Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 296 (2013) 253–259

www.elsevier.com/locate/entcs

LCT: A Parallel Distributed Testing Tool for Multithreaded Java Programs

Kari Kähkönen, Olli Saarikivi and Keijo Heljanko

Department of Information and Computer Science
School of Science
Aalto University
PO Box 15400, FI-00076 AALTO, Finland
{Kari.Kahkonen,Olli.Saarikivi,Keijo.Heljanko}@aalto.fi

Abstract

LIME Concolic Tester (LCT) is an open source automated testing tool that allows testing both sequential and multithreaded Java programs. The tool uses concolic testing to handle input values and dynamic partial order reduction (DPOR) combined with sleep sets to avoid exploring unnecessary interleavings of threads. The LCT tool has been designed for distributed use where the SMT constraint solving and test execution can be distributed to multiple processes on a network of workstations. In this paper we describe the architecture behind the tool and how it allows distributing concolic testing with DPOR and sleep set algorithms. This allows different execution paths of a given program to be tested in parallel. We evaluate the architecture and distributed algorithms of the tool on several Java benchmark programs.

Keywords: Concolic testing, distributed testing, symbolic execution

1 Introduction

Automated testing has the potential to improve reliability and reduce costs when compared to manually written test cases. One technique to automate testing is to use concolic testing which combines concrete and symbolic execution to explore different execution paths of a given sequential program. Concolic testing can be combined with dynamic partial order reduction (DPOR) and sleep set algorithms that allow the approach to be used to test multithreaded programs as well. Based on these algorithms, we have developed an open source tool called LCT (LIME Concolic Tester) that can automatically test both sequential and multithreaded Java programs. The tool has been designed for distributed use where a network of computers can be utilized to make the testing approach scale for larger programs than in the non-distributed case.

We have previously evaluated the distributed nature of our tool by testing single threaded programs in [7] and since then we have extended our tool to support multithreaded programs using the DPOR algorithm as described in [8]. The main

http://dx.doi.org/10.1016/j.entcs.2013.09.002

contributions of this paper are: (i) a tool oriented description of the distributed architecture of our system and the modifications needed to the DPOR and sleep set algorithms to make them usable in this architecture, and (ii) a new experimental evaluation of the distributed nature of our tool on multithreaded programs. In particular, the new experiments concentrate on cases where most of the test executions are generated due to different schedules that need to be explored. We show that even in this case the testing can be distributed as efficiently as with single threaded programs. The rest of the paper is structured as follows. Section 2 briefly describes concolic testing and dynamic partial order reduction algorithms, Section 3 gives an overview of LCT together with our modifications to the used algorithms, Section 4 covers the related work and Section 5 provides an experimental evaluation of the distributed architecture in the context of multithreaded programs.

2 Concolic Testing and Dynamic Partial Order Reduction

Concolic testing [6,9,4] (also known as dynamic symbolic execution) 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 at each branch point that specify the input values causing the program to take a specific branch. As an example, a program `x = x + 1; if (x > 0);` would generate constraints $input_1 + 1 > 0$ and $input_1 + 1 \leq 0$ at the if-statement given that the symbolic value $input_1$ is assigned initially to x.

A path constraint is a conjunction of symbolic constraints that correspond to each branch decision made in a given execution.To force a test execution to follow an unexplored execution path, a prefix of a previously explored path constraint is chosen and the last symbolic constraint in it is negated. To obtain concrete input values, path constraints are typically solved with SMT-solvers. The symbolic constraints form a symbolic execution tree and each test explores one path in this tree. As any two distinct subtrees of the symbolic execution tree can be explored independently, it is possible to parallelize the testing process efficiently. For more details on concolic testing, see e.g., [9].

For multithreaded programs the schedule affects the execution path as well. The nondeterminism caused by the thread interleavings can be handled in concolic testing by taking control of the scheduler and considering the schedule as an input to the system. To limit the number of thread interleavings that need to be explored, concolic testing can be combined with dynamic partial order reduction algorithms [5]. The basic idea behind these algorithms is to find transitions that are in race in the current execution and then introduce backtracking points to the execution tree such that the different interleavings of the transitions in race will eventually be explored.

