Design of a Modified Concolic Testing Algorithm with Smaller Constraints

Yavuz Koroglu and Alper Sen

Department of Computer Engineering, Bogazici University, Turkey
yavuz.koroglu@boun.edu.tr
alper.sen@boun.edu.tr

Abstract

Concolic testing is a well-known unit test generation technique. However, bottlenecks such as constraint solving prevents concolic testers to be used in large projects. We propose a modification to a standard concolic tester. Our modification makes more but smaller queries to the constraint solver, i.e. ignores some path conditions. We show that it is possible to reach the same branch coverage as the standard concolic tester while decreasing the burden on the constraint solver. We support our claims by testing several C programs with our method. Experimental results show that our modification improves runtime performance of the standard concolic tester in half of the experiments and results in more than 5x speedup when the unit under test has many infeasible paths.

1 Introduction

Software testing is an important but resource consuming part of software development process. According to a study in 2005, 79% of Microsoft developers write unit tests [11]. Therefore, automated unit testing is an important area of research. Concolic testing is an automated unit testing method first introduced in 2005 [8, 13]. Empirical study on concolic testing shows that scalability is an issue for concolic testing [11]. Therefore making concolic testing more scalable is an open research area.

We propose a modification to the input generation method of concolic testing, called Incremental Partial Path Constraints (IPPC). A standard concolic tester instruments the unit under test (UUT) to collect operations that either affect or depend on symbolic variables during a concrete execution. This sequence of operations is called an *execution trace*. A concolic tester symbolically reexecutes the execution trace to generate *path conditions* that solely depend on the symbolic variables. Then, the concolic tester negates the last path condition that is not negated before. At the final step, constraint solver is called only once with all path conditions to generate a new input vector. As noted in [8], solving more but smaller constraints against one large constraint (i.e. a constraint that contains large number of path conditions) is more efficient. Towards this goal, we modified the input generation step of a standard concolic tester such that we call the constraint solver multiple times, but using only a few path conditions at each invocation. Our experiments show that we can generate inputs that fall into the same equivalence class (i.e. inputs that force the program into generating the same execution trace) using fewer path conditions than a standard concolic tester. Although we pay the cost of more constraint solver calls on

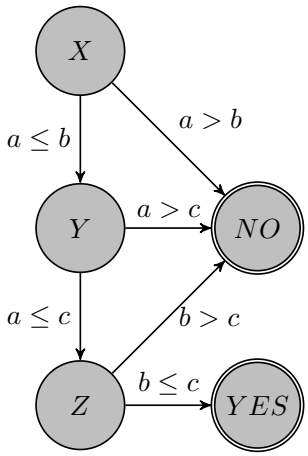


Figure 1: Control Flow Graph of *isSorted(a, b, c)*

average we show that our modification results in more than 5x speedup when UUT has many infeasible paths.

This paper makes two contributions. First, we propose an input generation strategy based on partial path constraints, called IPPC and incorporate our strategy into a standard concolic testing algorithm. Partial path constraints contain a subset of path conditions of an execution trace. Second, we implement our modification using the CREST tool [1], which is a concolic testing framework for C. Our results indicate that IPPC achieves the same branch coverage as the standard concolic tester while decreasing the burden on the constraint solver and leading to speedups.

We illustrate our approach on a small example in the next section. In section 4, we describe our approach in detail. We present our experimental results in section 5. We discuss the related work in section 3 and conclude in section 7.

2 Overview

In this section, we run both the standard concolic tester and IPPC on a small program unit which checks if three numbers are sorted or not, denoted by *isSorted(a, b, c)*. The control-flow graph of the unit under test (UUT) is given in Figure 1.

