AN ACTOR FORMALISM FOR OPEN CONCURRENT SYSTEMS*

R.K. Shyamasundar[†‡], Anna Patterson[‡] and Gul Agha[‡]

[†] Computer Science Group
Tata Institute of Fundamental Research
Bombay 400 005, India

[‡] Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA 61801

November 19, 1991

Abstract

This paper describes a universal model of concurrent computation based on an Actor formalism. Apart from the usual actor primitives, namely, branching, message send, actor creation, and local state change, we introduce a hierarchical structuring operator called *hibernate*. We formally describe the transitional semantics of a kernel actor language extended with the hibernate operator. The hibernate operator provides structuring capabilities for an asynchronous formalism which are similar to those provided by restriction (or hiding) in port-based synchronous communication formalisms. The operator also provides a way to realize multi-party interactions such as broadcasts. Our formalism uses the distributed name generation scheme of actors; it thus avoids problems of scope intrusion and extrusion which occur in other models of concurrency. The formalism is illustrated by means of some examples including two embeddings of the call-by-name λ-calculus, respectively, lazy and with futures, and an example of multi-party interaction in a group of cooperating robots. Finally, we compare the proposed model with the chemical abstract machine model and discuss some open problems.

1 INTRODUCTION

Functional and object-based approaches have provided important methods for encapsulating computations. λ-calculus is an example of encapsulation by means of procedural abstraction. Carl Hewitt's

*This work was initiated while the first author was a visitor at the University of Illinois at Urbana-Champaign during May-July 1991. It has been made possible by support provided by a Young Investigator Award from the Office of Naval Research (ONR contract number N00014-90-J-1899), by an Incentives for Excellence Award from the Digital Equipment Corporation Faculty Program, and by joint support from the Defense Advanced Research Projects Agency and the National Science Foundation (NSF CCR 90-07195). The authors acknowledge helpful comments and criticism from Chris Houck, Shingo Fukui, and Daniel Sturman.

work on actors was perhaps the first model to demonstrate the power of the *concurrent object-based paradigm* [Hew77]. The basic rationale underlying the actor approach is to view the behavior of objects as functions of incoming communications [Agh86]. Actors are self-contained, interactive, independent components of a computing system that communicate by asynchronous message passing. The basic actor primitives are:

create: creating an actor from a behavior description and a set of variable bindings (possibly including names of existing actors);

send: sending a message to an actor;

become: an actor changing its local state.

Note that messages may contain *names* (mail addresses) of other actors. Thus, actor systems may be dynamically configured: new actors may be created and connections among actors can be made and broken as computation proceeds. Note that in the Actor model, messages are guaranteed to be eventually delivered. In the semantics discussed in this paper, we ignore this fairness requirement. However, one approach to handling fairness is through a second transition relation (for example, Costa and Sterling [CS84] define such a relation for CCS).

The concept of actors was originally proposed by Hewitt [Hew77]. The actor model has been formally characterized by means of power domain semantics [Cli81], by a transition system [Agh86], and by Colored Petri Nets [SVN91]. Complexity measures for actor programs have been defined [BVN91]. The model has been proposed as a basis for multiparadigm programming [Agh89] and has been used as a programming model for multicomputers [AS88].

Recently, Milner and his colleagues [MPW91] have proposed a new formalism, called the π-calculus, which captures dynamic configurability. An interesting difference between the π-calculus and Milner's earlier Calculus of Communicating Systems (CCS) is that the π-calculus makes no distinction between port names and data values: the purpose of communication in the π-calculus is to transmit (or receive) a name which will admit further communications. The surprising fact is that such a simple generalization is powerful enough to encode computations over arbitrary data types.[1] There has also been considerable progress towards formalizing an object paradigm as a basis of computational encapsulation (for example, see [Wal90, BB90]).

One of the main motivations for models of concurrency (such as [MPW91, Bou89]) is to develop a formalism for concurrent communicating processes which would be the concurrent analogue of the λ-calculus. In particular, the λ-calculus would be a sub-calculus of the concurrent formalism. However, it is not clear whether the π-calculus, for example, can do all the reductions of the λ-calculus in the sense of [BB90].

In functional programming, reduction sequences are ordered by currying. On the other hand, CCS and π-calculus generalize the λ-calculus by binding abstractions to ports. Furthermore, both models are commutative and associative – a property which follows from the the nondeterministic nature of composition in models of concurrency. Note that commutativity and associativity implies objects may be specified in any order. Because there is no name generation facility in the π-calculus, modeling recursion requires the use of relabeling and restriction to address problems of scope extrusion and scope intrusion. On the other hand, the distributed name generation scheme of the Actor model avoids problems of scope intrusion and extrusion. In particular, we do not need restriction and relabeling operators to model arbitrary recursion.

[1] Explicit creation of processes has been included in Milner's recent proposal [Mil90].

The rest of the paper is organized as follows: Section 2 describes an Actor formalism and gives its SOS semantics. Section 3 introduces the hibernate operator. Section 4 gives some examples. Section 5 provides a comparison with other formalisms and discusses some open problems.

