Targeted Test Input Generation using Symbolic–Concrete Backward Execution[⋆]

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

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

ABSTRACT

Knowing inputs that cover a specific branch or statement in a program is useful for debugging and regression testing. Symbolic backward execution (SBE) is a natural approach to find such targeted inputs. However, SBE struggles with complicated arithmetic, external method calls, and data-dependent loops that occur in many real-world programs. We propose *symcretic execution*, a novel combination of SBE and concrete forward execution that can efficiently find targeted inputs despite these challenges. An evaluation of our approach on a range of test cases shows that symcretic execution finds inputs in more cases than concolic testing tools while exploring fewer path segments. Integration of our approach will allow test generation tools to fill coverage gaps and static bug detectors to verify candidate bugs with concrete test cases.

Categories and Subject Descriptors

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

Keywords

Concolic; Symcretic; Backward Execution; Goal-Directed

1. INTRODUCTION

The distribution of bugs in real-world programs tends to be highly non-uniform [14, 2]. Thus a test suite that covers most of a program may nevertheless fail to cover the parts that contain many bugs. Generally, the goal of test generation tools is to maximize the *overall* coverage [19, 31, 30, 32, 26, 16]. However, it has been argued that this yields test inputs that are often of limited use for developers [17]. We take an alternative approach: our goal is to automatically find *targeted test inputs* that cover a specific branch or statement in the code. Developers can then use such targeted inputs, for example, to triage a reported bug [6, 38], or to verify that a suspicious instruction pattern is an actual problem, or to add a test case to cover a specific code change [36].

A natural approach for finding targeted inputs is to use *symbolic backward execution* (SBE) [3, 6, 7]. SBE explores a program in the 'reverse' direction of normal (forward) execution. Starting from a specific target statement, SBE continues until it reaches an entry point—thus considering only those execution paths that can reach the target. By collecting a set of constraints (the *path condition*) during this exploration, SBE builds a symbolic characterization of the execution path it explored. A path condition is similar to a weakest precondition; solving it yields inputs that drive the program down the characterized path to cover the target.

Unfortunately, symbolic backward execution poses some challenges:

(1) Because path conditions may contain arbitrary integer constraints, they may be undecidable [8], or solving them may be computationally infeasible. In this case, when asked to check the satisfiability of a path condition, a decision procedure may reply *unknown*.

(2) Symbolic decision procedures cannot reason about external methods such as *native* methods in Java.

(3) Data-dependent loops can require an arbitrary number of iterations to find a satisfiable path condition, leading to an unbounded search space.

Following the general idea of concolic execution [19, 31], we show how to combine *sym*bolic backward execution with con*crete* execution to efficiently find targeted inputs despite these challenges. Our approach, called *symcretic execution*, operates in two phases:

Phase I. Symbolic backward execution is used to find a feasible execution path from the given target to any of the program's entry points. Unlike prior approaches [6], symcretic execution 'skips' over constraints that are problematic for the symbolic decision procedure and defers their solution until the second phase.

Phase II. Concrete forward execution begins when the symbolic backward execution reaches an entry point. Executing a trace of the program along the discovered path, this phase uses heuristic search to find inputs that satisfy the constraints that were skipped in Phase I.

The integration of concrete execution allows symcretic execution to solve a range of arithmetic constraints that are too hard for symbolic decision procedures and enables the effective handling of external methods. Moreover, if a loop along a path requires too many symbolic traversals,

[⋆] An extended abstract [11] of this report was presented at the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014), September 15–19, 2014, Västerås, Sweden.
`http://dx.doi.org/10.1145/2642937.2642951`

```
 1 public void challenges(int x, double u) {
 2   int res = 0;
 3   int i = 0;
 4   while (i < x) {
 5     int tmp = i % 2;
 6     if (tmp == 0) {
 7       res = res − 1;
 8     } else {
 9       res = res + 17;
10     }
11     i++;
12   }
13   if (res == 8192) {                     // Error condition 1
14     if (Math.sin(u) > 0) {               // Error condition 2
15       throw new AssertionError();
16     } else ...                   // Long and deep computation
17   } else ...                     // Long and deep computation
18 }
```

Figure 1: Example program whose data-dependent loop (line 4), non-linear integer arithmetic (line 5), and call to an external method (line 14) make it hard for symbolic execution to find inputs that trigger the exception in line 15.

symcretic execution treats the loop as call to an external method—thus delegating the problem of finding the right number of iterations to the cheaper concrete phase.

This paper contains the following research contributions:

- It describes the *symcretic execution* algorithm for finding targeted program inputs (section 4). To the best of our knowledge, symcretic execution is the first algorithm to use concrete execution to mitigate undecidable or infeasible constraints, external method calls, and data-dependent loops in symbolic backward execution.

- We compare symcretic execution with related techniques (section 5) and evaluate an implementation of our algorithm on a corpus of small programs (section 7). We show that our approach is feasible and more efficient than concolic execution for targeted input generation.

2. MOTIVATION

Suppose that during a code review and cleanup, we discover that the test suite fails to throw the exception on line 15 of the program shown in Figure 1. To add a test case that covers this line, we have to find inputs for an entry point of the program that lead to the execution of this line. However, manually deriving such targeted inputs is tedious and can be complicated. For example, the `challenges` method in Figure 1 must be called with the input `x = 1024` to satisfy the first error condition, `res == 8192`, on line 13.

Instead of manual derivation, automated test generation techniques can be used to find targeted inputs. One of the strongest techniques is *conc*rete–symb*olic* (concolic[1]) execution [19, 31]. Concolic execution explores a program by running it on concrete input values, for example `x = 0` and `u = 1.0`, and at the same time using symbolic execution to collect the constraints of the followed program path. This *path condition* characterizes the set of all concrete inputs that drive the program down the followed path. To explore another path in the program, concolic execution derives a new set of concrete inputs by negating one of the constraints and solving the derived path condition. If the path condition cannot be solved, concolic execution uses concrete execution to improve coverage while sacrificing completeness.

Targeted Input Generation

The goal of concolic execution and other automated test generation techniques is not to cover a specific target but to achieve high overall coverage. These techniques try to explore as much of a given program as possible to discover a bug, or to generate a test suite that is as complete as possible. In contrast, our objective is similar to that of SBE [3, 6, 7]: instead of covering as much as possible, we are interested in covering specific, relevant targets in a program. Any part of a program that does not contribute to this goal (for example lines 16 and 17 in Figure 1) is irrelevant; exploring it wastes resources.

SBE starts at the target and explores the program in the opposite direction of normal (forward) execution until it reaches an *entry point* (e.g., a public method). During the exploration, it maintains the path condition of the followed path. After reaching an entry point, it solves the path condition to obtain concrete inputs that lead to the execution of the target. For example, if the target is line 7 in Figure 1, the execution starts on this line and steps backwards, collecting the constraint $tmp = 0$. Moving further towards the top, it constructs the path condition

$$tmp = 0 \wedge tmp = i \bmod 2 \wedge i < x \wedge i = 0 \wedge res = 0.$$

Solving the path condition yields an input (such as `x = 1`) that would trigger the execution of the desired target line 7. However, SBE faces challenges mentioned in section 1: (1) the modulo operation on line 5 forces state-of-the-art decision procedures such as the Z3 SMT[2] solver [9] to reply *unknown* after few traversals of the loop; (2) the `Math.sin` method on line 14 is native and may not have an interpretation in the solver; and (3) the data-dependent loop on line 4 must be traversed 1024 times to yield `res = 8192`.

