Dynamically Scalable Concolic Testing through the SCORE Framework

Moonzoo Kim

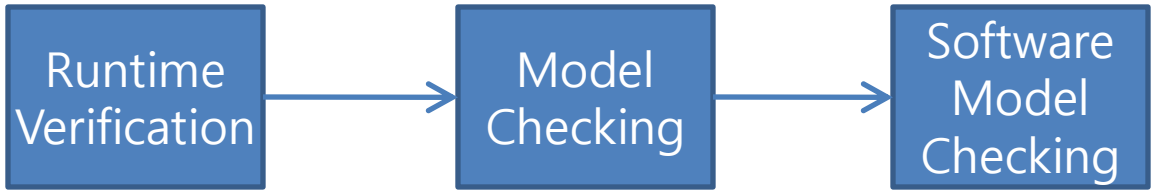
Provable Software Lab, CS Dept., KAIST

Collaborated with Yunho Kim

Personal Research Roadmap

(Fight State Explosion w/ Intelligent Automated Analysis for Real-world Applications)

- Past: RV (dynamic) && MC (static)



- Current: Extended Concolic Testing

- Future: Concolic Testing with Intelligence

Overview

- Introduction
- Concolic Testing
- Distributed Concolic Testing
- Conclusion and Future Work

Motivation

- Concolic testing can provide
  - an effective way to detect hidden bugs by,
  - generating test cases in an automatic and exhaustive manner

- However, our previous work[SBMF'09] revealed that
  - it took a large amount of time, because
  - there exist a huge number of formulas to solve

- **Solution: distributed algorithm for concolic testing**

Concolic Testing Framework

- A combined approach of dynamic concrete analysis and static symbolic analysis
- Automated Unit Testing of real-world C Programs
  - Execute a unit under test on automatically generated test inputs, so that all possible execution paths are explored (i.e., aiming path coverage)
- In a nutshell
  1. A target C program is statically instrumented with probes, which record symbolic path conditions
  2. The instrumented C program is executed with given input values
     - Initial input values are assigned randomly
  3. Use a concrete execution over a concrete input to obtain a symbolic execution path formula φ_i
  4. One branch condition of φ_i is negated to generate the next symbolic execution path formula ψ_i
  5. A constraint solver gets concrete input values to satisfy ψ_i
     - Ex. ψ_i : (x < 2) && (2x + 3y < 7). One solution is x=1 and y=1
  6. Repeat step 1 until all feasible execution paths are explored

