Testing of Concurrent Programs

Miguel Isabel*

Complutense University of Madrid, Madrid, Spain
miguelis@ucm.es

Abstract

Testing concurrent systems requires exploring all possible non-deterministic interleavings that the concurrent execution may have, as any of the interleavings may reveal erroneous behaviour. This introduces a new problem: the well-known state space problem, which is often computationally intractable. In the present thesis, this issue will be addressed through: (1) the development of new *Partial-Order Reduction Techniques* and (2) the combination of static analysis and testing (*property-based testing*) in order to reduce the combinatorial explosion. As a preliminary result, we have performed an experimental evaluation on the SYCO tool, a CLP-based testing framework for actor-based concurrency, where these techniques have been implemented. Finally, our experiments prove the effectiveness and applicability of the proposed techniques.

1998 ACM Subject Classification F.1.1 Models of Computation

Keywords and phrases Property-based Testing, Partial Order Reduction, Deadlock-Guided Testing, Deadlock Detection, Systematic Testing

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.18

1 Introduction

Due to increasing performance demands, application complexity and multi-core parallelism, concurrency is omnipresent in today's software applications. It is widely recognized that concurrent programs are difficult to develop, debug, test and analyze. This is even more so in the context of concurrent *imperative* languages that use a global memory (so called heap) to which the different tasks can have access. These accesses introduce additional hazards not present in sequential programs such as race conditions, data races, deadlocks, and livelocks. Therefore, software validation techniques urge especially in the context of concurrent programming.

Testing is the most widely-used methodology for software validation. However, due to the non-deterministic interleaving of tasks, traditional testing for concurrent programs is not as effective as for sequential programs. In order to ensure that all behaviors of the program are tested, the testing process, in principle, must systematically explore all possible ways in which the tasks can interleave. This is known as *systematic testing* [1] in the context of concurrent programs. Such full systematic exploration of all task interleavings produces the well known state explosion problem and is often computationally intractable (see, e.g., [2] and its references).

We consider actor systems [3], a model of concurrent programming that has been regaining popularity lately and that is being used in many systems (such as Go, ActorFoundry, Asynchronous Agents, Charm++, E, ABS, Erlang, and Scala). The Actor Model is having extensive influence on commercial practice. For example, Twitter has used actors for scalability, also, Microsoft has used the actor model in the development of its asynchronous

* Supervised by Elvira Albert & Miguel Gómez-Zamalloa.

Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016).
Editors: Manuel Carro, Andy King, Neda Saeedloei, and Marina De Vos; Article No. 18; pp. 18:1–18:5

Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

agents library. Actor programs consist of computing entities called actors, each with its own local state and thread of control, that communicate by exchanging messages asynchronously.

An actor configuration consists of the local state of the actors and a set of pending tasks. In response to receiving a message, an actor can update its local state, send messages, or create new actors. In the computation of an actor system, there are two non-deterministic choices: first which actor is selected, and then which task of its pending tasks is scheduled. The actor model is characterized by inherent concurrency of computation within and among actors, dynamic creation of actors, and interaction only through direct asynchronous message passing with no restriction on message arrival order. The interaction using non-preemptive asynchronous communication (i.e., the execution of a task cannot be interrupted by another one), together with the fact that there is no shared memory among different actors, facilitates in general the application of formal methods.

In particular, for the sake of systematic testing, one can assume [1] that the evaluation of all statements of a task takes place serially (without interleaving with any other task) until it releases the processor (gets to a return instruction).

Compared to multi-threaded systems, this reduces a lot the state explosion problem. However, a naive exploration of the search space to reach all possible system configurations still does not scale. The challenge of systematic testing of concurrent programs in general is to avoid as much as possible the exploration of redundant paths which lead to the same configuration and paths which are not leading to the satisfaction of some property.

2 Goals of the Research

The focus of this thesis project is the development and application of new techniques for actor-based systems testing, which allow to carry out the validation process efficiently (reducing the combinatorial explosion) and, therefore, applicable to large systems; and the adaptation of these techniques to other concurrency models and widely-used languages. In order to reduce the state space explored, we are going to address the problem from different angles:

- guiding the execution towards paths satisfying some property, and pruning the uninteresting ones (*property-based testing*),
- avoiding the exploration of redundant paths which lead to the same configuration (*Partial-Order Reduction techniques* [4]),
- applying these techniques in the context of symbolic execution, and
- developing a CLP-based framework incorporating all these new techniques.

3 State of the Art

The main goal of testing is bug detection. There are different kinds of bugs that one can aim at catching. In concurrent programs, *deadlocks* are one of the most common programming errors and, thus, a main goal of verification and testing tools is, respectively, proving deadlock freedom and *deadlock detection*. Therefore, one of the properties we could be interested in is deadlock detection, guiding the execution only towards those paths that might lead to deadlock, and prune those that we know certainly cannot lead to deadlock.

