Solving Complex Path Conditions through Heuristic Search on Induced Polytopes

Peter Dinges
University of Illinois
Urbana–Champaign, USA
pdinges@acm.org

Gul Agha
University of Illinois
Urbana–Champaign, USA
agha@illinois.edu

ABSTRACT

Test input generators using symbolic and concolic execution must solve path conditions to systematically explore a program and generate high coverage tests. However, path conditions may contain complicated arithmetic constraints that are infeasible to solve: a solver may be unavailable, solving may be computationally intractable, or the constraints may be undecidable. Existing test generators either simplify such constraints with concrete values to make them decidable, or rely on strong but incomplete constraint solvers. Unfortunately, simplification yields coarse approximations whose solutions rarely satisfy the original constraint. Moreover, constraint solvers cannot handle calls to native library methods. We show how a simple combination of linear constraint solving and heuristic search can overcome these limitations. We call this technique *Concolic Walk.* On a corpus of 11 programs, an instance of our Concolic Walk algorithm using tabu search generates tests with two- to three-times higher coverage than simplification-based tools while being up to five-times as efficient. Furthermore, our algorithm improves the coverage of two state-of-the-art test generators by 21% and 32%. Other concolic and symbolic testing tools could integrate our algorithm to solve complex path conditions without having to sacrifice any of their own capabilities, leading to higher overall coverage.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Symbolic execution*

General Terms

Algorithms

Keywords

Concolic Testing; Local Search; Non-Linear Constraints

FSE'14, November 16–21, 2014, Hong Kong, China
ACM 978-1-4503-3056-5/14/11
http://dx.doi.org/10.1145/2635868.2635889

1. INTRODUCTION

Thorough testing of programs is crucial, but time consuming and expensive [1]. Automatic test input generators can simplify testing by supplying values that trigger different behaviors in the program, including hidden corner-cases. We are interested in generating inputs for programs whose control-flow depends on complex arithmetic operations such as non-linear and trigonometric functions, for example the TSAFE[1] program used to prevent airplane collisions. Such programs, which are common in the domain of cyber-physical systems, pose major challenges for test input generators based on symbolic and concolic execution.

Symbolic and concolic test generators [43, 19, 36, 8, 12, 7, 41] attain their strength—high coverage—by picking a distinct path in the program for each round of input generation. They do this by characterizing a fresh path's branch conditions as a set of symbolic constraints, and solving this *path condition* to obtain concrete inputs that drive the program down the path. A central problem in this approach, as exemplified in TSAFE, is translating the arithmetic constraints of the path condition into the theory of the underlying solver. The difficulties in translating are:

(1) Non-linear integer constraints often make it infeasible to solve the path condition. Such constraints are even undecidable in general [11].

(2) Path conditions can contain calls to (uninterpreted) library methods, such as trigonometric functions, about which the solver cannot reason [44].

Classic concolic testing mitigates these problems by replacing troublesome symbolic terms with their concrete values [19, 36, 41, 34]. This reduction of symbolic reasoning to simple evaluation allows concolic testing to explore paths whose branch conditions lie outside the solver's theory. However, such simplification restricts the search space rather arbitrarily. The result is that it can find a solution only in a few cases (see section 5).

Another mitigation strategy for symbolic testing is relying on a domain-specific solver that can interpret all occurring arithmetic operations and library methods [37, 6]. While performing well within the target domain, the approach requires a solver extension for every new operation. Existing solvers based on heuristic search furthermore ignore the symbolic structure of the path condition.

The *Concolic Walk* algorithm introduced in this paper solves path conditions through a novel blend of symbolic

[1]Tactical Separation Assisted Flight Environment

reasoning, concrete evaluation, and heuristic search, which overcomes the limitations of previous approaches. The algorithm is based on a geometric interpretation of the problem: We regard assignments of values to the variables appearing in a path condition as points in a *valuation space*. Intuitively thinking of the valuation space as $\mathbb{R}^n$, we find a solution point to a path condition by combining the following ideas:

- The solutions of the linear constraints in the path condition define a contiguous convex region in the space (a polytope).[2] All solutions to the whole path condition must lie within this polytope.

- All terms appearing in the constraints that comprise the path condition, including library methods, can be evaluated. Hence, we can assign each constraint an evaluation-based fitness function that measures *how close* a valuation point is to satisfying the constraint. This allows us to find solutions through heuristic search.

- Many non-linear terms are at least piece-wise continuous. Numerical optimization techniques akin to Newton's method can thus accelerate the solution search.

In particular, our algorithm (1) splits the path condition into linear and non-linear constraints; (2) finds a point in the polytope induced by the linear constraints with an off-the-shelf solver; and then, (3) starting from this point, uses *adaptive search* [9] within the polytope, guided by the constraint fitness functions, to find a solution to the whole path condition.

This paper contains the following research contributions:

- We introduce the *Concolic Walk* (CW) algorithm, a novel combination of symbolic reasoning and heuristic search for solving complex arithmetic path conditions (section 3). The algorithm is sound and complete for linear constraints and supports non-linear constraints and calls to native library methods.

- We evaluate an implementation (section 4) of the CW algorithm on a corpus of 11 programs whose path conditions include mostly non-linear constraints. We show (section 5) that the algorithm (1) generates tests with two- to three-times higher coverage than simplification-based tools while being up to five-times as efficient; and (2) considerably improves the coverage of state-of-the-art test generators such as Pex [41].

In its current form, the CW algorithm is limited to solving arithmetic constraints in path conditions. However, it can be combined with approaches for solving pointer constraints [36, 26] or heuristic–concolic object generation [23] to fill this gap. For purpose of exposition, we limit the discussion to test input generation, but the CW algorithm also applies to other uses of concolic execution such as regression testing [24, 40], specification mining [10], and property checking [22, 3, 4].

2. MOTIVATION

A common goal for test input generators is producing inputs that cover as many different execution paths as possible. While random inputs are fast to generate, they mostly cover

[2]We ignore "not equal" constraints to convey the essential idea of our algorithm.

```
 1 static void example1(int x, int y) {
 2   int z = x * y;                                // non−linear operation
 3   if (x == z)
 4     if (x > 2)
 5       assert false : "Found error";
 6 }
 7
 8 static void example2(double u) {
 9   // work with the binary representation of u
10   long v = Double.doubleToRawLongBits(u);      // native method
11   long w = v & 0xff000;
12   if (w > 0)
13     assert false : "Found error";
14 }
```

Figure 1: Example Java methods with complex path conditions. In method example1, the path to the error (line 5) has the non-linear path condition $x = x \cdot y \wedge x > 2$. Neither jCUTE [36] nor mixed concrete–symbolic solving [34] discover input values that satisfy this path condition. In method example2, the path condition for the error (line 13) contains a call to an uninterpreted library method. Neither SPF-CORAL [37] nor Pex [41] (for the C# version) discover input values that satisfy the path condition.

the *common* paths in a program. Narrow branch conditions, such as $x = x \cdot y$, are unlikely to be met by random values for x and y. Instead, most generated inputs will execute the same path $x \neq x \cdot y$, resulting in repeated tests of the same program behavior.

To cover narrow branches and avoid repetition, symbolic and *conc*rete–symb*olic* (concolic) input generation [43, 19, 36, 8, 12, 7, 41] take a more systematic approach. They first collect all branch conditions along a target path, picking a single clause from disjunctive conditions, and represent them as conjunction of symbolic constraints. Then, they solve this *path condition* to obtain concrete inputs. The path condition characterizes the set of all concrete inputs that lead the program down this path. A solution thus satisfies all branch conditions, including narrow ones. Furthermore, solving a different path condition every time prevents repetitions that may occur with random inputs.

