8

Actors: A Model for Reasoning about Open Distributed Systems

Gul A. Agha, Prasannaa Thati, Reza Ziaei

Open Systems Laboratory
Department of Computer Science,
University of Illinois at Urbana-Champaign,
Urbana, Illinois, USA
Email: {agha,thati,ziaei}@cs.uiuc.edu, *Web:* http://www-osl.cs.uiuc.edu

8.1 Introduction

Open distributed systems are often subject to dynamic change of hardware or software components, for example, in response to changing requirements, hardware faults, software failures, or the need to upgrade some component. In other words, open systems are reconfigurable and extensible: they may allow components to be dynamically replaced or be connected with new components while they are still executing. The Actor theory we describe in this paper abstracts some fundamental aspects of open systems. Actors provide a natural generalization for objects – encapsulating both data and procedures. However, actors differ from sequential objects in that they are also units of concurrency: each actor executes asynchronously and its operation may overlap with other actors. This unification of data abstraction and concurrency is in contrast to language models, such as Java, where an explicit and independent notion of thread is used to provide concurrency. By integrating objects and concurrency, actors free the programmer from having to write explicit synchronization code to prevent harmful concurrent access to data within an object.

There are several fundamental differences between actors and other formal models of concurrency. First, an actor has a unique and persistent identity, although its behavior may change over time. Second, communication between actors is asynchronous and fair (messages sent are eventually received). Third, an actor's name may be freely given out – without, for example, enabling other actors to adopt the same name. Finally, new actors may be created with their own unique and persistent names. These characteristics provide reasonable abstraction for open distributed systems. In fact, actors provide a realistic model for a number of practical implementations, including those of software agents [AJ99].

1

The outline of the paper is as follows. The next section relates actors to other models of concurrency. Section 3 presents an introduction to actors. Section 4 presents the syntax and semantics of a simple actor language. Section 5 completes the discussion of the language semantics, along with a brief description of a notion of equivalence. Section 6 describes an example which shows how actor theory can be used to reason about open systems. The final section outlines current research directions and provides some perspective. The treatment in this paper is of necessity rather high-level. Interested readers should refer to the citations for technical details of the work as well as secondary references to the literature.

8.2 Related Work

A number of formal models have been proposed to formalize fundamental concepts of concurrent computation involving interaction and mobility. We relate actors to the most prominent of these: namely, the π-calculus [Mil93, Mil99] and its variants [HT91, Bou92].

The π-calculus evolved out of an earlier formal model of concurrency called the Calculus of Communicating Systems (CCS) [Mil89]. Processes in CCS are interconnected by a static topology. In order to overcome the limitations of CCS which did not model actor-like systems with their dynamic interconnection topology, the π-calculus was developed. The π-calculus enables dynamic interconnection by allowing channel names to be communicated.

The Actor model and π-calculus are similar in the sense that both model *concurrent* and *asynchronous processes, communication of values*, and *synchronization*. However, the two formalisms make different ontological commitments. We examine the most significant of these differences.

- The central difference between the π-calculus and the Actor model is that names in the former identify stateless communication channels, while names in the latter identify persistent agents. Representation of the object paradigm in π-calculus requires imposing a type system [San98, Wal95]. However, the usage of actor names embodies additional semantic properties not captured by these type systems. For instance, an actor has a unique name, and it may not create new actors with names received in a message. A typed π-calculus which also enforces these additional constraints is presented in [Tha00].
- Actors provide buffered, asynchronous communication as a primitive while communication in the π-calculus is synchronous. It is possible to simulate one in terms of the other, but such simulations insert a degree of

complication in reasoning, while at the same time such simulations only approximate the abstractions. Although synchronous communication can be useful for inferring pair-wise group knowledge – a necessary condition for joint action, it should be observed that process actions in both models are asynchronous, thus the synchronous communication in the π-calculus is not useful for any notion of joint action. The Actor model is closer to real distributed systems; one consequence of this proximity of asynchronous communication and distributed systems is that synchronous communication is not as efficient as a default communication mechanism in distributed systems (see [Agh86, Kim97, VA98]).

- Message delivery in the Actor model is fair, which allows greater modularity in reasoning (see Section 8.4.2). It is possible to add different notions of fairness in π-calculus and its variants, but there is no standard notion of fairness in these models.

Programming languages that have been developed based on π-calculus, such as the Nomadic π-calculus [SWP99], generally adopt key aspects of the Actor model. The Nomadic π-calculus was conceived primarily to study communication primitives for interaction between mobile agents. An agent in a Nomadic π-calculus is essentially a process with a unique name which communicates with other agents via asynchronous messages. The reader may note the similarity with the Actor model.

The Nomadic π-calculus model does have other aspects which are not shared with the Actor model. The model extends the basic ideas in π-calculus with notions of *sites* and *migrating agents*. Every agent is associated with a current host site, and agents may migrate between sites during their execution. The calculus identifies two kinds of communication primitives: location dependent primitives which require the knowledge of the current location of the target agent, and location independent primitives which do not.

In contrast, actors are not associated with a host. Moreover, to use the terminology in [Nee89] actor names are *pure*: they do not contain any information about the creation or location of an actor. However, variants of the Actor model exist in which actor names contain both creation and current location information. The agent definition based on actors explicitly models location [AJ99], and location information have been added to actor names to provide universal naming for the World Wide Computer model [Var00].
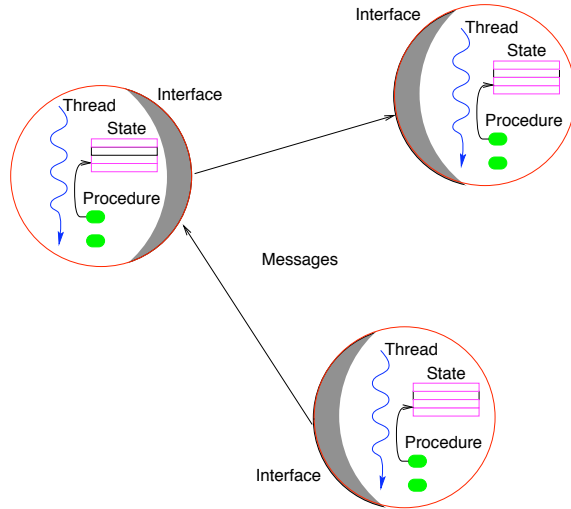
Fig. 8.1. Actors encapsulate a thread and state. The interface is comprised of public methods which operate on the state.

8.3 Actors

The Actor model provides an effective method for representing computation in real-world systems. Actors extend the concept of objects to concurrent computation [Agh86]. Recall that objects encapsulate a state and a set of procedures that manipulate the state; actors extend this by also encapsulating a thread of control (see Figure 8.1). Each actor potentially executes in parallel with other actors. It may know the addresses of other actors and can send messages to such actors. Actor addresses may be communicated in messages, allowing dynamic reconfiguration and name mobility. Finally, new actors may be created; such actors have their own unique addresses.

