group paths based on symbolic outputs. This is done with the goal of test-suite construction, where testing will expose possible failures in the program. However, failure of a test case does not only come from unexpected outputs - it can also come from program crashes. Thus, for the paths which we do not explore if they contain program crashes - these will not be exposed by the test-suite computed by our technique. For example, given `if(x>0)`{`p[i] = 0;`}, it is possible that the branch is never evaluated to true in our exploration process, hence any possible `ArrayIndexOutOfBoundsException` in the access to `p[i]` will not be spotted by the generated test-suite. For branches that are not in the relevant slice of any trace, our technique do not guarantee that both directions of the branch will be explored. Realistically, our test-suite construction could be supplemented by techniques to statically detect possible program crashes, such as memory errors [Xie et al. 2003].

*Approximation of relevant slice.* Due to the conservative nature of static analysis used in computing relevant slice, our technique may over-approximate the potential dependencies and hence the relevant slice. Since more branches will appear in the over-approximate relevant slice (than what should appear in the actual relevant slice), therefore the computed *relevant-slice conditions* from the over-approximated relevant slice will be stronger than the *relevant-slice conditions* that would have been computed from the actual relevant slice. In that case, we may explore more than one paths that have the same *relevant-slice condition*. Consider the following program.

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

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

```
1  int foo(int x, int y, int z){//input variables
2    int out; // output variable
3    String s = null;
4    int a;
5    int b = 2;
6    if(x - y > 0) //b1
7      a = x;
8    else
9      a = y;
10    if(x + y > 10) //b2
11     b = a;
12   if(z*z > 3)  //b3
13     s = "square(z) > 3";
14   else
15     s = "square(z) <= 3";
16   out = b;
17   System.out.println(s); //slicing criteria
18   return out; //slicing criteria
19 }
```

Fig. 15: Sample program with multiple outputs

*Different output types and multiple outputs.* Programs produce outputs in various ways including return value, side-effect on heap variables, direct interaction with files, etc. Our technique naturally considers all these output types of outputs on users' demand. When multiple outputs exist, users simply need to include all these outputs in the slicing criteria. Consequently, the computed relevant slices contain all statements affecting at least one of the outputs. However, including more outputs in slicing criteria increases the size of relevant slices and relevant slice conditions, which reduces the effectiveness of our technique. Let us use the example in Figure 1 to illustrate this issue. In our earlier discussion, we have been considering the return statement in line 16 as the slicing criteria. Suppose now we also want to consider the two printing statements in line 12 and 14 as outputs. We first need to transform the program into the program in Figure 15, so that the slicing criteria post-dominate the program entry. After the transformation, both line 17 and line 18 in Figure 15 are set as slicing criteria. Applying our technique, our tool generates the following summary with 6 entries as opposed to the 3-entry summary in Section 1 when only the return statement is set as the slicing criteria.

— If $x - y > 0$ and $x + y > 10$ and $z * z > 3$, then $s ==$ `"square(z) > 3"` and $out == x$
— If $x - y > 0$ and $x + y > 10$ and $z * z \leq 3$, then $s ==$ `"square(z) <= 3"` and $out == x$
— If $x - y \leq 0$ and $x + y > 10$ and $z * z > 3$, then $s ==$ `"square(z) > 3"` and $out == y$
— If $x - y \leq 0$ and $x + y > 10$ and $z * z \leq 3$, then $s ==$ `"square(z) <= 3"` and $out == y$
— If $x + y \leq 10$ and $z * z > 3$, then $s ==$ `"square(z) > 3"` and $out == 2$
— If $x + y \leq 10$ and $z * z \leq 3$, then $s ==$ `"square(z) <= 3"` and $out == 2$

With more outputs, our technique is less efficient. However, for the above program, our technique is still more efficient than full path exploration, which will exercise all eight paths.