For example, assume we want to find an input for the example1 method in Figure 1 that covers the path along the statements in the lines 3, 4, and 5. To drive the execution down this path, x and y must satisfy the path condition $x = x \cdot y \wedge x > 2$. Given a suitable decision procedure, we can solve the path condition to obtain, for instance, the concrete inputs x=3 and y=1.

Solver Limitations and Mitigation Strategies

Unfortunately, a complete symbolic decision procedure for general non-linear integer constraints cannot exist [11]. Furthermore, a decision procedure can typically only find solutions if it has an interpretation for all appearing operations. If a path condition contains arbitrary integer arithmetic or, importantly, calls to opaque library methods—like the native *cos* method in Java—, then finding a solution is hard [44].

A common approach to work around the solver limitations is to simplify (under-approximate) the path condition: parts of the path condition that the solver cannot handle are first executed on concrete inputs and then replaced with

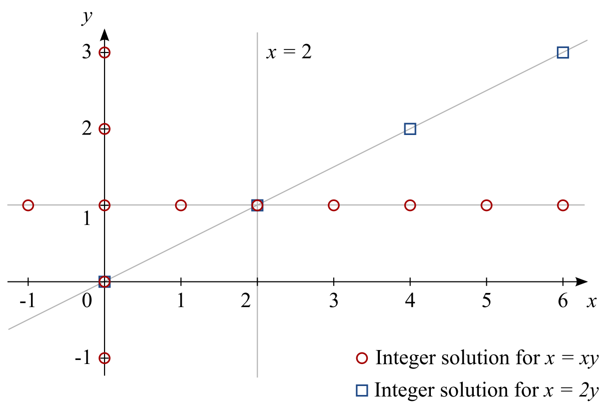

Figure 2: Solutions of the non-linear equation $x = x{\cdot}y$ (circles) and its linearization $x = 2 \cdot y$ (squares). No solution for $x = 2 \cdot y$, satisfies the path condition $x = x \cdot y \wedge x > 2$, which shows the danger of blindly simplifying path conditions.

the concrete results. Simplification has been applied while constructing the path condition, and while solving it. Classic concolic test generators such as jCUTE and Pex simplify at construction time. For example, jCUTE relies on a linear constraint solver. When building the path condition for the path 2, 3, 4, 5 in Figure 1, it replaces the non-linear expression $x \cdot y$ with $2 \cdot y$ if $x = 2$ when the expression is added. This yields the path condition $x = 2{\cdot}y \wedge x > 2 \wedge x = 2$ if we include the constraint $x = 2$ that is required to make the simplification sound [18, 34]. In contrast, *mixed concrete–symbolic solving* [34] simplifies at solution time. To solve a path condition, mixed solving splits it into resolvable and *complex* constraints, solves the resolvable ones directly, and uses the solution to simplify and concretely execute the complex constraints. The execution results, in turn, serve to simplify the complex constraints.

Figure 2 shows how simplification produces bad approximations. The simplified path condition of above example is unsatisfiable because of a bad approximation due to a random choice of x. Likewise, mixed solving fails to cover the path because the only feedback from the concrete execution to the constraint solver is ruling out non-working values, which is often insufficient to find a solution (section 5). Observe that both simplification techniques are problematic because of *blind commitment* to concrete values, regardless of other constraints on the variables.

Stronger Solvers

The number of such bad approximations decreases as the strength of the solver increases. The Pex concolic test generator [41], for example, relies on the Z3 SMT[3] solver [31] to find inputs that satisfy a path condition. Z3 supports (some) non-linear integer operations in its constraints, and hence Pex discovers the error in line 5 of Figure 1. Likewise, SPF-CORAL, a combination of the *Symbolic PathFinder* (SPF) symbolic execution tool [33, 35] and a constraint solver based on heuristic search (CORAL [37, 6]), discovers the error.

While this extends the domain of programs that can be handled and enables coverage of more paths than before, it does not fully fix the interpretation problem. Specifically, programs may call hitherto unknown library methods. For example, an implementation of trigonometric functions converts floating-point numbers to bit-vectors, as exemplified in Figure 1. Such methods would require a solver extension, arguably a maintenance nightmare. Other interactions, such as database queries, are even harder to integrate.

[3]Satisfiability Modulo Theories

3. ALGORITHM

The *Concolic Walk* (CW) algorithm for solving path conditions addresses the aforementioned challenges of relying on decision procedures (section 2). Specifically, it treats linear and non-linear constraints differently: to solve the linear constraints, it uses an off-the-shelf solver; to solve the non-linear constraints, it uses heuristic search based on concrete execution (evaluation). Using concrete execution allows the algorithm to handle opaque library methods without further extension. At the same time, the algorithm never simplifies or approximates the path condition; this avoids blind commitment to concrete values.

3.1 Synopsis

The algorithm distinguishes between linear and non-linear constraints because linear constraint systems are decidable and efficient solvers exist. Furthermore, linear constraints have a useful geometric interpretation: Assume that we relax the domain of each variable in the path condition to $\mathbb{R}$. Then we can regard an assignment of values to each variable as point in a *valuation space*, which corresponds to $\mathbb{R}^n$. In the valuation space, the solutions of a linear constraint form a half-space; a conjunction of linear constraints therefore describes an intersection of half-spaces, which is a convex (hence, contiguous) region—a convex polytope.[4] In Figure 2, the polytope is the region right of the line $x = 2$.

All variable assignments that are global solutions to the whole path condition must lie within the polytope because the points outside violate the linear constraints. To find a global solution, the CW algorithm thus picks points in the polytope and evaluates the non-linear constraints on them to check whether these are satisfied, too.

An efficient way to pick random points in the polytope is a random walk. However, if global solutions are sparse within the polytope, a random walk has slim chances of discovering one. The algorithm therefore combines the random walk with a search heuristic that guides the walk towards promising regions. For this, each non-linear constraint is assigned a fitness function—based on evaluating the terms in the constraint—that measures how close the current point is to a solution of the constraint.

From the many meta-heuristics (simulated annealing, genetic algorithms, etc.), we chose the *adaptive search* variant [9] of tabu-search [16, 17] for its ease of adding a non-random neighbor-picking strategy (see below). In each iteration, adaptive search picks the variable that appears in the most violated constraints and examines neighbor points that differ only in this variable. A variable becomes *tabu* for several iterations if changing its value failed to yield a better neighbor. The search moves towards the fittest neighbor, like hill-climbing, but escapes local minima with the help of the tabu mechanism.