A concrete way to think of actors is that they represent an abstraction over concurrent architectures. An actor runtime system provides an abstract program interface (API) for services such as global addressing, memory management, fair scheduling, and communication. It turns out that the actor API can be efficiently implemented, thus raising the level of abstraction while reducing the size and complexity of code on concurrent architectures [KA95].

Note that the Actor model is, like the π-calculus, general and inherently parallel. Asynchronous communication in actors directly preserves the available potential for parallel activity: an actor sending a message does not have

to necessarily wait for the recipient to be ready to receive (or process) a message. Of course, it is possible to define actor-like buffered, asynchronous communication in terms of synchronous communication, provided dynamic actor (or process) creation is allowed. On the other hand, more complex communication patterns, such as remote procedure calls, can also be expressed as a sequence of asynchronous messages [Agh90]. Higher level actor languages often provide a number of communication abstractions.

8.4 A Simple Actor Language

It is possible to extend any sequential language with actor constructs. We use the call-by-value λ-calculus for this purpose. Here we will present a variant of the language presented in [AMST96] together with its formal syntax and semantics.

8.4.1 Syntax

We assume countably infinite sets $\mathbb{X}$(variables) and $\mathbb{At}$ (atoms). $\mathbb{At}$ contains t and nil for booleans, as well as constants for natural numbers, $\mathbb{N}$. We assume a countably infinite set of actor addresses. To simplify notation we identify this set with $\mathbb{X}$, and call the variables used in this way, i.e. the free variables in an actor configuration (see Section 8.4.2) as actor names. We also assume a set of (possibly empty) sets of n-ary operations, $\mathbb{F}_n$ on $\mathbb{At}$ for each $n \in \mathbb{N}$, and $\mathbb{F} = \bigcup_{n \in \mathbb{N}} \mathbb{F}_n$. $\mathbb{F}$ contains arithmetic operations, recognizers isatom for atoms, isnat for numbers, ispair for pairs, branching br, pairing pr, 1st, 2nd, and the following actor primitives: actor primitives send, newactor, and ready.

send(a, v) creates a new message:
  • with receiver a, and
  • contents v
newactor(b) creates a new actor:
  • with behavior b, and
  • returns its address
ready(b) captures local state change:
  • replaces the behavior of the executing actor with b
  • frees the actor to accept another message.

The sets of value expressions $\mathbb{V}$, and expressions $\mathbb{E}$ are defined inductively as follows:

Definition 1

$$\mathbb{V} = \mathbb{A}t \cup \mathbb{X} \cup \lambda\mathbb{X}.\mathbb{E} \cup \boldsymbol{pr}(\mathbb{V}, \mathbb{V})$$
$$\mathbb{E} = \mathbb{A}t \cup \mathbb{X} \cup \lambda\mathbb{X}.\mathbb{E} \cup \boldsymbol{app}(\mathbb{E}, \mathbb{E}) \cup \mathbb{F}_n(\mathbb{E}_n)$$

We let x, y, z range over $\mathbb{X}$, v range over $\mathbb{V}$, and e range over $\mathbb{E}$. To simplify the presentation of examples we use several abbreviations. The function `br` is a strict conditional, and the usual conditional construct `if` can be defined as the following abbreviation:

$$\texttt{if}(e_0, e_1, e_2) \quad \text{abbreviates} \quad \texttt{app}(\texttt{br}(e_0, \lambda z.e_1, \lambda z.e_2), \texttt{nil}) \qquad \text{for } z \text{ fresh}$$

Similarly, `let`, `seq`, and `rec` are the usual syntactic sugar: `let` is used for creating local bindings, seq is used as a sequencing primitive, and `rec` is the `Y` combinator used for recursion in call-by-value λ-calculus. Finally, `letactor` is a convenient abbreviation used for actor creations.

$$\texttt{letactor}\{x := e\}e' \quad \text{abbreviates} \quad \texttt{let}\{x := \texttt{newactor}(e)\}e'$$

Actor behaviors are represented as `lambda` abstractions. Delivery of a message m is simply the application of actor's behavior b to m, denoted by $\texttt{app}(b, m)$. The motivation behind the actor constructs is to provide the minimal extension that is necessary to lift a sequential language to a concurrent one supporting object-style encapsulation (of state and procedures) and coordination.

In Section 8.3.2, we provide an operational semantics for our language in terms of a transition relation on actor configurations.

Example

We provide a few examples to illustrate the Actor model. Since we are not concerned with the structure of messages, we represent messages abstractly by assuming functions to create messages, and to test or extract their contents. For example, we assume that $\texttt{mkget}(c)$ creates a 'get' message with content c and $\texttt{get?}(m)$ returns true if m is a 'get' message.

Sink. The first example is the behavior of an actor that ignores every message that it receives and becomes itself:

$$B_{\texttt{sink}} = \texttt{rec}(\lambda b.\lambda m.\texttt{ready}(b))$$

Cell. The second example is an actor that models the behavior of a variable store as used in imperative programming. We call this actor a *cell* and it responds to two sorts of messages. A `get` message contains the address of an actor requesting the value of the cell, and a `set` message which contains

a new value to replace cell's old value. The following code specifies the behavior of a cell actor.

$$B_{\text{cell}} = \texttt{rec}(\lambda b.\lambda c.\lambda m.$$
$$\texttt{if}(\texttt{get?}(m),$$
$$\texttt{seq}(\texttt{send}(cust(m),c),\texttt{ready}(b(c)))$$
$$\texttt{if}(\texttt{set?}(m),$$
$$\texttt{ready}(b(\texttt{contents}(m))),$$
$$\texttt{ready}(b(c)))))$$

Evaluating

```
letactor{a := B_cell(0)}
    seq(send(a, mkset(3)), send(a, mkset(4)), send(a, mkget(b)))
```

will result in the actor b receiving a message containing either 0, 3, or 4, depending on the arrival order of messages sent to cell a.

Tree Product. Our third example is a divide and conquer problem which illustrates how synchronization primitives can be modeled using actors. Suppose we want to determine the product of the leaves of a tree. We assume that every internal node of the tree has exactly two children, and that the leaves are integers. A divide and conquer strategy is to calculate the product of the leaves of each subtrees and then multiply the results. The sequential implementation of this algorithm can be represented by the following recursive function:

$$\texttt{treeprod} = \texttt{rec}(\lambda f.\lambda tree.$$
$$\texttt{if}(\texttt{isnat}(tree),$$
$$tree,$$
$$f(\texttt{left}(tree)) * f(\texttt{right}(tree)))$$

However, the same strategy can be used to obtain a parallel algorithm that concurrently evaluates products of subtrees. To synchronize the calculation of subtree products, we use *join continuation* actors which guarantee that several concurrent sub-computations are complete before beginning a computation that depends on the results of the sub-computations. The behavior $B_{\texttt{treeprod}}$ below implements a concurrent evaluation of tree products.

