Sophisticated Testing of Concurrent Programs

Zdeněk Letko
Brno University of Technology
Božetěchova 2, Brno
CZ 612 66, Czech Republic
Email: iletko@fit.vutbr.cz

Abstract

Search-based techniques were successfully applied to many different areas of testing but according to our knowledge there are no works that applies search-based techniques to testing of concurrent software, yet. This PhD paper describes plans and already achieved preliminary results with applying search-based techniques to testing of concurrent software. In particular, we plan to combine noise injection techniques for testing of concurrent software, various concurrency coverage measures, and several dynamic analyses with search-based optimization techniques.

1. Introduction

As concurrent programming is far more demanding than sequential, its increased use in the last decade have led to a significantly increased number of defects that appear in commercial software due to errors in synchronization. This stimulates a more intensive research in the field of detecting of such defects. Despite persistent efforts of wide community of researchers, a satisfiable solution of this problem for common programming languages like Java does not exist.

Finding concurrency defects in concurrent programs is a challenging task. The traditional testing approach does not work well here because concurrency defects often appear only under special conditions, e.g., when a specific interleaving of threads occurs. Concurrent programs are also much harder to analyze then sequential programs because one has to consider all possible interleavings and interactions of involved threads and therefore such analyzes suffer from exponential time requirements.

In this paper, we propose an approach that combines *dynamic analysis* which is able to precisely detect a concurrency defect along a given execution path, *noise injection technique* which allows us to influence thread interleaving, and *search-based technique* that is used to control noise generator with intention to maximize number of different interleavings we see during multiple executions. Our approach is sound if a sound dynamic analysis which does not produce false alarms is used and incomplete because usually we do not analyze all possible interleavings. On the other hand, our approach is able to analyze more different interleavings than traditional testing having similar time requirements and if suitable search technique is used, we analyze mostly those interleavings which are interesting to analyze.

The paper is organized as follows. The related works are briefly mentioned in the next section. The third section introduces the way how search-based techniques can be applied to concurrency testing. Then, the main directions of our research are pinpointed and finally, the already achieved preliminary results are shortly mentioned.

2. Related Works

Search-based Testing. Search-based testing applies metaheuristic search techniques to the problem of test data generation. In principle, the test adequacy criterion is encoded as a fitness function, and a search technique uses this function as a guide to achieve a maximal test adequacy. The search-based approach is very generic because different fitness functions can be defined to capture different test objectives.

Search techniques can be divided into local, global and hybrid techniques. The *local search techniques* [9], [5] seek for a better solution of the problem in the neighborhood of the currently known best solution. Such techniques provide a relatively good performance but can get stuck in a local optimum. The *global search techniques* [9], [5] seek for a better state in the whole state space. Such methods overcome the problem of local optima but usually need many more

states to be explored. Global search techniques are often implemented using evolutionary algorithms [9]. The third class of techniques, referred to as *hybrid search techniques* [5], tries to combine the previous two approaches to get rid of their disadvantages.

There have been published many works in the area of search-based testing in the past two decades. The approach has so-far been successfully applied to many different testing approaches covering functional, non functional, and state-based properties (c.f., e.g., structural, exception, stress, regression, and integration testing) [4], [5].

Finding Bugs in Concurrent Software. One of the most common approaches used for detecting concurrency bugs is *testing*, typically extended in some way to cope with the fact that concurrency bugs often appear only under very special scheduling. To increase chances of spotting a concurrency bug, various ways of *influencing the scheduling* are used (apart from running the same test many times). An example of this approach is random or heuristic noise injection used in the ConTest tool [2] or a systematic exploration of all schedules up to some number of context switches as used in the CHESS tool [10].

Testing can be improved by specific *dynamic analyses* which try to extrapolate the behaviour seen within a testing run and to warn about a possible defect even if such defect was not in fact seen in the testing run. Many dynamic analyses have been proposed for detecting special classes of bugs, such as data races [3], atomicity violations [7], and deadlocks [6]. These techniques may find more bugs than classical testing but, on the other hand, their computational complexity is usually higher and some of them can produce false alarms.

An alternative to testing and dynamic analyses is the use of various *static analyses* [11] or *model checking* [1]. Static analyses try to avoid execution of the given program or to execute it on a highly abstract level only. Model checking (sometimes viewed as a heavy-weight static analysis) tries to systematically explore all possible states of a model representing the analyzed program. These approaches are usually not precise enough and therefore produce false alarms or are too demanding and do not scale well.