[4]Recall that each clause of an *or* branch condition is treated as a separate path. The path condition is therefore a pure conjunction of constraints. Our interpretation ignores "not equal" constraints; these cut slices out of the polytope.

$$
\begin{aligned}
\langle \text{Exp} \rangle &::= \langle \text{Var} \rangle \mid \langle \text{Lit} \rangle \mid \langle \text{Call} \rangle \mid \langle \text{Exp} \rangle \circ \langle \text{Exp} \rangle \mid (\langle \text{Exp} \rangle) \\
\langle \text{Call} \rangle &::= \langle \text{Fun} \rangle() \mid \langle \text{Fun} \rangle(\ \langle \text{Var} \rangle\ (,\ \langle \text{Var} \rangle)^\star\ ) \\
\langle \text{Cond} \rangle &::= \langle \text{Var} \rangle \sim \langle \text{Var} \rangle \\
\langle \text{Stmt} \rangle &::= \langle \text{Var} \rangle = \langle \text{Exp} \rangle \\
&\quad \mid \textsf{if}\ (\ \langle \text{Cond} \rangle\ )\ \langle \text{Stmt} \rangle\ \textsf{else}\ \langle \text{Stmt} \rangle \\
&\quad \mid \textsf{while}\ (\ \langle \text{Cond} \rangle\ )\ \langle \text{Stmt} \rangle \\
&\quad \mid \{\ \langle \text{Stmt} \rangle\ \big(;\ \langle \text{Stmt} \rangle\big)^\star\ \} \\
\langle \text{Prog} \rangle &::= \langle \text{Stmt} \rangle\ \big(;\ \langle \text{Stmt} \rangle\big)^\star
\end{aligned}
$$

Figure 3: Syntax of the example language, with variables Var, literals Lit, function symbols Fun, binary operations $\circ \in \{+, -, \cdot, /, \text{mod}\}$, and relations $\sim \in \{<, \leq, \geq, >, =, \neq\}$.

In addition to picking random neighbors, the CW algorithm furthermore tries to exploit (piece-wise) continuity of non-linear terms. Given the values of the fitness function at the current point and a neighbor, it estimates a third point where the constraint *would* be satisfied, were it linear. Such estimation equals a single step of the bisection method for numerical zero-finding and can accelerate the search.

Example

Consider the partial plot of the valuation space for the variables x and y in Figure 2. The path condition $x = x \cdot y \wedge x > 2$, consists of the linear constraint $x > 2$ and the non-linear constraint $x = x \cdot y$. The linear constraint $x > 2$ describes an unbounded convex polytope: the half-plane right of the line $x = 2$. All feasible solutions must be contained within this polytope. To discover a solution to the non-linear constraint, we start a random walk at an arbitrary point in the polytope, for instance $(x, y) = (4, -1)$. Modifying x, we randomly generate the neighbors $(3, -1)$ and $(8, -1)$, choosing $(3, -1)$ because it is closer to satisfying the equation $x = x \cdot y$. Modifying x again does not yield a better neighbor; all of these lie outside the polytope. Thus, x is marked as tabu and the next iteration modifies y. Aside from a random neighbor $(3, 2)$, we estimate a zero for the linear parameterization of $x - x \cdot y$ on the line $(3, -1)$–$(3, 2)$, which yields the solution $(3, 1)$.

3.2 Terms and Definitions

Before formalizing the CW algorithm in the next section, we briefly define the terms used. To simplify the exposition, we describe the algorithm for a small imperative programming language whose Java-like syntax is shown in Figure 3. The algorithm itself is independent of the target language and applies to any language for which the CEVAL and SEVAL functions can be defined accordingly.

The example language supports the basic imperative statements. It lacks support for function definitions to keep matters simple, but expressions may contain calls to opaque library functions that have been defined elsewhere. The only data types are integers and real numbers because of our focus on solving arithmetic constraints.

Concrete Execution

An evaluation function CEVAL models the concrete operational small-step semantics of the language. In practice, CEVAL could take the shape of an interpreter or virtual machine. Formally, CEVAL returns the successor configuration for a given program *prog* and a *concrete environment* $\chi : \text{Var} \to \mathbb{R} \cup \{\bot\}$:

$$\text{CEVAL}(\textit{prog}, \chi) = (\textit{prog}', \chi'),$$

where $\textit{prog}'$ is the program derived from *prog* by executing one step, and χ' is derived from χ by applying the effects of this step. Using record notation $\langle z : c \rangle$ for the environment χ with $\chi[z] = c$ and $\chi[y] = \bot$ for $y \neq z$, we thus have $\text{CEVAL}\big(\mathsf{y{=}2(x{-}2)}, \langle x : 23 \rangle\big) = \big(\bot, \langle y : 42, x : 23 \rangle\big)$. As shorthand for expression evaluation, we write $\text{CEVAL}(e, \chi)$ for $\chi'[z]$ where $(\bot, \chi') = \text{CEVAL}(z = e, \chi)$ with a fresh variable z. When a program runs, it follows an *execution path*, which is a sequence of steps $i \hookrightarrow j$ from statement number i in the program to one of its successors j.

Symbolic Execution

Symbolic execution of the language is encapsulated by the function SEVAL. For an execution step $i \hookrightarrow j$, SEVAL builds a symbolic description of the step's effects. The description consists of two parts: a *symbolic environment* σ, and a set of control-flow constraints P. In symbols,

$$\text{SEVAL}(\textit{prog}, i \hookrightarrow j, \sigma, P) = (\sigma', P').$$

The symbolic environment σ' captures updates to the program state as expressions. It assigns each variable x an expression e that, when evaluated in a concrete environment χ, yields the same value that x would have after executing the step concretely. Thus, e must be precise and cannot approximate the effects through simplification. However, there are no structural transparency requirements on e; SEVAL may encapsulate the effects as opaque function calls. Using the expression evaluation notation from above, we have

$$\text{CEVAL}(e, \chi) = \chi'[x] \quad \text{for} \quad (\bot, \chi') = \text{CEVAL}(\textit{prog}_i, \chi),$$

where $\textit{prog}_i$ is the i-th statement in *prog*. For example, $\text{SEVAL}\big(\mathsf{y{=}2(x{-}2)}, 0 \hookrightarrow 1, \langle x : x \rangle, \emptyset\big) = \big(\langle y : 2(x-2), x : x \rangle, \emptyset\big)$ so that evaluating the expression assigned to y yields 42 as above. We assume that SEVAL performs the variable renaming necessary to support reassignments.

The second half of the description built by SEVAL, the set P', collects the symbolic constraints of traversed conditionals. If the execution step $i \hookrightarrow j$ follows the "true" branch of an if- or while-statement with the condition $x \sim y$, then SEVAL derives P' by adding the constraint $\sigma'(x) \sim \sigma'(y)$ to P; if the step follows the "false" branch, SEVAL adds $\sigma'(x) \not\sim \sigma'(y)$ to P; otherwise it copies P. A concrete execution of the program thus follows the path given to SEVAL only if the values in the concrete environment satisfy all constraints in P'. Hence, P' is the *path condition*.

Constraints

Formally, the constraints in the path condition are triples $(\ell, \sim, r) \in \text{Exp} \times \{\leq, <, >, \geq, =, \neq\} \times \text{Exp}$, written as $\ell \sim r$. A constraint is satisfied in an environment χ if the relation denoted by $\sim$ holds between the values $\text{CEVAL}(\ell, \chi)$ and $\text{CEVAL}(r, \chi)$. We say that the path condition P is satisfied in χ if each of its constraints is satisfied in χ. In this case, we write $\chi \models P$.

