10

Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems

Gul A. Agha
Open Systems Laboratory
Department of Computer Science, 1304 W. Springfield Avenue, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA, Email: agha@cs.uiuc.edu, *Web:* http://www-osl.cs.uiuc.edu

Abstract
This paper discusses mechanisms addressing the complexity of building and maintaining Open Distributed Systems. It is argued that a new programming paradigm based on modular specification of interaction patterns is required to address the complexity of such systems. Our research is based on developing abstraction mechanisms to simplify the task of developing and maintaining open systems. We define actors as a model of concurrency for open systems. We then review a number of programming abstractions that are useful in modular specification and implementation of open systems. Such abstractions include activators, protocols, synchronizers, and actorspaces. We observe that defining such abstractions for complex interaction patterns requires a sort of system decomposition that is not supported by standard execution models of concurrent programming, including actors and process algebras. Rather, a suitable meta-architecture is needed to allow the implementation of abstract representations of interaction patterns. Currently there is no entirely satisfactory formal theory of meta-architectures.

Keywords
Actors, concurrency, object-oriented programming, open systems, real-time systems, fault-tolerance, coordination, reflection.

1 INTRODUCTION

Systems in the real world consist of many distributed, asynchronous components which are *open* to interaction with their environment. We will call such systems *open (distributed)* *systems.* The functionality of an open system is not defined by the result of evaluating an expression; instead the relative state of components, the relative timing of actions, locality and distribution of the computation, etc., are all critical to the correctness of the system. Moreover, open 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 and components to be connected with new components while they are still executing.

Complex interaction patterns arise between different components of an open system. Our contention is that to simplify the task of implementing open systems in the real world, we must be able to abstract different patterns of interaction between components. On the other hand, models of concurrency are generally based on a rather low-level execution model – namely, message passing as the mechanism to support interaction between components.* Unfortunately, programming using only message passing is somewhat worse than programming in assembler: sending a message is not only a jump, it may spawn concurrent activity! The goal of our research is to find a set of abstractions which enable interaction patterns between concurrent components to be represented by modular and reusable code.

We use Actors as our model of concurrent computation. But the problem I address, namely how to reliably build complex open systems, and the techniques I develop are equally applicable to any existing model of concurrency. The problem with conventional models of concurrency, and programming paradigms based on these models, is that they do not allow one to write abstract, reusable code which captures interaction patterns.

Consider requirements such as security, availability, and atomicity. How can we write code for such requirements in a modular manner, i.e., by defining a module which when composed with an arbitrary application guarantees a certain interaction policy for the application? For example, a security policy may be implemented by encrypting and decrypting messages; current techniques require that we modify the code for each actor to implement the encryption and decryption. What we would like to do is simply define a module for encryption/decryption and compose it with an arbitrary group of actors to impose a security policy on that group of actors. We may need to impose different security policies on different groups of actors or even on the same group at a different point in time. So what we want is the ability to write a reusable module for security, one which is neither hand-coded for each application, nor hard-coded in the runtime system.

I will begin with an introduction to actors and then return to the problem of separation of design concerns, abstraction, and composition that I have outlined above. The treatment in this paper is of necessity rather high-level; the goal is to provide an overview of our research. Interested readers should refer to the citations for technical details of the work as well as secondary references to the literature.

*I will ignore shared variable models; such models violate data encapsulation, an essential feature for modular software development.

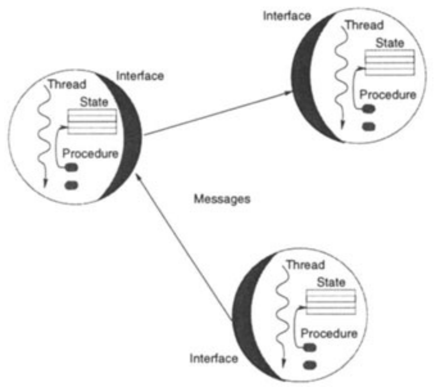

Figure 1 Actors encapsulate a thread and state. The interface is comprised of public methods which operate on the state.

2 ACTORS

The Actor model provides a flexible method for representing computation in real-world systems. Actors extend the concept of objects to concurrent computation (Agha, 1986). 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. Each actor potentially executes in parallel with other actors and may send messages to actors it knows the addresses of. Actor addresses may be communicated in messages, allowing dynamic interconnection. Finally, new actors may be created; such actors have their own unique addresses.

It is possible to extend any sequential language with the actor constructs. For example, the call-by-value λ-calculus is extended in (Agha et al., 1996). Specifically, the following operators are added to expressions:

send(a, v) creates a new message:
- with receiver a, and
- contents v

newactor(e) creates a new actor:
- which is evaluating the expression e, and
- returns its address

ready(b) captures local state change:

- alters the behavior of the actor executing the `ready` expression to b
- frees that actor to accept another message.

