

Uniqueness of the Paths Explored by the Distributed Concolic Algorithm

Moonzoo Kim and Yunho Kim
CS Dept. KAIST, South Korea

Gregg Rothermel
CSE Dept. Univ. of Nebraska, Lincoln, US

1 Concolic Testing Process

This section presents an overview of the original (non-distributed) concolic testing process that performs static instrumentation of a target program to extract symbolic path formulas, which is the way SCORE operates. The concolic testing process proceeds via the following steps:

1. *Declaration of symbolic variables.* Initially, a user must specify which variables should be handled as symbolic variables, based on which symbolic path formulas are constructed.

2. *Instrumentation.* A target source program is statically instrumented with probes, which record symbolic path conditions from a concrete execution path when the target program is executed. For example, at each conditional branch, a probe is inserted to record the branch condition/symbolic path condition; then, the instrumented program is compiled into an executable binary file.

3. *Concrete execution.* The instrumented binary is executed with given input values. For the first execution of the program, initial input values are assigned randomly. From the second execution onwards, input values are obtained from Step 6.

4. *Obtain a symbolic path formula ϕ_i .* The symbolic execution part of the concolic execution collects symbolic path conditions over the symbolic input values at each branch point encountered for along the concrete execution path for a test case tc_i . Whenever each statement s of the target program is executed, a corresponding probe inserted at s updates the map of symbolic variables if s is an assignment statement, or collects a corresponding symbolic path condition, c , if s is a branch statement. Thus, a symbolic path formula ϕ_i is built at the end of the i th execution as $c_1 \wedge c_2 \dots \wedge c_n$ where c_n is the last path condition executed and c_k is executed earlier than c_{k+1} for all $1 \leq k < n$.

5. *Generate a new symbolic path formula ψ_i .* Given a symbolic path formula ϕ_i obtained in Step 4, to obtain the next input values, ψ_i is generated by negating one path condition c_j and removing subsequent path conditions (i.e., $\psi_i = c_1 \wedge c_2 \dots \wedge \neg c_j$).

```

01:int main() {
02:  int x, y,z, max_num=0;
03:  SYM_INT(x); // Declaration of x, y, z
04:  SYM_INT(y); // as symbolic integer
05:  SYM_INT(z); // variables
06:
07:  if(x >= y) {
08:    // SYM_COND(x,y, ">=");
09:    if(y >= z) {
10:      // SYM_COND(y,z, ">=");
11:      max_num = x;
12:    } else {
13:      // SYM_COND(y,z, "<");
14:      if (x >= z){
15:        // SYM_COND(x,z, ">=");
16:        max_num = x;
17:      } else {
18:        // SYM_COND(x,z, "<");
19:        max_num = z;
20:      }
21:    }
22:  } else { ...}
23:  printf("%d is the largest number among\
24:         { %d,%d,%d}", max_num, x,y,z);
25:  // SMT_Solve();
26:}

```

Figure 1: Example used to illustrate concolic testing

For example, if a depth first search (DFS) strategy is used, as it often is, to explore the symbolic path formula, then c_j is the last symbolic path condition in ϕ_i whose negated path condition has not been executed previously. If ψ_i is unsatisfiable, another path condition $c_{j'}$ is negated and subsequent path conditions are removed until a satisfiable path formula is found. If there are no further new paths to try, the algorithm terminates.

6. *Select the next input values tc_{i+1} .* A constraint solver such as a Satisfiability Modulo Theory (SMT) solver generates a model that satisfies ψ_i . This model determines the next concrete input values to try (i.e., tc_{i+1}), and the concolic testing procedure iterates from Step 3 using these input values.

We illustrate this process through an example involving Figure 1, which returns the largest number from three given integers.

1. *Declaration of symbolic variables.* A user declares x , y , and z as symbolic integer variables by using `SYM_INT()` (lines 3-5).

2. *Instrumentation.* A concolic testing tool (i.e., SCORE) inserts a probe to record a corresponding path condition at each `then` branch in an automated manner. Similarly, at each `else` branch, a probe is inserted to record a corresponding path condition. In Figure 1, probes inserted through instrumentation are shown as comments. For example, at line 10, `SYM_COND(y, z, ">=")` is inserted to record path condition $y \geq z$.

Similarly, `SYM_COND (y, z, "<")` is inserted at line 13 to record path condition $y < z$.

