LCT: An Open Source Concolic Testing Tool for Java Programs[1]

Kari Kähkönen, Tuomas Launiainen, Olli Saarikivi, Janne Kauttio, Keijo Heljanko and Ilkka Niemelä

Department of Information and Computer Science
School of Science
Aalto University
PO Box 15400, FI-00076 AALTO, Finland
{Kari.Kahkonen,Tuomas.Launiainen,Janne.Kauttio,Keijo.Heljanko, Ilkka.Niemela}@tkk.fi, Olli.Saarikivi@iki.fi

Abstract

LCT (LIME Concolic Tester) is an open source concolic testing tool for sequential Java programs. In concolic testing the behavior of the tested program is explored by executing it both concretely and symbolically at the same time. LCT instruments the bytecode of the program under test to enable symbolic execution and collects constraints on input values that can be used to guide the testing to previously unexplored execution paths. The tool also supports distributing the test generation and execution to multiple processes that can be run on a single workstation or even on a network of workstations. This paper describes the architecture behind LCT and demonstrates through benchmarks how the distributed nature of the tool makes testing more scalable.

Keywords: Symbolic execution, concolic testing, constraint solving, bytecode instrumentation.

1 Introduction

Automated testing has the potential to improve reliability and reduce costs when compared to manually written test cases. One such technique that has recently received interest is concolic testing which combines concrete and symbolic execution. Based on this technique, we have developed an open

[1] Work financially supported by Tekes - Finnish Agency for Technology and Innovation, Conformiq Software, Elektrobit, Nokia, Space Systems Finland, and Academy of Finland (projects 126860, 128050 and 139402).

This paper is electronically published in the proceedings of
the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation
URL: http://software.imdea.org/ pierreganty/bytecode2011/

source tool called LCT (LIME Concolic Tester) that can automatically explore different execution paths and generate test cases for sequential Java programs.

In this paper we give an overview of LCT and also demonstrate the distributed nature of the tool with benchmarks where several Java programs are tested in a way that multiple instances of the program under test are concurrently taking part in the testing process.

The main improvements in LCT over existing Java concolic testing systems such as jCUTE [10] are the following: (i) the use of bitvector SMT solver Boolector [1] makes the symbolic execution of Java more precise as integers are not considered unbounded, (ii) the twin class hierarchy [4] instrumentation approach of LCT allows the Java core classes to be instrumented, (iii) the tool architecture supports distributed testing; and (iv) the tool is freely available as open source. Distributed constraint solving has been previously employed by the Microsoft fuzzing tool SAGE [6] that uses a distributed constraint solver Disolver while LCT uses a non-distributed constraint solver but can work on several branches of the symbolic execution tree in parallel.

The rest of the paper is structured as follows. Section 2 briefly describes the algorithm behind concolic testing, Section 3 gives an overview of LCT and Sect. 4 discusses the experiments that have been done with the tool.

2 Concolic Testing

Concolic testing [5,11,2] is a method where a given program is executed both concretely and symbolically at the same time in order to explore the different behaviors of the program. The main idea behind this approach is to, at runtime, collect symbolic constraints on inputs to the system that specify the possible input values that force the program to follow a specific execution path. Symbolic execution of programs is typically made possible by instrumenting the system under test with additional code that collects the constraints without disrupting the concrete execution. For a different approach that uses a non-standard interpreter of bytecode instead of instrumentation, see Symbolic Java PathFinder [9].

Concolic testing starts by first executing a program under test with any concrete input values. The execution of the program can branch into different execution paths at branching statements that depend on input values. When executing such statements, concolic testing constructs a symbolic constraint that describes the possible input values causing the program to take the true or false branch at the statement in question. A *path constraint* is a conjunction of these symbolic constraints and new test inputs for the subsequent test runs are generated by solving them. Typically this is done by using SMT (Satisfiability-Modulo-Theories) solvers with integer arithmetic or bit-vectors as the underlying theory. By continuing this process, concolic testing can