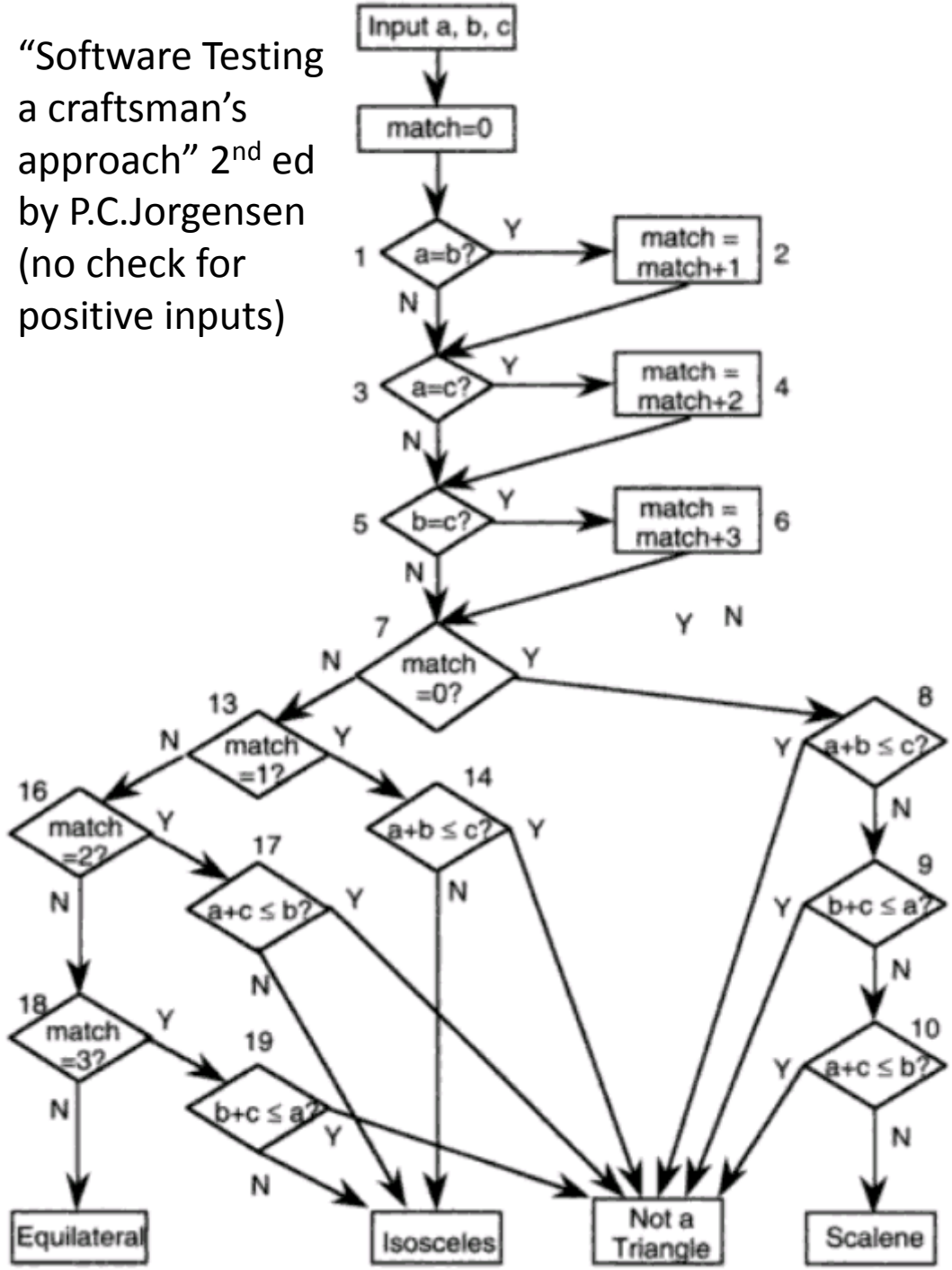
Instrumentation Example

```
01:#include<SYMBOLIC.h>
02:int main() {
03: int x, y,z, max_num=0;
04: SYMBOLIC_int(x); SYMBOLIC_int(y); SYMBOLIC_int(z); // symbolic input declaration
05: if(x >= y) { // SYMBOLIC_sym_cond(x,y, ">=");
06:   if(y >= z) { // SYMBOLIC_sym_cond(y,z, ">=");
07:     max_num = x;
08:   } else { // SYMBOLIC_sym_cond(y,z, "<");
09:     if (x >= z){ // SYMBOLIC_sym_cond(x,z, ">=");
10:       max_num = x;
11:     } else { // SYMBOLIC_sym_cond(x,z, "<");
12:       max_num = z;
13:     }
14:   }
15: } else { ...}
16: printf("%d is the largest number among {%d,%d,%d}", max_num, x,y,z);
17: // if tc1:x=1,y=1,z=0, then ϕ1:x>=y ∧ y>= z and ψ1:x>=y ∧ ¬(y>= z)
18: // SYMBOLIC_Solve(ψi);
```

Inserted probes to obtain symbolic path conditions at run-time

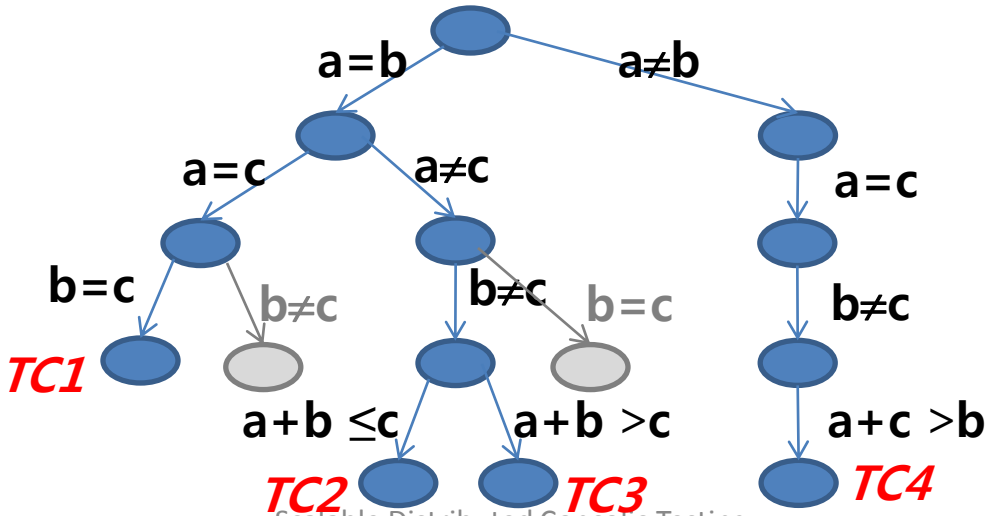
```
1 #include <SYMBOLIC.h>
2 main() {
3     int a,b,c, match=0;
4     SYMBOLIC_int(a); SYMBOLIC_int(b); SYMBOLIC_
       int(c);
       // filtering out invalid inputs
5     if(a <= 0 || b<= 0 || c<= 0) exit();
6     printf("a,b,c = %d,%d,%d:",a,b,c);
7
8     //0: Equilateral, 1:Isosceles,
      // 2: Not a traiangle, 3:Scalene
9     int result=-1;
10     if(a==b) match=match+1;
11     if(a==c) match=match+2;
12     if(b==c) match=match+3;
13     if(match==0) {
14         if( a+b <= c) result=2;
15         else if( b+c <= a) result=2;
16         else if(a+c <= b) result =2;
17         else result=3;
18     } else {
20         if(match == 1) {
20             if(a+b <= c) result =2;
21             else result=1;
22         } else {
23             if(match ==2) {
24                 if(a+c <=b) result = 2;
25                 else result=1;
26             } else {
20                 if(match==3) {
20                     if(b+c <= a) result=2;
29                     else result=1;
30                 } else result = 0;
31             } }}
32     printf("result=%d\n",result);
33 }
```