3. *Concrete execution.* Initial input values for the symbolic variables are randomly chosen. We assume that x , y , and z are assigned 1, 1, and 0 as initial random values, respectively (i.e., $tc_1 = \langle 1, 1, 0 \rangle$). Then, the instrumented target program executes lines 2-11 and lines 23-26.

4. *Obtain a symbolic path formula ϕ_i .* During the concrete execution of lines 2-11, the probes record two symbolic path conditions $x \geq y$ and $y \geq z$ through `SYM_COND (x, y, ">=")` (line 8) and `SYM_COND (y, z, ">=")` (line 10) respectively. Thus, the symbolic formula $\phi_1 = (x \geq y) \wedge (y \geq z)$ is obtained for the first iteration.

5. *Generate a new symbolic path formula ψ_i .* If a DFS algorithm is used, ψ_1 is $(x \geq y) \wedge \neg(y \geq z)$.

6. *Select the next input values.* At line 25, the target program finishes its first iteration and invokes a constraint solver to solve ψ_1 . Suppose that an SMT solver solves ψ_1 and generates 1, 1, and 2 for x , y , and z as a solution (i.e., $tc_2 = \langle 1, 1, 2 \rangle$). Then, the target program starts the second iteration with these values, and the entire process from Step 3 is repeated.

2 Concolic Algorithm

Algorithm 1 presents the concolic testing algorithm that corresponds to Steps 3 through 6 just detailed. Algorithm 1 receives a current test case tc_i , neg_limit_i , and an index i to the test case tc_i as parameters. neg_limit_i is an index to the path condition (PC) in ϕ_i beyond which PCs should not be negated (lines 7-9) (i.e., c_k should not be negated for $k < neg_limit_i$), where ϕ_i is a symbolic path formula obtained from an execution path on tc_i . (The use of neg_limit prevents the recursive `Concolic()` call in line 13 from generating redundant test cases). Initially, tc_1 is given as a random value, neg_limit_1 is 1, and $i = 1$.

The algorithm generates symbolic path formulas ψ_i s by negating PCs of ϕ_i one by one in decreasing order (line 9). Then, it generates new test cases tc_{i+1} by solving the ψ_i s (line 11). From each new test case tc_{i+1} generated in the loop, the algorithm generates further test cases to explore execution paths that share a common prefix (i.e., $c_1 \wedge \dots \wedge \neg c_j$) by calling `Concolic(tc_{i+1}, j + 1, i + 1)` in a recursive manner (line 13).

Note that the concolic algorithm traverses the execution tree of a target program in a depth first search (DFS) order and, under the assumption that ϕ_i truly reflects $path_i$ and `Solve(ψ_i)` can solve ψ_i ,¹ In addition, generated test cases do *not* explore the same execution path again (see Corollary 1).

Theorem 1 (UNIQUENESS OF GENERATED TEST CASES)

$\forall k, l \geq 1. (k \neq l \rightarrow tc_k \neq tc_l)$ in Algorithm 1.

¹In practice, a program P may contain complex arithmetic or binary library calls that cannot be solved or reasoned about by SMT solvers. Thus, the concolic algorithm generates symbolic path formulas without such conditions, and in these cases this may result in the generation of identical symbolic path formulas from different execution paths.

Input: tc_i : i th test case to run neg_limit_i : a position of the PC in ϕ_i beyond which PCs should not be negated i : a number of test cases generated so far**Output:** n_{tc} : a number of test cases generated so farA set of generated test cases (i.e., tc_{i+1} s of line 11)

```

1 Concolic( $tc_i, neg\_limit_i, i$ ) {
2 // Step 3: Concrete execution
3  $path_i$  = an execution path of a target program running on  $tc_i$ 
4 // Step 4: Obtain a symbolic path formula  $\phi_i = c_1 \wedge \dots \wedge c_n$ 
5  $\phi_i$  = a symbolic path formula obtained from  $path_i$ 
6  $j = |\phi_i|$ ; //  $|\phi_i| = n$  where  $c_n$  is the last PC in  $\phi_i$ 
7 while  $j \geq neg\_limit_i$  do
8   // Step 5: Generate  $\psi_i$  for the next input values
9    $\psi_i = c_1 \wedge \dots \wedge c_{j-1} \wedge \neg c_j$ ;
10  // Step 6: Selecting the next input values
11   $tc_{i+1} = Solve(\psi_i)$ ; // NULL if  $\psi_i$  is unsatisfiable
12  if  $tc_{i+1}$  is not NULL then
13    |  $i = Concolic(tc_{i+1}, j + 1, i + 1)$ ;
14  end
15   $j = j - 1$ ;
16 end
17  $n_{tc} = i$ ;
18 return  $n_{tc}$ ;
19 }
```