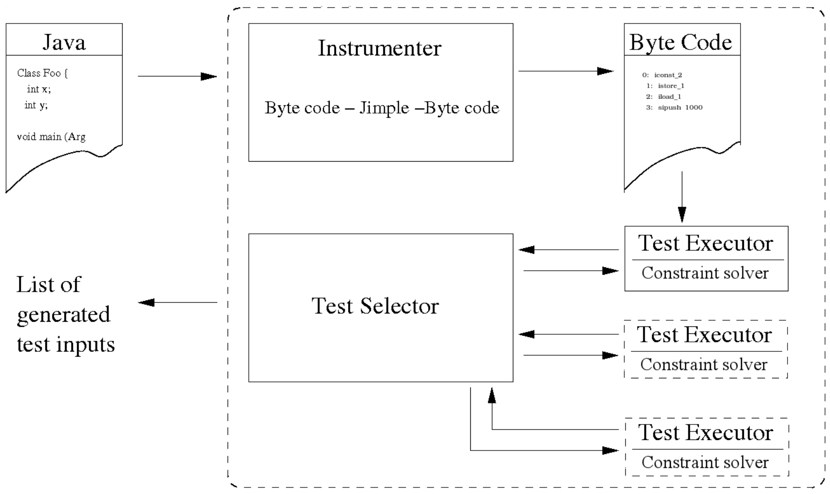

Fig. 1. The architecture of LCT

attempt to explore all distinct execution paths of the given program. These execution paths can be expressed as a *symbolic execution tree* which is a structure where each path from root to a leaf node represents an execution path and each leaf node has a path constraint describing the input values that force the program to follow that specific path.

The concrete execution in concolic testing provides the benefit that it makes available accurate information about the program state which might not be easily accessible when using only static analysis and allows the program to contain calls to uninstrumented native code libraries as symbolic values can always be approximated with concrete ones. Furthermore, as each test is run concretely, concolic testing reports only real bugs.

3 Tool Details

The architecture of LCT is shown in Figure 1 and it can be seen as consisting of three main parts: the instrumenter, the test selector and the test executors. To test a given program, the input locations are first marked in the code using methods provided by LCT. For example, `int x = LCT.getInteger()` generates an int type input value for `x`. After the input variables have been marked in the program, it is given to the instrumenter that enables symbolic execution by instrumenting the program using a tool called Soot [12]. The resulting program is called test executor. The test selector constructs a symbolic execution tree based on the constraints collected by the test executors and selects which path in the symbolic execution tree is explored next. The communication between the test selector and test executors is implemented using TCP sockets. This way it is easy to distribute the testing process. LCT reports uncaught exceptions as defects and the executed tests can also be archived as a test suite for offline testing.

LCT provides the option to use Yices [3] or Boolector [1] as a constraint solver. LCT uses linear integer arithmetic to encode the constraints when Yices is used and bit-vectors are used with Boolector. LCT has support for all primitive data types in Java as symbolic inputs with the exception of float and double data types as there is no native support for floating point variables in the used constraint solvers. LCT can also generate input objects that have their fields initialized as new input values in a similar fashion to [10].

3.1 Performing Symbolic Execution

To form symbolic constraints, it is necessary to know for each variable its symbolic value in addition to the concrete value. To track the symbolic values, LCT maintains a mapping from variables to their symbolic values. To update these memory maps and to construct path constraints, LCT follows closely the approach described in [5,10]. Every assignment statement in the program is instrumented with symbolic statements to update the symbolic memory maps. Every branching statement must also be instrumented with statements that create the symbolic constraints for both resulting branches so that the necessary path constraints can be constructed. A more detailed description of the instrumentation process can be found in [7].

The test selector maintains a symbolic execution tree based on the symbolic constraints generated by the test executors. LCT supports multiple strategies such as depth-first, breadth-first and randomized searches to explore the tree. For each test run, the test selector chooses an unexplored path from the symbolic execution tree. To explore this path, the test server sends the path constraint to a text executor that solves the constraint and uses the resulting values for a new test run. This way constraint solving is not done in a single centralized place which could cause a performance bottleneck.

3.2 Limitations

LCT has been designed for sequential Java programs. Multi-threading support is currently under development and will be released soon in version 2.0. There are also some cases where LCT can not obtain full path coverage for supported Java programs. LCT is not able to do any symbolic reasoning if the control flow of the program goes to a library that has not been instrumented (e.g., calling a library that has been implemented in a different programming language). Instrumenting Java core classes can also be problematic, therefore we have implemented custom versions of some of the most often required core classes to alleviate this problem. The program under test is then automatically modified to use the custom versions of these classes instead of their original counterparts. This approach can be seen as an instance of the twin

Benchmark	Paths	Runtimes / Speedups		
		1 test executor	10 test executors	20 test executors
AVL tree	3840	16m 57s	2m 6s / 8.1	1m 8s / 15.0
Quicksort (5 elements)	541	3m 11s	21s / 5.2	13s / 8.4
Quicksort (6 elements)	4683	28m 22s	3m 29s / 8.1	1m 39s / 17.2
Greatest common divisor	2070	11m 12s	1m 13s / 9.2	38s / 17.7

Table 1
Results of the experimental evaluation of the distributed architecture of LCT.

class hierarchy approach presented in [4].