2 ACTORS

In this section, we formally describe a language framework for actors. In order to simplify the presentation, we have made a number of changes in the model. In particular, we model an individual actor as a sequential object; concurrency is then limited to that between objects. However, there is no recursion inside an actor; thus it is possible to statically transform the code into a concurrent, atomic set of actions triggered by different incoming messages. The language we describe may be considered a lower level representation of actors similar to that in Cantor [AS88].

The following terminology will be used in the formal definition of the language. *Object-name* is a value denoting the identity of actors and messages. Note that in [Agh86] the term *mail address* is used for the name of an actor and *tag* for the name of message. We assume a countably infinite set of names exist. *Object-var* is a variable name which is bound to elements of *Object-name*. *Variables* are identifiers whose values range over object names, integers or booleans. We assume that these sets are pairwise disjoint. *Bexp* are propositional boolean expressions. *Aexp* are arithmetic expressions. In particular, we assume primitive functions such as comparisons, addition, multiplication and integer division. We also assume that well-formed arithmetic and boolean expressions may be evaluated. The tables below summarize the notation and syntax of our kernel actor language.

2.1 Operational Semantics

We define the semantics of our Actor language in the style of Plotkin's structured operational semantics (SOS). The state of an actor program is captured by a configuration of the form $\langle\langle \aleph, \sigma \rangle\rangle$, where $\aleph$ is a finite set of objects representing the program at various stages of execution and σ is a state function which maps variables (in their context) to values. Note that there are no shared variables between actors; thus, a state function is distributed between the objects in a system and we will consider the projections of the state function onto individual actors. Possible transformations of actor systems are represented by transitions on configurations. An actor program is a finite collection of objects where an object is either an actor or a message.

Actors: An actor may repeatedly receive a messages and, in response, takes a sequence of primitive actions. We model these two steps, or *micro-transitions*, explicitly even though an actor cannot be interrupted between them.[2] The *behavior* of an actor is a finite sequence of actions (instructions) together with a local environment. An actor is denoted by $\langle t, \mu(\xi) : B[i] \rangle$ where t is the actor's name, ξ is the environment providing bindings for its local state and acquaintance variables, B is a sequence of actions, and i represents an instruction counter. Note that an actor is ready to accept a message when its instruction counter is `0`; all explicitly specified actions are defined after the instruction counter is at `1`.

Messages: A message is sent to a specific target actor. A message is denoted by $\langle t, \overline{\mu}(d)\xi \rangle$ where $\overline{\mu}(d)$ denotes that the message is targeted to an actor whose name is bound to d. The content of the message is given by ξ where ξ is an environment mapping variables to values.

[2] In particular, this separates sending a message from its receipt: it has been argued that such a distinction is useful in formalizing broadcast [SNP87].

Symbol	**Type of Element**
$t, t', t_1, \ldots$	Object (actor or message) names
$\mathcal{A}$	Set of valid actor names
$self,\ d, d_1 \ldots$	Variables for actor names
$x, x_1, \ldots$	Local variables
$\xi, \xi_1, \ldots$	Local Environments
$B, B_1, \ldots$	Behaviors
$\langle t, \mu(\xi) : B[i] \rangle$	Actors with internal instruction counter at i
$\langle t, \overline{\mu}(d)\, \xi \rangle$	Messages
$\aleph, \aleph', \ldots$	Set of objects
$a, b, a_1, \ldots$	Hibernating partition tags
$\{\![\aleph]\!\}, \aleph^{a_1, \ldots, a_n}, \ldots$	Hibernating partition (set of objects)
$\sigma, \sigma', \ldots$	State
$\langle\!\langle \aleph, \sigma \rangle\!\rangle, \langle\!\langle \aleph', \sigma' \rangle\!\rangle, \ldots$	Configurations
μ	Actor designator
$\overline{\mu}$	Message designator
$\rightarrow$	Message receipt transition
$\Rightarrow$	Primitive action transition

Figure 1: Notation used in this paper.

A *configuration*, $\langle\!\langle aleph, \sigma \rangle\!\rangle$ consists of a finite set of objects (actors and unprocessed messages) and the current state function (representing the collection of all local environments):

$$\aleph = (\bigcup_{i=1}^{n} \{\langle t_i, \mu(\xi_i) : B_i[k] \rangle\}) \ \cup \ (\bigcup_{i=1}^{n} \{\langle t_i, \overline{\mu}(t_i')\xi_i \})$$

When appropriate we will consider $\aleph$ as a map from a set of object names to a set of closures, where a closure maps environments to a finite sequence of actions. Recall that we assume the existence of an evaluation function which maps expressions in a given state to their values. The evaluation function will be denoted by $[\![\ ]\!] \ : \ \mathrm{Exp} \times \sigma \ \rightarrow \ \mathcal{V}$, where $\mathcal{V}$ is the domain of the primitive values (integers, booleans, actor names, etc.).

A local environment maps variables to values; upon acceptance of a message, a local environment is modified by composing it with the environment specified in the incoming message.[3] Let ξ_1 and ξ_2 be two environments and let $Dom(\xi)$ be the set of parameters defined within environment ξ. Then the "join" or composition of ξ_1 and ξ_2, written $\bullet(\xi_1, \xi_2)$, can be expressed as follows:

$$\bullet(\xi_1,\, \xi_2)(x) = \begin{cases} \xi_2.x & \text{if } x \in \ Dom(\xi_2) \\ \xi_1.x & \text{otherwise} \end{cases}$$

[3] In fact, to maintain actor encapsulation, we will allow only 'undefined' variables to be specified by an incoming message.

