Deconstructing Dynamic Symbolic Execution

Thomas BALL [??] and Jakub DANIEL [??]
[a] *Microsoft Research*
[b] *Charles University*

Abstract. Dynamic symbolic execution (DSE) is a well-known technique for automatically generating tests to achieve higher levels of coverage in a program. Two keys ideas of DSE are to: (1) seed symbolic execution by executing a program on an initial input; (2) use concrete values from the program execution in place of symbolic expressions whenever symbolic reasoning is hard or not desired. We describe DSE for a simple core language and then present a minimalist implementation of DSE for Python (in Python) that follows this basic recipe. The code is available at https://www.github.com/thomasjball/PyExZ3/ (tagged "v1.0") and has been designed to make it easy to experiment with and extend.

Keywords. Symbolic Execution, Automatic Test Generation, White-box Testing, Automated Theorem Provers

1. Introduction

Static, path-based symbolic execution explores one control-flow path at a time through a (sequential) program P, using an automated theorem prover (ATP) to determine if the current path p is feasible [4, 11]. Ideally, symbolic execution of a path p through program P yields a logic formula ϕ_p that describes the set of inputs I (possibly empty) to program P such that for any $i \in I$, the execution $P(i)$ follows path p.

If the formula ϕ_p is unsatisfiable then I is empty and so path p is not feasible; if the formula is satisfiable then I is not empty and so path p is feasible. In this case, a model of ϕ_p provides a witness $i \in I$. Thus, a model-generating ATP can be used in conjunction with symbolic execution to automatically generate tests to cover paths in a program. Combined with a search strategy, one gets, in the limit, an exhaustive white-box testing procedure, for which there are many applications [2, 3, 9].

The formula ϕ_p is called a *path-condition* of the path p. We will see that a given path p can induce many different path-conditions. A path-condition ψ_p for path p is *sound* if every input assignment satisfying ψ_p defines an execution of program P that follows path p [7]. By its definition, the formula ϕ_p is sound and the best representation of p (as for all sound path-conditions ψ_p, we have that $\psi_p \implies \phi_p$). In practice, we attempt to compute sound under-approximations of

```
i = an input to program P
while defined(i):
   p = path covered by execution P(i)
   cond = pathCondition(p)
   s = ATP(Not(cond))
   i = s.model()
```

Figure 1 . Pseudo-code for dynamic symbolic execution

ϕ_p such as ψ_p. However, we also find it necessary (and useful) to compute unsound path-conditions.

A path-condition can be translated into the input language of an ATP, such as Z3[5], which provides an answer of "unsatisfiable", "satisfiable" or "unknown", due to theoretical or practical limitations in automatically deciding satisfiability of various logics. In the case that the ATP is able to prove "satisfiable" we can query it for satisfying model in order to generate test inputs. A path-condition for p can be thought of as function from a program's primary inputs to a Boolean output representing whether or not p is executed under a given input. Thus, we are asking the ATP to invert a function when we ask it to decide the satisfiability/unsatisfiability of a path-condition.

The static translation of a path p through a program P into the most precise path-condition ϕ_p is not a simple task, as programming languages and their semantics are very complex. Completely characterizing the set of inputs I that follow path p means providing a symbolic interpretation of every operation in the language so that the ATP can reason about it. For example, consider a method call in Python. Python's algorithm for method resolution order (see MRO) depends on the inheritance hierarchy of the program, a directed, acyclic graph that can evolve during program execution. Symbolically encoding Python's method resolution order is possible but non-trivial. There are other reasons it is hard or undesirable to symbolically execute various operations, as will be explained in detail later.

1.1. Dynamic symbolic execution

Dynamic symbolic execution (DSE) is a form of path-based symbolic execution based on two insights. First, the approach starts by executing program P on some input i, seeding the symbolic execution process with a feasible path [10, 12, 13]. Second, DSE uses concrete values from the execution $P(i)$ in place of symbolic expressions whenever symbolic reasoning is not possible or desired [1, 8]. The major benefit of DSE is to simplify the construction of a symbolic execution tool by leveraging concrete execution behavior (given by actually running the program). As DSE combines both concrete and symbolic reasoning, it also has been called "concolic" execution [14].