Figure 2 shows an execution of the standard concolic testing algorithm on *isSorted(a, b, c)*. A standard concolic tester starts from an arbitrary input ($[0,0,0]$ in this case), executes the UUT using the input values. The concolic tester collects the execution trace of this concrete execution and generates the path constraint $\pi_0 = [a \leq b, a \leq c, b \leq c]$. The last condition of π_0 is negated and we get $\pi_1 = [a \leq b, a \leq c, b > c]$. We invoke the constraint solver with π_1, denoted as CS(3), where 3 represents the number of path conditions in π_1. We get $[0,1,0]$ as the new input from the constraint solver. We execute the program with new inputs and collect the path constraint $\pi_2 = [a \leq b, a \leq c, b > c]$. Now, we negate the second condition of π_2 and remove all conditions after the second condition and get $\pi_3 = [a \leq b, a > c]$. We explain the reasons to exclude path conditions that come after the negated condition in Section 4.2. We get $[0,1,-1]$ by invoking CS(2). Concrete execution of the program results in $\pi_4 = [a \leq b, a > c]$. Then, we negate the only remaining path condition and get $\pi_5 = [a > b]$ after removing the last condition. CS(1) generates $[2,1,-1]$. The concolic tester stops generating test inputs after executing the program since the resulting path constraint $\pi_6 = [a > b]$ has no path condition that is not negated before. At the

$i_1 = [0,0,0]$	[initial inputs]
$\pi_0 = [a \leq b, a \leq c, b \leq c]$	[collected]
$\pi_1 = [a \leq b, a \leq c, b > c]$	[generated]
$i_2 = [0,1,0]$	[CS(3)]
$\pi_2 = [a \leq b, a \leq c, b > c]$	[collected]
$\pi_3 = [a \leq b, a > c]$	[generated]
$i_3 = [0,1,-1]$	[CS(2)]
$\pi_4 = [a \leq b, a > c]$	[collected]
$\pi_5 = [a > b]$	[generated]
$i_4 = [2,1,-1]$	[CS(1)]
$\pi_6 = [a > b]$	[collected]
STOP	[Path Exhaustion]

Figure 2: A Sample Execution of Standard Concolic Testing on *isSorted(a, b, c)*

end, we have invoked the constraint solver three times with an average path constraint length of two.

Figure 3 illustrates the execution of our IPPC algorithm on *isSorted(a, b ,c)*. In this case, each time we generate a new path constraint, we do not immediately call the constraint solver. We define the constraint obtained by executing the program a *full path constraint*. We generate a subset of the full path constraint, ϕ^0, which we call a *partial path constraint*. Partial path constraint ϕ^0 is initialized as an array which contains only the negated path condition. Partial path constraint generation phase is called *overapproximation*. We call the constraint solver with ϕ^0. In case the constraint solver does not generate an input which satisfies π, we add the remaining path conditions of π to ϕ^0 and generate larger path constraints such as ϕ^1, ϕ^2 etc. until the constraint solver generates an input that satisfies π. For our example, we only made three calls to the constraint solver with average constraint length of one.

3 Related Work

The idea of incremental constraint solving for concolic testing is as old as concolic testing itself and suggested along with CUTE, one of the first concolic testing tools [13]. However their incremental solving idea should not be confused with the one that we present in this paper. In CUTE, since they negate only one path condition, they keep the variables which are irrelevant to that path condition fixed. Therefore, they solve for variables which are only relevant to the negated path condition and decrease the burden on the constraint solver [13]. In our approach, CREST still does the incremental solving optimization of CUTE. On top of that, we also produce path contraints with fewer path conditions than a standard concolic tester.

Solving only a subset of path conditions instead of the whole path constraint is an approach recently used in finding integer overflows [15]. The motivation is supported by observing that many of the path conditions of an execution trace are irrelevant w.r.t. integer overflows. In our work, we make a more general assumption that some of the path conditions should be irrelevant to the execution trace itself. Experimental results support our motivational assumption.

All concolic testers require a *constraint solver*. Z3 and Yices are commonly used constraint solvers

$i_1 = [0,0,0]$	[initial inputs]
$\pi_0 = [a \leq b, a \leq c, b \leq c]$	[collected]
$\pi_1 = [a \leq b, a \leq c, b > c]$	[generated]
$\phi_1^0 = [b > c]$	[overapproximation]
$i_2 = [0,1,0]$	[CS(1)]
i_2 satisfies π_1	$CONTINUE$
$\pi_2 = [a \leq b, a \leq c, b > c]$	[collected]
$\pi_3 = [a \leq b, a > c]$	[generated]
$\phi_3^0 = [a > c]$	[overapproximation]
$i_3 = [1,1,0]$	[CS(1)]
i_3 satisfies π_3	$CONTINUE$
$\pi_4 = [a \leq b, a > c]$	[collected]
$\pi_5 = [a > b]$	[generated]
$\phi_5^0 = [a > b]$	[overapproximation]
$i_4 = [2,1,0]$	[CS(1)]
i_4 satisfies π_5	$CONTINUE$
$\pi_6 = [a > b]$	[collected]
STOP	[Path Exhaustion]