$\langle Program \rangle$ ::= $\langle Object \rangle$ | $\langle Object \rangle$, $\langle Program \rangle$
$\langle Object \rangle$ ::= $\langle Actor \rangle$ | $\langle Message \rangle$
$\langle Actor \rangle$ ::= $\underline{\langle}$ *Object-Name*$(\langle Environment \rangle)$: $\langle Behavior \rangle[\langle Non\text{-}negative\ Integer \rangle]\underline{\rangle}$
$\langle Environment \rangle$ ::= [$\langle Variables \rangle$:= $\langle Values \rangle$, ..., $\langle Variables \rangle$:= $\langle Values \rangle$]
$\langle Behavior \rangle$ ::= $\langle Action \rangle$ | $\langle Action \rangle$, $\langle Behavior \rangle$
$\langle Action \rangle$ ::= `unlock` ($\langle List\text{-}of\text{-}Variables \rangle$) |
`assign` ($\langle Variable \rangle$, $\langle Expression \rangle$) |
`test-and-jump` ($\langle Bexp \rangle$,$\langle Positive\ Integer \rangle$)) |
`create` ($\langle Object\text{-}Var \rangle$,$\langle Behavior \rangle$,$\langle Environment \rangle$) |
`send` ($\langle Object\text{-}Var \rangle$, $\langle Environment \rangle$)
$\langle List\text{-}of\text{-}Variables \rangle$ ::= ϵ | $\langle$ *Variable* $\rangle$ | $\langle$ *Variable* $\rangle$, $\langle List\text{-}of\text{-}Variables \rangle$
$\langle Message \rangle$ ::= $\langle$ *Object-Name* (*Object-Name*) $\langle Environment \rangle \rangle$

Figure 2: Syntax of Actor Programs

where x is a state variable.

Environments are defined as records containing bindings of variables, denoted $[x_1 := v_1, \ldots x_n := v_n]$. We assume a primitive constructor which yields environments. Furthermore, we define environment expressions be of the form $e \equiv [x_1 := e_1, \ldots, x_n := e_n]$, and $[\![e]\!] \equiv [x_1 := [\![e_1]\!], \ldots x_n := [\![e_n]\!]]$. As a notational convenience, we will use $\xi[x := v]$, to denote the composition $\bullet(\xi, [x := v])$. The projection of σ onto an actor t is $\sigma \mid_t = \xi$ if $\langle t, \mu(\xi) : B[i] \rangle \in \aleph$ and undefined otherwise.

2.2 Distributed Name Generation

In Actor systems, unique names (mail addresses and tags) are used to distinguish between different objects [Agh86]. There are various ways of representing names; one such representation is an extensible numeric vector. For instance, if ω is a name for some actor t, then $\omega.n$, where n is some nonnegative integer, can be the name of an actor created by t. Thus, if we start with a set of unique names associated with actors in a configuration, then the names generated by the various actors remain unique provided that we make unique extensions to the names.

2.3 Transition Rules

Micro-transitions corresponding to receiving a message will be called *compositions* (denoted $\rightarrow$), and micro-transitions corresponding to taking primitive actions will be called *primitive behavior* transitions (denoted $\Rightarrow$). The composite transitions involving the receipt of a message by an actor and the actions it takes in response, denoted $\rightarrow\Rightarrow$, is atomic. Note that the sequence of actions of an actor takes in response to a given message is always deterministic; all nondeterminism is the result of the arrival order of messages.

The transition relations are defined on individual actors. We use the convention that $\aleph'$ and σ' denote the possible changes in a set of objects $\aleph$ caused by transitions on objects other than the objects to which the transition rule is being applied. Although we define an interleaving semantics, this allows the possibility of parallel transitions.

2.3.1 Primitive Behavior Transitions

We describe the micro-transitions corresponding to actions. Assume that the actor has already accepted a message (and appropriately extended its environment). There are five kinds of primitive actions: assignment, test-and-jump, create, message send, and unlock. *Assignment* modifies an actor's local environment. *Test-and-Jump* provides (forward) branching. *Create* allows creation of new actors; recall that we assume the existence of a local name generation scheme which provides new unique names. *Send* provides for point-to-point, asynchronous communication, i.e., the behavior of a sender is not dependent on the receipt of a message by its target. *Unlock* resets the instruction counter to be 0 and sets some local variables to be 'undefined' ($\perp$); only undefined variables may be specified by a subsequent incoming message. Other local variables will retain their bindings. Thus, the integrity of local data encapsulated in an actor is guaranteed.

Transition Rule 1 (Assignment) *Let* $B[i] = \texttt{assign}(x, v)$ *be the current action given by the behavior of an actor* t *where* $i > 0$, *then*

$$\langle\!\langle \aleph \cup \{\langle t, \mu(\xi) : B[i]\rangle\}, \sigma\rangle\!\rangle \;\Rightarrow \langle\!\langle \aleph' \cup \{\langle t, \mu(\xi[x := [\![v]\!]]) : B[i+1]\rangle\}, \sigma'\rangle\!\rangle$$

where $\sigma' \mid_t \equiv \xi[x := v]$, *provided that if* $[\![v]\!] \in \mathcal{A}$ *then* $[\![v]\!] \in Dom(\xi)$.

