Higher-Order Test Generation

Patrice Godefroid
Microsoft Research
pg@microsoft.com

Abstract

Symbolic reasoning about large programs is bound to be imprecise. How to deal with this imprecision is a fundamental problem in program analysis. Imprecision forces approximation. Traditional static program verification builds "may" over-approximations of the program behaviors to check universal "for-all-paths" properties, while automatic test generation requires "must" under-approximations to check existential "for-some-path" properties.

In this paper, we introduce a new approach to test generation where tests are derived from *validity proofs* of first-order logic formulas, rather than *satisfying assignments* of quantifier-free first-order logic formulas as usual. Two key ingredients of this *higher-order test generation* are to (1) represent complex/unknown program functions/instructions causing imprecision in symbolic execution by *uninterpreted functions*, and (2) record *uninterpreted function samples* capturing input-output pairs observed at execution time for those functions. We show that higher-order test generation generalizes and is more precise than simplifying complex symbolic expressions using their concrete runtime values. We present several program examples where our approach can exercise program paths and find bugs missed by previous techniques. We discuss the implementability and applications of this approach. We also explain in what sense dynamic test generation is more powerful than static test generation.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Testing, Verification

Keywords Automatic Test Generation, Software Model Checking, Uninterpreted Functions

1. Introduction

Automatic code-driven test generation aims at proving existential properties of programs: does there exist a test input that can exercise a specific program branch or statement, or follow a specific program path, or trigger a bug? Test generation dualizes traditional program verification and static program analysis aimed at proving universal properties which holds for all program paths, such as "there are no bugs of type X in this program".

PLDI'11, June 4–8, 2011, San Jose, California, USA.

Symbolic reasoning about large programs is bound to be imprecise. If perfect bit-precise symbolic reasoning was possible, static program analysis would detect standard programming errors without reporting false alarms. How to deal with this imprecision is a fundamental problem in program analysis. Traditional static program verification builds "may" over-approximations of the program behaviors in order to prove correctness, but at the cost of reporting false alarms. Dually, automatic test generation requires "must" under-approximations in order to drive program executions and find bugs without reporting false alarms, but at the cost of possibly missing bugs.

Most of the program analysis literature discusses program verification for universal properties. Yet, except for static type systems, the biggest practical impact of program analysis so far has been bug finding, not proving the absence of bugs. The study of effective program verification techniques for existential properties (i.e., "sound bug finding") has recently experienced quite a resurgence. A catalyst is arguably recent work on systematic dynamic test generation [15], and related extensions and tools (e.g., [2, 6, 17, 23, 24]). Over the last few years, these techniques have been made more scalable [16], and have been used to find many new security vulnerabilities in Windows [12] and Linux [22] applications.

Work on automatic code-driven test generation can roughly be partitioned into two groups: *static* versus *dynamic* test generation. Static test generation [20] consists of analyzing a program P *statically*, by reading the program code and using symbolic execution techniques to simulate abstract program executions in order to attempt to compute inputs to drive P along specific execution paths or branches, *without ever executing the program*. On the other hand, *dynamic test generation* [21] consists of executing the program P starting with some given or random concrete inputs, gathering symbolic constraints on inputs at conditional statements along the execution, and then using a constraint solver to infer variants of the previous inputs in order to steer the next execution of the program towards an alternative program branch; this process can be repeated with the goal of *systematically* executing all (or as many as possible) feasible program paths, while checking each execution using run-time checking tools (like Purify, Valgrind or AppVerifier) for detecting various types of errors [15].

It is argued in [15] that dynamic test generation is more powerful than static test generation because imprecision in symbolic execution can be alleviated using concrete values and randomization: whenever symbolic execution does not know how to generate a constraint for a program statement depending on some inputs, one can always simplify this constraint using the concrete runtime values of those inputs. To illustrate this point, consider the following program example [11]:

```
int obscure(int x, int y) {
  if (x == hash(y)) return -1;   // error
  return 0;                      // ok
}
```

Assume the constraint solver cannot "symbolically reason" about the function `hash` (perhaps because it is too complex or simply because its code is not available). This means that the constraint solver cannot generate two values for inputs `x` and `y` that are guaranteed to satisfy (or violate) the constraint `x == hash(y)`. In this case, static test generation cannot generate test inputs to drive the execution of the program `obscure` through either branch of the conditional statement: static test generation is *helpless* for a program like this. Note that, for test generation, it is not sufficient to know that the constraint `x == hash(y)` is satisfiable for *some* values of `x` and `y`, it is also necessary to generate *specific values* for `x` and `y` that satisfy or violate this constraint.

In contrast, dynamic test generation can easily generate, for a fixed value of `y`, a value of `x` that is equal to `hash(y)` since the latter concrete value is known at runtime. By picking randomly and then fixing the value of `y`, we can, in the next test execution, set the value of the other input `x` either to `hash(y)` or to something else in order to force the execution of the then or else branches, respectively, of the test in the function `obscure`.

In summary, static test generation is unable to generate test inputs to control the execution of the program `obscure`, while dynamic test generation can *easily* drive the executions of that same program through all its feasible program paths. In realistic programs, imprecision in symbolic execution typically creeps in in many places, and dynamic test generation allows test generation to recover from that imprecision. Dynamic test generation can be viewed as extending static test generation with additional runtime information, and is therefore more general and powerful.

But how much more powerful? How often can this concretization trick be used? It would not work in the case of a constraint like `hash(x)==hash(y)+1`. Does there exist an algorithm to determine in which cases concretization "works" and when it does not? Can concretization be modeled symbolically and therefore simulated by static symbolic execution and test generation? If so, what is the fundamental difference between static and dynamic test generation? Can one formalize both and *prove* (and clarify how and why) they are different? Is it possible to deal with imprecision in symbolic reasoning differently, in order to enable even more powerful test generation?

The purpose of this paper is to answer all these questions, which are central to test generation and program analysis. We start by carefully formalizing "concretization" as introduced in [15], and show that it may or may not generate sound path constraints (Section 3). We then introduce (Section 4) a new more general form of test generation, which we call *higher-order* because it uses a higher-order logic representation of program paths. Higher-order test generation uses *uninterpreted functions* to represent unknown functions or instructions during symbolic execution, records *uninterpreted function samples* capturing concrete input-output pairs observed at execution time for those functions, and generates new test inputs from *validity proofs* of first-order logic formulas with uninterpreted functions. We then show (in Section 5) that higher-order test generation can not only fully "simulate" concretization when the latter is done in a sound manner, but that it is also more general and powerful. We discuss how to implement this approach in practice in Section 6, and present an application (in Section 7) which requires the power of higher-order test generation: parsers with input lexers using hash functions for fast keyword recognition. We conclude (in Section 9) by clarifying in what sense dynamic test generation is more powerful than static test generation.

2. Background: Systematic Dynamic Test Generation

Dynamic test generation (see [15] for further details) consists of running the program P under test both concretely, executing the actual program, and symbolically, calculating constraints on values stored in program variables v and expressed in terms of input parameters. Side-by-side concrete and symbolic executions are performed using a concrete store M and a symbolic store S, which are mappings from *memory addresses* (where program variables are stored) to concrete and symbolic values respectively. A *symbolic value* is any expression e in some theory $\mathcal{T}$ where all free variables are exclusively input parameters. For any program variable v, $M(v)$ denotes the *concrete value* of v in M, while $S(v)$ denotes the *symbolic value* of v in S. For notational convenience, we assume that $S(v)$ is always defined and is simply $M(v)$ by default if no symbolic expression in terms of inputs is associated with v in S. When $S(v)$ is different from $M(v)$, we say that that program variable v is "symbolic", meaning that the value of program variable v is a function of some input(s) which is represented by the symbolic expression $S(v)$ associated with v in the symbolic store. We also extend this notation to allow $M(e)$ to denote the concrete value of symbolic expression e when evaluated with the concrete store M. The notation $+$ for mappings denotes updating; for example, $M' = M + [m \mapsto e]$ is the same map as M, except that $M'(m) = e$.

