Concolic Testing

Koushik Sen
EECS Department,
UC Berkeley, CA, USA.
ksen@cs.berkeley.edu

ABSTRACT

Concolic testing automates test input generation by combining the concrete and symbolic (concolic) execution of the code under test. Traditional test input generation techniques use either (1) concrete execution or (2) symbolic execution that builds constraints and is followed by a generation of concrete test inputs from these constraints. In contrast, concolic testing tightly couples both concrete and symbolic executions: they run simultaneously, and each gets feedback from the other.

We have implemented concolic testing in tools for testing both C and Java programs. We have used the tools to find bugs in several real-world software systems including SGLIB, a popular C data structure library used in a commercial tool, a third-party implementation of the Needham-Schroeder protocol and the TMN protocol, the scheduler of Honeywell's DEOS real-time operating system, and the Sun Microsystems' JDK 1.4 collection framework. In this tutorial, we will describe concolic testing and some of its recent extensions.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Symbolic execution, Testing tools

Keywords

concolic testing, random testing, explicit path model-checking, data structure testing, unit testing, testing C programs

1. INTRODUCTION

Today's software systems suffer from poor reliability, with software errors costing the U.S. economy upwards of $60 billion annually [13]. Our techniques to ensure software reliability are far from the level of maturity attained by other engineering disciplines that create critical infrastructure.

Testing is the predominant technique in industry to ensure software quality. Exhaustive test generation is essential for effective and rigorous testing. However, existing exhaustive techniques for automated test generation [1, 4, 5, 9–11, 15, 19–21], based on symbolic execution and model checking, usually require a lot of time and fail to scale for large software due to the limitations of underlying theorem provers and symbolic analyzers. Random testing [2, 3, 6, 7, 14, 16], in contrast, is fast and scalable, but it is not exhaustive—random testing usually tests a small fraction of all program behaviors.

Concolic testing [8,12,17,18] combines random testing and symbolic execution to partly remove the limitations of random testing and symbolic execution based testing: concrete values from random testing are used to partly overcome the limitations of symbolic execution, and symbolic execution is used to generate concrete test inputs that give better coverage than random testing. We have implemented concolic testing for C and Java programs in two publicly available tools called CUTE and jCUTE, respectively. We have used CUTE and jCUTE to find bugs in several real-world software systems including SGLIB, a popular C data structure library used in a commercial tool, implementations of the Needham-Schroeder protocol and the TMN protocol, the scheduler of Honeywell's DEOS real-time operating system, and the Sun Microsystems' JDK 1.4 collection framework. In this tutorial, we give an overview of concolic testing and its implementations.

2. OVERVIEW OF CONCOLIC TESTING

In concolic testing, our goal is to generate data inputs that would exercise all the feasible execution paths (up to a given length) of a sequential program having memory graphs as inputs. We next describe the essential idea behind concolic testing.

Concolic testing uses concrete values as well as symbolic values for the inputs and executes a program both concretely and symbolically. This is called *concolic execution*. The concrete execution part of concolic execution constitutes the normal execution of the program. The symbolic execution part of concolic execution collects symbolic constraints over the symbolic input values at each branch point encountered along the concrete execution path. At the end of the concolic

ASE'07, November 4–9, 2007, Atlanta, Georgia, USA.

execution, the algorithm has computed a sequence of symbolic constraints corresponding to each branch point. We call the conjunction of these constraints a *path constraint*. Observe that all input values that satisfy a given path constraint will explore the same execution path.

Concolic testing is based on concolic execution. Concolic testing first generates random values for primitive inputs and the `NULL` value for pointer inputs. Then the algorithm does the following in a loop: it executes the code concolically with the generated input. At the end of the execution a symbolic constraint in the path constraint is negated and solved using constraint solvers to generate a new test input that directs the program along a different execution path. The loop is repeated with the newly generated test input. The loop continues until the algorithm has explored all feasible distinct execution paths using a depth-first search strategy.

A complication arises from the fact that for some symbolic constraints, our constraint solver may not be powerful enough to compute concrete values that satisfy the constraints. To address this difficulty, such symbolic constraints *are simplified by replacing some of the symbolic values with concrete values*. Because of this, concolic testing is complete only if given an oracle that can solve all constraints in a program, and the length and the number of paths is finite. Note that because the algorithm does concrete executions, it is sound, i.e. all bugs it infers are real.

3. IMPLEMENTATION

In concolic testing, since we execute a program both concretely and symbolically simultaneously, we can implement concolic execution through program instrumentation. Program instrumentation inserts function calls throughout the program. During an execution of an instrumented program, the original code of the program performs the concrete execution and the inserted function calls perform the symbolic execution without interfering with the normal execution of the program. We believe that our implementation of concolic testing using program instrumentation is a novel idea which enabled us to quickly and elegantly implement symbolic execution without writing a full-fledged symbolic interpreter. Moreover, since our symbolic execution piggy-backs the normal execution, we have the capability to approximate the symbolic execution of some statements by their concrete outcomes. For example, if we call a function for which we do not have the code for symbolic execution, we use the outcome of its concrete execution as an approximation of its symbolic execution.

In this tutorial, we shall describe a simple implementation of concolic testing where the inserted functions calls record the statements executed and generate a simple ASCII file called a trace file. Concolic execution is then implemented as a program that takes this trace file and generates an input file. Our experience shows that such a program can be written in Python or similar other scripting languages using at most 250 lines of code.

Acknowledgements

Part of this work is done in collaboration with Gul Agha, Patrice Godefroid, Nils Klarlund, and Darko Marinov. This work is supported in part by the NSF Grant CNS-0720906.

4. REFERENCES

[1] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating Test from Counterexamples. In *Proc. of the 26th ICSE*, pages 326–335, 2004.

[2] D. Bird and C. Munoz. Automatic Generation of Random Self-Checking Test Cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[3] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proc. of 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 268–279, 2000.

[4] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2:215–222, 1976.

[5] P. D. Coward. Symbolic execution systems-a review. *Software Engineering Journal*, 3(6):229–239, 1988.

[6] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.

[7] J. E. Forrester and B. P. Miller. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*, 2000.

[8] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[9] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th Int. Conf. on TACAS*, pages 553–568, 2003.

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

[11] G. Lee, J. Morris, K. Parker, G. A. Bundell, and P. Lam. Using symbolic execution to guide test generation: Research articles. *Softw. Test. Verif. Reliab.*, 15(1):41–61, 2005.

[12] R. Majumdar and K. Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426. IEEE, 2007.

[13] The economic impacts of inadequate infrastructure for software testing. National Institute of Standards and technology, Planning Report 02-3, May 2002.

[14] J. Offut and J. Hayes. A Semantic Model of Program Faults. In *Proc. of ISSTA'96*, pages 195–200, 1996.

[15] L. Osterweil. Integrating the testing, analysis and debugging of programs. In *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, pages 73–102, New York, NY, USA, 1984. Elsevier North-Holland, Inc.

[16] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *19th European Conference Object-Oriented Programming*, 2005.

[17] K. Sen and G. Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification (CAV'06)*, LNCS, 2006. (To Appear).

[18] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*. ACM, 2005.

[19] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[20] S. Visvanathan and N. Gupta. Generating test data for functions with pointer inputs. In *17th IEEE International Conference on Automated Software Engineering*, 2002.

[21] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Procs. of TACAS*, 2005.