3. APPROACH

Following the general idea of concolic execution, we propose to overcome the aforementioned drawbacks of symbolic backward execution by combining it with concrete execution. Our approach, *sym*bolic–con*crete* (symcretic) execution, consists of the two phases outlined in this section.

Phase I uses SBE to try to find a feasible execution path from the target statement to an entry point. Specifically, starting from the target statement, it explores the program's control-flow graph backwards and uses an abstract interpreter to construct the path condition. Branches in the search path, for example statements with multiple predecessors or call-sites of virtual methods, are explored depth-first. After each search step, the algorithm checks the satisfiability of the current path condition with a symbolic decision procedure. The search continues if the path condition is satisfiable. It backtracks if the condition is unsatisfiable. If the decision procedure cannot answer the query, the algorithm removes the most recent constraint from the path condition, treating it as *potentially* satisfiable and deferring its solution to the second phase.

[1]Concolic execution is also known as *Directed Automated Random Testing* and *Dynamic Symbolic Execution*.

[2]Satisfiability Modulo Theories

```
1 public void simplified_challenges(int x, double u) {
2   int res = x + 23;
3   if (res == 8192) {
4     if (Math.sin(u) > 0) {
5       throw new AssertionError();
6     }
7   }
8 }
```

Figure 2: Program from Figure 1 without the loop.

Phase I also constructs a *trace* of the program along the followed path. At each search step, the algorithm prepends the trace with the current statement, regardless of whether it was removed from the path condition or not. For removed statements, the algorithm furthermore adds a call to the special change() method that marks the statement's result as needing adjustment in the second phase. Because the search follows a single execution path, if-statements and other conditionals are not directly added to the trace. Instead, the algorithm adds a call to the special fit() method that signals which of the conditional's branches the search traversed. Boolean connectives of conditions are encoded in the control-flow, which implies that all conditions along the path are non-compound and valid inputs must satisfy their conjunction. Once the search reaches the beginning of an entry point, the second phase begins.

Phase II uses *heuristic search* on the trace to find input values that satisfy constraints that were problematic in Phase I. Specifically, the algorithm repeatedly evaluates the program trace on input values, determines how *close* the branch conditions in the trace are to being satisfied, and modifies some of the inputs to move closer to a full solution. Symcretic execution does not prescribe which heuristic search algorithm to use; possible choices include genetic algorithms and the Concolic Walk algorithm [10].

We illustrate our approach on the program in Figure 2. Assume we select line 5 as target. Using SBE, we obtain the path condition $\mathsf{Math.sin}(u) > 0 \wedge res = 8192 \wedge res = x+23$. Unfortunately, our symbolic decision procedure cannot solve the path condition because it cannot reason about the native method Math.sin. Symcretic execution therefore skips the problematic constraint $\mathsf{Math.sin(u)} > 0$, which results in the satisfiable path condition $res = 8192 \wedge res = x + 23$ with solution x = 8169. Simultaneously, symcretic execution creates a trace of the program:

```
void trace1(int x, double u) {   // Phase II instructions:
  int res = x + 23;
  fit(res, '==', 8192);           // Find inputs with res == 8192
  double v = Math.sin(u);
  change(v);                      // Adjust inputs that influence v
  fit(v, '>', 0);                 // Find inputs with v > 0
}
```

The call to the change() method in the trace signals that the value of v must be found by heuristic search. Phase II thus begins by executing the trace on the inputs x = 8169 and u = 0.0—solutions obtained during Phase I. By evaluating the calls to the fit() method, Phase II determines that the constraint $v > 0$ is not yet satisfied. It therefore adjusts one of the inputs that influence v (here: u) and re-executes the trace. This process continues until a solution has been found or the time budget has been exceeded.

Data-Dependent Loops

Another challenge for symbolic execution are data-dependent loops that require many iterations, such as the loop on line 4 of Figure 1. Triggering the error on line 15 requires x = 1024 iterations of the loop, a number far beyond typical loop-unrolling bounds. For example, the state-of-the-art concolic testing tool Pex [32] fails to find the right number of iterations even with extended exploration limits. To discover this input, symcretic execution starts from line 15, collects the required constraints $\mathsf{Math.sin(u)} > 0 \wedge res = 8192$, and starts unrolling the loop. After a number of traversals, it exceeds the maximum number of iterations and gives up on the loop. It therefore treats the loop as though it were a call to an external *loop method* whose body is the loop body, whose parameters are the variables read inside the loop, and whose return values are the values written inside the loop. In this way, Phase I jumps over the loop and continues on line 3. After taking the last two symbolic steps, the trace for the execution path looks as follows:

```
void trace2(int x, double u) {
  int res = 0;
  int i = 0;
  res, i = extractedLoop(res, i, x);   // Wraps lines 4–12 in Fig. 1
  change(res);
  change(i);
  fit(res, '==', 8192);
  double v = Math.sin(u);
  change(v);
  fit(v, '>', 0);
}
```

The body of the extractedLoop method consists of lines 4 to 12 in Figure 1. The second phase of symcretic execution uses heuristic search to find inputs that (1) influence res, i, and v; and (2) satisfy the goal conditions $res = 8192$ and $v > 0$.

4. SYMCRETIC EXECUTION

This section describes the symcretic execution algorithm, which, given a program and a *target* statement, finds inputs that drive the program towards executing that target statement.

4.1 Terms and Definitions

To keep the exposition simple, we define symcretic execution for a small imperative programming language with basic arithmetic and method[3] calls; see Figure 3. Calls can refer to methods defined in the program, or to external library methods whose body remains hidden. Methods can return multiple values at once, which we will use to encapsulate loops.

We assume that the program that is subject to symcretic execution passes the customary semantic checks: type-checking succeeds, used variables have been declared and assigned, and method calls have the right number of arguments and return values. The program call-graph G that the execution follows uses the **public** methods as entrypoints.