A `let` construct may be used to allow mutual reference of newly created actors. The behavior of an actor is represented by a `lambda` abstraction, the acceptance of a message by function application, i.e., app(b,m), which represents b applied to message m. The motivation behind the actor constructs is to provide the minimal extension that is necessary to extend a sequential language to a concurrent one supporting object-style encapsulation and coordination.

Instantaneous snapshots of actor systems are called *configurations*; actor computation 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 a configuration and a set of actors *external* to a configuration which may receive messages from the actors within.

Definition (Actor Configurations): An *actor configuration* with actor map, α, multiset of messages, μ, receptionists, ρ, and external actors, χ, is written

$$\langle \alpha \mid \mu \rangle_{\chi}^{\rho}$$

where ρ, χ are finite sets of actor addresses, α maps a finite set of addresses to their behavior, μ is a finite multiset of (pending) messages, and if $A = \mathrm{Dom}(\alpha)$, i.e., the domain of α, then:

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

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

An actor may be in one of two kinds of states: busy or ready to accept a message. For an actor with address a, we indicate these states as follows:

- $(b)_a$ ready to accept a message, where b is a lambda abstraction representing its behavior;
- $[e]_a$ busy executing e, e represents the actor's current (local) processing state.

Now we can extend the local transitions defined for a sequential language ($\overset{\lambda}{\mapsto}$), by providing transitions for the actor program as shown in Figure 2 (assume that R is the reduction context in which the expression currently being evaluated occurs).

Based on a slight variant of the transition system described above, a rigorous theory of actor systems is developed in (Agha et al., 1996). Specifically, we define and study various notions of testing equivalence on actor expressions and configurations. The model we have developed provides fairness, namely that any enabled transition eventually fires. 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

$$e \overset{\lambda}{\mapsto}_{\mathrm{Dom}(\alpha)\cup\{a\}} e' \Rightarrow \langle\, \alpha, [e]_a \mid \mu \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha, [e']_a \mid \mu \,\rangle^{\rho}_{\chi}$$

$$\langle\, \alpha, [R[\mathtt{newactor(e)}]]_a \mid \mu \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha, [R[a']]_a, [e]_{a'} \mid \mu \,\rangle^{\rho}_{\chi} \qquad a' \text{ fresh}$$

$$\langle\, \alpha, [R[\mathtt{ready}(v)]]_a \mid \mu \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha, (v)_a \mid \mu \,\rangle^{\rho}_{\chi}$$

$$\langle\, \alpha, [R[\mathtt{send}(v_0, v_1)]]_a \mid \mu \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha, [R[\mathtt{nil}]]_a \mid \mu, m \,\rangle^{\rho}_{\chi} \qquad m = \texttt{<}v_0 \Leftarrow v_1\texttt{>}$$

$$\langle\, \alpha, (v)_a \mid \texttt{<}a \Leftarrow cv\texttt{>}, \mu \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha, [\mathrm{app}(v, cv)]_a \mid \mu \,\rangle^{\rho}_{\chi}$$

$$\langle\, \alpha \mid \mu, m \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha \mid \mu \,\rangle^{\rho'}_{\chi}$$
$$\text{if } m = \texttt{<}a \Leftarrow cv\texttt{>},\ a \in \chi, \text{ and } \rho' = \rho \cup (\mathrm{FV}(cv) \cap \mathrm{Dom}(\alpha))$$

$$\langle\, \alpha \mid \mu \,\rangle^{\rho}_{\chi} \mapsto \langle\, \alpha \mid \mu, m \,\rangle^{\rho}_{\chi\cup(\mathrm{FV}(cv)-\mathrm{Dom}(\alpha))}$$
$$\text{if } m = \texttt{<}a \Leftarrow cv\texttt{>},\ a \in \rho \text{ and } \mathrm{FV}(cv) \cap \mathrm{Dom}(\alpha) \subseteq \rho$$

Figure 2 Actor transitions.

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; thus, if upon composition, new 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").

The notion of equivalence is defined by adding an observable distinguished *event* to the set of transitions. This technique is a variant of operational equivalence defined in (Plotkin, 1975). Two actor expressions may be plugged into a context to see if the event occurs in one or the other case. Two expressions are considered equivalent if they have the same observations over all possible contexts.

The nondeterminism in the arrival order of the messages in an actor computation gives rise to three notions of observation over a computation tree. Notice there are many computational paths in the tree. Now it is possible that the event occurs in every computational path (*must* happen); occurs in some but not all computational paths (*may* happen), or never occurs.