Transition Rule 2 (Test-and-Jump) *Let* $B[i] = \texttt{test-and-jump}(e, j)$ *be the current action given by the behavior of an actor* t *for some* $i, j > 0$,

$$\langle\!\langle \aleph \cup \{\langle t, \mu(\xi) : B[i]\rangle\}, \sigma\rangle\!\rangle \Rightarrow \begin{cases} \langle\!\langle \aleph' \cup \{\langle t, \mu(\exp) : B[i+1]\rangle\}, \sigma'\rangle\!\rangle & \textit{if } [\![e]\!] = \texttt{true} \\ \langle\!\langle \aleph' \cup \{\langle t, \mu(\exp) : B[i+1+j]\rangle\}, \sigma'\rangle\!\rangle & \textit{otherwise} \end{cases}$$

Transition Rule 3 (Message Send) *Let* $B[i] = \texttt{send}(x, e)$ *for some* $i > 0$, *be the current action given by the behavior of an actor* t, *and let* $\xi(x) \in \mathcal{A}$ *then*

$$\langle\!\langle \aleph \cup \{\langle t, \mu(\xi) : B[i]\rangle\}, \sigma\rangle\!\rangle \;\Rightarrow \langle\!\langle \aleph' \cup \{\langle t, \mu(\xi) : B[i+1]\rangle\} \cup \{\langle t', \overline{\mu}([\![x]\!])[\![e]\!]\rangle\}, \sigma'\rangle\!\rangle$$

where t' *is a unique new name, provided that if* $e = [x_1 := e_1, \ldots, x := e_n]$, *and for some* i, $[\![e_i]\!] \in \mathcal{A}$ *then* $[\![e_i]\!] \in Dom(\xi)$.

Transition Rule 4 (Create) *Let* $B[i] = \texttt{create}(x, B', e)$, *for some* $i > 0$ *be the current action given by the behavior of an actor* t, *and let* $[\![e]\!] = \xi'$, *where* ξ' *is an environment then,*

$$\langle\!\langle \aleph \cup \{\langle t, \mu(\xi) : B[i]\rangle\}, \sigma\rangle\!\rangle \Rightarrow \langle\!\langle \aleph' \cup \{\langle t', \mu(\xi') : B'[0]\rangle\} \cup \{\langle t, \mu(\xi[x := t']) : B[i+1]\rangle\}, \sigma'\rangle\!\rangle$$

where t' *is a unique new name,* $\sigma' \mid_t = \xi[x := t']$ *and* $\sigma' \mid_{t'} = \xi'$.

Transition Rule 5 (Unlock) *Let* $B[i] = \texttt{unlock}(x_1, ..., x_n)$ *be the current action given by the behavior of an actor* t *for* $i > 0$ *then,*

$$\langle\!\langle \aleph \cup \{\langle t, \mu(\xi) : B[i]\rangle\}, \sigma\rangle\!\rangle \Rightarrow \langle\!\langle \aleph' \cup \{\langle t, \mu(\xi') : B[0]\rangle\}, \sigma'\rangle\!\rangle$$

where $\forall i((0 < i \leq n)\, \xi'(x_i) = \perp)$ *and the corresponding environment is recorded in* $\sigma' \mid_t$.

2.3.2 Composition Transitions

An actor takes actions only when it accepts a message; i.e., it remains unchanged until it receives a message targeted to it. Otherwise, it extends its local environment using the incoming message and proceeds to execute the sequence of actions specified by its behavior. In either case, the message is consumed. The first two rules below express these facts. The rest of the rules simply specify that other objects do not interact. Note that there is no explicit parallel actions in our formalism. The potential for parallel actions, i.e. concurrency, is implicitly represented by the possible interleavings of transitions on different objects.

Transition Rule 6 (Composition) *Let an actor named t receive a message then it extends its environment by the incoming message and is ready to take the next action specified by its behavior.*

$$\langle\!\langle \aleph \cup \{\langle t, \mu(\xi) : B[0]\rangle, \cup\langle t', \overline{\mu}(t)\xi'\rangle\}, \sigma\rangle\!\rangle \rightarrow \langle\!\langle \aleph' \cup \{\langle t, \mu(\bullet(\xi,\xi')) : B[1]\rangle\}, \sigma'\rangle\!\rangle$$

where $\sigma' \mid_t = \bullet(\xi, \xi')$ *provided that* $\forall x \in Dom(\xi')\, \xi(x) = \perp$.

2.3.3 Meaning of a Program

By definition, a program consists of a finite set of actors and messages. The meaning of a program is the set of all sequences of configurations that are reachable from the initial configuration. Thus, if C_0 is the initial configuration defined by a program, then the meaning of the corresponding program is a collection of sequences of configurations of the form $\langle C_0, C_1, C_2, \cdots, \rangle$ where $C_i \rightarrow\!\Rightarrow C_{i+1}$. A natural question to ask is whether the meaning of an actor program depend on the order of specification or grouping of the actors and messages in it? Equivalently, is the specification associative and commutative? The answer is trivially yes. Commutativity and associativity follow from the fact that composition is never forced between any two objects in a configuration.

