Actors

Rajesh K. Karmani, Gul Agha
Open Systems Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign
{rkumar8, agha}@illinois.edu

I. Definition

Actors is a model of concurrent computation for developing parallel, distributed and mobile systems. Each actor is an autonomous object that operates concurrently and asynchronously, receiving and sending messages to other actors, creating new actors, and updating its own local state. An actor system consists of a collection of actors, some of whom may send messages to, or receive messages from, actors outside the system.

II. Preliminaries

An *actor* has a name that is globally unique and a behavior which determines its actions. In order to send an actor a message, the actor's name must be used; a name cannot be guessed but it may be communicated in a message. When an idle is idle, and it has a pending message, the actor accepts the message, and does the computation defined by its behavior. As a result the actor may take three types of actions: *send messages*, *create new actors*, and *update* its local state. An actor's behavior may change as it modifies its local state. Actors do not share state: an actor must explicitly send a message to another actor in order to affect the latter's behavior. Each actor carries out its actions concurrently (and asynchronously) with other actors. Moreover, the path a message takes, as well as network delays it may encounter, are not specified. Thus the arrival order of messages is indeterminate. The key semantic properties of the standard Actor model are *encapsulation* of state and *atomic* execution of a method in response to a message, *fairness* in scheduling actors and in the delivery of messages, and *location transparency* enabling distributed execution and mobility.

A. Advantages of the Actor Model:

In the object-oriented programming paradigm, an object encapsulates data and behavior. This separates the interface of an object (what an object does) from the its representation (how it does it). Such separation enables modular reasoning about object-based programs and facilitates their evolution. Actors extend the advantages of objects to concurrent computations by separating control (where and when) from the logic of a computation.

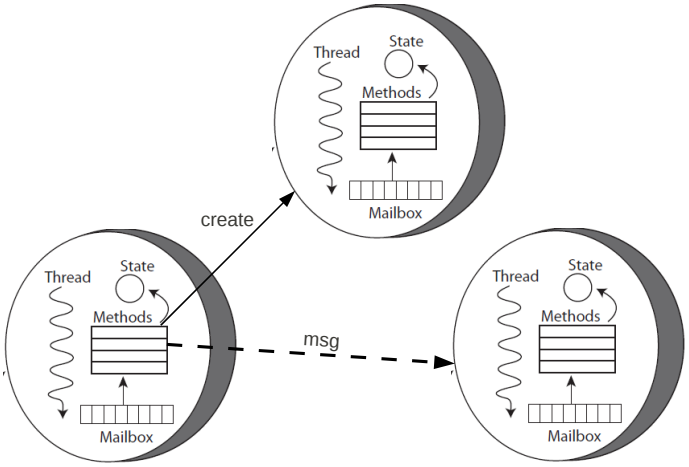

Figure 1. Actors are concurrent objects which communicate through messages and may create new actors. An actor may be viewed as an object augmented with its own control, a *mailbox* and a globally unique, immutable *name*.

The Actor model of programming [1] allows programs to be decomposed into self-contained, autonomous, interactive, asynchronously operating components. Due to their asynchronous operation, actors provide a model for the nondeterminism inherent in distributed systems, reactive systems, mobile systems, and any form of interactive computing.

B. History:

The concept of actors has developed over three decades. The earliest use of the term *actors* was in Carl Hewitt's *Planner* [2] where the term referred to rule-based active entities which search a knowledge base for patterns to match, and in response, trigger actions. For the next two decades, Hewitt's group worked on actors as agents of computation, and it evolved as a model of concurrent computing. A brief history of actor research can be found in [3]. The commonly used definition of actors today follows the work of Agha (1985) which defines actors using a simple operational semantics [1].

In recent years, the Actor model has gained in popularity with the growth of parallel and distributed computing platforms such as multi-core architectures, cloud computers and sensor networks. A number of actor languages and frameworks have been developed. Some early actor languages include *ABCL*, *POOL*, *ConcurrentSmalltalk*, *ACT++*, *CEiffel* (see [4] for a review of these) and HAL [5]. Actor languages and frameworks in current use include *Erlang* (from Ericsson) [6], *E* (Erights.org), *Scala* Actors library (EPFL) [7], *Ptolemy* (UC Berkeley) [8], *ASP* (INRIA) [9], *JCoBox* (University of Kaiserslautern) [10], *SALSA* (UIUC and RPI) [11], *Charm++* [12] and *ActorFoundry* [13] (both from UIUC), the *Asynchronous Agents Library* [14] and *Axum* [15] (both from Microsoft), and *Orleans* framework for cloud computing from Microsoft Research [16]. Some well-known open source applications built using actors include Twitter's message queuing system and Lift Web Framework, and among commercial applications are Facebook Chat system and Vendetta's game engine.