Three distinct well-known equivalence relations may now be defined. In *may* equivalence, *always occurs* is as good as *sometimes occurs* (*i.e.* that is, either is a sufficient condition for proving equivalence); in *must* equivalence *never occurs* is as good as *only sometimes occurs*. *Convex* equivalence requires the two sets to coincide (the intersection of the two equivalences). An important result is that, in the presence of fairness, the three forms of equivalence collapse to two, namely, *may* and *convex*. Thus, while fairness makes some aspects of reasoning harder – we cannot simply use co-induction in proofs – it simplifies others.

We have developed methods for proving laws of equivalence and have developed proof techniques that simplify reasoning about actor systems. In particular, the proof techniques allow us to use canonical multi-step transitions as well as reduce the number of contexts that need to be considered. Finally, note that the composition of configurations defines an algebra.

A concrete way to think of actors is that they represent an abstraction over concurrent architectures. An actor runtime system provides the interface to services such as global addressing, memory management, fair scheduling, and communication. It turns out that these services can be efficiently implemented, thus raising the level of abstraction while reducing the size and complexity of code on concurrent architectures (Kim and Agha, 1995).

Note that the Actor model is, like the theory of higher order nets or the π-calculus, general and inherently parallel. Moreover, asynchronous communication in actors directly preserves the available potential for parallel activity: an actor sending a message does not have to necessarily block until the recipient is 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 series of asynchronous messages (Agha, 1990).

A key difference between actors and objects defined using ports in the π-calculus is that actor names (addresses) are uniquely tied to the identity of an actor – giving out an actor name does not enable the recipient to receive messages directed to that actor. It is also possible to model a system of actors by means of a higher order net and vice versa (Sami and Vidal-Naquet, 1991), although such a model does not satisfactorily account for the open systems aspects of actors.

3 LOCAL SYNCHRONIZATION CONSTRAINTS

Consider a system of actors consisting of producers and consumers. A *producer* puts elements in a *buffer* and a *consumer* consumes them. Thus a buffer has two methods – namely get and

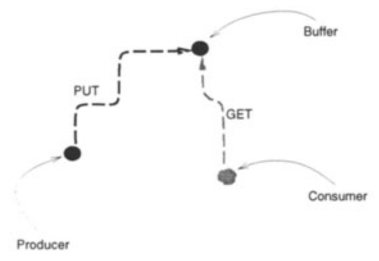

Figure 3 Synchronization between a producer, consumer and intermediate buffer.

put: a producer invokes the put method to store in the buffer a value that it has generated while a consumer retrieves the next available value by sending a get message to the buffer actor (see Figure 3). An example of such a buffer is a print spooler.

Because actors are asynchronous and autonomous, when a consumer does a get, it does not know if the buffer has any elements in it. In languages with synchronous communication, the buffer process would have an input guard over a channel on which it services requests from consumers and the guard would prevent messages from coming in when the buffer process is empty. The consumer then has to busy wait until such time as the buffer is no longer empty. This solution is generally undesirable: it wastes net bandwidth and complicates programming. Instead, we would like the request from the consumer to be locally buffered until the buffer is ready to process get requests (see figure 4. The figure is from (Frølund, 1992)).

A local synchronization constraint is a programming construct which allows a declarative expression of a complex sequence of actions involving testing and storing messages. Using the abstraction, we may write the desired message ordering constraint as a logical formula over the state of an actor; we do not have to program the details of testing and storing messages to enforce the ordering (which can get quite complicated, particularly if optimizations are used to improve efficiency). The programming construct also encourages higher-level reasoning about the system, one which abstracts the low level implementation details.

Abstractly, an actor configuration now includes a constraint map. For each actor and message, the constraint map yields a truth value. If the constraint holds, the message may be processed; otherwise the transition to receive the message cannot fire. The constraint map changes as the state of an actor changes. The semantics of local synchronization constraints will be subsumed by a more complex multi-actor coordination constraint construct that we describe in Section 7.

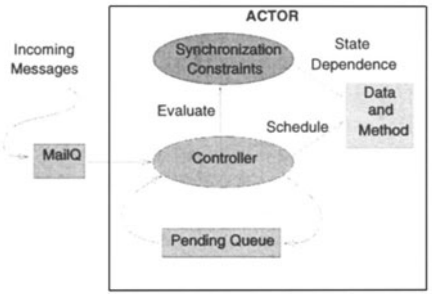

Figure 4 Implementing synchronization constraints.

4 COMMUNICATION AND CONTINUATIONS

It is often possible to encode the computation and communication behavior defined in one model of concurrency in terms of another while preserving the intuitive properties of the behavior. This suggests that there is some sort of equivalence between many different, relatively powerful models of concurrency, although we do not have a consensus about what defines a universal model for concurrency. However, from a programming language point of view, such generality is not enough.