Algorithm 1 The *Concolic Walk* algorithm for solving the path condition P. Notation: $P = \{\ell_i \sim_i r_i \mid i\}$ is a set of constraints, $x, y \in \mathrm{Var}$ are variables, and $\alpha, \beta, \gamma, \varepsilon, \mu, \tau : \mathrm{Var} \to \mathbb{R}$ are environments whose elements are 0 by default. Operations between environments apply point-wise. The VARS function returns the set of variables appearing in an expression, constraint, or path condition.

```
1: procedure SolveWithConcolicWalk($P$)
2:     $L \leftarrow \{c \in P \mid c$ is linear$\}$                ▷ polytope
3:     $N \leftarrow P \setminus L$                     ▷ non-linear constraints
4:     $\alpha \leftarrow$ SolveLinear($L$)              ▷ starting point
5:     if $\alpha = \bot$ then return $\bot$
6:     $i \leftarrow 0$                           ▷ iteration (step) counter
7:     while $\alpha \not\models N$ do                  ▷ $\alpha$ is not a solution
8:         if $i > I \cdot |N|$ then return $\bot$
9:         $i \leftarrow i + 1$
10:        if $\forall y \in$ vars($P$) : $\tau[y] > 0$ then  ▷ all vars tabu
11:            $\alpha \leftarrow$ RandomStep($\alpha$, vars($P$), $L$)
12:            $\tau \leftarrow \langle y : 0 \mid y \in$ vars($P$)$\rangle$
13:        end if
14:        $e_\alpha \leftarrow 0$                           ▷ error at $\alpha$
15:        $\varepsilon \leftarrow \langle y : 0 \mid y \in$ vars($P$)$\rangle$     ▷ error per variable
16:        for $c \in N$ do
17:            $w \leftarrow$ ComputeError($c, \alpha$)
18:            $e_\alpha \leftarrow e_\alpha + w$
19:            $\varepsilon \leftarrow \varepsilon + \langle y : w \mid y \in$ vars($c$)$\rangle$
20:        end for
21:        $x \leftarrow$ VarWithMaxValue($\varepsilon - \langle y : \infty \mid \tau[y] > 0\rangle$)
22:        $e_\mu, \mu \leftarrow$ FindBestNeighbor($\alpha, x, L, N$)
23:        if $e_\mu < e_\alpha$ then            ▷ found better neighbor
24:            $\alpha \leftarrow \mu$
25:            $e_\alpha \leftarrow e_\mu$
26:            $\tau \leftarrow \tau - \langle y : 1 \mid \tau[y] > 0\rangle$)
27:        else
28:            $\tau[x] \leftarrow T$                  ▷ $x$ is tabu for $T$ steps
29:        end if
30:    end while
31:    return $\alpha$
32: end procedure
```

As motivated in subsection 3.1, we distinguish linear from non-linear constraints to exploit their decidability and geometric interpretation. A constraint $\ell \sim r$ is linear if it can be transformed into a normal form

$$\sum_{i=1}^{n} a_i x_i \sim b$$

with variables x_i and constants $a_i, b \in \mathbb{R}$.

3.3 Concolic Walk Algorithm

Algorithm 1 formalizes the CW algorithm. The algorithm accepts a path condition P as input and returns a concrete environment that satisfies P, or $\bot$ if it could not find such environment.

In preparation for the random walk, the algorithm extracts the polytope description from P and generates a starting point within the polytope (lines 2–5). The polytope description, denoted L, simply consists of all linear constraints in P. The starting point is the solution for L returned by a linear constraint solving function SolveLinear. In dimensions unconstrained by L, the point has arbitrary entries. Assuming that SolveLinear is sound and complete, P is unsatisfiable if no solution for L exists, and the algorithm hence returns $\bot$. Otherwise, the random walk starts and continues until either a solution was found (line 7), or the iteration budget was exhausted (line 8).

Algorithm 2 Error score for the constraint $\ell \sim r$ in the environment χ that captures "how badly" the constraint is violated. The procedure computes the score by executing the symbolic expressions ℓ, r on the concrete inputs in χ.

```
1: procedure ComputeError($\ell \sim r, \chi$)
2:     $d \leftarrow$ ceval($\ell, \chi$) $-$ ceval($r, \chi$)
3:     if $\sim$ is $=$ then
4:         return $|d|$
5:     else if $\sim$ is $\neq$ then
6:         if $d \neq 0$ then return 0 else return 1
7:     else
8:         if $d \sim 0$ then return 0 else return $|d| + 1$
9:     end if
10: end procedure
11: procedure ComputeError($\{\ell_i \sim_i r_i \mid i\}, \chi$)
12:     return $\sum_i$ ComputeError($\ell_i \sim_i r_i, \chi$)
13: end procedure
```

In each iteration, the algorithm tries to find an environment with a lower *error score* than the current environment α. To do so, it (1) finds the variable x with the highest error score (lines 14–20); (2) generates neighbor environments for α by modifying this variable's value; (3) picks the neighbor μ with the lowest error score (both line 21); (4) and makes μ the current environment if is better than α (lines 23–26). Algorithm 2 shows the function for computing the error scores; the function resembles Korel's *branch functions* [27].

The algorithm maintains a *tabu* counter τ for each variable to escape local error-score minima. If modifying a variable x failed to yield a better neighbor, it is marked as tabu for T iterations (line 28). Variables marked as tabu cannot be selected for modification: they are assigned an error score of $-\infty$ before choosing the maximal one (line 21). Consequently, the algorithm explores other directions if one seems to lead nowhere. Every iteration a better neighbor was found, all tabu counters are decremented (line 26). However, some constraints, like $x \cdot y > 0$, require changing multiple variables at the same time and cause the algorithm to declare each variable tabu, one after another. Thus, if all variables are tabu, the algorithm modifies the values of all variables and resets the tabu counters (lines 10–13).

Neighbor Selection

Generating neighbors and picking the best one has been extracted to the FindBestNeighbor function (Algorithm 3). R times, the function generates two neighbor environments β and γ for its input α, remembering the overall best one.

The environment β is the result of taking a random step in the polytope L along the axis of the given variable x; see Algorithm 4. The function RandomStep guarantees that the returned environment lies within the polytope.

The environment γ is the result of linear approximation: for a random unsatisfied constraint $\ell \sim r \in N$ that contains x, the BisectionStep function estimates the environment that would satisfy the constraint if both ℓ and r were linear

Algorithm 3 Choosing the best among environments that differ from α in their x-entry and lie inside the polytope L (satisfy L).

```
 1: procedure FindBestNeighbor($\alpha, x, L, N$)
 2:     $e_\mu \leftarrow \infty$                                   ▷ error at $\mu$
 3:     for $R$ iterations do
 4:         $\beta \leftarrow$ RandomStep($\alpha, x, L$)        ▷ in the polytope
 5:         $c \leftarrow$ OneOf($\{c \in N \mid \alpha \not\models c$ and $x \in$ vars$(c)\}$)
 6:         $\gamma \leftarrow$ BisectionStep($c, \alpha, \beta$)    ▷ may be outside
 7:         $e_\beta \leftarrow$ ComputeError($N, \beta$)
 8:         $e_\gamma \leftarrow$ ComputeError($N, \gamma$)
 9:         if $e_\beta < e_\mu$ then
10:             $e_\mu \leftarrow e_\beta$
11:             $\mu \leftarrow \beta$
12:         end if
13:         if $\gamma \models L$ and $e_\gamma < e_\mu$ then
14:             $e_\mu \leftarrow e_\gamma$
15:             $\mu \leftarrow \gamma$
16:         end if
17:     end for
18:     return $e_\mu, \mu$
19: end procedure
```

Algorithm 4 Taking a random step from χ in the y_i-directions within the polytope L. The algorithm uses rejection sampling with at most M samples. Parameter S affects the step radius. The function AdjustToSatisfy restores linear equalities that were violated during randomization by re-computing affected values from ν.

```
1: procedure RandomStep($\chi, \{y_i \mid i\}, L$)
2:     $E \leftarrow \{\ell = r \in L\}$                       ▷ linear equations
3:     for $M$ iterations do
4:         $\nu \leftarrow \chi + \langle y_i : \text{NormalRandom}(0, S) \mid i\rangle$
5:         $\nu \leftarrow$ AdjustToSatisfy($E, \nu$)
6:         if $\nu \models L$ then return $\nu$
7:     end for
8:     return $\chi$
9: end procedure
```

functions. This can be seen as performing one step of the bisection method in the hope of jumping to a region where a solution is close. Setting $v_\alpha = \text{ceval}(\ell - r, \alpha)$ and $v_\beta = \text{ceval}(\ell - r, \beta)$, the slope of the linear approximation is $t = -v_\alpha/(v_\beta - v_\alpha)$, and γ is the zero of $t \cdot (\beta - \alpha) + \alpha$. If $v_\alpha = v_\beta$, a random slope is used.