III. Illustrative Actor Language

In order to show how actor programs work, consider a simple imperative actor language ActorFoundry that extends Java. A class defining an actor behavior extends `osl.manager.Actor`. Messages are handled by methods; such methods are annotated with `@message`. The `create(class, args)` method creates an actor instance of the specified actor class, where `args` correspond to the arguments of a constructor in the class. Each newly created actor has a unique name that is initially known only to the creator at the point where the creation occurs.

Listing 1 Hello World! program in ActorFoundry

```
1  public class HelloActor extends Actor {
2
3    @message
4    public void greet() throws RemoteCodeException
5    {
6        ActorName other = null;
7        send(stdout, "print", "Hello");
8        other = create(WorldActor.class);
9        send(other, "audience");
10   }
11 }
12
13 public class WorldActor extends Actor {
14
15   @message
16   public void audience() throws RemoteCodeException
17   {
18       send(stdout, "print", "World");
19   }
20 }
```

A. Execution Semantics

The semantics of ActorFoundry can be informally described as follows. Consider an ActorFoundry program P that consists of a set of actor definitions. An actor communicates with another actor in P by sending asynchronous (non-blocking) messages using the `send` statement: `send`*(a,msg)* has the effect of *eventually* appending the contents of *msg* to the mailbox of the actor *a*. However, the call to `send` returns immediately i.e. the sending actor does not wait for the message to arrive at its destination. Because actors operate asynchronously, and the network has indeterminate delays, the arrival order of messages is *nondeterministic*. However, we assume that messages are *eventually* delivered (a form of *fairness*).

At the beginning of execution of P, the mailbox of each actor is empty and some actor in the program must receive a message from the environment. The ActorFoundry runtime first creates an instance of a specified actor and then sends a specified message to it, which serves as P's entry point.

Each actor can be viewed as executing a loop with the following steps: remove a message from its mailbox (often implemented as a queue), decode the message, and execute the corresponding method. If an actor's mailbox is empty, the actor blocks–waiting for the next message to arrive in the mailbox (such blocked actors are referred to as *idle actors*). The processing of a message may cause the actor's local state to be updated, new actors to be created and messages to be sent. Because of the encapsulation property of actors, there is no interference between messages that are concurrently processed by different actors.

An actor program 'terminates' when every actor created by the program is idle and the actors are not open to the environment (otherwise the environment could send new messages to their mailboxes in the future). Note that an actor program need not terminate–in particular, certain interactive programs and operating systems may continue to execute indefinitely.

Listing 1 shows the HelloWorld program in ActorFoundry. The program comprises of two actor definitions, the `HelloActor` and the `WorldActor`. An instance of the `HelloActor` can receive one type of message, the `greet` message, which triggers the execution of `greet` method. The `greet` method serves as P's entry point, in lieu of the traditional `main` method.

On receiving a `greet` message, the `HelloActor` sends a `print` message to the `stdout` actor (a built-in actor representing the standard output stream) along with the string "Hello". As a result, "Hello" will eventually be printed on the standard output stream. Next, it creates an instance of the `WorldActor`. The `HelloActor` sends a a `audience` message to the `WorldActor`, thus delegating the printing of "World" to it. Note that due to asynchrony in communication, it is possible for "World" to be printed before "Hello".

DRAFT

IV. Synchronization

Synchronization in actors is achieved through communication. Two types of commonly used communication patterns are Remote Procedure Call (RPC) -like messaging and local synchronization constraints. Language constructs can enable actor programmers to specify such patterns. Such language constructs are definable in terms of primitive actor constructs, but providing them as first-class linguistic objects simplifies the task of writing parallel code.

A. RPC-like Messaging

RPC-like communication is a common pattern of message-passing in actor programs. In RPC-like communication, the sender of a message waits for the reply to arrive before the sender proceeds with processing other messages. For example, consider the pattern shown in Figure 2 for a client actor which requests a quote from a travel service. The client wishes to wait for the quote to arrive before it decides whether to buy the trip, or to request a quote from another service.