Search-based Testing of Concurrent Software. According to our knowledge, search-based techniques have not been applied to concurrency testing yet. But, recently published work [12] describes application of a search technique to control state space exploration within a model checker with intention to explore areas that are more likely to contain errors. Our approach share the same idea of applying a search technique to focus on scenarios that more likely lead to an error but we focus on testing. Therefore, we do not need to construct model and handle state space of the tested application.

3. Search-based Concurrency Testing

The success of search-based software testing is based on two facts. First, the input space within which test data is sought is typically well defined but so large that it is usually infeasible to explore the whole input space. Second, the test goal can be expressed as a fitness function. The reason why the search-based testing has probably not been used for testing of concurrency is that there were not suitable definitions of input spaces and fitness functions. In the rest of this section, specifics of concurrency testing are briefly described and then definitions of input spaces and fitness functions are provided.

Specifics of Concurrency Testing. Testing is usually based on the idea that a test is executed and results (or partial results) are compared with the expected results. If the results obtained from the test correspond to the expected results, the test passes. If not, the test fails. Concurrency introduces non-determinism into the execution and one has to check that a program produces the same results for the same inputs regardless of which of the many legal orderings of operations within the program execution occurred. Therefore, testing of concurrent software is known to be very difficult since a test that has already passed in many executions may suddenly fail just because the order of operations during the execution differs from all previous executions. Concurrency testing tools therefore focus on observing many different legal orderings of program operations during *multiple executions of the same test* with the intention to find an ordering that makes the test fail.

Different legal orderings during multiple executions of a test can be achieved either by using a modified scheduler that schedules operations in an order that has not been seen yet or by the noise injection technique. The biggest problem of the modified scheduler approach is compatibility. There exist various implementations of Java and implementations of schedulers they use may differ, e.g., due to differences in underlying platforms. Therefore, we focus on the noise injection approach. In this approach, the application is instrumented and calls to a *noise maker* are injected at selected places. The noise maker tries to cause a delay (generally referred to as *noise*) when called. The selected places are those whose relative order among the threads can impact the result; such as entrances and exits from synchronized blocks, accesses to shared

variables, and calls to various synchronization primitives. The decisions whether to insert a noise can be random so different interleavings are attempted at each run or based on heuristics that try to reveal typical bugs for a particular synchronization construction.

Input Space. In our proposed approach, the input space for concurrency testing combines classic test inputs, noise maker inputs, noise parameter inputs, and an input dimension for the number of iterations to be considered. The task of identifying *classic test inputs* is the same as, for instance, for the structural testing and therefore existing approaches can be used to identify them.

The *noise maker inputs* influence the behavior of the applied noise maker. There could be two different kinds of such inputs: direct and indirect. The *direct noise maker inputs* tell the noise maker where exactly to put a noise. In this case, the search technique is used to identify a subset from the set of all possible locations in the code where it is suitable to cause a noise. The *indirect noise maker inputs* do not tell the noise maker where exactly to put a noise but set parameters influencing the procedure that decides whether to cause a noise at a particular location (e.g., a probability of causing the noise, enabling usage of some specific heuristic, etc.).

The *noise parameter inputs* define what kind of noise to use. The kind of noise is given by the type of language construction that is used by the noise maker to produce the noise and the strength of the produced noise. Finally, the *iteration count input* determines how many times the test must be performed because as was stressed above, a repeated execution of the same test with the same parameters can lead to different interleavings.

It is evident that what we do here is that we enlarge the classic input space by adding dimensions needed by the noise maker. This makes the input space much larger and hence more suited for advanced search-based techniques.

Fitness Functions. A fitness function is an objective function that prescribes the optimality of a solution and is highly dependent on the goal that a search technique tries to reach. Identifying suitable fitness functions for testing concurrency is one of our research goals and therefore we describe here only concurrency related measures on top of which the fitness functions can be built. As the most promising candidates for the inputs of fitness functions, we consider concurrency coverage measures, outputs of dynamic analyses, and classical measures like duration of the test and the number of bugs detected during a test execution.

To evaluate how well the concurrent behavior is tested, one needs some suitable coverage measures like those described in [13] (referred here as *concurrency coverage measures*). There exist several concurrency coverage models. These models usually combine code and concurrency coverage information. For instance, the *synchronization coverage* model [13] consists of tasks that describe all possible "interesting" behaviors of selected synchronization primitives. For instance, in the case of a synchronized block (defined using the Java keyword `synchronized`), the related tasks are: *synchronization visited*, *synchronization blocking*, and *synchronization blocked*. The synchronization visited task is basically just a code coverage task. The other two are reported when there is an actual contention between synchronized blocks—when a thread t_1 reached a synchronized block A and stopped because another thread t_2 was inside a block B synchronized on the same lock. In this case, block A is reported as blocked, and block B as blocking (both, in addition, as visited).