*Scalability issues.* Apart from considering more outputs, lack of inherent parallelism in a program also reduces the effectiveness of our technique. If a program contains little inherent parallelism, the relevant slice of an input $t$ may contain the majority of the execution trace of input $t$. In such case, the improvement of our technique over path exploration based on path condition is limited. Although our technique considerably improves the efficiency of the path exploration, the path explosion problem still exists. In the worst case, the number of *relevant-slice conditions* grows exponentially with the number of branches in the program.

*SMT solver support.* The proofs presented in Section 3.2 assume that the underlying SMT solver is both sound and complete. The SMT solvers we have used are sound. That is, if the solver declares

```
1  int foo(int i){ //input variable
2    int a[2];
3    int out = 0;//output variable
4    a[0] = 0;
5    a[1] = 1;
6    if(a[i] > 0){
7      out = 2;
8    }
9    return out;
10 }
```

Fig. 16: Example program showing imprecise array modelling

that a formula is satisfiable (unsatisfiable), then the formula is indeed satisfiable (unsatisfiable). We now examine the completeness aspect of the assumption. In general, off-the-shelf SMT solvers are not complete for *relevant-slice conditions* generated from real programs. However, SMT solvers could be complete for formulae within certain theories. For example, the STP [Ganesh and Dill 2007] solver is sound and complete for quantifier-free formulae in the theory of bit-vectors and arrays. Therefore, if a subject program only uses fixed-size integer variables and the fixed-size integers are modeled as bitvector arrays, the STP solver then acts as a decision procedure for such formulae. In our experiment, Z3 is used as the underlying SMT solver. Z3 is not complete for non-liner integer operations. Although being incomplete, various heuristics incorporated into Z3 have been shown to be effective for solving formulae with non-liner integer operations in practice. Z3 has three types of output: `sat`, `unsat` and `unknown`. The `sat` output indicates that the formula is satisfiable and the `unsat` output indicates that the formula is unsatisfiable. The `unknown` output suggests that Z3 fails due to incompleteness. However, we did not observe any `unknown` output from Z3 in our evaluation presented in Section 5. If incompleteness of the underlying SMT solver occurs, our path exploration could be incomplete.

*Modeling of bytecode semantics in implementation.* Finally, we note that the completeness proof of Algorithm 1 in Section 3.2 also assumes 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. Let us consider the example program in Figure 16. Suppose the slicing criteria is at line 9. The precise *relevant-slice condition* for the execution trace of input $i == 0$ is $i \geq 0 \wedge i < 2 \wedge a[i] > 0 \wedge ((i == 0 \wedge a[i] == 0) \vee (i == 1 \wedge a[i] == 1))$. To get this precise *relevant-slice condition*, we need to trace all the possible assignments to any element of array `a[]`. However, in our implementation, we compute an approximated *relevant-slice condition* by concretizing the array index of any array reference as is done in other dynamic symbolic execution engine, such as BitBlaze [Song et al. 2008]. Therefore, the approximated *relevant-slice condition* we get is $a[1] > 0$, which could be reduced to true in this example. Because of this approximation, exploration then misses the case that the branch at line 6 could be evaluated to false.

## 7. RELATED WORK

The technique proposed in this paper is based on dynamic path exploration[Sen et al. 2005; Godefroid et al. 2005] and relevant slicing[Wang and Roychoudhury 2008; Agrawal et al. 1993; Gyimóthy et al. 1999]. 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 [Godefroid 2007], function summaries are generated and exploited. In [Godefroid et al. 2008], 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 the context of software evolution, to avoid redundant path exploration, Person et al. [Person et al. 2011] propose incremental symbolic execution that focuses on path conditions affected by program changes.