$$B_{\texttt{treeprod}} =$$
$$\texttt{rec}(\lambda b.\lambda self.\lambda m.$$
$$\texttt{if}(\texttt{notvalidtree}(\texttt{tree}(m)),$$
$$\texttt{seq}(\texttt{send}(\texttt{cust}(m),\texttt{error}),$$
$$\texttt{ready}(b(self))),$$

$$\texttt{if}(\texttt{isnat}(\texttt{tree}(m)),$$
$$\texttt{seq}(\texttt{send}(\texttt{cust}(m),\texttt{tree}(m)),$$
$$\texttt{ready}(b(\textit{self}))),$$
$$\texttt{letactor}\{jc := B_{\texttt{joincont}}(\texttt{cust}(m),0,\texttt{nil})\}$$
$$\texttt{seq}(\texttt{send}(\textit{self},\texttt{mkprd}(\texttt{left}(\texttt{tree}(m)),jc)),$$
$$\texttt{seq}(\texttt{send}(\textit{self},\texttt{mkprd}(\texttt{right}(\texttt{tree}(m)),jc))$$
$$\texttt{ready}(b(\textit{self}))))))))$$

The behavior of the join continuation actor is specified as:

$$B_{\texttt{joincont}} =$$
$$\texttt{rec}(\lambda b.\lambda \textit{cust}.\lambda \textit{nargs}.\lambda \textit{firstnum}.\lambda \textit{num}$$
$$\texttt{if}(\texttt{eq}(\textit{nargs},0),$$
$$\texttt{ready}(b(\textit{cust},1,\textit{num})),$$
$$\texttt{seq}(\texttt{send}(\textit{cust},\textit{firstnum} * \textit{num}),$$
$$\texttt{ready}(B_{\texttt{sink}}))))$$

Note that an actor with behavior $B_{\texttt{treeprod}}$ can evaluate multiple tree product requests concurrently. Specifically, the evaluation of a new tree product request can begin even before the evaluation of any previous requests is complete. The structure of many parallel computations, such as parallel search, is very similar.

8.4.2 Reduction Semantics for Actor Configurations

Instantaneous snapshots of actor systems are called *configurations*. The operational semantics of our language is defined by a transition relation on configurations. The notion of open systems is captured by defining a dynamic interface to a configuration, i.e. by explicitly representing a set of *receptionists* which may receive messages from actors outside the configuration and a set of actors *external* to the configuration which may receive messages from the actors within.

An *actor configuration* with actor map α, multi-set of messages μ, receptionists ρ, and external actors χ, is written

$$\langle \alpha \mid \mu \rangle^{\rho}_{\chi}$$

where ρ and χ are finite sets of actor addresses, α maps a finite set of addresses to their behavior, μ is a finite multi-set of (pending) messages. A message m contains the address of the actor it is targeted to and the message contents, $a \triangleleft v$. We restrict the contents to be any values constructed from

(beta-v) $R[\![\mathtt{app}(\lambda x.e, v)]\!] \overset{\lambda}{\mapsto} R[\![e[x := v]]\!]$

(delta) $R[\![\delta(v_1, \dots, v_n)]\!] \overset{\lambda}{\mapsto} R[\![v']\!]$

        where $\delta \in F_n$, $v_1, \dots, v_n \in At^n$, and $\delta(v_1, \dots, v_n) = v'$.

(eq) $R[\![\mathtt{eq}(v_0, v_1)]\!] \overset{\lambda}{\mapsto} \begin{cases} R[\![\mathtt{t}]\!] & \text{if } v_0 = v_1 \in At \\ R[\![\mathtt{nil}]\!] & \text{if } v_0, v_1 \in At \text{ and } v_0 \neq v_1 \end{cases}$

Fig. 8.2. Relation $\overset{\lambda}{\mapsto}$ on λ expressions.

atoms and actor addresses using the pairing constructor `pr`. We call these values as communicable values and let *cv* range over them.

Let $\langle\, \alpha \mid \mu \,\rangle^{\rho}_{\chi}$ be a configuration, and if $A = \text{Dom}(\alpha)$ (domain of α) then the following properties must hold:

(0) $\rho \subseteq A$ and $A \cap \chi = \emptyset$,

(1) if $a \in A$, then $\text{FV}(\alpha(a)) \subseteq A \cup \chi$, where $\text{FV}(\alpha(a))$ represents the free variables of $\alpha(a)$; and if $v_0 \triangleleft v_1$ is a message with content v_1 to actor address v_0, then $\text{FV}(v_i) \subseteq A \cup \chi$ for $i < 2$.

To describe local transitions at an actor, we decompose uniquely a non-value expression into a reduction context filled with a redex. A redex identifies the next sub-expression that is to be evaluated according to the reduction strategy (which in our case is left-first, call-by-value) [FF86]. Redexes are of two kinds: purely functional and actor redexes. The actor redexes are `send`(a, v), `newactor`(b) and `ready`(b). Reduction rules for the functional case are defined by a relation $\overset{\lambda}{\mapsto}$ on E as shown in Figure 8.2.

The transition relation i.e. $\mapsto$ on actor configurations is defined by the rules shown in Figure 8.3. The rules are all labeled to indicate the kind of reduction and any additional parameters. The notation $[e]_a$ denotes the (singleton) actor map which maps the name a to expression e.

The `<fun:`a`>` rule simply says that an actor's internal computation is defined by the semantics of the sequential language its behavior is written in. The `<new:`a, a'`>` rule says that a new actor with *fresh* name a' (no external actor or an actor already in the configuration can have the same name) is created and ready to receive messages. The new actor's name, a' is returned to the creating actor as the result of the `newactor` operation. The `<send:`a, m`>` rule defines the asynchronous semantics of message send. The new message is put in the message pool and the sending actor can continue

<fun:a>

$$e \stackrel{\lambda}{\mapsto} e' \Rightarrow \langle \alpha, [e]_a \mid \mu \rangle_\chi^\rho \mapsto \langle \alpha, [e']_a \mid \mu \rangle_\chi^\rho$$

<new:a, a'>

$$\langle \alpha, [R[\![\texttt{newactor}(v)]\!]]_a \mid \mu \rangle_\chi^\rho \mapsto \langle \alpha, [R[\![a']\!]]_a, [\texttt{ready}(v)]_{a'} \mid \mu \rangle_\chi^\rho \qquad a' \text{ fresh}$$

<send:a, m>

$$\langle \alpha, [R[\![\texttt{send}(v_0, v_1)]\!]]_a \mid \mu \rangle_\chi^\rho \mapsto \langle \alpha, [R[\![\texttt{nil}]\!]]_a \mid \mu, m \rangle_\chi^\rho \qquad m = v_0 \triangleleft v_1$$

<rcv:a, cv>

$$\langle \alpha, [R[\![\texttt{ready}(v)]\!]]_a \mid a \triangleleft cv, \mu \rangle_\chi^\rho \mapsto \langle \alpha, [\texttt{app}(v, cv)]_a \mid \mu \rangle_\chi^\rho$$