3 A HIERARCHICAL ABSTRACTION

The Actor formalism defined in the previous sections does not provide explicit support for hierarchical decomposition of systems. However, different constructs may be used to simulate different aspects of structuring. For example, anonymous resources may be implemented through the concept of *receptionists* [Agh86]. Such high level structures have to be explicitly represented in terms of the primitives. As a consequence, reasoning about the structures requires reasoning in terms of the low-level transition relations defined. In this section, we define a new operator, called *hibernate*, in terms of transitions on actors. Hibernate provides hierarchical structuring abilities. In particular, hibernate can model computer networks and operating systems which may refuse to accept requests after acquiring a certain number or types of requests. Another way to characterize hibernation is as a mechanism for specifying a *synchronization constraint* which states that the sender of a message must belong to some specific (dynamic) set. Two other methods for enforcing synchronization constraints in actors are *insensitive* actors [Agh86], which require creation of internal buffers, and *enabled sets* which uses reflection over mail queues [TS89].

An individual actor may react to a message by adopting a hibernate behavior. A group of actors may collectively hibernate; such behavior is initiated by a single leader. The leader must then initiate or define the 'work' for the hibernating group. Note that the our semantics of hibernation does not provide any security (any actor may cause its acquaintances to hibernate). A security mechanism would require providing a capability mechanism beyond mail addresses. The work for the group is defined in terms of the first messages after hibernation sent by the leader. Because we name messages, it is possible to extend a hibernating group to include some messages which have already been sent. In order to do so, we simply need to modify our send transition to allow the names of messages sent to be bound (and thus known). For brevity, we will not do so here.

Intuitively, hibernate can be thought of as enclosing some actors, and perhaps some specified pending messages to them, in a membrane with a one-way valve which lets out messages not targeted to actors within the membrane. However, the membrane does not allow any messages or actors which exist outside the membrane, or are created by an actor outside the membrane, to enter the membrane. Furthermore, the membrane can 'expand' internally as new actors and messages are created within the enclosure. Membranes may be nested; if two membranes have a non-zero intersection, one must contain the other. A hibernating actor may autonomously remove itself from the membrane; in this case, all messages targeted to the actor may be removed from the membrane. In this case, an actor must remove itself from the innermost membrane first. A set of objects in a configuration, $\aleph$, may be (recursively) partitioned into a finite number of (nonempty) subsets representing hibernating membranes. A unique name is assigned to each membrane. Membrane (group) names are syntactically distinct from object names and will be used to extend object names.

3.1 Transition Rules for Hibernation

An actor may specify a hibernation action on a group of objects which are in its current hibernation partition. We define hibernation only on a group of actors but the extension to a group of actors and (unprocessed) messages is obvious. Note that all actors in a newly hibernating set, except the actor specifying the hibernation action, are free to accept a message at the time they are freshly tagged. Thus all (unprocessed) messages which have been previously sent to such an actor, will not be delivered unless and until the actor leaves the hibernating partition. Actors may remove themselves from a hibernating partition, or the partition may be automatically removed, if no further activity is possible within the partition.

Transition Rule 7 (Hibernate) *Let* $B[k] = \texttt{hibernate}(t_1 \ldots t_n)$*, for some* $k > 0$ *and* $n \geq 0$*, be the current action given by the behavior of an actor named* $t_0.\overline{a}$ *where* $\overline{a}$ *is a possibly empty sequence of hibernating group names, and let* b *be the new unique partition tag and* $t_1 \ldots t_n \in Dom(\aleph)$ *and* $t_1 \ldots t_n \in \mathcal{A}$ *and* $\forall i((0 \leq i \leq n)\exists x \xi(x) = t_i.\overline{a})$*, then*

$$\langle\!\langle \aleph \cup \bigcup_{i=1}^{n} \{\langle t_i.\overline{a}, \mu(\xi_i) : B[0]\rangle\} \cup \{\langle t_0.\overline{a}, \mu(\xi_0) : B[k]\rangle\}, \sigma\rangle\!\rangle \;\Rightarrow$$
$$\langle\!\langle \aleph' \cup \bigcup_{i=1}^{n} \{\langle t_i.\overline{a}.b, \mu(\xi_i') : B[0]\rangle\} \cup \{\langle t.\overline{a}.b, \mu(\xi') : B[k+1]\rangle\}, \sigma'\rangle\!\rangle$$

where $\xi_i'(x) = t_i.\overline{a}.b$ *if* $\xi(x) = t_i.\overline{a}$*, otherwise* $\xi_i'(x) = \xi(x)$*.*

Transition Rule 8 (Unhibernate) *Let* $B[i] = \texttt{unhibernate}$*, for some* $i > 0$*, be the current action given by the behavior of an actor* t *which has a hibernating partition tag* $\overline{a}.b$*, where* $\overline{a}$ *is a possibly empty sequence of hibernation tags. Then the actor* t *coming out of hibernation, removes its innermost tag:*

$$\langle\!\langle \aleph \cup \{\langle t.\overline{a}.b, \mu(\xi) : B[i]\rangle\}, \sigma\rangle\!\rangle \Rightarrow \langle\!\langle \aleph' \{\langle t.\overline{a}, \mu(\xi) : B[i+1]\rangle\}, \sigma'\rangle\!\rangle$$

