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 [9, 1]. 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 [12, 20, 21, 18, 10]. However, it has been argued that this yields test inputs that are often of limited use for developers [11]. 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 [3, 24], or to verify that a suspicious instruction pattern is an actual problem, or to add a test case to cover a specific code change [23].

A natural approach for finding targeted inputs is to use symbolic backward execution (SBE) [2, 3, 4]. 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, 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 [12, 20], we show how to combine symbolic backward execution with concrete 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 [5], 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.
public void challenges(int x, double u) {
    int res = 0;
    int i = 0;
    while (i < x) {
        int tmp = i % 2;
        if (tmp == 0) {
            res = res + 17;
        } else {
            res = res - 1;
        }
        i++;
    }
    if (res == 8192) { // Error condition 1
        throw new AssertionError();
    } else ...
    // Long and deep computation
    else ...
    // Long and deep computation
}

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.

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, 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. 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 4) and evaluate an implementation of our algorithm on a corpus of small programs (section 5). 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 concrete-symbolic (concolic\(^1\)) execution [12, 20]. 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 [2, 3, 4]: 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

\[
tmp = 0 \land \text{tmp mod 2} \land \text{i < x} \land i = 0 \land \text{res} = 0.
\]

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 [5] 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 \text{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, symbolic-concrete (symcretic) execution, consists of the two phases outlined in this section. A detailed description and formalization of both phases is available in the full version of this paper.

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.

\(^1\)Concolic execution is also known as Directed Automated Random Testing and Dynamic Symbolic Execution.

\(^2\)Satisfiability Modulo Theories
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 [21] 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 $\text{Math.sin}(u) > 0 \land \text{res} = 8192$, 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:

```java
public void simplified_challenges(int x, double u) {
    int res = x + 23;
    if (res == 8192) {
        if (Math.sin(u) > 0) {
            throw new AssertionError();
        }
    }
}
```

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**

Like concolic execution, symcretic execution is stronger than symbolic execution because of its ability to mitigate solver limitations through concrete 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 3. 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 state and begins stepping backwards. It first adds the constraint $y = 1$ to the path condition, and next $y > 0$, which yields the unsatisfiable path condition $y = 1 \land y > 0$. 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,

```java
int res = x + 23;
if (res == 8192) {
    if (Math.sin(u) > 0) {
        throw new AssertionError();
    }
}
```
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) {
  9          error();
 10      }
 11  }
12 }

Figure 3: Program with an unreachable error condition in line 9. While symcretic execution recognizes the unreachable after two steps, concolic execution explores $2^n$ execution paths 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 4: Program for which slicing improves symcretic execution.

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 $\Phi \
\wedge y > 0 \land y \neq 0$, where $\Phi$ describes the path above the if-statement. If the concolic execution follows the common exploration strategy [20], 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 $\Phi$. Concolic execution therefore explores up to $2^n$ irrelevance paths in the method before giving up.

In some cases, guiding concolic execution [7] 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 [22]. 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 4, 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) [17] 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 [14]. 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 5 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.

5. EVALUATION

We now compare our implementation of symcretic execution with two other input generators: Symbolic PathFinder (SPF) [18] and jCUTE [19]. 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.
easy-loop is a simple data-dependent loop that was used to the tool traverses before reaching the target. A segments

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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>LoC</th>
<th>If</th>
<th>L.</th>
<th>T.</th>
</tr>
</thead>
<tbody>
<tr>
<td>hard-loop</td>
<td>Figure 1</td>
<td>19</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>dart</td>
<td>Concolic example</td>
<td>16</td>
<td>2</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>unreachable</td>
<td>Figure 3</td>
<td>20</td>
<td>11</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>slicing</td>
<td>Figure 4</td>
<td>18</td>
<td>10</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>narrow</td>
<td>Figure 5</td>
<td>14</td>
<td>2</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>easy-loop</td>
<td>Decrementing loop</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>trityp</td>
<td>Triangle classification</td>
<td>49</td>
<td>10</td>
<td>-</td>
<td>3</td>
</tr>
</tbody>
</table>

Experiment Setup

We have implemented symcretic execution of a subset of Java in a tool called Cilocnoc (concolic backwards). Cilocnoc relies on WALA [8] to process class files. The symbolic backward execution engine of Cilocnoc uses Z3 [5] to solve primitive constraints, and a custom solver for object-shape constraints. The heuristic phase finds inputs using the Concolic Walk algorithm [6].

Table 1 lists the programs used in our evaluation. Each program represents a specific challenge for symbolic and concolic execution (see section 4). The dart, easy-loop, and trityp programs are examples that appear in related work: dart is close to the standard example for concolic execution [12]; easy-loop is a simple data-dependent loop that was used to evaluate JAUT [4]; and trityp is the classic highly-developing program for classifying triangles. The remaining programs consist of the methods shown in the Figures 1, 3, 4, and 5. 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 4, the data-dependency guidance proposed in prior work [7] 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?

Cilocnoc finds inputs for all reachable targets, 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 of our experiments support the hypothesis that symcretic execution is more efficient than concolic and symbolic execution. For all targets, except one case in trityp, Cilocnoc explores fewer path segments than its competitors and, at the same time, discovers all desired inputs. On the unreachable program, Cilocnoc benefits from being able to recognize unreachable branches as discussed in section 4: 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.

6. RELATED WORK

Backward execution is a common technique in data-flow analysis. Building on the IFDS data-flow framework, Chandra et al. [3] develop a backward analysis called Snugglebug that symbolically computes the weakest precondition of a target statement at a program entry point. Snugglebug’s focus is shrinking the search space by lazily constructing the call graph. 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. Manevich et
al. [16] use backward analysis based on IFDS to find typestate violations. However, the approach is limited to pointer operations and cannot reason about arithmetic constraints.

Constraint logic programming (CLP) 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 to CLP rules [13]. However, it is unclear how to extend the approach for handling native code or complicated non-linear arithmetic.

Backward analysis is also the foundation for some heuristics that guide symbolic forward-execution towards a target statement. Similarly to backward slicing [22], Zamfir and Candea [24] compute which control flow edges must be passed to reach the target. Among the paths containing these edges, they prioritize the paths with the lowest estimated number of operations. Ma et al. [15] propose a search heuristic that follows the call-chain backwards from the target method. However, inside each method, it uses forward search to find the call site. Do et al. [7] use the chaining approach 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.

7. 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.

In future work, we plan to accelerate the search for potentially feasible paths by integrating heuristics that steer the search towards entry points, by supporting conflict-driven back-jumping, and by lazily expanding called methods. Furthermore, we plan to complete the support for objects, and add support for arrays and static fields to the Cilocnoc tool.

8. 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.

9. REFERENCES