Consider the fact that although asynchronous communication is a natural way of representing primitive interaction in distributed systems, other forms of communication, such as remote procedure calls, are often useful. In actor terms, an rpc-like communication is represented by an actor entering a "wait" state after sending a request to another actor; in this state all messages other than a reply from the called actor are deferred. In other words, expressing RPC-like communication in terms of primitive actors, requires unfolding the continuation behavior into a separate actor. The need for such transformation places an unnecessary burden of book-keeping on the programmer.

No model of concurrency can, and perhaps should, allow all communication abstractions to be directly expressed; in fact, different models make some forms easier to express than others. For example, a communication abstraction that is relatively simple to express in Petri Nets as well as Guarded Horn Clause (or Concurrent Logic) languages is invocation by a set of messages. This communication represents a common schema where an actor carries out some useful actions only after some arbitrary set of messages has been received (*input synchronization*). A simple example of input synchronization is a monitor which responds to conditions based on readings of two different sensors (see Figure 5).

In a high level actor language, a new programming abstraction must be introduced to

```
method survey(sens-1,sens-2 : recep( real ); )
        var r1,r2 : real;
        while true
              sens-1 ? r1 and sens-2 ? r2 where non-safe(r1,r2)
                      → process-data(r1,r2);
end survey
```

Figure 5 A survey method in an activator monitors values received by two sensors, sens-1 and sens-2. If the values indicate a non-safe state, action is taken and the values are removed. In an activator, only one method acts on given data at a time, thus one match atomically removes the values.

allow an abstract specification of a behavior that is activated by a set of messages rather than a single message. In (Frølund and Agha, 1996), we define such an abstraction, called *activators*, which allows specification of both *input synchronization* and *reply synchronization*; reply synchronization generalizes rpc-like communication to support the concurrent invocation of a group of actors. A compiler can translate activators into a collection of actors which maintain the local state necessary to capture the partial set of messages received. A difference between activators and concurrent logic languages is the atomic removal of messages matching the pattern in a method; in other words, messages in activators are *use once* rather than instantiations of the bindings of shared variables as in concurrent logic programming. The idea is to provide mutual exclusion of independantly specified actions. This is reminiscent of a token being removed in a petri net transition.

5 INTERACTION POLICIES

An interaction policy may be expressed in terms of the interfaces of actors and implemented by using appropriate protocols to coordinate between actors (see Figure 6).

Consider a common interaction policy, namely, atomicity. Atomicity may be realized by using a protocol such as *two phase commit.* Notice that the implementation of a two phase commit is quite involved: it requires exchanging a number of messages between different actors. Current techniques for developing distributed software require developers to implement interaction policies and application behavior together, significantly complicating code. The lack of modularity not only makes it hard to reason about code, it limits its re-usability and portability. Moreover, the resulting code is brittle: modifying an interaction policy to satisfy changing requirements requires modifying the code of each relevant component and then reasoning about the entire system, essentially from scratch. One cannot, for example, simply pull out a two phase commit and replace it with a three phase commit.

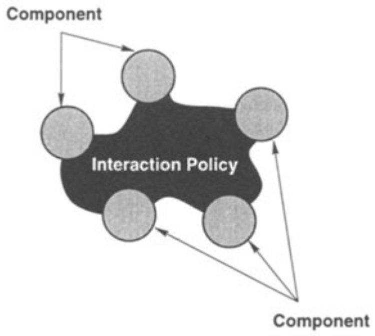

Figure 6 A distributed system consists of a set of components carrying out local computations and interacting in accordance with a set of *policies.*

Composition of configurations is not sufficient to capture the kind of composition we need here. In the first place, in standard actor semantics, we cannot even express a two-phase commit protocol by a configuration of actors; what we need to express a two phase commit as an abstraction is the ability to write meta-programs with distributed scope. A two phase commit meta-program imposes a role for each actor, specifically, trapping and tagging incoming and outgoing messages to implement the protocol. Such customization of an individual actor's mail system may be further limited only for the duration of an interaction.

We implement interaction policies using a linguistic construct called protocol. As described in the next section, a protocol imposes a certain role on each actor governed by the protocol: in essence it mediates the interactions between actors to ensure that each relevant actor implements its end of the interaction policy.

6 PROTOCOLS

We have developed a programming language which allows protocols to be specified and linked to actors written in conventional actor languages (Sturman and Agha, 1994). In our language, a protocol abstraction, such as one for the two phase commit protocol, may be defined. The abstraction may be instantiated by specifying a particular group of actors and other initialization parameters. The runtime system then uses specific forms of reflection which are sufficient to enable the protocol to be enforced. Specifically, the runtime supports

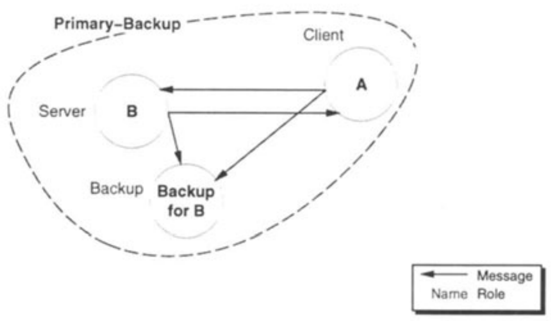

Figure 7 The Primary-Backup protocol. With respect to this protocol, the actor **A** serves role of client, **B** serves the role of a server and **Backup for B** serves the role of a backup. Interactions between these actors are mediated by a protocol which ensures the correct behavior of the protocol.

meta-level operations which dynamically modify the mail system and store and retrieve actor states.