Without a high-level language abstraction to express RPC-like message pattern, a programmer has to explicitly implement the following steps in their program:

1) the client actor sends a request;
2) the client then checks incoming messages;
3) if the incoming message corresponds to the reply to its request, the client takes the appropriate action (accept the offer, or keep searching);
4) if an incoming message does not correspond to the reply to its request, the message must be handled (for example, by being buffered for later processing), and the client continues to check messages for the reply.

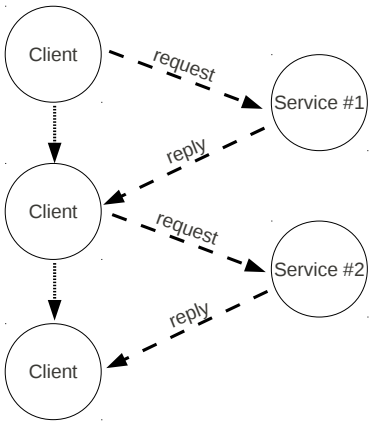

Figure 2. A client actor requesting quotes from multiple competing services using RPC-like communication. The dashed slanted arrows denote messages and the dashed vertical arrows denote that the actor is waiting or is blocked during that period in its life.

RPC-like messaging is almost universally supported in actor languages and libraries. RPC-like messages are particularly useful in two kinds of common scenarios. One scenario occurs when an actor wants to send an ordered sequence of messages to a particular recipient–in this case, it wants to ensure that a message has been received before it sends another. A variant of this scenario is where the sender wants to ensure that the the target actor has received a message before it communicates this information to another actor. A second scenario is when the state of the requesting actor is dependent on the reply it receives. In this case, the requesting actor cannot meaningfully process unrelated messages until it receives a response.

Because RPC-like messages are similar to procedure calls in sequential languages, programmers have a tendency to overuse them. Unfortunately, inappropriate usage of RPC-like messages introduces unnecessary dependencies in the code. This may not only make the program's execution more inefficient than it needs to be, it can lead to *deadlocks* and *livelocks* (where an actor ignores or postpones processing messages, waiting for an acknowledgement that never arrives).

B. Local Synchronization Constraints

Asynchrony is inherent in distributed systems and mobile systems. One implication of asynchrony is that the number of possible orderings in which messages may *arrive* is exponential in the number of messages that are 'pending' at any time (i.e., messages that have been sent but have not been received). Because a sender may be unaware of what the state of the actor it is sending a message to will be when the latter receives the message, it is possible that the recipient may not be in a state where it can process the message it is receiving. For example, a spooler may not have a job when some printer requests one. As another example, messages to actors representing individual matrix elements (or groups of elements) asking them to process different iterations in a parallel Cholesky decomposition algorithm need to be monotonically ordered. The need for such orderings leads to considerable complexity in concurrent programs, often introducing bugs or inefficiencies due to suboptimal implementation strategies. For example, in the case of Cholesky decomposition, imposing a global ordering on the iterations leads to highly inefficient execution on multicomputers [17].

Consider the example of a print spooler. Suppose a 'get' message from an idle printer to its spooler may arrive when the spooler has no jobs to return the printer. One way to address this problem is for the spooler to refuse the request. Now the printer needs to repeatedly poll the spooler until the latter has a job. This technique is called *busy waiting*; busy waiting can be expensive–preventing the waiting actor from possibly doing other work while it "waits", and it results in unnecessary message traffic. An alternate is to the

spooler buffer the 'get' message for deferred processing. The effect of such buffering is to change the order in which the messages are processed in a way that guarantees that the number of messages put messages to the spooler is always greater than the number of get messages processed by the spooler.

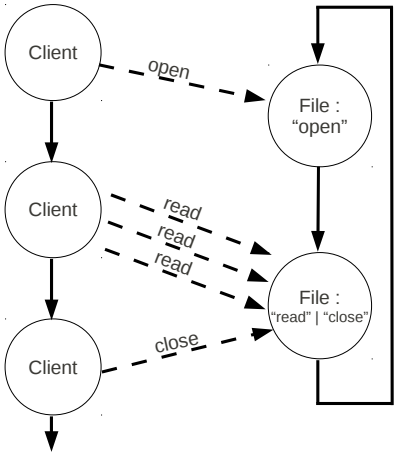

Figure 3. A file actor communication with a client is constrained using local synchronization constraints. The vertical arrows depict the timeline of the life of an actor and the slanted arrows denote messages. The labels inside a circle denote the messages that the file actor can accept in that particular state.