<out:m>

$$\langle \alpha \mid \mu, m \rangle_\chi^\rho \mapsto \langle \alpha \mid \mu \rangle_\chi^{\rho'}$$

$$\text{if } m = a \triangleleft cv, \ a \in \chi, \text{ and } \rho' = \rho \cup (\mathrm{FV}(cv) \cap \mathrm{Dom}(\alpha))$$

<in:m>

$$\langle \alpha \mid \mu \rangle_\chi^\rho \mapsto \langle \alpha \mid \mu, m \rangle_{\chi \cup (\mathrm{FV}(cv) - \mathrm{Dom}(\alpha))}^\rho$$

$$\text{if } m = a \triangleleft cv, \ a \in \rho \text{ and } \mathrm{FV}(cv) \cap \mathrm{Dom}(\alpha) \subseteq \rho$$

Fig. 8.3. Actor transitions.

its execution. The <rcv:a, cv> rule says that an actor can receive a message only when it is ready. In fact, execution of the **ready** operation blocks the actor's thread until the delivery of a message. The delivery is performed by applying the new behavior to the message. The last two rules, <out:m> and <in:m>, capture the openness of the configurations by allowing exchange of messages between the configuration and its environment. Note the dynamic nature of the interface and that the exchange of messages is restricted by the interface.

Because our language is untyped, creation of actors with ill-formed behaviors (i.e. behaviors that are not λ abstractions), and creation of messages with ill-formed contents (i.e. contents that are not communicable values) is possible. But the reduction system will prevent such ill-formed behaviors and messages from being used.

Example

Consider the following actor behavior that creates new cell actors upon request:

$B_{\text{c-maker}} =$
　　rec($\lambda b.\lambda self.\lambda m.$
　　　　letactor$\{newcell := B_{\text{cell}}(0)\}$
　　　　　　seq(send(cust(m), $newcell$)),
　　　　　　　　ready($b(self)$))))

An initial actor configuration containing a cell maker actor is given below:

$$\langle\ [\texttt{ready}(B_{\text{c-maker}}(cm))]_{cm}\ |\ \rangle_{\emptyset}^{\{cm\}}$$

Let's say this actor configuration makes an input transition with the label <in:$cm \triangleleft$mkcell(a)>. The resulting configuration will be:

$$\langle\ [\texttt{ready}(B_{\text{c-maker}}(cm))]_{cm}\ |\ cm \triangleleft \texttt{mkcell}(a)\ \rangle_{\{a\}}^{\{cm\}}$$

And after a <rcv:$cm, cm \triangleleft$mkcell(a)>, a series of fun transitions, and a <send:$a, a \triangleleft a'$> transition, we reach the following configuration:

$$\langle\ [\texttt{ready}(B_{\text{c-maker}}(cm))]_{cm}, [\texttt{ready}(B_{\text{cell}}(0))]_{a'}\ |\ a \triangleleft a'\ \rangle_{\{a\}}^{\{cm\}}$$

And with a final <out:$a \triangleleft a'$>, the following configuration will result:

$$\langle\ [\texttt{ready}(B_{\text{c-maker}}(cm))]_{cm}, [\texttt{ready}(B_{\text{cell}}(0))]_{a'}\ |\ \rangle_{\{a\}}^{\{cm,a'\}}$$

Following this transition, the actor name a' will be known to the outside world and further calls to cm will result in new cells being created.

8.4.3 *Local Synchronization Constraints*

Different actors carry out their operations asynchronously. This means that the sender of a message may not know what the state of a recipient is at the time it sends the message. Moreover, an actor may not be able to process particular types of messages while in certain states. For example, a lock that is currently owned by a process cannot accept any further requests to acquire the lock until it is released by the current owner. In models relying on synchronous messages, this is handled by guards on ports: different types of
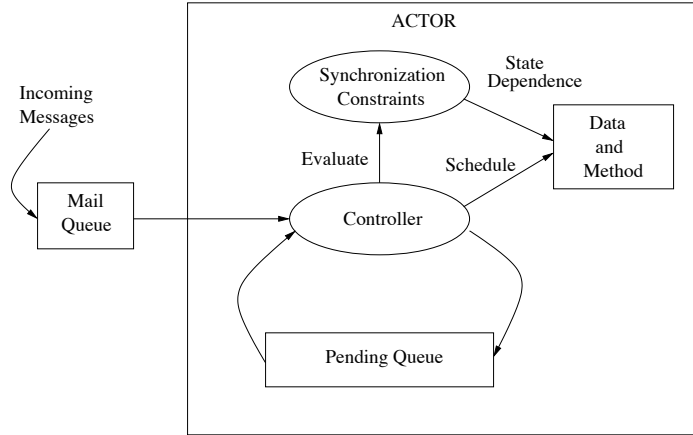
Fig. 8.4. An actor with local synchronization constraints.

messages are received at different ports, and ports may be disabled/enabled depending on the local process state and message contents, thereby blocking a communication.

In actors, message send is asynchronous and non-blocking. Different approaches to selectively process external communications may be taken to address the problem. One solution is to let an actor explicitly buffer the incoming communications that it is not ready to process (cf. insensitive actors [Agh86]). In the Rosette actor language, Tomlinson and Singh [TKS+89] proposed a mechanism which associates with each potential state of actor an *enabled set* specifying the particular *methods* the recipient actor is willing to invoke. The actor then processes the earliest received message in its queue which invokes a method in its current enabled set. The effect is to delay the processing of a message until such time that an actor is in a state where it is able to process it.

In this paper, we use a variation of this concept called *local synchronization constraints*. Local synchronization constraints are so called because their scope of influence is a single actor [Fro96]. A local synchronization constraint is a predicate that constrains the delivery of messages. Delivery of a message to a constrained actor is delayed until the message satisfies the constraint (see Figure 8.4). An actor's synchronization constraint reflects the state of the actor and therefore is updated every time an actor moves into its next state by executing a `ready` operation.

To account for synchronization constraints we slightly modify the standard

```
<rcv:   a, cv>
```
$$\langle \alpha, [R[\![\mathtt{ready}(v, c)]\!]]_a \mid a \triangleleft cv, \mu \rangle^\rho_\chi \mapsto \langle \alpha, [\mathtt{app}(v, cv)]_a \mid \mu \rangle^\rho_\chi$$
$$\text{if } \mathtt{app}(c, cv) = \mathtt{t}$$

Fig. 8.5. Transitions for Actor Configurations with Local Synchronization Constraints.

language of actors as in Figure 8.3. In the new language, the `ready` primitive is modified by adding a second argument: a synchronization constraint which is a predicate over messages. Consequently, the rule `rcv` must be modified to capture the intended semantics of synchronization constraints.