Consider the primary backup protocol in Figure 7. By modifying the communication architecture, the protocol may be installed on any group of actors. An important advantage is that such installation may be dynamic and limited in duration. Thus, a protocol's behavior may change over time. We specify a protocol as a single object with a mutable local state and operations over the state; the operations may be used to change the configuration of the protocol. To specify a protocol, we define an abstraction which may be instantiated with different actors. Protocols do not depend on the internal representation of actors; they may however require the state of actors to be copied as a black box. The advantage of a message-passing model and decoupled components is that the ability to modify the communication architecture provides great flexibility in customizing interactions. Figure 8 shows the code for a primary backup protocol.

An important aspect of protocols is that protocol instances may be composed. Thus, a given actor may be in different roles with respect to different protocol instances. Figure 9 shows the composition of two protocols. Such composition is sometimes realized by layering in the meta-level architecture, much like the concept of filters (Aksit et al., 1993). A description of the implementation and semantics of protocols, including details about these examples, may be found in (Sturman, 1996).

Notice that the behavior of an actor system in the presence of protocols may be quite different from its behavior otherwise. Our pragmatic experience suggests that reasoning about

```
protocol PrimaryBackup {
  int redundancy;
  int ticks;

  initialize(actor svr, int red, int t) {
    ticks = t;
    redundancy = red;
    assume(svr,server);
    assume(svr.clone(),getRole(backup));
  }

  operation addClient(actor c) {
    assume(c,client);
  }

  role server {
    Copy all messages and periodically
      send state updates to backup;
  }

  role client {
    Removes tags from server messages;
  }

  role backup {
    Receive messages and state from server;
  }
}
```

Protocol Definition

```
role server {
  Timer t;
  int tag;

  initialize() {
    t.set(ticks,update);
    backup.update(snapshot());
    tag = 0;
  }

  event deliver(Msg m) {
    backup.deliver(m);
    deliverMsg(m);
  }

  event transmit(Msg m) {
    m.add(tag);
    tag = tag + 1;
    m.send();
  }

  method update() {
    backup.update(snapshot());
    t.set(ticks,update);
  }
}
```

Server Role

Figure 8 Primary-Backup Protocol. The protocol definition (left figure) defines roles for each participant in the protocol. Each role (*e.g.* right figure) defines protocol specific *methods* and *events* which the role implements. An event is a meta-level customization of a particular operation done by the runtime system for the actor assuming the given role.

distributed applications is simplified by our meta-programming system; after all, code size is reduced orders of magnitude, and an application is decomposed into more intuitive units corresponding to the more abstract requirements specification. Specifically, one can reason about the protocol behavior at an abstract level rather than at the level of its implementation.

However, the semantics of actor systems in the presence of meta-level operations remains poorly understood; in particular, the effect of such operations on the theory of actor equivalence is not understood. Recent research based on actors has made progress on the problem of reasoning in the presence of meta-actors, specifically, by defining a reasoning system and

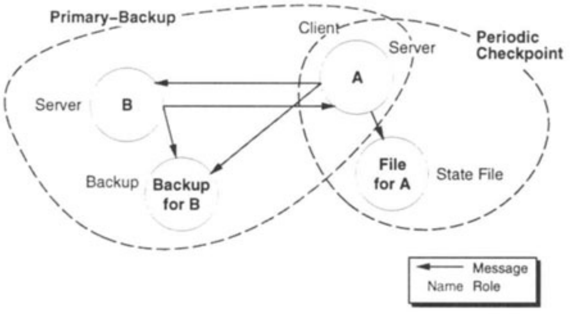

Figure 9 Primary-Backup overlapping with a Checkpoint protocol. The same actor may serve in different roles with respect to different protocol instances.

using it to prove the correctness of a meta-level algorithm for taking a global snapshot of a running distributed system of actors (Venkatasubramanian and Talcott, 1995).

7 DECLARATIVE COORDINATION CONSTRAINTS

To further abstract over possible mechanisms implementing interaction policies, we focus on coordination constraints. As a gross simplification, coordination constraints determine *when* actions take place rather than *what* individual actors do. Such constraints naturally generalize local synchronization constraints to multi-actor systems. The key idea is to maintain encapsulation of actors and express the coordination patterns in terms of the interfaces of the relevant group of actors.