"Software Testing a craftsman's approach" 2nd ed by P.C.Jorgensen (no check for positive inputs)



Concolic Testing the Triangle Program

Test case	Input (a,b,c)	Executed symbolic path formula φ	Generated symbolic path formula ψ for next input	Solution for ψ
1	1,1,1	$a=b \wedge a=c \wedge b=c$	$a=b \wedge a=c \wedge b\neq c$	Unsat
			$a=b \wedge a\neq c$	1,1,2
2	1,1,2	$a=b \wedge a\neq c \wedge b\neq c \wedge a+b \leq c$	$a=b \wedge a\neq c \wedge b\neq c \wedge a+b > c$	2,2,3
3	2,2,3	$a=b \wedge a\neq c \wedge b\neq c \wedge a+b > c$	$a=b \wedge a\neq c \wedge b=c$	Unsat
			$a\neq b$	2,1,2
4	2,1,2	$a\neq b \wedge a=c \wedge b\neq c \wedge a+c>b$	$a\neq b \wedge a=c \wedge b\neq c \wedge a+c \leq b$	2,5,2



Original Concolic Algorithm

Input:
tc_i: ith 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 far
A 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 $\phi_i = c_1 \wedge ...c_n$
5 ϕ_i = a symbolic path formula obtained from $path_i$
6 $j = \mid \phi_i \mid$; // $\mid \phi_i \mid = n$ where c_n is the last PC in ϕ_i
7 **while** $j >= neg\_limit_i$ **do**
8 // Step 5: Generate ψ_i for the next input values
9 $\psi_i = c_1 \wedge ... \wedge c_{j-1} \wedge \neg c_j$;
10 // Step 6: Selecting the next input values
11 $tc_{i+1} = Solve(\psi_i)$; // NULL if ψ_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 }

Proof for Non-Redundant Test Cases and Non-Redundant Path Exploration

Theorem 1 (UNIQUENESS OF GENERATED TEST CASES)
$\forall k, l \geq 1.(k \neq l \rightarrow tc_k \neq tc_l)$ *in Algorithm 1.*

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.

Distributed Concolic Algorithm

Input:
startup: a flag to indicate whether a current node is a startup node or not.
Output:
a set of generated test cases (i.e., tc_{i+1} of line 12)

```
1  DstrConcolic(startup) {
2  q_tc = ∅; // a queue containing (tc, neg_limit)'s
3  i = 1;
4  if startup then
5      tc_1 = random value; // initial test case
6      Add (tc_1, 1) to q_tc;
7  else
8      Send a request for test cases to n';
9      Receive (tc, neg_limit)'s from n';
10     Add (tc, neg_limit)'s to q_tc;
11 end
12 while true do
13     while | q_tc |> 0 do
14         Remove (tc_i, neg_limit_i) from q_tc;
15         // Step 3: Concrete execution
16         path_i = an execution path of a target program running on tc_i
17         // Step 4: Obtain a symbolic path formula φ_i
18         φ_i = a symbolic path formula obtained from path_i
19         j =| φ_i |;
20         while j >= neg_limit_i do
21             // Step 5: Generate ψ_i fro the next input values
22             ψ_i = c_1 ∧ ... ∧ c_{j-1} ∧ ¬c_j ;
23             // Step 6: Select the next input values
24             tc_{i+1} = Solve(ψ_i);
25             if tc_{i+1} is not NULL then
26                 Add (tc_{i+1}, j + 1) to q_tc;
27                 i = i + 1;
28             end
29             j = j − 1;
30         end
31     end
32     if there is a test case in another node n' then
33         Send a request for test cases to n'
34         Receive (tc, neg_limit)'s from n'
35         Add (tc, neg_limit)'s to q_tc
36     else
37         Halt; // no test cases exist in all nodes
38     end
```

