SYMCRETIC TESTING OF PROGRAMS

BY

PETER DINGES

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Abstract

Targeted inputs are input values for a program that lead to the execution of a user-specified branch or statement. Targeted inputs are useful: In debugging, for example, they allow programmers to follow the execution towards the program point where a bug occurred. In testing, they constitute a test case that covers a new piece of code. A natural approach to find targeted inputs is symbolic backward execution. However, symbolic backward execution struggles with complicated arithmetic, external method calls, and data-dependent loops that occur in many real-world programs.

This dissertation describes *symcretic execution*, a novel method for efficiently finding targeted inputs. Symcretic execution overcomes the limitations of symbolic backward execution by integrating it with concrete forward execution. The approach consists of two phases: In its first phase, symcretic execution uses symbolic backward execution to find a feasible execution path from the given target to any of the program's entry points. Unlike prior approaches, symcretic execution 'skips' over constraints that are problematic for the symbolic decision procedure and defers their solution until the second phase. The second phase of symcretic execution begins when the symbolic execution reaches an entry point. In this phase, symcretic execution uses concrete forward execution and heuristic search to find inputs that satisfy the constraints that were skipped in the first phase. A comparison with related approaches and an empirical evaluation suggest that symcretic execution finds more inputs that result in relevant executions while avoiding the exploration of uninteresting paths.

The heuristic search algorithm employed in the second phase of symcretic execution must be able to handle complicated arithmetic and external method calls. The dissertation therefore introduces an algorithm called *concolic walk*. The concolic walk algorithm also applies to solving path conditions in customary symbolic and concolic execution and is thus presented in this more general setting. The concolic walk algorithm is a heuristic search based on a geometric interpretation of the task of finding inputs. An evaluation of the algorithm shows that it finds more solutions (and hence improves coverage) than the simplification-based heuristics that have been used in concolic testing. Moreover, the concolic walk algorithm improves the effectiveness of state-of-the-art concolic test generators that are using powerful specialized constraint solvers.

Thank you for taking interest
in my research.

Table of Contents

Chapter 1

Introduction

Our information society stands on fickle ground. Errors pervade the software we use to write letters, buy books, listen to music, and talk to our family, friends, and customers. Software errors crash rockets [44] and burn about $60 billion through lost business every year [80].

Many software errors can be prevented by thorough testing, but testing is rarely thorough because manually defining test cases is time-consuming and expensive. Already, testing accounts for half of the costs of software development [9]. Test generators (e.g., [66, 68, 18, 81, 40]) address this problem by automating parts of the task. The NIST[1] estimates that automated testing tools can reduce the cost of software testing by about one third, saving billions of dollars [80].

The goal of automatic test generation is finding a set of input values that cover—lead to the execution of—a set of target statements in the program. If the intention is to create a comprehensive test suite for the program, the target set is any yet uncovered branch or statement. If the intention is to add a new test case after a change to the program, the set may be a singleton that only contains the changed statement. If the intention is to find bugs, the target set may consist of suspicious statements such as `1/x`, which may trigger a division by zero error if there are inputs that set `x` to zero.

Regardless of the intention, efficient test generation is challenging because programs often accept an astronomical number of input value combinations.[2] Yet, the program processes many of these combinations along the same execution path. For example, the conditional branch `if (x > 0)` processes all `x`

[1]National Institute of Standards and Technology

[2]A single 32-bit integer variable can have more than $8 \cdot 10^9$ different values. Two support over $(8 \cdot 10^9)^2$ combinations.

greater than zero along the true–path. Such inputs are equivalent in terms of testing; they lead to observing the same behavior. Including more than one representative from an equivalence class means repetition without gain.

Knowing the input equivalence classes can thus help to make testing efficient as it allows avoiding repetition. The input classes of a program are determined by the constraints that its conditional branch statements impose on the inputs along the execution paths. For example, the conditional `if (x > 0)` establishes two classes of inputs `x`: those greater than zero, and those at most zero. Finding the input classes hence reduces to collecting the constraints along the execution paths in the program [66]. The one needed representative input combination for each path can be found by solving the constraints.

Unfortunately, there are two major challenges:

1. The set of constraints along an execution path is undecidable in general because it may contain arbitrary non-linear integer constraints [24]. Even if the constraints are decidable, solving them may be computationally infeasible.

2. If the program contains loops, the number of execution paths can be infinite. Even if all loops are bounded, the number of execution paths can still grow exponentially in the path length.

Randomization and a combination of concrete execution with the symbolic solving of the path constraints have been used to mitigate these challenges when generating test suites [50, 91]. The resulting *conc*rete–symb*olic* (concolic) approaches generate inputs that cover large parts of the program. However, if execution paths contain complex arithmetic constraints, existing ways of combining concrete and symbolic solving can be too weak to find satisfying inputs. Furthermore, concolic approaches focus on overall coverage, which limits their efficiency when trying to cover a single specific target. There are two sources of inefficiency:

1. Guiding the concrete execution that drives these approaches towards the target requires choosing the right inputs ahead of time. Wrong inputs lead to the exploration of irrelevant program paths.

2. The approaches lack a concept of unreachability. Even if an if-statement such as `if (x > 0)` right before the target, for example a potential division by zero such as `1/x`, prevents covering the target, the explorative approaches do not stop; they keep exploring paths in the hope of finding covering inputs.

The goal of the research described in this dissertation is to find an effective and efficient way of generating target-specific inputs without sacrificing the advantages of concolic testing. To be effective in finding inputs, the developed approach closely integrates symbolic and heuristic constraint solving. The resulting novel algorithm (chapter 5) also applies to classic symbolic and concolic input generation techniques, where it improves the strength of state-of-the-art approaches. To be efficient in finding inputs, the developed approach uses symbolic and concrete execution in two phases (chapter 3 and chapter 4). The first phase, symbolic backward execution (section 4.2), tries to find a feasible execution path from the target statement to an entry point. Going backwards, it avoids exploring irrelevant paths that do not lead to the target. Whenever the symbolic constraint solver faces an undecidability or exploration problem, the execution skips the symbolic constraints—treating them as *potentially* satisfiable instead of giving up like existing approaches [19, 21]. Once the symbolic execution reaches the beginning of a program entry point, the second phase begins. The second phase, heuristic solving (section 4.3), tries to find input values that satisfy previously problematic constraints by concretely executing a trace of the program along the discovered path.

1.1 Thesis Statement

This dissertation focuses on the problem of generating inputs that cover a user-specified target statement or condition in a program. The dissertation describes a novel solution to the problem, called *symcretic execution*, that can be more effective and efficient than classic input generation techniques. Symcretic execution (1) avoids irrelevant paths that do not reach the target; (2) excludes infeasible paths; (3) mitigates the aforementioned problems of undecidable constraints and data-dependent loops by integrating heuristic constraint solving based on concrete execution of (parts of) the program. The hypothesis investigated in this dissertation then is:

A combination of symbolic backward execution and heuristic constraint solving can achieve high effectiveness and efficiency in generating target-specific inputs for sequential programs that operate on primitive values.

1.2 Contributions

This dissertation contains the following research contributions:

- It defines *symcretic execution*, a novel method for finding inputs that lead to the execution of a specific target in a program. Symcretic execution is the first algorithm that integrates concrete execution into symbolic backward execution.

- It introduces the idea of treating loops in the program as external methods and solving them with heuristic search.

- It describes the *concolic walk* algorithm, a novel combination of symbolic reasoning and heuristic search for solving complex arithmetic path conditions. The algorithm is sound and complete for linear constraints and supports non-linear constraints and calls to native library methods. In the context of solving path conditions, it is the first to use a polytope interpretation for linear constraints, and the first to exploit (potential) piece-wise continuity of non-linear constraints.

- It contains an empirical evaluation of the effectiveness and efficiency of these algorithms when applied to Java programs.

Chapter 2

Background and Related Work

The goal of automatic test generation is finding a set of input values that cover a set of targets in a program. Among others, targets can be statements or branches in the program; an input covers these if it leads to their execution. Randomly generating the input values is fast and cheap [81], but often covers only targets that lie on *common* paths [16]. It fails to discover the *rare*—but all the more interesting—paths because drawing the right combination of inputs from the large set of possible combinations is highly unlikely. For example, narrow branch conditions such as $x = x \cdot y$ are unlikely to be met by random values for x and y. Instead, most generated inputs will execute the same path $x \neq x \cdot y$, resulting in repeated tests of the same program behavior. In general, random exploration of a large search space is prone to miss sparse solutions.

2.1 Symbolic Testing

A systematic approach to generating test inputs is to use *symbolic execution* [66, 67, 14, 22, 60] to explore the program's execution paths. Collecting the constraints along a path and computing their solution yields input values that drive the program down this path[1]. Symbolic execution statically analyzes a program by running it on *symbolic* input values. In contrast to *concrete* values like the integer 42, symbolic values represent *any* element from their domain. Thus, where concrete execution binds the integer inputs x and y to 42 and 23, symbolic execution binds them to $m \in \mathbb{Z}$ and $n \in \mathbb{Z}$. Adding x and y under symbolic execution yields the symbolic value $m + n \in \mathbb{Z}$.

[1]Assuming a deterministic program.

If-statements on the execution path add *constraints*: if the path takes the true-branch of the statement `if (y>0)`, then $n > 0$. The *path condition* is the conjunction of all constraints along the path. Test inputs can be generated by systematically executing the different paths in the program and solving the path condition. As opposed to random testing [81], symbolic execution can therefore reliably discover error-triggering inputs even if their probability of occurrence approaches zero.

The idea of using symbolic execution for testing goes back to King and his interactive symbolic interpreter called EFFIGY [66, 67]. Shortly after EFFIGY's introduction, Boyer [14] and Clarke [22] presented similar systems that replaced interactivity with concrete test input generation. These systems collected the path constraints through forward substitution. At the same time, Howden proposed an (unimplemented) symbolic testing method that collected the constraints using backward substitution [60].

Test generation solely based on symbolic execution [65, 112, 17, 28, 16, 84, 5, 54, 4] is limited by the power of the used constraint solver. Recent generalizations allow symbolic execution to handle references [91] and object-oriented languages [65, 27, 29]. However, if an execution path contains a constraint that lies outside the solver's theory, the test generator cannot find concrete input values that drive the program down this path. Examples of such constraints are cryptographic hash functions and system calls whose effect depends on the program's environment [50]. General decision procedures exist only for rather limited theories; the satisfiability of a system of non-linear integer equations is undecidable in general [24]. Furthermore, constraints whose solution takes too long can be considered undecidable in practice.

2.2 Concolic Testing

Concrete execution does not suffer from the limitations of a constraint solver; to compute the effects of a system call, concrete execution simply issues the call. The insight behind *concolic execution*[2] [113, 50, 91] is that *conc*rete

[2]Concolic execution is also known as *Dynamic Symbolic Execution* (DSE) and *Directed Automated Random Testing* (DART).

and symb*olic* execution complement each other. Concolic testing executes a program on concrete values and collects the symbolic constraints along the followed path. After the program has finished, negating one of the constraints and solving this new path condition yields a set of concrete inputs that drives the program down a different path. Thus, concolic execution covers paths at most once; it avoids the repetitions of random testing. Furthermore, following the concrete execution eliminates the need to symbolically reason about the correct number of loop iterations. At the same time, concrete values allow for simplifying constraints outside the solver's theory. For example, replacing a symbolic value m with its concrete value 23 converts the constraint $mn > 0$ into the easily solvable linear constraint $23n > 0$.

The combination of concrete and symbolic execution has been proposed by Godefroid et al. [50] and extended to programs using pointers by Sen et al. [91]. It has received wide attention in the software engineering community: Godefroid et al.'s paper [50] has been cited over 1300 times; Sen et al.'s paper [91] over 900 times[3]. The approach has proved its ability to cope with realistic sequential programs and is the foundation of several state-of-the-art testing tools used in industry [103, 51, 52].

2.3 Path Condition Solving

A drawback of the simplification of concolic execution is *blind commitment* to a concrete value: after setting $m = 23$, branches further down the execution path that require $m \neq 23$ can no longer be explored [49, 86]. In addition, simplification can yield coarse approximations of the constraints (see chapter 5).

Instead of eagerly committing to a concrete value when a constraint exceeds the solver's capabilities, other blends of symbolic and concrete execution delay simplification. For example, *mixed concrete–symbolic solving* [86] tries to leverage concrete execution to solve undecidable arithmetic constraints during symbolic execution. To solve a path condition, mixed solving splits the path

[3]According to Google Scholar on Apr. 3, 2014.

condition into decidable and *complex* constraints, solves the decidable ones directly, and uses the solution to simplify and concretely execute the complex constraints. The execution results, in turn, serve to simplify the decidable constraints. A similar separation between simple and complex constraints has been used in solvers based purely on randomization and path condition simplification [97, 98]. The concolic walk algorithm introduced in chapter 5 also employs separation but achieves significantly better performance.

In the converse direction, Gotlieb and Petit [55, 56] show how to facilitate random testing of specific program parts by constructing a domain for uniformly sampling inputs that satisfy a path condition. The approach is based on constraint propagation and refutation. It cannot handle uninterpreted functions or bit-wise operations in the path condition.

Other approaches seek to extend the capabilities of arithmetic solvers by integrating concrete execution. The CORAL solver [94] uses Particle-Swarm Optimization [64] to solve constraints that include rich mathematical operations like exponentiation and trigonometric functions. A recent extension [13] improves CORAL's efficiency by seeding the initial solution population through interval solving. FloPSy [70] is a plugin for Pex [103] that solves floating-point constraints with heuristic search. While performing well within their specific domain, these solvers are closed and lack callback interfaces to include in the constraints previously unknown functions such as native methods in Java.

2.4 Path Exploration Heuristics

Efficient path exploration is a central challenge for symbolic and concolic test generators: Loops and recursion can lead to an infinite number of paths. Even if the path length is limited, the branches introduced by if-statements can result in an exponential number of paths.

Finding relevant paths and achieving good coverage with the generated test cases therefore depends on efficient search techniques. One strategy for systematic path exploration is to use depth-first search with iterative deepening [112]. Other heuristics try to increase the overall program coverage, for example through randomization [15], or by picking the *fittest* path for exploration in each iteration [115]. However, it seems unlikely that simple heuristics will be effective in finding rare events in a search space as large as that of the execution paths of a program.

