Finding Errors in Python Programs Using Dynamic Symbolic Execution

Samir Sapra[1], Marius Minea[2], Sagar Chaki[1], Arie Gurfinkel[1], and Edmund M. Clarke[1]

[1] Carnegie Mellon University, Pittsburgh, PA, USA **
[2] Politehnica University of Timişoara, Romania

Abstract. For statically typed languages, dynamic symbolic execution (also called concolic testing) is a mature approach to automated test generation. However, extending it to dynamic languages presents several challenges. Complex semantics, fragmented and incomplete type information, and calls to foreign functions lacking precise models make symbolic execution difficult. We propose a symbolic execution approach that mixes concrete and symbolic values and incrementally solves path constraints in search for alternate executions by lazily instantiating axiomatizations for called functions as needed. We present the symbolic execution model underlying this approach and illustrate the workings of our prototype concolic testing tool on an actual Python software package.

1 Introduction

Dynamic symbolic execution (DSE) has been very successful for generating tests and finding errors. It accumulates path constraints over symbolic inputs rather than executing with concrete values. Java Pathfinder [5], Pex [7], or KLEE [2] try all possible program paths, using full symbolic models also for environment interactions. Concolic testing, a variant used in DART [3], CUTE [6] and CREST [1], generates symbolic constraints guided by concrete executions, and then modifies them to explore alternate paths. Approximating some execution fragments through concretization proves useful even in the absence of complete models.

Compared to existing work in the context of static typing, symbolic execution for Python as dynamically typed language raises a series of new challenges:
i) The language complexity makes symbolic execution difficult: First, a more complex theory is needed to express path conditions precisely. Python objects have dictionaries of attributes and hence one needs to handle strings and maps. Dictionary keys can be added dynamically and can be arbitrary hashables, not just integers or strings. A variety of runtime errors and exceptions related to dynamic features are handled in different ways.

** This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

Moreover, Python is often used to glue together components in other languages, for which we may not have models. Library functions are often in native code, thus values become concretized during execution and can no longer be tracked symbolically. This work avoids the cost and complexity of eager full symbolic execution by using path constraints that mix concrete and symbolic values. These constraints are solved incrementally in a search for satisfying program inputs, lazily instantiating axiomatized models of executed functions as they are needed.
ii) Type information for objects is incomplete and fragmented. Type constraints are implicitly accumulated from successful runtime checks (objects must have the accessed attributes, be iterable, callable, etc.). An object's type may not be completely known: `x[1]` could be indexing a list, tuple, string, dictionary, or user-defined type. This complicates formalizing and tracking type constraints and also means a program can hide many more bugs. Since many conditions can be flipped to explore alternate types and program paths, it is crucial to steer this search efficiently and avoid exploring uninteresting execution paths. This work selects relevant conditions based on data dependencies, and uses the solver output (unsatisfiable core) to direct the choice of alternate paths.

A Motivating Example. Version 0.93 of `dnuos` (`https://bitheap.org/dnuos/`) – which creates collections of audio files – crashes on an empty directory. The bug is in function `uniq` (line 2 in Fig. 1) – accessing a list without a non-emptiness check. A faulty run has `uniq` called from `types` (l. 7) on a list created with `map` (l. 6) from method `streams`, which filters (l. 10) a list returned by `children`. The latter iterates (l. 13) over a list produced by `os.listdir` for the input pathname.

```
1  def uniq(list):
2      list[0] = [ list[0] ]
3      return reduce(lambda A,x: x in A and A or A+[x], list)
4  def types(self):
5      if self._types != None: return self._types
6      types = map(lambda x: x.type(), self.streams())
7      self._types = uniq(types)
8  def streams(self):
9      ...
10     list = filter(_is_audio_file, self.children())
11 def children(self):
12     if self._children: return self._children
13     self._children = map(lambda x: os.path.join(self.path,x),
14                          os.listdir(self.path))
15     return self._children
```

Fig. 1. Code fragments from `dnuos` for processing a list of audio files.

The challenges are: (i) to detect such errors automatically, i.e., finding a buggy path starting from a successful run (here, on a non-empty directory); and (ii) applying DSE in coverage mode to detect as many errors as possible.

We first describe in Sec. 2 the architecture of our concolic testing engine and the systematic search for alternate execution paths by lazily instantiating the needed constraints. Sec. 3 then briefly illustrates key aspects of i) the formalism used to represent Python path conditions, ii) the symbolic bytecode semantics, and iii) the axiomatization of library functions. Finally in Sec. 4, we show how these are tied together in our prototype CutiePy by revisiting the above example.