The new semantic rule is shown in Figure 8.5. All the other rules remain the same. Note that according to the side condition of the rule, if the computation of $\mathtt{app}(c, a \triangleleft cv)$ does not terminate, the condition will never hold and therefore the delivery will not take place – which is what we intuitively expect. However, the operational semantics as given is loose – since evaluating the constraint has no side-effects, an implementation could concurrently test the constraint against several messages, but then accept only one of the messages for which the constraint is satisfied (this is similar to the semantics of Dijkstra's guarded command). It should be noted that most actor languages ensure termination of testing synchronization constraints by disallowing recursion in constraints.

Finally, note that it is possible to translate the actors with local synchronization constraints into actors obeying the primitive semantics ([AKP95]). A proof that this translation is semantics preserving can be found in [MT99].

Example

The example in this section demonstrates how synchronization constraints can modularly control delivery of messages.

Consider the cell actor example again. Now, suppose we want to modify the cell to turn it into a single element buffer. In other words, we want to add the restriction that a `put` message be delivered only when the cell is empty and a `get` message delivered only when the cell is not empty. Assume the following abstract functions on messages: $\mathtt{put?}(m)$, $\mathtt{get?}(m)$.

The local synchronization constraints over a cell `cell` can be represented as predicate functions over messages as follows.

$$C_{\text{full}} \quad = \quad \lambda m.\texttt{get?}(m)$$
$$C_{\text{empty}} = \quad \lambda m.\texttt{put?}(m)$$

These constraints must be set by the actor when a **ready** operation is performed. Now, a single element buffer can be implemented as follows.

$$
\begin{aligned}
B_{\text{single-buffer}} \;=\; &\texttt{rec}(\lambda b.\lambda(v, sc).\lambda m. \\
&\quad \texttt{if}(\texttt{get?}(m), \\
&\qquad \texttt{seq}(\texttt{send}(cust(m), v), \\
&\qquad\qquad \texttt{ready}(b(v, C_{\text{empty}}), C_{\text{empty}})), \\
&\qquad \texttt{if}(\texttt{set?}(m), \\
&\qquad\qquad \texttt{ready}(b(\texttt{contents}(m), C_{\text{full}}), C_{\text{full}})), \\
&\qquad\qquad \texttt{ready}(b(c, sc), sc))) \qquad\qquad \text{; bad message}
\end{aligned}
$$

In the rest of this paper we assume that **ready(b)** abbreviates **ready($b, \lambda m.\texttt{true}$)**.

8.5 Theory

In this section we describe a theory of Actor computation. The basis of this theory was introduced in [AMST96]. Here we summarize the main elements of the theory and then introduce a proof technique based on i/o-path correspondence developed in [MT99].

8.5.1 Computation Trees and Paths

The behavior of an actor system will be represented by computation trees and paths. We write $\kappa_0 \xrightarrow{l} \kappa_1$ if $\kappa_0 \mapsto \kappa_1$ according to the rule labeled by l in Figure 8.3.

Definition 2 *The **computation tree** for a configuration κ, written as $\mathcal{T}(\kappa)$, is defined to be the set of all finite sequences of labeled transitions of the form $[\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < n]$ for some $n \in N$, with $\kappa = \kappa_0$. We call such sequences **computation sequences** and let ν range over them.*

Definition 3 *The sequences of a computation tree are partially ordered by the initial segment relation. A **computation path** from a configuration κ is a maximal linearly ordered set of computation sequences in $\mathcal{T}(\kappa)$. Note that a path can also be regarded as a (possibly infinite) sequence of labeled transitions.*

We use $\mathcal{T}^\infty(\kappa)$ to denote the set of all paths from κ, and let π range over computation paths. When thinking of a path as a possibly infinite sequence we write $[\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \bowtie]$ where $\bowtie \in N \cup \omega$ is the length of the sequence.

Since the result of a transition is uniquely determined by the starting configuration and the transition label, computation sequences and paths can also be represented by their initial configuration and the sequence of transition labels. The sequence of configurations can be computed by induction on the index of occurrence. We assume this representation of computation paths in the rest of this paper.

8.5.2 Fairness

The model we have developed provides fairness, namely that any enabled transition eventually fires. Under this assumption, not all paths are considered to be admissible. Fairness is an important requirement for reasoning about eventuality properties. It is particularly relevant in supporting modular reasoning.

There are two important consequences of fairness which illustrate its usefulness. The first of these is that each actor makes progress independent of how busy other actors are. Therefore, if we compose one configuration with another which has an actor with a nonterminating computation, computation in the first configuration may nevertheless proceed as before, for example, if actors in the two configurations do not interact. A second consequence is that messages are eventually delivered. This allows reasoning based on composition with some contexts to be carried forward: thus, if upon composition with a richer context, other requests may be sent to a particular server actor, previous requests sent to that server will still be received (provided the server itself does not "fail").

We now formally define fairness in our model. We say a label l is *enabled* in configuration κ if there is some κ' such that $\kappa \xrightarrow{l} \kappa'$.

Definition 4 *A path* $\pi = [\kappa_i \xrightarrow{l_i} \kappa_{i+1} \mid i < \bowtie]$ *in* $\mathcal{T}^\infty(\kappa)$ *is* **fair** *if each enabled transition eventually happens or becomes permanently disabled. That is, if l is enabled in κ_i and is not of the form* `<in:m>`*, then* $\kappa_j \xrightarrow{l} \kappa_{j+1}$ *for some* $j \geq i$*, or l has the form* `<rcv:  a,cv>` *and for some* $j \geq i$*, a is busy and never again becomes ready to accept a message. For a configuration* κ *we define* $\mathcal{F}(\kappa)$ *to be the subset of* $\mathcal{T}^\infty(\kappa)$ *that contains only fair paths.*

Note that every finite computation path is fair since, by maximality, all of the enabled transitions must have happened.

8.5.3 Interaction Paths and Path Correspondence

In this section, we introduce a notion of equivalence based on the idea of interaction paths [Tal96].

Definition 5 *An **interaction-path** ip is a subsequence of a computation path π, containing all and only the transitions labels in π that are of form:* <out:m> *and* <in:m>. *We say that ip is the observable projection (or just the projection) of π.*

In other words, an interaction-path is a computation path with all internal transitions removed. From now on, we will follow the convention of using π to range over interaction paths, and $\tau_0, \tau_1, \ldots$ to range over their transition labels. We also use the list notation $[\tau_0, \tau_1, \ldots]$ to represent (both finite and infinite) interaction paths.

The notion of fairness on computation paths naturally induces a similar notion on interaction paths.

Definition 6 *An interaction path is observably fair if it is the projection of a fair computation path.*

Note that an observably fair (just fair from now on) interaction path could be the projection of both fair and an unfair computation paths, hence the name *observational fairness*.

A strong motivation for a semantics based on interaction paths is to focus on the observable behavior of systems as the only criteria for investigating their equivalence and defining their meaning. Any method that makes some part of internal behavior explicit in the model, will undesirably distinguish systems which are otherwise equivalent from an external observer's view.

Now we define our notion of equivalence on configurations based on their set of interaction-paths.