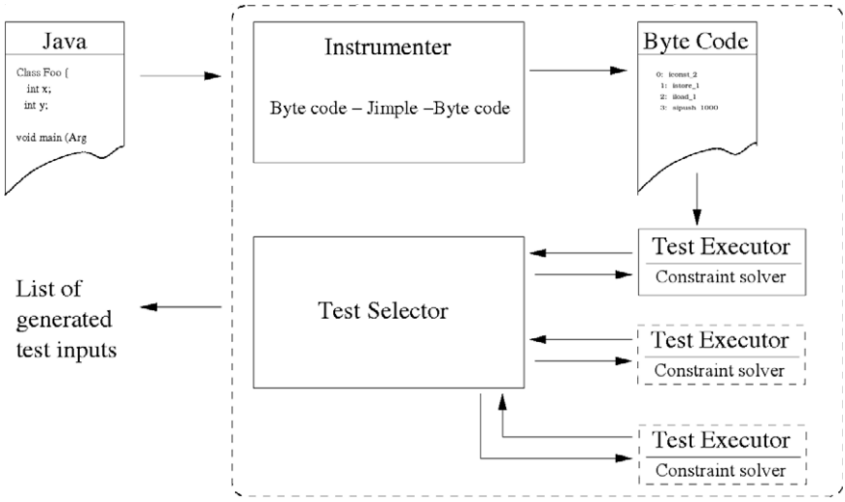

Fig. 1. The architecture of LCT

3 Tool Details

The architecture of LCT follows the client-server model and is shown in Figure 1. LCT consists of three main parts: the instrumenter, the server (test selector) and the clients (test executors). To test a given Java program, the input locations are first marked in the code. For example, `int x = LCT.getInteger()` indicates that an int type input will be generated for the variable x. After this the program is given to the instrumenter that modifies the program by adding new code to it that enables symbolic execution. For this step LCT uses a program transformation framework called Soot [10] and adds for most statements symbolic counterparts that perform the same operations symbolically. To make the instrumentation of Java programs easier, a given program is first translated into an intermediate language called Jimple that offers a simplified syntax. After the instrumentation the program is translated back to bytecode. The resulting program is called a test executor that works as a client. When the client is run, it sends information (e.g., constraints) generated during runtime to the server which in turn constructs a symbolic execution tree based on this information. When a client finishes a test execution, it requests new input values from the server. The server then chooses which path in the symbolic execution tree is explored next and sends the corresponding thread schedule and path constraint to the client which then solves it to obtain the concrete input values. This way the constraint solving is distributed to the clients and prevents the constraint solving from becoming a bottleneck for the parallelization of the testing process.

The communication between the server and clients is implemented using TCP sockets that makes it easy to distribute the testing to multiple workstations. The constraints generated during test executions are expressed in bitvector theory and Boolector [2] is used as the constraint solver. To avoid exploring unnecessary interleavings when testing multithreaded programs, the tool uses dynamic partial-order reduction and sleep set algorithms. In order to use these algorithms in our dis-

tributed setting, we have made some modifications to them that are described next.

3.1 Dynamic Partial Order Reduction and Sleep Sets in a Distributed Setting

DPOR is stateless in the sense that previously visited states are not needed for identifying races. However, for backtracking there does need to be a way to reach previous states. There are several ways to achieve this [5]. LCT uses re-execution of the program, as it is a natural fit for combining with concolic testing. This is because the path constraints in concolic testing encode sets of concrete states and even though a new path constraint shares a prefix with an old one, the inputs solved from the new constraint may not drive the program to any previously visited concrete state. Re-execution is a convenient way reach a concrete state that satisfies the new path constraint.