Static analysis and testing are two different ways of detecting deadlocks that often complement each other and thus it seems quite natural to combine them. Static analysis evaluates an application by examining its code but without executing it. In contrast, testing consists of executing the application for concrete input values. Since a deadlock can manifest

only on specific sequences of task interleavings, in order to apply testing for deadlock detection, the testing process must systematically explore all task interleavings.

The primary advantage of *systematic testing* [1, 5] for deadlock detection is that it can provide the detailed deadlock trace with all information that the user needs in order to fix the problem. However, there is an important shortcoming, as we said before, although recent research tries to avoid redundant exploration as much as possible [5, 6, 7, 8], the search space of systematic testing (even without redundancies) can be huge. This is a threat to the application of testing in concurrent programming.

Partial-order reduction (POR) [9] is a general theory that helps mitigate this combinatorial explosion by formally identifying equivalence classes of redundant explorations. Early POR algorithms were based on different static analyses to detect and avoid exploring redundant derivations. The state-of-the-art POR algorithm [10], called DPOR (Dynamic POR), improves over those approaches by dynamically detecting and avoiding the exploration of redundant derivations on-the-fly. Since the invention of DPOR, there have been several works [1] proposing improvements, variants and extensions in different contexts to the original DPOR algorithm.

The most notable one is [2] which proposes an improved DPOR algorithm which further reduces redundant computations ensuring that only one derivation per equivalence class is generated. Some of these works [1] have addressed the application of POR to the context of actor systems from different perspectives. The most recent one [2] presents the TransDPOR algorithm, which extends DPOR to take advantage of a specific property in the dependency relations in pure actor systems, namely transitivity, to explore fewer configurations than DPOR.

4 Current Status of the Research

The first way of reducing the combinatorial explosion that we have explored is by means of *property-based testing* and, in particular, guiding the testing process towards those paths leading to deadlock. Static analysis [15, 16, 17, 18] and testing [20, 21, 22, 23] are two different ways of detecting deadlocks that often complement each other, and, thus it seems quite natural to combine them.

In Integrated Formal Methods 2016, we have presented a seamless combination of static analysis and testing for effective deadlock detection (*deadlock-guided testing*) [11] that works as follows: an existing static deadlock analysis [12] is first used to obtain *abstract* descriptions of potential deadlock cycles. Now, given an abstract deadlock cycle, we guide the systematic execution towards paths that might contain a representative of that abstract deadlock cycle, by discarding paths that are guaranteed not to contain such a representative. The main idea is as follows: (1) From the abstract deadlock cycle, we generate *deadlock-cycle constraints*, which must hold in all states of derivations leading to the given deadlock cycle. (2) We extend the execution semantics to support deadlock-cycle constraints, with the aim of stopping derivations as soon as cycle-constraints are not satisfied, which do not lead to a deadlock of the given cycle. So those executions are stopped as soon as they are guaranteed not to lead to a state satisfying the deadlock-state constraints.

5 Experiments & Preliminary Results

We have implemented the SYCO tool [13], a testing tool for *concurrent objects* which is available at http://costa.ls.fi.upm.es/syco. The whole testing framework for this actor-based

language has been implemented by means of CLP. It consists of two basic parts: first, the imperative program is compiled into an equivalent CLP program and, second, systematic testing is performed on the CLP program by relying only on CLP's evaluation mechanisms.

In our approach, the whole testing process is formulated using CLP only, and without the need of defining specific operators to handle the different features. This, on the one hand, has the advantage of providing a clean and uniform formalization. And, more importantly, since systematic testing is performed on an equivalent CLP program, we can often obtain the desired degree of coverage by using existing evaluation strategies on the CLP side. This gives us flexibility and parametricity w.r.t. the adequacy criteria. SYCO is based on aPET [14], a Partial-Evaluation based TCG tool by symbolic execution.

The experiments have been performed using as benchmarks: (1) classical concurrency patterns containing deadlocks and (2) deadlock free versions of them, for which deadlock analyzers give false positives. We have compared the results obtained using a systematic testing setting and a deadlock-guided testing setting. Regarding the first set of benchmarks, we achieve significant gains w.r.t systematic testing and, thus, this proves the applicability, effectiveness and impact of *deadlock-guided testing*. Finally, for the examples that are deadlock free, we are also able to prove deadlock freedom for most cases where static analysis reports false positives.