Every method m defined in the program has an associated control-flow graph $\textsc{cfg}(m)$. The basic blocks of this control-flow graph contain at most one statement each; variable declarations and block statements do not appear. Basic blocks without a statement are *empty.* To shorten the notation, we

[3]We use the term *method* to distinguish functions defined in the program from functions that are part of our algorithm.

$$\begin{aligned}
\langle\text{Ids}\rangle &::= \langle\text{Id}\rangle\ (,\ \langle\text{Id}\rangle)^\star \\
\langle\text{Arith}\rangle &::= \langle\text{Id}\rangle \mid \langle\text{Lit}\rangle \mid \langle\text{Id}\rangle \circ \langle\text{Id}\rangle \\
\langle\text{Call}\rangle &::= \langle\text{Mthd}\rangle() \mid \langle\text{Mthd}\rangle(\ \langle\text{Ids}\rangle\ ) \\
\langle\text{Cmp}\rangle &::= \langle\text{Id}\rangle \sim \langle\text{Id}\rangle \mid \langle\text{Id}\rangle \sim \langle\text{Lit}\rangle \\
\langle\text{Decls}\rangle &::= \langle\text{Type}\rangle\ \langle\text{Id}\rangle\ (,\ \langle\text{Type}\rangle\ \langle\text{Id}\rangle)^\star \\
\langle\text{Stmt}\rangle &::= \langle\text{Decls}\rangle\ ; \\
&\quad \mid \langle\text{Id}\rangle = \langle\text{Arith}\rangle\ ; \\
&\quad \mid \langle\text{Ids}\rangle = \langle\text{Call}\rangle\ ; \\
&\quad \mid \textsf{if}\ (\ \langle\text{Cmp}\rangle\ )\ \langle\text{Stmt}\rangle\ \textsf{else}\ \langle\text{Stmt}\rangle \\
&\quad \mid \textsf{while}\ (\ \langle\text{Cmp}\rangle\ )\ \langle\text{Stmt}\rangle \\
&\quad \mid \{\ \langle\text{Stmt}\rangle^\star\ \} \\
&\quad \mid \textsf{return}\ \langle\text{Ids}\rangle\ ; \\
\langle\text{Def}\rangle &::= \langle\text{Type}\rangle\ \big(,\ \langle\text{Type}\rangle\big)^\star\ \langle\text{Mthd}\rangle\ (\ \langle\text{Decls}\rangle\ )\{\ \langle\text{Stmt}\rangle\ \} \\
\langle\text{Entr}\rangle &::= \textsf{public}\ \langle\text{Def}\rangle \\
\langle\text{Prog}\rangle &::= \langle\text{Entr}\rangle^+ \mid \langle\text{Def}\rangle^\star
\end{aligned}$$

Figure 3: Syntax of the example language, with primitive types Type**, variable identifiers** Id**, literals** Lit**, method symbols** Mthd**, binary operations** $\circ \in \{+,-,*,/,\%,<<,>>\}$**, and relations** $\sim \in \{<,<=,>=,>,==,!=\}$**.**

identify non-empty basic blocks with the statement they contain. For a basic block b in $\textsc{cfg}(m)$, we write $\textsc{cfgPred}(b)$ for b's set of predecessors in the graph.

4.2 Phase I: Symbolic Execution

The depth-first search for a feasible execution path, formalized as Algorithm 1, drives the symbolic backward execution, which is the first phase of symcretic execution. Starting from the target statement, the algorithm explores the program backwards, trying to find a feasible path to the first statement of any of the program entrypoints. At the same time, it generates the program trace used by the second symcretic execution phase, heuristic solving.

The search proceeds by generating a set of next step alternatives (line 2) and recursively visiting them in turn (line 15). Step alternatives are encapsulated as *choice* structures to allow a more uniform treatment. In the simplest case, a choice represents a predecessor of the current statement in the control-flow graph. Other kinds of choices represent method calls and returns; see the detailed discussion below.

Given a choice structure c, the UPDATESTATE function called in line 3 adds a corresponding constraint to the path condition Φ, and furthermore updates the other parts of the search state as explained later. For example, if c represents stepping to the assignment i = i + 1, then UPDATESTATE adds a constraint $i_0 = i_1 + 1$ to Φ, where i_0 and i_1 are the symbols that represent i's state after and before the assignment.

Next, the algorithm asks a decision procedure whether the updated path condition is satisfiable. If it is, the search continues at the next statement (line 15)—unless the search has reached the beginning of an entrypoint method, in which case the second phase of symcretic execution begins (line 10). If the path condition became unsatisfiable, the search path is a dead end. A call to the RESTORESTATE function therefore reverts all choice-specific state updates, for example removing

Algorithm 1 Depth-first exploration of the program. The algorithm drives the symbolic backward execution phase of symcretic execution. In the algorithm, Φ denotes the current path condition.

```
 1: procedure SYMBOLICPHASE(stmt)
 2:     for $c \in$ CHOICES(stmt) do
 3:         UPDATESTATE(c)              ▷ update path condition $\Phi$
 4:         if $\Phi$ is unsat then
 5:             RESTORESTATE(c)
 6:             continue                ▷ try next choice or backtrack
 7:         else if $\Phi$ is unknown then
 8:             RESTOREPCANDMARKUSES(c)
 9:         end if
10:         if NEXTSTMT(c) = $\bot$ then              ▷ at entry
11:             if CONCRETEPHASE() is sat then
12:                 exit                ▷ found path and inputs
13:             end if
14:         else
15:             SYMBOLICPHASE(NEXTSTMT(c))  ▷ next step
16:         end if
17:         RESTORESTATE(c)                    ▷ undo updates
18:     end for
19: end procedure
```

added constraints from the path condition, before the next choice is explored. The procedure returns if no such choice exists, meaning that the search backtracks.

In case the decision procedure cannot determine the satisfiability of the updated path condition (line 7), the algorithm assumes that the path is *potentially feasible*. The algorithm therefore records in the trace that the constraint φ added by the current choice requires heuristic solving. At the same time, it removes φ from the path condition to restore the decidability of Φ. Restoring Φ is necessary to detect other constraints along the path that require heuristic solving.

Search State

Symcretic execution organizes the symbolic backward execution using a global *search state*. The first component of this search state is the path condition Φ discussed above. The second component is the program trace *Trc*, which the second symcretic execution phase uses for heuristic solving. The third component is the call stack *Stck* that is necessary to support method calls and returns.

A frame on the stack stores the name of the method to which it belongs, as well as the call site. It furthermore contains the local variable environment e, which maps the syntactic variable identifiers in the code, for example x, to their symbolic values at the current execution point, for example x_0. We use record notation $\langle \mathsf{x} : x_0 \rangle$ for the *mutable* environment e with $e[\mathsf{x}] = x_0$. Unlike traditional records, looking up an undefined entry creates and returns a fresh value: $e[\mathsf{y}] = y_0$ with y_0 a fresh symbol if $y \notin e$. After the look-up, we have $y \in e$ and $e[y] = y_0$. Point-wise updates are denoted as $e[\mathsf{x} \mapsto x_1]$, deleting entries as $e[\mathsf{x} \mapsto \bot]$.