The pseudo-code of Figure 1 shows the high level process of DSE. The variable `i` represents an input to program `P`. Execution of program `P` on the input `i`

```
def max2(s,t):
    if (s < t):
        return t
    else:
        return s

def max4(a,b,c,d):
    return max2(max2(a,b),max2(c,d))
```

Figure 2 . Easy example: computing the maximum of four numbers in Python.

traces a path `p`, from which a logical formula `pathCondition(p)` is constructed. Finally, the ATP is called with the negation of the path-condition to find a new input (that hopefully will cover a new path). This pseudo-code elides a number of details that we will deal with later.

Consider the Python function `max4` in Figure 2 , which computes the maximum of four numbers via three calls to the function `max2`. Suppose we execute `max4` with values of zero for all four arguments. In this case, the execution path p contains three comparisons (in the order `(a < b)`, `(c < d)`, `(a < c)`), all of which evaluate false. Thus, the path-condition for path p is `(not(a<b) and not(c<d) and not(a < c))`. Negating this condition yields `((a<b) or (c<d) or (a<c))`. Taking the execution ordering of the three comparisons into account, we derive three expressions from the negated path-condition to generate new inputs that will explore execution prefixes of path p of increasing length:

- *length 0*: `(a<b)`
- *length 1*: `not (a<b) and (c<d)`
- *length 2*: `not (a<b) and not (c<d) and (a<c)`

The purpose of taking execution order into account should be clear, as the comparison `(a<c)` only executes in the case where `(not (a<b) and not (c<d))` holds. Integer solutions to the above three systems of constraints are:

- `a == 0 and b == 2 and c == 0 and d == 0`
- `a == 0 and b == 0 and c == 0 and d == 3`
- `a == 0 and b == 0 and c == 2 and d == 0`

In the three cases above, we sought solutions that kept as many of the variables as possible equal to the original input (in which all variables are equal to 0). Execution of the `max4` function on the input corresponding to the first solution produces the path-condition `((a<b) and not(c<d) and not(b < c))`, from which we can produce more inputs. For this (loop-free function), there are a finite number of path-conditions. We leave it as an exercise to the reader to enumerate them all.

```
def fermat3(x,y,z):
   if (x > 0 and y > 0 and z > 0):
      if (x*x*x + y*y*y == z*z*z):
          return "Fermat and Wiles were wrong!?!"
   return 0
```

Figure 3 . Hard example for symbolic execution

```
def dart(x,y):
  if (unknown(x) == y):
     return 1
  return 0
```

Figure 4 . Another hard example for symbolic execution

1.2. Leveraging concrete values in DSE

We now consider several situations where we can make use of concrete values in DSE. In the realm of (unbounded-precision) integer arithmetic (e.g., bignum integer arithmetic, as in Python 3.0 onwards), it is easy to come up with tiny programs that will be *very difficult*, if not *impossible*, for any symbolic execution tool to deal with, such as the function `fermat3` in Figure 3 .

Fermat's Last Theorem, proved by Andrew Wiles in the late 20th century, states that no three positive integers x, y, and z can satisfy the equation $x^n + y^n = z^n$ for any integer value of n greater than two. The function `fermat3` encodes this statement for $n = 3$. It is not reasonable to have a computer waste time trying to find a solution that would cause `fermat3` to print the string `"Fermat and Wiles were wrong!?!"`. In cases of complex (non-linear) arithmetic operations, such as `x*x*x`, we might choose to handle the operation concretely.

There are a number of ways to deal with the above issue: one is to recognize all non-linear terms in a symbolic expression and replace them with their concrete counterparts during execution. For the `fermat3` example, this would mean that during DSE the symbolic expression `(x*x*x + y*y*y == z*z*z)` would be reduced to the constant `False` by evaluation on the concrete values of variables `x`, `y` and `z`.

Besides difficult operations (such as non-linear arithmetic), other examples of code that we might treat concretely instead of symbolically include functions that are hard to invert, such as cryptographic hash functions, or low-level functions that we do not wish to test (such as operating system functions). Consider the code in Figure 4 , which applies the function `unknown` to argument `x` and compares it to argument `y`. By using the name `unknown` we simply mean to say that we wish to model this function as a black box, with no knowledge of how it operates internally.

In such a case, we can use DSE to execute the function `unknown` on a specific input (say `5013`) and observe its output (say `42`). That is, rather than execute `unknown` symbolically and invoke an ATP to invert the function's path-condition, we simply treat the call to `unknown` concretely, substituting its return value (in this case `42`) for the specialized expression `unknown(5013) == y` to get the predicate `(42 == y)`.

Adding the constraint `(x == 5013)` yields the sound but rather specific path-condition `(x == 5013) and (42 == y)`. Note that the path-condition `(42 == y)` is not sound, as it admits any value for the variable `x`, which likely includes many values for which `(unknown(x) == y)` is false.