- To handle symbolic path formulas one-by-one explicitly, distributed concolic testing algorithm uses a queue instead of recursion
  - Invoking *Concolic()* function(line 10 in original) is matched to add a formula to q_{pf}(line 15 in distributed)
- In other words, recursive Concolic() in the original algorithm is translated into a while loop with q_{pf} in the distributed concolic

No Redundant Computation in Distributed Concolic Algorithm

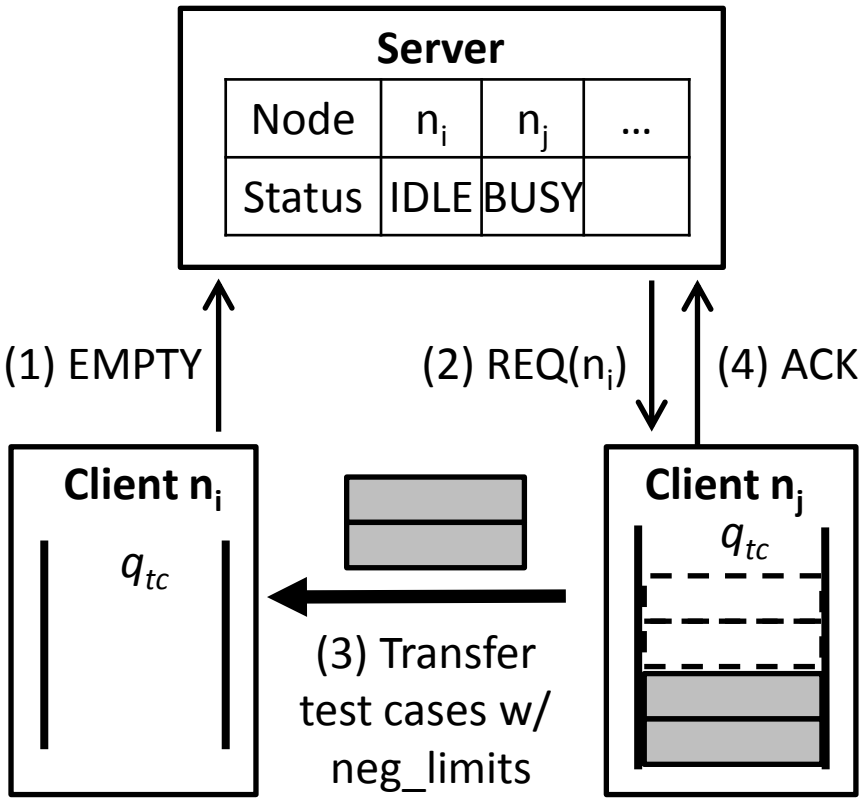
Theorem 2 *If there is only one node and the node runs $DstrConcolic()$ with $startup$ as $true$ in Algorithm 2, the node does* not *generate redundant test cases.*

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 13-31) 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 the Algorithm 2 terminates, the total test cases generated by m distributed nodes are the same as the test cases generated by one node. Consequently, Theorem 3 follows from Theorem 2.