State Updates

Algorithm 2 defines the UPDATESTATE function used in Algorithm 1 that modifies the search state according to a choice structure. The function recognizes three kinds of choice structures: call, return, and predecessor choices, which are

Algorithm 2 Application of the effects of a choice structure c to the search state, which consists of the path condition Φ, the program trace *Trc*, and the call stack *Stck*.

```
1: procedure UPDATESTATE($c$)
2:     $e \leftarrow \text{ENV}\big(\text{TOP}(Stck)\big)$    ▷ local variable environment
3:     if $c$ is CCHOICE($stmt, cs$) then    ▷ call
4:         if $cs = \bot$ then return
5:         $m \leftarrow \text{METHOD}(stmt)$
6:         $n \leftarrow \text{PARAMCOUNT}(m)$
7:         $e \leftarrow e[\text{ARG}(cs, i) \mapsto e[\text{PARAM}(m, i)] \mid 1 \leq i \leq n]$
8:         POP($Stck$)
9:     else if $c$ is RCHOICE($stmt, p, r$) then    ▷ return
10:        assert $p$ is "x1,...,xn = f(...)"
11:        if $r$ is "return y1,...,yn" then
12:            $e' \leftarrow \langle \mathsf{y}_i : e[\mathsf{x}_i] \mid i = 1, \ldots, n\rangle$
13:            PUSH($Stck$, STACKFRAME($e'$, f, $p$))
14:        else    ▷ external method
15:            add change($e[\mathsf{x}_1]$, ..., $e[\mathsf{x}_n]$) to front of $Trc$
16:            add STMT($e[\mathsf{x}_1]$, ..., $e[\mathsf{x}_n]$=f(...)) to front of $Trc$
17:        end if
18:    else if $c$ is PCHOICE($stmt, p$) then    ▷ intra-proc.
19:        if $p$ is "if (x∼y)" or "while (x∼y)" then
20:            if $(stmt, p)$ is true branch in the CFG then
21:                $\varphi \leftarrow e[\mathsf{x}] \sim e[\mathsf{y}]$
22:            else
23:                $\varphi \leftarrow e[\mathsf{x}] \not\sim e[\mathsf{y}]$
24:            end if
25:            add fit($\varphi$) to front of $Trc$    ▷ store branch
26:        else
27:            if $p$ is "x = y" then
28:                $\varphi \leftarrow e[\mathsf{x}] = e[\mathsf{y}]$
29:            else if $p$ is "x = $\ell$" then
30:                $\varphi \leftarrow e[\mathsf{x}] = \ell$
31:            else if $p$ is "x = y ∘ z" then
32:                $\varphi \leftarrow e[\mathsf{x}] = e[\mathsf{y}] \circ e[\mathsf{z}]$
33:            end if
34:            $e \leftarrow e[\mathsf{x} \mapsto \bot]$
35:            add STMT($\varphi$) to front of $Trc$
36:        end if
37:        $\Phi \leftarrow \Phi \land \varphi$
38:    end if
39: end procedure
```

constructed by the CCHOICE, RCHOICE, and PCHOICE functions. All of these contain the current and next statement.

For all call and return choices (lines 3 and 9), the function binds the method argument and return values in the respective environment before updating the stack. Note that because of backward execution, calls and returns have reversed effects: Call choices signal that the execution reached the beginning of the current method; it therefore continues at the call site, popping the stack. Return choices push a new frame on the stack to derive the return values by exploring the method starting from the return statement. Exploring the method's body adds the relevant statements to the program trace, effectively inlining it. Thus, only return choices for external methods, whose body is hidden, add an entry to the trace (lines 15 and 16). The special change method marks the arguments of the external method for modification during the concrete phase.

Algorithm 3 Generation of the set of next search steps.

```
1: procedure CHOICES($stmt$)
2:     if CFGPRED($stmt$) = $\emptyset$ then    ▷ at method entry
3:         if LEN($Stck$) > 1 then    ▷ known caller
4:             $cs \leftarrow \text{CALLSITE}\big(\text{TOP}(Stck)\big)$    ▷ jump to caller
5:             return {CCHOICE($stmt, cs$)}    ▷ single choice
6:         end if
7:         if $m \in Entr$ then    ▷ reached entrypoint
8:             return {CCHOICE($stmt, \bot$)}
9:         end if
10:        $C \leftarrow \big\{\text{CCHOICE}(stmt, cs) \mid cs \in \text{CALLSITES}(m)\big\}$
11:        return $C$    ▷ all possible callers
12:    end if
13:    if $stmt$ is loop exit and loop is new then
14:        $h \leftarrow \text{LOOPHEADER}(stmt)$
15:        $l \leftarrow \text{LOOPMETHOD}(stmt)$
16:        $C \leftarrow \{\text{RCHOICE}(stmt, h, l)\}$
17:    else
18:        $C \leftarrow \emptyset$
19:    end if
20:    for $p \in \text{CFGPRED}(stmt)$ do    ▷ traverse body
21:        if $\tau[stmt, p] \geq \mathrm{L}$ then continue
22:        $\tau[stmt, p] \leftarrow \tau[stmt, p] + 1$
23:        if $stmt$ is "x1,...,xn = m(y1,...,yn)" then
24:            $C \leftarrow C \cup \{\text{RCHOICE}(stmt, p, \text{RETSTMT}(\mathsf{m}))\}$
25:        else
26:            $C \leftarrow C \cup \{\text{PCHOICE}(stmt, p)\}$
27:        end if
28:    end for
29:    return $C$
30: end procedure
```

For predecessor choices (line 18), the function updates the path condition (line 37), and the variable environment e. The constraint φ that it adds to the path condition is a direct translation of the choice's next statement. For example, the assignment x = y yields the constraint $x_0 = y_0$ if e maps x and y to the symbols x_0 and y_0 (line 28). For assignments, the function additionally removes the assigned variable identifier from the environment: going backwards, an assignment means that the value used below has not yet been determined (line 34).

Predecessor choices furthermore add an entry to the trace. For conditionals, the added entry is a call to a special fit method that records the traversed branch (line 25). The concrete phase uses the fit method to check whether a set of concrete inputs drives the program down the intended path. For assignment statements, the function cannot add the assignment directly because this, in combination with method inlining, could cause clashes between variable identifiers. Therefore, the function instead adds the constraint φ, which uses unique symbols, translated to a statement (line 35). For example, $\text{STMT}(x_0 = y_0)$ = "x0 = y0;".

Choice Generation

Algorithm 3 returns the next search steps available at a statement *stmt*. The algorithm distinguishes whether the statement is the first in the current method. If it is, then the search continues at the method's caller, which is signaled by call choice structure (lines 2–12). Otherwise, the search traverses the method body or jumps to a callee, which is signaled by predecessor and return choice structures (lines 13–29).

Three cases can arise if *stmt* is the first statement in the method m: (1) The current method m was called by another method during the search (line 3). In this case, the search continues at the caller. (2) No other method called m and m is an entrypoint (line 7). In this case, the search found a potentially feasible path and the concrete phase of symcretic execution begins. Recall that using $\bot$ as next statement tells Algorithm 1 that a feasible path was found. (3) No other method called m, but m is not an entrypoint. In this case, the search continues at any of m's call sites.

When *stmt* is not the first statement in m, the available next search steps are the predecessors of *stmt* in the control-flow graph of m (line 26). However, if *stmt* is a method call, the search must first traverse the callee before continuing at the predecessor (the call site). The algorithm therefore generates return choice structures in this case (line 24). The auxiliary function RETSTMT finds the (unique) return statement in a method; it yields $\bot$ if the method is external, allowing the UPDATESTATE function to distinguish these cases.

Loops

To avoid getting stuck during the exploration of loops, the algorithm ignores predecessors that are connected via edges that have been traversed too often (line 21). This can result in returning an empty choice set C, which causes Algorithm 1 to backtrack.

