Abstractions, Semantic Models and Analysis Tools for Concurrent Systems: Progress and Open Problems
(Extended Abstract)

Gul Agha(✉)

University of Illinois at Urbana-Champaign, Champaign, USA
agha@illinois.edu
http://osl.cs.illinois.edu

Abstract. The growth of mobile and cloud computing, cyberphysical systems and the internet of things has arguably made scalable concurrency the central to computing. Actor languages and frameworks have been widely adopted to address scalability. Moreover, new tools that combine static and dyamic analysis are making software safer. This presentation describes the actor programming model and reasoning tools for scalable concurrency. As we scale up cyberphysical applications and build the internet of things, a key limitation of current languages and tools becomes apparent: the difficulty of representing quantitative and probabilistic properties and reasoning about them. The paper concludes by discussing some techniques to address reasoning about the behavior of complex scalable concurrent applications.

1 Introduction

The increasing use of web services, web applications, cloud computing, multicore computers, and sensor networks have made concurrency central to software development. The software industry is adapting to these changes by adopting concurrent programming as a key to achieving the performance goals of software product lines. Because many applications require scalable computing, the Actor model of concurrent computation [12] has naturally found increased applicability. A number of actor languages and frameworks are being used by software developers in industry today. Erlang [4], originally developed by Ericsson, has been used to implement Ericsson's backbone system, the Facebook Chat system [5], and the Vendetta game engine. Google released DART [2], an actor language for the *in-browser application* market. Actors in Scala are used by Twitter to program its backbone system [6]. Microsoft's Orleans actor framework [13] has hundreds of industrial users. Other applications written using one of these actor frameworks include WhatsApp [3], LinkedIn, the Halo 4 game engine [1], and the British National Health Service backbone [7]. We first describe the actor model and then discuss how we can test actor programs. We then discuss some techniques to scale up reasoning in order to increase confidence in large systems.

R. De Nicola and E. Kühn (Eds.): SEFM 2016, LNCS 9763, pp. 3–8, 2016.
DOI: 10.1007/978-3-319-41591-8_1

The Actor Model. An *actor* is an autonomous, interacting unit of computation with its own memory. An actor could be a computer node, a virtual process, or a thread with only private memory. If we were trying to model a shared memory computation using actors, we would have to represent each variable as a (simple) actor. But scalable systems require greater abstraction and the Actor model is more useful for modeling such systems. Each actor operates asynchronously rather than on a global clock. This models distributed systems where precise synchronization between actions of subcomponents is not feasible.

Actors communicate by sending each other messages. Because there is no action at a distance, these messages are by default asynchronous. Abstractions for synchronous messaging must be defined using asynchronous messages (e.g., [9]). Finally, an actor may create new actors. In software, this models creation of new concurrent objects and in operating system it can model process creation. In the case of hardware, it may model adding modules to an existing system. The concept of actors is closely related to that of autonomous agents [16].

Variants of the Actor Model. In real-time systems, sometimes a global clock is used [20]. In modeling networks, probability is added to transition [8,11]. However, as I noted in 2003, there is a need for more complex models of time than the current extremes of asynchronous or synchronous computation and communication. In physics, the notion of distance and the speed of light bounds the synchrony of events at different objects. Similarly, a richer model of concurrent systems should have a notion of "virtual" distance with which the degree of synchronization varies. However, this degree of synchronization need not be exact: the model needs to incorporate probability so that we can reason about the stochastic nature of message delivery, or about failures, or other hard to control variables.

2 Concolic Testing

The behavior of a computing system can be represented as a *binary tree* (a higher arity tree can be reduced to a binary tree), where the internal nodes of a computation tree represent decision points (resulting from *conditional statements* or from *nondeterminism*), and the branches represent (one or more) sequential steps. Note that the nondeterminism may be a way of modeling the results of different mechanisms: probabilistic transitions, scheduling of actors, or communication delays. For simplicity, we will call these nondeterministic choices *scheduling choices*. System verification is a process of examining a tree of potential executions to see if some property holds at each step.

The most common form of correctness reasoning is *testing*. Testing involves executing a system, which in turn requires picking some data values for the inputs and fixing an order for the scheduling choices. In order to make testing feasible, only a finite approximation of the potentially infinite behavior is considered. Such approximation is done in two ways: first, by restricting the domain of inputs. Second, by bounding the depth of the loops. The bound on the depth is typically arbitrary.

Of course, termination is undecidable, but more pragmatically, even though for many computations termination may be decidable, it may not be feasible to automatically determine what bound to use for a loop. Finally, only a small number of potential scheduling choices are considered.