It turns out that two types of coordination are often useful. One type imposes precedence constraints on otherwise asynchronous events at different actors, and the other requires such events to be atomic (loosely speaking, to co-occur). By providing a language abstraction, called a *synchronizer*, to express these two types of coordination constraints, we are able to show that the task of distributed programming may be further simplified (Frølund and Agha, 1993). A synchronizer may observe messages flowing through to a group of actors; it has a local state which may change as a result of these observations. Thus the local state does its own bookkeeping for the part of the history of a group that is relevant to the coordination between actors in the group. By doing the extra bookkeeping, we avoid violating the data encapsulation of individual objects and thus maintain their representa-

```
AllocationPolicy =
{
 init prev := 0

 enter(adm1,adm2,max)
    prev = max  disables (adm1.request  or adm2.request),
    (adm1.request  or  adm2.request)  updates prev  := prev + 1,
    (adm1.release  or  adm2.release)  updates prev  := prev - 1,
}
```

Figure 10 A synchronizer which enforces a global bound, **max**, on the number of resources allocated by two actors, **adm1** and **adm2**. We assume that each request uses a single resource from a pool of resources.

tion independence. Synchronizers add considerable flexibility to a system since they may be dynamically added and removed, and may be composed.

Figure 10 shows an example of a synchronizer. To maintain a consistent state of a synchronizer, serialization must be maintained over different possible groups of messages that affect the state of the synchronizer. A given implementation of synchronizers may satisfy this consistency requirement through different possible concurrency control mechanisms.

Besides the need for concurrency control, the fact that synchronizers may be superimposed, and may be dynamically added or removed, means that implementing a system to support synchronizers is fairly challenging. See (Frølund, 1996) for a discussion of one strategy for their efficient implementation. The interesting point to note is that the particular set of protocols used to implement synchronizers – i.e., provide fair scheduling, mutual exclusion, atomicity and so on – are all reused by different declarative coordination constraints imposed on an application using synchronizers.

As one might expect, the semantics of synchronizers fundamentally alters the nature of transitions in actor systems. Essentially, synchronizers provide a declarative mechanism for customizing the behavior of meta-level actors that collectively represent the scheduler. Observe that synchronizers do not add to the set of possible observable events in an actor system; they merely rule out certain interleavings of events.

A transition system for actors in the presence of synchronizers is also defined in (Frølund, 1996). The essential idea is to add a set of synchronizers, σ, to configurations. A function C_{map} maps σ to a function that takes a multiset of message target pairs to a boolean value. The receipt of a message is generalized to the receipt of a multiset of (one or more messages), where such receipt is only possible if the boolean value resulting from evaluating C_{map} function on the messages is true. The function T_{map} captures the change in the state of synchronizers in

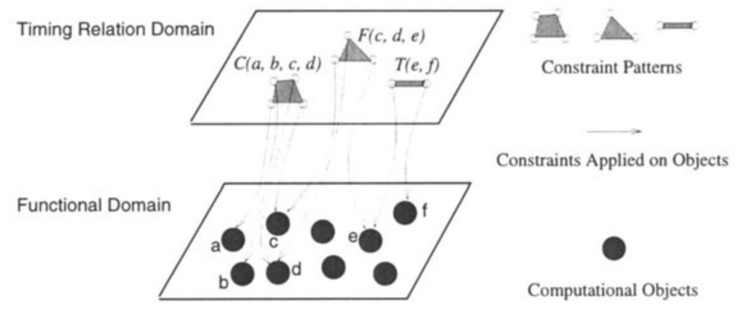

Figure 11 Mapping between the timing relation and functional domain.

response to the receipt of the messages. The transitions below capture the semantics of a synchronizer; the first one is a new transition, while the second modifies message reception. We use $\cup$, for union and $\uplus$ for multiset union.

$$\langle\, \alpha\,,\, [R[\texttt{enforce}(\texttt{synch}(r))]]_x \mid \mu \mid \sigma\, \rangle^{\rho}_{\chi} \mapsto \langle\, \alpha\,,\, [R[\texttt{nil}]]_x \mid \mu \mid \sigma \cup \{\texttt{synch}(r)\}\, \rangle^{\rho}_{\chi}$$

$$\frac{M = \{\langle a_1 \Leftarrow cv_1\rangle, \ldots, \langle a_n \Leftarrow cv_n\rangle\} \quad (C_{map}(\sigma))(M)}{\langle\, \alpha\,,\, (v_1)_{x_1}\,, \ldots,\, (v_n)_{x_n} \mid \mu \uplus M \mid \sigma\, \rangle^{\rho}_{\chi} \mapsto \langle\, \alpha\,,\, [v_1\; cv_1]_{a_1}\,, \ldots,\, [v_n\; cv_n]_{a_n} \mid \mu \mid T_{map}(\sigma, M)\, \rangle^{\rho}_{\chi}}$$