The program P manipulates the memory (concrete and symbolic stores) through statements, or *commands*, that are abstractions of the machine instructions actually executed. We assume a command can be an *assignment* of the form $v := e$ (where v is a program variable and e is an expression), a *conditional statement* of the form `if` e `then` C' `else` C'' where e denotes a boolean expression, and C' and C'' denote the unique[1] next command to be evaluated when e holds or does not hold, respectively, or `stop` corresponding to a program error or normal termination.

Given an input vector I assigning a concrete value I_i to the i-th input parameter, the evaluation of a program defines a unique finite[2] *program execution* $s_0 \stackrel{C_1}{\rightarrow} s_1 \ldots \stackrel{C_n}{\rightarrow} s_n$ that executes the finite sequence $C_1 \ldots C_n$ of commands and goes through the finite sequence $s_1 \ldots s_n$ of program states. Each *program state* is a tuple $\langle C, M, S, pc \rangle$ where C is the next command to be evaluated, and pc is a special meta-variable that represents the current path constraint. For a finite sequence w of commands (i.e., a control path w), a *path constraint* pc_w is a quantifier-free first-order logic formula over theory $\mathcal{T}$ that is meant to characterize the input assignments for which the program executes along w. The path constraint is *sound and complete* when this characterization is exact, i.e., when the two following conditions are satisfied.

DEFINITION 1. A path constraint pc_w is *sound* if every input assignment satisfying pc_w defines a program execution following path w. ■

DEFINITION 2. A path constraint pc_w is *complete* if every input assignment following path w is a satisfying assignment, or *model*, of pc_w. ■

Path constraints are generated during dynamic symbolic execution by collecting input constraints at conditional statements, as illustrated in Figure 2. Figure 1 illustrates how to symbolically evaluate expressions e occurring in individual program instructions (line 14 should be ignored for now). The notation $\&v$ denotes the

[1] We assume program executions are sequential and deterministic.

[2] We assume program executions terminate. In practice, a timeout prevents non-terminating program executions and issues a runtime error.

```
1  evalSymbolic(e) =
2    match (e):
3      case v: // Program variable v
4        return S(&v)
5      case +(e1, e2): // Addition
6        f1 = evalSymbolic(e1)
7        f2 = evalSymbolic(e2)
8        if f1 and f2 are constants
9            return evalConcrete(e)
10       else
11           return createExpression('+',f1,f2)
12     etc.
13     default: // default for unhandled expression
14       // pc = pc ∧ ⋀_{xi∈e}(xi = Ii)
15       return evalConcrete(e)
```

Figure 1. Symbolic expression evaluation.

```
1  Procedure executeSymbolic(P,I) =
2    initialize M0 and S0
3    path constraint pc = true
4    C = getNextCommand()
5    while (C ≠ stop)
6      match (C):
7        case (v := e):
8          M = M + [&v ↦ evalConcrete(e)]
9          S = S + [&v ↦ evalSymbolic(e)]
10       case (if e then C' else C''):
11         b = evalConcrete(e)
12         c = evalSymbolic(e)
13         if b then pc = pc ∧ c
14         else pc = pc ∧ ¬c
15     C = getNextCommand() // end of while loop
```

Figure 2. Symbolic execution.

address at which the value of program variable v is stored. To simplify the presentation, we assume that all program variables have some unique initial concrete value in the initial concrete store M_0, and that the initial symbolic store S_0 identifies the program variables v whose values are program inputs (for all those, we have $S_0(v) = x_i$ where x_i is the symbolic variable corresponding to the input parameter I_i). Initially, pc is defined to `true`. By construction, all symbolic variables appearing in pc are variables x_i corresponding to program inputs I_i.

Systematic dynamic test generation [15] consists of systematically exploring all (or in practice many) feasible control-flow paths of the program under test by using path constraints and a constraint solver. Given a program state $s = \langle C, M, S, pc\rangle$ and a constraint solver for theory $\mathcal{T}$, if C is a conditional statement of the form `if` e `then` C' `else` C'', any satisfying assignment of the formula $pc \wedge c$ (respectively $pc \wedge \neg c$) where $c =$ evalSymbolic(e) in state s, defines program inputs that will lead the program to execute the `then` (resp. `else`) branch of the conditional statement. By systematically repeating this process, such a *directed search* can enumerate (in theory) all possible path constraints and eventually execute all feasible program paths.

The search is exhaustive provided that the generation of the path constraint (including the underlying symbolic execution) and the constraint solver for the given theory $\mathcal{T}$ are both *sound and complete*, that is, for all program paths w, the constraint solver returns a satisfying assignment for the path constraint pc_w *if and only if* the path is feasible (i.e., there exists some input assignment leading to its execution). If those conditions hold, in addition to finding errors such as the reachability of bad program statements (like `assert(false)`), a directed search can also prove their absence, and therefore obtain a form of program *verification*.

THEOREM 1. *(adapted from [15]) Given a program P as defined above, a directed search using a path constraint generation and a constraint solver that are both sound and complete exercises all feasible program paths exactly once.*

Thus, if a program statement has not been executed when the search is over, this statement is not executable in any context. In practice, path constraint generation and constraint solving are usually not sound and complete.

Note that the above formalization and theorem do apply to programs containing loops or recursion, as long as all program executions terminate. However, in the presence of a single loop whose number of iterations depends on some unbounded input, the number of feasible program paths becomes infinite. In practice, search termination can always be forced by bounding input values, loop iterations or recursion, at the cost of potentially missing bugs.

3. Sound and Unsound Concretization

3.1 Concretization and Must Abstraction

When a program expression cannot be expressed in the given theory $\mathcal{T}$ decided by the constraint solver, it can be simplified using concrete values of sub-expressions, or replaced by the concrete value of the entire expression. This case corresponds to line 13 of Figure 1. Let us call *concretization* the process of replacing a symbolic expression by its current concrete value during dynamic symbolic execution.

In the presence of concretizations, path constraint generation is in general no longer "sound and complete" since constraints become approximate and path constraints no longer capture accurately program path feasibility. (In the original DART algorithm of [15], some completeness flag would then be set off and the outer loop in Figure 2 of [15] would run forever.) Moreover, Theorem 1 no longer holds since its assumptions are no longer satisfied.

Loosely speaking, concretizing a symbolic expression under-approximates its set of possible values by a singleton set containing its unique current runtime value. In that sense, concretization can be viewed as a *"must abstraction"* which is *sound for bug-finding*. Must abstractions capture existential reachability properties that hold on *some* but not all program executions.

A sound path constraint (see Definition 1) is an example of must abstraction [17]. Note that, if a sound path constraint pc_w is satisfiable, *then* the corresponding program path w is feasible. But the converse does not necessarily hold: an algorithm for generating sound path constraints may fail to generate path constraints for some feasible program paths, and hence may fail to exercise some code and may miss bugs.

3.2 Unsound Concretization

Strictly speaking, however, concretization alone does not guarantee a sound path constraint generation. Consider the following program example.

```
int foo(int x, int y) {
  if (x == hash(y)) {
    ...
    if (y == 10) return -1; // error
  }
  ...
}
```

Assume that the function `hash` is "unknown", that the program is run with the input values `x=567` and `y=42`, that `hash(42)` is 567, and hence that the execution takes the `then` branch of the first conditional statement. The path constraint generated by the DART algorithm of Figures 1 and 2 (i.e., *without* line 14 of Figure 1) is

$$x = 567 \wedge y \neq 10$$

Indeed, the expression `hash(y)` which (we assume) is outside $\mathcal{T}$ is replaced by its concrete value 567 by line 15 of Figure 1. But the algorithm does not "record" this concretization at the first conditional statement, and allows a symbolic constraint $y \neq 10$ to be generated on `y` later on. This path constraint correctly captures the current concrete execution (since x is indeed 567 and y is indeed different from 10 for this run), but it is *not sound*: for x equal to 567 and some value of y different from 10, the input assignment satisfies the path constraint but does not define a program execution following the same execution path if `hash(y)` is not 567.