2.5 Backward Execution

Another strategy is to search backwards: starting from the desired event, the search spreads backwards through the program until it reaches an entry point. Backward analysis is the foundation for several heuristics that guide symbolic forward execution towards a target instruction. Similarly to backward slicing [104], Zamfir and Candea [117] 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. Ma et al. [73] propose a search heuristic that follows the call-chain backwards from the target method. Inside each method, they use forward search to find the call site. Do et al. [35] use the *chaining approach* of Ferguson and Korel [38] 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.

Backwards search is a fundamental search strategy of constraint logic programming (CLP). CLP adds numerical constraints to logic programming and therefore supports the major components of symbolic execution: inference with backtracking, symbolic reasoning over numerical values, and symbolic reasoning over data structures (terms). Using backwards search, CLP can therefore be used for the symbolic backwards execution of programs. However, existing approaches [5, 54, 53, 21] delegate the scalability challenges of symbolic execution to the CLP environment. Since the CLP environment is domain-general, it cannot employ optimization to accelerate the search for feasible execution paths. Hence, it is unlikely that the approach will be able to handle realistic programs. In addition, CLP environments are typically closed, meaning that they cannot easily include external functions in the constraints.

Backwards execution is also common in data-flow analysis. Building on the IFDS data-flow framework [88], Chandra et al. [19] develop a backwards analysis that symbolically computes the weakest pre-condition of a target instruction. The tool, called *Snugglebug*, achieves scalability by constructing the call graph on-demand, which shrinks the search space. Like compositional symbolic execution [48, 6], Snugglebug lazily expands called methods [74] and summarizes their effects. The *PSE* tool by Manevich et al. [76] likewise uses backwards analysis based on IFDS. It focuses on detecting typestate [96] violations like dereferenced null-pointers, which it detects by applying the effects of operations to the reversed typestate automaton.

2.6 Search-Based Software Testing

Combinations of heuristic search and symbolic reasoning have been explored in the context of search-based software testing (SBST) [78]. Like concolic testing, SBST searches for test inputs that meet a coverage criterion. In contrast to concolic testing, it relies on heuristic search instead of a symbolic constraint solver to find such inputs. SBST iteratively selects inputs that, according to a fitness function, seem closer to a solution. However, inputs can vary in granularity, ranging from primitive values to method sequences for constructing objects. Common heuristics for finding better inputs are *genetic algorithms* (GA), as well as the *alternating variable method* [68], which is similar to adaptive search [23].

Heuristic search can be slow in discovering the specific solutions of narrow branch conditions like $m = 42$. A number of approaches thus suggests to accelerate the search through symbolic reasoning: The Evacon tool [62] constructs high coverage tests for object-oriented programs by alternating between generating method sequences for object creation via GA, and generating primitive method arguments via concolic execution. Other techniques introduce a mutation operator in the GA that yields new test individuals by concolically executing an existing test and flipping a branch condition [75, 43]. Symbolic execution has also been used to derive fitness functions that represent the search landscape more accurately [8], leading to more efficient search.

Chapter 3

Overview of Symcretic Execution*

This chapter demonstrates the general ideas behind symcretic execution. The next chapter formalizes and evaluates the algorithm. Chapter 5 describes an algorithm that is suitable for the concrete phase of symcretic execution.

3.1 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 3.1. 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 3.1 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 such as concolic execution can be used to find targeted inputs. However, 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 symbolic backward execution [14, 19, 21] (SBE): instead of covering as much as possible, we are interested in covering specific, relevant

*This chapter and the next are based on the paper *Targeted Test Input Generation Using Symbolic–Concrete Backward Execution* by Peter Dinges and Gul Agha, which appeared in the proceedings of 29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014) [32, 33]. Permission to reprint has been granted by ACM. `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 3.1: 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.

targets in a program. Any part of a program that does not contribute to this goal (for example lines 16 and 17 in Figure 3.1) is irrelevant.

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 3.1, the execution starts on this line and steps backwards, collecting the constraint $tmp = 0$. Moving further towards the top, it constructs the path condition

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

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 the challenges mentioned in chapter 1:

1. The modulo operation on line 5 forces state-of-the-art decision procedures such as the Z3 SMT[1] solver [25] 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.

3. The data-dependent loop on line 4 must be traversed 1024 times to yield res = 8192, which may exceed the exploration budget.

3.2 Algorithm Synopsis

Following the general idea of concolic execution, we propose to overcome the aforementioned drawbacks of *sym*bolic backward execution by combining it with con*crete* execution. Symcretic execution consists of two phases:

Phase I: Symbolic Backward Execution

The first phase of symcretic execution uses symbolic backward execution 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.

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,

[1]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 3.2: Program from Figure 3.1 without the loop.

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: Concrete Execution

The second phase of symcretic execution 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 (chapter 5).

We illustrate our approach on the program in Figure 3.2. 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 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 3.1. 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 [103] 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 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. 3.1
  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 3.1. 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 $v > 0$.

Chapter 4

Symcretic Execution

This chapter formalizes and evaluates the symcretic execution algorithm, which, given a program and a target statement, finds inputs that drive the program towards executing the 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[1] calls; see Figure 4.1. 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 entry points.

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 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.

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

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

Figure 4.1: Syntax of the example language, with a set of primitive types Type, variable identifiers Id, literals Lit, method symbols Mthd, binary operations $\circ \in \{+\,,-,*,/,\%,<<\,,>>\}$, and relations $\sim\, \in \{<\,,<=\,,>=,>,==,!=\}$.

4.2 Phase I: Symbolic Phase

The depth-first search for a feasible execution path, formalized as Algorithm 4.1, 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 entry points. 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

Algorithm 4.1: Depth-first exploration of the program. The algorithm drives the symbolic backward execution phase of symcretic execution. In the algorithm, P denotes the current path condition.

```
1: procedure symbolicPhase(stmt)
2:     for $c \in$ choices(stmt) do
3:         updateState($c$)                          ▷ update path condition $P$
4:         if $P$ is unsat then
5:             restoreState($c$)
6:             continue                          ▷ try next choice or backtrack
7:         else if $P$ is unknown then
8:             restorePcAndMarkUses($c$)
9:         end if
10:        if nextStmt($c$) = $\perp$ then                         ▷ at entry
11:            if enterConcretePhase() 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
```

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 P, 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 P, 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 entry point 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 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 P. Restoring P 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 P 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 4.2 defines the UPDATESTATE function used in Algorithm 4.1 that modifies the search state according to a choice structure. The function recog-

Algorithm 4.2: Application of the effects of a choice structure c to the search state, which consists of the path condition P, the program trace *Trc*, and the call stack *Stck*.

```
procedure UPDATESTATE($c$)
    $e \leftarrow \textsc{env}(\textsc{top}(Stck))$                    ▷ local variable environment
    if $c$ is CALLCHOICE($stmt$, $cs$) then                ▷ call with call site $cs$
        if $cs = \bot$ then return                          ▷ entry point
        $m \leftarrow \textsc{method}(stmt)$
        $n \leftarrow \textsc{paramCount}(m)$
        $e \leftarrow e[\textsc{arg}(cs, i) \mapsto e[\textsc{param}(m, i)] \mid 1 \le i \le n]$        ▷ bind args.
        $\textsc{pop}(Stck)$
    else if $c$ is RETCHOICE($stmt, p, r$) then                      ▷ return
        assert $p$ is "x1,...,xn = f(...)"
        if $r$ is "return y1,...,yn" then
            $e' \leftarrow \langle \mathsf{y}_i : e[\mathsf{x}_i] \mid i = 1, \dots, n \rangle$
            $\textsc{push}(Stck, \textsc{stackFrame}(e', \mathsf{f}, p))$          ▷ create callee env.
        else                                                  ▷ external method
            add change($e[\mathsf{x}_1], ..., e[\mathsf{x}_n]$) to front of Trc
            add MAKESTMT($e[\mathsf{x}_1], ..., e[\mathsf{x}_n] = \mathsf{f}(...)$) to front of Trc
        end if
    else if $c$ is PREDCHOICE($stmt, p$) then          ▷ intra-procedural step
        if $p$ is "if (x∼y)" or "while (x∼y)" then
            if $(stmt, p)$ is true branch in the CFG then
                $\varphi \leftarrow e[\mathsf{x}] \sim e[\mathsf{y}]$
            else
                $\varphi \leftarrow e[\mathsf{x}] \not\sim e[\mathsf{y}]$
            end if
            add fit($\varphi$) to front of Trc                      ▷ store taken branch
        else
            if $p$ is "x = y" then
                $\varphi \leftarrow e[\mathsf{x}] = e[\mathsf{y}]$
            else if $p$ is "x = $\ell$" then
                $\varphi \leftarrow e[\mathsf{x}] = \ell$
            else if $p$ is "x = y ∘ z" then
                $\varphi \leftarrow e[\mathsf{x}] = e[\mathsf{y}] \circ e[\mathsf{z}]$
            end if
            $e \leftarrow e[\mathsf{x} \mapsto \bot]$
            add MAKESTMT($\varphi$) to front of Trc
        end if
        $P \leftarrow P \wedge \varphi$
    end if
end procedure
```

nizes three kinds of choice structures: call, return, and predecessor choices, which are constructed by the callChoice, retChoice, and predChoice 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 after binding the arguments at the call site to the method parameters. 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.

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, makeStmt($x_0 = y_0$) = "x0 = y0;".

Choice Generation

Algorithm 4.3 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 a 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 entry point (line 7). In this case, the search found a potentially feasible path and the concrete phase of symcretic execution begins. Recall that using $\perp$ as next statement tells Algorithm 4.1 that a feasible path was found. (3) No other method called m, but m is not an entry point. 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 $\perp$ 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 4.1 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

Algorithm 4.3: 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:              $cs \leftarrow$ CALLSITE(TOP($Stck$))                  ▷ jump to caller
5:              return {CALLCHOICE($stmt, cs$)}                        ▷ single choice
6:          end if
7:          if $m \in Entr$ then                                       ▷ reached entry point
8:              return {CALLCHOICE($stmt, \bot$)}
9:          end if
10:         $C \leftarrow$ {CALLCHOICE($stmt, cs$) | $cs \in$ CALLSITES($m$)}
11:         return $C$                                                 ▷ all possible callers
12:     end if
13:     if $stmt$ is loop exit and loop is new then
14:         $h \leftarrow$ LOOPHEADER($stmt$)
15:         $l \leftarrow$ LOOPMETHOD($stmt$)
16:         $C \leftarrow$ {RETCHOICE($stmt, h, l$)}
17:     else
18:         $C \leftarrow \emptyset$
19:     end if
20:     for $p \in$ CFGPRED($stmt$) do                                 ▷ traverse body
21:         if $\tau[stmt, p] \geq$ L then continue                     ▷ edge traversed too often
22:         $\tau[stmt, p] \leftarrow \tau[stmt, p] + 1$
23:         if $stmt$ is "x1,...,xn = m(y1,...,yn)" then              ▷ method call
24:             $C \leftarrow C \cup$ {RETCHOICE($stmt, p,$ RETSTMT(m))}
25:         else
26:             $C \leftarrow C \cup$ {PREDCHOICE($stmt, p$)}
27:         end if
28:     end for
29:     return $C$
30: end procedure
```

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 Phase

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 introduced in chapter 5. 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 4.2 and the calls to the function restorePcAndMarkUses appearing in Algorithm 4.1. 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:

$$\mathit{Trc} \ ::= \ \langle \text{Id} \rangle = \langle \text{Arith} \rangle \mid \langle \text{Id} \rangle = \langle \text{Call} \rangle \mid \mathsf{fit}(\langle \text{Id} \rangle \sim \langle \text{Id} \rangle) \mid \mathsf{change}(\langle \text{Id} \rangle).$$

The calls to the fit method record the conditions of traversed branches, compare Algorithm 4.2. Interpreting the method as *fitness* function allows the concrete phase to measure how close each condition is to satisfaction, and to

$$
\begin{aligned}
\textsc{deps}(\mathsf{x = y}) &= \{\mathsf{y}\} \cup \textsc{deps}(\mathsf{y}) \\
\textsc{deps}(\mathsf{x = y \circ z}) &= \{\mathsf{y}, \mathsf{z}\} \cup \textsc{deps}(\mathsf{y}) \cup \textsc{deps}(\mathsf{z}) \\
\textsc{deps}\big(\mathsf{..., x, ... = m(z_1, ..., z_n)}\big) &= \{\mathsf{z}_i \mid 1 \leq i \leq n\} \\
&\quad \cup \bigcup_{1 \leq i \leq n} \textsc{deps}(\mathsf{z}_i) \\
\textsc{deps}(stmt) &= \emptyset \text{ otherwise.}
\end{aligned}
$$

Figure 4.2: Dependency data flow equations.

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 4.2 shows the data-flow equations that define the deps function.

Recall that every variable in the trace is assigned at most once because the makeStmt 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 $\textsc{def}(\mathsf{x})$ denote this statement, and let I 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

$$R = I \cap \bigcup_{\mathsf{x} \in R_0} \text{DEPS}(\mathsf{x})$$

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

$$B = I \cap \bigcup_{\mathsf{x} \in B_0} \text{DEPS}(\mathsf{x})$$

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

4.4 Discussion

This section compares 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 4.3, which are variations of common [50, 86] examples that demonstrate how concolic

```
 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 4.3: 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.

execution [50, 91] overcomes limitations of pure symbolic execution. Assuming for a moment that the decision procedure only handles linear constraints, pure symbolic execution tools [87, 16] fail to find inputs that trigger the error in either method. In contrast, concolic execution tools [50, 91, 103], 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 4.4: 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.

```
void trace(x, y) {
  a = x * x * x;
  change(a);
  b = y + 3;
  fit(a, "==", b);
}
```

Evaluating the trace, the second symcretic execution phase uses heuristic search to 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 4.4. 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 $y = 0$ to the path condition, and next $y > 0$, which yields the unsatisfiable path condition $y = 0 \wedge y > 0$. 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 $P \wedge y > 0 \wedge y \neq 0$, where P describes the path above the if-statement. If the concolic execution follows the common exploration strategy [91], 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 P. Concolic execution therefore explores up to $2^{|P|}$ irrelevant paths in the method before giving up.