3.4 Discussion

The CW algorithm uses a sound and complete off-the-shelf solver for linear constraints. Hence, it is sound and complete for path conditions that contain only linear constraints. For non-linear path conditions, the algorithm uses heuristic search. The search ends with a negative result when it has exhausted its iteration budget. Thus, it is not complete. However, it is sound because a returned environment α must satisfy the non-linear path condition N to escape the main loop, and all generated neighbors lie within the polytope L. Consequently, α satisfies the original path condition $P = L \cup N$.

4. IMPLEMENTATION

We have implemented the algorithm described in subsection 3.3 as an extension of *Symbolic PathFinder* (SPF) [33, 35]. SPF is a symbolic execution engine built on top of the JPF verification framework. Our extension works with existing SPF test-drivers; the only change required is enabling the extension by setting a configuration flag. The implementation, as well as the evaluation harness and the raw data are available for download on our website.[5]

5. EVALUATION

This section evaluates how effective and efficient the *Concolic Walk* (CW) algorithm is in solving path conditions with non-linear arithmetic constraints. To make the results comparable and link them to a practical application of the algorithm, we measure effectiveness as the coverage of generated test cases on a focused program corpus. The corpus consists of programs whose path conditions contain mostly non-linear constraints. In subsection 5.1, we evaluate the performance of the CW algorithm. In subsection 5.2, we investigate how the parameters of the algorithm influence the coverage.

Program Corpus

Table 1 lists the programs used in the evaluation together with the type of non-linear operations appearing in their path conditions. Due to our focus on non-linear arithmetic constraints, the programs are samples from a population of programs that use such constraints; they do not represent *common* programs. To avoid a bias towards specific strengths of our approach and to foster comparability, we use mostly examples from works presenting other approaches to concrete–symbolic and randomized solving of path conditions: The *coral* program[6] is a collection of 65 benchmark functions used to evaluate the CORAL constraint solver [37]. The functions consist of a single if-statement whose condition includes a mixture of complex mathematical operations like calls to trigonometric functions. *opti* contains the six non-linear benchmark functions that were part of evaluating the FloPSy floating-point constraint solver [28]. The *dart*, *power*, *sine*, *stat*, and *tsafe* programs are part of the *Symbolic PathFinder* distribution. *stat* computes the mean and standard deviation of a list of numbers; and *tsafe* is an aviation safety program that predicts and resolves the loss of separation between airplanes. *blind* is an implementation of Figure 1, and *hash* tries to provoke five collision variants in a common hash function [5]. *tcas* features involved, but linear, control-flow. It demonstrates how the evaluated algorithm behaves on *classical* testing problems. Finally, the *ray* application[7] is a simple ray tracing renderer.

Method

We generate test cases for each program in the corpus and record the coverage of the generated tests with JaCoCo.[8] To account for randomness, we repeat this process seven times and verify the statistical significance of observed differences in

[5]http://osl.cs.illinois.edu/software/

[6]http://pan.cin.ufpe.br/coral/Download.html

[7]http://groups.csail.mit.edu/graphics/classes/6.837/F98/Lecture20/RayTrace.java

[8]http://www.eclemma.org/jacoco/

Table 1: Programs used to evaluate the CW algorithm. The *LoC* column lists the number of source code lines in the program, excluding comments and empty lines. The *Operations* column describes the type of operations appearing in the path condition.

Program	*Operations*	*LoC*
coral	Trigonometric functions, polynomials	335
blind	Multinomial	17
hash	Polynomial, shift, bit-wise xor	54
opti	Exponentials, square roots	48
dart	Polynomials, required overflow	18
power	Exponential function	31
ray	Polynomials (dot product)	304
sine	Float to bit-vector conversion	289
stat	Mean and std. dev. computation	113
tcas	Constant equality checks	157
tsafe	Trigonometric functions	88

the coverage and generation time distributions through non-parametric Mann–Whitney U-tests.[9] To account for varying difficulty, we perform one test per program between the seven measurements of the two compared algorithms. Unless stated otherwise, the test is two-tailed and the significance level is $\alpha = 0.01$. To prevent small programs from dominating the mean coverage of the algorithms, we weight each program's contribution by the program's lines of code when computing the arithmetic mean.

The kind of coverage reported depends on the type of the program. We distinguish between benchmarks and other programs. Benchmarks encode a single constraint that must be solved as a conditional in an if-statement. For these, we report whether one of the generated test cases covers the target statement, which indicates that the encoded constraint was solved. Branch coverage is a bad measure in this case because short-circuit logic operators manifest as branches in the control flow, meaning that despite solving the constraint, some "false" branches may remain uncovered. The other programs contain a more diverse range of execution paths that should be explored. For these, we report the total branch coverage of the generated test cases.

To avoid adverse effects on a tool's performance from handling object-creation [44], each test invokes a single driver method with just numerical parameters. Thus, the challenge of creating objects as test inputs is absent in our setup.

5.1 Effectiveness and Efficiency

This section discusses the performance of the Concolic Walk algorithm.

RQ1: Is the CW *algorithm more effective than simplification for solving complex arithmetic path conditions?*

We consider the null-hypothesis that simplification *is* as effective as the CW algorithm in solving path conditions with non-linear arithmetic constraints. To test this hypothesis, we compare the coverage achieved by our algorithm implementation against that of two other tools that use solvers for linear constraints, but otherwise rely on simplification:

[9]scipy.stats.mannwhitneyu() in SciPy v0.13.3

Table 2: Coverage and generation (wall-clock) time of the test cases generated for each program. The *Med.* columns show the median of seven runs; the *Var.* columns show the respective variance. For benchmarks, the coverage is the percentage of covered target statements; for other programs, it is the branch coverage. Dots denote zeros. The *LoC-Weighted Avg.* row lists the arithmetic mean over all programs, using the LoC as weights to prevent small programs from dominating the numbers.

	Coverage (%)		*Time (sec.)*	
Program	*Med.*	*Var.*	*Med.*	*Var.*
coral	78	2.0	1.4 min.	1.6
blind	100	·	1.1	·
hash	80	·	5.6	0.1
opti	50	·	7.0	0.1
dart	86	·	1.2	·
power	100	·	1.2	·
ray	90	·	11 min.	1.9
sine	55	3.7	1.9	0.2
stat	75	·	1.2	·
tcas	91	·	45.6	1.6
tsafe	92	·	4.8	0.1
LoC-Weighted Avg.	78	1.2	1.2 min.	0.5

- SPF-Mixed [33, 35] is a variant of *Symbolic PathFinder* that attempts to solve a non-linear arithmetic path condition by solving the decidable (simple) part, using the solution to concretely execute the complex part, and then further simplifying all constraints using the results [34]. As suggested in the paper, we enable the randomization heuristic of SPF-Mixed. In addition, we increase the maximum number of solving tries to three per path condition instead of the default of one.