2 An Architecture for Concolic Testing

Dynamic symbolic execution is driven by a concrete execution with some initial inputs. A symbolic execution engine is run in lockstep, working with symbolic constraints over program variables, rather than concrete values. Given formal semantics for every instruction, these constraints can be accumulated in a *path condition*, which includes all branch conditions taken on the path, and characterizes all inputs for which the program will take the same path. To explore a new path, a branch condition is flipped and, together with the path condition leading to it, is passed to a solver which returns inputs to exercise the new path.

Our symbolic execution framework is distinguished by *how constraints are expressed and collected* for each instruction, and *how branches are flipped*. We describe the former in Sec. 3, including what to do when fully symbolic execution is not feasible. Here we outline how to find new executions (Algorithm 1).

A path condition is a list of clauses that are either *definitions* or *conditions*. Definitions have the form $v = f(s_1, \ldots, s_k)$ with v a variable and s_i constants or variables. Conditions can be explicit (program branch or loop conditions) or implicit, denoting statement execution without error (e.g., predicates hasattr, iscallable, isiterable). Program and library functions can be interpreted (fully formalized, cf. Sec. 3) or uninterpreted, if there is no complete model for them.

Given a path condition, the dependence set $Dep(C)$ of a clause is defined as:
– for a condition, $Dep(C)$ is the set of clauses that share variables with C.
– for a definition $v = f(s_1, \ldots, s_k)$, $Dep(C)$ is the set of all conditions that contain variables from C, plus any definitions of variables in the right-hand side of C.

Define $Dep^+(C)$ transitively as the smallest set such that $Dep(C) \subseteq Dep^+(C)$ and if $C' \in Dep^+(C)$ is an interpreted clause, then $Dep(C') \subseteq Dep^+(C)$.

Let $\mathcal{I}$ be the set of program inputs and $FV(\mathcal{C})$ be the set of variables in the set of clauses $\mathcal{C}$ that do not appear on the left-hand side of a definition.

Algorithm 1 Selection of alternate execution paths

```
1: function FLIP([r_1, r_2, ..., r_k], flipped)
2:     Φ ← flipped ∪ Dep^+(flipped)
3:     while sat(Φ) do
4:         if FV(Φ) ⊆ I then return sat_assignment
5:         else strengthen Φ with lemmas for FV(Φ) \ I
6:     u ← max{i | i ≤ k ∧ r_i ∈ unsat_core(Φ)}
7:     for r_j is explicit condition, j = u downto 1 do
8:         if (Inputs = FLIP([r_1, r_2, ... r_{j-1}], ¬r_j)) ≠ ∅ then return Inputs
9:     return ∅ (* failed *)
```

Starting from a condition to be flipped, the algorithm propagates relevant constraints (obtained in l. 2) to program inputs. A key point is that uninterpreted functions are instantiated with lemmas (l. 5) lazily and incrementally. Thus we do not need an eager fully symbolic execution and can accommodate concretization, e.g., due to library functions, when collecting the path condition. Symbolic constraints for these functions are re-introduced to the extent needed. If no inputs are found for the chosen path, we flip the last condition affecting

the unsatisfiable core, attempting to preserve the longest possible prefix of the given execution, and the process is repeated. To generate maximal test coverage rather than force a specific path, conditions are simply flipped one by one. In particular, flipping implicit conditions (hasattr, isiterable) can find type errors.

3 A Symbolic Execution Model for Python

We briefly present the key points of a theory for expressing symbolic constraints from Python program executions. We describe the sorts and functions used, and give examples of bytecode semantics and axioms for Python library functions.

Sorts: We use the standard sorts Bool and Int, and an uninterpreted ObjectSort for Python objects. Since Python types are also objects, we define a sub-sort PyType of ObjectSort and constants for Python predefined types (PyBool, PyInt, PyNoneType, PyTuple, PyList, PyListiterator, PyDict) etc. For this prototype we do not distinguish plain from long integers, and do not handle floating point.