By negating the last constraint of this unsound path constraint and solving the new path constraint

$$x = 567 \wedge y = 10$$

one gets a new test input that should drive the program towards the error, but results instead in a *divergence* [15], i.e., an unexpected program path being taken if `hash(10)` is different from 567.

The risk of divergences in the presence of unsound path constraints is not a new observation: it is discussed in [15] and motivates the need for comparing the actual path taken by the program under test with the expected path w derived from each path constraint pc_w. When additional constraints are automatically injected in path constraints for checking additional program properties such as the absence of buffer overflows, every new test input generated violating such injected constraints should be executed to confirm the bug before reporting it to the user, in order to avoid reporting false alarms due to divergences from unsound path constraints.

3.3 Sound Concretization

To generate sound path constraints, we propose the following new variant of the DART algorithm: whenever a symbolic expression e is concretized during symbolic execution, for all symbolic variables x_i occurring in e, a new *concretization constraint* $x_i = I_i$ is added to the path constraint, as illustrated in line 14 of Figure 1, which we now assume is un-commented. This implies that the value of each such symbolic variable x_i is fixed to a constant equal to the corresponding current input value I_i *below* in the path constraint. Let us call this procedure *sound concretization*.

Indeed, we now show that sound concretization results in sound path constraints.

THEOREM 2. *The algorithm of Figures 1 and 2* with *sound concretization, i.e., including line 14 of Figure 1, generates sound path constraints.*

Proof: The proof relies on the assumption that all sources of imprecision in symbolic execution are detected and trigger the `default` case in the procedure `evalSymbolic` shown in Figure 1. In every such case, line 14 is executed and a concretization constraint is injected in the path constraint. Otherwise, in all other cases, symbolic execution of individual instructions (assignments or conditional statements) is assumed to be precise, i.e., both sound and complete.

Consider a path constraint pc_w generated by this algorithm during the execution of a program path w with an input vector $I = \langle I_i | \forall i \rangle$. For every symbolic variable x_i (generalizing program input I_i) occurring in pc_w, two cases are possible. Either there is a concretization constraint forcing x_i to be equal to I_i. Or all constraints on x_i in pc_w are both sound and complete (symbolic execution is precise so far for all individual instructions involving x_i). Either way, all values of x_i satisfying pc_w satisfy all the tests on inputs along w and hence lead to a program execution following the same path w. Since the same argument holds for all symbolic variables x_i, pc_w is sound. ■

Unlike ordinary constraints derived from conditional statements executed by the program under test, concretization constraints should not be negated later in the directed search, because negating these constraints will not define alternate path constraints corresponding to new program paths. The only purpose of concretization constraints is to guarantee soundness of path constraints.

EXAMPLE 1. Consider again the example of function `foo` shown in Section 3.2. Assume again we run with program with inputs `x=567` and `y=42`, and that `hash(42)` is 567. With sound concretization, a concretization constraint $y = 42$ is generated when symbolically evaluating `hash(y)` in the first conditional statement, and dynamic symbolic execution generates the sound path constraint

$$y = 42 \wedge x = 567 \wedge y \neq 10$$

After negating the last constraint, the resulting constraint

$$y = 42 \wedge x = 567 \wedge y = 10$$

is not satisfiable, and no new test is generated to try to cover the `then` branch of the second conditional statement. ■

Sound concretization generates sound path constraints and eliminates divergences. But in practice, sound concretization is not necessarily "better" than DART's default unsound concretization, for two reasons.

First, a drawback of sound concretization is that it reduces the ability to generate new tests.

EXAMPLE 2. Consider the following program using the same `hash` function:

```
int foo-bis(int x, int y) {
  if (x != hash(y)) {
    ...
    if (y == 10) return -1; // error
  }
  ...
}
```

Assume the program is run with inputs `x=33` and `y=42`, and that `hash(42)` is 567. Sound concretization generates the sound path constraint

$$y = 42 \wedge x \neq 567 \wedge y \neq 10$$

After negating the last constraint, the resulting constraint

$$y = 42 \wedge x \neq 567 \wedge y = 10$$

is not satisfiable, no new test is generated to try to cover the `then` branch of the second conditional statement, and the error is missed. In contrast, unsound concretization would generate the path constraint

$$x \neq 567 \wedge y \neq 10$$

After negating the last constraint, a constraint solver would easily solve the (unsound) path constraint

$$x \neq 567 \wedge y = 10$$

and generate a new test that is *likely* (but not guaranteed) to hit the error, assuming `hash(10)` is likely different from the value of `x` whatever its value is. This is an example of a *"good divergence"*. ■

Second, and perhaps most importantly, sound concretization is *much harder to implement* than unsound concretization, since it requires detecting explicitly *all* sources of imprecision in symbolic execution — including conservatively estimating all possible inputs and outputs of all individual instructions and all unknown/library/operating-system functions used by the program under test —, while unsound concretization can simply be implemented by handling some program instructions and ignoring the others.

Finally, note that adding line 14 of Figure 1 is just one way to inject concretization constraints and that other variants are possible. For instance, the injection of concretization constraints for symbolic variables x_i occurring in a concretized expression e could be delayed during symbolic execution until e is actually being used in some constraint (if any) in the path constraint pc_w. This way, examples such as

```
...
x := hash(y);
if (y == 10) return -1; // error
...
```

could be handled with sound concretization by postponing injecting a concretization constraint for `y` from when `hash(y)` is computed to when program variable `x` is being tested (if at all), and a constraint to cover the other branch of the test `(y == 10)` could be generated and solved.

4. Higher-Order test Generation

We now introduce a more general form of test generation, which we call *higher-order* because it uses a higher-order logic representation of path constraints. Higher-order test generation requires three steps:

1. uninterpreted functions are used to represent unknown functions or instructions during symbolic execution;
2. new test inputs are derived from validity proofs of first-order logic formulas with uninterpreted functions;
3. concrete input-output value pairs need be recorded as uninterpreted function samples that are used when generating new concrete test inputs.

We now discuss these three steps in detail one by one.

4.1 Symbolic Execution with Uninterpreted Functions

Another well-known approach for reasoning about unknown functions is to represent those using *uninterpreted functions*. Figure 3 presents a more general algorithm for dynamic symbolic execution where unknown functions or instructions are represented *explicitly* using uninterpreted function symbols. This algorithm extends the standard symbolic execution procedure of Figure 2 with the new lines marked with `*`. Whenever an unknown function or instruction `f` is encountered during symbolic execution (line 10 of Figure 3), an uninterpreted function symbol f uniquely representing the function/instruction is used to represent the symbolic return value of the function call, which is defined as the application of the function to its symbolic input arguments (line 12). Symbolic execution resumes after the function returns.

By unknown function, we mean any function whose code is not available or not precisely representable by a symbolic expression of the theory $\mathcal{T}$ handled by the constraint solver for whatever reason (such as hash or crypto functions, operating-system functions, environment/library functions outside of the main scope of analysis, etc.). Similarly, by unknown instruction, we mean any atomic program instruction not handled by the symbolic evaluation procedure, i.e., involving some symbolic expression previously concretized in line 13 of Figure 1. For simplicity, we represent such unknown instructions by uninterpreted functions as well; line 13 of Figure 1 is thus no longer reachable, by construction, with the new algorithm.[3]

In Figure 3, `args` denotes a list of arguments. Each argument is a variable v whose value is an input to the function call (we consider a call-by-value function model here, for simplicity).

[3] Any symbolic expression e including an unknown function/instruction application $f(args)$ as sub-expression is equivalent to $v_{f(args)} := f(args)$ followed by e where $f(args)$ is replaced by $v_{f(args)}$.

Consider again the example of function `foo` shown in Section 3.2. When symbolically executing the first conditional statement `(x == hash(y))`, a fresh uninterpreted function symbol h is introduced to represent the unknown function `hash`. If the `then` branch of the first conditional statement is taken, the path constraint generated is then

$$x = h(y)$$

The new symbolic execution performed by the algorithm of Figure 3 typically generates more symbolic values than the standard symbolic execution procedure of Figure 2, since it represents unknown functions with "symbolic" uninterpreted functions instead of using concretization and falling back on concrete values. Therefore, the new algorithm typically generates more symbolic constraints in the path constraint pc (lines 17 and 18). We can prove that those path constraints are always sound.