Figure 3: A Sample Execution of IPPC on *isSorted(a, b, c)*

[6, 3]. Details on the complexity of constraint solving can be found in [18]. Standard constraint solvers are not able to solve all types of constraints [11]. There are constraint solvers which solve nonlinear constraints [9] and constraint solvers which solve constraints involving high-precision rational numbers [16]. Yun et al. has modified CREST to use Z3 constraint solver in order to support nonlinear arithmetic [2]. We could easily implement IPPC modification on top CREST-z3, although plain CREST is sufficient for our experiments.

Concolic testing is not restricted to sequential programs. There are concolic testers which test concurrent software [7]. Therefore, IPPC can also be used to test concurrent units.

Many concolic testers (e.g. CUTE) use bounded depth first search (*bDFS*) instead of bounding the number of iterations as in IPPC and CREST [13, 12]. *bDFS* can be viewed as an attempt to keep the number of path conditions limited. We believe that our approach makes bounding the number of iterations more preferable to bounding the depth of the search as in *bDFS* since experiments show that IPPC already reduces the size of path constraints significantly.

Random branch selection (RND) is an approach introduced in CREST [1]. In random branch selection one negates each path condition of a path constraint with probability 0.5. A variant of this strategy negates only one path condition at random and calls the constraint solver with that path condition.

Control flow directed search (CFG) is a search strategy which guides the search using the static structure of the UUT. CFG assigns weights to edges of the control flow graph of the UUT and calculates distances to each unvisited branch. Then, CFG solves a path constraint which leads to the unvisited branch with the least distance to the current execution trace [1]. We compare our approach with both

CFG and RND.

A recent study on CREST proposes a new search strategy, called DYNASTY, which uses the control flow graph of the UUT to guide the search as in CFG technique [5]. They modify CREST as in our work and increase branch coverage with fewer iterations. Their approach is based on avoiding infeasible paths, whereas our approach decreases the penalty of hitting an infeasible path by using partial path constraints.

4 Incremental Partial Path Constraint (IPPC)

In this section, we describe IPPC in detail.

4.1 Background

We have previously stated that a concolic tester generates path conditions from an execution trace of the UUT. We define the following using this fact:

Definition 1. A *path condition* is a function of symbolic variables to $\{T, F\}$.

Definition 2. An *input vector* is an assignment to all symbolic variables of UUT.

Definition 3. A *full path constraint* is an array whose elements form a set of path conditions.

Definition 4. Any array whose elements form a subset of the elements of the full path constraint is called a *partial path constraint*.

Definition 5. An input vector is said to $satisfy$ a path constraint if and only if the input vector makes all conditions of the path constraint T.

Conjunction of all conditions in a partial path constraint is an overapproximation of conjunction of all conditions in the full path constraint. Therefore the process of generating partial path constraints from full path constraints is called an *overapproximation*.

Definition 6. An algorithm which takes a path constraint and returns an input vector which satisfies that constraint is called a *constraint solver*. If there exists no such input vector, it returns *null*.

We used the Yices constraint solver, which is the default solver in CREST [6, 1].

4.2 Algorithm Design of IPPC

We implemented IPPC on top of CREST's standard concolic testing algorithm (denoted by DFS) as shown in Algorithm 1. We initially give $nIterations \leftarrow 0$, $TestInput \leftarrow random\_input$ and $TestInputs \leftarrow \emptyset$. We can see from line 1 that the algorithm is bounded by the maximum number of iterations. At line 2, we check the input vector because constraint solver is assumed to return *null* if an input vector can not be generated, i.e. given path constraint is infeasible. We increment iterations only if the path constraint is satisfiable. At line 4, we execute UUT with $input$ and save the full path constraint of the execution trace as π. Lines 8 and 9 ensure that the algorithm performs a depth first search (DFS). We always negate the last unvisited condition of a path constraint. If there exists no path condition that is not negated before, we stop because all paths are executed. At line 11, constraintSolver takes a path constraint and returns a satisfying input vector. We don't terminate the solver, we use it in it's built-in incremental mode. Notice that we exclude the path conditions which come after the negated path condition. Any input vector which satisfies path conditions up to the negated path condition must generate the same execution trace

```
   input  : uut (unit under test)
            maxIterations
            testInput
            testInputs
            nIterations
   output: testInputs
 1 if nIterations < maxIterations then
 2     if testInput ≠ null then
 3         nIterations ←nIterations +1;
 4         π ←execute(uut, testInput);
 5         add(testInputs, testInput);
 6     end
 7     for i ←sizeOf(π)−1 to 0 do
 8         if !isNegatedBefore(π[i]) then
 9             setNegatedBefore(π[i]);
10             π[i] ← ¬π[i];
11             testInput ← constraintSolver({π[0]...π[i]});
12             return crest(uut, maxIterations, testInput, testInputs, nIterations);
13         end
14     end
15 end
16 return testInputs;
```