If pending messages are buffered explicitly inside the body of an actor, the code specifying the functionality (the *how* or representation) of the actor is mixed with the logic determining the order in which the actor processes the messages (the *when*). Such mixing violates the software principle of *separation of concerns*. Researchers have proposed various constructs to enable programmers to specify the correct orderings in a modular and abstract way, specifically, as logical formulae (predicates) over the state of an actor and the type of messages. Many actor languages and frameworks provide such constructs; examples include local synchronization constraints in ActorFoundry, and pattern matching on sets of messages in Erlang and Scala Actors library.

C. Patterns of Actor Programming

Two common patterns of parallel programming are *pipeline* and *divide-and-conquer* [18]. These patterns are illustrated in Fig. 4(a) and Fig. 4(b), respectively.

An example of the pipeline pattern is an image processing network (see Figure 4(a)) in which a stream of images is passed through a series of filtering and transforming stages. The output of the last stage is a stream of processed images. This pattern has been demonstrated by an image processing example, written using the *Asynchronous Agents Library* [14], which is part of the Microsoft Visual Studio 2010.

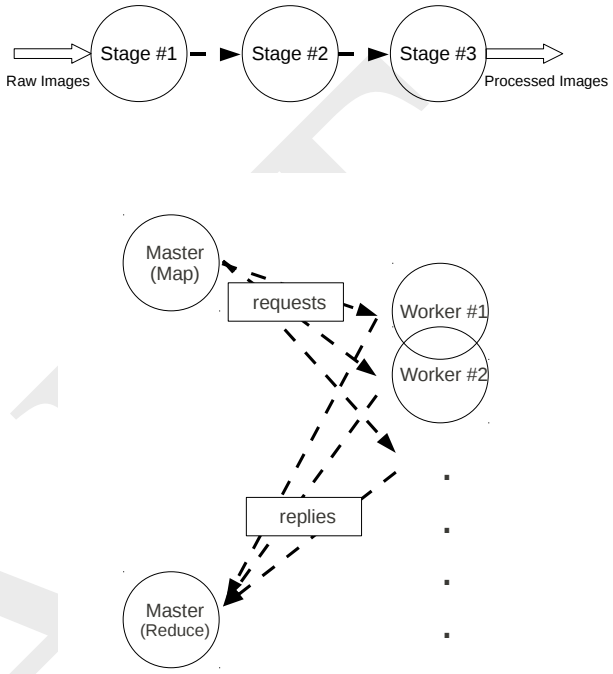

Figure 4. Patterns of actor programming (from top) (a) Pipeline pattern (b) Divide-and-Conquer pattern.

A *map-reduce* graph is an example of the divide-and-conquer pattern (see Figure 4(b)). A master actor maps the computation onto a set of workers and the output from each of these workers is reduced in the 'join continuation' behavior of the master actor (possibly modeled as a separate actor) (e.g., see [19]). Other examples of divide-and-conquer pattern are naïve parallel quicksort [20] and parallel mergesort. The synchronization idioms discussed above may be used in succinct encoding of these patterns in actor programs since these patterns essentially require ordering the processing of some messages.

V. Semantic Properties

As mentioned earlier, some important semantic properties of the pure Actor model are: encapsulation and atomic execution of methods (where a method represents computation in response to a message), fairness, and location transparency [21]. We discuss the implications of these properties.

Note that not all actor languages enforce all these properties. Often the implementations compromise some actor

properties, typically because it is simpler to achieve efficient implementations by doing so. However, it is possible by sophisticated program transformations, compilation and runtime optimizations to regain almost all the efficiency that is lost in a naïve language implementation, although doing so is more challenging for library-like actor frameworks [21]. By failing to enforce some actor properties in an actor language or framework implementation, actor languages add to the burden of the programmers, who have to then ensure that they write programs in a way that does not violate the property.