THEOREM 3. *The algorithm of Figure 3 generates sound path constraints.*

Proof: The proof relies on the assumption that all sources of imprecision in symbolic execution are detected in line 10 of the procedure `executeSymbolic` in Figure 3 and are representable by uninterpreted functions (line 12), which implies that every unknown function/instruction is deterministic and with a known input-output signature. For all other cases, symbolic execution of individual instructions (assignments or conditional statements) is assumed to be precise, i.e., both sound and complete.

Consider a path constraint pc_w generated by this algorithm during the execution of a program path w with an input vector $I = \langle I_i | \forall i \rangle$. At any time during the symbolic execution along w, all direct dependencies on inputs are tracked precisely via the symbolic store, either in sound and complete manner via the procedure `evalSymbolic` of Figure 1, or using uninterpreted function applications. Therefore, at every conditional statement C executed along w, if a constraint c involving some symbolic variable x_i is added in C to the path constraint pc_w, all input values of x_i satisfying c take the same branch as the current concrete value I_i of x_i. Since the same argument holds for all symbolic variables x_i and all symbolic constraints in pc_w, every input assignment I' satisfying the path constraint pc_w defines a program execution following the given path w, which means that pc_w is sound. ■

The previous theorem states that, *if* an input assignment satisfies a path constraint pc_w generated by the algorithm of Figure 3, *then* the corresponding program execution will follow path w. For instance, considering again our running example, any input assignment to the variables x and y that satisfies the path constraint $x = h(y)$ will take the `then` branch of the conditional statement in function `foo`. But this result alone does not prescribe *how* to compute such values and generate tests from path constraints with uninterpreted function symbols. This problem is discussed next.

4.2 Generating Tests from Validity Proofs

When uninterpreted functions are used in path constraints to model imprecision in symbolic execution, test generation from such path constraints *must* be performed from *validity proofs*, instead of satisfiability proofs, as will be shown shortly. This requires path constraints to be *post-processed* before calling the validity checker: every ordinary symbolic variable representing a program input is *existentially quantified*, resulting in a first-order logic formula of the form

$$\exists x : \phi(f, x)$$

while every uninterpreted function symbol f is implicitly left *universally quantified* in the validity check. Remember that first-order logic does not allow explicit quantification over functions: universal function quantification is implicit when checking validity, while

```
1   Procedure executeSymbolic(P,I) =
2     initialize M_0 and S_0
3     path constraint pc = true
4     C = getNextCommand()
5     while (C ≠ stop)
6       match (C):
7         case (v := e):
8           M = M + [&v ↦ evalConcrete(e)]
9           S = S + [&v ↦ evalSymbolic(e)]
10  *     case (v := f(args)): // f is an unknown function or instruction
11  *       M = M + [&v ↦ evalConcrete(f(args))]
12  *       S = S + [&v ↦ createExpression('f',evalSymbolic(args))]
13  *       Add (M(&v),createExpression('f',evalConcrete(args))) to IO_F;
14        case (if e then C' else C''):
15          b = evalConcrete(e)
16          c = evalSymbolic(e)
17          if b then pc = pc ∧ c
18          else pc = pc ∧ ¬c
19      C = getNextCommand() // end of while loop
```

Figure 3. Symbolic execution with uninterpreted functions.

existential function quantification is implicit when checking satisfiability.[4]

We emphasize that representing unknown functions by uninterpreted function symbols in validity queries is *not* new in program verification. Indeed, for *verification*, the set of possible behaviors of unknown functions needs to be *over-approximated* to guarantee a *may* abstraction that can be used to prove *correctness*, hence the use of (implicit) *universal quantification*. However, what is new here (to the best of our knowledge) is our use of (implicit) *universal quantification* for uninterpreted function symbols for *test generation*, instead of (implicit) *existential quantification* with satisfiability queries as usual in the context of test input generation. We discuss this further in Section 8.

We now illustrate this important difference, and why we need it. Consider again the example of the `obscure` function used in the introduction:

```
int obscure(int x, int y) {
  if (x == hash(y)) return -1;   // error
  return 0;                      // ok
}
```

Let us assume again that the function `hash` is "unknown", that the program is run first with the input values `x=33` and `y=42`, and that `hash(42)` is 567 and hence that the first execution takes the `else` branch of the conditional statement. With the standard symbolic execution of Figure 2, the single constraint appearing in the path constraint pc is

$$x \neq 567$$

Next, the *satisfiability* of the negation of this constraint, namely

$$x = 567$$

is checked by the constraint solver. Since it is satisfiable, the satisfying assignment returned by the constraint solver is transformed into a new input vector, namely `x=567` and `y=42`, that will drive the next execution of the program along the `then` branch of the conditional statement. Note how input variable x is existentially quantified in the *satisfiability check* performed by the constraint solver.

In contrast, with the new symbolic execution procedure of Figure 3, the single constraint appearing in the path constraint is now

$$x \neq h(y)$$

where h denotes the uninterpreted function symbol representing the unknown function `hash`. After post-processing, the *validity* of the formula

$$\exists x, y : x = h(y)$$

is checked by the constraint solver (i.e., for all h). If the formula is valid, a *test-generation strategy* is derived from the *validity proof* of the formula, viewed as a strategy for making the formula always true. In this case, the formula is valid and the strategy is

"fix y, then set x to the value $h(y)$".

In other words, with the new algorithm, new tests are derived from *validity proofs*, instead of *satisfying assignments* as usual.

Indeed, we have no choice: if uninterpreted functions are used to model imprecision in symbolic execution, test generation can no longer be performed from satisfiability checks. In the above example, checking the satisfiability of the formula

$$x = h(y)$$

(where h, x and y are thus all implicitly quantified existentially) may return satisfying assignments that are unusable for test generation since the existential quantifier over h allows the constraint solver to *invent* some specific arbitrary function h that helps it prove satisfiability. For instance, the constraint solver may define function h such that $h(0) = 0$ and then return `x=0` and `y=0` as satisfying assignments. Since such a function h may differ from the specific unknown function `hash` called in the program under test, sound test generation is not possible when existentially quantifying h.

4.3 The Need for Uninterpreted Function Samples

Test strategies derived from validity proofs are *necessary but not sufficient* to compute concrete input vectors, which is required in test generation. For instance, with the above test strategy, the "value $h(y)$" is not derived from the validity proof: a concrete value for x can be obtained only when some values for y and $h(y)$ are known. In general, the value of $h(y)$ for a given y can only be known at runtime. (Unless the function h is completely known and not too complex to be represented as a logic formula; then a constraint solver using constant propagation starting from some concrete input value y could simulate the execution of h with value y as argument and compute the value $h(y)$; however, even in this case, constant propagation is orders of magnitude slower and less scalable than simply running the actual code implementing the function h with value y as argument.)

[4] A logic formula is *satisfiable* if there exists a variable assignment that makes the formula true, and it is *valid* if all variable assignments make the formula true.

In the above example, we need to know that if y is set to 42, then $h(y)$ is 567 in order to set x to that value following the test-generation strategy derived from the validity proof.

In other words, the new symbolic execution algorithm using uninterpreted functions also needs to *record runtime concrete values* to allow for test generation of specific concrete input values. Let us call this step *uninterpreted function sampling*.

Specifically, we can record the concrete value of any function application such as $h(y)$ during dynamic symbolic execution as well as the concrete value of each of its arguments, as shown in line 13 of Figure 3: a pair $(c, f(\text{evalConcrete}(args)))$ is recorded for each function application where evalConcrete$(args)$ denotes the list of concrete values of each function argument and c denotes the concrete return value of the function applied to those concrete argument values. In the above example, the record pair is thus $(567, h(42))$, meaning that $567 = h(42)$.