- jCUTE [36] is a classic concolic testing tool. During the construction of a path condition, it substitutes parts of non-linear terms with their concrete run-time value to ensure that all constraints remain linear. In our setup, jCUTE randomizes the initial concrete values and explores paths in random order.

Both simplification-based tools achieve a much lower coverage: the weighted average of the median coverages is 41% for jCUTE and 26% for SPF-Mixed; the respective standard deviations are 1.8% and 0.5%. This is far from the 78% coverage achieved by our algorithm (s.d. 1.2%), see Table 2. On each program, our algorithm achieves a higher coverage than jCUTE. Except for *stat* and *tcas*, the same is true for SPF-Mixed. All differences are significant. Furthermore, the inputs generated by our algorithm subsume the inputs of the other tools except for a small fraction of *ray* and *stat* inputs, which shows that the differences are true improvements. The draw between our algorithm and SPF-Mixed on *tcas*, which lacks non-linear operations, indicates that the differences originate in the management of non-linear constraints.

Consequently, we reject the null-hypothesis; the results suggest that the CW algorithm is more effective than the simplification used in both tools.

RQ2: Can the algorithm improve the effectiveness of concolic test generators that use strong solvers?

As discussed in section 2, strong constraint solvers allow test generators to solve more path conditions directly, thereby reducing the need to resort to approximations like simplification. For such tools, the CW algorithm can serve as a fallback strategy whenever the solver cannot handle the path condition. However, such cases might be rare with state-of-the-art solvers. We therefore investigate the null-hypothesis that combining state-of-the-art concolic test generators with the CW algorithm does *not* improve the achieved coverage. To test the hypothesis, we consider combinations of our algorithm with two tools:

- SPF-CORAL, a symbolic execution tool that relies on the CORAL solver [37, 6]. CORAL targets non-linear arithmetic constraints and uses the Particle Swarm Optimization [25] search heuristic to find solutions. We use SPF-CORAL in the default configuration set in the SPF repository.
- Pex[10] [41], a concolic test generator that employs fitness-guided exploration [45] and a strong SMT solver (Z3 [31]). Because our experiment focuses on coverage rather than user responsiveness, we increase Pex's solver timeout to five seconds and allow it to run up to 500 iterations without generating new tests.

To avoid the cost of implementing the tool combinations, we simulate them by unifying the generated tests. For each run and each program, we take the union of the inputs generated by SPF-CORAL and the CW algorithm, and likewise for Pex. This allows us to measure the increase in coverage that our algorithm contributes; inputs that lead to duplicated execution of already-covered program paths have no effect on the coverage.

The combination with the CW algorithm raises the averaged median coverage of SPF-CORAL from 62% to 82% (s.d. 0.2% and 0.9%), and that of Pex from 68% to 82% (s.d. 0.5% and 1.5%). Both tools achieve higher coverage on the *coral*, *opti*, *ray*, and *sine* programs. SPF-CORAL furthermore improves on *hash*, *dart*, and *tcas*, while Pex improves on *tsafe*. In the case of *sine*, this is not surprising: the problematic example2 method in Figure 1 is a snippet of this program. In the case of *coral*, the tools and our algorithm complement each other: the union of test inputs achieves higher coverage than each set of inputs alone.

A *one-tailed* Mann–Whitney U-test on the coverages of each program between SPF-CORAL or Pex and its combination with our algorithm indicates that all improvements are significant. We therefore reject the null-hypothesis and conclude that our algorithm can improve the effectiveness of test generators that use strong constraint solvers.

RQ3: How efficient is the CW *algorithm?*

Efficiency, that is, test coverage achieved per unit of generation time, depends on implementation choices. To reduce the impact of unrelated details—such as the depth-first path exploration strategy that leads to 5 min. timeouts in *ray*—, we compare our SPF-based implementation against the two SPF-based test generators SPF-Mixed and SPF-CORAL.

[10]Pex is a C# tool. For the evaluation, we translated the whole program corpus to C#, generated inputs with Pex, and made the inputs into Java unit tests. The coverage for Pex is thus measured like that of other tools and directly comparable.

Averaged over all programs, our CW implementation (1.1% coverage / second) is about 1.6 times as efficient as SPF-CORAL (0.7%/s) and 5.5 times as efficient as SPF-Mixed (0.2%/s) in generating test inputs. One reason for the higher efficiency is the inability of SPF-Mixed and SPF-CORAL to generate inputs for the *hash* and *sine* programs, which contain bit-operations and library calls. However, even on the *coral* benchmarks, which lack such problems, our algorithm is only slightly slower than SPF-Mixed (1.4 min. vs. 1.2 min), but delivers considerably more solutions (78% vs. 12%); it is 1.8 times as fast as SPF-CORAL while achieving 92% of the coverage (78% vs. 85%). The coverage differences of all programs are significant ($\alpha = 0.01$), as are the time differences for SPF-CORAL. For SPF-Mixed, only 6 of 11 times differ significantly ($\alpha = 0.05$). In summary, these numbers suggest that the CW algorithm is more efficient than its two competitors.

5.2 Influence of Algorithm Parameters

This section discusses how the parameters of the CW algorithm influence its effectiveness. In our experiments, we vary a single parameter at a time and compare the variation's coverage with the baseline coverage shown in Table 2.

RQ4: How much does the tabu mechanism improve the effectiveness of the algorithm?

Without the marking of variables as tabu, the overall coverage drops to 62% (s.d. 0.9%), which is 0.79 times (Table 3) the 78% baseline coverage. Thus, the tabu mechanism contributes a relative performance increase of 26% to the algorithm. This performance change appears consistently over all runs of the algorithm; the difference is significant for all programs except *sine*, whose hard floating-point to bit-vector conversion emphasizes the random walk aspect of our algorithm. Once tabu marking is enabled, however, the chosen number of tabu iterations seems to have little influence on the algorithm's performance: increasing it from the default $T = \min(3, |\textsc{vars}(N)|/2)$ to $T = \min(5, |\textsc{vars}(N)|)$, for example, lacks any effect on the coverage.

RQ5: How much does the bisection step improve the effectiveness of the algorithm?

Disabling the estimation of solutions through linear approximation—the bisection step explained in subsection 3.3—reduces the overall coverage to 0.92 times the baseline. Thus, the bisection step improves the overall coverage from 72% (s.d. 1.4%) to 78%, a relative increase of about 8%. Without bisection, the algorithm consistently achieves lower coverage on the *coral*, *hash*, and *opti* benchmarks (significant for $\alpha = 0.01$). It therefore seems that the bisection step steers the random walk towards promising areas, resulting in more found solutions.

RQ6: What influence do the number of neighbors and number of steps have on the performance?

For the majority of programs in the corpus, granting the algorithm more steps to find a solution or choosing among more neighbors has little effect on the coverage. The average coverage for just 10 steps per constraint is 0.95 times the baseline coverage. Likewise, allowing more than the 150 baseline steps per constraint leads to the same overall coverage.

Table 3: Fraction of the baseline coverage (Table 2) that the algorithm variations achieve. Values below 1.0 mean less coverage than the baseline algorithm; values above 1.0 mean more coverage. Dots ($\cdot$) denote the value 1.0 and indicate no change. Each variation changes a single parameter of the baseline algorithm, which uses the following settings (see Algorithm 1): $I = 150$ steps per constraint, $R = 10$ neighbors, $T = \min(3, |\mathbf{vars}(N)|/2)$ tabu iterations, and bisection enabled. Each fraction is that of the median of seven runs.