In some cases, guiding concolic execution [35] 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 [104]. Slices are therefore similar to the traces that symcretic execution collects along the followed execution path. Similar to a *dynamic* slice, the trace follows

```
 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 4.5: Program for which slicing improves symcretic execution.

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 4.5, 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) [78] 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 [68]. The concrete phase of symcretic execution can be regarded as a special instance of applying SBST to the program trace.

```
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 4.6: 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.

Heuristic search can be slow in discovering the specific solutions of narrow branch conditions. For example, the method `narrow` in Figure 4.6 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.

4.5 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 [114]. Another limitation is lack of support for arrays and static fields. Section 6.1 discusses the challenges of supporting objects in symcretic execution.

Cilocnoc relies on WALA [36] to read class files, build the program call graph, and construct control-flow graphs. The symbolic backward execution engine of Cilocnoc uses Z3 [25] to solve primitive constraints, and a custom solver for object-shape constraints. The heuristic phase finds inputs with the *concolic walk* algorithm, see chapter 5.

4.6 Evaluation

In this section, we empirically compare our implementation of symcretic execution (Cilocnoc) against two other input generators: Symbolic PathFinder[2] [87] and jCUTE[3] [89]. To measure the effectiveness and efficiency in generating target-specific inputs, we define target statements for a set of small programs (Table 4.1) and count how many search steps each tool takes before either finding inputs that reach the target or giving up (Table 4.2).

Experiment Setup

Table 4.1 lists the programs used in our evaluation. Each program represents a specific challenge for symbolic and concolic execution (see section 4.4). The *dart*, *easy-loop*, *trityp*, *sine*, *tcas*, and *tsafe* programs are examples that appear in related work: *dart* is close to the standard example for concolic execution [50, 86] (see Figure 4.3); *easy-loop* is a simple data-dependent loop that was used to evaluate JAUT [21]; and *trityp* is the classic highly-branching program for classifying triangles. The *sine*, *tcas*, and *tsafe* programs are part of the Symbolic PathFinder distribution and represent programs with trigonometric or bit-vector computations. The remaining programs consist of the methods shown in the Figures 3.1, 4.4, 4.5, and 4.6. In each program, we place target statements inside of branches that we wish to cover.

[2] `http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc`

[3] `http://os.cs.illinois.edu/software/jcute`

Table 4.1: 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 *Ifs* and *Loops* columns show the number of if-statements and and loops in the program, the *Targets* column contains the number of targets.

Program	*Description*	*LoC*	*Ifs*	*Loops*	*Targets*
dart	Figure 4.3	10	2	·	1
easy-loop	Decrementing loop	15	1	1	1
hard-loop	Figure 3.1	21	2	1	1
narrow	Figure 4.6	14	2	·	1
sine	Sine implementation	169	19	·	2
slicing	Figure 4.5	18	10	·	1
tcas	Aircraft collision avoidance	126	6	·	3
trityp	Triangle classification	49	10	·	3
tsafe	Loss of separation detection	82	11	·	5
unreach	Figure 4.4	20	11	·	1

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 (see chapter 5). 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 4.4, the data-dependency guidance proposed in prior work [35] would have little impact on the programs in our corpus. All tools explore the program depth-first without depth bound but with a 1 minute 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 (Table 4.3). 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 also takes about 1 seconds per program on an Intel Core i7 notebook with 2 GB of RAM.

All three tools use random values to solve path conditions. Whether a tool can cover a target or not is therefore subject to randomness. In addition, Cilocnoc's program exploration order, and therefore the number of explored path segments, depends on the arbitrary order in which it arranges the target program's basic blocks in memory. To counter these effects, we repeat all measurements seven times and verify the statistical significance of observed differences in the segment count and tool run time distributions through non-parametric Mann–Whitney U-tests.[4] To account for varying difficulty, we perform one test per program between the seven measurements (averaged over the program's targets) of the two compared tools. The tests are two-tailed and the significance level is $\alpha = 0.01$.

Results: Is Symcretic Execution Effective?

The data in Table 4.2 shows that Cilocnoc finds inputs for more reachable targets than the other tools, which suggests that symcretic execution is effective in finding target-specific inputs.

Both Cilocnoc and SPF-CW cover a similar number of targets, whereas jCUTE covers about half that number. The difference can be explained by jCUTE's eager simplification of path conditions: jCUTE replaces problematic constraints in the path condition with concrete values when they leave the theory of the underlying solver. In contrast, both Cilocnoc and SPF-CW defer using concrete values until they query the path condition's satisfiability. They can therefore incorporate side-conditions that were collected after the

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

Table 4.2: Average number of path segments explored before covering a target, and average total number of path segments explored per program. All segment counts are the arithmetic mean over seven runs. The significance of the differences is verified through a series of Mann–Whitney U-tests ($\alpha = 0.01$). The *Total segments explored* columns displays the number of segments explored before either covering all targets, or before reaching the 1 minute time limit, whichever occurred first. A dot ($\cdot$) denotes a zero.

		Targets covered (∅)			*Segments to cover target (∅)*			*Total segments explored (∅)*		
Program	*Tgts.*	jCUTE	SPF-CW	Cilocnoc	jCUTE	SPF-CW	Cilocnoc	jCUTE	SPF-CW	Cilocnoc
dart	1	·	1	1	·	1	1	3	1	1
easy-loop	1	0.43	1	1	902	1 318	13	5 018 183	1 318	13
hard-loop	1	·	·	1	·	·	25	968 308	19 748	25
narrow	1	·	·	1	·	·	3	8	5	3
sine	2	·	1	·	·	19	·	20	55	4
tcas	3	3	3	3	228	53	235	5 417	160	705
trityp	3	1	3	3	8	10	8	55	29	25
tsafe	3	0.43	2	2	9	47	2	58	233	33
slicing	·	·	·	·	·	·	·	2 570	1 533	1 023
unreach	·	·	·	·	·	·	·	2 862	1 533	1
$\sum$	15	4.86	11	12	·	·	·	5 007 484	24 614	1 833

Table 4.3: Adjusted run time for each tool per program. All values are in seconds. Adjusted means that 1 second was subtracted from the jCUTE and Cilocnoc times to account for pre-processing time that could be eliminated by caching. The mean and standard deviations are computed over seven runs. A hyphen (–) denotes a timeout after 1 minute.

	jCUTE		SPF-CW		Cilocnoc	
Program	*Mean*	*Std. dev.*	*Mean*	*Std. dev.*	*Mean*	*Std. dev.*
dart	1.34	0.09	0.91	0.23	0.87	0.08
easy-loop	–	–	–	–	0.89	0.10
hard-loop	–	–	–	–	2.38	0.27
narrow	1.82	0.92	0.76	0.06	0.90	0.13
sine	2.66	0.14	1.45	0.04	3.59	0.53
tcas	44.99	0.38	–	–	2.08	0.17
trityp	3.20	0.26	–	–	0.93	0.05
tsafe	44.99	0.38	3.53	0.15	2.08	0.17
slicing	–	–	1.73	0.06	1.45	0.07
unreach	–	–	1.80	0.06	0.97	0.06

problematic constraint was initially encountered (see the detailed discussion of this effect in chapter 5).

Benefiting from a strong symbolic solver, Cilocnoc uses concrete execution for just four programs: the *easy-loop*, the *hard-loop*, the *sine*, and the *tsafe* program. For both loop programs, the concrete execution accelerates the exploration of the program's data dependent loop and help it cover the targets within the 1 minute time limit. On the *sine* and *tsafe* programs, Cilocnoc and SPF-CW employ concrete execution and the concolic walk algorithm to solve an external method call, as well as trigonometric constraints. Notably, the concolic walk algorithm of Cilocnoc fails to solve the respective constraint in the *sine* program whereas the same algorithm succeeds for SPF-CW. The *narrow* program can be covered by Cilocnoc because its symbolic solver knows about bit-vector arithmetic and the irregularity of negating the smallest integer. In contrast, SPF-CW uses a weaker symbolic solver and therefore cannot cover the target.

Results: How Efficient is Symcretic Execution?

The results in Table 4.2 support our hypothesis that symcretic execution is more efficient than concolic and symbolic execution.

For both loop programs, Cilocnoc side-steps the costly symbolic exploration of the loop. As a consequence, it explores 13 instead of 5 018 183 (jCUTE) and 1 318 (SPF-CW) path segments in the *easy-loop* program, and 25 instead of 968 308 (jCUTE) and 19 748 (SPF-CW) segments in the *hard-loop* program. On the *unreach* program, Cilocnoc benefits from exploring the program backwards and being able to recognize unreachable branches as discussed in section 4.4: instead of exceeding the 1 minute time limit, exploring 2 862 (jCUTE) or 1 533 segments (SPF-CW), it explores a single segment.

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 an order of magnitude slower than Cilocnoc.

4.7 Summary

Inputs that cover a specific target are useful in debugging and testing. Unfortunately, concolic testing and search-based software testing are inefficient in generating such target-specific inputs because (1) the driving concrete execution is hard to guide towards the target; and (2) they cannot detect if a path is unreachable, which prevents them from stopping the program exploration. Symcretic execution is an alternative target-specific input generation technique that avoids these problems. Evaluating symcretic execution on a range of test cases shows that it finds inputs in more cases than concolic testing tools while exploring fewer paths.

Chapter 5

Heuristic Solving of Complex Path Conditions[⋆]

This chapter describes the *concolic walk* (CW) algorithm for solving complex arithmetic constraints and path conditions. While useful for the concrete phase of symcretic execution, the algorithm likewise applies to symbolic and concolic testing, where it increases the coverage. This chapter therefore discusses the algorithm in this better-known context.

A central problem in symbolic and concolic testing is translating the arithmetic constraints of the path condition into the theory of the underlying solver. The difficulties in translating are:

1. Non-linear integer constraints often make it infeasible to solve the path condition. Such constraints are even undecidable in general [24].
2. Path conditions can contain calls to (uninterpreted) library methods, such as trigonometric functions, about which the solver cannot reason [114].

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

[⋆]This chapter is based on the paper *Solving Complex Path Conditions through Heuristic Search on Induced Polytopes* by Peter Dinges and Gul Agha, which appeared in the proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT/FSE'14) [31]. Permission to reprint has been granted by ACM.
`http://dx.doi.org/10.1145/2635868.2635889`

- The solutions of the linear constraints in the path condition define a contiguous convex region in the space (a polytope).[1] All solutions to the whole path condition must lie within this polytope.

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

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

In particular, our algorithm (1) splits the path condition into linear and non-linear constraints; (2) finds a point in the polytope induced by the linear constraints with an off-the-shelf solver; and then, (3) starting from this point, uses *adaptive search* [23] within the polytope, guided by the constraint fitness functions, to find a solution to the whole path condition.

5.1 Motivation

Assume we want to find an input for the `example1` method in Figure 5.1 that covers the path along the statements in the lines 3, 4, and 5. To drive the execution down this path, `x` and `y` must satisfy the non-linear path condition $x = x \cdot y \land x > 2$. Given a suitable decision procedure, we can solve the path condition to obtain, for instance, the concrete inputs `x=3` and `y=1`.

Mitigating Solver Limitations

Unfortunately, a complete symbolic decision procedure for general non-linear integer constraints cannot exist [24]. Furthermore, a decision procedure can typically only find solutions if it has an interpretation for all appearing

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

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

Figure 5.1: Example Java methods with complex path conditions. In method example1, the path to the error (line 5) has the non-linear path condition $x = x \cdot y \wedge x > 2$. Neither jCUTE [89] nor mixed concrete–symbolic solving [86] discover input values that satisfy this path condition. In method example2, the path condition for the error (line 13) contains a call to an uninterpreted library method. Neither SPF-CORAL [94] nor Pex [103] (for the C# version) discover input values that satisfy the path condition.

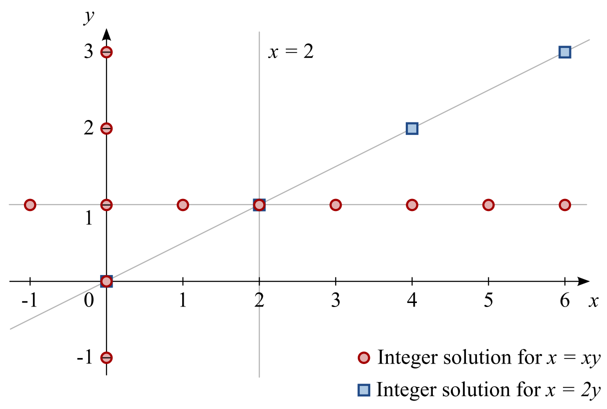

Figure 5.2: Solutions of the non-linear equation $x = x \cdot y$ (circles) and its linearization $x = 2 \cdot y$ (squares). No solution for $x = 2 \cdot y$, satisfies the path condition $x = x \cdot y \wedge x > 2$, which shows the danger of blindly simplifying path conditions.

operations. If a path condition contains arbitrary integer arithmetic or, importantly, calls to opaque library methods—like the native *cos* method in Java—, then finding a solution is hard [114].

A common approach to work around such solver limitations is to simplify (under-approximate) the path condition: parts of the path condition that the solver cannot handle are first executed on concrete inputs and then replaced with the concrete results. Simplification has been applied while constructing the path condition, and while solving it. Classic concolic test generators such as jCUTE and Pex simplify at construction time. For example, jCUTE relies on a linear constraint solver. When building the path condition for the path 2, 3, 4, 5 in Figure 5.1, it replaces the non-linear expression $x \cdot y$ with $2 \cdot y$ if $x = 2$ when the expression is added. This yields the path condition $x = 2 \cdot y \wedge x > 2 \wedge x = 2$ if we include the constraint $x = 2$ that is required to make the simplification sound [49, 86]. In contrast, *mixed concrete–symbolic solving* [86] simplifies at solution time. To solve a path condition, mixed solving splits it into resolvable and *complex* constraints, solves the resolvable ones directly, and uses the solution to simplify and concretely execute the complex constraints. The execution results, in turn, serve to simplify the complex constraints.