A. Encapsulation and Atomicity

Encapsulation implies that no two actors share state. This is useful for enforcing an object-style decomposition in the code. In sequential object-based languages, this has led to the natural model of atomic change in objects: an object invokes (sends a message to) another object, which finishes processing the message before accepting another message from a different object. This allows us to reason about the behavior of the object in response to a message, given the state of the target object when it received the message. In a concurrent computation, it is possible for a message to arrive while an actor is busy processing another message. Now if the second message is allowed to interrupt the target actor and modify the target's state while the target is still processing the first message, it is no longer feasible to reason about the behavior of the target actor based on what the target's state was when it received the first message. This makes it difficult to reason about the behavior of the system as such interleaving of messages may lead to erroneous and inconsistent states.

Instead, the target actor processes messages one at a time, in a single *atomic step* consisting of all actions taken in response to a given message [3]. By dramatically reducing the nondeterminism that must be considered, such atomicity provides a *macro-step semantics* which simplifies reasoning about actor programs. Macro-step semantics is commonly used by correctness-checking tools; it significantly reduces the state-space exploration required to check a property against an actor program's potential executions (e.g., see [22]).

B. Fairness

The Actor model assumes a notion of *fairness* which states that every actor makes progress if it has some computation to do, and that every message is eventually delivered to the destination actor, unless the destination actor is permanently disabled. Fairness enables modular reasoning about the liveness properties of actor programs [3]. For example, if an actor system A is composed with an actor system B where B includes actors that are permanently busy, the composition does not affect the progress of the actors in A. An familiar example where fairness would be useful is in browsers. Problems are often caused by the composition of browser components with third-party plug-ins: in the absence of fairness, such plug-ins sometimes result in browser crashes and hang-ups.

C. Location Transparency

In the Actor model, the actual location of an actor does not affect its name. Actors communicate by exchanging messages with other actors, which could be on the same core, on the same CPU, or on another node in the network. Location transparent naming provides an abstraction for programmers, enabling them to program without worrying about the actual physical location of actors. Location transparent naming facilitates automatic migration in the runtime, much as indirection in addressing facilitates compaction following garbage collection in sequential programming.

Mobility is defined as the ability of a computation to migrate across different nodes. Mobility is important for load-balancing, fault-tolerance and reconfiguration. In particular, mobility is useful in achieving scalable performance, particularly for dynamic, irregular applications [23]. Moreover, employing different distributions in different stages of a computation may improve performance. In other cases, the optimal or correct performance depends on runtime conditions such as data and work load, or security characteristics of different platforms. For example, web applications may be migrated to servers or to mobile clients depending on the network conditions and capabilities of the client [24].

Mobility may also be useful in reducing the energy consumed by the execution of parallel applications. Different parts of an application often involve different parallel algorithms and the energy consumption of an algorithm depends on how many cores the algorithm is executed on and at what frequency these cores operate [25]. Mobility facilitates dynamic redistribution of a parallel computation to the appropriate number of cores (i.e., to the number of cores that minimize energy consumption for a given performance requirement and parallel algorithm) by migrating actors. Thus mobility could be an important feature for energy-aware programming of multi-core (*manycore*) architectures. Similarly energy savings may be facilitated by being able to migrate actors in sensor networks and clouds.

VI. Implementations

Erlang is arguably the best known implementation of the Actor model. It was developed to program telecom switches at Ericsson about 20 years ago. Some recent actor implementations have been listed earlier. Many of these implementations have focused on a particular domain such as the Internet (SALSA), distributed applications (Erlang and E), sensor networks (ActorNet), and more recently multi-core processors (Scala Actors library, ActorFoundry, and many others in development).

DRAFT

It has been noted that a faithful but naïve implementation of the Actor model can be highly inefficient [21] (at least on the current generation of architectures). Consider three examples:

1) An implementation that maps each actor to a separate process may have a high cost for actor creation.
2) If the number of cores is less than the number of actors in the program (sometimes termed *CPU oversubscription*), an implementation mapping actors to separate processes may suffer from high context switching cost.
3) If two actors are located on the same sequential node, or on a shared-memory processor, it may be an order of magnitude more efficient to pass a reference to the message contents rather than to make a copy of the actual message contents.

These inefficiencies may be addressed by compilation and runtime techniques, or through a combinations of the two. The implementation of the ABCL language [26] demonstrates some early ideas for optimizing both intra-node and inter-node execution and communication between actors. The Thal language project [23] shows that encapsulation, fairness and universal naming in an actor language can be implemented efficiently on commodity hardware by using a combination of compiler and runtime. The Thal implementation also demonstrates that various communication abstractions such as RPC-like communication, local synchronization constraints and join expressions can also be supported efficiently using various compile-time program transformations.