1.3. Overview

This introduction elides many important issues that arise in implementing DSE for a real language, which we will focus on in the remainder of the paper. These include how to:

- Identify the code under test P and the symbolic inputs to P;
- Trace the control flow path p taken by execution $P(i)$;
- Reinterpret program operations to compute symbolic expressions;
- Generate a path-condition from p and the symbolic expressions;
- Generate a new input i' by negating (part of) the path-condition, translating the path-condition to the input language of an ATP, invoking the ATP, and lifting a satisfying model (if any) back up to the source level;
- Guide the search to expose new paths.

The rest of this paper is organized as follows. Section 2 describes an instrumented typing discipline where we lift each type (representing a set of concrete values) to a symbolic type (representing a set of pairs of concrete and symbolic values). Section 3 shows how strongest postconditions defines a symbolic semantics for a small programming language and how strongest postconditions can be refined to model DSE. Section 4 describes an implementation of DSE for the Python language in the Python language that follows the instrumented semantics pattern closely (full implementation and tests available at PyExZ3, tagged "v1.0"). Section 5 describes the symbolic encoding of Python integer operations using two decision procedures of Z3: linear arithmetic with uninterpreted functions in place of non-linear operations; fixed-width bit-vectors with precise encodings of most operations. Section 6 offers a number of ideas for projects to extend the capabilities of PyExZ3.

2. Instrumented Types

We are given a universe of classes/types U; a type $T \in U$ carries along a set of operations that apply to values of type T, where an operation $o \in T$ takes an argument list of typed values as input (the first being of type T) and produces a

single typed value as output. Nullary (static) operations of type T can be used to create values of type T (such as constants, objects, etc.)

A program P has typed input variables $v_1 : T_1 \ldots v_k : T_k$ and a body from the language of statements S:

$S \rightarrow v := E$
| **skip**
| S_1 ; S_2
| **if** E **then** S_1 **else** S_2 **end**
| **while** E **do** S **end**

The language of expressions (E) is defined by the application of operations to values, where constants (nullary operations) and program variables form the leaves of the expression tree and non-nullary operators form the interior nodes of the tree. For now, we will consider all values to be immutable. That is, the only source of mutation in the language is the assignment statement.

To introduce symbolic execution into the picture, we can imagine that a type $T \in U$ has (one or more) counterparts in a symbolic universe U'. A type $T' \in U'$ is a subtype of $T \in U$ with two purposes:

- First, a value of type T' represents a pair of values: a concrete value c of (super)type T and a symbolic expression e. A symbolic expression is a tree whose leaves are either nullary operators (i.e., constants) of a type in U or are Skolem constants representing the (symbolic) inputs ($v_1 \ldots v_k$) to the program P, and whose interior nodes represent operations from types in U. We refer to Skolem constants as "symbolic constants" from this point on. Note that symbolic expressions do not contain references to program variables.
- Second, the type T' redefines some of the operations $o \in T$, namely those for which we wish to compute symbolic expressions. An operation $o \in T'$ has the same parameter list as $o \in T$, allowing it to take inputs with types from both U and U'. The return type of $o \in T'$ generally is from U' (though it can be from U). Thus, $o \in T'$ is a proper function subtype of $o \in T$. The purpose of $o \in T'$ is to: (1) perform operation $o \in T$ on the concrete values associated with its inputs; (2) build a symbolic expression tree rooted at operation o whose children are the trees associated with the inputs to o.

Figure 5 presents pseudo code for the instrumentation of a type T via a type T'. The class `Symbolic` is used to hold an expression tree (`Expr`). Given a class $T \in U$, a symbolic type $T' \in U'$ is defined by inheriting from both T and `Symbolic`. This ensures that a T' can be used wherever a T is expected.

A type such as T' only can be constructed by providing a concrete value c of type T and a symbolic expression e to the constructor for T'. This will be done in exactly two places:

- by the creation of symbolic constants associated with the primary inputs ($v_1 \ldots v_k$) to the program;

```
class T' : T, Symbolic {
  T'(c:T, e:Expr) : T(c), Symbolic(e) {}

  override o(this:T, f1:T_1, ... , fk:T_k) : R' {
    var c := T.o(this, f1, ... ,fk)
    var e := new Expr(T.o, expr(self), expr(f1), ..., expr(fk))
    return new R'(c,e)
  }
  ...
}

class R' : R, Symbolic { ... }

function expr(v) = v instanceof Symbolic ? v.getExpr() : v
```

Figure 5 . Type instrumentation to carry both concrete values and symbolic expressions.