Figure 5.2 shows how simplification produces bad approximations. The simplified path condition of above example is unsatisfiable because of a bad approximation due to a random choice of x. Likewise, mixed solving fails to cover the path because the only feedback from the concrete execution to the constraint solver is ruling out non-working values, which is often insufficient to find a solution (section 5.7). Observe that both simplification techniques are problematic because of *blind commitment* to concrete values, regardless of other constraints on the variables.

Stronger Solvers

The number of such bad approximations decreases as the strength of the solver increases. The Pex concolic test generator [103], for example, relies on the Z3 SMT[2] solver [25] to find inputs that satisfy a path condition. Z3 supports (some) non-linear integer operations in its constraints, and hence Pex discovers the error in line 5 of Figure 5.1. Likewise, SPF-CORAL, a combination of the *Symbolic PathFinder* (SPF) symbolic execution tool [85, 87] and a constraint solver based on heuristic search (CORAL [94, 13]), discovers the error.

While this extends the domain of programs that can be handled and enables coverage of more paths than before, it does not fully fix the interpretation problem. Specifically, programs may call hitherto unknown library methods. For example, an implementation of trigonometric functions converts floating-point numbers to bit-vectors, as exemplified in Figure 5.1. Such methods would require a solver extension, arguably a maintenance nightmare. Other interactions, such as database queries, are even harder to integrate.

5.2 Algorithm

This section describes the concolic walk (CW) algorithm for solving path conditions. The algorithm addresses the aforementioned challenges of relying on decision procedures by treating linear and non-linear constraints differently: to solve the linear constraints, it uses an off-the-shelf solver; to solve the

[2]Satisfiability Modulo Theories

non-linear constraints, it uses heuristic search based on concrete execution (evaluation). Using concrete execution allows the algorithm to handle opaque library methods without further extension. At the same time, the algorithm never simplifies the path condition to avoid blind commitment to concrete values.

Synopsis

The algorithm distinguishes between linear and non-linear constraints because linear constraint systems are decidable and efficient solvers exist. Furthermore, linear constraints have a useful geometric interpretation: Assume that we relax the domain of each variable in the path condition to $\mathbb{R}$. Then we can regard an assignment of values to each variable as point in a *valuation space*, which corresponds to $\mathbb{R}^n$. In the valuation space, the solutions of a linear constraint form a half-space; a conjunction of linear constraints therefore describes an intersection of half-spaces, which is a convex (hence, contiguous) region—a convex polytope.[3] In Figure 5.2, the polytope is the region right of the line $x = 2$; in Figure 5.3, it is the shaded region right of the line $x = 2$ and above the line $x + y = 2$.

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

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

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

From the many meta-heuristics (simulated annealing, genetic algorithms, etc.), we chose the *adaptive search* variant [23] of tabu-search [46, 47] for its ease of adding a non-random neighbor-picking strategy (see below). In each iteration, adaptive search picks the variable that appears in the most violated constraints and examines neighbor points that differ only in this variable. A variable becomes *tabu* for several iterations if changing its value failed to yield a better neighbor. The search moves towards the fittest neighbor, like hill-climbing, but escapes local minima with the help of the tabu mechanism.

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

Example

Consider the partial plot of the valuation space for the variables x and y in Figure 5.3. We are interested in solving the path condition

$$x = x \cdot y \land x \geq 2 \land x + y \geq 2,$$

which consists of the linear constraints $x \geq 2$ and $x + y \geq 2$, and the non-linear constraint $x = x \cdot y$. The linear constraints describe an unbounded convex polytope; the figure shows (a part of) the polytope as shaded area. All feasible solutions of the path condition must be contained within this polytope. To discover a solution of the non-linear constraint (circles in the figure), we ask the off-the-shelf linear constraint solver for an arbitrary point in the polytope and start a random walk. Suppose our starting point is $(x, y) = (4, -1)$. Choosing the variable x for modification, we randomly generate the neighbors $(3, -1)$ and $(6, -1)$. From these, we pick $(3, -1)$ because it is closer to satisfying the non-linear constraint $x = x \cdot y$ (step 1 in the figure). Modifying x again does not yield a better neighbor; all of these lie outside the polytope (step 2). Thus, x is marked as tabu and the next

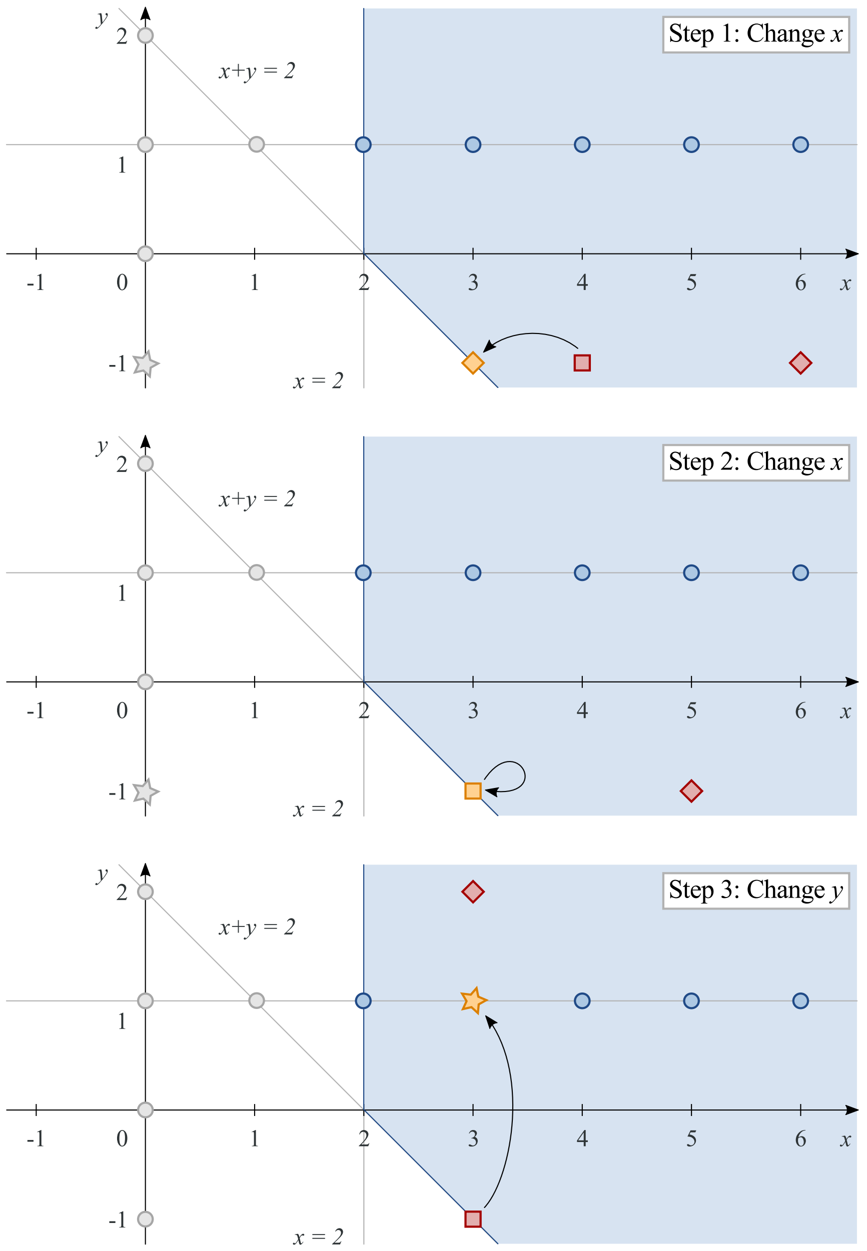

Figure 5.3: Example step sequence when solving the path condition $x = x \cdot y \wedge x \geq 2 \wedge x + y \geq 2$ with the concolic walk algorithm. In the figure, circles mark solutions to the non-linear constraint $x = x \cdot y$. The shaded area highlights the unbounded polytope described by the linear constraints $x \geq 2$ and $x + y \geq 2$. Solutions to the path conditions are circles that are contained in the polytope. The figure denotes the current point as a square; generated candidates as diamonds; and estimated solutions from bisection as stars.

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

Figure 5.4: Syntax of the example language, with variables Var, literals Lit, function symbols Fun, binary operations $\circ \in \{+, -, \cdot, /, \text{mod}\}$, and relations $\sim \in \{<, \leq, \geq, >, =, \neq\}$.

iteration modifies y. Aside from a random neighbor $(3, 2)$, we estimate a zero for the linear parameterization of $x - x \cdot y$ on the line $(3, -1)$–$(3, 2)$, which yields the solution $(3, 1)$ (step 3).

5.3 Terms and Definitions

Before formalizing the CW algorithm in the next section, we briefly define the terms used. To simplify the exposition, we describe the algorithm for a small imperative programming language whose Java-like syntax is shown in Figure 5.4. The algorithm itself is independent of the target language and applies to any language for which the CEVAL and SEVAL functions can be defined accordingly.

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

Concrete Execution

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

$$\textsc{ceval}(\mathit{prog}, \chi) = (\mathit{prog}', \chi'),$$

where prog' is the program derived from *prog* by executing one step, and χ' is derived from χ by applying the effects of this step. Using record notation $\langle z : c\rangle$ for the environment χ with $\chi[z] = c$ and $\chi[y] = \bot$ for $y \neq z$, we thus have $\textsc{ceval}\big(\mathsf{y{=}2(x{-}2)}, \langle x : 23\rangle\big) = \big(\bot, \langle y : 42, x : 23\rangle\big)$. As shorthand for expression evaluation, we write $\textsc{ceval}(e, \chi)$ for $\chi'[z]$ where $(\bot, \chi') = \textsc{ceval}(z = e, \chi)$ with a fresh variable z.

When a program runs, it follows an *execution path*, which is a sequence of steps $i \hookrightarrow j$ from statement number i in the program to one of its successors j.

Symbolic Execution

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

$$\textsc{seval}(\mathit{prog}, i \hookrightarrow j, \sigma, P) = (\sigma', P').$$

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

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

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

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

Constraints

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

As motivated in section 5.2, we distinguish linear from non-linear constraints to exploit their decidability and geometric interpretation. A constraint $\ell \sim r$ is linear if it can be transformed into a normal form

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

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

Algorithm 5.1: The *concolic walk* algorithm for solving the path condition P. Notation: $P = \{\ell_i \sim_i r_i \mid i\}$ is a set of constraints, $x, y \in \text{Var}$ are variables, and $\alpha, \beta, \gamma, \varepsilon, \mu, \tau : \text{Var} \to \mathbb{R}$ are environments whose elements are 0 by default. Operations on environments apply point-wise. The VARS function returns the set of variables appearing in an expression, constraint, or path condition; see Figure 5.5.

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

$$\begin{aligned}
\textsc{vars}(e) &= \begin{cases} \{x\} & \text{if } e = x; \\ \{a_1, \ldots, a_n\} & \text{if } e = f(a_1, \ldots, a_n); \\ \textsc{vars}(e_1) \cup \textsc{vars}(e_2) & \text{if } e = e_1 \circ e_2; \\ \textsc{vars}(e) & \text{if } e = (e); \\ \emptyset & \text{otherwise.} \end{cases} \\
\textsc{vars}(\ell \sim r) &= \textsc{vars}(\ell) \cup \textsc{vars}(r) \\
\textsc{vars}(P) &= \bigcup_i \textsc{vars}(\ell_i \sim_i r_i) \qquad \text{for } P = \{\ell_i \sim_i r_i \mid i\}
\end{aligned}$$

Figure 5.5: Definition of the VARS function that returns the set of variables appearing in an expression, constraint, or path condition.

5.4 Concolic Walk Algorithm

Algorithm 5.1 formalizes the CW algorithm. The algorithm accepts a path condition P as input and returns a concrete environment that satisfies P, or $\perp$ if it could not find such environment.

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

In each iteration, the algorithm tries to find an environment with a lower *error score* than the current environment α. To do so, it (1) finds the variable x with the highest error score (lines 14–20); (2) generates neighbor environments for α by modifying this variable's value; (3) picks the neighbor μ with the lowest error score (both line 21); (4) and makes μ the current environment if is better than α (lines 23–26). Algorithm 5.2 shows the function for computing the error scores; the function resembles Korel's *branch functions* [68]. The auxiliary VARS function is defined in Figure 5.5.

Algorithm 5.2: Error score for the constraint $\ell \sim r$ in the environment χ that captures "how badly" the constraint is violated. The procedure computes the score by executing the symbolic expressions ℓ, r on the concrete inputs in χ.

```
procedure ComputeError($\ell \sim r, \chi$)
    $d \leftarrow$ CEVAL($\ell, \chi$) $-$ CEVAL($r, \chi$)
    if $\sim$ is $=$ then
        return $|d|$
    else if $\sim$ is $\neq$ then
        if $d \neq 0$ then return $0$ else return $1$
    else
        if $d \sim 0$ then return $0$ else return $|d| + 1$
    end if
end procedure

procedure ComputeError($\{\ell_i \sim_i r_i \mid i\}, \chi$)
    return $\sum_i$ ComputeError($\ell_i \sim_i r_i, \chi$)
end procedure
```

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

Neighbor Selection

Generating neighbors and picking the best one has been extracted to the FindBestNeighbor function (Algorithm 5.3). R times, the function generates two neighbor environments β and γ for its input α, remembering the overall best one.

Algorithm 5.3: Choosing the best among environments that differ from α in their x-entry and lie inside the polytope L (satisfy L).

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

The environment β is the result of taking a random step in the polytope L along the axis of the given variable x; see Algorithm 5.4. The function RandomStep guarantees that the returned environment lies within the polytope.

The environment γ is the result of linear approximation: for a random unsatisfied constraint $\ell \sim r \in N$ that contains x, the BisectionStep function estimates the environment that would satisfy the constraint if both ℓ and r were linear functions. This can be seen as performing one step of the bisection method in the hope of jumping to a region where a solution is close. Setting $v_\alpha = \textsc{ceval}(\ell - r, \alpha)$ and $v_\beta = \textsc{ceval}(\ell - r, \beta)$, the slope of the linear approximation is $t = -v_\alpha/(v_\beta - v_\alpha)$, and γ is the zero of $t \cdot (\beta - \alpha) + \alpha$. If $v_\alpha = v_\beta$, a random slope is used.