			Benchmarks				*Other*						
Algorithm Variation		*W.Avg.*	*coral*	*blind*	*hash*	*opti*	*dart*	*power*	*ray*	*sine*	*stat*	*tcas*	*tsafe*
Tabu disabled	$(T = 0)$	0.79	0.57	·	0.25	·	·	0.64	0.79	0.93	·	·	0.92
Tabu extended	$(T \geq 5)$	·	·	·	·	·	·	·	·	·	·	·	·
Bisection disabled		0.92	0.86	·	0.5	0.33	·	·	·	0.89	·	·	·
10 steps	$(I = 10)$	0.95	0.82	·	·	·	0.75	·	1.04	0.89	·	·	·
75 steps	$(I = 75)$	1.01	1.02	·	·	·	·	·	·	1.04	·	·	·
300 steps	$(I = 300)$	·	1.04	·	·	·	·	·	·	0.96	·	·	·
3 neighbors	$(R = 3)$	·	1.02	·	·	·	·	·	·	·	·	·	·
25 neighbors	$(R = 25)$	1.01	·	·	·	·	·	·	·	1.04	·	·	·
100 neighbors	$(R = 100)$	·	1.04	·	·	·	·	·	0.98	·	·	·	·

It appears that for most programs, the constraints are simple enough that few steps suffice to find a solution. However, for the complex constraints of *coral*, more steps increase the coverage, but only for 10 steps per constraint is the difference to the baseline significant ($\alpha = 0.05$). The price for the 27% relative coverage improvement from 10 steps to 300 steps per constraint is a relative slowdown of 23%: the test generation time grows from 1.3 minutes (s.d. 0.3s) for 10 steps to 1.6 minutes (s.d. 3.2s) for 300 steps per constraint.

The number of neighbors generated per step seems to have little influence on the overall coverage. The coverage differences are statistically significant ($\alpha = 0.05$) only for 100 and for 3 neighbors. The slowdown from generating more neighbors is similar to that of increasing the number of steps.

6. LIMITATIONS AND FUTURE WORK

Evaluation—Threats to Validity

We try to ensure *conclusion validity* of our evaluation by checking the statistical significance of measured differences with a robust non-parametric test at a high level $\alpha = 0.01$. One threat to the *construct validity* of our experiments is the use of coverage as effectiveness metric. While branch coverage is a good predictor for the bug-detection capability of test suites [15], it does not measure how useful the generated inputs are for the user. Lacking a user study, we are unfortunately limited to this common surrogate metric. Another threat, in particular for RQ1, is the aggregate nature of coverage: two test suites with similar coverage may complement each other, making them incomparable. We mitigate this risk in by checking whether the test suite with higher coverage subsumes the one with lower coverage.

The *internal validity* of our experiments is threatened by our comparison of different tools. Many implementation details, not just constraint solving, influence the performance. We try to lessen this problem by (a) including a program in the corpus (*tcas*) that lacks non-linear operations and ensuring that the compared tools have similar effectiveness (RQ1); and (b) limiting the efficiency comparison to tools sharing the same SPF-based infrastructure (RQ3).

Finally, the *external validity* of our evaluation is threatened by our focused corpus. Despite over one third of programs being excerpts of realistic programs, the corpus does not constitute a random sample of programs with non-linear path conditions. Consequently, our results may generalize poorly. A larger study would mitigate this risk, but is unfortunately too expensive at this time as the used tools (SPF, jCUTE, Pex) require significant manual setup.

Algorithm

The CW algorithm currently cannot create objects as test inputs. While it could employ, among others, feedback-directed randomization [32] or heuristic search [23, 13] to fill this gap, we plan to investigate in future work how exactly object randomization relates to the notions of continuity and neighborhood used by its local search strategy.

For arithmetic constraints, the algorithm assumes an (at least) piece-wise continuity of the constraint error score functions to identify the most promising neighbor. While the assumption seems to work well in practice, other modes of finding solutions may be more effective for highly non-continuous operations like hash functions.

The algorithm currently makes no provisions for disjunctive constraints, for which the linear constraints can describe non-contiguous regions. While disjunctions cannot occur if boolean connectives are encoded in the program's control-flow, as assumed in this paper, tools that work under different assumptions may have to spawn multiple instances of the algorithm. Support for non-contiguous regions within a subset of dimensions would improve the algorithm's applicability in such scenarios.

Implementation

The CW implementation described in section 4 assumes that the native methods occurring in constraints are pure, that is, lack side-effects. A more general implementation could purge this assumptions by integrating setup and tear-down methods—as in unit tests—that are executed before and after each constraint evaluation, and by maintaining the program's execution order for native methods. Furthermore, a better implementation could accelerate the algorithm (a) by executing constraint expressions directly in the JVM instead of interpreting them; (b) by caching the error scores of constraints whose inputs remained unchanged during a step; and (c) by using memoized solutions [42, 46] to seed the starting point.

7. RELATED WORK

The Concolic Walk algorithm (CW) presented in this paper uses a mix of heuristic search and symbolic reasoning to generate inputs that satisfy a path condition.

Search-Based Software Testing

Combinations of heuristic search and symbolic reasoning have been explored in the context of search-based software testing (SBST) [30]. Like our approach, SBST searches for test inputs that meet a coverage criterion by iteratively selecting inputs that, according to a fitness function, seem closer to a solution. However, inputs can vary in granularity, ranging from primitive values to method sequences for constructing objects. Common heuristics for finding better inputs are genetic algorithms (GA), as well as the Alternating Variable Method (AVM) [27], which is similar to the adaptive search [9] used by us.

Heuristic search can be slow in discovering the specific solutions of narrow branch conditions. A number of approaches thus suggests to accelerate the search through symbolic reasoning: The Evacon framework [23] constructs high coverage tests for object-oriented programs by alternating between generating method sequences for object creation via GA, and generating primitive method arguments via concolic execution. Other techniques introduce a mutation operator in the GA that yields new test individuals by concolically executing an existing test and flipping a branch condition [29, 14]. Symbolic execution has also been applied to derive fitness functions that represent the search landscape more accurately [2], thereby improving the efficiency of the search heuristic. These approaches differ from ours in that they use symbolic reasoning as part of heuristic search steps. In contrast, the CW algorithm uses heuristic search to solve a symbolic path condition. The robustness and strength that symbolic solving gains from the CW algorithm would thus benefit the hybrid SBST approaches without any further modification.

Symbolic and Concolic Execution

Search heuristics have been applied to concolic testing. The Fitnex approach [45] improves the coverage of concolic testing by picking the *fittest* path for exploration in each iteration. However, Fitnex is independent of the path condition solver. In particular, it does not influence how classic concolic execution [19, 36, 41] replaces unsolvable constraints with concrete values (see section 2).

Instead of simplifying such constraints, our algorithm evaluates them to avoid the coarse approximations from blind commitment to concrete values. This notion of using concrete execution to solve undecidable arithmetic constraints during symbolic execution has been explored as *mixed concrete–symbolic solving* [34] (see section 2). Like our approach, mixed solving seeks to avoid blind commitment and therefore does not eagerly simplify the path condition with run-time values. However, simplification remains a central strategy. Another difference is that mixed solving makes no assumptions about the type of directly solvable constraints. Hence it must rely on the solver to provide all concrete inputs for executing the complex constraints. Yet, no mechanism guides the solver towards potential solutions for the complex constraints when re-solving the simple constraints. A similar disconnect between simple and complex constraints limits the performance of solvers based purely on randomization and substitution-based simplification [38, 39].

