2014 13th International Symposium on Parallel and Distributed Computing

Actors Programming for the Mobile Cloud

(Invited Talk)

Gul Agha
Department of Computer Science
University of Illinois at Urbana-Champaign
http://osl.cs.uiuc.edu

***Abstract*—Actor programming languages provide the kind of inherent parallelism that is needed for building applications in the mobile cloud. This is because the Actor model provides encapsulation (isolation of local state), fair scheduling, location transparency, and locality of reference. These properties facilitate building secure, scalable concurrent systems. Not surprisingly, very large-scale applications such as Facebook chat service and Twitter have been written in actor languages. The paper introduces the basics of the actor model and gives a high-level overview of the problem of coordination in actor systems. It then describes several novel methods for reasoning about concurrent systems that are both effective and scalable.**

I. Introduction

The Actor model is a universal model of concurrent computation in distributed systems. Because the model was developed from its inception to represent concurrent computations, it scales naturally. This has made it a natural fit for large applications in the cloud. Consider a chat room application; the application is typical of a number of cloud-based web applications that involve distributed computation, such as social media sites and popular online games. Not surprisingly, the most successful scalable chat room application (involving tens of millions of users), Facebook, has been written in Erlang, an actor language. As *Facebook Engineering* observed in their commentary on this choice:[1]

> "..the actor model has worked really well for us, and we wouldn't have been able to pull that off in C++ or Java. Several of us are big fans of Python and I personally like Haskell for a lot of tasks, but the bottom line is that, while those languages are great general purpose languages, none of them were designed with the actor model at heart."

Similarly, Twitter is written in Scala. As Alex Payne, the pioneering developer of Twitter explained in a talk entitled "*How and Why Twitter Uses Scala*:"[2]

> "When people read about Scala, it's almost always in the context of concurrency. Concurrency can be solved by a good programmer in many languages, but it's a tough problem to solve. Scala has an Actor library that is commonly used to solve concurrency problems, and it makes that problem a lot easier to solve."

[1]https://www.facebook.com/notes/facebook-engineering/chat-stability-and-scalability/51412338919

[2]http://blog.redfin.com/devblog/2010/05/how_and_why_twitter_uses_scala.html

Actor programming languages provide the kind of inherent parallel scalability needed for such applications. Morever, the actor model facilitates mobility, providing opportunities for improved parallel performance and enabling ways to reduce energy consumption. In some ways, actors simplify reasoning about concurrent systems: they provide a macro-step semantics, dramatically reducing the number of possible computations that need to be considered. Still, reasoning about large concurrent systems remains a challenge. After describing the actor model, I will give a flavor of reasoning techniques that we have developed to facilitate reasoning about actor systems.

The goal of this high-level overview is to provide some intuitions about actor programmig and the state of reasoning about scalable concurrent systems. It should serve as a starting point to the relevant literature. The overview is not comprehensive. In particular, I do not discuss topics related to compiling actor languages, actor runtimes, work load management, garbage collection, distribution strategies, etc. (For some of these topics, see [15], [29], [41]). Moreover, I have focused on research in my own research group to the exclusion of much other good research.

II. The Actor Model

An *actor* is an autonomous, interacting unit of computation with its own memory [5], [1]. An actor could be a computer node, a virtual process, or a thread with only private memory. The actor model is universal: if we were trying to model a shared memory computation using actors, we would represent each variable as an 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 components is not feasible.

At the semantic level, an actor language provides for concurrent execution of programs, extending a sequential programming language with three primitive concurrency operators:

- *send* transmits a message to a specified actor. The message will be buffered in a *mail queue* at the destination until the actor is ready to process it. Each actor has a unique *mail address* which is used to specify a target for communication. Mail addresses may also be communicated in a message, allowing for a dynamic communication topology.

978-0-7695-5265-1/14 $31.00 © 2014 IEEE
DOI 10.1109/ISPDC.2014.31

- *new* is used to dynamically create an actor (with a fresh address) using the specified actor behavior (class name) and parameters.
- *become* marks the end a method, the actor can now process the next message in its queue.

