Using “concolic” testing to find bugs in python code (Work in progress)

Samir Sapra, Sagar Chaki, Arie Gurfinkel, Edmund Clarke
Carnegie Mellon University

NSF Expeditions in Computing

Introduction

The most expensive phase of software development is testing, often consuming over half the budget of a software project. A study by NIST [1] has estimated that software defects cost $60 billion a year in the U.S. economy alone. Unfortunately, test generation is a traditionally manual process.

Recent work on "concolic testing" has alleviated this problem, for languages like C and Java. Tools in this area include JPF-SE [2], DART [3], CUTE [4], jCUTE [5], CREST, KLEE [6], EXE [7], Pex [8] and SAGE [9].

However, an increasing amount of software is now being written in so-called scripting/interpreted languages like python. Our work seeks to extend to these languages the benefits of "concolic testing."

The term "concolic" derives from the fact that the technique is a hybrid of concrete and symbolic testing. Some authors refer to concolic testing as *explicit path model checking*.

Why study concolic testing?

- ✓ Unlike traditional testing, concolic testing is automated.
- ✓ Unlike random automated testing, concolic testing is able to use fewer inputs to achieve greater branch coverage.
- ✓ Concolic testing takes advantage of SMT solvers. With recent and continuing improvements in these solvers, concolic testing tools gain the ability to handle more and more complex programs.
- ✓ Given a program with types and data structures that are too complex, many formal verification techniques will simply not handle them. Concolic testing "degrades gracefully" by falling back to concrete execution where it can't reason symbolically.

Limitations

- ✘ Concolic testing is not able to take advantage of human insight into the software under test, in the manner that a manual tester would be able to.
- ✘ Unlike formal verification techniques, concolic testing does not strive for completeness. Since it is a testing technique, its goal is to reach error locations in a program, not to certify the absence of bugs.

Overview of concolic testing

Concolic testing repeatedly runs a given program, each time with different inputs. The goal is to cover as many branches of execution as possible, with the ultimate goal of bringing the program to an error state.

#1) For its initial stage, a concolic testing algorithm is usually "seeded" with a concrete execution of the program. (This may possibly be done by feeding random inputs.)
Given the initial concrete execution, the tool computes, for each program point, the symbolic state of the program at that point.

The ***symbolic state*** at a program point includes:
- Program Counter
- Symbolic values of the various variables of the program
- Path condition (described below) that must be satisfied for program execution to reach this point.

The ***path condition*** is a quantifier-free formula (in some theory) over the variables of the program. The path condition determines execution path: an instantiation of program variables results in a given execution path iff that instantiation satisfies the path constraint.

For example, for inputs x = 3, y = 2, the resulting concrete execution is highlighted below

```
1:    if (x > y):
2:        if (y > 0):
3:            return 0
4:        else:
5:            assert(False)
```

and the path condition at line 3 is $(x_0 > y_0)$ AND $(y_0 > 0)$

#2) After a concrete execution completes, one of the path constraints is negated, and the result is sent to an SMT solver. The SMT solver will return a model that forces execution along some other path.

In our example, negating a constraint might give $(x_0 > y_0)$ AND $(y_0 <= 0)$

for which the SMT solver might return $x_0 = 1$, $y_0 = -1$

#3) Step 1 is repeated. *In our case we have found a bug!*

```
1:    if (x > y):
2:        if (y > 0):
3:            return 0
4:        else:
5:            assert(False)
```

- ➢ When the symbolic constraints are too complex for the SMT solver, the concolic tool replaces some symbols with concrete values.
- ➢ Branch exploration can continue indefinitely! Most tools use bounded depth-first search.

Current Stage

We have designed an initial concolic testing algorithm to handle basic python features. Our next steps are to refine it and begin implementing it so as to gain practical experience with it.

Observations & Challenges

By far the most significant challenge is python's dynamic type system. For example:

- ➢ Types for function arguments typically are not known until run-time
- ➢ A variable may point to an object of one type, but may then later be re-assigned to point to an object of another type.
- ➢ Classes may be created dynamically, and objects may add fields at run-time
- ➢ The exec() function allows an input string to be executed as python code

Literature Cited

1. The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and Technology, Planning Report 02-3, May 2002.
2. S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: a symbolic execution extension to Java PathFinder. In TACAS'07, 2007.
3. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In PLDI'05, June 2005.
4. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In ESEC/FSE'05, Sep 2005.
5. K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In CAV'06, 2006.
6. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI'08, Dec 2008.
7. C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically generating inputs of death. In CCS'06, Oct--Nov 2006.
8. N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In TAP'08, Apr 2008.
9. P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In NDSS'08, Feb. 2008.