Pure symbolic execution has to give up when it cannot find a path through a loop within the set bound L. Symcretic execution mitigates this problem by wrapping such loops in *loop methods* and delegating the solving to the concrete phase (line 16). The body of the loop method contains all statements in the loop body, that is, all statements that are both (1) predecessors of the loop header; and (2) dominated by the loop header. The parameters of the loop method are the variables that are read in the body; the return values are the variables that are assigned in the body. For every loop, the loop method choice is added only once, when its (unique) exit is first encountered (line 13). This avoids heuristically solving loops after partially unrolling them.

4.3 Phase II: Concrete Execution

The second phase of symcretic execution uses the program trace from the first phase to find inputs that satisfy the symbolically undecidable constraints. To find such inputs, the second phase relies on heuristic search, which repeatedly evaluates the program trace on input values, determines how close the branch conditions in the trace are to being satisfied, and modifies some of the inputs to move closer to a full solution. This continues until a solution is found, or until the search exceeds its budget. If the search fails, the algorithm returns to the symbolic phase to generate a new trace. Symcretic execution does not demand a specific heuristic search algorithm; example choices are genetic algorithms and the Concolic Walk algorithm [10]. In the remainder of this section, we show how a heuristic search algorithm can leverage the information recorded in the trace.

The trace is constructed by Algorithm 2 and the function RESTOREPCANDMARKUSES appearing in Algorithm 1. Evidently, the trace is a straight-line sequence of assignment statements, interspersed with calls to external and loop methods, as well as calls to the special methods fit and change:

$$
\begin{aligned}
\textsc{deps}(\mathsf{x} = \mathsf{y}) &= \{\mathsf{y}\} \cup \textsc{deps}(\mathsf{y}) \\
\textsc{deps}(\mathsf{x} = \mathsf{y} \circ \mathsf{z}) &= \{\mathsf{y}, \mathsf{z}\} \cup \textsc{deps}(\mathsf{y}) \cup \textsc{deps}(\mathsf{z}) \\
\textsc{deps}\big(..\mathsf{x}.. = \mathsf{m}(\mathsf{z}_1, .., \mathsf{z}_n)\big) &= \{\mathsf{z}_i \mid 1 \le i \le n\} \\
&\quad \cup \bigcup_{1 \le i \le n} \textsc{deps}(\mathsf{z}_i) \\
\textsc{deps}(stmt) &= \emptyset \text{ otherwise.}
\end{aligned}
$$

Figure 4: Dependency data flow equations

$$
\begin{aligned}
Trc \ ::= &\ \langle \text{Id} \rangle = \langle \text{Arith} \rangle \mid \langle \text{Id} \rangle = \langle \text{Call} \rangle \\
&\mid \mathsf{fit}(\langle \text{Id} \rangle \sim \langle \text{Id} \rangle) \mid \mathsf{change}(\langle \text{Id} \rangle)
\end{aligned}
$$

The calls to the fit method record the conditions of traversed branches, compare Algorithm 2. Interpreting the method as *fitness* function allows the concrete phase to measure how close each condition is to satisfaction, and to modify the inputs accordingly. However, only some of the branch conditions in the trace may require heuristic solving. Therefore, only inputs that affect such conditions should be modified; the other inputs should remain constant, set to the values determined by symbolically solving the path condition. To distinguish these cases, the RESTOREPCANDMARKUSES function therefore marks the variables appearing in constraints that made the path condition undecidable with a call to the change method. Besides marking the variables in the trace, the method has no effect.

In preparation for heuristic search, the concrete phase thus determines which inputs of the trace should be modified to satisfy the branch conditions. As discussed, only inputs that influence a fit call have to be modified, and only inputs that influence a change call should be modified. Both sets of input variables can be determined with the help of the DEPS function, which assigns each statement in the trace the set of variables upon which it depends. Figure 4 shows the data-flow equations that define the DEPS function.

Recall that every variable in the trace is assigned at most once because the STMT function ensures unique variable names. Each variable x that is not an input parameter therefore possesses a unique statement $stmt \in Trc$ in which x appears on the left-hand-side of the equals sign. Let DEF(x) denote this statement, and let P denote the set of trace inputs. Using the fit and change methods to find variables of interest, we determine the set R of inputs that require heuristic solving as

$$R = P \cap \bigcup_{\mathsf{x} \in R_0} \textsc{deps}(\mathsf{x})$$

with $R_0 = \{\mathsf{x} \mid \mathsf{change}(\mathsf{x}) \in Trc\}$. Likewise, the set B of inputs that influence branches along the path is

$$B = P \cap \bigcup_{\mathsf{x} \in B_0} \textsc{deps}(\mathsf{x})$$

with $B_0 = \{\mathsf{x}, \mathsf{y} \mid \mathsf{fit}(\mathsf{x} \sim \mathsf{y}) \in Trc\}$. Combining these, the set of input parameters that the heuristic search should modify is $B \cap R$. Input parameters in $P \setminus B$ can be ignored because they do not influence branches. Input parameters in $P \setminus R$ were not part of any approximation during the symbolic phase. The values determined by the symbolic solver therefore satisfy

```
 1 void dart(int x, int y) {
 2   int a = x * x * x;
 3   int b = y + 3;
 4   if (a == b) {
 5     error();
 6   }
 7 }
 8 void external(float u) {
 9   int v = Float.floatToRawIntBits(u); // native method
10   if (v == 0) {
11     error();
12   }
13 }
```

Figure 5: Two methods that are problematic for symbolic execution, but easy for concolic execution. Method dart contains non-linear integer arithmetic, which is undecidable in general. Method external contains a call to an external method about which the symbolic solver cannot reason.

the dependent branches. However, they *may* be changed to satisfy branches more easily. Their deterministic solutions can serve to seed the heuristic search.

In summary, the program trace models a potentially feasible execution path in the program. Using the fit method, a heuristic search algorithm can determine how close a set of inputs is to satisfying the conditions of traversed branches. It should modify the input parameters in the set $B \cap R$ to move towards a solution.

5. DISCUSSION

We now compare symcretic execution to related techniques, using examples to illustrate differences and similarities.

Comparison with Symbolic Execution

Like concolic execution, symcretic execution is strictly more powerful than symbolic execution. Consider the two methods shown in Figure 5, which are variations of common [19, 27] examples that demonstrate how concolic execution [19, 31] overcomes limitations of pure symbolic execution. Assuming for a moment that the decision procedure only handles linear constraints, pure symbolic execution tools [28, 5] fail to find inputs that trigger the error in either method. In contrast, concolic execution tools [19, 31, 32], can simplify the cubic term in line 2 of method dart by replacing x with its concrete value, say, 2 in the path condition. The resulting constraint $8 = y+3$ is linear; solving it yields the desired input value y=5 for x=2. Likewise, issuing the native method call with the default input u=0.0f shows that this is the desired solution.