8 REAL-TIME REQUIREMENTS

Many real world systems need to respond in real-time. While synchronizers provide a mechanism for constraining the order of events, such constraints are ordinal rather than metric and thus unsuitable as a representation for real-time. However, using a linguistic abstraction similar to synchronizers can simplify the design, implementation and reasoning about distributed real-time systems. Specifically, such a programming abstraction must modularize timing properties, separating their specification from the representation of the functional behavior of actors. In contrast to conventional real-time language constructs, which express timing requirements on actions within an object, we think of timing requirements as constraints relative to message execution rather than as constraints internal to actors (Ren et al., 1996). Because real-time constraints are separately specified, using generic software in real-time systems becomes feasible (see Figure 11).

Semantics for concurrent programming languages usually focus on qualitative aspects, which is insufficient for real-time programming languages. It is critical for real-time applications that quantitative aspects are analyzed and explained. We have shown how an operational semantics of a simple actor language which has been extended to allow timing constraints on messages, may be translated into an underlying real-time formalism, namely, timed graphs. The net result is to provide the real-time semantics of an actor language (Nielsen and Agha, 1996). Moreover, the semantics provides a loose specification of programs, rather than a semantics of actual implementations. An implementation represents a refinement of this specification.

9 NAMING AND GROUPS

Groups of actors are an important unit of representation; for example, in defining protocols we can assign roles to a group of actors rather than an individual actor. The Actorspace model allows an abstract specification of a group of actors (Callsen and Agha, 1994). An actorspace associates an actor with specific attributes; the sender of a message specifies a destination pattern which is pattern matched against the attributes of actors in the actorspace.

A simple analogy with set theory illustrates the difference between naming in actors and actorspaces. A set may be defined by enumerating its elements, or by specifying a characteristic function which defines a subset in a domain. The first method is analogous to actor communication (where an explicit collection of mail addresses of actors must be specified), whereas the second method corresponds to actorspace communication. Of course, in conventional mathematics the two ways of characterizing sets are equivalent since the properties of mathematical objects are static; by contrast, actors may dynamically change their attributes.

Actorspace provides a transparent way of managing groups of actors. It generalizes the notion of ports in process calculi, in as much as a port name may not uniquely encapsulate an object. Moreover, Actorspace provides a multicast primitive. Multicast is useful, for example, when you want the group to change its collective behavior, as in enacting a new policy. Figure 12 shows a simple example of an actorspace. A car assembly requires certain types of parts which may be available through different vendors, sets that may themselves be changing over time. Which vendor fills a request may not be germane to the assembly process. Such requests may be mediated through an actorspace. Finally, meta-level operations may be associated with an actorspace. For example, an actorspace manager may transparently schedule requests to ensure load balancing.

10 CONCLUSIONS

I have discussed a number of ways in which the development of concurrent systems may be simplified by abstractions. In particular, I have argued the need for abstracting over patterns of interaction between meta-level actors, as well as the mechanisms they use to manage base-level actors. Such abstractions provide programming language support for defining higher-level coordination structures, reducing code size and complexity by orders of magnitude.

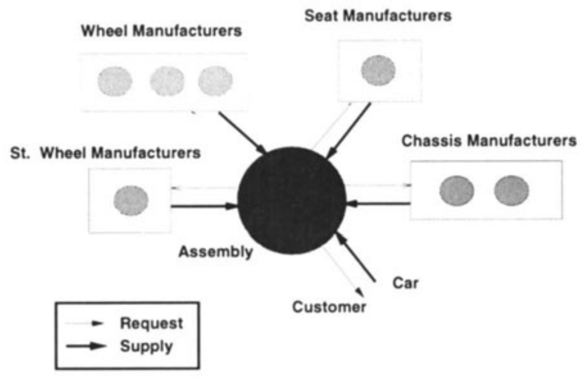

Figure 12 A car assembly factory. The assembly sends requests to actorspaces whose membership may dynamically change.

Programming abstractions are useful only if methods are developed for efficiently implementing them, as well as for reasoning about their composition. Observe that in our framework, it is not sufficient to think of composition as a way of plugging together different modules; rather the implementation of services in a runtime system must itself be thought of as a system of actors that is composed with an application. In other words, parts of what would conventionally be thought of as the runtime system may be customized and reconfigured.

Reflection provides an essential mechanism by which to implement higher level, application specific coordination structures in a dynamic and transparent way. I should emphasize that I do not regard unrestricted computational reflection, where a complete interpreter is available at runtime, as an application programmer's tool. Rather, it is necessary only to export certain meta-level operations to an application to enable limited runtime customization. Thus the implementation of interaction abstractions should be automated using the reflective architecture. In any event, considerable research in this area remains to be done before a useful programming methodology, supported by robust formal methods, becomes available.