An actor α, currently hibernating in a partition, $\aleph^{\overline{a}}$ may create another hibernating partition, $\aleph^{\overline{a}.b}$, where all objects in the new hibernating $\aleph^{\overline{a}.b}$ are objects in the original partition. A hibernating partition, $\aleph^{\overline{a}}$, is innermost if it has no hibernating subpartitions, i.e., none of its objects have a tag $\overline{a}.\overline{b}$ where $\overline{b}$ is a nonempty sequence of hibernation tags. When an innermost hibernating partition is inactive – i.e., when there are no pending messages in it and all its actors are free (their instruction counter is at 0), the partition may be removed. Because actors in inner partitions may send messages out, outer partitions are not removed until the inner ones have been removed.

Transition Rule 9 (Inactive Partition) *Let $\aleph^{\overline{a}.b}$ be an inactive hibernating partition, i.e., all its objects are unlocked actors which have tags $\overline{a}.b$, then the group of actors in the partition comes out of hibernation atomically:*

$$\frac{\forall \alpha \in \aleph^{\overline{a}.b}\, \exists t, \xi, B(\alpha = \langle t.\overline{a}.b,\, \mu(\xi) : B[0]\rangle)}{\langle\!\langle \aleph \cup \aleph^{\overline{a}.b}, \sigma \rangle\!\rangle \rightarrow\!\!\Rightarrow \langle\!\langle \aleph' \cup \aleph^{\overline{a}}, \sigma' \rangle\!\rangle}$$

3.2 Primitive Behaviors Under Hibernate

The rules for primitive actions of a hibernating actor are specified as follows. The rules for `assignment` and `unlock` remain the same except that the actors remain hibernating. The `send` rule remains the same except that the message is tagged with the current partition, much like the create rule below. Thus the message may be processed by an actor within the same hibernating partition by using the same composition rule as before. In order to allow the message to be delivered to an actor outside the current membrane, we need an additional rule which renames the message to the actor's current partition, provided that the actor's current partition is contains the partition where the message originates. In particular, this ensures that all actors which are not in any hibernating partition will receive every message sent to them.

Transition Rule 10 (Message ReMark) *A message destined for an actor which is above the message's hibernating partition is delivered.*

$$\langle\!\langle \aleph \cup \{\langle t.\overline{a}.\overline{b}, \overline{\mu}(t'.\overline{a}.\overline{b})\xi\rangle \cup \langle t'.\overline{a}, \mu(\xi') : B[k]\rangle\}, \sigma\rangle\!\rangle \Rightarrow \langle\!\langle \aleph' \cup \{\langle t.\overline{a}, \overline{\mu}(t'.\overline{a})\xi\rangle \cup \langle t'.\overline{a}, \mu(\xi') : B[k+j]\rangle\}, \sigma'\rangle\!\rangle$$

where $\overline{a}$ is be a possibly empty sequence of hibernation tags.

Transition Rule 11 (Create) *Let $B[i] =$ `create`(x, B', e), for some i, be the current action given by the behavior of an actor t, if $[\![e]\!] = \xi'$, where ξ' is an environment then,*

$$\langle\!\langle \aleph \cup \{\langle t.\overline{a}.b, \mu(\xi) : B[i]\rangle\}, \sigma\rangle\!\rangle \Rightarrow \langle\!\langle \aleph' \cup \{\langle t'.\overline{a}.b, \mu(\xi') : B'[0]\rangle\} \cup \{\langle t.\overline{a}.b, \mu(\xi[x := t'.\overline{a}.b]) : B[i+1]\rangle\}, \sigma'\rangle\!\rangle$$

where t' is a new unique name, $\sigma' \mid_t = \xi[x := t'.\overline{a}.b]$ and $\sigma' \mid_{t'.\overline{a}.b} = \xi'$.

4 EXAMPLES

In this section, we describe four examples: a higher order function which applies a function to all its arguments (MapCar), two embeddings of the call-by-name λ-calculus (respectively, lazy and with futures) and an example of multi-party coordination between robots. In our λ-calculus examples, we create new actors (whose names are unknown to the rest of the universe) and these actors concurrently carry out sub-computation corresponding to given reductions. The results are then sent to a *join-continuation*. Join-continuations are shared continuations often used in actor systems – they collect results from multiple computations in order compute to compute the final result and forward it to their caller (see [Agh90a, Agh90b]. In these examples, locality of namespace provides the necessary separation of different subcomputations. On the other hand, hibernation is used in cases where we are representing real resources by means of actors. Because resources may not generally be created on the fly, it is not meaningful to simply create new computational processes to maintain non-interference of computations. The example of cooperating robots uses hibernation.