Definition 7 *For a configuration κ, its set of interaction paths $\mathcal{I}(\kappa)$ is the set of observable projections of each computation path in $\mathcal{F}(\kappa)$.*

Definition 8 *We say two actor configurations C_1 and C_2 are equivalent under **path correspondence**, if they have the same set of recipients and external actors (same "interface") and their set of interaction paths are equal.*

Alternately, we can define the set of all finite prefixes of paths in $\mathcal{I}(\kappa)$ as the meaning of configuration κ. In [AMST96] equivalence relations were introduced based on the notion of testing. An *observable* 0-ary event was

added to the transitions, and configurations were tested by composing them with observation contexts (configurations). Two actor configurations were equivalent if their behaviors were the "same" in all observation contexts. Three notions of equivalences were defined. Two configurations are *must* equivalent provided some computation paths in one of them do not exhibit the observable event iff some computation paths in the other do not. Two configurations are *may* equivalent provided some paths in one of them exhibit the observable event iff some paths in the other do. Finally, two configurations are *convex* equivalent if they are both *may* and *must* equivalent. It was shown that under the fairness assumption the three equivalences collapse to just two, with the convex and must equivalences being identical. It is known that the notion of equivalence based on sets of finite prefixes of interaction paths is identical to the *may* equivalence, and the equivalence in Definition 7 is at least as strong as the *must* equivalence.

8.6 An Example Proof of Path Correspondence

In this section we show by an example how the theory of actors can help us verify, in a rather rigorous way, the correctness of systems modeled as actor configurations. We will use the tree product example from Section 8.4.1 and we will show the equivalence of two actor configurations: one based on the sequential implementation and the other based on the concurrent implementation of tree product.

We first need to define an actor behavior based on the sequential definition of `treeprod` given in Section 8.4.1:

$B_{\mathtt{seqtp}}$ =
  $\mathtt{rec}(\lambda b.\lambda self.\lambda m.$
    $\mathtt{if(\ notvalidtree(tree}(m)),$
      $\mathtt{seq(\ send(cust}(m), \mathtt{error}),$
        $\mathtt{ready}(b(self)))),$
      $\mathtt{seq(\ send(cust}(m), \mathtt{treeprod(tree}(m))),$
        $\mathtt{ready}(b(self))))$

The following configuration contains an actor with behavior $B_{\mathtt{seqtp}}$ and is called $C_{\mathtt{seq}}$:

$$C_{\mathtt{seq}} = \langle\ [\mathtt{ready}(B_{\mathtt{seqtp}})]_{tp}\ \big|\ \rangle_{\emptyset}^{\{tp\}}$$

We will verify the correctness of the following configuration by showing its path correspondence to $C_{\mathtt{seq}}$:

$$C_{\mathtt{conc}} = \big\langle\ [\mathtt{ready}(B_{\mathtt{treeprod}})]_{tp}\ \big|\ \big\rangle^{\{tp\}}_{\emptyset}$$

The proof idea is to show that the sets of interaction paths of both configurations are the same. Although we can prove that the two configurations are equivalent in any environment, to simplify matters, we assume that all messages targeted to *tp* are well-formed, that is, they consist of a pair of an actor name and a finite binary tree with leaves containing integers. Moreover, the external customers always send external actors as the customer name. This way we don't have to worry about requests with *tp* in the customer field.

We also assume that the function `treeprod` is correct in the sense that it terminates and returns the product of the numbers at the leaves of the tree. These can be proved by simple induction.

Definition 9 *Let c be an actor name and t be a binary tree with integers at its leaves. We say an input transition label $\tau_{in} = \textbf{\textit{<in:}}tp \triangleleft \textbf{\textit{mkprd}}(c,t)\textbf{\textit{>}}$ has a matching output transition label $\tau_{out} = \textbf{\textit{<out:}}c \triangleleft p\textbf{\textit{>}}$ if p is the product of the leaves of t.*

Definition 10 *We say a (possibly finite) path $\pi = [\tau_1, \tau_2, \dots]$ is a **tree-product path** if it satisfies the following properties:*

P1 : Every input transition label τ_i has the form $\textbf{\textit{<in:}}tp\triangleleft\textbf{\textit{mkprd}}(cust, tree)\textbf{\textit{>}}$ where cust is an actor name different from tp, and tree is a finite binary tree with integers as its leaves. And every output transition label τ_j has the form $\textbf{\textit{<out:}}cust \triangleleft p\textbf{\textit{>}}$, where cust is an actor name different from tp and p is an integer.

P2 : Let

$$\begin{aligned} I &= \{i \mid \tau_i \text{ is an input transition label}\} \\ J &= \{j \mid \tau_j \text{ is an output transition label}\} \end{aligned}$$

There exists a bijection $f_\pi : I \to J$ such that for all $i \in N$ $f_\pi(i) > i$ and that $\tau_{f_\pi(i)}$ is a matching output for τ_i.

Lemma 1 *Every path π of C_{seq} is a* tree-product *path.*

PROOF: We need to prove that any path $\pi = [\pi_1, \pi_2, \dots]$ of $C_{\mathtt{seq}}$ has properties P1 and P2.

According to the `<in:>` rule, only messages targeted to actors in the reception set can enter a configuration. Therefore, only messages sent to *tp* can enter $C_{\mathtt{seq}}$. We also assumed that all messages are pairs of a customer

actor and a tree. It is also immediate from the code that the only kind of message sent out of the configuration is of the form $cust \lhd p$ for some actor name $cust$, which is never tp, and some integer p. Therefore, property P1 holds.

To prove P2, let $\tau_i = $ <in:$tp\lhd$mkprd($cust, tree$)> be some input transition which will put the message mkprd($cust, tree$) in the configuration. Fairness assumption implies that this message will eventually be delivered to tp. From fairness assumption again, we know that tp's computation can always proceed. And as the behavior of tp is terminating, a message of the form $cust \lhd p$, with p being the tree product of $tree$, will finally be sent out. This message in turn triggers a transition of the form $\tau_j = $ <out:$cust \lhd p$> with $j > i$. We can form a map f by mapping all such i's to their corresponding j's. This map will be a bijection as tp's behavior can not generate more than one message per each request and there is no pending outgoing messages in the original configuration.

Definition 11 *For configurations C, C', we say $C \Longrightarrow C'$ if $C = C'$ or for some sequence of configurations $C^1, \ldots, C^n$, and transition labels $l_1, \ldots, l_n$ that are neither input nor output labels, $n > 0$, we have $C_{seq} \xrightarrow{l_1} C^1_{seq} \xrightarrow{l_2} \ldots \xrightarrow{l_n} C'_{seq}$. Further, for an input or output transition label τ we say $C \overset{\tau}{\Longrightarrow} C'$ if for some configurations C_1, C_2, we have $C \Longrightarrow C_1 \xrightarrow{\tau} C_2 \Longrightarrow C'$.*

Thus, if $C \Longrightarrow C'$ then configuration C can evolve into C' without interacting with its environment, and if $C \overset{\tau}{\Longrightarrow} C'$ then C can evolve into C' by performing a single (input or output) interaction with its environment.