The ability to create new actors facilitates the representation of dynamic applications such as our example chat system where new users may be created. The ability of actors to communicate mail addresses of actors enables actions such as join or leave a chat room. Moreover, natural extensions such as the ActorSpace model [2] allow (potentially overlapping) groups of actors to be defined, facilitating multicast in a dynamically changing environment.

It is important to note that these concepts are not tied to any specific programming language: a sequential language may be extended with object style encapsulation (using records or closures) and these three concurrency primitives. Actor languages enable us to provide a representation which abstracts execution details such as placement, scheduling, name resolution, buffering, etc. An actor language may be used as a high-level concurrent shell to provide interoperability between heterogeneous components (written in different languages and running on different types of platforms).

Actor semantics provides encapsulation (isolation of local state), fair scheduling, location transparency (location independent naming), locality of reference (facilitating security), and transparent migration. These properties enable compositional design and simplify reasoning [4], and improve performance as applications and architectures scale [16]. For example, because actors communicate using asynchronous messages, an actor does not hold any system resources while sending and receiving a message. This is in contrast to the shared-memory model where threads occupy system resources such as a system stack and possibly other locks while waiting to obtain a lock. Thus actors provide failure isolation while potentially improving performance.

Asynchronous communication between actors reduces synchronization overhead and network traffic, increasing system throughput. Because actors are self-contained and location independent, dynamically tuning performance by relocating server applications becomes easier. For example, the load of computers on a cloud may be dynamically balanced by remote creation, replication, or migration of applications. Alternatively, specific policies may be separately specified and used to facilitate trade-offs such as availability versus consistency.

A. Synchronization and Coordination

While the basic actor model is defined in terms of asynchronous messaging, the order in which messages are processed do affect the behavior of an actor. In many applications, application programmers want to prune some of the nondeterminism by restricting the possible orders in which messages are processed. Two commonly used abstractions that constrain the message order are *request-reply messaging* and *local synchronization constraints*. These abstractions can be efficiently implemented in terms of using actors [16]. Local synchronization constraints capture ordering requirements (cf. session types [9] projected onto an individual actor). Such constraints can be expressed in a language declarative language.

Besides local synchronization constraints which specify the acceptable ordering of messages at a given actor, there may be constraints on the order in which messages are processed at different actors. For example, a constraint on `Server` may specify that a chat room must exist before another user actor can join that chat room. Thus a message creating the chat room must be processed before requests to join it can be processed.

Such requirements can be also be expressed in a constraint language and translated into constraints such as precedence and atomicity between actions at different actors [12]. It turns out that dynamically imposed actor-level constraints can themselves lead to problems as re-configuration privileges may be abused by malicious or faulty actors. One way to address this is by scoping rules which constrain ordering with respect to a group of actors but do not affect messages from other actors [11]. More generally, interaction policies can be treated as first-class objects [38].

Constraints between actors can be context dependent. In some contexts, strict consistency requirements are essential, in others, eventual consistency suffices. Other systems may tolerate speculative actions and rollback. The translation of constraints can be customized in an using reflective meta-actors (e.g. [7], [6]). Satisfying requirements for timeliness of response, security, performance, energy use or other QoS parameters, often requires careful mapping of computations to underlying resources [30]. One can reason about the behavior of such systems in a two-level semantics (cf. [42]).

Mobility introduces further complexity. Actors may be executed on the client platform or on a server. The client platform may be a mobile device with limited computation capabilities or energy supply. Moreover, there may be security and bandwidth consideratioons. This becomes particularly relevant in applications on the mobile cloud, or mobile devices connecting to the cloud. One useful consequence of using the actor model is that it provides the freedom to execute and migrate individual actors seemlessly [14]. This property can help improve efficiency through dynamic placement and migration [30].