These pairs $(c, f(\text{evalConcrete}(args)))$ of recorded values have two purposes.

- They are used to *interpret* a test generation strategy derived from a validity proof in order to assign concrete values to function applications (such as $h(y)$) appearing in the strategy and generate concrete values to new input tests.
- They can also be used to generate additional constraints of the form $c = f(\text{evalConcrete}(args))$ as an *antecedent* to the path constraint that is passed to the constraint solver. Such constraints restrict the possible interpretations for uninterpreted function symbol f, and increases the chance of validity.

The latter can be more powerful and is necessary for higher-order test generation to always subsume sound concretization, as will be shown in the next section. Therefore, we adopt this second option in what follows.

To sum up, given a path constraint pc using a set X of symbolic variables, a set F of uninterpreted functions and a set IO_F of recorded input-output function samples, the post-processed formula $POST(pc)$ obtained by post-processing pc in high-order test generation is the first-order logic formula defined by

$$POST(pc) = \exists X : A \Rightarrow pc$$

where $\exists X$ denotes that all symbolic variables $x_i \in X$ are existentially quantified, $\Rightarrow$ denotes logical implication, and A is the conjunction of equality constraints $c = f(\text{evalConcrete}(args))$ for all $(c, f(\text{evalConcrete}(args))) \in IO_F$. In what follows, we call A the *antecedent* of $POST(pc)$.

For instance, consider again our running example with a path constraint pc containing a single constraint $x = h(y)$ and a single recorded pair $(567, h(42))$. Then $POST(pc)$ is

$$\exists x, y : (567 = h(42)) \Rightarrow (x = h(y))$$

Sometimes, the antecedent does not help the validity proof itself, as in the previous example, and only helps for generating actual concrete test values. But sometimes, the antecedent is necessary to prove validity, as shown in Section 5.3.

In practice, recording *all* concrete arguments and return values of *all* uninterpreted function applications used in a path constraint can be prohibitively expensive for long program executions. Moreover, it is unfortunately hard to predict which concrete values will be needed later in the path constraint, i.e., which concrete values are concretizations of symbolic expressions with uninterpreted functions on which there are tests *below* in the path constraint. However, it is possible to track only *some* sources of imprecision and only represent those using uninterpreted functions, and to track and represent only *some* input-output pairs for tracked functions. Implementability issues will be discussed further in Section 6.

5. Comparison

In this section, we compare the test-generation power of higher-order test generation (Section 4) with sound and unsound concretization (Section 3).

5.1 Higher-Order Test Generation and Unsound Concretization are Incomparable

As discussed at the end of Section 3.3, sound and unsound concretization are incomparable in general, since unsound concretization can lead to (bad or good) divergences that will not occur with sound concretization. Similarly, by Theorem 3, higher-order test generation generates sound path constraints and is thus incomparable to unsound concretization in general, for the same reasons.

EXAMPLE 3. Consider the function

```
int bar(int x, int y) {  // x,y are inputs
  if ((x == hash(y)) AND (y == hash(x))) {
    ... // error
  }
 . ..
}
```

Given random inputs $x = 33$ and $y = 42$ and assuming `hash(42)=567` and `hash(33)=123`, unsound concretization will generate an unsound path constraint

$$x \neq 567 \vee y \neq 123$$

whose negation $x = 567 \wedge y = 123$ is satisfiable and generates a new test input pair $\langle x = 567, y = 123 \rangle$ which will likely lead to a divergence. In contrast, higher-order test generation will generate a sound path constraint

$$x \neq h(y) \vee y \neq h(x)$$

After post-processing, the validity of the formula $\exists x, y : x = h(y) \wedge y = h(x)$ will be checked. But no new test will be generated since this formula is invalid (in general, unless we learn some additional property of h such as there exists an x such as $x = h(h(x))$, for instance). ■

5.2 Higher-Order Test Generation is as Powerful as Sound Concretization

In the remainder of this section, we will therefore restrict the comparison of higher-order test generation to sound concretization. Both algorithms generate sound path constraints (see Theorems 2 and 3). We now show that high-order test generation is at least as powerful as test generation with sound concretization.

Given a path constraint $pc = \bigwedge_{1 \leq i \leq n} c_i$, let $ALT(pc)$ denote the new *alternate* path constraint defined by the conjunction of the negation of the last constraint c_n of pc with all previous constraints c_j with $j < i$ in pc. We thus have

$$ALT(pc) = \neg c_n \wedge \bigwedge_{1 \leq i \leq (n-1)} c_i$$

Remember that, as explained in Section 2, all nonempty prefixes of a path constraint are also path constraints (except those ending with a concretization constraint).

Let pc_w^{SC} denote a path constraint generated for a program path w with sound concretization (Section 3.3) and whose last constraint is not a concretization constraint. Let pc_w^{UF} be the path constraint generated with higher-order test generation (Section 4.1) for the same program path w. Given any theory $\mathcal{T}$, let $\mathcal{T} \cup \mathcal{T}_{EUF}$ denote the theory combining $\mathcal{T}$ with the theory of equality with uninterpreted functions (EUF). If pc_w^{SC} and $ALT(pc_w^{SC})$ are quantifier-free formulas over $\mathcal{T}$, then pc_w^{UF} and $ALT(pc_w^{UF})$ are quantifier-

free formulas over $\mathcal{T} \cup \mathcal{T}_{EUF}$, while $POST(ALT(pc_w^{UF}))$ is a first-order logic formula over $\mathcal{T} \cup \mathcal{T}_{EUF}$, by construction.

THEOREM 4. *(Simulation Theorem)*
If $ALT(pc_w^{SC})$ is satisfiable, then $POST(ALT(pc_w^{UF}))$ is valid.

Proof:
We show how to derive a validity proof for $POST(ALT(pc_w^{UF}))$ from any satisfying assignment for $ALT(pc_w^{SC})$.

Whenever a complex/unknown expression $e(x_i) \notin \mathcal{T}$ occurs during symbolic execution with sound concretization, a concretization constraint $x_i = I_i$ is introduced in pc_w^{SC}, $e(x_i)$ becomes $e(I_i)$, and all future expressions $e'(x_i)$ depending on x_i become $e'(I_i)$. In contrast, in higher-order test generation, every occurrence of a complex/unknown expression $e(x_i) \notin \mathcal{T}$ becomes $f_e(x_i)$ where f_e is an uninterpreted function symbol representing e, and the pair (evalConcrete$(e(I_i))$, $f_e(I_i)$) is being recorded.

Consider any symbolic variable $x_i \in X$ for which there is a concretization constraint $x_i = I_i$ in $ALT(pc_w^{SC})$. Consider any expression $e(x_i) \notin \mathcal{T}$ concretized into $e(I_i)$ and occurring in $ALT(pc_w^{SC})$. In $POST(ALT(pc_w^{UF}))$, $e(x_i)$ is represented by $f_e(x_i)$ and evalConcrete$(e(I_i)) = f_e(I_i)$ is in the antecedent.

In $POST(ALT(pc_w^{UF}))$, repeat the following process for all the symbolic variables x_i with a concretization constraint in $ALT(pc_w^{SC})$ and for all the functions f_e using those variables as arguments: substitute all the occurrences of x_i by I_i and then all the occurrences of any function $f_e(I_i)$ by evalConcrete$(e(I_i))$. (Note that this last step would not be possible if evalConcrete$(e(I_i)) = f_e(I_i)$ was not present, i.e., known and recorded, in the antecedent of POST(ALT(pc^{UF})).)

At the end, we are left with a formula $\phi(X')$ where X' denotes the set of all remaining symbolic variables $x_i' \in X$ for which there are no concretization constraints in $ALT(pc_w^{SC})$. Therefore, we know that all expressions of the form $e(x_i')$ in $\phi(X')$ are in $\mathcal{T}$, that they did not introduce imprecision in symbolic execution, and that they are represented in the exact same way in $ALT(pc_w^{SC})$.