Lemma 2 *Let $\pi = [\tau_1, \tau_2, \ldots]$ be a (possibly finite) interaction path that satisfies properties P1 and P2. Let's pick f_π to be some bijection as referred to in P2. There exists a sequence of configurations $C^0_{seq}, C^1_{seq}, \ldots$ with $C^0_{seq} = C_{seq}$, such that for every $n > 0$, C^n_{seq} has the following properties:*

    S1 The actor tp is in ready state in C^n_{seq}.

    S2 For every input transition $\tau_i = $ <in:$tp \lhd$ mkprd($cust, tree$)>, the message instance $tp \lhd$ mkprd($cust, tree$) corresponding to transition τ_i is undelivered in C^n_{seq} if and only if $f_\pi(i) > n$.

    S3 $C^{n-1}_{seq} \overset{\tau_n}{\Longrightarrow} C^n_{seq}$

PROOF: We prove this by constructing a recursive function g that maps an interaction path π to a sequence of configurations satisfying the three properties stated in the lemma.

Since f_π is a bijection we have $\tau_1 = $ `<in:`$tp \triangleleft$`mkprd(`$cust, tree$`)>` for some *cust* and *tree*. Let $g(\tau_1) = C^1_{\tt seq}$ where

$$C^1_{\tt seq} = \langle\, [\mathtt{ready}(B_{\mathtt{seqtp}})]\,_{tp}\ \big|\ tp \triangleleft \mathtt{mkprd}(cust, tree)\,\rangle^{\{tp\}}_\emptyset$$

From the transition rules in Figure 8.3 we can conclude that $C^0_{\tt seq} \overset{\tau_1}{\Longrightarrow} C^1_{\tt seq}$. It is easy to verify that $C^1_{\tt seq}$ satisfies properties S1, S2, and S3.

Next we define $g(\tau_n)$ for $n > 1$. We distinguish two cases:

- $\tau_n = $ `<in:`$tp \triangleleft \mathtt{mkprd}(cust, tree)>$ (for some *cust* and *tree*): Let $g(\tau_n) = C^n_{\tt seq}$ be the configuration obtained by adding the message $tp \triangleleft (cust, tree)$ to the messages in $C^{n-1}_{\tt seq}$. Then $C^{n-1}_{\tt seq} \overset{\tau_n}{\longrightarrow} C^n_{\tt seq}$. It is easy to verify that $C^n_{\tt seq}$ satisfies properties S1, S2 and S3.
- $\tau_n = $ `<out:`$cust \triangleleft p>$ (for some *cust* and p): Let $i = f^{-1}_\pi(n)$. Therefore, $\tau_i = $ `<in:`$tp \triangleleft \mathtt{mkprd}(cust, tree)>$ for some *tree* with $p = \mathtt{treeprod}(tree)$. Assuming that $C^{n-1}_{\tt seq} = g(n-1)$, we can state the rest of the proof in the following steps:
  - (i) We know that in $C^{n-1}_{\tt seq}$, the message corresponding to τ_i has not been delivered. This follows from S2 and the fact that $f_\pi(i) > n-1$.
  - (ii) From S1 we know that tp is ready in $C^{n-1}_{\tt seq}$.
  - (iii) $C^{n-1}_{\tt seq}$ can perform a `<rcv:`$tp, tp \triangleleft \mathtt{mkprd}(cust, tree)>$, followed by a number of `<fun:`$tp>$, and finally a `<send:`$tp, cust \triangleleft p>$. This follows from fairness and the assumption that tp's behavior terminates and returns the tree product of *tree*.
  - (iv) The resulting configuration after the send transition contains an outgoing message of the form $cust \triangleleft p$. So an output transition with label τ_n can be performed. Hence, $C^{n-1}_{\tt seq} \overset{\tau_n}{\Longrightarrow} C^n_{\tt seq}$.
  - (v) It remains to show that $C^n_{\tt seq}$ satisfies S1, S2, and S3. From the code it follows that tp becomes ready to receive next message after sending the message. So S1 holds. S2 holds as the only message delivered in this step was the one corresponding to τ_i. And $f_\pi(i) = n$. S3 follows from the previous step of the proof. So we can let $g(n) = C^n_{\tt seq}$.

The construction described above forces a certain scheduling order on transitions. This order is fair since no enabled transition remains enabled forever.

Lemma 3 *Every* tree-product *path π can be observed from an execution of* C_{seq}.

PROOF: The lemma follows from lemma 2 and the observation that the computation path constructed in the proof of lemma 2 is fair. The interaction path π is just the observable projection of the computation path constructed in lemma 2.

Lemma 4 (correctness of $B_{\texttt{treeprod}}$) *Applying $B_{treeprod}$ to a message of the form $tp \triangleleft (cust, tree)$ will eventually result in sending exactly one message of the form $cust \triangleleft p$ where p is the tree product of tree.*

PROOF: Proof is by induction on the height of the tree. From $B_{\texttt{treeprod}}$ we can easily see that when the tree is just a leaf, its value is returned to the customer, hence validating the truth of the lemma for trees of height zero.

For $n > 0$, assuming that the lemma is true for trees of height smaller than n, we prove that the lemma is true for trees of height n. Recall that by our assumption every internal node, and hence the root, of the tree has two children. Following the fairness assumption, The rest of the proof will use the fact that actors' internal computation can always make progress.

The code creates a join-continuation actor, initialized with the customer's name. From the code we can infer the following facts:

- Only one join-continuation actor is created per input message.
- Customer's name is not used by $B_{\texttt{treeprod}}$ in any other part of the code.
- A continuation actor sends exactly one message to its customer iff it receives two messages containing integers.

The actor tp sends two messages to itself with two parameters: the join-continuation actor's name as the customer, and the left (or right) subtree. Both subtrees have a smaller height than the original tree, therefore according to the induction hypothesis, the product of their leaves will eventually be sent to the join-continuation actor.

As no one else is aware of the join-continuation actor's name, these two messages will be the only messages delivered to it. This conclusion plus the three facts above imply that the join-continuation actor will eventually send exactly one message to the original customer (from the fairness requirement, this message will be delivered). The content of the message is the product of the two numbers sent to the join-continuation actor, which in turn are the tree-products of the left and right subtrees.

Lemma 5 *Every path π of C_{conc} is a* tree-product *path.*

PROOF: The proof is the same as that for $C_{\texttt{seq}}$ except that in the argument for P2, we use lemma 4 instead of correctness of treeprod. Note that in

the argument for P1 it is essential to show that *tp* is the only receptionist at any time. This follows from the observation that every message sent by *tp* and its join continuations to other than self contain just an integer. This implies that names of internal actors are never sent to external actors.

Lemma 6 *Every* tree-product *path π can be observed from an execution of* C_{conc}.

PROOF: The same proof as for C_{seq}, except that lemma 4 is used instead of correctness of `treeprod`.