Algorithm 1: Original concolic algorithm

Suppose that there exist $k, l \geq 1$ such that $k \neq l$ and $tc_k = tc_l$. Then, there exist corresponding symbolic path formulas ψ_{k-1} and ψ_{l-1} whose solution is $tc_k (= tc_l)$. (Since tc_1 is given as a random initial value, $\psi_0 = true$.) Since ψ_{k-1} and ψ_{l-1} are symbolic path formulas, if $tc_k = tc_l$, then $\psi_{k-1} = \psi_{l-1}$ (contrapositive of Lemma 2). However, Lemma 1 shows that there are no $k, l \geq 0$ such that $k \neq l$ and $\psi_k = \psi_l$. Contradiction.

Corollary 1 (UNIQUENESS OF EXPLORED PATHS)

Concolic() in Algorithm 1 does not explore the same path again. In other words, $\forall k, l \geq 1. (k \neq l \rightarrow \phi_k \neq \phi_l)$

From Lemma 1 and Lemma 2.

Lemma 1 (UNIQUENESS OF GENERATED SYMBOLIC PATH FORMULAS)

$\forall k, l \geq 0. (k \neq l \rightarrow \psi_k \neq \psi_l)$ in Algorithm 1.

First, if $k = 0$ Lemma 1 is trivially true, since $\psi_0 = true$ and $\forall l \geq 1. \psi_0 \neq \psi_l$.

Second, suppose that $1 \leq k < l$. There are two places where a new symbolic formula ψ_l is generated:

- ψ_k is different from ψ_l which is generated at the earliest subsequent iteration where a new test case tc_{l+1} is generated, because $|\psi_k| > |\psi_l|$ as j (i.e., $|\psi_l|$) decreases monotonically through iterations.
- Case 2: inside a recursive *Concolic()* call (line 13)
 ψ_k is different from any ψ_l that is generated inside of a recursive *Concolic*($tc_{k+1}, j + 1, k + 1$) call. This is because $\forall l. |\psi_k| < |\psi_l|$ where ψ_l is generated inside of *Concolic*($tc_{k+1}, j + 1, k + 1$). Note that $|\psi_k| = j$ and $\forall l. |\psi_l| > j$, since *neg_limit* $_{k+1}$ in *Concolic*($tc_{k+1}, j + 1, k + 1$) is $j + 1$ (line 7).

Therefore, since $|\psi_k| \neq |\psi_l|$, $\psi_k \neq \psi_l$.

For the cases where $k > l$, the above proof applies in a similar manner. Therefore, there exists no $k, l \geq 0$ such that $k \neq l$ and $tc_k = tc_l$.

Lemma 2 $\forall k, l \geq 0. (\psi_k \neq \psi_l \rightarrow tc_{k+1} \neq tc_{l+1})$ and $\forall k, l \geq 0. (\psi_k \neq \psi_l \rightarrow \phi_{k+1} \neq \phi_{l+1})$ in Algorithm 1

Suppose that $\psi_k = c_{k1} \wedge c_{k2} \dots \wedge c_{kj_k}$ and $\psi_l = c_{l1} \wedge c_{l2} \dots \wedge c_{lj_l}$. There are three cases to handle to prove Lemma 2.

First, when $k = l$, Lemma 2 is trivially true.

Second, for the cases where $k, l \geq 0 \wedge k < l \wedge \psi_k \neq \psi_l$ ($k < l$ indicates that ψ_k is generated before ψ_l is generated), there are two relationships between ψ_k and ψ_l . Note that all generated symbolic path formulas (ψ_i 's) start from the same root of the execution tree (i.e., *main* ()) of a target program).

- Case 1: ψ_k is *not* a prefix of ψ_l

There is at least one path condition c_{km} such that $\neg c_{km}$ is a path condition of ψ_l , since every symbolic path formula starts from the same program entry point. Thus, tc_{k+1} (a solution of ψ_k) cannot satisfy ψ_l and $tc_{k+1} \neq tc_{l+1}$. In addition, $\phi_{k+1} \neq \phi_{l+1}$, since ψ_k and ψ_l are prefixes of ϕ_{k+1} and ϕ_{l+1} respectively.