Algorithm 5.4: Taking a random step from χ in the y_i-directions within the polytope L. The algorithm uses rejection sampling with at most M samples. Parameter S affects the step radius. The function AdjustToSatisfy restores linear equalities that were violated during randomization by recomputing affected values from ν.

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

5.5 Discussion

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

Checking the satisfaction of constraints, computing their error scores, and generating linear approximations (in function BisectionStep) requires the concrete values of the involved expressions. The CW algorithm does not mandate a specific way of obtaining these values: they could be computed by an interpreter for the symbolic expressions, or they could be extracted from an instrumented version of the original program. Therefore, the algorithm can be used in both symbolic and concolic execution.

5.6 Implementation

We have implemented the algorithm described in section 5.4 as an extension of Symbolic PathFinder (SPF) [85, 87]. SPF is a symbolic execution engine built on top of the JPF verification framework.[4] Our extension works with existing SPF test-drivers; the only change required is enabling the extension by setting a configuration flag. No further annotations or code changes are necessary. The implementation, as well as the evaluation harness are available for download on our website.[5]

The extension acts as a *PCAnalyzer* in SPF. It intercepts path condition satisfiability queries and answers them with the CW algorithm. Instead of re-executing the program to obtain the concrete value of symbolic expressions in the path condition, the extension evaluates the expressions with an interpreter. While limiting the implementation's performance, the design allows evaluating the constraints in the path condition independently of each other. Using the program, control-dependencies would enforce solving the constraints in the order of their appearance [68].

Our implementation leverages the support for uninterpreted function symbols in path conditions that was added to SPF as part of mixed concrete–symbolic solving [86]. The function symbols wrap library functions, which can be called concretely via reflection when solving the path condition. Since the circumstances of the functions' invocations depend on SPF's backtracking, wrapping functions with side-effects can lead to solutions that fail in a different context such as a generated unit test.

Unlike their definitions, the implementation of the error score functions must handle underflows and overflows of integers, as well as not-a-number floating point anomalies. Underflows are assigned an error score of the smallest integer value; overflows the largest integer value; and not-a-number has an error score of 1/8 of the maximum **double** value.

[4] `http://babelfish.arc.nasa.gov/trac/jpf/`

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

5.7 Evaluation

This section evaluates how effective and efficient the *concolic walk* (CW) algorithm is in solving path conditions with non-linear arithmetic constraints. To make the results comparable and link them to a practical application of the algorithm, we measure effectiveness as the coverage of generated test cases on a focused program corpus. The corpus consists of programs whose path conditions contain mostly non-linear constraints. In subsection 5.7.1, we evaluate the performance of the CW algorithm. In subsection 5.7.2, we investigate how the parameters of the algorithm influence the coverage.

Program Corpus

Table 5.1 lists the programs used in the evaluation together with the type of non-linear operations appearing in their path conditions. Due to our focus on non-linear arithmetic constraints, the programs are samples from a population of programs that use such constraints; they do not represent *common* programs. To avoid a bias towards specific strengths of our approach and to foster comparability, we use mostly examples from works presenting other approaches to concrete–symbolic and randomized solving of path conditions: The *coral* program[6] is a collection of 65 benchmark functions used to evaluate the CORAL constraint solver [94]. The functions consist of a single if-statement whose condition includes a mixture of complex mathematical operations like calls to trigonometric functions. *opti* contains the six non-linear benchmark functions that were part of evaluating the FloPSy floating-point constraint solver [70]. The *dart*, *power*, *sine*, *stat*, and *tsafe* programs are part of the Symbolic PathFinder distribution. *stat* computes the mean and standard deviation of a list of numbers; and *tsafe* is an aviation safety program that predicts and resolves the loss of separation between airplanes. *blind* is an implementation of Figure 5.1, and *hash* tries to provoke five collision variants in a common hash function [11]. *tcas* features involved, but linear, control-flow.

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

Table 5.1: Programs used to evaluate the CW algorithm. The *LoC* column lists the number of source code lines in the program, excluding comments and empty lines. The *Operations* column describes the type of operations appearing in the path condition.

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

It demonstrates how the evaluated algorithm behaves on *classical* testing problems. Finally, the *ray* application[7] is a simple ray tracing renderer.

Method

We generate test cases for each program in the corpus and record the coverage of the generated tests with JaCoCo.[8] To account for randomness, we repeat this process seven times and verify the statistical significance of observed differences in the coverage and generation time distributions through non-parametric Mann–Whitney U-tests.[9] To account for varying difficulty, we perform one test per program between the seven measurements of the two compared algorithms. Unless stated otherwise, the test is two-tailed and the significance level is $\alpha = 0.01$. To prevent small programs from dominating

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

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

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

the mean coverage of the algorithms, we weight each program's contribution by the program's lines of code when computing the arithmetic mean.

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

To avoid adverse effects on a tool's performance from handling object-creation [114], each test invokes a single driver method with just numerical parameters. Thus, the challenge of creating objects as test inputs is absent in our setup.

5.7.1 Effectiveness and Efficiency

This section discusses the performance of the CW algorithm, answering the following research questions:

RQ1: Is the CW algorithm more effective in generating test inputs for programs with non-linear arithmetic path conditions than simplification-based approaches?

RQ2: On such programs, can it improve the effectiveness of test generators that employ strong constraint solvers?

RQ3: How efficient is the CW implementation compared to similar test generators?

Table 5.2: Coverage and generation (wall-clock) time of the test cases generated for each program. The *Med.* columns show the median of seven runs; the *Var.* columns show the respective variance. For benchmarks, the coverage is the percentage of covered target statements; for other programs, it is the branch coverage. Dots denote zeros. The *LoC-Weighted Avg.* row lists the arithmetic mean over all programs, using the LoC as weights to prevent small programs from dominating the numbers.

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

RQ1: Is the CW algorithm more effective than simplification for solving non-linear arithmetic path conditions?

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

- SPF-Mixed [85, 87] is a variant of Symbolic PathFinder that attempts to solve a non-linear arithmetic path condition by solving the decidable (simple) part, using the solution to concretely execute the complex part, and then further simplifying all constraints using the results [86]. As suggested in the paper, we enable the randomization heuristic of

SPF-Mixed. In addition, we increase the maximum number of solving tries to three per path condition instead of the default of one.

- jCUTE [89] is a classic concolic testing tool. During the construction of a path condition, it substitutes parts of non-linear terms with their concrete run-time value to ensure that all constraints remain linear. In our setup, jCUTE randomizes the initial concrete values and explores paths in random order.

Both simplification-based tools achieve a much lower coverage: the weighted average of the median coverages is 41% for jCUTE and 26% for SPF-Mixed; the respective standard deviations are 1.8% and 0.5%. This is far from the 78% coverage achieved by our algorithm (s.d. 1.2%), see Table 5.2. On each program, our algorithm achieves a higher coverage than jCUTE. Except for *stat* and *tcas*, the same is true for SPF-Mixed. All differences are significant. Furthermore, the inputs generated by our algorithm subsume the inputs of the other tools except for a small fraction of *ray* and *stat* inputs, which shows that the differences are true improvements. The draw between our algorithm and SPF-Mixed on *tcas*, which lacks non-linear operations, indicates that the differences originate in the management of non-linear constraints.

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

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

As discussed in section 5.1, strong constraint solvers allow test generators to solve more path conditions directly, thereby reducing the need to resort to approximations like simplification. For such tools, the CW algorithm can serve as a fallback strategy whenever the solver cannot handle the path condition. However, such cases might be rare with state-of-the-art solvers. We therefore investigate the null-hypothesis that combining state-of-the-art concolic test generators with the CW algorithm does *not* improve the achieved coverage. To test the hypothesis, we consider combinations of our algorithm with two tools:

- SPF-CORAL, a symbolic execution tool that relies on the CORAL solver [94, 13]. CORAL targets non-linear arithmetic constraints and uses the Particle Swarm Optimization [64] search heuristic to find solutions. We use SPF-CORAL in the default configuration set in the SPF repository.

- Pex[10] [103], a concolic test generator that employs fitness-guided exploration [115] and a strong SMT solver (Z3 [25]). Because our experiment focuses on coverage rather than user responsiveness, we increase Pex's solver timeout to five seconds and allow it to run up to 500 iterations without generating new tests.

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

The combination with the CW algorithm raises the averaged median coverage of SPF-CORAL from 62% to 82% (s.d. 0.2% and 0.9%), and that of Pex from 68% to 82% (s.d. 0.5% and 1.5%). Both tools achieve higher coverage on the *coral*, *opti*, *ray*, and *sine* programs. SPF-CORAL furthermore improves on *hash*, *dart*, and *tcas*, while Pex improves on *tsafe*. In the case of *sine*, this is not surprising: the problematic example2 method in Figure 5.1 is a snippet of this program. In the case of *coral*, the tools and our algorithm complement each other: the union of test inputs achieves higher coverage than each set of inputs alone.

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

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

RQ3: How efficient is the CW algorithm?

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

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

5.7.2 Influence of Algorithm Parameters

This section discusses how the parameters of the CW algorithm influence its effectiveness. In our experiments, we vary a single parameter at a time and compare the variation's coverage with the baseline coverage shown in Table 5.2.

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

Without the marking of variables as tabu, the overall coverage drops to 62% (s.d. 0.9%), which is 0.79 times (Table 5.3) the 78% baseline coverage. Thus, the tabu mechanism contributes a relative performance increase of 26% to

Table 5.3: Fraction of the baseline coverage (Table 5.2) that the algorithm variations achieve. Values below 1.0 mean less coverage than the baseline algorithm; values above 1.0 mean more coverage. Dots (·) denote the value 1.0 and indicate no change. Each variation changes a single parameter of the baseline algorithm, which uses the following settings (see Algorithm 5.1): $I = 150$ steps per constraint, $R = 10$ neighbors, $T = \min(3, |\textsc{vars}(N)|/2)$ tabu iterations, and bisection enabled. Each fraction is that of the median of seven runs.

			Benchmarks				*Other*						
Algorithm Variation		*W.Avg.*	*coral*	*blind*	*hash*	*opti*	*dart*	*power*	*ray*	*sine*	*stat*	*tcas*	*tsafe*
Tabu disabled	$(T = 0)$	0.79	0.57	·	0.25	·	·	0.64	0.79	0.93	·	·	0.92
Tabu extended	$(T \geq 5)$	·	·	·	·	·	·	·	·	·	·	·	·
Bisection disabled		0.92	0.86	·	0.5	0.33	·	·	·	0.89	·	·	·
10 steps	$(I = 10)$	0.95	0.82	·	·	·	0.75	·	1.04	0.89	·	·	·
30 steps	$(I = 30)$	0.98	0.94	·	·	·	0.92	·	·	0.96	·	·	·
75 steps	$(I = 75)$	1.01	1.02	·	·	·	·	·	·	1.04	·	·	·
300 steps	$(I = 300)$	·	1.04	·	·	·	·	·	·	0.96	·	·	·
3 neighbors	$(R = 3)$	·	1.02	·	·	·	·	·	·	·	·	·	·
25 neighbors	$(R = 25)$	1.01	·	·	·	·	·	·	·	1.04	·	·	·
50 neighbors	$(R = 50)$	·	·	·	·	·	·	·	·	·	·	·	·
100 neighbors	$(R = 100)$	·	1.04	·	·	·	·	·	0.98	·	·	·	·

the algorithm. This performance change appears consistently over all runs of the algorithm; the difference is significant for all programs except *sine*, whose hard floating-point to bit-vector conversion emphasizes the random walk aspect of our algorithm. Once tabu marking is enabled, however, the chosen number of tabu iterations seems to have little influence on the algorithm's performance: increasing it from the default $T = \min(3, |\textsc{vars}(N)|/2)$ to $T = \min(5, |\textsc{vars}(N)|)$, for example, lacks any effect on the coverage.

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

Disabling the estimation of solutions through linear approximation—the bisection step explained in section 5.2—reduces the overall coverage to 0.92 times the baseline. Thus, the bisection step improves the overall coverage from 72% (s.d. 1.4%) to 78%, a relative increase of about 8%. Without bisection, the algorithm consistently achieves lower coverage on the *coral*, *hash*, and *opti* benchmarks (significant for $\alpha = 0.01$). It therefore seems that the bisection step steers the random walk towards promising areas, resulting in more found solutions.

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

For the majority of programs in the corpus, granting the algorithm more steps to find a solution or choosing among more neighbors has little effect on the coverage. The average coverage for just 10 steps per constraint is 0.95 times the baseline coverage. Likewise, allowing more than the 150 baseline steps per constraint leads to the same overall coverage. It appears that for most programs, the constraints are simple enough that few steps suffice to find a solution. However, for the complex constraints of *coral*, more steps increase the coverage, but only for 10 steps per constraint is the difference to the baseline significant ($\alpha = 0.05$). The price for the 27% relative coverage improvement from 10 steps to 300 steps per constraint is a relative slowdown of 23%: the test generation time grows from 1.3 minutes (s.d. 0.3s) for 10 steps to 1.6 minutes (s.d. 3.2s) for 300 steps per constraint.

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

5.8 Limitations and Future Work

Evaluation—Threats to Validity

We try to ensure *conclusion validity* of our evaluation by checking the statistical significance of measured differences with a robust non-parametric test at a high level $\alpha = 0.01$. One threat to the *construct validity* of our experiments is the use of coverage as effectiveness metric. While branch coverage is a good predictor for the bug-detection capability of test suites [45], it does not measure how useful the generated inputs are for the user. Lacking a user study, we are unfortunately limited to this common surrogate metric. Another threat, in particular for RQ1, is the aggregate nature of coverage: two test suites with similar coverage may complement each other, making them incomparable. We mitigate this risk in by checking whether the test suite with higher coverage subsumes the one with lower coverage.

The *internal validity* of our experiments is threatened by our comparison of different tools. Many implementation details, not just constraint solving, influence the performance. We try to lessen this problem by (1) including a program in the corpus (*tcas*) that lacks non-linear operations and ensuring that the compared tools have similar effectiveness (RQ1); and (2) limiting the efficiency comparison to tools sharing the same SPF-based infrastructure (RQ3).

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

Algorithm

The CW algorithm currently cannot create objects as test inputs. While it could employ, among others, feedback-directed randomization [81] or heuristic search [62, 40] to fill this gap, we plan to investigate in future work how exactly object randomization relates to the notions of continuity and neighborhood used by its local search strategy.