Thus, by construction, the consequent of $\phi(X')$ is syntactically equivalent to $ALT(pc_w^{SC})$ when all its concretization constraints are removed. Since all occurrences of all uninterpreted function symbols f_e have been eliminated from the consequent of $\phi(X')$, the universal quantification over those functions in $\phi(X')$ becomes void intuitively. The same holds for the existential quantification for all symbolic variables $x_i \notin X'$ that no longer appear in $\phi(X')$. Since the consequent of $\phi(X')$ is logically equivalent to $ALT(pc_w^{SC})$, if the latter is satisfiable, then $\exists X' : A \Rightarrow ALT(pc_w^{SC})$ is valid. This implies that $\exists X : A \Rightarrow ALT(pc_w^{UF})$ is valid (by setting the value of each variable $x_i \in X \setminus X'$ to I_i). ■

We emphasize that the previous theorem holds *only if* uninterpreted function samples are used. Otherwise, higher-order test generation may not be able to simulate sound concretization, as illustrated by the following example.

EXAMPLE 4. Consider the function

```
int pub(int x, int y) {  // x,y are inputs
  if ((hash(x) > 0) AND (y == 10)) return -1 // error
  ...
}
```

Given random inputs $x = 1$ and $y = 2$ and assuming `hash(1)=5`, sound concretization will generate a sound path constraint

$$x = 1 \wedge y \neq 10$$

(after simplifying $5 > 0$ to `true`). The alternate path constraint $x = 1 \wedge y = 10$ is satisfiable and generates a new test input pair $\langle x = 1, y = 10 \rangle$ to cover the `then` branch of the conditional statement. In contrast, higher-order test generation *without* uninterpreted function samples will generate a sound path constraint

$$h(x) > 0 \wedge y \neq 10$$

However, after post-processing of the alternate path constraint attempting to cover the `then` branch, the validity of the formula

$$\exists x, y : h(x) > 0 \wedge y = 10$$

will be checked. But no new test will be generated since this formula is invalid (to see this, consider the function h such that $h(x) = 0$ for all x, for instance). If instead we consider higher-order test generation *with* uninterpreted function samples, we then obtain after post-processing the formula

$$\exists x, y : (h(1) = 5) \Rightarrow (h(x) > 0 \wedge y = 10)$$

which is valid (by setting $\langle x = 1, y = 10 \rangle$). ■

An important remark is that Theorem 4 only compares the path-constraint generation capabilities of higher-order test generation and sound concretization. But it does *not* state that if there exists a constraint solver that can prove the satisfiability of $ALT(pc_w^{SC})$, then there exists a constraint solver that can prove the validity of $POST(ALT(pc_w^{UF}))$. Thus, when we say that "higher-order test generation is as powerful as sound concretization", we assume we are given *perfect* constraint solvers for both satisfiability and validity checking.

5.3 Higher-Order Test Generation is More Powerful Than Sound Concretization

The previous theorem states that higher-order test generation is at least as powerful as sound concretization. Is it more powerful? The answer is yes, for three reasons.

First, since higher-order test generation uses $\mathcal{T} \cup \mathcal{T}_{EUF}$, it can infer test strategies thanks to axioms included in the theory of equality with uninterpreted functions (EUF), which are not available to sound concretization, which only uses $\mathcal{T}$.

EXAMPLE 5. Higher-order test generation can generate tests from validity proofs of post-processed path constraints such as

$$\exists x, y : f(x) = f(y)$$

thanks to the theory of equality with uninterpreted functions. (Solution strategy: set $x = y$). In contrast, sound concretization would force the concretization of $x, y, f(x)$ and $f(y)$, and would not be able to generate a test to cover a path with such a path constraint. ■

Second, higher-order test generation can sometimes leverage concrete input-output pairs that are part of the antecedent of a post-processed path constraint in order to prove the validity of formulas that would otherwise be invalid.

EXAMPLE 6. Consider the post-processed path constraint

$$\exists x, y : f(x) = f(y) + 1$$

This formula is in general invalid (to see this, consider a function f that always returns 0). However, assume that it is dynamically observed that $f(0) = 0$ and $f(1) = 1$. Then these recorded pairs can be part of the antecedent of the post-processed path constraint, which becomes

$$\exists x, y : (f(0) = 0 \wedge f(1) = 1) \Rightarrow f(x) = f(y) + 1$$

This formula is valid (solution strategy: set $x = 1$ and $y = 0$). In either case, sound concretization would force the concretization of $x, y, f(x)$ and $f(y)$ and would not be able to generate new tests. ■

Third, higher-order test generation can sometimes generate test strategies that involves a *sequence of new tests*, whose purpose is to

collect *additional function samples* in a targeted manner, instead of a single new test as usual. Let us call this new type of test generation *multi-step test generation*.

EXAMPLE 7. Consider again the example of function `foo` of Section 3.2, reproduced here for convenience:

```
int foo(int x, int y) {
  if (x == hash(y)) {
    ...
    if (y == 10) return -1; // error
  }
  ...
}
```

Starting with $x = 33$ and $y = 42$ and assuming `hash(42)=567`, this first test takes the else branch of the first conditional statement. After negating the last constraint, we obtain the post-processed alternate path constraint

$$\exists x, y : (h(42) = 567) \Rightarrow x = h(y)$$

This formula is valid and we generate a new input vector $\langle x = 567, y = 42\rangle$. We run this new test and we now take the then branch of the first conditional statement followed by the else branch of the second conditional statement. After negating the last constraint, we obtain the post-processed alternate path constraint

$$\exists x, y : (h(42) = 567) \Rightarrow (x = h(y) \wedge y = 10)$$

This formula is valid, and a test strategy derived from the validity proof is "set $y = 10$, set $x = h(10)$". However, since the value of $h(10)$ has never been sampled, it is currently unknown!

A new *intermediate* test with, say, $\langle x = 567, y = 10\rangle$ is necessary to learn the value of $h(10)$, say 66. Only then can a second input vector $\langle x = 66, y = 10\rangle$ be generated to finish interpreting the previous test strategy, to exercise the `then` branch of the second conditional statement and hit the `error`. ■

This is an example of two-step test generation. Of course, such examples can easily be generalized to k-step test generation for any k bounded by the number of program inputs.

Another related yet orthogonal suggestion would be to include in the antecedent of post-processed alternate path constraints generated with higher-order test generation, not only all the input-output value pairs observed for the current run, but also all value pairs observed during *all previous* runs.

6. Discussion: Implementability

As explained in the introduction, the main purpose of this paper is to carefully study the power of recent test generation techniques such as DART which are quickly gaining popularity. It is also to understand the fundamental difference between static and dynamic test generation. In the process, we proposed higher-order test generation as a powerful test generation technique generalizing sound concretization. How practical is higher-order test generation?

For large applications such as those targeted by whitebox fuzzing [16], exhaustively tracking *all* sources of imprecision during symbolic execution is problematic. Such imprecision can be due to unhandled individual instructions (for instance, the complete x86 instruction set contains hundreds of instructions described in a 1,000+ pages manual with exotic bit-manipulations, floating-point/SSE instructions, etc.), operating-system calls (should kernel execution be symbolic and if so up to what depth?), complex functions (for hashing, encrypting, compressing, encoding, CRC-ing data), etc. For instance, a single symbolic execution of Excel with 45K input bytes executes nearly a billion x86 instructions (see Figure 6 of [16]), including many input-tainted unhandled ones and many system calls. Moreover, some of this imprecision is hard to capture using uninterpreted functions because the real functions may look nondeterministic and/or with complex or unknown input-output signatures (such as malloc, rand, fork, etc. which take as inputs large/unknown parts of the operating-system state and may have many hidden side effects). Finally, capturing at execution time *all* observed input-output value pairs is problematic as well. For large applications, all this would slow down an already slow symbolic execution and generate gigantic path constraints that would overwhelm even the best engineered constraint solvers.

Therefore, we envision a more *focused role* for higher-order test generation in practice, targeted at reasoning about specific user-identified complex or unknown functions that must be dealt with in order to properly test an application. One such application is presented in the next section.