The Kilim framework develops a clever post-compilation continuation-passing style (CPS) transform ("weaving") on Java-based actor programs for supporting light-weight actors that can pause and resume [27]. Kilim and Scala also add type systems to support safe but efficient messages among actors on a shared node [27], [28]. Recent work suggests that *ownership transfer* between actors, which enables safe and efficient messaging, can be statically inferred in most cases [29].

On distributed platforms such as cloud computers or grids, because of latency in sending messages to remote actors, an important technique for achieving good performance is communication-computation overlap. Decomposition into actors and the placement of actors can significantly determine the extent of this overlap. Some of these issues have been effectively addressed in the Charm++ runtime [12]. Decomposition and placement issues are also expected to show up on scalable manycore architectures since these architecture cannot be expected to support constant time access to shared memory.

Finally note that the notion of garbage in actors is somewhat complex. Because an actor name may be communicated in a message, it is not sufficient to mark the forward acquaintances (references) of *reachable* actors as reachable. The inverse acquaintances of reachable actors that may be potentially active need to be considered as well (these actors may send a message to a reachable actor). Efficient garbage collection of distributed actors remains an open research problem because of the problem of taking efficient distributed snapshots of the reachability graph in a running system [30].

VII. Tools

Several tools are available to aid in writing, maintaining, debugging, model checking and testing actor programs. Both Erlang and Scala have a plug-in for the popular, open source IDE (Integrated Development Environment) called Eclipse (`http://www.eclipse.org`). A commercial testing tool for Erlang programs called QuickCheck [31] is available. The tool enables programmers to specify program properties and input generators which are used to generate test inputs.

JCute [32] is a tool for automatic unit testing of programs written in a Java actor framework. Basset [33] works directly on executable (Java bytecode) actor programs and is easily retargetable to any actor language that compiles to bytecode. Basset understands the semantic structure of actor programs (such as the macro-step semantics), enabling efficient path exploration through the Java Pathfinder (JPF)– a popular tool for model checking programs [34]. The term rewriting system Maude provides an Actor module to specify program behavior; which has been used to model check actor programs [35]. There has also been work on runtime monitoring of actor programs [36].

VIII. Extensions and Abstractions

A programming language should facilitate the process of writing programs by being close to the conceptual level at which a programmer thinks about a problem, rather than at the level at which it may be implemented. Higher level abstractions for concurrent programming may be defined in interaction languages which allow patterns to be captured as first-class objects [37]. Such abstractions can be implemented through an adaptive, reflective middleware [38]. Besides programming abstractions for concurrency in the pure (asynchronous) actor model, there are variants of the Actor model, such as for real-time, which extend the model [39], [8]. Two interaction patterns are discussed to illustrate the ideas of interaction patterns.

A. Pattern-directed communication

Recall that a sending actor must know the name of a target actor before the sending actor can communicate with the target actor. This property, called *locality*, is useful for compositional reasoning about actor programs–if it is known that only some actors can send a message to an actor A, then it may be possible to figure out what types of messages A may get and perhaps specify some constraints on the order in which it may get them. However, real-world

programs generally create an open system which interacts with their external environment. This means that having ways of discovering actors which provide certain services can be helpful. For example, if an actor migrates to some environment, discovering a printer in that environment may be useful.

Pattern-directed communication allows programmers to declare properties of a group of actors, enabling the use of the properties to discover actual recipients are chosen at runtime. In the *ActorSpace* model, an actor specifies recipients in terms of patterns over properties that must be satisfied by the recipients. The sender may send the message to all actors (in some group) that satisfy the property, or to a single representative actor [40]. There are other models for pattern based communication. In *Linda*, potential recipients specify a pattern for messages they are interested in [41]. The sending actors simply inserts a message (called *tuple* in Linda) into a *tuple-space*, from where the receiving actors may read or remove the tuples if the tuple matches the pattern of messages the receiving actor is interested in.

B. Coordination

Actors help simplify programming by increasing the granularity at which programmers need to reason about concurrency–viz., they may reason in terms of the potential interleavings of messages to actors, instead of in terms the interleavings of accesses to shared variables within actors. However, developing actor programs is still complicated and prone to errors. A key cause of complexity in actor programs is the large number of possible interleaving of messages to groups of actors: if these message orderings are not suitably constrained, some possible execution orders may fail to meet the desired specification.