Even with these restrictions, the space of possible behaviors is generally too large to examine fully. To overcome the problem, *symbolic testing* was proposed. The idea of symbolic testing is quite simple. Instead of using concrete values for data inputs, a symbolic value (variable) can be associated with each value. Then at each branch point, a constraint is generated on the variable. If there are values for which the constraint holds, the branch in which the constraint is true is explored, carrying the constrained forward. At the next branch point, the constraint on that branch is added to the constraint which has been carried forward, and again solved to see if there are values satisfying it. Similarly, if there are values satisfying the negation of the constraint, the other branch is explored. During the exploration, the symbolic state is checked to see if the constraints implied by the specification could be violated.

The problem with using symbolic testing is that the constraints involved are often unsolvable or computationally intractable. For example, if these constraints involve some complex functions or use of dynamic memory. In this case, it is unclear if a branch might be taken. When a constraint at a branch point cannot be solved, tools based on symbolic checking assume that both branches might be taken, leading to a large number of erroneous bug reports and causing tool users to ignore the tool.

To overcome this difficulty, *concolic testing* was proposed [15][1]. The idea is to simultaneously do concrete testing and symbolic testing on the same system. When a constraint cannot be solved, use randomization to simplify the constraint and find a partial solution. This increases coverage, but of course, does not provide completeness. We extended this concept to systems with dynamic memory, and to systems with concurrency, both the actor and the Java multi-threaded variety.

In case of concurrent systems, there are a large number of possible executions which are result in the same causal structure. This is because independent events (e.g. those on two different actors that have no causal relation) are simply interleaved. However, considering different orders may not affect the outcome. It is important to reduce or eliminate the number of such redundant executions as there are an exponential number of choices. Such reductions are called *partial order reduction*. A number of techniques, such as a macro-step semantics for actors have been developed to facilitate partial order reduction. We have also used concolic testing to dynamically detect interleavings that are redundant.

Concolic testing has been implemented in two tools which enable automatic unit (as opposed to system) testing of software written in C [24], JAVA [23], and actor programs [22]. Although the idea behind concolic testing is rather simple, concolic testing has proved very effective in efficiently finding previously undetected bugs in real-world software, in some cases, in software with a large

[1] Although the term first appears in [24].

user base which had gone through testing before being deployed. It has since been adopted in a number of commercial tools, including PEX from Microsoft[2].

Partial Order Reductions for Testing. In case of concurrent systems, there are a large number of possible executions which are result in the same causal structure [21]. This is because independent events (e.g. those on two different actors that have no causal relation) are simply interleaved. However, considering different orders may not affect the outcome. It is important to reduce or eliminate the number of such redundant executions as there are an exponential number of choices. Such reductions are called *partial order reduction*. A number of techniques, such as a macro-step semantics for actors have been developed to facilitate partial order reduction [10].

We have also used concolic testing to dynamically detect interleavings that are redundant. The macro-step semantics of actors is independent of the particularly library or language used: because the processing of a message by an actor is atomic, it can be done to an arbitrary depth before another actor takes a transition. Such properties have been used to provide a path exploration interface which can be common to all actor frameworks in Java [19].

Concolic Walk. When concolic testing encounters a constraint that a constraint solver cannot handle, it needs to do a heuristic search in the space defined by the subset of constraints that it can solve. Many heuristics have been proposed to address this problem. In fact, we have shown that it is often possible to essentially use a combination of linear constraint solving and heuristic search to address this problem [14].

3 Reasoning About Large-Scale Concurrent Systems

In large-scale concurrent systems, we are often interested in probabilistic guarantees on the behavior of the system, and in quantitative properties (energy consumption, throughput, etc.) of such systems.

Statistical Model Checking. One possibility is sampling the behavior of a system: in the real world, engineers often use Monte Carlo simulations to analyze systems. This process can be made more rigorous by expressing the desired properties of a system in a formal logic such as *continuous stochastic logic* (CSL). We have proposed using an approach we call *Statistical Method Checking* to verify properties expressed in a sublogic of CSL [25]. This work was extended to verify properties involving *unbounded untils* in [26]. The methods are implemented in a tool called VESTA [26] which has been used in a number of applications.

[2] http://research.microsoft.com/en-us/projects/pex/.

Verifying Quantitative Properties in Large State Spaces. The notion of global state also needs to be richer. In statistical physics, one often looks at the probability distribution over states to reason about aggregate properties such as temperature. Similarly, we can effectively measure certain quality of service parameters by representing the global state of systems not as a nondeterministic interleaving of the local states of components, but as a superposition of the individual states. We have developed a notion of state based on this concept. The notion of state as a probability mass function allows us to use a variant of linear temporal logic to express properties and to solve the model checking problem by using linear algebra for systems that are Markovian [17]. The technique is particularly useful for systems such as sensor networks [18].