Search-Based Constraint Solving

Search heuristics similar to the adaptive search used in our approach have been used for solving complex arithmetic constraints. In contrast to the CW algorithm, the resulting solvers are domain-specific and lack callback interfaces to include previously unknown functions in the constraints. The CORAL solver [37] uses Particle-Swarm Optimization [25] and AVM to solve constraints that include rich mathematical operations like exponentiation and trigonometric functions. A recent extension [6] improves CORAL's efficiency by seeding the initial solution population through interval solving. FloPSy [28] is a plugin for Pex [41] that solves floating-point constraints with heuristic search methods, including AVM.

Random Testing

To facilitate random testing of specific program parts, Gotlieb and Petit [20, 21] show how to construct a domain for uniformly sampling inputs that satisfy a path condition. The approach is based on constraint propagation and refutation. Unlike our approach, it cannot handle uninterpreted functions or bit-wise operations in the path condition.

8. CONCLUSIONS

The path conditions of programs may contain calls to library methods and complicated arithmetic constraints that are infeasible to solve. Yet, test input generators based on symbolic and concolic execution must solve such path conditions to systematically explore the program paths and produce high coverage tests. Existing approaches either simplify complicated constraints, or rely on specialized constraint solvers. However, simplification yields few solutions; and specialized constraint solvers lack support for native library methods. To address both limitations, this paper introduces the CW algorithm for solving path conditions. An evaluation on a corpus of small to medium sized programs shows that the algorithm generates tests with higher coverage than simplification-based tools and moreover improves the coverage of state-of-the-art concolic test generators.

9. ACKNOWLEDGMENTS

The authors would like to thank Mohsen Vakilian and Tao Xie for insightful comments on early drafts of this paper. This publication was made possible in part by sponsorship from the Army Research Office under award W911NF-09-1-0273, and from the Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement FA8750-11-2-0084. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The first author was also supported by a fellowship from the Graduate College at the University of Illinois at Urbana–Champaign.

10. REFERENCES

[1] The economic impacts of inadequate infrastructure for software testing. Planning report 02-03, National Institute of Standards, May 2002.

[2] A. I. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. E. J. Vos. Symbolic search-based testing. In *ASE 2011*, pages 53–62. IEEE, 2011.

[3] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA 2008*, pages 3–14. ACM, 2008.

[4] N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons, S. Tetali, and A. V. Thakur. Proofs from tests. *IEEE Trans. Software Eng.*, 36(4):495–508, 2010.

[5] J. Bloch. *Effective Java*. Addison-Wesley, second edition, 2008.

[6] M. Borges, M. d'Amorim, S. Anand, D. H. Bushnell, and C. S. Păsăreanu. Symbolic execution with interval solving and meta-heuristic search. In *ICST 2012*, pages 111–120. IEEE, 2012.

[7] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI 2008*, pages 209–224. USENIX Association, 2008.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS 2006*, pages 322–335. ACM, 2006.

[9] P. Codognet and D. Diaz. Yet another local search method for constraint solving. In *SAGA 2001*, volume 2264 of *LNCS*, pages 73–90. Springer, 2001.

[10] V. Dallmeier, N. Knopp, C. Mallon, S. Hack, and A. Zeller. Generating test cases for specification mining. In *ISSTA 2010*, pages 85–96. ACM, 2010.

[11] M. Davis. Hilbert's tenth problem is unsolvable. *American Mathematical Monthly*, 80:233–269, 1973.

[12] X. Deng, Robby, and J. Hatcliff. Kiasan: A verification and test-case generation framework for java based on symbolic execution. In *ISoLA 2006*, page 137. IEEE, 2006.

[13] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *SIGSOFT FSE 2011*, pages 416–419. ACM, 2011.

[14] J. P. Galeotti, G. Fraser, and A. Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *ISSRE 2013*, pages 360–369. IEEE, 2013.

[15] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *ISSTA 2013*, pages 302–313. ACM, 2013.

[16] F. Glover. Tabu search - part i. *INFORMS Journal on Computing*, 1(3):190–206, 1989.

[17] F. Glover. Tabu search - part ii. *INFORMS Journal on Computing*, 2(1):4–32, 1990.

[18] P. Godefroid. Higher-order test generation. In *PLDI 2011*, pages 258–269. ACM, 2011.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI 2005*, pages 213–223. ACM, 2005.

[20] A. Gotlieb and M. Petit. Path-oriented random testing. In *Proceedings of the First International Workshop on Random Testing*, 2006.

[21] A. Gotlieb and M. Petit. A uniform random test data generator for path testing. *Journal of Systems and Software*, 83(12):2618–2626, 2010.

[22] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT FSE 2006*, pages 117–127. ACM, 2006.

[23] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE 2008*, pages 297–306. IEEE, 2008.

[24] W. Jin, A. Orso, and T. Xie. Automated behavioral regression testing. In *ICST 2010*, pages 137–146. IEEE Computer Society, 2010.

[25] J. Kennedy and R. Eberhart. Particle swarm optimization. In *IEEE Internat. Conf. on Neural Networks*, volume 4, pages 1942–1948. IEEE, 1995.

[26] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS 2003*, volume 2619 of *LNCS*, pages 553–568. Springer, 2003.

[27] B. Korel. Automated software test data generation. *IEEE Trans. Software Eng.*, 16(8):870–879, 1990.

[28] K. Lakhotia, N. Tillmann, M. Harman, and J. de Halleux. FloPSy - search-based floating point constraint solving for symbolic execution. In *ICTSS 2010*, volume 6435 of *LNCS*, pages 142–157. Springer, 2010.

[29] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *ASE 2011*, pages 436–439. IEEE, 2011.

[30] P. McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.

[31] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*. Springer, 2008.

[32] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *OOPSLA 2007 Companion*, pages 815–816. ACM, 2007.

[33] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of java bytecode. In *ASE 2010*, pages 179–180. ACM, 2010.

[34] C. S. Păsăreanu, N. Rungta, and W. Visser. Symbolic execution with mixed concrete-symbolic solving. In *ISSTA 2011*, pages 34–44. ACM, 2011.

[35] C. S. Păsăreanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.

[36] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV 2006*, volume 4144 of *LNCS*, pages 419–423. Springer, 2006.

[37] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu. CORAL: Solving complex constraints for

Symbolic PathFinder. In *NASA Formal Methods*, pages 359–374, 2011.

[38] M. Takaki, D. Cavalcanti, R. Gheyi, J. Iyoda, M. d'Amorim, and R. B. C. Prudêncio. A comparative study of randomized constraint solvers for random-symbolic testing. In *NASA Formal Methods*, pages 56–65, 2009.

[39] M. Takaki, D. Cavalcanti, R. Gheyi, J. Iyoda, M. d'Amorim, and R. B. C. Prudêncio. Randomized constraint solvers: a comparative study. *ISSE*, 6(3):243–253, 2010.

[40] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. eXpress: guided path exploration for efficient regression test generation. In *ISSTA 2011*, pages 1–11. ACM, 2011.

[41] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *TAP 2008*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.

[42] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *SIGSOFT FSE 2012*, page 58. ACM, 2012.

[43] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA 2004*, pages 97–107. ACM, 2004.

[44] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *ICSE 2011*, pages 611–620. ACM, 2011.

[45] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN 2009*, pages 359–368. IEEE, 2009.

[46] G. Yang, S. Khurshid, and C. S. Păsăreanu. Memoise: a tool for memoized symbolic execution. In *ICSE 2013*, pages 1343–1346. IEEE / ACM, 2013.