While the first phase of symcretic execution is symbolic and therefore faces the same problems as symbolic execution, the second phase provides a fall-back mechanism that allows it to solve some of the problematic constraints. In method dart, for example, the symbolic phase struggles with the cubic term as it explores the program from the error statement in line 5 towards the method's beginning. It therefore omits the constraint from the path condition and finds a potentially feasible execution path. The trace along this path consists of the statements in lines 2–3, a call marking the target fit(a, '==', b), and a call marking the variable a as requiring randomization:

```
 1 void unreachable(int x1, int x2, int x3 ..., int xn) {
 2   int y = 0;
 3   if (x1 > 0) { y = y + 1; } else { y = y + 2; }
 4   ...
 5   if (xn > 0) { y = y + 1; } else { y = y + 2; }
 6
 7   if (y > 0) {
 8     if (y == 0) {    // Error condition for, e.g., division−by−zero
 9       error();
10     }
11   }
12 }
```

Figure 6: Program with an unreachable error condition in line 9. While symcretic execution recognizes the unreachability after two steps, concolic execution explores 2^n execution paths before giving up.

```
1 void trace(x, y) {
2   a = x * x * x;
3   change(a);
4   b = y + 3;
5   fit(a, '==', b);
6 }
```

Evaluating the trace, the second symcretic execution phase uses heuristic search to try and find inputs that satisfy all branch targets. For the external method, the trace consists of just the external method call; as with concolic execution, trying the default value u=0.0f reveals it as solution. For both examples, symcretic execution therefore finds matching inputs.

Comparison with Concolic Execution

Unlike concolic execution, symcretic execution can avoid exploring irrelevant paths, for example if the target is unreachable as in the unreachable method shown in Figure 6. The method contains an error condition that is prevented by a guarding if-statement. Trying to find inputs that trigger the error, symcretic execution starts its symbolic phase at the error statement in line 9 and starts stepping backwards. It first adds the constraint $y = 1$ to the path condition, and next $y > 0$, which yields the unsatisfiable path condition $y = 1 \wedge y > 0$. This two-step search path is branch-free; the search thus explored (the first segments of) the *only* backwards path towards the method entry. As a consequence, symcretic execution ends after these two steps with a proof that the error in line 9 cannot occur.

Concolic execution starts its exploration of the unreachable method at the top. Once the execution has passed the initial computation, which can be long and contain many branches, it arrives at the if-statement in line 7. Assuming that y > 0 holds, the execution cannot explore the (unreachable) branch in the next line, leading to a path condition $\Phi \wedge y > 0 \wedge y \neq 0$, where Φ describes the path above the if-statement. If the concolic execution follows the common exploration strategy [31], it tries to derive the next set of inputs by inverting the last constraint in the path condition and solving it. However, the new path condition is unsatisfiable—it contains both $y > 0$ and $y = 0$—leading to backtracking. As concolic execution cannot recognize the unreachability of the target statement, this repeats for every constraint in Φ. Concolic execution therefore explores up to $2^{|\Phi|}$ irrelevant paths in the method before giving up.

```
 1 void slicing(int x1, int x2, int x3 ..., int xn) {
 2   // None of the blocks uses or defines y
 3   if (x1 > 0) { ... } else { ... }
 4   ...
 5   if (xn > 0) { ... } else { ... }
 6
 7   int y = 0;
 8   if (y == 1) {
 9     error();
10   }
11 }
```

Figure 7: Program for which slicing improves symcretic execution.

```
1 void narrow(int x) {
2   int y;
3   if (x >= 0) { y = x; } else { y = −x; }      // y = Math.abs(x);
4   if (y < 0) {
5     error();                 // Reachable for x = Integer.MIN_VALUE
6   }
7 }
```

Figure 8: Program that is problematic for search-based software testing, but not for symcretic execution. The narrow branch condition in line 4 relies on an artifact of machine arithmetic. The solution is hard to discover for heuristic search, but not for symbolic bit-vector solvers.

In some cases, guiding concolic execution [12] via data dependencies can reduce the number of paths that are explored before the search gives up. However, even with this reduction, the number of explored irrelevant paths can still be large. In our (admittedly contrived) example, the branch condition in line 8 that prevents covering the target statement depends on every block of the preceding if-statements. The guidance therefore achieves no reduction at all.

Comparison with Backward Slicing

A *(backward) slice* of a program with respect to a slicing criterion consists of all the statements in the program upon which the criterion depends [33]. Slices are therefore similar to the traces that symcretic execution collects along the followed execution path. Similar to a *dynamic* slice, the trace follows a single execution path. Unlike slicing, the trace is not fixed by the program inputs, but by the path condition—which represents the class of all program inputs for this path at once. A further, more important difference is that the slice is a partial program, whereas the trace is a straight-line sequence of statements in which all control-flow has been *unrolled*.

Symcretic execution currently does not slice the program. However, slicing can accelerate symcretic execution by reducing the number of paths that have to be explored. For example, when targeting the error statement in line 9 of the slicing method in Figure 7, slicing removes the n irrelevant conditionals in the lines 2–5. Having much of the necessary information for slicing available during symcretic execution, we plan to integrate it in future work.

Comparison with Search-Based Software Testing

Search-based software testing (SBST) [25] finds test inputs that meet a coverage criterion by iteratively selecting inputs that, according to a fitness function, seem closer to a solution. In contrast to our focus on primitive values, inputs can vary in granularity, ranging from primitive values to method sequences for constructing objects. Common heuristics for finding better inputs are genetic algorithms, as well as the Alternating Variable Method [21]. The concrete phase of symcretic execution can be regarded as a special instance of applying SBST to the program trace.

Heuristic search can be slow in discovering the specific solutions of narrow branch conditions. For example, the method narrow in Figure 8 fails if called with the minimal value for integers because, in two's complement, the additive inverse of the smallest integer does not fit into the available bits. Therefore, it is $x = -x$, but $x \neq 0$. This exceptional behavior for one out of 2^{32} integers (assuming 32-bit) is problematic for heuristic search because the fitness function will typically optimize the condition $x = -x$ for the solution x=0. However, symbolic solvers that support bit-vector arithmetic know about these special cases and can solve the conditions directly. Assuming such a solver, the symbolic phase therefore gives symcretic execution an advantage over SBST.

6. IMPLEMENTATION

We have implemented symcretic execution of a subset of Java in a tool called Cilocnoc (*concolic* backwards). Given the class files of a program and a call site of a special CILOCNOC_TARGET() marker function, the tool tries to find inputs for any of the program's public methods that trigger the call.

Cilocnoc fully supports arithmetic on primitives and method calls. It furthermore implements limited support for objects. However, input objects are described as simple textual object graphs; the tool does does not solve the object-creation problem [34]. Another limitation is lack of support for arrays and static fields. We plan to add support for these in the future.

Cilocnoc relies on WALA [13] to read class files, build the program call graph, and construct control-flow graphs. The symbolic backward execution engine of Cilocnoc uses Z3 [9] to solve primitive constraints, and a custom solver for object-shape constraints. The heuristic phase finds inputs with the *Concolic Walk* algorithm [10].

7. EVALUATION

In this section, we empirically compare our implementation of symcretic execution (Cilocnoc) against two other input generators: *Symbolic PathFinder*[4] [28] and jCUTE[5] [30]. To measure the effectiveness and efficiency in generating target-specific inputs, we define target statements for a set of small programs (Table 1) and count how many search steps each tool takes before either finding inputs that reach the target or giving up (Table 2).