Communication between Nodes

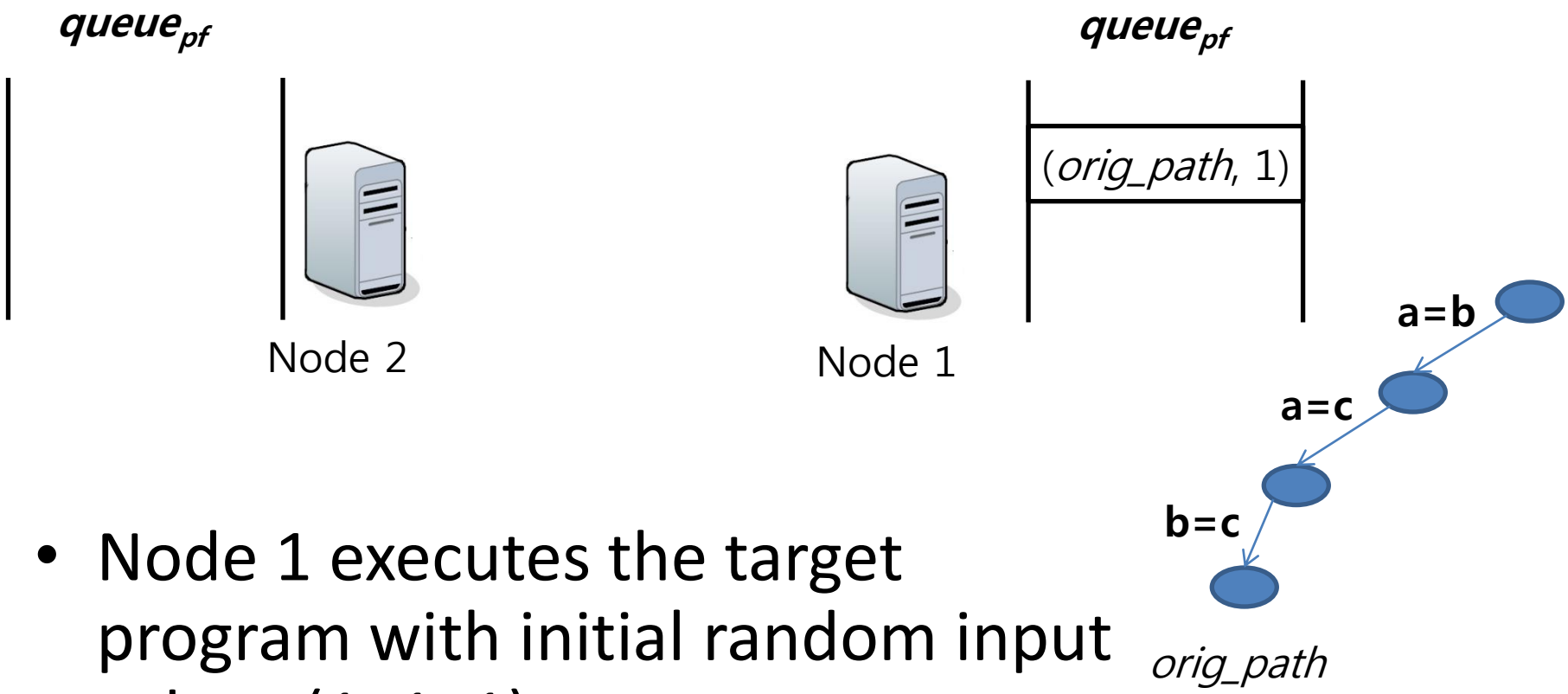


- There is one server and multiple clients
  - One client is designated as a startup client
- To transfer test cases, a client should communicate with the server
  - Note that communication only occurs when q_{tc} is empty

Distributed Concolic Algorithm on the Triangle Example

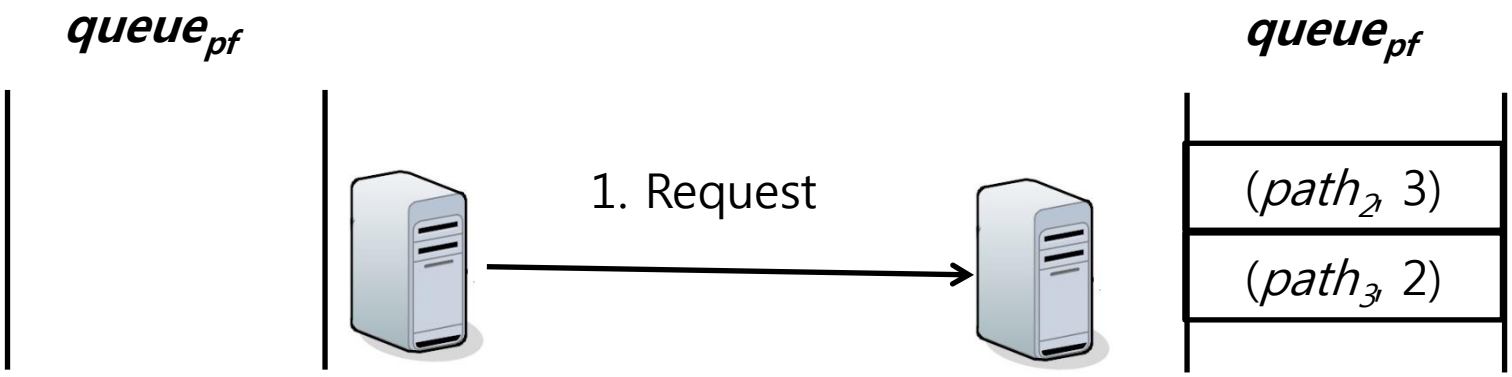
- Suppose that we have two nodes and node 1 is a starting node