Mobility can also be useful in deciding where to execute based on contexts that are also dynamic: a rule-based language can be used to specify policies that govern the placement of actors. The application of the rules may depend on computational capabilities of platforms, currently available bandwidth, security requirements, etc. Actors may need to be created on different platforms based on the policies expressed in the rules [8]. Similar considerations arise in implementing actor languages for sensor networks [26].

Flexibility in placement and migration of actors can also facilitate in reducing energy consumption in a computation. Specifically, building distributed systems with processors which have a tunable frequency, we can provide additional opportunities to use parallelism to reduce energy consumption. This involves a trade-off between lower energy consumption for computation resulting from the use of a larger number of energy efficient processors (which operate at a lower frequency), and the greater energy required for communication because of increased parallelism. This trade-off has to be balanced by the desired performance requirements, and requires an analysis of the structure of an algorithm and the characteristics of the architecture it is executed on [19], [18], [17]

B. Variants of the Actor Model

Variants of the actor model are sometimes used. For example, in real-time systems, sometimes a global clock is used [31]. In cyber-physical systems, a probabilistic programming language with continuous variables is needed [20]. How these modifications affect the semantics is a complex question.

Although some variants have been developed, I have long believe there is a need for work on a more fundamental model of concurrency for cyber-physical systems. Current models use either asynchronous or synchronous actors (processes) 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, the degree of synchronization need not be exact (contrary to physics): in computation, 'distance' is a virtual construct and may be a probabilistic notion. The work in [20] was a small step in that direction.

III. Reasoning about Actor Systems

The problem of verifying or checking the correctness of concurrent systems has long been a difficult one. With the development of cloud computing, the importance of addressing this problem continues to grow. I briefly describe some methods that have been developed at Illinois which address this problem. In my view, these methods, while useful, remain rather rudimentary. The goal of my research has been to develop methods that are more effective as well as scalable. I have tried to look at the problem of reasoning about parallel and distributed systems from different perspectives.

A key difficulty in verifying concurrent systems (such as that on the cloud) is the large number of states that such systems can possibly be in. In sequential systems, the complexity of the verification problem is a result of the indeterminacy in the data inputs. In concurrent systems, this is compounded by the indeterminacy resulting from asynchrony. Observe that asynchrony is not an artefact of a particular model: it is natural in distributed systems because a synchronous model of time would require a level of fine-grained synchronization between actors that is prohibitively expensive. Because checking every possible state that a system may transition to is generally infeasible, *testing* remains the predominant technique applied to software systems.

A. Improving 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 such as probabilistic transitions, scheduling of actors, and communication delays. For simplicity, we will call these nondeterministic *scheduling choices*. System verification is a process of examining a tree of potential executions to see if some property holds at each node of the tree (state of a system).

There are many reasons why a computation tree may be infinite. For one, the domain of inputs to a system may be unbounded. For another, infinite loops (or recursive calls) are unfolded in the tree and these loops can be non-terminating. Then there is nondeterminism caused by concurrency; scheduling choices naturally give rise to the possibility of unbounded postponement. Even if the number of potential paths is very large, it is difficult to simply explore all states a system may be in.

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. For sequential programs, 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. In case of concurrent programs, the approximation involves considering only a small subset of potential scheduling choices.

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, the idea of concolic testing (first called DART) was proposed [13]. The idea is to simultaneously do concrete testing and symbolic testing on the same system. When a constraint cannot be solved, use simplification to deal with the constraint and find a partial solution space. This increases coverage, but of course, does not provide completeness. We extended this concept to systems with dynamic memory in C programs [34], and to systems with concurrency, both the actor [32] and the Java multi-threaded variety [33].

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 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[3].

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 [4]. 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 implemented in a path exploration tool (*Basset*) which provides a common platform for all actor frameworks in Java [27].