- by the instrumented operations as shown in Figure 5 .

An instrumented operation o on arguments (`this`, `f1`, …, `fk`) first invokes its corresponding underlying operator $T.o$ on arguments (`this`, `f1`, …, `fk`) to get concrete value `c`. It then constructs a new expression tree `e` rooted at operator $T.o$, whose children are the result of mapping the function `expr` over (`this`, `f1`, …, `fk`). The helper function `expr(v)` evaluates to an expression tree in the case that `v` is of `Symbolic` type (representing a type in U') and evaluates to `v` itself, a concrete value of some type in U, otherwise. Finally, having computed the values `c` and `e`, the instrumented operator returns R'`(c,e)`, where R is the return type of operator $T.o$, and R' is a subtype of R from universe U'.

Looked at another way, the universe U' represents the "tainting" of types from U. Tainted values flow from program inputs to the operands of operators. If an operator has been redefined (as above) then the taint propagates from its inputs to its outputs. On the other hand, if the operator has not been redefined, then it will not propagate the taint. In the context of DSE, "taint" means that the instrumented semantics carries along a symbolic expression tree e along with a concrete value c.

The choice of types from the universe U' determines how symbolic expressions are constructed. For each $T \in U$, the "most symbolic" (least concrete) choice is the T' that redefines every operator of T (as shown in Figure 5). The "least symbolic" (most concrete) choice is $T' = T$ which redefines no operators. Let $symbolic(T)$ be the set of types in U' that are subtypes of T. The types in $symbolic(T)$ are partially ordered by subset inclusion on the set of operators from T they redefine.

3. From Strongest Postconditions to DSE

The previous section showed how symbolic expressions can be computed via a set of instrumented types, where the expressions are computed as a side-effect of the execution of program operations. This section shows how these symbolic expressions can be used to form a *path-condition* (which then can be compiled into a logic formula and passed to an automated theorem prover to find new inputs to drive a program's execution along new paths). We derive a *path-condition* directly from the *strongest postcondition* (symbolic) semantics of our programming language, refining it to model the basic operations of an interpreter.

3.1. Strongest Postconditions

The strongest postcondition transformer SP [6] is defined over a predicate P representing a set of pre-states and a statement S from our language. The transformer $SP(P, S)$ yields a predicate Q such that for any state s satisfying predicate P, the execution of statement S from state s, if it does not go wrong or diverge, yields a state s' satisfying predicate Q. The strongest postcondition for the statements in our language is defined by the following five rules:

1. $SP(P,\ x := E) \triangleq \exists y\,.\,(x = E[\,x \to y\,]) \wedge P[\,x \to y\,]$
2. $SP(P,\ \textbf{skip}) \triangleq P$
3. $SP(P, S_1; S_2) \triangleq SP(SP(P, S_1),\ S_2)$
4. $SP(P, \textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}) \triangleq$
   $SP(P \wedge E, S_1) \vee SP(P \wedge \neg E, S_2)$
5. $SP(P, \textbf{while}\ E\ \textbf{do}\ S\ \textbf{end}) \triangleq$
   $SP(P, \textbf{if}\ E\ \textbf{then}\ S;\ \textbf{while}\ E\ \textbf{do}\ S\ \textbf{end}\ \textbf{else skip end})$

Rule (1) defines the strongest postcondition for the assignment statement. The assignment is modeled logically by the equality $x = E$ where any free occurrence of x in E is replaced by the existentially quantified variable y, which represents the value of x in the pre-state. The same substitution ($[x \to y]$) is applied to the pre-state predicate P.

Rules (2)-(5) define the strongest postcondition for the four control-flow statements. The rules for the **skip** statement and sequencing (;) are straightforward. Of particular interest, note that the rule for the **if-then-else** statement splits cases on the expression E. It is here that DSE will choose one of the cases for us, as the concrete execution will evaluate E either to be true or false. This gives rise to the path-condition (either $P \wedge E$ or $P \wedge \neg E$). The recursive rule for the **while** loop unfolds as many times as the expression E evaluates true, adding to the path-condition.