Algorithm 1: Concolic Testing Algorithm Implementing DFS with Maximum Iterations (CREST)

up to the occurrence of the negated path condition. Negated path condition will force the program to a different execution trace. So, path conditions coming after the negated condition is not a part of the new execution trace and therefore should be removed. If the unnecessary path conditions are not excluded, they may cause false path constraint infeasibilities. False infeasibilities force the concolic tester not to generate any input vector for the path constraint and therefore cause a decrease in test coverage.

IPPC is exactly the same as Algorithm 1 except we replace line 11 with *input* $\leftarrow$ IPPC_Solve $(UUT, \{\pi[0]...\pi[i]\}, \{\pi[i]\})$. We describe IPPC_Solve in Algorithm 2. We keep the solver running in incremental mode.

Line 1 of Algorithm 2 invokes the constraint solver with ϕ. If the constraint solver is unable to generate an input vector, we conclude that the full path constraint is also *infeasible* and return *null*. We prove the infeasiblity of a full path constraint given the infeasibility of a partial path constraint in Theorem 1. The for loop starting at line 5 checks if the input vector satisfies the full path constraint. If *input* satisfies π, we can return *input*. If not, we *learn* the unsatisfied path condition by adding it to ϕ and generate a new input vector. This process may be continued until $\phi = \pi$ in the worst case where either an input can be generated or π is infeasible given that the constraint solver is sound (if exists, returns a correct test input) and complete (terminates and is correct for all cases). This worst case condition is rare or non existent at least in our experiments.

4.3 Correctness and Completeness

To be able to produce readable proofs, we assume all boolean operations on a path constraint are performed on the conjunction of all conditions of the path constraint.

```
   input  : uut (unit under test)
            π (full path constraint)
            φ (partial path constraint)
   output: testInput
 1 testInput ← constraintSolver(φ);
 2 if testInput = null then
 3 |   return null;
 4 end
 5 for i ← 0 to sizeOf(π) − 1 do
 6 |   if !sat(testInput, π[i]) then
 7 |   |   append(φ, π[i]);
 8 |   |   return IPPC_Solve (uut, π, φ);
 9 |   end
10 end
11 return testInput;
```

Algorithm 2: IPPC_Solve

Theorem 1. *If a partial path constraint ϕ is unsatisfiable, then its full path constraint π is also unsatisfiable.*

Proof. Since ϕ is unsatisfiable, $\neg\phi$ is valid. Since ϕ is an overapproximation of π, $\pi \rightarrow \phi$ by definition. Therefore by modus tollens, $\neg\pi$ is valid. In other words, π is unsatisfiable. □

We now show the correctness of IPPC_Solve. We change only one line in Algorithm 1. Therefore, to prove the correctness of IPPC, we only need to ensure that for all path constraints, IPPC_Solve generates an input vector that has the same property as the input vector generated by the constraint solver. If such an input vector exists, a constraint solver always generates an input vector that satisfies the full path constraint.