B. Runtime Verification

Systems may have specific requirements that should be checked. In this case, monitors can be inserted into the code. For example, if a safety condition such as $x \geq 100$ should always be true, such a condition can be turned into a conditional test and inserted in the right places in the code. This monitoring process is sometimes called *runtime verification* because one actually checks the system while it is executing. We have worked on methods for specifying safety properties and automatically generating and inserting monitors for them in a piece of code. Some researchers have proposed using monitors also in deployed code and allowing them to throw exceptions.

In distributed systems, properties ordering the state of actors must be expressible in causal terms [36]. For example, let p be the property that an actor a_1 is in a state σ_1 only if some other actor a_0 was previously in state σ_0. p cannot necessarily be checked. This is because all knowledge is local: it may be that actor a_0 was in state σ_0 but no other actor knows about it. Moreover, given the absence of global time, the ordering between the transitions on the local states of different actors is not total. So information about a state change must be communicated. Thus properties, much as in modern physics, must be expressed in terms of *observers* and what they know. This leads to *epistemic (or knowledge) distributed temporal logics*. For example, while a property of the form p may not be meaningful, the following property p' would be: an actor a_1 transitions to a state σ_1 only if it knows that some other actor a_0 was previously in state σ_0.

There is an interesting consequence of distributed monitoring. If we can infer causality, it is possible to *predict* violations of safety properties (bugs) even if they have not occurred in a particular execution. Consider the following scenario. Suppose the actor a_1 knows that the actor a_0 was previously in state σ_0 before it transitions to state σ_1. Thus the property p' holds. However, if a_1 didn't (directly or indirectly) use this fact in transitioning to state σ_1, we know that there *could* have been a violation of our safety condition. In case of multithreaded and real-time systems, where a global clock is reasonable to assume, it is meaningful to express properties implying a global order of the events. In this case, by tracking causal information, one can look at all causally consistent shuffles of a single execution to do predictive monitoring [35].

C. Learning-based Verification

Although testing (and model checking) may use some optimizations to avoid checking redundant states or paths, these techniques explore paths or states one at a time. We have worked with a different idea. Suppose we could learn a model of a system from its sample executions or *traces*. Now we could take this model and intersect it with a model for the negation of the specification. If the intersection of the two models is empty, we know that the system satisfies its specification. If not, we have a counter-example, a trace which shows that the system has a bug.

As one might imagine, it is not easy to come up with the model of a system, where executing the model yields the same result as the system. In fact, it is not possible for an arbitrary system whose model has infinite state space. However, even though the state space is infinite in systems, it is often used in a relatively "simple" way. In particular, many concurrent protocols can be represented with finite state automata which is communicating with first-in first-out (FIFO) queues. It turns out that it is possible to learn such a model using computational learning theory.

Two techniques have been developed for figuring out the model of a system. These techniques assume that the model is a finite state automaton and they are guaranteed to find an equivalent automaton. The first technique, called *passive learning*, works by giving the learner some example elements or strings that are accepted by the automaton (*positive samples*), and some examples of strings which are not (*negative*

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

samples). The learner then hypothesizes a model of the system. This model is refined using larger and larger samples until the hypothesis stops changing.

The second learning technique, called *active learning*, allows the learner to make queries that an oracle can answer. These queries can be of two sorts: "Is a given string accepted by the automaton?" and "Is the model that has been hypothesized by the learner equivalent to the real automaton?"

As I mentioned earlier, by executing a system, we can get traces which are positive samples. The difficult questions are how to get negative samples, answer queries about particular traces, or answer equivalence queries. I will describe our solution for passive learning [40]. It turns out to be more a little more complicated in case of active learning but that technique may be even more useful [39].

The insight which helps us get negative samples is as follows. We can store a *witness* for each reachable state; for example, this witness could be the path taken (i.e., the particular transitions made) to arrive at the state. Now a negative example is an invalid state, witness pair. For passive learning, when the learner makes a hypothesis, we proceed to intersect the proposed automaton with the negation of the specification. If the intersection is non-empty, we have a state, witness pair which doesn't meet the specification. Now an important property of the hypotheses made by the learner using passive learning is that they may be *overapproximations*, but will not be underapproximations. In other words, they may accept strings that are not accepted by the original automaton, but will not reject strings that are. So given the example which does not meet the specification, we do not know if it is actually a string in the actual automaton.