The description of examples in this section has been simplified by using some syntactic sugar. A list of parameters corresponding to a method may be partially filled by a message and the actions specified in a behavior are not executed until all of the state variables have arrived. This allows us to implicitly specify join-continuations. The state parameters appear after the name and before the colon and the message parameters appear in square brackets before the double colon. Message parameters must be instantiated anew each time a method is invoked; on the other hand, state parameters need not be rebound. In these examples, whenever an actor uses partial instantiation in a message, we send the variable names along with their new bindings to enable us to clearly indicate the positions of the parameters. Often state parameters are initialized during creation; these initial expressions appear on the same line as the create. We use a form of conditionals `if` e_0 A_1 A_2, where A_2 is optional, to mean the usual branch. The form `do` is used to group a sequence of commands to be treated as a single block for conditionals. Where appropriate, we use methods as in object-based style. In this case, the actions are triggered by the arrival of the relevant subset of the parameters. Note that both the conditionals and methods can be easily transformed into test-and-jump commands. Finally, we assume that simple pattern matching, represented by $\equiv$ in the code, is available to extract parameters.

MapCar

MapCar is a command which takes a list and a function and returns a list whose contents is the function applied to each element. Example:

$$((10\ 12\ 2)\ x^2) \rightarrow\ (100\ 144\ 4)$$

Formally Calling Actors To call the MapCar function, an outside actor would actually issue the statements below for the answer to be returned to an actor whose name is bound to `customer`.

```
(let ((x (create TopLevel (customer))))
        (send-to x (list, f)))
```

Each actor named is just short hand for the inline expansion of the whole behavior statement.

```
TopLevel (Cust): [FunArgs, Function]::
          (let ((x2  (create FunApply(Nil))))
              (let ((x1 (create KeepList (x2, Cust))))
                  (send-to x1 (list: (cdr FunArgs), f: Function, u: Nil))
                  (send-to x2 (elem: (car FunArgs), f: Function))
                  (send-to x2 (Acq: x1))
                  (unlock (Funargs, Function))))
```

The definitions for *KeepList* and *FunApply* appear below. *KeepList* was created with acquaintances; the acquaintance for *FunApply* was not available upon creation therefore x_1 was sent later in a message. The state parameters appear after the name of the actor, the message parameters appear in the square brackets. Note however that state parameters may be instantiated by a message or during creation. In the case where an actor has no state parameters the field will be denoted by an empty list ().

```
KeepList (Acq, Cust): [list, f, u, fa]::
       (if (= Nil list)
           (do (send-to Cust 'u)
               (unlock (list,f,u,fa)))
           (do (assign (list, (cdr list)))
               (assign (u, (append fa u)))
               (send-to Acq ((car list), f))
               (unlock (fa))))

FunApply (Acq): [elem, f]::
       (if (= Nil elem)
           (send-to  Acq (fa: Nil))
           (send-to  Acq (fa: (f elem))))
       (unlock (elem,f)))
```

λ-Calculus

In this section, we discuss the encoding of *lambda-calculus* in our framework. We show how a term can be encoded into a set of actors and the interaction among the objects, denoting the terms, achieves the intended reductions of the λ-calculus. A number of manipulations to keep track of names of actors generated to represent new computational processes become necessary. Observe that this is also true in the encoding of *lazy reduction* in Mobile processes. However, these objects have no direct correspondence to objects in underlying calculus.

Recall the syntax of λ-calculus. If $M,\ N \in \Lambda$, then the syntax of the terms is given by:

$$M ::= x \mid (\lambda x.M) \mid (MN)$$

where x stands for any variable. Let x not occur free in M. The rules:

$$\begin{array}{lcl} \alpha\text{-}rule & : & (\lambda x.M) \rightarrow (\lambda y.M[x/y]) \\ \beta\text{-}rule & : & (\lambda x.M_1)M_2 \rightarrow (M_1)[x/M_2] \\ \eta\text{-}rule & : & (\lambda x.Mx) \rightarrow M \\ \xi\text{-}rule & : & (M = N) \Rightarrow \lambda x.M = \lambda x.N \end{array}$$

Let $\lambda\vec{x}.M \equiv \lambda x_1 \cdots x_n.M \equiv \lambda x_1(\lambda x_2(\cdots(\lambda x_n(M))..))$. We can now model *lazy reduction* through the interaction of the objects. The *lazy reduction* relation [Abr88], $\rightarrow$ over Λ is the smallest relation such that:

$$(\lambda x.M)N \rightarrow M[x/N]$$

$$\textbf{if } M \rightarrow M' \textbf{ then } MN \rightarrow M'N$$

Note that the reduction strategy is completely deterministic. Any term M can be written as $M_0M_1\cdots M_m$ for some $(m \geq 0)$ where M_0 is not an application. Then, $M \rightarrow M'$ for some $M' \iff m \geq 1$ and M_0 is λxN, and then M' is $N[x/M_1]M_2\cdots M_m$. So for each M there is at most one irreducible term M' such that $M \rightarrow^* M'$, and further, M' is either of the form $\lambda x.N$ or of the form $xN_1, \cdots, N_n$ $(n \geq 0)$.

We encode the λ-terms into our formalism by providing an actor Θ which maps λ-terms to a set of actors and messages (i.e., creates the set of actors and messages). To avoid unnecessary technical details, we only provide the definitions of a set of actors rather than the actual execution. Let u be the customer address and expect the final answer in variable *ans*; also u is the only element of the initial list. Let the function `append(id, list)` be interpreted as *id.list* where "." denotes concatenation. Let $[\![M]\!]$ denote the expression evaluated by the corresponding method, and the function *quote* M yields the unevaluated expression. The *parse* routine parses an expression and returns a value which will be equal to a literal constant `Var-expr`, `Abs-expr`, or `Apply-expr`.