LCT also does a similar non-aliasing assumption as the jCUTE tool. The code "`a[i] = 0; a[j] = 1; if (a[i] != 0) ERROR;`" is an example of this. LCT assumes that the two writes do not alias and, thus, does not generate constraint $i = j$ required to reach the `ERROR` label.

4 Experiments

We have evaluated LCT (version 1.1.0) and its distributed architecture by testing multiple Java programs so that varying number of test executors were running concurrently. The tests included: AVL tree (`http://cs-people.bu.edu/mullally/cs112/code/GraphicalAvlTree.java`), Quick sort (`http://users.cis.fiu.edu/~weiss/dsaajava/code/DataStructures/`) and Greatest common divisor (GCD) (`http://commons.apache.org/`). For AVL tree a test driver was used that called either add or remove methods nondeterministically with an integer marked as the input to the methods for five times. For Quick sort the test driver used the algorithm to sort a fixed size array of input integers and checked afterwards that the array had been sorted correctly. Finally, for GCD the test driver called the algorithm with integer inputs that were limited to be between 0 and 50.

The results of our experimental evaluation are shown in Table 4 that shows the number of execution paths explored, total run-times and speedups compared to running only one test executor. The computers used in the experiments had Intel Core 2 Duo processors running at 1.8 GHz together with 2 GB of RAM. For single test executor cases, the test selector and executor were run on the same machine. In our test environment, it made negligible difference whether this single test executor was run on the same or different machine as the test selector. For cases with multiple test executors, each computer was used to run two test executors at the same time while the test server was run on a separate computer (i.e., the test selector was able to utilize two cores). As the results show, LCT is able to take advantage of the increased resources efficiently with speedups ranging from 8.4 to 17.7 in the 20 test executor case. This is because individual test runs are highly independent. We have also used LCT to compare it with random testing in the context of Java Card applets. In this experiment LCT was used on a large number of mutants of the program under test. Further details of this experiment can be found in [8].

5 Conclusions

This paper introduces the LCT tool that is available together with source code and user guide from: `http://www.tcs.hut.fi/Software/lime/` as part of the LIME Interface Test Bench. We have demonstrated how the distributed nature of the tool makes concolic testing more scalable. We are currently adding support for the C language and multi-threaded Java programs to LCT.

References

[1] Brummayer, R. and A. Biere, *Boolector: An efficient SMT solver for bit-vectors and arrays*, in: *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, Lecture Notes in Computer Science **5505** (2009), pp. 174–177.

[2] Cadar, C., V. Ganesh, P. M. Pawlowski, D. L. Dill and D. R. Engler, *EXE: automatically generating inputs of death*, in: *Proceedings of the 13th ACM conference on Computer and communications security (CCS 2006)* (2006), pp. 322–335.

[3] Dutertre, B. and L. de Moura, *A Fast Linear-Arithmetic Solver for DPLL(T)*, in: *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*, Lecture Notes in Computer Science **4144** (2006), pp. 81–94.

[4] Factor, M., A. Schuster and K. Shagin, *Instrumentation of standard libraries in object-oriented languages: The twin class hierarchy approach*, in: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)* (2004), pp. 288–300.

[5] Godefroid, P., N. Klarlund and K. Sen, *DART: Directed automated random testing*, in: *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)* (2005), pp. 213–223.

[6] Godefroid, P., M. Y. Levin and D. A. Molnar, *Automated whitebox fuzz testing*, in: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008* (2008), pp. 151–166.

[7] Kähkönen, K., *Automated test generation for software components*, Technical Report TKK-ICS-R26, Helsinki University of Technology, Department of Information and Computer Science, Espoo, Finland (2009).

[8] Kähkönen, K., R. Kindermann, K. Heljanko and I. Niemelä, *Experimental comparison of concolic and random testing for Java Card applets*, in: J. van de Pol and M. W. 0002, editors, *SPIN*, Lecture Notes in Computer Science **6349** (2010), pp. 22–39.

[9] Pasareanu, C. S., P. C. Mehlitz, D. H. Bushnell, K. Gundy-burlet, M. Lowry, S. Person and M. Pape, *Combining unit-level symbolic execution and system-level concrete execution for testing NASA software*, in: *ISSTA*, 2008, pp. 179–180.

[10] Sen, K., "Scalable automated methods for dynamic program analysis," Doctoral thesis, University of Illinois (2006).

[11] Sen, K. and G. Agha, *CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools*, in: *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*, Lecture Notes in Computer Science **4144** (2006), pp. 419–423, (Tool Paper).

[12] Vallée-Rai, R., P. Co, E. Gagnon, L. J. Hendren, P. Lam and V. Sundaresan, *Soot - a Java bytecode optimization framework*, in: *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999)* (1999), p. 13.