Recall that local synchronization constraints postpone the dispatch of a message based on the contents of the messages and the local state of the receiving actor (see Section IV-B). *Synchronizers*, on the other hand, change the order in which messages are processed by a group of actors by defining constraints on ordering of messages processed at different actors in a group of actors. For example, if a withdrawal and deposit messages must be processed atomically by two different actors, a Synchronizer can specify that they must be scheduled together. Synchronizers are described in [42].

In the standard actor semantics, an actor which knows the name of a target actor may send the latter a message. An alternate semantics introduces the notion of a *channel*; a channel is used to establish communication between a given sender and a given recipient. Recent work on actor languages has introduced *stateful channel contracts* to constrain the order of messages between two actors. Channels are a central concept for communication between actors in both Microsoft's Singularity platform [43] and Microsoft's Axum language [15], while they can be optionally introduced between two actors in Erlang. Channel contracts specify a protocol that governs the communication between the two end-points (actors) of the channel. The contracts are stated in terms of state transitions based on observing messages on the channel.

From the perspective of each end-point (actor), the channel contract specifies the *interface* of the other end-point (actor) in terms of not only the type of messages but also the ordering on messages. In Erlang, contracts are enforced at runtime, while in Singularity a more restrictive notion of typed contracts make it feasible to check the constraints at compile time.

IX. Current Status and Perspective

Actor languages have been used for parallel and distributed computing in the real world for some time (e.g. Charm++ for scientific applications on supercomputers [12], Erlang for distributed applications [6]). In recent years, interest in actor-based languages has been growing, both among researchers and among practitioners. This interest is triggered by emerging programming platforms such as multicore computers and cloud computers. In some cases, such as cloud computing, web services and sensor networks, the Actor model is a natural programming model because of the distributed nature of these platforms. Moreover, as multicore architectures are scaled, multicore computers will also look more more like traditional multicomputer platforms. This is illustrated by the 48-core Single-Chip Cloud Computer (SCC) developed by Intel [44] and the 100-core TILE-Gx by Tilera [45]. However, the argument for using actor-based programming languages is not simply that they are a good match for distributed computing platforms; it is that actors is a good model in which to think about concurrency. Actors simplify the task of programming by extending object-based design to concurrent (parallel, distributed, mobile) systems.

References

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] C. Hewitt. PLANNER: A language for proving theorems in robots. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pages 295–301. Morgan Kaufmann Publishers Inc., 1969.

[3] Gul Agha, Ian A. Mason, Scott Smith, and Carolyn Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(01):1–72, 1997.

[4] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Comput. Surv.*, 30(3):291–329, 1998.

[5] Chris Houck and Gul Agha. Hal: A high-level actor language and its distributed implementation. In *21st International Conference on Parallel Processing (ICPP), vol. II*, pages 158–165, 1992.

DRAFT

[6] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[7] P. Haller and M. Odersky. Actors That Unify Threads and Events. In *9th International Conference on Coordination Models and Languages*, volume 4467 of *Lecture Notes in Computer Science*. Springer, 2007.

[8] Edward A. Lee. Overview of the ptolemy project. Technical Report UCB/ERL M03/25, University of California, Berkeley, 2003.

[9] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous sequential processes. *Information and Computation*, 207(4):459 – 495, 2009.

[10] Jan Schäfer and Arnd Poetzsch-Heffter. Jcobox: generalizing active objects to concurrent components. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.

[12] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, 1993.

[13] Mark Astley. *The Actor Foundry: A Java-based Actor Programming Environment*. Open Systems Laboratory, University of Illinois at Urbana-Champaign, 1998-99.

[14] Microsoft Corporation. Asynchronous agents library. *http://msdn.microsoft.com/en-us/library/dd492627(VS.100).aspx*.

[15] Microsoft Corporation. Axum programming language. *http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx*.

[16] Gabriel Kliot James Larus Ravi Pandya Jorgen Thelin Sergey Bykov, Alan Geller. Orleans: A framework for cloud computing. Technical Report MSR-TR-2010-159, Microsoft Research, 2010.

[17] G. Agha and W.Y. Kim. Parallel programming and complexity analysis using actors. In *Massively Parallel Programming Models, 1997. Proceedings. Third Working Conference on*, pages 68–79. IEEE, 2002.