For example, ψ_k in Figure 2(a) is not a prefix of ψ_l . ψ_k has $c1$, but ψ_l has $\neg c1$ which results in $tc_{k+1} \neq tc_{l+1}$ and $\phi_{k+1} \neq \phi_{l+1}$.

- Case 2: ψ_k is a prefix of ψ_l

Suppose that a symbolic path formula $\phi_{k+1} = c_{(k+1)1} \wedge c_{(k+1)2} \dots \wedge c_{(k+1)t}$ is obtained from an execution path on tc_{k+1} (a solution of ψ_k). Then, there is at least one path condition $c_{(k+1)m}$ such that $\neg c_{(k+1)m}$ is a path condition of ψ_l . This is because every symbolic path formula starts from the same root and $c_{(k+1)m}$ should be negated to generate a subsequent ψ_l where $m \geq neg_limit_{k+1}$. Note that *neg_limit* $_{k+1}$ restricts the range of path conditions to negate so as to prevent repetitive negations on same path conditions (line 7 of Algorithm 1) for all subsequent ψ_l 's. Thus, tc_{l+1} cannot satisfy ϕ_{k+1} . Given that a solution of ψ_k (i.e.,

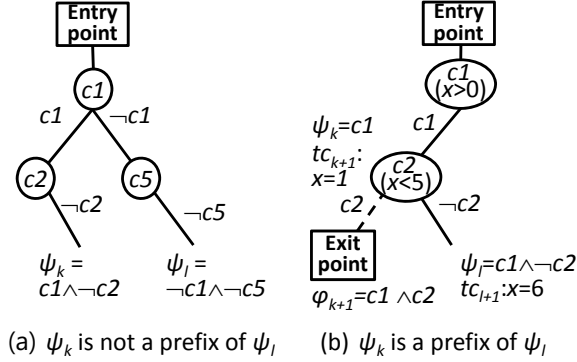


Figure 2: Two relationships between ψ_k and ψ_l

tc_{k+1}) is a solution of ϕ_{k+1} , $tc_{k+1} \neq tc_{l+1}$. In addition, for a similar reason, $\phi_{k+1} \neq \phi_{l+1}$.

For example, $\psi_k = c1(x > 0)$ in Figure 2(b) is a prefix of $\psi_l = c1 \wedge \neg c2((x > 0) \wedge \neg(x < 5))$. Suppose that $Solve()$ generates a solution to $c1$ as $x = 1$ (i.e., $tc_{k+1} : x = 1$). Then, the next symbolic path formula ψ_l contains $\neg c2$, since one path condition in ϕ_{k+1} should be negated to generate ψ_l and $c2$ is located lower than $c1$. Thus, $\psi_l = c1 \wedge \neg c2$ and its solution tc_{l+1} can be $x = 6$.

Third, for the cases where $k, l \geq 1 \wedge k > l \wedge \psi_k \neq \psi_l$, a similar proof for the second case applies.

3 Distributed Concolic Algorithm

The distributed concolic algorithm relies on the fact that the following core part of Algorithm 1 can be processed *independently*:

From **each new test case** tc_{i+1} in the loop, the algorithm generates **further test cases** to explore all possible execution paths that share a common prefix (i.e., $c_1 \wedge \dots \wedge \neg c_j$) by calling $Concolic(tc_{i+1}, j + 1, i + 1)$ in a recursive manner (line 13).

In other words, the loop in Algorithm 1 generates test case pairs (i.e., $(tc_{i+1}, j + 1)$ s), each of which can be processed by a distributed node independently. To generate test cases in a distributed manner, a node processes (tc, neg_limit) in its queue q_{tc} (lines 14-33 of Algorithm 2) and stores a new pair $(tc_{i+1}, j + 1)$ in q_{tc} (line 27). In this way, a recursive $Concolic()$ call (line 13 of Algorithm 1) is transformed into a loop with q_{tc} in Algorithm 2 (lines 14-33). Or, a node transfers (tc, neg_limit) s in q_{tc} to other nodes whose q_{tc} is empty so that the other nodes can process (tc, neg_limit) s.

Initially, one startup node running $DstrConcolic()$ starts the testing process (lines 6-7) and begins generating test cases. Then, the node transfers generated test cases to other nodes that request initial test cases (lines 9-10). If q_{tc} is empty (exiting

the loop of lines 14-33) and the q_{tc} s of all distributed nodes are empty, the algorithm terminates (line 39). Otherwise (i.e., there is another node n'' that has test cases), a current node n requests test cases from another node n'' (line 35) and receives test cases from n'' (line 36). The received test cases are then added into q_{tc} (line 37) and the algorithm continues from line 13.