Theorem 2. *(a) IPPC_Solve generates* null *whenever the constraint solver in Algorithm 1 at Line 11 generates* null*. (b) Otherwise IPPC_Solve always generates an input vector that satisfies the full path constraint.*

Proof. Proof of (a) is trivial due to lines 1-4 of Algorithm 2. We can see from lines 5-10, that Algorithm 2 would not stop until the input vector satisfies the full path constraint. Therefore proof of (b) is trivial as well. □

Theorem 3. *IPPC_Solve is correct and eventually terminates, i.e. IPPC_Solve is complete.*

Proof. IPPC_Solve is correct by Theorem 2. The second part of the proof is as follows. At any time, partial path constraint ϕ is either (a) unsatisfiable, or an input vector i is generated. If an input vector i is generated, either (b) i satisfies π, or (c) we add a new path condition of full path constraint to ϕ. If (a), IPPC_Solve returns *null* and terminates. If (b), IPPC_Solve returns i and terminates. If (c) happens $N-1$ times, where N denotes the number of path conditions in π, $\phi \leftrightarrow \pi$ must hold. In other words, we build up the partial path constraint up to the full path constraint. In that case, the constraint solver must either generate *null* or generate a satisfying input vector, therefore the algorithm terminates. □

Table 1: List of Benchmarks

UUT	KLOC	#vars
gcd	0.05	2
bsort	0.05	30
sqrt	0.06	1
prime	0.1	1
factor	0.2	1
replace	0.5	20
ptokens	0.6	40
grep	15	10

Table 2: Experimental Results (Best results are highlighted)

UUT	#Itr	Total CS Calls				Avg Const. Size				Branch Coverage (%)				Time Elapsed (sec)			
		DFS	CFG	RND	IPPC	DFS	CFG	RND	IPPC	DFS	CFG	RND	IPPC	DFS	CFG	RND	IPPC
gcd	1000	5681	866	1105	11004	161.0	4.1	2.9	1.9	100	100	100	100	22.6	5.7	**3.9**	7.5
	5000	25801	4364	5653	64844	188.6	5.7	2.8	1.9	100	100	100	100	119.5	24.8	**21.6**	34.7
	10000	47367	8761	11287	120678	206.3	5.9	2.8	1.9	100	100	100	100	234.5	48.7	**45.7**	125.5
bsort	100	117	53	104	1346	145.7	148.5	88.9	10.3	100	100	100	100	1.1	**0.8**	0.9	1.3
	500	711	284	537	10306	308.9	165.3	115.5	12.7	100	100	100	100	7.6	5.0	**4.8**	9.2
	1000	1462	571	1063	25023	342.5	165.0	132.6	14.3	100	100	100	100	15.5	**8.7**	9.5	21.9
sqrt	1000	1010	980	1064	1006	17.2	2.3	1.8	1.0	94.4	94.4	94.4	94.4	4.3	4.5	4.3	**3.9**
	5000	5001	4894	5372	5006	24.0	2.3	1.8	1.0	94.4	94.4	94.4	94.4	30.1	23.1	22.2	**20.3**
	10000	10010	9820	10732	10014	22.7	2.3	1.8	1.0	94.4	94.4	94.4	94.4	45.9	42.0	42.2	**39.1**
prime	100	1275	92	111	132	103.3	7.5	4.8	1.2	**92.5**	85	62.5	**92.5**	2.3	0.5	0.5	**0.4**
	175	3946	165	212	253	186.6	8.0	4.0	1.2	**92.5**	90	70	**92.5**	10.1	1.0	1.1	**0.8**
	250	4166	236	287	1688	189.6	7.2	4.1	1.8	**92.5**	92.5	82.5	**92.5**	13.5	1.6	1.6	**1.5**
factor	50	2678	80	110	7693	222.1	2.9	4.0	1.9	**94.2**	**94.2**	76.9	**94.2**	8.6	**0.2**	**0.2**	1.5
	75	5734	111	145	25255	249.7	3.4	4.7	1.9	**94.2**	**94.2**	83.2	**94.2**	23.5	**0.3**	**0.3**	4.6
	100	11157	148	243	34218	310.9	2.8	16.0	1.9	**94.2**	**94.2**	89.4	**94.2**	120.3	0.6	**0.5**	6.4
replace	1000	1129	1024	1024	6847	16.6	25.9	31.7	3.8	85.2	**88.7**	85.2	85.2	**4.1**	4.5	**4.1**	5.8
	5000	5623	5193	5167	43695	21.5	28.4	32.4	4.8	88.7	**90.8**	88.7	88.7	**22.0**	23.8	22.4	31.4
	10000	10936	10270	10301	103590	24.5	28.0	33.0	5.6	88.7	**90.8**	90.8	88.7	**43.1**	45.5	44.6	66.9
ptokens	1000	1181	1355	1319	38415	591.0	136.8	102.2	29.8	**85.1**	79.8	79.8	**85.1**	30.0	9.4	**8.6**	18.5
	5000	6394	6976	6739	253829	1949.9	148.7	103.2	37.5	**85.1**	84.4	84.9	**85.1**	212.4	48.4	**47.7**	106.1
	10000	13394	13838	13510	432262	2241.6	163.7	99.7	30.5	**91.5**	89.6	88.9	**91.5**	321.7	106.8	**91.2**	210.8
grep	100	100	N/A	121	436	59.0	N/A	362.0	3.2	16.7	N/A	16.7	16.7	**0.3**	N/A	0.4	0.4
	220	269	N/A	272	1671	178.4	N/A	332.2	4.7	16.7	N/A	16.7	16.7	**0.9**	N/A	**0.9**	1.0
	340	439	N/A	451	3671	324.8	N/A	350.1	8.3	16.7	N/A	16.7	16.7	1.6	N/A	**1.5**	1.9