In [Santelices and Harrold 2010], 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 [Santelices and Harrold 2010] also compute a "path family condition" for each path family, which could characterize that path family. The path partition based on *relevant-slice condition* is different from the notion of "path family" in [Santelices and Harrold 2010] in the sense that "path family" is more abstract. For example, all program paths in Figure 1 can be grouped into one path family, but they are grouped into three partitions by our technique. At the same time, "path family condition" does not guarantee the same input-output relationship, whereas *relevant-slice condition* does. The main difference between our work and [Santelices and Harrold 2010] lies in the static vs. dynamic nature of the two techniques. The work in [Santelices and Harrold 2010] 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 [Santelices and Harrold 2010] over-approximates their "path family conditions" if needed. Clearly, the dynamic nature of our method makes it more suitable for test generation. Note that the effect of program statements written in real-life programming languages are hard to precisely model as symbolic formulae. In such a situation, under-approximation is a practical simplification, since it amounts to concretizing parts of the formula.

Other researchers have also used the notion of "path equivalence" to alleviate the path explosion problem. However, which paths are considered equivalent are different between our work and earlier works. The difference in the definition of path equivalence originates from the different goals of our work and earlier works. In [Boonstoppel et al. 2008], the goal is to explore all possible program states. Based on this goal, two paths are equivalent if the symbolic states of all live variables are the same. In contrast, we only consider the variables that can affect the output – two paths are equivalent if they have the same symbolic expression for the output. In [McMillan 2010], the goal is to reach some critical locations in a program. Therefore, two paths are equivalent if they cannot reach any critical locations for the same reason (blocked by the same condition).

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

Our *relevant-slice condition* can be used to minimize an existing test-suite[Harrold et al. 1993; Wong et al. 1995]. If a test-suite contains two test cases that have the same *relevant-slice condition*, these two test cases compute the output in the same way. Therefore, we can choose to eliminate one of them to make the test-suite smaller.

The work in [Csallner et al. 2008] explores paths to generate program invariants. For each path explored, the path condition serves as a pre-condition and the symbolic program output is treated as a post-condition. Thus, each explored path produces a program invariant which is defined as such a (pre-condition, post-condition) pair. Similar approaches are used in [Godefroid 2007; Person et al. 2008] to generate method summaries. Instead of using path condition, we can generate such program invariants using *relevant-slice conditions* — the *relevant-slice condition* is the pre-condition and for each *relevant-slice condition* explored, there is a unique symbolic output which serves as the post-condition. Moreover, the invariants generated using *relevant-slice conditions* will be simpler (as *relevant-slice conditions* are smaller than path conditions) and fewer (since a single *relevant-slice condition* groups more paths).

Our backward symbolic execution is different from call-chain-backward symbolic execution or CCBSE [Ma et al. 2011]. CCBSE is essentially a backward search procedure that starts from a target program location and searches for a feasible program path reaching the target. Forward symbolic execution is still used as sub-procedure in the search process. In contrast, our technique traverses traces/relevant slices backwardly to compute path conditions/*relevant-slice conditions*.

## 8. DISCUSSION

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

We experimentally compare the efficiency and coverage of our method with respect to path search method based on symbolic execution. The path partitioning computed by our method can be exploited in various 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. We have also shown the use of our method in test-suite augmentation. We generate test-cases to augment an existing test-suite by focusing on the different partitions across two program versions. Finally, we show that our method can help uncover finite state machine specifications from real program implementations.

## REFERENCES

CTAS Weather Control Requirements. `http://scesm04.upb.de/case-study-2/requirements.pdf`.

AGRAWAL, H., HORGAN, J. R., KRAUSER, E. W., AND LONDON, S. 1993. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*. ICSM '93. IEEE Computer Society, Washington, DC, USA, 348–357.

BOONSTOPPEL, P., CADAR, C., AND ENGLER, D. 2008. Rwset: Attacking path explosion in constraint-based test generation. *Tools and Algorithms for the Construction and Analysis of Systems 4963*, 351–366.

CSALLNER, C., TILLMANN, N., AND SMARAGDAKIS, Y. 2008. Dysy: dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*. ICSE '08. ACM, New York, NY, USA, 281–290.

DE MOURA, L. AND BJØRNER, N. 2008. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems 4963*, 337–340.

DO, H., ELBAUM, S., AND ROTHERMEL, G. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering 10,* 4, 405–435.