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

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

Implementation

The CW implementation described in section 5.6 assumes that the native methods occurring in constraints are pure, that is, lack side-effects. A more general implementation could purge this assumptions by integrating a notion of setup and tear-down methods—as in unit tests—that are executed before and after each constraint evaluation, and by maintaining the program's execution order for native methods. Furthermore, a better implementation could accelerate the algorithm (1) by executing constraint expressions directly in the JVM instead of interpreting them; (2) by caching the error scores of constraints whose inputs remained unchanged during a step; and (3) by using memoized solutions [110, 116] to seed the starting point.

5.9 Summary

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

Chapter 6

Research Directions

The previous chapters presented all algorithms in the context of small sequential programming languages that operate on primitive values. In this chapter, we discuss the challenges of applying symcretic execution to richer languages that support objects and concurrency.

6.1 Support for Objects

Many programs operate not only on primitive data such as integers and floating point numbers, but also on structured data objects and references to such objects. Extending symcretic execution to these programs requires adjustments to both of its phases.

6.1.1 Symbolic Phase

Supporting objects in the first phase of symcretic execution requires a symbolic representation of objects and references to them. The representation must support read and write operations to the named *fields* that comprise an object, as well as constraints on the values of fields that may appear in branch conditions. The representation must furthermore support operations and constraints on the references themselves. An example operation on references is pointer arithmetic, example constraints are equality or sub-type constraints. Equality constraints between references must be transitive: asserting $r = r'$ for symbolic references r and r' must imply that $r.f = r'.f$ for all fields f.

Unfortunately, as of 2014, SMT solvers do not typically include a theory of symbolic objects and references. Support for these must be added manually.

Two choices for adding support are encoding references at the SMT solver level, or adding a second, dedicated solver that synchronizes its results with the SMT solver.

Supporting symbolic objects at the level of the SMT solver allows different encodings. A first option is to encode objects via arrays [19] if the (SMT) constraint solver supports the theory of arrays [77]. In this encoding, references are symbolic integers. A declared field defines an array whose entries are the values of the field for the object with that index. Thus, reading a field, for example $z = r.\mathsf{f}$, corresponds to an array read operation $\mathsf{select}(f, r)$, where f is the (symbolic) array and r is the index; writing a field, for example $r.\mathsf{f} = z$, corresponds to an array write operation $\mathsf{store}(f, r, z)$. While simple to add, this encoding does not support pointer arithmetic because it does not capture the underlying memory layout.

A second option for supporting objects at the level of the SMT solver is to encode them as boolean satisfiability problems, similar to relational logic solvers [63, 105]. The challenge with such an encoding is that techniques developed for relational logic assume a finite universe. For symcretic execution, this universe must be derived automatically from the code of the input program. Like the array encoding, this encoding lacks support for pointer arithmetic.

Support for objects can also be added at a more direct level by combining a dedicated solver for reference constraints with the SMT solver for arithmetic constraints. A simple implementation of such direct object management assigns each symbolic reference a dedicated object that stores the symbolic field values. This creates the opportunity to apply existing lazy object-initialization techniques that have been developed for symbolic execution [65, 28]. However the challenge becomes coordinating the two solvers because field values may be subject to primitive constraints, and equality between references must be transitive, including primitive field values. Thus, much like an SMT solver itself [26], the reference solver must communicate its assumptions to the arithmetic solver and vice versa. It may therefore be beneficial to instead implement direct reference handling as a dedicated theory in the SMT solver. Research on such theories and their efficient implementation will not only

```
 1 static Object makeObject() {
 2   return new Object();
 3 }
 4
 5 static void aliasError() {
 6   Object r1 = makeObject();
 7   Object r2 = makeObject();
 8   if (r1 != r2) {
 9     error();
10   }
11 }
```

Figure 6.1: Example program that shows how potential aliases inflate the search space.

benefit symcretic execution, but benefit techniques based on symbolic and concolic execution in general.

Aliases

Regardless of the object encoding, the symbolic backward execution must account for aliasing of symbolic references. Two references r and r' are *aliased* if they refer to the same object [3]. Aliasing matters because reading the field $r.f$ after first writing $r.f$ and then $r'.f$ yields the second written value if r and r' are aliased, and the first value if they are not. Hence, the backward execution must enforce the respective aliasing constraints on symbolic references as it explores the execution path. In contrast, concolic test generators largely avoid aliasing problems because the concrete execution determines the aliasing of all references except for those that are inputs.

The program in Figure 6.1 exemplifies how potential aliases inflate the search space during backward execution. The aliasError method contains an error in line 9 that occurs if the makeObject factory method were to return a different reference for both calls. Starting from the error and stepping backwards, the execution collects the constraint $r_1 \neq r_2$ for the symbolic references r_1 and r_2. Neither reference has appeared in the program thus far. Therefore, r_1 can either point to **null** or a fresh object o_1. Likewise, r_2

can point to **null** or a fresh object o_2. However, r_2 may also point to o_1. In total, line 8 allows for six reference–object assignments, only two of which the constraint $r_1 \neq r_2$ rules out. Further references in the method would contribute additional factors to the number of assignments because each one could alias an existing object, or refer to a fresh object. Depending on how object support was implemented, either the constraint solver or the backward execution itself must cope with this exponential growth of value assignments.

A heuristic strategy for mitigating the growth of value assignments is to bound the nesting level of the object graph [27]. In this approach, symbolic objects discovered by following more than k references can no longer point to a fresh object; they must always point to an existing object. The intuition is that the reference constraints in path conditions can often be satisfied through *small* object graphs. However, further research is necessary to determine conditions under which k-bounding is complete, and how often these hold in practice.

Translating points-to information [7, 95] into reference constraints can mitigate the growth of value assignments. For example, knowing that the calls to the makeObject method in lines 6 and 7 of Figure 6.1 return the same reference would exclude the satisfying reference–object assignments, thereby making the error unreachable. Unfortunately, the benefits of points-to information can be limited in important cases because the customary identification of static object allocation sites with may-alias classes [3] can be too imprecise if too little context is included. For example, assuming context-insensitive analysis, all (fresh) objects returned by a factory method belong to the same may-alias class of references. Such alias information would therefore have no effect on the symcretic execution of a program similar to that in Figure 6.1, in which makeObject returns new objects. Nevertheless, adopting and integrating a points-to analysis into symcretic execution, possibly as pre-processing step, could shrink the search space significantly. The development and study of such a points-to analysis is an important direction for future work.

Type Constraints

In typed object-oriented languages like Java, constraints can limit the type of object to which a reference may point. These *type constraints* originate from branches that use runtime type information, such as Java's **instanceof** expression, and from calls to virtual methods.

Virtual method calls are dispatched depending on the actual type of the receiver object: calling a virtual method m on a reference r of static type A invokes the implementation of m in one of A's sub-classes. Thus, if r points to an object of type B, then B's implementation of m is executed. Virtual method calls therefore act as type switches. Assuming that the complete type hierarchy is known and static, the call r.m() is equivalent to the following chain of branches:

```
if (r.getClass() == A) { A.m(r); }          // Pass r as 'this' to A.m()
else ...                                    // List sub-classes
else if (r.getClass() == C) { C.m(r); }
```

The symbolic execution phase must hence track the types of symbolic objects. Otherwise, implementations of virtual methods belonging to different types may be invoked on the same object, leading to invalid execution paths.

However, insisting on a single type per symbolic object can result in repeated executions. A sub-class shares the implementation of a virtual method with its super-class if it does not override the method. If the symbolic phase considers only one type at a time, an object that can have a range of (sub-) types requires multiple executions—one per type. This means the same virtual method implementation may be considered repeatedly. To avoid this inefficiency, type constraints for objects should be implemented as *sub*-type constraints that designate sub-trees (or sub-graphs with multiple inheritance) in the class hierarchy.

Adding support for sub-type constraints is a central aspect of adapting symcretic execution to object-oriented languages. One strategy for implementing such constraints is to push an algebraic encoding of the program's type hierarchy into the SMT solver [10]. Another strategy is to create a custom solver theory [92] for sub-type constraints. This can lead to significant performance improvements over the algebraic encoding. Work on extending the solvers to support incremental solving could improve performance further.

6.1.2 Concrete Phase

Supporting objects in the concrete phase of symcretic execution poses both research and engineering challenges.

The research challenges concern improving heuristic search algorithms that can solve constraints that involve objects and graph structures. Such constraints may be generated by the symbolic phase when it cannot solve the constraints, for example because they involve external method calls, or because they are too expensive to solve symbolically. A key aspect affecting the strength of a heuristic solver is how the solver measures the fitness of candidate solutions (*neighbors* in the concolic walk algorithm). While numeric constraints, such as $x \geq 3$ offer an intuitive way of measuring fitness, compare Algorithm 5.2 on page 52, object and type constraints lack such an intuitive fitness metric. Object fitness metrics and their integration into heuristic solvers have been explored in the context of search-based software testing (SBST; see section 2.6). However, SBST typically does not include extensive symbolic analysis of the target program. The proposed fitness metrics and solving algorithms could therefore benefit from an integration of knowledge, such as partial solutions, gained during the symbolic phase of symcretic execution.

The engineering challenges concern the generation and execution of the program traces on which the heuristic search operates. If the traces are executed by an interpreter, as in Cilocnoc then the interpreter must be extended considerably: to support object operations, the interpreter must implement the type hierarchy, virtual method calls, field operations, and some form of memory management. If the traces are executed natively, then the trace generation mechanism must consider a range of side conditions to ensure that it produces valid programs. For example, instantiated classes must be concrete. Thus, a concrete class must be chosen in cases where object constraints are expressed on abstract classes, or on ranges of sub-classes.

6.1.3 Object Creation Problem

The preceding discussion has focused on challenges related to supporting objects within the program execution. However, partial object-oriented programs, such as libraries, often query the state of an input object to determine the execution path. For example, an if-statement may contain the condition list.size > 2 for an input object called list. Because list is an input, the query whether list.size > 2 is satisfiable cannot be answered by exploring the program path. Instead, answering the query requires determining whether one of the object's legal states satisfies the condition, that is whether it is possible to generate an object in a satisfying state by only using the object's public interface. This problem is known as the object creation problem (OCP) [114, 102]. It is common to input generation techniques based on symbolic execution.

Solving the OCP requires finding a sequence of method invocations, starting with a constructor call, that updates the object's state such that it satisfies all given constraints. In our list example, the problem is creating a List instance with at least three elements. One possible solution is the following call sequence:

```
List<Object> list = new LinkedList<Object>();
list.add(new Object());
list.add(new Object();
list.add(new Object());
```

The OCP is undecidable in general. To see this, consider an input object with a single method that simulates a step of a Turing machine [106] for a given input. When the machine reaches an accepting state, the method sets the object's finished field to **true**. Determining whether an object o exists that satisfies the constraint o.finished = **true** is equivalent to solving the halting problem.

Finding effective and efficient methods for solving OCPs is an important research direction. Because OCPs occur in many symbolic executions of partial object-oriented programs [114], better solving methods will improve the usefulness of a wide range of programming tools.

6.2 Support for Concurrency

The output of a concurrent program can depend on its schedule, that is, on the ordering in which the program operations occur. For example, the order of the write operations determines which value will be read from a variable after concurrently writing the values 23 and 42 to it. Covering a target in a concurrent program thus requires knowledge of a suitable schedule because the target may be reachable only under some schedules. This makes the schedule one of the inputs that an input generator must find.

Unfortunately, concurrency aggravates the problem of having a large number of execution paths to explore that is faced by input generators. At every scheduling point, concurrency adds branches to the execution path regarding the order of the events that could happen concurrently at this point. As a result, a concurrent program can have exponentially more execution paths than its sequential version, even if techniques like partial order reduction [107, 82, 83] are applied.

Extending symcretic execution to concurrent programs requires two components: an efficient way to construct schedules during the symbolic phase, and a method of integrating scheduling into the concrete phase.

Actor Model

We discuss the challenge of extending symcretic execution to concurrent programs in the context of the Actor model of concurrency [57]. Our choice is motivated by two of its properties:

1. The big-step semantics [1] of the Actor model reduce the number of scheduling points, and therefore the number of execution paths.
2. The communication between concurrent entities is explicit, which simplifies the analysis. In contrast, communication via shared memory, which is common in multi-threaded programming, can be hard to detect statically.

Despite our focus on Actors, the presented problems transfer to symcretic execution of multi-threaded programs and programs that mix the paradigms [99].

```
 1 actor a {                           // Create an Actor instance 'a'.
 2   int pongCount = 0;                // Declare an integer field 'pongCount' in 'a'.
 3
 4   message init() {                  // Handler for the system startup message.
 5     b!ping();                       // Send message 'ping' to Actor 'b'.
 6     b!pingping();
 7   }
 8   message pong() {                  // Handler for 'pong' messages.
 9     this.pongCount++;
10   }
11 }
12
13 actor b {
14   message ping() {
15     a!pong();
16   }
17   message pingping() {
18     a!pong();
19     a!pong();
20   }
21 }
```

Figure 6.2: Simple ping–pong program that demonstrates the used Actor language. The program consists of two Actors a and b (lines 1 and 13). Actor a sends two messages, ping and pingping to Actor b, which replies with one, respectively two, pong messages (lines 5–6 and 15, 18–19). Upon receiving a pong message, Actor a increments its pongCount field.

Actors [57, 2] can be thought of as concurrently executing objects that communicate via fair asynchronous message passing. Upon receiving a message, an Actor processes the message in a single atomic step, using a single thread of execution. While processing the message with the respective message handler, the Actor can change its state, send messages, and create other Actors. Unlike threads, Actors never share state. Messages may arrive out of order: despite an Actor first sending the message m_1 and then m_2 while processing a message, the recipient may first see m_2 and then m_1.