5 Experimental Study

We implemented IPPC on top of CREST tool. Our implementation of IPPC and experimental results are available online [10]. CREST implements the standard concolic testing algorithm (DFS) and two different improvements, denoted by CFG and RND. We compared IPPC with these three techniques. Techniques we used in experiments are as follows.

1. **DFS:** Standard concolic testing algorithm.
2. **CFG:** Control flow directed testing algorithm.
3. **RND:** Random branch testing heuristic on top of the standard concolic testing algorithm.
4. **IPPC:** Our Incremental Partial Path Constraint algorithm.

Brief explanations of CFG and RND algorithms are given in Section 3.

We carried out the experiments on a virtual Linux guest with 1024MB memory and one CPU hosted by a MacBook Pro with an Intel Core i7 2.9 GHz GPU and 8GB Memory. We collected the following information for each experiment:

1. **Time elapsed:** Time spent to test the UUT.
2. **Total CS Calls:** Total number of constraint solver calls made by the concolic tester.
3. **Avg Const. Size:** Average size of path constraints solved by the constraint solver.
4. **Branch Coverage:** Measurement of total branch coverage of UUT.

Concolic testing involves a degree of randomness due to the randomness of initial inputs. Therefore we performed 10 executions of each experiment and took average values of each measure. All of our experimental results can be found in Table 2.

Table 1 shows the program units we used in our experiments. Column *KLOC* represents the lines of code measured in thousands. Column *#vars* represents the number of symbolic variables.

We implemented five small benchmarks, *gcd*, *bsort*, *sqrt*, *prime* and *factor* to conduct our initial experiments. *gcd* implements the binary greatest common divisor algorithm. We downloaded and modified the bubble sort algorithm *bsort*, which sorts a given array of integers [17]. *sqrt* takes the floor of the square root of a given integer. *prime* decides whether a given integer is prime or not. *factor* is an integer factorization algorithm. *replace* and *grep* are benchmarks that come with CREST framework and used in several research studies on concolic testing [1, 5, 14]. *ptokens* is the printtokens benchmark available at Software Infrastructure Repository (SIR) [4]. *ptokens* tokenizes the given string according to a grammar. For reasons described in Section 6, our implementation of *gcd*, *prime* and *factor* do not contain any bitwise masking or modulo operations. Instead, we decide divisibility via only subtraction and comparison operations. Also, *sqrt* does not use any floating point operations since CREST has no symbolic equivalent of floating point variables.

In our experiments, we used programs in different sizes. None of the given programs are too large in size so they can be tested in reasonable time without getting into scalability issues. In our experimental set, we argue that the structure of UUT is related to the performance of the testers that we use rather than the sizes of the programs. We observed that small programs such as *factor* can have very long runtimes (120 sec). We argue that IPPC will perform well not when the program is small or large, but when the program contains many infeasible paths. Although being small, *prime* and *factor* contain many infeasible paths. Although being large, *grep* and *replace* did not contain many infeasible paths.