[18] Gul Agha. Concurrent object-oriented programming. In *in Communications of the ACM, Association for Computing Machinery, vol. 33, no. 9, pp 125-141, September*, 1990.

[19] Thomas Huining Feng and Edward A. Lee. Scalable models using model transformation. In *1st International Workshop on Model Based Architecting and Construction of Embedded Systems (ACESMB)*, September 2008.

[20] V. Singh, V. Kumar, G. Agha, and C. Tomlinson. Scalability of parallel sorting on mesh multicomputers. In *Parallel Processing Symposium, 1991. Proceedings., Fifth International*, pages 92 –101, apr. 1991.

[21] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: a comparative analysis. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20, New York, NY, USA, 2009. ACM.

[22] Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In *Fundamental Approaches to Software Engineering (FASE) with ETAPS*, 2010.

[23] WooYoung Kim and Gul Agha. Efficient support of location transparency in concurrent object-oriented programming languages. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 39, New York, NY, USA, 1995. ACM.

[24] Po-Hao Chang and Gul Agha. Towards context-aware web applications. In *7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, 2007.

[25] Vijay Anand Korthikanti and Gul Agha. Towards optimizing energy costs of algorithms for shared memory architectures. In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 157–165, New York, NY, USA, 2010. ACM.

[26] Akinori Yonezawa, editor. *ABCL: an object-oriented concurrent system*. MIT Press, Cambridge, MA, USA, 1990.

[27] S. Srinivasan and A. Mycroft. Kilim: Isolation typed actors for Java. In *Procedings if the European Conference on Object Oriented Programming (ECOOP)*, 2008.

[28] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In Theo DHondt, editor, *ECOOP 2010 Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 354–378. Springer Berlin / Heidelberg, 2010.

[29] Stas Negara, Rajesh Kumar Karmani, and Gul Agha. Inferring ownership transfer for efficient message passing. In *To appear in the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, New York, NY, USA, 2011. ACM.

[30] Nalini Venkatasubramanian, Gul Agha, and Carolyn Talcott. Scalable distributed garbage collection for systems of active objects. In *in Y. Bekkers and J. Cohen (editors), International Workshop on Memory Management, ACM SIGPLAN and INRIA, St. Malo, France, Lecture Notes in Computer Science, vol. 637, pp 134-148, Springer-Verlag, September*, 1992.

[31] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 2–10, New York, NY, USA, 2006. ACM.

[32] K. Sen and G. Agha. Automated systematic testing of open distributed programs. In *Fundamental Approaches to Software Engineering (FASE)*, volume 3922 of *Lecture Notes in Computer Science*, pages 339–356. Springer, 2006.

DRAFT

[33] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A framework for state-space exploration of java-based actor programs. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 468–479, Washington, DC, USA, 2009. IEEE Computer Society.

[34] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proceedings of the 15th IEEE international conference on Automated software engineering*, ASE '00, pages 3–, Washington, DC, USA, 2000. IEEE Computer Society.

[35] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All about maude - a high-performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer-Verlag, Berlin, Heidelberg, 2007.

[36] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Efficient decentralized monitoring of safety in distributed systems. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 418–427, Washington, DC, USA, 2004. IEEE Computer Society.

[37] Daniel Sturman and Gul Agha. A protocol description language for customizing failure semantics. In *in Proceedings of the Thirteenth Symposium on Reliable Distributed Systems, pp 148-157, IEEE Computer Society Press, October*, 1994.

[38] Mark Astley, Daniel Sturman, and Gul Agha. Customizable middleware for modular distributed software. *Commun. ACM*, 44(5):99–107, 2001.

[39] Shangping Ren and Gul A. Agha. Rtsynchronizer: language support for real-time specifications in distributed systems. *SIGPLAN Not.*, 30(11):50–59, 1995.

[40] Gul Agha and Christian J. Callsen. Actorspace: an open distributed programming paradigm. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 23–32, New York, NY, USA, 1993. ACM.

[41] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32:444–458, April 1989.

[42] G. Agha, S. Frolund, WY Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(2):3–14, 1993.

[43] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, 2006.

[44] Intel Corporation. Single-chip Cloud Computer. *http://techresearch.intel.com/ProjectDetails.aspx?Id=1*.

[45] Tilera Corporation. TILE-Gx Processor Family. *http://tilera.com/products/processors/TILE-Gx_Family*.

DRAFT