Figure 6.2 shows an example Actor program. The program consists of two Actors a and b (lines 1 and 13) that exchange messages: Actor a sends two messages, ping and pingping to Actor b, which replies with one, respectively

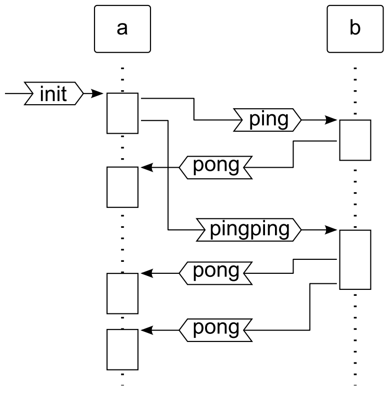

Figure 6.3: Sequence diagram showing one of the message schedules of the ping–pong program from Figure 6.2. Other schedules exist: for example, the ping and pingping messages could arrive in reversed order because they are sent as a reaction to the same message.

two, pong messages (lines 5–6 and 15, 18–19). Upon receiving a pong, Actor a increments its pongCount field. Figure 6.3 shows one possible schedule of the program.

To simplify the exposition, the used language makes the following provisions: The actor keyword defines an Actor instance (as opposed to a class of Actors). Actor names are visible globally; thus, Actor a knows Actor b and vice versa without any introduction. Actors send messages using the bang operator (!) and define handlers for the messages they understand using the message keyword. Upon receiving a message, the respective handler is executed. The system starts by sending the init message to the first defined Actor (here: a).

6.2.1 Symbolic Phase

A central problem when extending symbolic backward execution to Actor programs is the efficient construction and navigation of the message schedule. First, in contrast to the sequential case discussed in section 4.2, the backward execution must now consider that reaching a target may require multiple messages that were sent concurrently (in the happens-before sense [71]) by independent Actors. Consequently, the search for an execution path from a target to an entry point can no longer rely on simply moving backwards in

```
 1 class SynchronousError {
 2   int count = 0
 3
 4   void init() {
 5     this.failIfPositive();                    // Function call
 6     this.count++;
 7   }
 8   void failIfPositive() {
 9     if (this.count > 0) {
10       error();
11     }
12   }
13 }
```

Figure 6.4: Sequential program in which an object checks an error condition using method calls (synchronous messages). Within the init method, the call to the failIfPositive() method occurs before incrementing the count field. Thus, the error condition (line 10) cannot be triggered.

the call graph. Second, the backward execution must consider that a target may be reachable despite some of the participating Actors being stuck in infinite loops. The remainder of this section illustrates both challenges.

Backward Exploration of Actor Programs

Recall that the symbolic phase of symcretic execution of sequential programs explores the call graph backwards—from callee to caller—until it finds an entry point. Intuitively, this backward exploration corresponds to unwinding the call stack in the single thread of execution. For example, moving from the failIfPositive method in line 8 of the program displayed in Figure 6.4 to its call site in line 5 corresponds to entering the stack frame of the init method, which must have been under that of the failIfPositive method. The call sequence graph in Figure 6.5 illustrates this step.

The sequential backward execution relies on two guarantees: (1) At any program point, the state was influenced only by operations that lie along the program path from the point to the program entry point. (2) Operations that follow a call site cannot influence the execution within the called method.

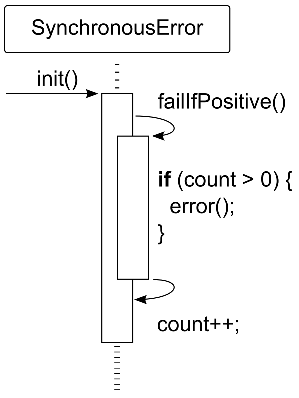

Figure 6.5: Sequence diagram showing the (only) schedule of events in the program from Figure 6.4.

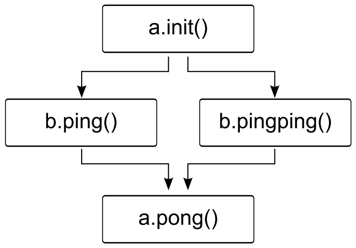

Figure 6.6: Message send graph of the ping–pong program from Figure 6.2. The nodes of the graph are the message handlers defined in the program. Two nodes m_1 and m_2 are connected by a (directed) edge from m_1 to m_2 if m_1 sends a message of type m_2.

Thus, while moving backwards from the target in line 10 of the example program (Figure 6.4), it is safe to ignore the increment of the count field in line 6 because it must happen after the call. Likewise, no other operation could have incremented the count field, which guarantees that the target is unreachable.

For Actor programs, this call graph exploration can be adapted to an exploration of the *message send graph*. The nodes of a program's message send graph are the defined message handlers. Two nodes m_1 and m_2 are connected by a (directed) edge from m_1 to m_2 if m_1 sends a message of type m_2. Similar to a method call site, the position in m_1's body where the send occurs is called the *message send site*. Figure 6.6 shows the message send graph of the ping–pong program from Figure 6.2.

```
 1 actor asynchronousError {
 2   int count = 0
 3
 4   message init() {
 5     this!failIfPositive();                  // Message send
 6     this.count++;
 7   }
 8   message failIfPositive() {
 9     if (this.count > 0) {
10       error();
11     }
12   }
13 }
```

Figure 6.7: Program in which an Actor checks an error condition using asynchronous messages. Within the init message handler, sending the failIfPositive() message occurs before incrementing the count field. However, as message handlers must complete before the next message can be processed, the error condition (line 10) is triggered.

Unfortunately, following message send edges backwards lacks the guarantees of following call edges because message delivery is asynchronous and message handlers always take full effect in an Actor before the net message is processed.

The backward exploration thus cannot always ignore statements that follow a send site. For example, consider the Actor-version of the previous example, which is shown in Figure 6.7. The Actorized example replaces the call to the failIfPositive method with a message send (line 5). As the init handler must complete before the sent message can be handled, the Actor's count field has value 1 when the failIfPositive handler executes. Consequently, the error on line 10 is triggered. The sequence diagram (Figure 6.8) illustrates the execution. The symbolic phase must therefore explore message handlers starting from the end, not the send site, or it may miss feasible targets.

A more severe problem is that the state of an Actor at a program point is not necessarily fully determined by operations along the program path that runs from the point to the program entry point. Instead, the Actor's state may depend on messages sent in parallel along other program paths. The

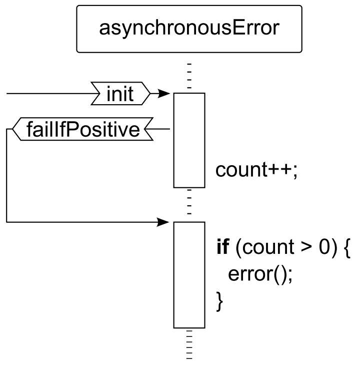

Figure 6.8: Sequence diagram showing the (only) schedule of events in the program from Figure 6.7.

order in which an Actor processes such messages—the schedule—determines its state.

For example, assume that we want to cover the error condition in line 14 of the program shown in Figure 6.9. Backwards execution within the enclosing failIfOne message handler yields the path condition **this**.count $= 1$. Following the message send graph (Figure 6.10) backwards leads to the init handler, which is the program's entry point. As discussed above, the backward execution must start from the end of init, which generates three symbolic messages: the failIfOne message that brought us here, and two increment messages. These three messages are causally independent and may arrive at the target in any order. The path condition is satisfied only if one increment message is processed before the failIfOne message.

The example shows that the symbolic phase of symcretic execution of Actor programs must do more than traverse the message send graph backwards. Upon reaching the top of a message handler, it must determine which messages in which order are necessary to satisfy the path condition within the current message handler. In the example, upon reaching line 12 after starting from the target in line 14, it must determine that (exactly) one increment message must arrive before the failIfOne message. Thus, it must solve a variant of the object creation problem to find the required messages. Next, the backward execution must find a set of suitable send sites that generate these messages. The send sites must allow the messages to arrive in a satisfying order. Backward

```
 1 actor scheduleDependentError {
 2   int count = 0;
 3
 4   message init() {                    // When the system starts
 5     this!failIfOne();                 // send three messages to self.
 6     this!increment();
 7     this!increment();
 8   }
 9   message increment() {
10     this.count++;
11   }
12   message failIfOne() {
13     if (this.count == 1) {            // Satisfied for the message schedule
14       error();                        // increment → failIfOne → increment.
15     }
16   }
```

Figure 6.9: Actor program in which the reachability of the error condition (line 14) depends on the message schedule. The three messages sent in the init handler are independent and may arrive in arbitrary order. The error condition is triggered only if exactly one of the increment messages arrives before the failIfOne message.

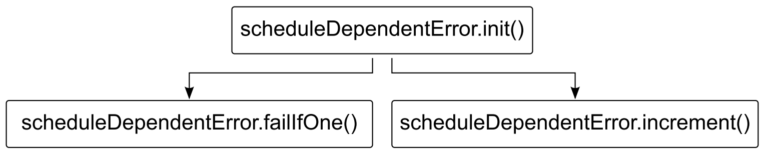
scheduleDependentError.init()
scheduleDependentError.failIfOne()
scheduleDependentError.increment()

Figure 6.10: Message send graph of the program shown in Figure 6.9.

execution must then continue at the message handlers containing the send sites. Ultimately, the process must converge towards a single entry point message handler.

Extending symcretic execution to Actor programs therefore depends on a viable approach to solving object creation problems.

Reduction Techniques

Ignoring the undecidable nature of the problem for a moment, exhaustive path and schedule exploration, for example using iterative deepening [93, 69], is viable only for small programs because of the exponential growth of the search space in the number of branches and scheduling points.

If the search for an exact solution becomes infeasible, the symbolic phase may choose to approximate the schedule. Schedules and execution paths obtained through approximation can then guide the concrete execution. The following three candidate approximations have been explored by the author in previous work:

Synchronization Constraints [30]. Synchronizers [42, 41] are declarative synchronization constraints that can be imposed on groups of Actors. The constraints express under which conditions an Actor is able to handle a message. Until the conditions are met, the message stays in the Actor's message queue. The constraints have a global effect and affect all messages an Actor receives. Conceptually, a Synchronizer can be seen as a special kind of Meta-Actor [79, 109] that observes and limits the message dispatch of other Actors. The conventional form of Synchronizers supports disabling and atomicity constraints:

- *Disabling constraints* prevent the constrained Actor from handling messages that match a given pattern. For example, by disabling the handlers for all but the initialization message, a disabling constraint ensures that an Actor dispatches (starts to process) the initialization message before it dispatches any other message.

- *Atomicity constraints* coordinate groups of Actors by bundling messages into indivisible sets. A constraint enforces that either all the messages in a set are dispatched, or none of them are (there is no partial delivery). The constraint provides *spatial* atomicity. An atomicity constraint can, for example, implement a simple online music payment scheme by fusing the *deduct money from credit card* message with the *enable download* message.

Atomic Sets [34]. Another approximation are *atomic sets* [37, 108], which coarsen the concurrency structure of the program and thus shrink the search space. Atomic sets can be dynamically inferred in a pre-processing step [34].

An atomic set is a group of data fields inside an object indicating that the fields are connected by a consistency invariant. Objects can contain multiple disjoint atomic sets. *Aliases* extend atomic sets beyond object boundaries. For example, consider the following consistency invariant of a list object: the value of the list's `length` field must equal the number of elements in the `entries` array used to store the list entries. Hence, the fields `length` and `entries` form an atomic set. Instead of requiring an explicit expression of the consistency invariant like `length == entries.length`, an atomic set is complemented by one or more *units of work*. A unit of work is a method that preserves the consistency of its associated atomic sets when executed sequentially. Thus, atomic sets can ensure the application's consistency by inserting synchronization operations that guarantee the sequential execution of all units of work. Atomic sets therefore reduce the amount of concurrency in the program, and consequently the number of schedules that have to be examined.

Session Types [20]. A further candidate approximation are *session types* [58, 59, 20], which express the order of messages sent by Actors [2] (or processes). The intended use of session types is to statically check whether a group of processes communicates according to a given specification. With the specification safely over-approximating the system behavior, it can be used to find candidate execution paths whose details have to be filled in using concrete execution.

In addition, limiting the exploration to unique representatives of every schedule class modulo the happens-before relation (partial-order reduction [107, 82, 83]) can shrink the search space significantly. Computing complete partial order reductions requires a full view of the search space, which is unavailable during backward execution. However, *dynamic* partial order reduction techniques [39, 100] can approximate the reduction under limited knowledge. The integration of such techniques into symcretic execution is the topic of future research.

Finally, treating scheduling as an orthogonal aspect to finding data inputs, as proposed in existing approaches and tools [90, 89, 84, 85, 87, 111, 72, 117], may prove insufficient for some programs. Dependencies between a concurrent program's schedule and its data inputs arise, for example, if the number of Actors created in the program depends on an input parameter. Ignoring the dependency can be problematic if reaching a target statement requires a minimum number of participating Actors. Adapting the existing approaches from their original usage scenarios to symcretic execution therefore requires further investigation.

Local Non-Termination

A second challenge for the efficient construction and navigation of the message schedule in Actor programs is local non-termination of Actors. In sequential programs, an infinite loop on the path leading to a target effectively makes the target unreachable. However, in Actor programs, a message handler of one Actor may fail to terminate, but the target may still be reachable in parallel.

For example, consider the program shown in Figure 6.11. The init handler of the infiniteLoop Actor (line 2) contains an infinite loop that sends fail messages to the triggeredError Actor. Thus, the init handler never terminates. Yet, a single fail message suffices to trigger the error in the triggeredError Actor, which is therefore reachable. A schedule navigation and construction mechanism should be able to cope with such cases without getting stuck.

```
 1 actor infiniteLoop {
 2   message init() {                      // When the system starts
 3     while (true) {                      // send messages in an infinite loop.
 4       triggeredError!fail();
 5     }
 6   }
 7 }
 8
 9 actor triggeredError {
10   message fail() {                      // Fail on the first received message.
11     error();
12   }
13 }
```

Figure 6.11: Actor program whose error condition in the triggeredError Actor (line 11) is reachable despite the infiniteLoop Actor being stuck in an infinite loop.

6.2.2 Concrete Phase

The discussion from integrating concurrency with the symbolic phase of symcretic execution extends to the concrete phase. The central research question is how to efficiently find schedules that satisfy the constraints derived during the symbolic phase and potentially learned during prior executions of the target program [34, 61]. Thus, an important direction of future research is devising fitness functions for schedules that allow quick convergence towards a solution. Prior work in this area [117] relies on ad-hoc fitness functions to solve the problem at hand; a systematic investigation, potentially incorporating coverage information [101], could help to better understand the strengths and weaknesses of such fitness functions. In addition to the definition of fitness functions, it is unclear how partial order reduction can be integrated with heuristic search and the scheduling constraints derived during the symbolic phase.