3.2. From SP to DSE

Assume that an execution begins with the assignment of initial values $c_1 \ldots c_k$ to the program P's inputs $V = \{v_1 : T_1 \ldots v_k : T_k\}$. To seed symbolic execution,

some of the types T_i are replaced by symbolic counterparts T_i', in which case v_i is initialized to the value $sc_i = T_i'(c_i, SC(v_i))$ instead of the value c_i, where $SC(v_i)$ is the symbolic constant representing the initial value of variable v_i. The symbolic constant $SC(v_i)$ can be thought of as representing any value of type T_i, which includes the value c_i.

Let V_s and V_c partition the variables of V into those variables that are treated symbolically (V_s) and those that are treated concretely (V_c). The initial state of the program is characterized by the formula

$$Init = (\bigwedge_{v_i \in V_s} v_i = sc_i) \ \wedge \ (\bigwedge_{v_i \in V_c} v_i = c_i) \tag{1}$$

Thus, we see that the initial value of every input variable is characterized by a symbolic constant sc_i or constant c_i. We assume that every non-input variable in the program is initialized before being used.

The strongest postcondition is formulated to deal with open programs, programs in which some variables are used before being assigned to. This surfaces in Rule (1) for assignment, which uses existential quantification to refer to the value of variable x in the pre-state.

By construction, we have that every variable is defined before being used. This means that the precondition P can be reformulated as a pair $< \sigma, P_c >$, where σ is a store mapping variables to values and P_c is the path-condition, a list of symbolic expressions (predicates) corresponding to the expressions E evaluated in the context of an **if-then-else** statement. Initially, we have that :

$$\sigma = \{(v_i, sc_i) | v_i \in V_s\} \cup \{(v_i, c_i) | v_i \in V_c\} \tag{2}$$

representing the initial condition $Init$, and $P_c = []$, the empty list. We use σ' to refer to the formula that the store σ induces:

$$\sigma' = \bigwedge_{(v,V) \in \sigma} (v = V) \tag{3}$$

Thus, the pair $< \sigma, P_c >$ represents the predicate $P = \sigma' \wedge (\bigwedge_{c \in P_c} c)$. A store σ supports two operations: $\sigma[x]$ which denotes the value that x maps to under σ; $\sigma[x \mapsto V]$, which produces a new store in which x maps to value V and is everywhere else the same as σ.

Now, we can redefine strongest postcondition for assignment to eliminate the use of existential quantification and model the operation of an interpreter, by separating out the notion of the store:

1. $SP(< \sigma, \ P_c >, \ x := E) \ \stackrel{\triangle}{=} \ < \sigma[x \mapsto eval(\sigma, E)], \ P_c >$

where $eval(\sigma, E)$ evaluates expression E under the store σ (where every occurrence of a free variable v in E is replaced by the value $\sigma[v]$). This is the standard substitution rule of a standard operational semantics.

We also redefine the rule for the **if-then-else** statement so that it chooses which branch to take and appends the appropriate symbolic expression (predicate) to the path-condition P_c:

4. $SP(< \sigma,\ P_c >,\ \textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}) \triangleq$
   let $choice\ =\ eval(\sigma, E)$ **in**
   if $choice$ **then** $SP(< \sigma,\ P_c\ ::\ expr(choice)\ >, S_1)$
   else $SP(< \sigma,\ P_c\ ::\ \neg expr(choice)\ >, S_2)$

The other strongest postcondition rules remain unchanged.

3.3. Summing it up

We have shown how the symbolic predicate transformer SP can be refined into a symbolic interpreter operating over the symbolic types defined in the previous section. In the case when every input variable is symbolic and every operator is redefined, the path-condition is equivalent to the *strongest postcondition* of the execution path p. This guarantees that the path-condition for p is *sound*. In the case where a subset of the input variables are symbolic and/or not all operators are redefined, the path-condition of p is not guaranteed to be sound. We leave it as an exercise to the reader to establish sufficient conditions under which the use of concrete values in place of symbolic expressions is guaranteed to result in sound path-conditions.

This section does not address the compilation of a symbolic expression to the (logic) language of an underlying ATP, nor the lifting of a satisfying assignment to a formula back to the level of the source language. This is best done for a particular source language and ATP, as detailed in the next section.

4. Architecture of PyExZ3

In this section we present the high-level architecture of a simple DSE tool for the Python language, written in Python, called PyExZ3. Figure 6 shows the class diagram (dashed edges are "has-a" relationships; solid edges are "is-a" relationships) of the tool.