Functions and predicates that model properties of Python objects include:

- $\mathsf{id} : \mathsf{ObjectSort} \to \mathsf{Int}$ provides each object with a unique identity
- $\mathsf{typeof} : \mathsf{ObjectSort} \to \mathsf{PyType}$ gives the type of an object
- $\mathsf{hasattr}_{name} : \mathsf{ObjectSort} \to \mathsf{Bool}$ if an object has an attribute called *name*
- $\mathsf{iscallable} : \mathsf{ObjectSort} \to \mathsf{Bool}$ to check for a function or method
- $\mathsf{isiterable} : \mathsf{ObjectSort} \to \mathsf{Bool}$ if object is array, dictionary, list, string, tuple

We also introduce several partial functions; they are used together with the corresponding conditions for the functions being defined:

- $\mathsf{intof} : \mathsf{ObjectSort} \to \mathsf{Int}$ maps an integer object (type PyInt) to its value
- $\mathsf{iterof} : \mathsf{ObjectSort} \to \mathsf{ObjectSort}$ makes an iterator from an isiterable object

The theory should be decidable and allow reasoning about objects whose types are being lazily discovered. We translate formulae expressed in this theory into linear integer arithmetic with uninterpreted functions and arrays (AUFLIA) over uninterpreted sorts and algebraic datatypes.

Axioms are encoded as universal sentences in an SMT solver (Z3 [4]). They are similar but distinct from the axiomatizations we introduce as partial interpretations of libraries or non-core language features. As example, we show an axiom for `list.append`, which returns `None` and mutates its first argument, as expressed using the function *Store* from the AUFLIA theory.

$$\forall r, newL, oldL, v : \mathsf{ObjectSort} \;.\; \mathsf{typeof}\; oldL = \mathsf{PyList} \;\Rightarrow$$
$$(newL, r) = \texttt{LIST\_APPEND}\;(oldL, v) \Rightarrow r = \mathsf{PyNone} \wedge \mathsf{typeof}\; newL = \mathsf{PyList}$$
$$\wedge\; \mathsf{lenof}\; newL = \mathsf{lenof}\; oldL + 1 \wedge \mathsf{seqof}\; newL = Store(\mathsf{seqof}\; oldL, \mathsf{lenof}\; oldL, v)$$

Concolic execution semantics: Our implementation uses the `sys.settrace()` API and a custom build of the CPython interpreter. Symbolic execution is done in lockstep with the concrete Python interpreter, which at the bytecode level operates as a stack machine. A (single-threaded) running program corresponds to a stack of *frames*, each representing some executing function called by the parent frame. Let S denote a frame's associated *valuestack* of object references o. Name binding and variable lookup in a frame is done using its `locals`, `globals`, and `__builtins__` dictionaries; here we assume only a locals context called C.

Symbolic semantics are defined for each bytecode instruction, reflecting decisions such as whether to symbolically model or concretize various operations. The symbolic valuestack Σ of an execution frame may contain both symbolic expressions e and concrete object references o (in case of concretization); any concrete entries agree with the concrete valuestack. The symbolic context Γ is a mapping from strings (variable names) to symbolic expressions.

We present as example two rules for the symbolic semantics of `BINARY_SUBSCR`:

$$\frac{C, o :: o' :: S \to C, o'' :: S \quad e' \text{ symbolic} \quad \texttt{typeof}(o') \texttt{ == PyList} \quad o \text{ non-negative}}{\Gamma, e :: e' :: \Sigma \Rightarrow \Gamma, \mathit{Select}\,(e', e) :: \Sigma \quad \mathit{assert}\ \mathsf{typeof}\, e = \mathsf{PyInt} \wedge \mathsf{typeof}\, e' = \mathsf{PyList} \wedge \mathsf{hasattr}_{\_\_getitem\_\_}\, e' \wedge \mathsf{lenof}\, e' \geq e + 1}$$

$$\frac{C, o :: o' :: S \to C, o'' :: S \quad e' \text{ symbolic}}{\Gamma, e :: e' :: \Sigma \Rightarrow \Gamma, o'' :: \Sigma \quad \mathit{assert}\ \mathsf{hasattr}_{\_\_getitem\_\_}\, e'}$$

Both rules track the indexed collection e' symbolically. In rule 1, o' has the actual type `PyList`, and successful execution implies that o is an integer, which allows us to derive size constraints and track the result of subscripting symbolically. In rule 2, we assert (append to the path condition) the only constraint we learn, $\mathsf{hasattr}_{\_\_getitem\_\_}\, e'$, and we push the concrete result $o'' = o'[o]$, onto Σ.