This is where the witness becomes important: we simply execute our system using the witness. If the example turns out to be a valid counter-example, we have found a bug. If not, we have a negative sample with which to refine our hypothesis. As you may have noticed, this approach does not actually require that our system be equivalent to a finite state automaton: as long as there is such an automaton which can overapproximate the system's behavior and meet the specification, we would terminate.

The use of learning provides a powerful technique. We have implemented a tool which uses this technique and shown that it can be used to reason about a number of concurrent protocols. Sometimes our learning technique is much faster than model checking in finding bugs or in proving a system correct. Sometimes it is not. However, the work is in an early stage and we do not understand its full range of applicability.

D. Euclidean Model Checking

Typically, we model the global state of a system as the cross product of individual states of actors in the system. This leads to an explosion in the number of potential states. Suppose we have a thousand nodes, each of which may be in one of 5 states. This means a possible 5^{1000} global states. However, if we are interested in certain sorts of global behavior–aggregate quantitative properties–it may not be necessary to consider all these global states. For example, suppose associated with each state is the amount of energy a node consumes when in that state (such an associated value mapping is called the *reward function* of the state). Now, if we have a frequency count of the nodes in each state, we can estimate the total energy consumed by the system. This suggests a model where the global state is a probability mass function vector representing the distribution of local states. In the above example, the size of the vector would be 5, one element for each possible state of a node. Each element of the vector represents the probability that each of the nodes in the system is in the particular state corresponding to element in the vector.

Given transitions between the global states, we can also compute how much energy has been consumed up to some point in time. Note that using such a global state assumes a certain symmetry (at least as a statistical approximation). We have also explored cases where there may be more than one type of node in a system (with its own associated Markovian behaviors). We have defined a temporal logic *iLTL* which can be used to write specifications where the global state of system is defined by such vectors [24].

In case the behavior of the system can be modeled by Discrete Time Markov Chains, we have developed a technique called *Euclidean model checking* [23]. Euclidean model checking works with transformations of probability distributions to check if the system satisfies some aggregate quantitative property. This provides an effective alternate to Monte Carlo simulations over possible initial states. We have used it to verify properties such communication bandwidth used, maximum expected queue lengths, energy consumption in sensor networks (e.g. [21], [23], and software reliability of many threaded concurrent software (i.e., software for servers) [22], [25].

E. Statistical Model Checking

A different technique we have developed has formalized properties of systems which evolve probabilistically. The idea is to to check formal specifications in a probabilistic Computation Tree Logic (CTL) based on Monte Carlo simulations. The system is treated as a blackbox and each run serves as a bernoulli trial. The key challenge is characterizing the behavior of the system with respect to the property. We do this by computing the probability that we would get the observed behavior if the specification was incorrect, based on the intervals of values that are outside the specification [37]. Statistical model checking has been used in a large number of applications. For example, it has been applied to mixed-analog circuits [43], cyberphysical systems [10], biological systems [28] and to validate protocols for security against DoS attacks [3].

IV. Summary and Conclusions

I have given a very broad view of the developments in programming concurrent systems using actors, emphasizing the challenges of scale that are inherent as we move to cloud computing and, even more broadly, to the Internet of

Things. While actors are a good foundation for providing scalable concurrency, we still need to think of novel ways orchestrating actors and reasoning about them. There will not be a single technique that can address all the aspects: it is a multi-faceted problem. There will be multiple language paradigms, for example, for orchestrating actors, and multiple reasoning techniques enabling us to reason about different types of properties.

Acknowledgements

This research has been supported in part by the National Science Foundation under grants CNS 10-35773, NSF CPS 10-35562; the Army Research Office under contract W911NF-09-1-0273, and Air Force Research Laboratory and the Air Force Office of Scientific Research, under agreement number FA8750-11-2-0084.