4.1. Loading the code under test

The `Loader` class takes as input the name of a Python file (e.g., `foo.py`) to import. The loader expects to find a function named `foo` inside the file `foo.py`, which will serve as the starting point for symbolic execution. The `FunctionInvocation` class wraps this starting point. By default, each parameter to `foo` is a `SymbolicInteger` unless there is decorator `@symbolic` specifying the type to use for a particular argument.

The loader provides the capability to reload the module `foo.py` so that the function `foo` can be reexecuted within the same process from the same

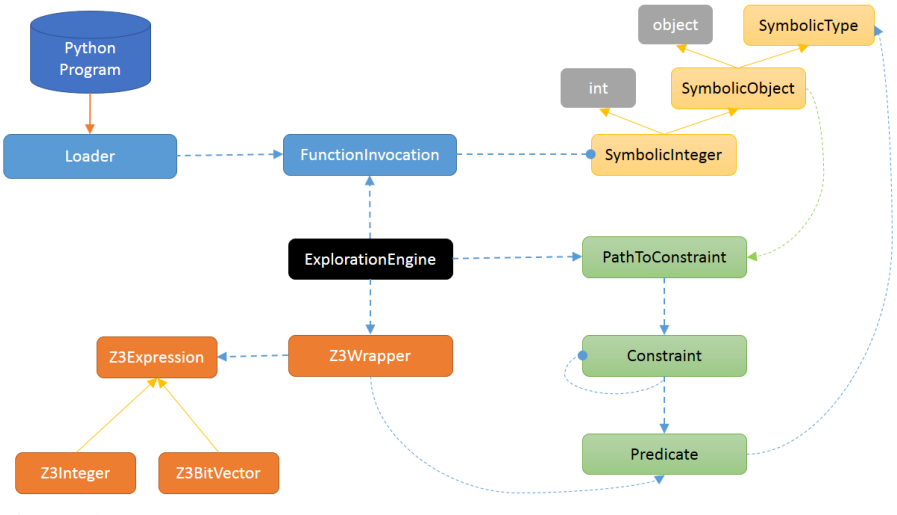

Figure 6 . Classes in PyExZ3

initial state with different inputs (see the class `ExplorationEngine`) via the `FunctionInvocation` class.

Finally, the loader looks for specially named functions `expected_result` (`expected_result_set`) in file `foo.py` to use as a test oracle after the path exploration (by `ExplorationEngine`) has completed. These functions are expected to return a list of values to check against the list of return values collected from the executions of the `foo` function. The presence of the function `expected_result` (`expected_result_set`) yields a comparison of the two lists as bags (sets). We use such weaker tests, rather than list equality, because the order in which paths are explored by the `ExplorationEngine` can easily change due to small differences in the input programs.

4.2. Symbolic types

Python supports multiple inheritance and, more importantly, allows user-defined classes to inherit from its built-in types (such as `object` and `int`). We use these two features two implement symbolic versions of Python objects and integers, following the instrumented type approach defined in Section 2.

The abstract class `SymbolicType` contains the symbolic expression tree and provides basic functions for constructing and accessing the tree. This class does double duty, as it is used to represent the (typed) symbolic constants associated with the parameters to the function, as well as the expression trees (per Section 2). Recall that the symbolic constants only appear as leaves of expression trees. This means that the expression tree stored in a `SymbolicType` will have instances of a `SymbolicType` as some of its leaves, namely those leaves representing the symbolic constants. The abstract class provides an `unwrap` method which returns the pair

of concrete value and expression tree associated with the SymbolicType, as well as a wrap method that takes a pair of concrete value and expression tree and creates a SymbolicType encapsulating them.

The class SymbolicObject inherits from both object and SymbolicType and overrides the basic comparison operations (__eq__, __neq__, __lt__, __le__, __gt__, and __ge__). The class SymbolicInteger inherits from both int and SymbolicObject and overrides a number of int's arithmetic methods (__add__, __sub__, __mul__, __mod__, __floordiv_) and bitwise methods (__and__, __or__, __xor__, __lshift__, __rshift__).

4.3. Tracing control-flow

As Python interprets a program, it will evaluate expressions, substituting the value of a variable in its place in an expression, applying operators (methods) to parameter values and assigning the return values of methods to variables. Value of type SymbolicInteger will simply flow through this interpretation, without necessitating any change to the program or the interpreter. This takes care of the case of the strongest-postcondition rule for assignment, as elaborated in Section 3.2.