Concretization when calling functions that lack models (e.g., native code) is one of the main obstacles to building symbolic path conditions. On return from such a function, we re-introduce a symbolic variable for the result. This helps track data flow in spite of concretization. Treating the function as uninterpreted also allows us to lazily ignore conditions which are irrelevant for the testing goal.

For mutable objects, we exploit the Python bytecode interpreter to introduce the additional indirection needed to update referrers. For dictionaries, we model only fields that are updated with symbolic values. In both cases, we limit symbolic modeling to items that are strictly needed.

4 Case Study and Conclusions

We explain a run of our tool CutiePy on the example of Fig. 1. The unit test executes the call `audiodir.Dir(i_filename).types()` on input `i_filename="./dummy"`, in an environment where the directory `./dummy` contains a single, valid music file. CutiePy produces a path condition with ~ 400 constraints (after instantiating partial interpretations). Of particular interest is generating a test that exercises the unchecked list access on line 2, which is compiled to bytecode `BINARY_SUBSCR`.

Since that list is produced by the built-in function `map` (Fig. 1, l. 6) written in C, we need a partial interpretation to reason about it. To achieve this, CutiePy replaces the standard map with a workable model given by the Python function: `audiodir.__builtins__["map"] = lambda f,L: [f(x) for x in L]`. Such models are inserted up-front and are present at what we designate as the 'first' call to FLIP. Thus, the path condition fragment Φ sent to Z3 by FLIP at line 3 is:

```
p385 == Not(lenof(v148) >= 1)                    flipped constraint
p384 == (typeof(v148) == PyList)
p377 == (v141, v148) == LIST_APPEND(v143, v147)  call to APPEND
p364 == (lenof(v143) == 0)                       execution of map model
p363 == (typeof(v143) == PyList)                 v143 empty initial list
```

Combined with the `APPEND` axiom of Sec. 3, Z3 finds these clauses inconsistent (unsatisfiable core: `[p385,p377,p364,p363]`). To break the unsat core, keeping a maximal execution prefix, lines 6–7 identify the condition `Not(exhausted(v144))` as candidate for *flipped*. Intuitively, we want to flip the branch just prior to `list.append` (which contradicts our goal of a zero-length list). CutiePy has reasoned about l. 6 in Fig. 1 at bytecode level, concluding that `map` must return a list whose iterator (`v144`) will be immediately exhausted, i.e., an empty list.

In two more recursive calls, FLIP propagates constraints to primary inputs, but not with the intended bug-revealing trace. In the first call, line 7 of FLIP identifies r_j = `(exhausted (nexted_once v29))` as the next candidate for *flipped*. CutiePy has determined through bytecode-level reasoning that `list` should have ≥ 2 elements. In the next recursive call to FLIP, Dep^+ finally reaches constraints on the primary input `i_filename`, pending models for `os.path.join` and `os.listdir`. Here Φ is found unsat, and in line 7 $r_j = r_u$ = `Not(exhausted(v29))`; however, by this point execution has already diverged from our intended one.

When the right r_j is picked in line 7, CutiePy will discover the bug in `uniq`: `exhausted(v144)` requires `filter` returning an empty list (again an immediately exhausted iterator), which in turn necessitates `map` returning an empty list in l. 13, which via a partial axiomatization of `os.path.listdir` and appropriately set up environment leads to the new primary input `i_filename="./emptydirectory"`.

When forcing execution to a particular point, flipping the right conditions impacts efficiency. A promising heuristic is to focus on loop conditions.

Our initial experiments have shown that for the dynamic features of Python a key challenge is tracking the right amount of symbolic information during execution. We show how to do this by lazily constructing and solving constraints, and using complete or partial axiomatizations of library functions as needed. Further evaluation will provide insight into the amount and types of bugs that can be automatically found, and how to tune the framework to effectively and efficiently zoom in on the most representative and relevant errors.

References

1. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: 23rd International Conference on Automated Software Engineering. pp. 443–446. ACM (2008)
2. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: 8th OSDI. USENIX (2008)
3. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Programming Language Design and Implementation. pp. 213–223. ACM (2005)
4. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
5. Păsăreanu, C.S., Rungta, N., Visser, W.: Symbolic execution with mixed concrete-symbolic solving. In: 20th ISSTA. pp. 34–44. ACM (2011)
6. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: 10th ESEC/13th SIGSOFT FSE. pp. 263–272. ACM (2005)
7. Tillmann, N., de Halleux, J.: Pex—white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)