References

[1] G. Agha. Concurrent object-oriented programming. *Commun. ACM*, 33(9):125–141, 1990.

[2] G. Agha and C. J. Callsen. Actorspaces: An open distributed programming paradigm. In M. C. Chen and R. Halstead, editors, *PPOPP*, pages 23–32. ACM, 1993.

[3] G. Agha, C. Gunter, M. Greenwald, S. Khanna, J. Meseguer, K. Sen, and P. Thati. Formal modeling and analysis of dos using probabilistic rewrite theories. In *Proceedings of International Workshop on Foundations of Computer Security, Chicago, IL*, pages 91–102, 2005.

[4] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *J. Funct. Program*, 7(1):1–72, 1997.

[5] G. A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990.

[6] M. Astley and G. Agha. Customizaton and compositon of distributed objects: Middleware abstractions for policy management. In *SIGSOFT FSE*, pages 1–9. ACM, 1998.

[7] M. Astley, D. C. Sturman, and G. Agha. Customizable middleware for modular distributed software. *Commun. ACM*, 44(5):99–107, 2001.

[8] P.-H. Chang and G. Agha. Towards context-aware web applications. In J. Indulska and K. Raymond, editors, *DAIS*, volume 4531 of *Lecture Notes in Computer Science*, pages 239–252. Springer, 2007.

[9] M. Charalambides, P. Dinges, and G. Agha. Parameterized concurrent multi-party session types. In N. Kokash and A. Ravara, editors, *FOCLASA*, volume 91 of *EPTCS*, pages 16–30, 2012.

[10] E. M. Clarke and P. Zuliani. Statistical model checking for cyber-physical systems. In T. Bultan and P.-A. Hsiung, editors, *ATVA*, volume 6996 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2011.

[11] P. Dinges and G. Agha. Scoped synchronization constraints for large scale actor systems. In M. Sirjani, editor, *COORDINATION*, volume 7274 of *Lecture Notes in Computer Science*, pages 89–103. Springer, 2012.

[12] S. Frølund and G. Agha. A language framework for multi-object coordination. In *ECOOP*, pages 346–360, 1993.

[13] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.

[14] C. R. Houck and G. Agha. Hal: A high-level actor language and its distributed implementation. In *ICPP (2)*, pages 158–165, 1992.

[15] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: A comparative analysis. In *PPPJ*, pages 11–20, 2009.

[16] W. Kim and G. Agha. Efficient support of location transparency in concurrent object-oriented programming languages. In *SC*, page 39, 1995.

[17] V. A. Korthikanti and G. Agha. Avoiding energy wastage in parallel applications. In *Green Computing Conference*, pages 149–163. IEEE, 2010.

[18] V. A. Korthikanti and G. Agha. Towards optimizing energy costs of algorithms for shared memory architectures. In F. M. auf der Heide and C. A. Phillips, editors, *SPAA*, pages 157–165. ACM, 2010.

[19] V. A. Korthikanti, G. Agha, and M. R. Greenstreet. On the energy complexity of parallel algorithms. In G. R. Gao and Y.-C. Tseng, editors, *ICPP*, pages 562–570. IEEE, 2011.

[20] N. Kumar, K. Sen, J. Meseguer, and G. Agha. A rewriting based model for probabilistic distributed object systems. In *FMOODS*, pages 32–46, 2003.

[21] Y. Kwon and G. Agha. Scalable modeling and performance evaluation of wireless sensor networks. In *IEEE Real Time Technology and Applications Symposium*, pages 49–58, 2006.

[22] Y. Kwon and G. Agha. A markov reward model for software reliability. In *IPDPS*, pages 1–6, 2007.

[23] Y. Kwon and G. Agha. Performance evaluation of sensor networks by statistical modeling and euclidean model checking. *TOSN*, 9(4):39, 2013.