ACKNOWLEDGEMENTS

Research described in this paper has been made possible by intense interaction with the past and present members of the Open Systems Laboratory including Mark Astley, Christian Callsen, Chris Houck, Svend Frølund, Shingo Fukui, Nadeem Jamali, WooYoung Kim, Brian Nielsen, Rajendra Panwar, Anna Patterson, Shangping Ren, Masahiko Saito, Susanne Schacht, R. K. Shyamasundar, Daniel Sturman, Takuo Watanabe, and Nalini Venkatasubramanian. I have also particularly benefited from discussions with Ian Mason, Scott Smith and Carolyn Talcott. This paper extends and incorporates (Agha, 1996). The research described has been supported in part by the Office of Naval Research (ONR contract numbers N00014-90-J-1899 and N00014-93-1-0273), the Digital Equipment Corporation, Hitachi, and the National Science Foundation (NSF CCR 93-12495).

REFERENCES

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

Agha, G. (1990). Concurrent Object-Oriented Programming. *Communications of the ACM*, 33(9):125–141.

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

Agha, G. A. (1996). Modeling concurrent systems: Actors, nets, and the problem of abstraction and composition. In Billington, J. and Reisig, W., editors, *Applications and Theory of Petri Nets 1996*, LNCS 1091, pages 1–10, Osaka, Japan. Springer-Verlag.

Aksit, M., Wakita, K., Bosch, J., Bergmans, L., and Yonezawa, A. (1993). Abstracting object interactions using composition filters. In Guerraoui, R., Nierstrasz, O., and Riveill, M., editors, *Object-Based Distributed Programming*, volume 791 of *Lecture Notes in Computer Science*, pages 152–184. ECOOP, Springer-Verlag.

Callsen, C. J. and Agha, G. A. (1994). Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, pages 289–300.

Frølund, S. (1992). Inheritance of synchronization constraints in concurrent object-oriented programming languages. In Madsen, O. L., editor, *Proceedings ECOOP '92*, LNCS 615, pages 185–196, Utrecht, The Netherlands. Springer-Verlag.

Frølund, S. (1996). *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization.* MIT Press.

Frølund, S. and Agha, G. (1993). A language framework for multi-object coordination. In *Proceedings of ECOOP 1993*, volume 707 of *Lecture Notes in Computer Science.* Springer Verlag.

Frølund, S. and Agha, G. (1996). Abstracting interactions based on message sets. In *Object-Based Models and Languages for Concurrent Systems*, volume 924, pages 107–124. Springer-Verlag. Lecture Notes in Computer Science.

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

Nielsen, B. and Agha, G. (1996). Semantics for an actor-based real-time language. In *Fourth International Workshop on Parallel and Distributed Real-Time Systems*, Honolulu. (to be published).

Plotkin, G. (1975). Call-by-name, call-by-value and the lambda calculus. *Theoretical Computer Science*, 1:125–159.

Ren, S., Agha, G., and Saito, M. (1996). A modular approach for programming distributed real-time systems. *Journal of Parallel and Distributed Computing.* (to appear).

Sami, Y. and Vidal-Naquet (1991). Formalization of the behavior of actors by colored petri nets and some applications. In *Conference on Parallel Architectures and Languages Europe, PARLE'91.*

Sturman, D. and Agha, G. (1994). A protocol description language for customizing failure semantics. In *The 13th Symposium on Reliable Distributed Systems, Dana Point, California.* IEEE.

Sturman, D. C. (1996). *Modular Specification of Interaction Policies in Distributed Computing.* PhD thesis, University of Illinois at Urbana-Champaign.

Venkatasubramanian, N. and Talcott, C. L. (1995). Reasoning about Meta Level Activities in Open Distributed Systems. In *Principles of Distributed Computing.*

BIOGRAPHICAL SKETCH

Gul Agha is Director of the Open Systems Laboratory at the University of Illinois at Urbana-Champaign. His research interests include models, languages, and tools for computing in open distributed systems. His book, *Actors: A Model of Concurrent Computing in Distributed Systems* stimulated considerable research in concurrent objects. Professor Agha is an ACM International Lecturer, Editor-in-Chief of *IEEE Concurrency*, and Associate Editor of *Theory and Practice of Object Systems* and *ACM Computing Surveys.* He is a recipient of the Incentives for Excellence Award from Digital Equipment Corporation, Naval Young Investigator Award from the US Office of Naval Research, and of a Fellowship at the University of Illinois Center for Advanced Study. Agha has served as Consulting Scientist to Microelectronics and Computer Technology Consortium and as a Visiting Professor at the University of Grenoble, France.