Experiment Setup

Table 1 lists the programs used in our evaluation. Each program represents a specific challenge for symbolic and concolic execution (see section 5). The *dart*, *easy-loop*, and *trityp* programs are examples that appear in related work: *dart* is close to the standard example for concolic execution [19,

[4]http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc
[5]http://os.cs.illinois.edu/software/jcute

Table 1: Programs used to evaluate symcretic execution. The *LoC* column lists the number of source code lines in the program, excluding comments and empty lines. The *If* and *L.* columns show the number of if-statements and and loops in the program, the *T.* column contains the number of targets.

Program	*Description*	*LoC*	*If*	*L.*	*T.*
hard-loop	Figure 1	19	2	1	1
dart	Concolic example	16	2	·	2
unreach	Figure 6	20	11	·	1
slicing	Figure 7	18	10	·	1
narrow	Figure 8	14	2	·	1
easy-loop	Decrementing loop	15	1	1	1
trityp	Triangle classification	49	10	·	3

27] (see Figure 5); *easy-loop* is a simple data-dependent loop that was used to evaluate JAUT [7]; and *trityp* is the classic highly-branching program for classifying triangles. The remaining programs consist of the methods shown in the Figures 1, 6, 7, and 8. In each program, we arbitrarily place target statements that we wish to cover.

We generate inputs for every program using the Cilocnoc, jCUTE, and SPF-CW tools. jCUTE is a classic concolic test generator that relies on a linear constraint solver. SPF-CW is a variant of Symbolic PathFinder that solves complex arithmetic path conditions—including calls to external methods—with the same Concolic Walk algorithm that Cilocnoc employs in its concrete phase. jCUTE and SPF-CW both generate high-coverage test suites for Java programs. Aiming for high overall coverage, neither tool implements a guiding heuristic towards a target statement. However, as discussed in section 5, the data-dependency guidance proposed in prior work [12] would have little impact on the programs in our corpus. All tools explore the program depth-first without depth bound but with a 20 second time limit.

During the input generation, we count the *execution path segments* the tool traverses before reaching the target. A segment is a straight-line sequence of statements between two branching points in the execution path. We choose this metric because it depends less on implementation choices than measuring execution time. Nevertheless, we also report the run times (in seconds) to give some intuition of the usefulness of the tools to programmers. The times exclude the duration of static setup tasks because the values generated by these tasks could (and should) be cached. For jCUTE, the static setup consists of instrumenting the target program's byte code; this adds about 1 second to the processing time of each program. For Cilocnoc, the static setup consists of loading and indexing the JDK class hierarchy, which takes about 1.4 seconds per program on an Intel Core i7 notebook with 2 GB of RAM.

Results: Is Symcretic Execution Effective?

The data in Table 2 shows that Cilocnoc finds inputs for all reachable targets (all except those in *unreach* and *slicing*), which suggests that symcretic execution is effective in finding branch-specific inputs. In contrast, the inputs generated by jCUTE reach just one of three targets in the *trityp* program and the single target in the *easy-loop* program; the other eight targets in the program corpus remain uncovered. The SPF-CW tool performs slightly better: it additionally covers both targets in the *dart* program.

Benefiting from a strong symbolic solver, Cilocnoc uses concrete execution for only three targets: those in the *easy-loop* and *hard-loop* programs, and the second target in the *dart* program, which contains an external method call. The target in the *narrow* program can be covered because the symbolic solver knows about bit-vector arithmetic and the irregularity of negating the smallest integer.

Results: How Efficient is Symcretic Execution?

The results in Table 2 support our hypothesis that symcretic execution is more efficient than concolic and symbolic execution. For all targets, except *trityp* target 3, Cilocnoc explores fewer paths than its competitors and, at the same time, discovers all desired inputs. On the *unreach* program, Cilocnoc benefits from being able to recognize unreachable branches as discussed in section 5: instead of exceeding the time limit, exploring 1,287 (jCUTE) or 766 segments (SPF-CW), it stops after just 0.4 seconds, or one segment. Furthermore, the extraction of loops considerably shortens the explored path on the *easy-loop* and *hard-loop* programs: whereas jCUTE and SPF-CW descend deeply into the respective loops (6,600 and 6,162 for jCUTE, 1,912 and 5,516 segments for SPF-CW), Cilocnoc delegates the loop traversal to the concrete phase after just 7 and 30 segments, which quickly finds a solution.

The results also show that pure symbolic execution has an exploration advantage over concolic execution. Unlike jCUTE, both SPF-CW and Cilocnoc (in the symbolic phase) support backtracking the search state. When a search path becomes infeasible before having reached the target, they can revert the changes of the last branch before descending into another branch of the search tree. In contrast, jCUTE has to re-execute the entire program starting from the beginning. Both SPF-CW and Cilocnoc can therefore explore paths much faster than jCUTE. For example, on the *slicing* program, jCUTE is more than twenty-fold slower than Cilocnoc.

8. RELATED WORK

Backward Execution

Backward execution is common in data-flow analysis. Building on the IFDS data-flow framework [29], Chandra et al. [6] develop a backward analysis called *Snugglebug* that symbolically computes the weakest precondition of a target statement at a program entrypoint. Snugglebug's focus is shrinking the search space by constructing the call graph on-demand. Snugglebug and our approach complement each other: using Snugglebug's search space reduction would accelerate Cilocnoc, while symcretic execution would allow Snugglebug to handle complicated arithmetic constraints, external method calls, and long data-dependent loops.

The *PSE* tool by Manevich et al. [24] likewise uses backward analysis based on IFDS. PSE generates feasible traces to observed typestate violations like a dereferenced null-pointer. It traverses the program backwards and applies the effects of operations to the reversed typestate automaton. PSE only handles pointer assignment and dereferencing; it cannot reason about arithmetic constraints. The analysis over-approximates the program behavior and can thus produce false positives.

Table 2: Number of path segments explored before covering the target. Starless entries in the segments column of Cilocnoc only use the symbolic phase to find inputs. Starred entries ($\star$) also use the concrete phase.