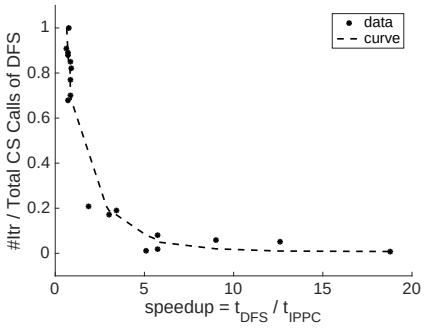
We show all experimental results in Table 2. CFG method is not applicable to *grep* due to a bug in CREST, therefore we used *N/A* to denote the results we could not observe. We used 1000, 5000 and 10000 as the maximum number of iterations (*#Itr*) in experiments. We decreased *#Itr* for *prime*, *factor*, *bsort* and *grep*, since those units have few distinct execution traces. When there are few distinct execution traces and *#Itr* is too high, DFS and IPPC are able to stop before completing *#Itr* iterations, since they check if all execution traces are explored or not. However, CFG and RND have no such stopping condition and iterate *#Itr* times. So, if we kept *#Itr* high for *prime*, *factor*, *bsort* and *grep*, it would unfairly result in bad runtimes for CFG and RND.

We show the best algorithms for each benchmark in Table 3 where *All Equal* means all techniques are equally well. The column denoted as *by Runtime First* compares techniques by runtime first, if techniques have similar runtimes, compares branch coverages. The last column compares techniques by coverage first. DFS does not perform well in both columns. In terms of runtimes RND is the best and IPPC is the second. In terms of coverage CFG and IPPC are the best. Hence, IPPC performs well in both categories.

We next compare IPPC with DFS in more detail since IPPC is derived from DFS. We observe from Table 4 that there exists a correlation between Avg Const. Size of DFS / Avg Const. Size of IPPC and speedup of IPPC over DFS. When the gap between the constraint sizes of DFS and IPPC increases, the speedup of IPPC over DFS increases with the slight exception of *grep*. Similarly, Figure 4 shows that whenever the number of constraint solver calls (Total CS Calls) of DFS is close to the maximum number of iterations (#Itr), DFS works faster. If DFS makes many more calls than the number of

Table 3: Best Concolic Testers

UUT	by Runtime First	by Coverage First
gcd	RND	RND
bsort	CFG	CFG
sqrt	IPPC	IPPC&DFS
prime	IPPC	IPPC&DFS
factor	RND	CFG
replace	DFS	CFG
ptokens	RND	IPPC
grep	All Equal	All Equal



UUT	Avg Const. Size Ratio (DFS / IPPC)	Speedup (t_{DFS}/t_{IPPC})
replace	4.4	0.6x
bsort	20.8	0.79x
sqrt	21.3	1.25x
grep	31.8	0.83x
ptokens	48.4	1.7x
gcd	97.5	2.77x
prime	115.6	9.1x
factor	137.3	9.8x

Figure 4 & Table 4: Relationship between Speedup and #Infeasible Constraints (Figure 4) and IPPC Speedup over DFS (Table 4).

iterations, IPPC works faster. Normally, DFS is expected to call constraint solver exactly once for each iteration. DFS makes more than one constraint solver call for an iteration only if the generated path constraint for that iteration is infeasible. In that case, DFS changes the path constraint and calls constraint solver repeatedly until a feasible path is found. We believe *factor* has the largest number of infeasibilities since DFS makes many constraint solver calls for few iterations. Figure 4 shows that the speedup of IPPC over DFS is above 5x when DFS generates four or more infeasible path constraints for each feasible path constraint (i.e. DFS makes more than five constraint solver calls for each iteration, #Itr/Total CS Calls of DFS < 0.2). Therefore, IPPC finds infeasibilities faster for all benchmarks.

In all experiments, the largest path constraint produced by IPPC had a length of 157, whereas the largest path constraints of DFS, CFG and RND had lengths of 2922, 1603 and 1391, respectively. We conclude that we eliminated the need for solving large path constraints to generate test inputs while keeping the runtimes fast and coverage high using IPPC.

We get exactly the same coverage results for both IPPC and DFS. We expect this since both algorithms perform a depth-first search. We argue that the coverage results being similar indicates that IPPC correctly does DFS on the UUT while improving the performance.

IPPC considers constraints that are 10x smaller than other techniques. We believe it is because that IPPC finds infeasibilities early, since most infeasibilities arise from a combination of few conditions in the constraint.