Algorithm Startup

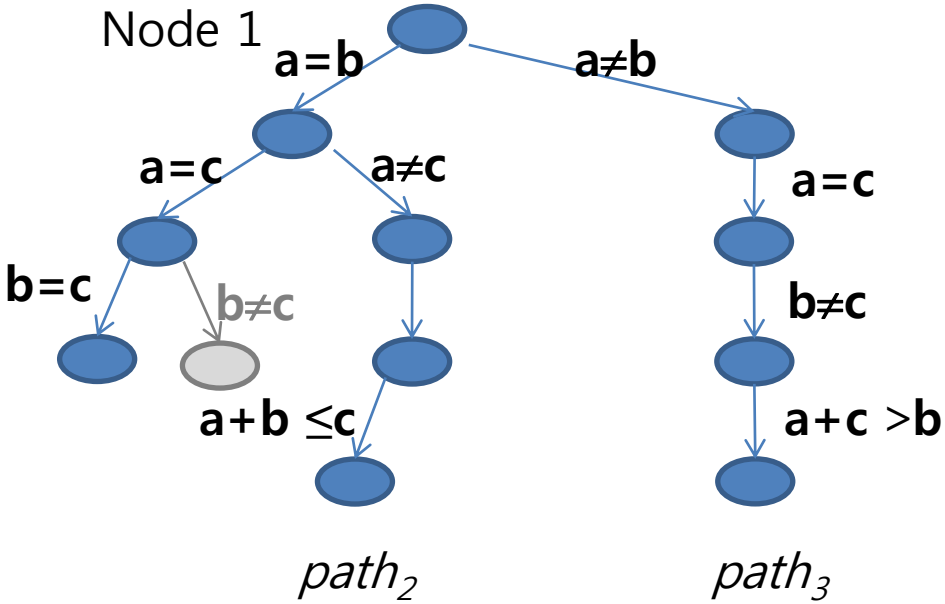


- Node 1 executes the target program with initial random input values (1, 1, 1)