		Segments to cover target			*Total segments explored (Time in sec.)*					
Program	*Target*	jCUTE	SPF-CW	Cilocnoc	jCUTE		SPF-CW		Cilocnoc	
hard-loop		–	5455	*22	13340	(20.0)	6480	(20.0)	22	(2.3)
dart	1	–	1	1	3	(0.5)	2	(1.0)	1	(0.2)
	2	–	1	*1	1	(0.5)	1	(1.0)	1	(0.1)
unreach		–	–	–	1287	(20.0)	766	(20.0)	1	(0.4)
slicing		–	–	–	1180	(20.0)	766	(20.0)	512	(0.8)
narrow		–	–	2	8	(2.0)	2	(1.0)	2	(0.2)
easy-loop		861	956	*7	6670	(20.0)	1912	(20.0)	7	(0.3)
trityp	1	8	3	3	55	(3.0)	10	(20.0)	3	(0.3)
	2	–	–	4	55	(3.0)	10	(20.0)	4	(0.1)
	3	–	–	14	55	(3.0)	10	(20.0)	14	(0.1)
Average					*2265*	*(9.2)*	*996*	*(10.7)*	*57*	*(0.5)*

Guided Forward-Execution

Backward analysis is also the foundation for some heuristics that guide symbolic forward-execution towards a goal statement. Similarly to backward slicing [33], Zamfir and Candea [38] compute which control flow edges must be passed to reach the goal. Among the paths containing these edges, they prioritize the paths with the lowest estimated number of operations. The concept is similar to Korel and Ferguson's *chaining approach* [15], which Do et al. [12] use to guide concolic execution towards uncovered code elements. The chaining approach chooses different inputs for a branch's reverse dependencies when it must take the branch but cannot solve it. Ma et al. [22] propose a search heuristic that follows the call-chain backwards from the target method. However, inside each method, they use forward search to find the call site. As discussed in section 5, such guidance heuristics can help to shrink the search space; their integration likely improves the efficiency of symcretic execution. In the context of bug triaging, search-space reduction could also be achieved by incorporating a log-based path-pruning phase similar to that of the SherLog tool by Yuan et al. [37]. In contrast to our approach, SherLog assumes the existence of a log of the failed program run. Analyzing this log, it identifies the set of viable execution paths that could have generated the output and uses symbolic execution to prune the set and derive concrete variable values.

Other heuristics that guide symbolic execution typically try to increase the overall program coverage [32, 4, 35]; they do not help to find a single target statement.

Constraint Logic Programming

Constraint logic programming (CLP) adds numerical constraints to logic programming. CLP therefore supports the major components of symbolic execution: inference with backtracking, symbolic reasoning over numerical values, and symbolic reasoning over data structures (terms). Building on this support, Gómez-Zamalloa et al. show how to obtain a test-case generator for bytecode programs by compiling the bytecode into CLP rules [20].

While symbolic execution can be expressed naturally in CLP, the approach of Gómez-Zamalloa et al. simply delegates the scalability challenges of symbolic execution to the CLP environment. For example, there is no clear way of how unsolvable constraints from native code or non-linear arithmetic should be handled.

9. CONCLUSIONS

Program inputs that cover a specific target are useful in debugging and regression testing. Symcretic execution combines symbolic backward execution and concrete forward execution to efficiently find targeted inputs even if a program contains complicated arithmetic, external method calls, or data-dependent loops. An experimental evaluation shows that symcretic execution finds inputs in more relevant cases than concolic testing tools while exploring fewer path segments.

Future Work

The formalization of symcretic execution in subsection 4.2 prescribes no specific order in which choices should be explored. While simplifying the exposition, this can make the search for a feasible execution path inefficient. In future work, we plan to integrate heuristics for steering the exploration, for example by preferring choices with shorter distance to an entrypoint in the call graph and less loops [6]. Additionally, we plan to support conflict-driven back-jumping in the exploration, as well as the lazy expansion of called methods [18, 1, 23]. Concerning our Cilocnoc tool, we plan to complete the support for objects, and add support for arrays and static fields. Finally, a tight integration with Cilocnoc would enable FindBugs to automatically generate test cases for discovered problems, allowing developers to focus on these true positives.

Acknowledgments

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] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *TACAS'08*, volume 4963 of *LNCS*, pages 367–381. Springer, 2008.

[2] C. Andersson and P. Runeson. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Trans. Software Eng.*, 33(5):273–286, 2007.

[3] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10(6):234–245, Apr. 1975.

[4] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE'08*, pages 443–446. IEEE, 2008.

[5] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*, pages 209–224. USENIX Association, 2008.

[6] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI'09*, pages 363–374. ACM, 2009.

[7] F. Charreteur and A. Gotlieb. Constraint-based test input generation for java bytecode. In *ISSRE'10*, pages 131–140. IEEE Computer Society, 2010.

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

[9] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[10] P. Dinges and G. Agha. Solving complex path conditions through heuristic search on induced polytopes. In *SIGSOFT FSE'14*. ACM, 2014.

[11] P. Dinges and G. Agha. Targeted test input generation using symbolic–concrete backward execution. In *ASE'14*. ACM, 2014.

[12] T. Do, A. C. M. Fong, and R. Pears. Precise guidance to dynamic test generation. In *ENASE'12*, pages 5–12. SciTePress, 2012.

[13] J. Dolby, S. J. Fink, and M. Sridharan. T. J. Watson libraries for analysis (WALA). `http://wala.sf.net`.

[14] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Software Eng.*, 26(8):797–814, 2000.

[15] R. Ferguson and B. Korel. Software test data generation using the chaining approach. In *ITC'95*, pages 703–709. IEEE, 1995.

[16] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *SIGSOFT FSE'11*, pages 416–419. ACM, 2011.

[17] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *ISSTA'13*, pages 291–301. ACM, 2013.

[18] P. Godefroid. Compositional dynamic test generation. In *POPL'07*, pages 47–54. ACM, 2007.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI'05*, pages 213–223. ACM, 2005.

[20] M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test case generation for object-oriented imperative languages in CLP. *TPLP*, 10(4-6):659–674, 2010.

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

[22] K.-K. Ma, Y. P. Khoo, J. S. Foster, and M. Hicks. Directed symbolic execution. In *SAS'11*, volume 6887 of *LNCS*, pages 95–111. Springer, 2011.

[23] R. Majumdar and K. Sen. Latest: Lazy dynamic test input generation. Technical Report UCB/EECS-2007-36, UC Berkeley, March 2007.

[24] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: Explaining program failures via postmortem static analysis. In *SIGSOFT FSE'04*, pages 63–72. ACM, 2004.

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

[26] C. S. Păsăreanu and N. Rungta. Symbolic PathFinder: symbolic execution of java bytecode. In *ASE'10*, pages 179–180. ACM, 2010.

[27] C. S. Păsăreanu, N. Rungta, and W. Visser. Symbolic execution with mixed concrete-symbolic solving. In *ISSTA'11*, pages 34–44. ACM, 2011.

[28] 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.

[29] T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*, pages 49–61. ACM, 1995.

[30] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV'06*, volume 4144 of *LNCS*, pages 419–423. Springer, 2006.

[31] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/SIGSOFT FSE'05*, pages 263–272. ACM, 2005.

[32] N. Tillmann and J. de Halleux. Pex—White box test generation for .NET. In *TAP'08*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.

[33] F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[34] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *ICSE'11*, pages 611–620. ACM, 2011.

[35] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN'09*, pages 359–368. IEEE, 2009.

[36] Z. Xu and G. Rothermel. Directed test suite augmentation. In S. Sulaiman and N. M. M. Noor, editors, *APSEC'09*, pages 406–413. IEEE Computer Society, 2009.

[37] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error diagnosis by connecting clues from run-time logs. In J. C. Hoe and V. S. Adve, editors, *ASPLOS'10*, pages 143–154. ACM, 2010.

[38] C. Zamfir and G. Candea. Execution synthesis: A technique for automated software debugging. In *EuroSys'10*, pages 321–334. ACM, 2010.