The strong-postcondition rule for a conditional test requires a little more work. In Python, any object can be tested in an if or while condition or as the operand of a Boolean operation (and, or, not) The Python base class object provides a method named __bool__ that the Python runtime calls whenever it needs to perform such a conditional test. This hook provides us what we need to trace the conditional control-flow of a Python execution. We override this method in the class SymbolicObject in order to inform the PathToConstraint object (defined later) of the symbolic expression for the conditional (as captured by the SymbolicInteger subclass).

Note that the use of this hook in combination with the tainted types will only trace those conditionals in a Python execution whose values inherit from SymbolicObject; by definition, "untainted" conditionals do not depend on symbolic inputs so there is no value in adding them to the path-condition.

4.4. Recording path-conditions

A Predicate records a conditional (more precisely the symbolic expression found in SymbolicInteger) and which way it evaluated in an execution. A Constraint has a Predicate, a parent Constraint and a set of Constraint children. Constraints form a tree, where each path starting from the root of the tree represents a path-condition. The tree represents all path-conditions that have been explored so far.

The class PathToConstraint has a reference to the root of the tree of Constraints and is responsible for installing a new Constraint in the tree when notified by the overridden __bool__ method of SymbolicObject. PathToConstraint also tracks whether or not the current execution is following an existing path in the tree and grows the tree as needed. In fact, it actually tracks whether or not the current execution follows a particular *expected path* in the tree.

The expected path is the result of the `ExplorationEngine` picking a constraint c in the tree, and asking the ATP if the path-condition consisting of the prefix of predicates up to but not including c in the tree, followed by the negation of c's predicate is satisfiable. If the ATP returns "satisfiable" (with a new input i), then the assumption is that path-condition prefix is sound (that is, the execution of the program on input i will follow the prefix).

However, it is possible for the path-condition to be unsound and for the executed path to diverge early from the expected path, due to the fact that not every operation has a symbolic encoding. The tool simply reports the divergence and continues to process the execution as usual (as a diverging path may lead to some other interesting part of the code).

4.5. From symbolic types to Z3

As we have explained DSE, the symbolic expressions are represented at the level of the source language. As detailed later in Section 5, we must translate from the source language to the input language of an automated theorem prover (ATP), in this case Z3. This separation of languages is quite useful, as we may have the need to translate a given symbolic expression to the ATP's language multiple times, to make use of different features of the underlying ATP. Furthermore, this separation of concerns allows us to easily retarget the DSE tool to a different ATP.

The base class `Z3Expression` represents a Z3 formula. The two subclasses `Z3Integer` and `Z3BitVector` represent different ways to model arithmetic reasoning about integers in Z3. We will describe the details of these encodings in Section 5.

The class `Z3Wrapper` is responsible for performing the translation from the source language (Python) to Z3's input language, invoking Z3, and lifting a Z3 answer back to the level of Python. The `findCounterexample` method does all the work, taking as input a list of `Predicate`s (called `assertions`) as well as a single `Predicate` (called the `query`). The `assertions` represent a path-condition prefix derived from the `Constraint` tree that we wish the next execution to follow, while `query` represents the predicate following the prefix in the tree that we will negate.

The method constructs the formula

$$(\bigwedge_{a \in asserts} a) \wedge \neg query \tag{4}$$

and asks Z3 if it is satisfiable. The method performs a standard syntactic "cone of influence" (CIF) reduction on the `asserts` with respect to the `query` to shrink the size of the formula. For example, if `asserts` is the set of predicates $\{(x < a), (a < 0), (y > 0)\}$ and the query is $(x = 0)$, then the CIF yields the set $\{(x < a), (a < 0)\}$, which does not include the predicate $(y > 0)$, as the variable y is not in the set of variables (transitively) related to variable x.

If the formula is satisfiable a model is requested from Z3 and lifted back to Python's type universe. Note that because of the CIF reduction, the model may not mention certain input variables, in which case we simply keep their values from the execution from which the `asserts` and `query` were derived.

4.6. Putting it all together

The class `ExplorationEngine` ties everything together. It kicks off an execution of the Python code under test using `FunctionInvocation`. As the Python code executes, building symbolic expressions via `SymbolicType` and its subclasses, callbacks to `PathToConstraint` create a path-condition, represented by `Constraint` and `Predicate`. Newly discovered `Constraints` are added to the end of a deque maintained by `ExplorationEngine`.

Given the first seed execution, `ExplorationEngine` starts the work of exploring paths in a breadth-first fashion. It removes a `Constraint` c from the front of its deque and, if c has not been already "processed", uses `Z3Wrapper` to find a new input (as discussed in the previous section) where c is the query (to be negated) and the path to c in the `Constraint` tree forms the assertions.