6 Threads to Validity

We assume that the constraint solver is sound, i.e. the constraint solver can find an input vector whenever there exists an input vector which satisfies the path constraint. In general, this assumption is not valid. We know that the constraint solver used in CREST, Yices [6, 11], does not find solutions for nonlinear path conditions (e.g. $x_2 x_1 < 12$). We also know that CREST may not be able to solve conditions involving modulo operation and bitwise masking [11]. All path conditions generated in our experiments are linear, in other words they can be written as $k \bowtie \sum_{i=1}^{N} c_i x_i$ where $\bowtie \in \{<, \leq, =, \neq, >, \geq\}$, $k \in \mathrm{R}$, $\forall i \in \{1, ..., N\}, c_i \in \mathrm{R}$ and N: total number of symbolic variables). All path conditions in our experiments are also free of modulo and bit masking operations. Hence, we safely assume that the constraint solver is sound in our environment. We also do not use any floating point arithmetic.

The UUT can have intermediary variables calculated from symbolic variables and therefore can have path conditions on those intermediary variables. All path conditions that CREST returns are on the initial symbolic variables. Therefore, even if all branch conditions in the code may seem trivial, CREST may fail to generate correct inputs if variables are nonlinear (e.g. multiplication of two symbolic variables).

We assume the UUT to be *sequential* and *deterministic*, i.e. if an input vector i produces an execution trace e, i will always produce e for this UUT. However, for example a process which depends on random numbers could violate this assumption. We carefully chose the experiments so that we never violate these basic assumptions.

We assume the UUT is *terminating*, since if UUT halts, so does the tester as well. The test cases we chose and the test cases in previous work are all terminating.

We report runtimes that we acquired from a virtual environment. We got similar results on an host machine as well.

It is possible that IPPC may learn partial path constraints up to a point that they become full path constraints. So in the worst case, a standard concolic tester is more efficient than IPPC. However, our experimental results show that we require only a small portion of the full path constraint to generate input vectors belonging to the same equivalence class, i.e. input vectors which generate same execution traces when given to UUT.

7 Conclusions and Future Work

In this paper, we designed a constraint solving strategy, called Incremental Partial Path Constraints (IPPC), which eliminates the need for solving large constraints to generate unit tests. We compared our design with other concolic testing algorithms in experiments. We observed that when there are many infeasible path constraints in a program, IPPC has more than 5x speedup over a standard concolic tester. We significantly reduce the number of path conditions required to generate an input vector and show that IPPC dominates other techniques in two of eight cases and has the best coverage levels in three of eight cases. We show that it is possible to reach high branch coverage while decreasing the burden on the constraint solver.

We believe that our constraint solving strategy contains room for improvement and shows promise for future work. The performance of IPPC on nonlinear path constraints (i.e. path constraints that contain at least one nonlinear path condition) or on concurrent software is an important question. Also, it is possible to further improve IPPC by input vector caching, i.e. trying the previously generated input vectors first to avoid calling constraint solver. We believe that our work sufficiently motivates further research on constraint solving strategies involving partial path constraints.

References

[1] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, 2008.

[2] CREST-z3, https://github.com/heechul/crest-z3.

[3] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 2008.

[4] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[5] Yu Dong, Mengxiang Lin, Kai Yu, Yi Zhou, and Yinli Chen. Achieving high branch coverage with fewer paths. In *Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, COMPSACW '11, 2011.

[6] Bruno Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer International Publishing, 2014.

[7] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, 2013.

[8] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005.

[9] Pierluigi Nuzzo, Alberto Puggelli, Sanjit A. Seshia, and Alberto Sangiovanni-Vincentelli. CalCS: SMT solving for non-linear convex constraints. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, 2010.

[10] `http://bitbucket.org/yavuzkoroglu/crest-ppc`.

[11] Xiao Qu and Brian Robinson. A case study of concolic testing tools and their limitations. In *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, ESEM '11, 2011.

[12] Koushik Sen. *Scalable Automated Methods for Dynamic Program Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, 2006. Available at `http://srl.cs.berkeley.edu/~ksen/doku.php`.

[13] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, 2005.

[14] Hyunmin Seo and Sunghun Kim. How we get there: A context-guided search strategy in concolic testing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, 2014.

[15] Stelios Sidiroglou-Douskos, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin C. Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, pages 473–486, 2015.

[16] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Păsăreanu. CORAL: Solving complex constraints for symbolic pathfinder. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, 2011.

[17] BubbleSort, `http://www.programmingsimplified.com/c/source-code/c-program-bubble-sort`.

[18] Edward P. K. Tsang. *Foundations of constraint satisfaction.* Computation in cognitive science. Academic Press, 1993.