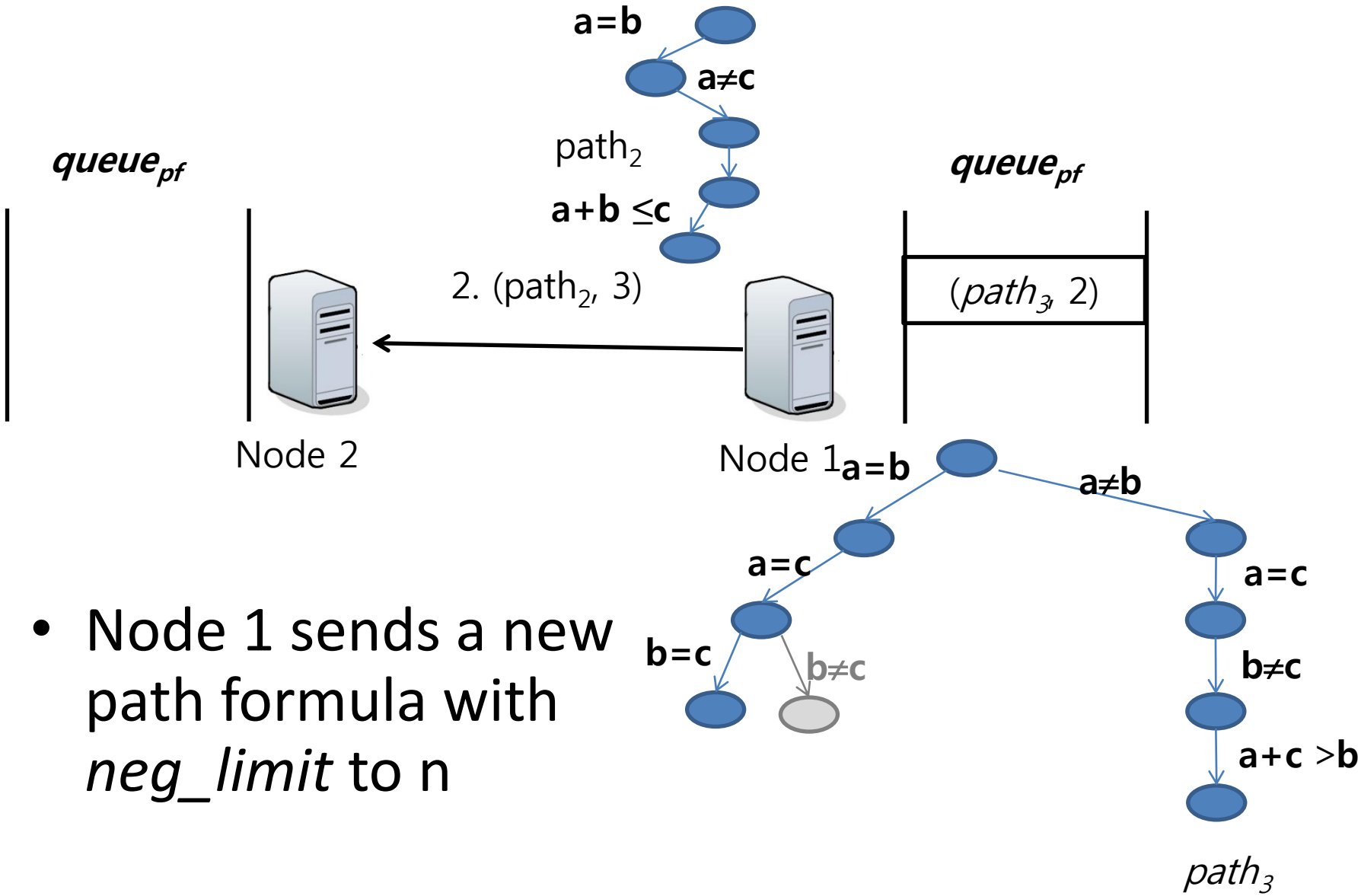
Request for a New Symbolic Path Formula



- Node 2 sends a request to 1 for a symbolic path formula

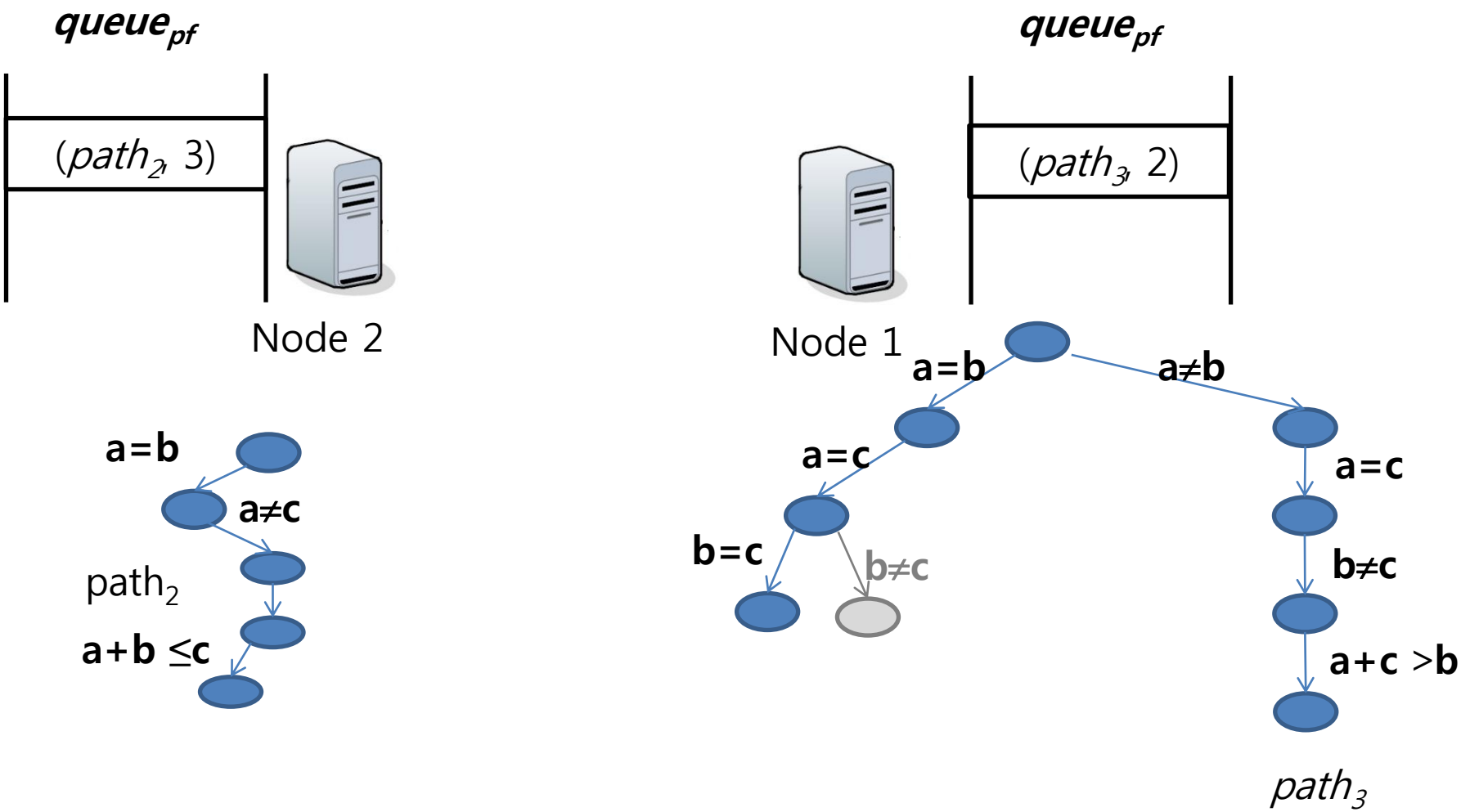


Send a New Symbolic Path Formula

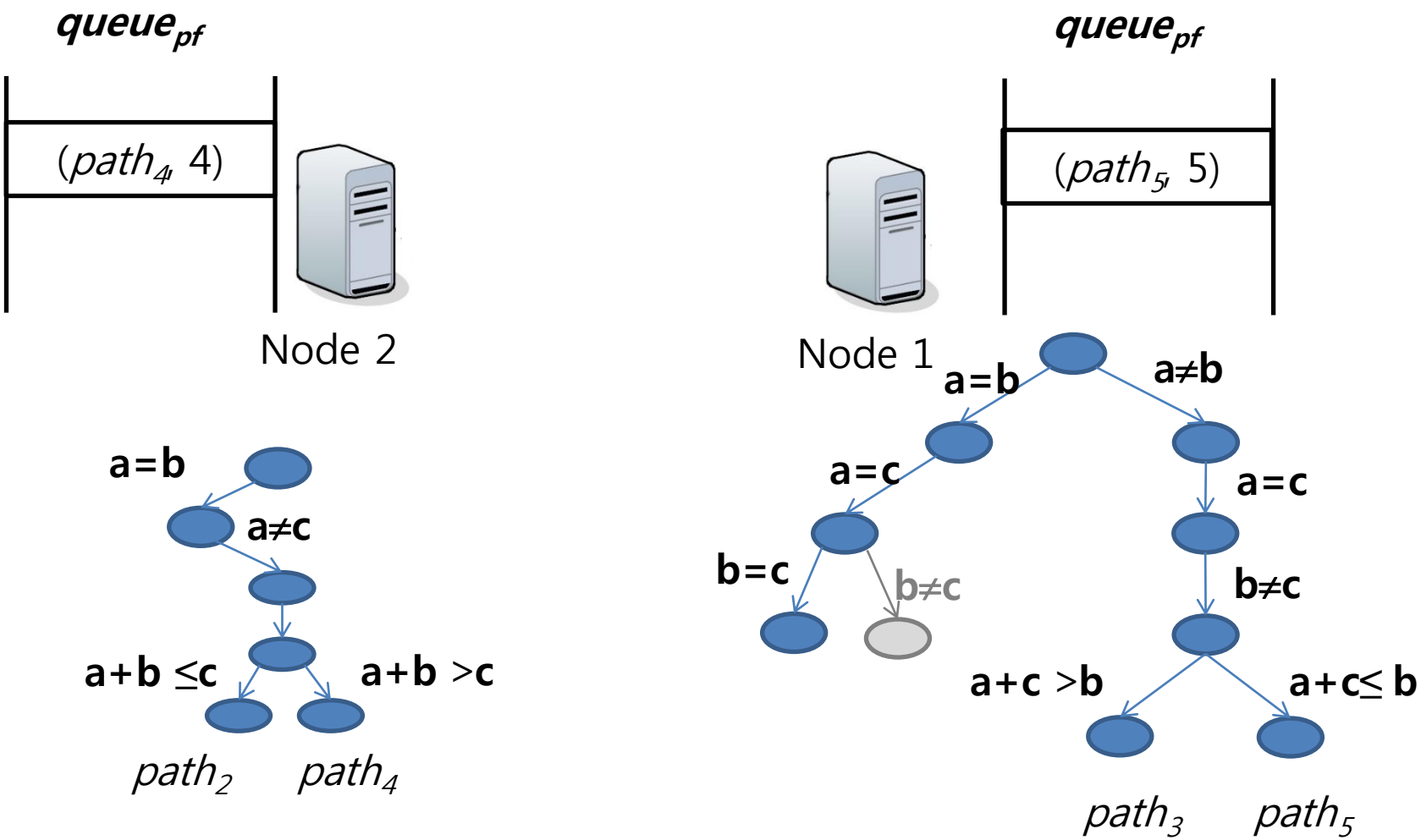


- Node 1 sends a new path formula with *neg_limit* to n

Receive a New Symbolic Path Formula



Continue Concolic Testing



Conclusion and Future Work

- Distributed concolic algorithm can decrease the time cost for concolic testing significantly
  - No redundant computation among computing nodes
- For larger scalability,
  - Fault tolerant features
  - Security problems
- For more practical application
  - Branch coverage
  - Utilizing external test cases