Another obstacle to the implementability of higher-order test generation is the relative lack of support for generating validity proofs by existing constraint solvers such as SMT solvers. Indeed, a first-order logic formula $\exists X : \phi(F, X)$ can be proved valid by checking whether its negation $\forall X : \neg\phi(F, X)$ is unsatisfiable with a *Satisfiability*-Modulo-Theories solver. For first-order logic formulas like those considered here, validity (equivalently unsatisfiability) is usually proved using saturation techniques [3]. Better tool support for generating saturation-based proofs (not just models for satisfiable instances) that are parsable by other tools would help extracting a test generation strategy from such proofs. More work is needed in this area. In fact, our paper can be viewed as a "requirement specification" for next-generation SMT solvers for test generation, a growing application area for those, by presenting higher-order test generation as a new possible application for those tools.

7. Application

In this section, we present an application which requires the power of high-order test generation: test generation for parsers with input lexers using hash functions for fast keyword recognition.

As observed in [14], dynamic test generation can be ineffective when testing applications with highly-structured inputs. Examples of such applications are compilers and interpreters. These applications process their inputs in stages, such as lexing, parsing and evaluation. Unfortunately, lexers often detect language keywords by comparing their pre-computed hash values with the hash values of strings read from the input. This effectively prevents symbolic execution and constraint solving from ever generating input strings that match those keywords since hash functions cannot be inversed (i.e., given a constraint `x == hash(y)` and a value for `x`, one cannot compute a value for `y` that satisfies this constraint). In those cases, test generation is defeated already in the first processing stages.

A typical code pattern is shown in Figure 4 in the appendix. This C code is an excerpt from the open-source flex lexer. Initially, the function `addsym` is called with every input-language keyword so that each of those are hashed (with function `hashfunct`) and stored in a hash table `table`. Once the hash table is populated, the parsing of the input starts. The input is being divided into chunks delimited by blank-spaces/tabs/etc. Each of those chunks are then parsed and the function `findsym` is called to check whether a chunk matches a keyword.

Because of the presence of function `hashfunct`, dynamic test generation may not be able to generate input strings ("chunks") matching specific keywords. In [14], it is shown how such a problematic lexer can be bypassed altogether for test generation of the subsequent input-processing stages by (1) instrumenting the lexer so that its return `symbol` values become symbolic inputs during symbolic execution, and (2) lifting the input space from character strings to sequences of symbols (token ids) using a grammar

specification of the input language being parsed. Unfortunately, instrumenting a lexer this way can be problematic for complex lexers, and this approach requires a user-supplied input-grammar specification.

In contrast, higher-order test generation provides a more automated approach to test generation through such lexers. The key but only thing the user is required to specify is the name of the function `hash` (like `hashfunct` or possibly a hash-function wrapper like `findsym` in the example of Figure 4) whose calls are then tracked during symbolic execution and represented using an uninterpreted function exactly as described in Section 4. During initialization, all the pairs ⟨`hashvalue`, `hash(keyword)`⟩ are being recorded to be included in the antecedent of post-processed path constraints. Whenever test generation needs a specific symbol to drive the parser through a new specific program branch, the theory of equality with uninterpreted functions combined with all the input-output value pairs recorded for `hash` makes it possible to effectively "inverse" this hash function for the finitely many keywords of the input language, which is sufficient for test generation for such applications. For instance, when an assignment statement of the form `symbol = hash(inputChunk)` is followed by a conditional statement of the form `if (symbol == 52) ...`, observing that `hash('while') = 52` is sufficient for higher-order test generation to generate an `inputChunk` equal to `while` in order to exercise the `then` branch of the conditional statement.

Note that, in some lexers, hash values are pre-computed and hard-coded in the source code. Then, it is not possible to observe at execution time all relevant input-output value pairs for such hash functions at the beginning of each execution. However, such input-output pairs could still be "learned" over time by starting the testing session with a representative set of well-formed inputs, observing the hash values of all the language keywords those inputs contain, and then using all pairs recorded in all previous executions in subsequent symbolic executions.

We have performed preliminary experiments with higher-order test generation in such a targeted manner in conjunction with the whitebox fuzzer SAGE [16] and using the Z3 SMT solver [9]. Since Z3 does not support the generation of saturation-based proofs, these experiments were conducted with an ad-hoc pre-processing step to eliminate uninterpreted functions in path constraints before calling Z3 for a satisfiability check as usual: (1) all uninterpreted function samples $h(c_1) = c_2$ are collected in a table IO_F (where c_1 and c_2 denote constants), and then (2) whenever a constraint of the form $h(x) = c_2$ occurs in a path constraint, it is replaced by a disjunction of constraints $x = c_1$ for all c_1 such that $h(c_1) = c_2$ (to handle hash collisions). This procedure is simple to implement but handles only limited cases and is far from simulating the full reasoning power of $\mathcal{T} \cup \mathcal{T}_{EUF}$. Nevertheless, experiments with a simple parser including a lexer similar to Figure 4 show that this partial implementation of higher-order test generation is sufficient to accurately drive program executions through the lexer. In contrast, regular dynamic test generation is no better than blackbox random testing because it is not able to drive executions through tests involving the hash function in the lexer.

8. Other Related Work

Abstracting program functions using uninterpreted functions is a well-known technique in verification-condition generation for program verification of universal properties (e.g., [4]). In that context, the set of all program behaviors is over-approximated in presence of program-analysis imprecision, and verification is established by checking the validity of a logic formula representing the entire program (or module). Uninterpreted function symbols in the formula are therefore implicitly universally quantified as in our work.

In the context of test generation, uninterpreted functions have been used for representing symbolic test summaries in compositional symbolic execution [1, 17]. There, a function summary is represented by a first-order logic formula using an uninterpreted function symbol representing the function. A function summary is defined as a disjunction (i.e., a set) of intraprocedural path constraints expressed in terms of the function inputs and outputs. Function summaries can be computed incrementally, to include more and more intraprocedural path constraints as they are discovered during a directed search. Test generation with function summaries is performed as usual by a satisfiability check, where all uninterpreted functions are therefore implicitly existentially quantified. This use of uninterpreted functions for function summaries is thus different from their use in higher-order test generation where uninterpreted functions represent imprecision in symbolic execution in *individual* path constraints. Both types of uninterpreted functions could actually be used simultaneously, as they are orthogonal, for "higher-order compositional test generation". Note that function summaries do not have to be represented using uninterpreted functions, and can be encoded directly in propositional logic instead [11].

We do not know of any other work where tests are derived from validity proofs, or where uninterpreted functions are used to model imprecision in symbolic execution *for test generation*.

The (implicit) alternation of universal function quantifiers and existential variable quantifiers in our post-processed path constraints can be viewed as a game between an unknown environment (controlling the unknown functions) and a test generator (controlling test inputs). To view test generation as a game is not new [5, 25]. Model-driven test generation for conformance testing, i.e., checking whether a blackbox implementation conforms to a whitebox specification, can be viewed as a game and, under specific assumptions, can be encoded logically using quantifier alternation, for instance using Quantified Boolean Formulas. However, we are not aware of any work on model-based test generation using uninterpreted functions or validity proofs of first-order logic formulas as in our work.

Given a program and a set of input parameters, test generation refers to the problem of generating input tests in order to exercise a specific program path, branch or statement. In contrast, static, dynamic and higher-order test generation denote specific approaches to solving the test generation problem. Test generation is only one way of proving existential reachability properties of programs, where specific concrete input values are generated to exercise specific program paths. More generally, such properties can be proved using so-called *must abstractions* of programs, without necessarily generating concrete tests. A must abstraction is defined as a program abstraction that preserves existential reachability properties of the program. For instance, in predicate abstraction [13, 17], a *must transition* from an abstract state A_1 to an abstract state A_2 implies that

$$\forall c_1 \in A_1 : \exists c_2 \in A_2 : c_1 \rightarrow c_2$$

that is, for every concrete state c_1 abstracted by A_1, there exists a program execution from c_1 to a concrete state c_2 such that c_2 is abstracted by A_2. Must transitions can be chained together to prove existential reachability properties of programs, i.e., *find bugs in a sound manner*. Sound path constraints are particular cases of must abstractions [17]. Note the analogy between the alternation of universal and existential quantifiers in the definition of must transition and in our post-processed path constraints.