The engineering challenges of supporting Actors in the concrete phase of symcretic execution concern the implementation of scheduling. The concrete phase must support concurrency while allowing the enforcement of scheduling constraints. The scheduling constraints are necessary to guide the execution towards a satisfying schedule while avoiding known non-solution schedules. At the same time, the scheduling constraints must respect concurrency effects in the target program: a simple linearization of the schedule that sequentializes the target program, for example, cannot handle local non-termination of individual Actors. Thus, the concrete execution environment must implement an Actor runtime system. In particular it must implement the scheduler.

Implementing the concrete phase as an interpreter simplifies meeting these requirements. However, building a faithful interpreter for the whole Actor language can be tricky as the language semantics may not be formalized. For example, there is no formal specification of the Akka framework [12]; the implementation is the specification. Another difficulty is achieving satisfying performance. Translating the trace generated by the symbolic phase to code that can be executed on the target platform avoids these problems. However, the challenges are similar to the ones for supporting objects.

Chapter 7

Conclusions

This dissertation focuses on the problem of the effective and efficient generation of program inputs that trigger the execution of a specific statement or other target in the program. Such target-specific inputs are useful, for example, in debugging, where they allow programmers to step through the program towards a bug, or in regression testing, where they constitute additional test cases that cover a changed piece of code.

The dissertation describes a novel target-specific input generation method called *symcretic execution* (chapter 3 and chapter 4). Symcretic execution first finds an execution path to the target and then solves the constraints along the path. It avoids considering irrelevant paths by exploring the program backwards, starting at the target and moving towards the program entry. At the same time, it excludes infeasible paths. To mitigate the problems of undecidable constraints and data-dependent loops, it integrates heuristic constraint solving based on concrete execution of (parts of) the program. A comparison with related approaches and an empirical evaluation show that symcretic execution finds more inputs than concolic testing while exploring fewer paths.

The problem of undecidable constraints, which motivates the concrete phase of symcretic execution, also appears in customary concolic and symbolic execution. The novel algorithm for solving complex arithmetic path conditions described in chapter 5 not only applies to the concrete phase of symcretic execution, but also to concolic and symbolic execution. The algorithm, called *concolic walk*, uses heuristic search based on a geometric interpretation of the task of finding inputs. An evaluation shows that the concolic walk algorithm finds more solutions than customary simplification-based heuristics and furthermore improves the strength of state-of-the-art concolic test generators.

The dissertation presents all algorithms in the context of small imperative sequential programming languages. While the algorithms can be extended to include objects and concurrency, making these extensions requires further research to maintain the efficiency of the algorithms. The respective challenges are discussed in chapter 6.

Bibliography

[1] Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, 1997.

[2] Gul A. Agha. *ACTORS — A model of concurrent computation in distributed systems.* MIT Press series in artificial intelligence. MIT Press, 1990.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 1986.

[4] Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Symbolic execution of concurrent objects in CLP. In *PADL'12*, volume 7149 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 2012.

[5] Elvira Albert, Miguel Gómez-Zamalloa, and Germán Puebla. Test data generation of bytecode by CLP partial evaluation. In *LOPSTR'08*, volume 5438 of *Lecture Notes in Computer Science*, pages 4–23. Springer, 2009.

[6] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *TACAS'08*, volume 4963 of *Lecture Notes in Computer Science*, pages 367–381. Springer, 2008.

[7] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, University of Copenhagen, 1994.

[8] Arthur I. Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja E. J. Vos. Symbolic search-based testing. In *ASE'11*, pages 53–62. IEEE, 2011.

[9] Boris Beizer. *Software Testing Techniques.* International Thomson Publishing, second edition, 1990.

[10] Nikolaj Bjørner. Engineering theories with Z3. In *APLAS'11*, volume 7078 of *Lecture Notes in Computer Science*, pages 4–16. Springer, 2011.

[11] Joshua Bloch. *Effective Java.* Addison-Wesley, second edition, 2008.

[12] Jonas Bonér, Viktor Klang, Roland Kuhn, et al. Akka library. `http://akka.io`.

[13] Mateus Borges, Marcelo d'Amorim, Saswat Anand, David H. Bushnell, and Corina S. Păsăreanu. Symbolic execution with interval solving and meta-heuristic search. In *ICST'12*, pages 111–120. IEEE, 2012.

[14] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT—A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10(6):234–245, April 1975.

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

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

[17] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *CCS'06*, pages 322–335. ACM, 2006.

[18] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

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

[20] Minas Charalambides, Peter Dinges, and Gul Agha. Parameterized concurrent multi-party session types. In Natallia Kokash and António Ravara, editors, *FOCLASA*, volume 91 of *Electronic Proceedings in Theoretical Computer Science*, pages 16–30. Open Publishing Association, 2012.

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

[22] Lori A. Clarke. A program testing system. In *Proceedings of the 1976 annual conference*, ACM '76, pages 488–491, New York, NY, USA, 1976. ACM.

[23] Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. In *SAGA'01*, volume 2264 of *Lecture Notes in Computer Science*, pages 73–90. Springer, 2001.

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

[25] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS'08*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[26] Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[27] Xianghua Deng, Jooyong Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE'06*, pages 157–166. IEEE Computer Society, 2006.

[28] Xianghua Deng, Robby, and John Hatcliff. Kiasan: A verification and test-case generation framework for java based on symbolic execution. In *ISoLA'06*, page 137. IEEE, 2006.

[29] Xianghua Deng, Robby, and John Hatcliff. Towards a case-optimal symbolic execution algorithm for analyzing strong properties of object-oriented programs. In *SEFM'07*, pages 273–282. IEEE Computer Society, 2007.

[30] Peter Dinges and Gul Agha. Scoped synchronization constraints for large scale actor systems. In *COORDINATION'12*, pages 89–103, 2012.

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

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

[33] Peter Dinges and Gul Agha. Targeted test input generation using symbolic-concrete backward execution. Technical report, University of Illinois at Urbana–Champaign, September 2014.

[34] Peter Dinges, Minas Charalambides, and Gul Agha. Automated inference of atomic sets for safe concurrent execution. In *PASTE'13*, pages 1–8. ACM, 2013.

[35] TheAnh Do, Alvis Cheuk M. Fong, and Russel Pears. Precise guidance to dynamic test generation. In *ENASE'12*, pages 5–12. SciTePress, 2012.

[36] Julian Dolby, Stephen J. Fink, and Manu Sridharan. T. J. Watson libraries for analysis (WALA). `http://wala.sf.net`.

[37] Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. A data-centric approach to synchronization. *ACM Trans. Program. Lang. Syst.*, 34(1):4, 2012.

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

[39] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL'05*, pages 110–121, 2005.

[40] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *SIGSOFT FSE'11*, pages 416–419. ACM, 2011.

[41] Svend Frølund. *Coordinating distributed objects - an actor-based approach to synchronization.* MIT Press, 1996.

[42] Svend Frølund and Gul Agha. A language framework for multi-object coordination. In *ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern*, volume 707 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 1993.

[43] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *ISSRE'13*, pages 360–369. IEEE, 2013.

[44] James Gleick. A bug and a crash. `http://www.around.com/ariane.html`. Retrieved on Apr. 4, 2014.

[45] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *ISSTA'13*, pages 302–313. ACM, 2013.

[46] Fred Glover. Tabu search — part i. *INFORMS Journal on Computing*, 1(3):190–206, 1989.

[47] Fred Glover. Tabu search — part ii. *INFORMS Journal on Computing*, 2(1):4–32, 1990.

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

[49] Patrice Godefroid. Higher-order test generation. In *PLDI'11*, pages 258–269. ACM, 2011.

[50] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI'05*, pages 213–223. ACM, 2005.

[51] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. Technical Report TR-2007-58, Microsoft Research, 2007.

[52] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: Whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[53] Miguel Gómez-Zamalloa, Elvira Albert, and Germán Puebla. Decompilation of java bytecode to prolog by partial evaluation. *Information & Software Technology*, 51(10):1409–1427, 2009.

[54] Miguel Gómez-Zamalloa, Elvira Albert, and Germán Puebla. Test case generation for object-oriented imperative languages in CLP. *TPLP'10*, 10(4-6):659–674, 2010.

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

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

[57] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI'73*, pages 235–245. Morgan Kaufmann, 1973.

[58] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP'98*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[59] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.

[60] William E. Howden. Methodology for the generation of program test data. *IEEE Trans. Computers*, 24(5):554–560, 1975.

[61] Jeff Huang, Charles Zhang, and Julian Dolby. CLAP: recording local executions to reproduce concurrency failures. In *PLDI'13*, pages 141–152. ACM, 2013.

[62] Kobi Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *ASE'08*, pages 297–306. IEEE, 2008.

[63] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: the alloy constraint analyzer. In *ICSE'00*, pages 730–733. ACM, 2000.

[64] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948 vol.4, 1995.

[65] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *TACAS'03*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[66] James C. King. A new approach to program testing. In *Programming Methodology*, pages 278–290, 1974.

[67] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

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

[69] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.

[70] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan de Halleux. FloPSy — Search-based floating point constraint solving for symbolic execution. In *ICTSS'10*, volume 6435 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2010.

[71] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[72] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *SPIN'01*, volume 2057 of *Lecture Notes in Computer Science*, pages 80–102. Springer, 2001.

[73] Kin-Keung Ma, Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *SAS'11*, pages 95–111, 2011.

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

[75] Jan Malburg and Gordon Fraser. Combining search-based and constraint-based testing. In *ASE'11*, pages 436–439. IEEE, 2011.

[76] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. Pse: explaining program failures via postmortem static analysis. In *SIGSOFT FSE'04*, pages 63–72, 2004.

[77] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

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

[79] José Meseguer and Carolyn L. Talcott. Semantic models for distributed object reflection. In *ECOOP 2002 - Object-Oriented Programming, 16th European Conference*, volume 2374 of *Lecture Notes in Computer Science*, pages 1–36. Springer, 2002.

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

[81] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for java. In *OOPSLA'07 Companion*, pages 815–816. ACM, 2007.

[82] Doron Peled. All from one, one for all: on model checking using representatives. In *CAV'93*, pages 409–423, 1993.

[83] Doron Peled. Ten years of partial order reduction. In *CAV'98*, pages 17–28. Springer, 1998.

[84] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA'08*, pages 15–26. ACM, 2008.

[85] Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic execution of java bytecode. In *ASE'10*, pages 179–180. ACM, 2010.

[86] Corina S. Păsăreanu, Neha Rungta, and Willem Visser. Symbolic execution with mixed concrete-symbolic solving. In *ISSTA'11*, pages 34–44. ACM, 2011.

[87] Corina S. Păsăreanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz, and Neha Rungta. Symbolic PathFinder: Integrating symbolic execution with model checking for Java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.

[88] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*, pages 49–61. ACM, 1995.

[89] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV'06*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006.

[90] Koushik Sen and Gul Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa Verification Conference*, pages 166–182, 2006.

[91] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for c. In *ESEC/SIGSOFT FSE'05*, pages 263–272. ACM, 2005.

[92] Elena Sherman, Brady J. Garvin, and Matthew B. Dwyer. A slice-based decision procedure for type-based partial orders. In *IJCAR'10*, volume 6173 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2010.

[93] David J. Slate and Lawrence R. Atkin. CHESS 4.5—the northwestern university chess program. In Peter W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer, 1983.

[94] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Păsăreanu. CORAL: Solving complex constraints for Symbolic PathFinder. In *NASA Formal Methods*, pages 359–374, 2011.

[95] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL'96*, pages 32–41. ACM, 1996.

[96] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[97] Mitsuo Takaki, Diego Cavalcanti, Rohit Gheyi, Juliano Iyoda, Marcelo d'Amorim, and Ricardo Bastos Cavalcante Prudêncio. A comparative study of randomized constraint solvers for random-symbolic testing. In *NASA Formal Methods*, pages 56–65, 2009.

[98] Mitsuo Takaki, Diego Cavalcanti, Rohit Gheyi, Juliano Iyoda, Marcelo d'Amorim, and Ricardo Bastos Cavalcante Prudêncio. Randomized constraint solvers: a comparative study. *ISSE'10*, 6(3):243–253, 2010.

[99] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *ECOOP'13*, volume 7920 of *Lecture Notes in Computer Science*, pages 302–326. Springer, 2013.

[100] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. Transdpor: A novel dynamic partial-order reduction technique for testing actor programs. In *FMOODS/-FORTE 2012*, pages 219–234. Springer, 2012.

[101] Samira Tasharofi, Michael Pradel, Yu Lin, and Ralph E. Johnson. Bita: Coverage-guided, automatic testing of actor programs. In *ASE'13*, pages 114–124. IEEE, 2013.

[102] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. Synthesizing method sequences for high-coverage testing. In *OOPSLA'11*, pages 189–206, 2011.

[103] Nikolai Tillmann and Jonathan de Halleux. Pex—White box test generation for .NET. In *TAP'08*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer, 2008.

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

[105] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In *TACAS'07*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

[106] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1937.

[107] Antti Valmari. Stubborn sets for reduced state space generation. In *Applications and Theory of Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.

[108] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL'06*, pages 334–345. ACM, 2006.

[109] Nalini Venkatasubramanian and Carolyn L. Talcott. Reasoning about meta level activities in open distributed systems. In *PODC*, pages 144–152, 1995.

[110] Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *SIGSOFT FSE'12*, page 58. ACM, 2012.

[111] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[112] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *ISSTA'04*, pages 97–107. ACM, 2004.

[113] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *EDCC'05*, volume 3463 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2005.

[114] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Precise identification of problems for structural test generation. In *ICSE'11*, pages 611–620. ACM, 2011.

[115] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN'09*, pages 359–368. IEEE, 2009.

[116] Guowei Yang, Sarfraz Khurshid, and Corina S. Păsăreanu. Memoise: a tool for memoized symbolic execution. In *ICSE'13*, pages 1343–1346. IEEE / ACM, 2013.

[117] Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys'10*, pages 321–334. ACM, 2010.