This distributed concolic algorithm does not generate redundant test cases (test cases that cover the same path), just as the non-distributed concolic algorithm does not (again, with the assumption that ϕ_i truly reflects $path_i$ and $Solve(\psi_i)$ can solve ψ_i). Theorems 2 and 3 confirm this property of the distributed algorithm in the SCORE framework.

Theorem 2 *If there is only one node and the node runs $DstrConcolic()$ with $startup$ as $true$ in Algorithm 2, the node does not generate redundant test cases.*

A brief proof sketch is as follows. We prove that Algorithm 2 with $startup$ as $true$ is equivalent to Algorithm 1. This can be shown by step-by-step transformation of the recursive $Concolic()$ of Algorithm 1 into the `while` loop (lines 14-33) with q_{tc} of Algorithm 2 (q_{tc} simulates a call stack to store actual parameters of recursive $Concolic()$). Then, from Theorem 1 and Corollary 1, Algorithm 2 does not generate redundant test cases.

Theorem 3 *Algorithm 2 does not generate redundant test cases among the distributed nodes.*

A brief proof sketch is as follows. Theorem 2 shows that $DstrConcolic()$ does not generate redundant test cases on one node. In addition, $DstrConcolic()$ generates test cases based on only (tc, neg_limit) in q_{tc} . Thus, a node n that receives (tc, neg_limit) s from a node n' generates the same test cases, as if n' generates the test cases from these (tc, neg_limit) s. In other words, when Algorithm 2 terminates, the total set of test cases generated by multiple distributed nodes are the same as the set of test cases generated by one node. Consequently, Theorem 3 follows from Theorem 2.

```

Input:
startup: a flag to indicate whether a current node n is a startup node or not.
Output:
 $TC_n$ : a set of test cases generated at a current node n (i.e.,  $tc_{i+1}$ s of line 25)
1 DstrConcolic(startup) {
2  $q_{tc} = \emptyset$ ; // a queue containing (tc, neg_limit)s
3  $TC_n = \emptyset$ ; // a set of generated test cases
4  $i = 1$ ;
5 if startup then
6   |  $tc_1 = \text{random value}$ ; // initial test case
7   | Add ( $tc_1, 1$ ) to  $q_{tc}$ ;
8 else
9   | Send a request for test cases to  $n'$ ;
10  | Receive (tc, neg_limit)s from  $n'$ ;
11  | Add (tc, neg_limit)s to  $q_{tc}$ ;
12 end
13 while true do
14   | while  $|q_{tc}| > 0$  do
15   |   Remove (tc, neg_limit) from  $q_{tc}$ ;
16   |   // Step 3: Concrete execution
17   |    $path_i = \text{an execution path of a target program running on } tc$ 
18   |   // Step 4: Obtain a symbolic path formula  $\phi_i$ 
19   |    $\phi_i = \text{a symbolic path formula obtained from } path_i$ 
20   |    $j = |\phi_i|$ ;
21   |   while  $j \geq neg\_limit$  do
22   |   |   // Step 5: Generate  $\psi_i$  for the next input values
23   |   |    $\psi_i = c_1 \wedge \dots \wedge c_{j-1} \wedge \neg c_j$ ;
24   |   |   // Step 6: Select the next input values
25   |   |    $tc_{i+1} = \text{Solve}(\psi_i)$ ;
26   |   |   if  $tc_{i+1}$  is not NULL then
27   |   |   |   Add ( $tc_{i+1}, j + 1$ ) to  $q_{tc}$ ;
28   |   |   |    $TC_n = TC_n \cup \{tc_{i+1}\}$ ;
29   |   |   |    $i = i + 1$ ;
30   |   |   end
31   |   |    $j = j - 1$ ;
32   |   end
33   end
34   | if there is a test case in another node  $n''$  then
35   |   | Send a request for test cases to  $n''$ 
36   |   | Receive (tc, neg_limit)s from  $n''$ 
37   |   | Add (tc, neg_limit)s to  $q_{tc}$ 
38   | else
39   |   | Halt; // no test cases exist in all nodes
40   | end
41 end
42 }

```

Algorithm 2: Distributed Concolic algorithm