At the beginning of each test execution the client retrieves from the server a sequence of scheduling decisions to be re-executed. Backtracking points need not be added during the re-execution, as any backtracking points identified will already have been added by a previous test execution. Otherwise DPOR is run as normal during re-execution, meaning that the vector clocks and other bookkeeping data for identifying backtracking points are maintained.

To enable re-execution, the client sends each scheduling decision made to the server, which adds them to the execution tree and remembers the client's current position in it. The backtracking points DPOR identifies are then sent to the server as indices into the execution tree along the client's path together with a set of alternate operations that are to be explored from that state. On subsequent test executions these alternate operations are explored by supplying a client with the collected scheduling decisions up to the backtracking state with the alternate operation appended to the sequence.

In model checking the reduced state space explored by a partial order reduction algorithm must often satisfy a *cycle proviso*, which prevents operations from being ignored in all states of a cycle in the state graph. Implementing the cycle proviso in a parallel setting is challenging, although some solutions have been proposed [1]. However, because DPOR is a stateless method with an acyclic state space we can avoid the cycle proviso, allowing for easier parallelization. Using multiple concurrent clients together with the backtracking search performed by DPOR is straightforward: when one client discovers a backtracking point another one may start a test execution to explore it before the first one has finished. While no client side modifications are required to enable this, the server has to be properly synchronized.

Sleep sets can be combined with DPOR to provide additional reduction when DPOR fails to identify accurate sets of operations to explore from backtracking states. The sleep set algorithm is based on the observation that after an operation t has been explored from some state s, then after other operations independent with t are explored from s it is not necessary to explore t again. To this end we associate with each reached state a *sleep set*, which is a set of operations that are not executed from that state.

To compute sleep sets, when a state s' is explored from s, the *candidate sleep set* for s' is the union of the sleep set of s and the set of operations already explored from s. This candidate sleep set is then filtered to only include operations that are independent with the operation that was executed to reach s'. The sleep set of the initial state is empty.

Our setting presents two complications to implementing sleep sets: (i) only the server knows which operations have been explored from a given state and (ii) only the client knows the dependencies between operations. Therefore we split the implementation between the server and client as follows.

In the beginning of the execution the server sends the candidate sleep set for the state the client will reach once it has re-executed the sequence of operations sent by the server. When the client reaches the state at the end of the sequence and on each state after that, the sleep set it has received is filtered of dependent operations. Each new sleep set obtained this way is sent to the server. Both the client and server respect the sleep set when executing operations and selecting backtracking points to explore, respectively.

The detailed descriptions of the modifications to DPOR and sleep set algorithms that allow them to be used in a client-server setting can be found in [8].

4 Related Work

An alternative way to distribute concolic testing is to partition the symbolic execution tree in such a way that individual workers explore independent partitions of the tree. The partitioning can be done either statically or dynamically. As the shape of the symbolic execution tree is not known beforehand, static partitioning rarely results in optimal load balancing between the workers. Dynamic partitioning addresses this problem and provides excellent scalability to a large number of workers. Dynamic partitioning, however, requires a more complex implementation when compared to the synchronizing server approach used in LCT. See [3] for one approach based on dynamic partitioning. In [11] an approach to distribute DPOR using partitioning is presented. This approach provides excellent scalability to the number of workers but in some cases results in exploring a same schedule multiple times. The synchronizing server approach does not have this problem.

5 Experiments

To evaluate the distributed architecture of LCT (version 2.2.1), we have used it to test several multithreaded Java programs with varying number of test executor clients that were run concurrently. We have previously shown that the distributed architecture works well for single threaded programs. However, it is not directly evident that the use of DPOR generates enough open branches fast enough to keep a large number of clients busy. Therefore these experiments concentrate on cases where most of the test runs are generated due to backtracking requests of the DPOR algorithm.