Theorem 1 C_{seq} *and* C_{conc} *are equivalent under path correspondence.*

PROOF: Immediate from the lemmas 1, 3, 5 and 6, and the definition of equivalence under path correspondence.

8.7 Discussion

We defined an equivalence notion based on interaction paths. Although equivalence based on interaction paths appears to be intuitive, in fact it distinguishes between more configurations than is reasonable. Consider a configuration whose behavior is represented by a tree consisting of an infinite path and another configuration whose behavior is represented by a tree consisting of all finite approximations to the path. These two trees cannot be distinguished in any actor context but do not have the same interaction paths. As briefly discussed earlier, an equivalence notion based on observations in arbitrary contexts was introduced in [AMST96].

We also described a proof technique for establishing equivalence between configurations based on interaction path correspondence. In earlier work, a proof technique was developed for establishing equivalence in more concrete terms, namely by establishing correspondence of the actual paths. A number of results were obtained in that work to show how reasoning could be simplified. These results rely on the ability to exploit asynchrony to shuffle transitions in a way that localizes differences in computations, and to use the concept of *holes* to formalize the aspects of computations that are independent of the local differences.

When compared to many other models of concurrency, the Actor model is very powerful: it supports local procedural and data abstraction, and provides a simple interface which abstracts the underlying name space management, scheduling, network, etc. The assumption of asynchrony often allows

only canonical message orders to be considered. The concept has been useful in diverse areas such as building animation languages, simulations, and enterprise integration systems.

On the other hand, the model is too low-level to allow us to easily reason about complex distributed software systems. For the same reason, such systems also remain very hard to specify and the software is often error-prone. We have argued that part of the reason for this difficulty is the fact that models of concurrency lack abstractions which represent the interaction patterns in a modular fashion. For example, we described the notion of local synchronization constraints which used to control the scheduling of messages at an actor based on the actor's state. More generally, such scheduling may have to constrained based on the history of a computation in a number of actors.

We have developed a number of such abstractions for specifying temporal coordination between actors [FA93, Fro96], real-time systems [Ren97], distributed interactions [Stu96], and dynamic communication groups [AC93, Cal94]. Such abstractions rely on a meta-architecture which allows dynamic customization of schedulers, name servers, and communication interfaces [AA98]. We believe that such architectures can promote more development of distributed systems as well as simplify the task of reasoning about systems by making both more modular. Some preliminary work in this area uses a two-level semantics [VT95]. However, the development of compositional methods for reasoning, as well as new specification techniques (for example, see [Smi98], remains an active area of research.

Acknowledgements

Bibliography

[AA98] Mark Astley and Gul Agha. Customization and composition of distributed objects: Middleware abstractions for policy management. *Sixth International Symposium on the Foundations of Software Engineering ACM SIGSOFT*, 23(6):1–9, November 1998.

[AC93] G. Agha and C.J. Callsen. ActorSpace: An open distributed programming paradigm. In *Principles and Practice of Parallel Programming '93*, 1993.

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.

[Agh90] Gul Agha. Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[AJ99] G. Agha and N. Jamali. Concurrent programming for distributed artificial intelligence. In Gerhard Weiss, editor, *Multiagent Systems: A Modern Approach to DAI*. MIT Press, 1999.

[AKP95] G. Agha, W. Kim, and R. Panwar. Actor languages for specification of parallel computations. In *DIMACS Series in Discrete Mathematics and Computer Science*, volume 18, pages 239–258. American Mathematical Society, 1995.

[AMST96] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 1996. to appear.

[Bou92] G. Boudol. Asynchrony and the pi-calculus. Technical Report 1702, Department of Computer Science, Inria Univeristy, May 1992.

[Cal94] Christian J. Callsen. *Open Heterogeneous Distributed Computing*. PhD thesis, Aalborg University, August 1994.

[FA93] Svend Frølund and Gul Agha. A language framework for multi-object coordination. In *Proceedings of ECOOP 1993*. Springer Verlag, 1993. LNCS 707.

[FF86] M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

[Fro96] S. Frolund. *Coordinating Distributed Objects: An Actor-Based Approach for Synchronization*. MIT Press, November 1996.

[HT91] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 133–147. Springer-Verlag, 1991.

[KA95] W. Kim and G. Agha. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In *Supercomputing '95*. IEEE, 1995.

[Kim97] W. Kim. *THAL: An Actor System for Efficient and Scalable Concurrent Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1997. http://www-osl.cs.uiuc.edu/.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil93] R. Milner. Elements of interaction turing award lecture. *Communications of the ACM*, 36(1):78–89, January 1993. Turing Award Lecture.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[MT99] I. A. Mason and C. L. Talcott. Actor languages their syntax, semantics, translation, and equivalence, 1999. to appear.

[Nee89] R.M. Needham. Names. In S. Mullender, editor, *Distributed Systems*, pages 89–101. Addison-Wesley, 1989.

[Ren97] Shangping Ren. *An Actor-Based Framework for Real-Time Coordination*. PhD thesis, Department Computer Science. University of Illinois at Urbana-Champaign, 1997.

[San98] D. Sangiorgi. An Interpretation of Typed Objects into Typed Pi-Calculus. *Information and Computation*, 143(1), 1998.

[Smi98] Scott Smith. On specification diagrams for actor systems. In C. Talcott, editor, *Proceedings of the Second Workshop on Higher-Order Techniques in Semantics*, Electronic Notes in Theoretical Computer Science. Elsevier, 1998.

[Stu96] Daniel C. Sturman. *Modular Specification of Interaction Policies in Distributed Computing*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

[SWP99] Peter Sewell, Pawel T. Wojciechowski, and Benjamin C. Pierce. Location Independent Communication for Mobile Agents: A Two Level Architecture. Technical Report 462, Computer Laboratory, University of Cambridge, 1999.

[Tal96] C. Talcott. Interaction Semantics for Components of Distributed Systems. In E.Najm and J.B. Stefani, editors, *Formal Methods for Open Object Based Distributed Systems*. Chapman & Hall, 1996.

[Tha00] Prasannaa Thati. Towards an Algebraic Formulation of Actors. Master's thesis, University of Illinois at Urbana-Champaign, 2000.

[TKS$^+$89] C. Tomlinson, W. Kim, M. Schevel, V. Singh, B. Will, and G. Agha. Rosette: An Object-Oriented Concurrent System Architecture. *Sigplan Notices*, 24(4):91–93, 1989.

[VA98] C. Varela and G. Agha. What after java? *Computer Networks and ISDN Systems: The International J. of Computer Telecommunications and Networking*, 1998.

[Var00] C. Varela. *World Wide Computing with Universal Actors: Linguistic Support for Coordination, Naming and Migration*. PhD thesis, University of Illinois at Urbana-Champaign, August 2000.

[VT95] N. Venkatasubramanian and C. L. Talcott. Reasoning about Meta-Level Activities in Open Distributed Systems. In *Principles of Distributed Computing*, 1995.

[Wal95] D. Walker. Objects in the Pi-Calculus. *Information and Computation*, 116(2):253–271, 1995.