A `Constraint` c in the tree is considered "processed" if an execution has covered c', a sibling of c in the tree that represents the negation of the predicate associated with c, or if constraint c has been removed from the deque.

5. From Python Integers to Z3 Arithmetic

In languages such as C and Java, integers are finite-precision, generally limited to the size of a machine word (32 or 64 bits, for example). For such languages, satisfiability of finite-precision integer arithmetic is decidable and can be reduced to Z3's theory of bit-vectors, where each arithmetic operation is encoded by a circuit. This translation permits reasoning about non-linear arithmetic problems, such as $\exists x, y, z : x * z + y \leq (z/y) + 5$.

Python (3.0) integers, however, are not finite-precision. They are only limited by the size of machine memory. This means, for example, that Python integers don't overflow or underflow. It also means that we can't hope to decide algorithmically whether or not a given equation over integer variables has a solution in general. Hilbert's famous 10th problem and its solution by Matiyasevich tells us that it is undecidable whether or not a polynomial equation of the form $p(x_1, \ldots, x_n) = 0$ with integer coefficients has an solution in the integers.

This means that we will resort to heuristic approaches in our use of the Z3 ATP. The special case of linear integer arithmetic (LIA) is decidable and supported by Z3. In order to deal with non-linear operations, we use uninterpreted functions (UF). Thus, if Z3 returns "unsatisfiable" we know that there is no solution, but if the Z3 "satisfiable", we must treat the answer as a "don't know". The class `Z3Integer` is used to translate a symbolic expression into the theory LIA+UF and check for unsatisfiability. We leave it as an implementation exercise to check if a symbolic expression can be converted to LIA (without the use of UF) in order to make use of "satisfiable" answers from the LIA solver.

If the translation to `Z3Integer` does not return "unsatisfiable", we use Z3's bit-vector decision procedure (via the class `Z3BitVector`) to heuristically try to find satisfiable answers, even in the presence of non-linear arithmetic. We start with bit-vectors of size $N = 32$ and *bound* the values of the symbolic constants to fit within 8 bits in order to find satisfiable solutions with small values. Also,

because Python integers do not overflow/underflow, the bound helps us reserve space in the bit-vector to allow the results of operations to exceed the bound while not overflowing the bit-vector. As long as Z3 returns "unsatisfiable" we increase the bound. If the bound reaches N, we increase N by 8 bits, leaving the bound where it is and continue.

If Z3 returns "satisfiable", it may be the case that Z3 found a solution that involved overflow in the bit-vector world of arithmetic (modulo $2^N - 1$). Therefore, the solution is validated back in the Python world by evaluating the formula under that solution using Python semantics. If the formula does not evaluate to the same value in both worlds, then we increase N by 8 bits (to create a gap between the bound and N) and continue to search for a solution.

The process terminates when we find a valid satisfying solution or $N = 64$ and the bound reaches 64 (in which case, we return "don't know").

6. Extensions

We have presented the basics of dynamic symbolic execution (for Python). A more thorough treatment would deal with other data types besides integers, such as Python dictionaries, strings and lists, each of which presents their own challenges for symbolic reasoning. There are many other interesting challenges in DSE, such as dealing with user-defined classes (rather than built-in types as done here) and multi-threaded execution.

7. Acknowledgements

Many thanks to the students of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering for their questions and feedback about the first author's lectures on dynamic symbolic execution. The following students of the summer school helpfully provided tests for the PyExZ3 tool: Daniel Darvas, Damien Rusinek, Christian Dehnert and Thomas Pani. Thanks also to Peter Chapman for his contributions.

References

[1] Cristian Cadar and Dawson R. Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of 12th International SPIN Workshop*, pages 2–23, 2005.

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

[3] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[4] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2 (3): 215–222, 1976.

[5] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the 14th International Conference of Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[6] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[7] Patrice Godefroid. Higher-order test generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–269, 2011.

[8] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[9] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55 (3): 40–44, 2012.

[10] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Generating test data for branch coverage. In *Proceedings of the Automate Software Engineering Conference*, pages 219–228, 2000.

[11] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19 (7): 385–394, 1976.

[12] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16 (8): 870–879, 1990.

[13] Bogdan Korel. Dynamic method of software test data generation. *Journal of Software Testing, Verification and Reliability*, 2 (4): 203–213, 1992.

[14] Koushik Sen and Gul Agha. CUTE and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of 18th Computer Aided Verification Conference*, pages 419–423, 2006.