Benchmark	Avg. paths 1 client	Avg. time 1 client	Avg. speedup 2 clients	5 clients	10 clients	20 clients[1]
Indexer (13)	671	285s	1.89	4.68	8.94	16.97
File System (18)	138	47s	1.92	4.55	8.88	14.91
Parallel Pi (5)	1252	250s	1.95	4.73	9.14	18.06
Synthetic 1 (3)	1020	176s	1.99	4.91	9.74	18.13
Synthetic 2 (3)	4496	783s	2.00	4.86	9.61	18.17

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

The Indexer and File System programs are from [5] where they are used to evaluate the DPOR algorithm. The Parallel Pi program implements a parallel algorithm for calculating the value of π. The synthetic programs are simple examples where a number of threads perform randomly generated sequences of shared variable accesses as well as local branching on input values. In the experiments, the server was run on 2.93GHz quadcore Linux workstation with 4GB of RAM. The clients were run mainly on 3.30GHz dualcore Linux workstations [1] with two clients per workstation.

The results of the experiments are shown in Table 1. As the order in which different thread interleavings are explored affects the performance of DPOR, different runs of our tool can result in different number of test runs for the same benchmark. To take this property of DPOR into account, each benchmark was run five times with a random initial thread schedule. The table shows the average number of execution paths explored and the number of seconds needed to test them in the case where only one client was used. For the cases where multiple clients were run concurrently, the table shows the average speedup obtained when compared to the single client case.

The results show that the architecture scales well at least up to 20 clients. This is because the time to run a single test execution, which consists of restarting JVM to initialize global state, solving paths constraints and running the program both concretely and symbolically takes significantly more time than the operations the server needs to do in a synchronized way. Furthermore, most of the time the number of open paths in the symbolic execution tree is large enough so that each client has work to do.

6 Conclusions

This paper introduces the LCT tool that is available together with source code from: http://www.tcs.hut.fi/Software/lime/ as part of the LIME Interface Test Bench. We have described the distributed architecture of the tool and our modifications to DPOR and sleep set algorithms required by the architecture. We have evaluated the distributed nature of the tool on several Java programs and shown that it improves the scalability of concolic testing of multithreaded programs.

[1] In the 20 client case, the additional ten clients were run on varying Linux workstations that were slightly faster or slower than the workstations used in the rest of the experiments. The performance differences were small to individual runtimes.

Especially, we have shown that the use of DPOR does not limit the search of new execution paths to be tested in such a way that a large number of parallel workers could not be utilized effectively.

Acknowledgement

This work has been 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), and Artemis-JU funded project RECOMP (Reduced Certification Costs Using Trusted Multi-core Platforms).

References

[1] Barnat, J., L. Brim and P. Rockai, *Parallel partial order reduction with topological sort proviso*, in: J. L. Fiadeiro, S. Gnesi and A. Maggiolo-Schettini, editors, *SEFM* (2010), pp. 222–231.

[2] 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.

[3] Bucur, S., V. Ureche, C. Zamfir and G. Candea, *Parallel symbolic execution for automated real-world software testing*, in: C. M. Kirsch and G. Heiser, editors, *EuroSys* (2011), pp. 183–198.

[4] 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.

[5] Flanagan, C. and P. Godefroid, *Dynamic partial-order reduction for model checking software*, in: J. Palsberg and M. Abadi, editors, *POPL* (2005), pp. 110–121.

[6] 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.

[7] Kähkönen, K., T. Launiainen, O. Saarikivi, J. Kauttio, K. Heljanko and I. Niemel, *LCT: An open source concolic testing tool for Java programs*, in: *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'2011)*, 2011, pp. 75–80.

[8] Saarikivi, O., K. Kähkönen and K. Heljanko, *Improving dynamic partial order reductions for concolic testing*, in: *Proceedings of the 12th International Conference on Application of Concurrency to System Design (ACSD'2012)* (2012).
URL http://users.ics.aalto.fi/osaariki/lct-improving-dpor.pdf

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

[10] 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.

[11] Yang, Y., X. Chen, G. Gopalakrishnan and R. M. Kirby, *Distributed dynamic partial order reduction based verification of threaded software*, in: D. Bosnacki and S. Edelkamp, editors, *SPIN*, Lecture Notes in Computer Science **4595** (2007), pp. 58–75.