Further input information (and derived measures) can be obtained from outputs of various concurrency-related *dynamic analyses* executed along the test. Besides detecting bugs, these analyses can produce interesting data like, for instance, the set of variables that were really accessed by several different threads, the set of variables over which a race condition was detected, etc.

4. Proposed Research

A search-based testing techniques. The first direction of our research considers designing suitable fitness functions that are suitable for the needs of concurrency testing. The common use cases are to *detect a concurrency bug* (find a set of locations where to put a noise to cause an ordering of operations that enables detection of a bug), to *make the detected bug reproducible* (find a set of locations where to put a noise that cause an ordering making the bug to reveal), etc.

Reduction of input space. As was described in Section 3, the input space for concurrency testing is much larger than, for instance, in structural testing. However, since putting noise to some program locations does not make sense (e.g., locations that do not influence concurrency) and putting noise to some program locations can have the same effect as having the noise somewhere else, there is an open space for various optimizations that reduce the input space. Of course, most of the reductions must be problem-specific.

Propose a suitable incorporation of dynamic and/or static analyses. Finally, the third direction

combines testing with dynamic and static analyses in order to detect concurrency bugs. As was mentioned in Section 3, dynamic analyses can be used not only to detect concurrency bugs but also to produce inputs for the fitness function. Static analysis can be used to further analyze tests and its outputs can be used in search algorithms.

5. Already Achieved Results

We have implemented a generic platform for search-based testing called SearchBestie [8][1] that provides an environment for experimenting with search-based techniques as well as applying these methods in the area of testing. We instantiate the generic platform for concurrency testing, by linking it to the concurrency testing tool ConTest [2] from IBM. ConTest provides us with a tool for instrumenting Java bytecode, a configurable noise maker, code and concurrency coverage generators, and a listeners infrastructure that can be used to develop and apply dynamic analyses.

Our experiments with SearchBestie are in a preliminary stage. So far, we have mainly shown the concept of interconnection of SearchBestie and ConTest. However, in our experiments [8] focused on various specifics of concurrency testing, we have also obtained certain evidence that search-based techniques can be useful in concurrency testing. To prove our ideas, we compared random testing with a search-based approach which uses the simplest greedy search algorithm for finding a configuration of ConTest (indirect noise maker inputs) that provides as high concurrency coverage as possible. Since we get in some cases better results by such a primitive search than by the random approach, we believe that there is a big potential in applying better search algorithms and various heuristics in this area.

Acknowledgment

This work was partly supported by the Czech Science Foundation (proj. P103/10/0306 and 102/09/H042) and the internal BUT FIT grant FIT-10-1.

References

[1] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[2] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[3] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 245–255, New York, NY, USA, 2007. ACM Press.

[4] M. Harman. Automated test data generation using search based software engineering. In *AST '07: Proceedings of the Second International Workshop on Automation of Software Test*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[5] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 99(RapidPosts):226–247, 2009.

[6] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 110–120, New York, NY, USA, 2009. ACM.

[7] Z. Letko, T. Vojnar, and B. Křena. Atomrace: data race and atomicity violation detector and healer. In *PADTAD '08: Proceedings of the 6th workshop on Parallel and distributed systems*, pages 1–10, New York, NY, USA, 2008. ACM.

[8] Z. Letko, T. Vojnar, B. Křena, and S. Ur. A platform for search-based testing of concurrent software, to appear in proceedings of PADTAD '10.

[9] P. McMinn. Search-based software test data generation: a survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, 2004.

[10] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280. USENIX Association, 2008.

[11] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[12] J. Staunton and J. A. Clark. Searching for safety violations using estimation of distribution algorithms. *Software Testing Verification and Validation Workshop, IEEE International Conference on Software Testing, Verification, and Validation*, 0:212–221, 2010.

[13] E. Trainin, Y. Nir-Buchbinder, R. Tzoref-Brill, A. Zlotnick, S. Ur, and E. Farchi. Forcing small models of conditions on program interleaving for detection of concurrent bugs. In *PADTAD '09: Proceedings of the 7th Workshop on Parallel and Distributed Systems*, pages 1–6, New York, NY, USA, 2009. ACM.

1. The submitted version of the paper can be downloaded from: http://www.fit.vutbr.cz/~iletko/pub/padtad10.pdf