GANESH, V. AND DILL, D. L. 2007. A decision procedure for bit-vectors and arrays. In *CAV*. Springer-Verlag, Berlin, Heidelberg.

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

GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '08. ACM, New York, NY, USA, 206–215.

GODEFROID, P., KLARLUND, N., AND SEN, K. 2005. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '05. ACM, New York, NY, USA, 213–223.

GYIMÓTHY, T., BESZÉDES, A., AND FORGÁCS, I. 1999. 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. Springer-Verlag, London, UK, 303–321.

HARROLD, M. J., GUPTA, R., AND SOFFA, M. L. 1993. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol. 2*, 270–285.

MA, K., YIT PHANG, K., FOSTER, J., AND HICKS, M. 2011. Directed symbolic execution. *Static Analysis Symposium*, 95–111.

MCMILLAN, K. 2010. Lazy annotation for program testing and verification. In *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds. Lecture Notes in Computer Science Series, vol. 6174. Springer, Berlin/Heidelberg, 104–118.

PERSON, S., DWYER, M. B., ELBAUM, S., AND PĂSĂREANU, C. S. 2008. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. SIGSOFT '08/FSE-16. ACM, New York, NY, USA, 226–237.

PERSON, S., YANG, G., RUNGTA, N., AND KHURSHID, S. 2011. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. PLDI '11. ACM, New York, NY, USA, 504–515.

QI, D., NGUYEN, H. D., AND ROYCHOUDHURY, A. 2011a. Path exploration based on symbolic output. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ESEC/FSE '11. ACM, New York, NY, USA, 278–288.

QI, D., NGUYEN, H. D. T., AND ROYCHOUDHURY, A. 2011b. Path exploration based on soymbolic output. Tech. rep., National University of Singapore, `http://dl.comp.nus.edu.sg/dspace/handle/1900.100/3347`. March.

QI, D., ROYCHOUDHURY, A., AND LIANG, Z. 2010. Test generation to expose changes in evolving programs. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ASE '10. ACM, New York, NY, USA, 397–406.

QI, D., ROYCHOUDHURY, A., LIANG, Z., AND VASWANI, K. 2009. DARWIN: an approach for debugging evolving programs. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ESEC/FSE '09. ACM, New York, NY, USA, 33–42.

SANTELICES, R., CHITTIMALLI, P. K., APIWATTANAPONG, T., ORSO, A., AND HARROLD, M. J. 2008. Test-suite augmentation for evolving software. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. ASE '08. IEEE Computer Society, Washington, DC, USA, 218–227.

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

SEN, K., MARINOV, D., AND AGHA, G. 2005. 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. ACM, New York, NY, USA, 263–272.

SOFTWARE, S. 2012. SQL Power Architect. `http://code.google.com/p/power-architect/`.

SONG, D., BRUMLEY, D., YIN, H., CABALLERO, J., JAGER, I., KANG, M. G., LIANG, Z., NEWSOME, J., POOSANKAM, P., AND SAXENA, P. 2008. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.* Hyderabad, India.

WANG, T. AND ROYCHOUDHURY, A. 2008. Dynamic slicing on Java bytecode traces. *ACM Trans. Program. Lang. Syst. 30*, 10:1–10:49.

WONG, W. E., HORGAN, J. R., LONDON, S., AND MATHUR, A. P. 1995. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th international conference on Software engineering*. ICSE '95. ACM, New York, NY, USA, 41–50.

XIE, Y., CHOU, A., AND ENGLER, D. 2003. 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. ACM, New York, NY, USA, 327–336.

XIN, B., SUMNER, W. N., AND ZHANG, X. 2008. Efficient program execution indexing. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '08. ACM, New York, NY, USA, 238–248.

XU, Z., KIM, Y., KIM, M., ROTHERMEL, G., AND COHEN, M. B. 2010. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. FSE '10. ACM, New York, NY, USA, 257–266.