[24] Y. Kwon and G. A. Agha. iltlchecker: A probabilistic model checker for multiple dtmcs. In *QEST*, pages 245–246, 2005.

[25] Y. Kwon and G. A. Agha. Verifying the evolution of probability distributions governed by a dtmc. *IEEE Trans. Software Eng*, 37(1):126–141, 2011.

[26] Y. Kwon, S. Sundresh, K. Mechitov, and G. Agha. Actornet: an actor platform for wireless sensor networks. In H. Nakashima, M. P. Wellman, G. Weiss, and P. Stone, editors, *AAMAS*, pages 1297–1300. ACM, 2006.

[27] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Basset: a tool for systematic testing of actor programs. In G.-C. Roman and K. J. Sullivan, editors, *SIGSOFT FSE*, pages 363–364. ACM, 2010.

[28] N. Miskov-Zivanov, P. Zuliani, E. M. Clarke, and J. R. Faeder. Studies of biological networks with statistical model checking: application to immune system cells. In J. Gao, editor, *BCB*, page 728. ACM, 2013.

[29] S. Negara, R. K. Karmani, and G. A. Agha. Inferring ownership transfer for efficient message passing. In *PPOPP*, pages 81–90, 2011.

[30] R. Panwar and G. Agha. A methodology for programming scalable architectures. *J. Parallel Distrib. Comput*, 22(3):479–487, 1994.

[31] S. Ren and G. Agha. Rtsynchronizer: Language support for real-time specifications in distributed systems. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 50–59, 1995.

[32] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In L. Baresi and R. Heckel, editors, *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006.

[33] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.

[34] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[35] K. Sen, G. Rosu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 123–138. Springer, 2004.

[36] K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In A. Finkelstein, J. Estublier, and D. S. Rosenblum, editors, *ICSE*, pages 418–427. IEEE Computer Society, 2004.

[37] K. Sen, M. Viswanathan, and G. Agha. Statistical model checking of black-box probabilistic systems. In *CAV*, pages 202–215, 2004.

[38] D. C. Sturman and G. Agha. A protocol description language for customizing semantics. In *SRDS*, pages 148–157, 1994.

[39] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively learning to verify safety for FIFO automata. In K. Lodaya and M. Mahajan, editors, *FSTTCS*, volume 3328 of *Lecture Notes in Computer Science*, pages 494–505. Springer, 2004.

[40] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to verify safety properties. In *ICFEM*, pages 274–289, 2004.

[41] N. Venkatasubramanian, G. Agha, and C. L. Talcott. Scalable distributed garbage collection for systems of active objects. In Y. Bekkers and J. Cohen, editors, *IWMM*, volume 637 of *Lecture Notes in Computer Science*, pages 134–147. Springer, 1992.

[42] N. Venkatasubramanian, C. L. Talcott, and G. Agha. A formal model for reasoning about adaptive qos-enabled middleware. *ACM Trans. Softw. Eng. Methodol*, 13(1):86–147, 2004.

[43] Y.-C. Wang, A. Komuravelli, P. Zuliani, and E. M. Clarke. Analog circuit verification by statistical model checking. In *ASP-DAC*, pages 1–6. IEEE, 2011.

Bio: Gul Agha is Professor of Computer Science, and of Electrical and Computer Engineering, at the University of Illinois at Urbana-Champaign. Dr. Agha's research is in the area of programming models and languages for open distributed and embedded computation. Dr. Agha is a Fellow of the IEEE. He served as Editor-in-Chief of IEEE Concurrency: Parallel, Distributed and Mobile Computing (1994-98), and of ACM Computing Surveys (1999-2007). He has published over 200 research articles and supervised over 20 PhD dissertations. His book on Actors, published by MIT Press, is among the most widely cited works. Besides work on semantics and implementation of actor languages, Agha has done pioneering research in concolic testing, predictive monitoring, computation learning for verification, statistical model checking and Euclidean model checking, and energy complexity of parallel algorithms.