Acknowledgements. The work on this paper has been supported in part by Air Force Research Laboratory and the Air Force Office of Scientific Research under agreement number FA8750-11-2-0084, and by National Science Foundation under grant number CCF-1438982.

References

1. Building Halo 4, a video game, using the actor model. http://www.infoq.com/news/2015/03/halo4-actor-model
2. Dart. http://www.dartlang.org
3. Erlang-powered Whatsapp. https://www.erlang-solutions.com/about/news/erlang-powered-whatsapp-exceeds-200-million-monthly-users
4. Erlang programming language
5. Facebook chat. https://www.facebook.com/note.php?note_id=14218138919
6. How and why Twitter uses Scala. https://www.redfin.com/devblog/2010/05/how_and_why_twitter_uses_scala.html
7. NHS to deploy Riak for new IT backbone. http://basho.com/posts/press/nhs-to-deploy-riak-for-new-it-backbone-with-quality-of-care-improvements-in-sight/
8. Agha, G., Gunter, C., Greenwald, M., Khanna, S., Meseguer, J., Sen, K., Thati, P.: Formal modeling and analysis of DOS using probabilistic rewrite theories. In: Workshop on Foundations of Computer Security (FCS 2005), vol. 20, pp. 1–15 (2005)
9. Agha, G., Houck, C.R., Panwar, R.: Distributed execution of actor programs. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua, D.A. (eds.) Languages and Compilers for Parallel Computing. LNCS, vol. 589, pp. 1–17. Springer, Heidelberg (1992)
10. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. **7**(1), 1–72 (1997)
11. Agha, G., Meseguer, J., Sen, K.: PMaude: rewrite-based specification language for probabilistic object systems. Electron. Notes Theoret. Comput. Sci. **153**(2), 213–239 (2006)
12. Agha, G.A.: ACTORS - A Model of Concurrent Computation in Distributed Systems. MIT Press Series in Artificial Intelligence. MIT Press, Cambridge (1990)
13. Bernstein, P.A., Bykov, S., Geller, A., Kliot, G., Thelin, J.: Orleans: distributed virtual actors for programmability and scalability. Technical report MSR-TR-2014-41, Microsoft Research (2014)

14. Dinges, P., Agha, G.: Solving complex path conditions through heuristic search on induced polytopes. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 425–436. ACM (2014)
15. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: Sarkar, V., Hall, M.W. (eds.) PLDI, pp. 213–223. ACM (2005)
16. Jamali, N., Thati, P., Agha, G.A.: An actor-based architecture for customizing and controlling agent ensembles. IEEE Intell. Syst. **2**, 38–44 (1999)
17. Kwon, Y., Agha, G.: Verifying the evolution of probability distributions governed by a DTMC. IEEE Trans. Softw. Eng. **37**(1), 126–141 (2011)
18. Kwon, Y., Agha, G.: Performance evaluation of sensor networks by statistical modeling and Euclidean model checking. ACM Trans. Sensor Netw. **9**(4), 39:1–39:38 (2013)
19. Lauterburg, S., Karmani, R.K., Marinov, D., Agha, G.: Basset: a tool for systematic testing of actor programs. In: Roman, G.-C., Sullivan, K.J. (eds.) 2010 Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 7–11 November 2010, Santa Fe, NM, USA, pp. 363–364. ACM (2010)
20. Ren, S., Agha, G.: RTsynchronizer: language support for real-time specifications in distributed systems. In: Gerber, R., Marlowe, T.J. (eds.) Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCT-RTS 1995), 21–22 June 1995, La Jolla, California, pp. 50–59. ACM (1995)
21. Sen, K., Agha, G.: CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)
22. Sen, K., Agha, G.: Automated systematic testing of open distributed programs. In: Baresi, L., Heckel, R. (eds.) FASE 2006. LNCS, vol. 3922, pp. 339–356. Springer, Heidelberg (2006)
23. Sen, K., Agha, G.: A race-detection and flipping algorithm for automated testing of multi-threaded programs. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 166–182. Springer, Heidelberg (2007)
24. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for C. In: Wermelinger, M., Gall, H. (eds.) Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 5–9 September 2005, Lisbon, Portugal, pp. 263–272. ACM (2005)
25. Sen, K., Viswanathan, M., Agha, G.: Statistical model checking of black-box probabilistic systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004)
26. Sen, K., Viswanathan, M., Agha, G.: Vesta: a statistical model-checker and analyzer for probabilistic systems. In: Second International Conference on the Quantitative Evaluaiton of Systems (QEST 2005), 19–22 September 2005, Torino, Italy, pp. 251–252. IEEE Computer Society (2005)