Other related work includes [8, 18] where must abstractions are built backwards from error states using static program analysis. This approach can detect program locations and states provably leading to error states (no false alarms), but may fail to prove reachability of those error states back from whole-program initial

states, and hence may miss bugs or report unreachable error states. Imprecision in symbolic reasoning, e.g., due to system calls or unknown functions, is modeled by assigning nondeterministic values to all possible modified (output) variables, which is less precise than using uninterpreted functions yet still assumes all unknown functions have known sets of possible outputs/side-effects. Building must abstractions statically can be fast but requires symbolic reasoning about the whole program. On the other hand, dynamic test generation is slower but more precise by allowing symbolic execution to degrade gracefully using concrete runtime values whenever symbolic reasoning is difficult.

Static test generation can be extended to concretize symbolic values whenever static symbolic execution becomes imprecise [19]. This approach not only requires to detect all sources of imprecision, but also one call to the constraint solver for each concretization to ensure that every synthesized concrete value satisfies prior symbolic constraints along the current program path. For the reasons discussed in Section 6, such requirements are not practical for large applications. In contrast, dynamic test generation avoids these two limitations by leveraging a specific concrete execution as an automatic fall back for symbolic execution [15].

Dynamic test generation is currently an active area of research and many other extensions and applications have been proposed, such as [2, 6, 10, 23, 24] to name just a few (see [7] for a recent survey). This other related work does not specifically focus on how to deal with imprecision in symbolic execution and could benefit from the techniques introduced in our work.

9. Conclusion

We presented higher-order test generation, a powerful new form of test generation, which can also be expensive as it requires tracking explicitly sources of imprecision in symbolic execution, using uninterpreted functions, recording input-output function samples, and checking validity of first-order logic formulas. We showed how this approach can perform novel forms of test generation, such as multi-step test generation, and drive the executions of input parsers with lexers using hash functions for fast keyword recognition.

We also showed that the key property of dynamic test generation that makes it more powerful than static test generation is *only* its ability to observe concrete values and to record those in path constraints. In contrast, the process of simplifying complex symbolic expressions using concrete runtime values can be accurately simulated using uninterpreted functions. However, those concrete values are necessary to effectively compute new input vectors, a fundamental requirement in test generation.

Acknowledgments. I thank Leonardo de Moura for several insightful discussions related to this work. I also thank Nikolaj Bjorner, Yuri Gurevich and Mihalis Yannakakis for helpful comments, and Andreas Podelski and the anonymous reviewers for their constructive comments to improve the presentation.

References

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *Proceedings of TACAS'2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 367–381, Budapest, April 2008. Springer-Verlag.

[2] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. M. Paradkar, and M. D. Ernst. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Trans. Software Eng.*, 36(4):474–494, 2010.

[3] L. Bachmair and H. Ganzinger. Resolution Theorem Proving. In *Handbook of Automated Reasoning*, pages 19–99. 2001.

[4] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of FMCO'2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer-Verlag, September 2006.

[5] A. Blass, Y. Gurevich, L. Nachmanson, and M. Veanes. Play to Test. In *Proceedings of FATES'2005*, Edinburgh, July 2005.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *ACM CCS*, 2006.

[7] C. Cadar, P. Godefroid, S. Khurshid, C.S. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic Execution for Software Testing in Practice – Preliminary Assessment. In *Proceedings of ICSE'2011*, Honolulu, May 2011.

[8] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: A Powerful Approach to Weakest Preconditions. In *Proceedings of PLDI'2009*, Dublin, June 2009.

[9] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. In *Proceedings of TACAS'2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Budapest, April 2008. Springer-Verlag.

[10] M. Emmi, R. Majumdar, and K. Sen. Dynamic Test Input Generation for Database Applications. In *Proceedings of ISSTA'2007*, pages 151–162, 2007.

[11] P. Godefroid. Compositional Dynamic Test Generation. In *Proceedings of POPL'2007*, pages 47–54, Nice, January 2007.

[12] P. Godefroid. Software Model Checking Improving Security of a Billion Computers. In *Proceedings of SPIN'2009*, volume 5578 of *Lecture Notes in Computer Science*, page 1, Grenoble, June 2009. Springer-Verlag.

[13] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based Model Checking using Modal Transition Systems. In *Proceedings of CONCUR'2001*, volume 2154 of *Lecture Notes in Computer Science*, pages 426–440, Aalborg, August 2001. Springer-Verlag.

[14] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of PLDI'2008*, pages 206–215, Tucson, June 2008.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005*, pages 213–223, Chicago, June 2005.

[16] P. Godefroid, M.Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS'2008*, pages 151–166, San Diego, February 2008.

[17] P. Godefroid, A.V. Nori, S.K. Rajamani, and S.D. Tetali. Compositional May-Must Program Analysis: Unleashing The Power of Alternation. In *Proceedings of POPL'2010*, pages 43–55, Madrid, January 2010.

[18] J. Hoenicke, K. R. M. Leino, A. Podelski, M. Schaf, and Th. Wies. It's doomed; we can prove it. In *Proceedings of 2009 World Congress on Formal Methods*, 2009.

[19] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized Symbolic Execution for Model Checking and Testing. In *Proceeding of TACAS'2003*, April 2003.

[20] J. C. King. Symbolic Execution and Program Testing. *Journal of the ACM*, 19(7):385–394, 1976.

[21] B. Korel. A Dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, pages 311–317, San Diego, November 1990.

[22] D. Molnar, X. C. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proc. of the 18th Usenix Security Symposium*, Aug 2009.

[23] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *IEEE Symposium on Security and Privacy*, pages 513–528, 2010.

[24] N. Tillmann and J. de Halleux. Pex - White Box Test Generation for .NET. In *Proceedings of TAP'2008*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer-Verlag, April 2008.

[25] M. Yannakakis. Testing, Optimization, and Games. In *Proceedings of LICS'2004*, pages 78–88, Turku, July 2004.

```
/* addsym - add symbol and definitions to symbol table
 *
 * -1 is returned if the symbol already exists, and the change not made.
 */

static int addsym (sym, str_def, int_def, table, table_size)
     register char sym[];
     char   *str_def;
     int     int_def;
     hash_table table;
     int     table_size;
{
        int     hash_val = hashfunct (sym, table_size);
        register struct hash_entry *sym_entry = table[hash_val];
        register struct hash_entry *new_entry;
        register struct hash_entry *successor;

        while (sym_entry) {
                if (!strcmp (sym, sym_entry->name)) {   /* entry already exists */
                        return -1;
                }

                sym_entry = sym_entry->next;
        }

        /* create new entry */
        new_entry = (struct hash_entry *)
                flex_alloc (sizeof (struct hash_entry));

        if (new_entry == NULL)
                flexfatal (_("symbol table memory allocation failed"));

        if ((successor = table[hash_val]) != 0) {
                new_entry->next = successor;
                successor->prev = new_entry;
        }
        else
                new_entry->next = NULL;

        new_entry->prev = NULL;
        new_entry->name = sym;
        new_entry->str_val = str_def;
        new_entry->int_val = int_def;

        table[hash_val] = new_entry;

        return 0;
}

/* findsym - find symbol in symbol table */

static struct hash_entry *findsym (sym, table, table_size)
     register const char *sym;
     hash_table table;
     int     table_size;
{
        static struct hash_entry empty_entry = {
                (struct hash_entry *) 0, (struct hash_entry *) 0,
                (char *) 0, (char *) 0, 0,
        };
        register struct hash_entry *sym_entry = table[hashfunct (sym, table_size)];

        while (sym_entry) {
                if (!strcmp (sym, sym_entry->name))
                        return sym_entry;
                sym_entry = sym_entry->next;
        }

        return &empty_entry;
}
```

Figure 4. Code excerpt from the flex lexer (file sym.c, flex-2.5.35, February 2008).