6 Open Issues & Future Work

The techniques developed so far address dynamic testing, but our approaches would be applicable also in static testing, where the execution is performed on constraint variables rather than on concrete values. These possible extensions will require the use of termination criteria which provide the desired degree of coverage. Our CLP-based framework will facilitate the application of these extensions.

The development of improvements in precision of *Partial Order Reduction* techniques and the study of their applicability to concurrent languages containing blocking synchronization instructions also remains as future work in the thesis project.

Up to now, we have only studied the combination of deadlock analysis and testing in order to reduce the combinatorial explosion. However, other types of analysis could be used in this approach, for instance: resource analysis, if we want to guide the exploration towards those paths which are consuming above a threshold; termination analysis, towards paths that do terminate but the analysis is not able to prove it; and starving analysis, which could guide the testing process towards paths leading to a starving situation.

References

1 K. Sen and G. Agha. Automated Systematic Testing of Open Distributed Programs. In *Proc. FASE'06*, Lecture Notes in Computer Science 3922, pages 339–356. Springer, 2006.

2 S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. Trans-DPOR: A Novel Dynamic Partial-Order Reduction Technique for Testing Actor Programs. In FMOODS/FORTE, volume 7273 of Lecture Notes in Computer Science, pages 219–234. Springer, 2012.

3 G.A. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, MA, 1986.

4 Patrice Godefroid. Partial-Order Methods for the Verification of Concurrent Systems. An Approach to the State-Explosion Problem, volume 1032 of Lecture Notes in Computer Science. Springer, 1996.

5 M. Christakis, A. Gotovos, and K. F. Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In ICST'13, pages 154–163. IEEE, 2013.

6 P. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas. Optimal Dynamic Partial Order Reduction. In *Proc. of POPL'14*, pages 373–384. ACM, 2014.

7 E. Albert, P. Arenas and M. Gómez-Zamalloa. Actor- and Task-Selection Strategies for Pruning Redundant State-Exploration in Testing. In *FORTE'14*, Pages 49–65, Springer.

8 C. Flanagan and P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In Proceedings of POPL'05, pages 110–121. ACM, 2005.

9 Patrice Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In Proceedings of CAV, volume 531 of LCNS, pages 176-185. Springer, 1991.

10 Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In Proceedings of POPL'05, pages 110–121. ACM, 2005.

11 E. Albert, M. Gómez-Zamalloa, M. Isabel. Combining Static Analysis and Testing for Deadlock Detection. In Proceedings of iFM'16, pages 409–424.

12 A. Flores-Montoya, E. Albert, and S. Genaim. May-Happen-in-Parallel based Deadlock Analysis for Concurrent Objects. In *FORTE'13*, Lecture Notes in Computer Science 7892. 2013.

13 E. Albert, M. Gómez-Zamalloa. M. Isabel. SYCO: a Systematic Testing Tool for Concurrent Objects. In Proceedings of CC'16, pages 269–270.

14 E. Albert, P. Arenas, M.Gómez-Zamalloa, P. Y. H. Wong: aPET: a test case generation tool for concurrent objects. In Proceedings of ESEC/SIGSOFT FSE'13, pages 595–598.

15 E. Giachino, C.A. Grazia, C. Laneve, M. Lienhardt, and P. Wong. Deadlock Analysis of Concurrent Objects – Theory and Practice, 2013.

16 S. P. Masticola and B. G. Ryder. A Model of Ada Programs for Static Deadlock Detection in Polynomial Time. In *Parallel and Distributed Debugging*. ACM, 1991.

17 M. Naik, C. Park, K. Sen, and D. Gay. Effective Static Deadlock Detection. In Proceedings of ICSE, pages 386-396. IEEE, 2009.

18 R. Agarwal, L. Wang and S. D. Stoller. Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In *HVC*, Lecture Notes in Computer Science 3875. Springer, 2006.

19 P. Joshi, M. Naik, K. Sen, and Gay D. An Effective Dynamic Analysis for Detecting Generalized Deadlocks. In Proceedings of FSE'10, pages 327–336. ACM, 2010.

20 P. Joshi, C. Park, K. Sen, and M. Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In Proceedings of PLDI'09. ACM, 2009.

21 A. Kheradmand, B. Kasikci, and G. Candea. Lockout: Efficient Testing for Deadlock Bugs. Technical report, 2013.

22 S. Savage, M. Burrows, G. Nelson, P. Sobalvarro and T. E. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM TCS*, 1997.

23 K. Havelund, Using Runtime Analysis to Guide Model Checking of Java Programs, In Proceedings of the 7th International SPIN Workshop, Springer-Verlag, 2000.

24 Javier Esparza. Model Checking Using Net Unfoldings. Sci. Comput. Program., pages 151–195, 1994.