Lazy Lambda Calculus Program

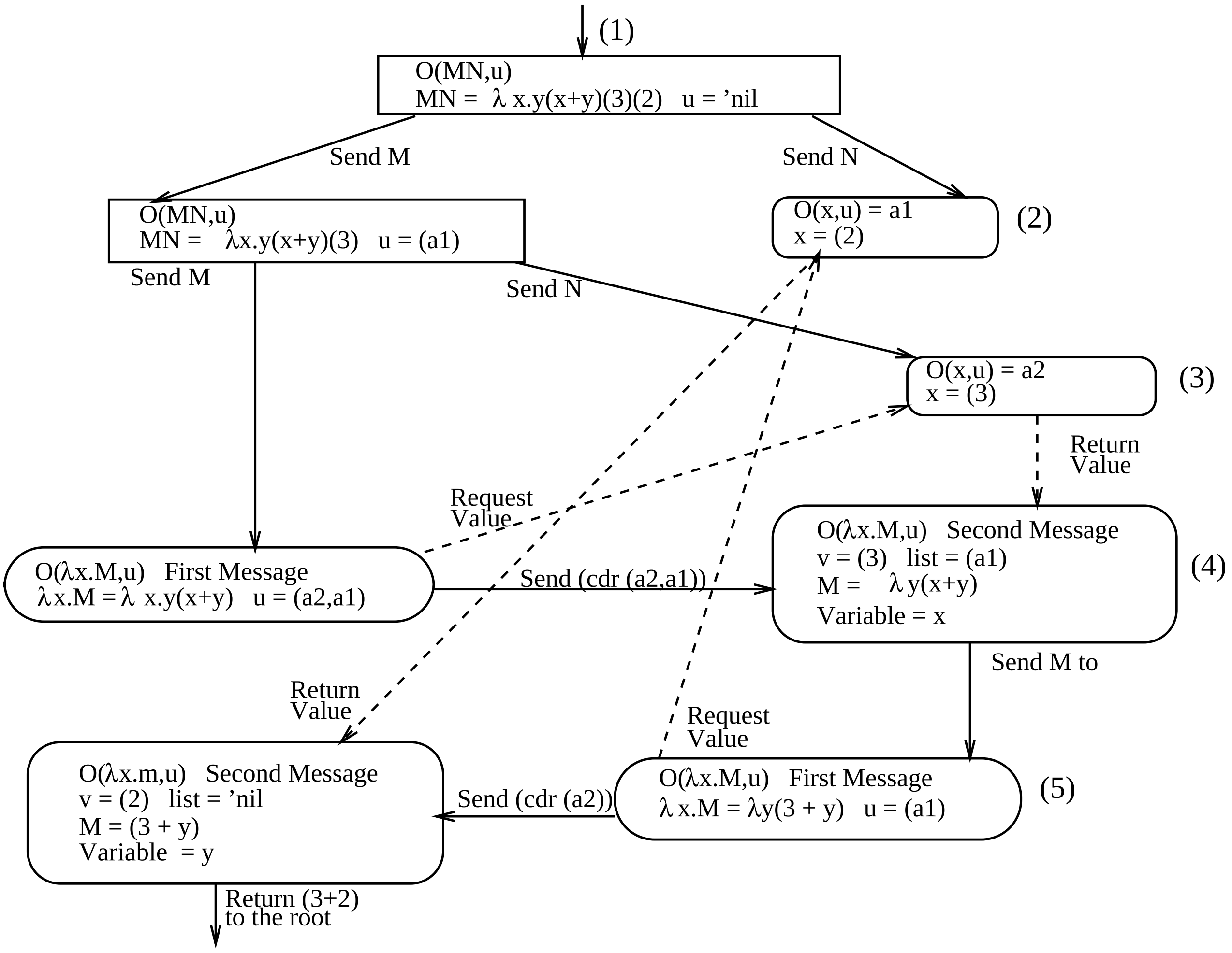
```
Θ (): [expr,list]::
          (if (= (parse expr) Var-expr)
               (send-to (car list) (ans: ⟦ expr ⟧))
               (if (= (parse expr) Abs-expr)
                   (let ((d (create Abstraction(self,Nil,Nil,Nil))))
                        (send-to d  ('lambda-abs (expr, list))))
                   (let ((d (create Application (self))))
                        (send-to d (expr, list)))))
          (unlock (expr,list))

Variable-cont (): [ans,name]::
          ((send-to name ('return-var (⟦ ans ⟧)))
           (unlock (ans,name)))

Application (θ-var): [≡ MN,list]::
                     (let ((d (create Variable-cont)))
                           (send-to θ-var (N, (cons d Nil)))
                           (send-to θ-var (M, (append d list)))
                           (unlock (≡MN,list)))

Abstraction (θ-var,M,x,saved-list):
    (methods
        ['lambda-abs (≡λx.M,list)]::
             ((send-to (car list) (name: self))
              (assign (M, M))
              (assign (x, x))
              (assign (saved-list, (cdr list)))
              (unlock (≡λx.M,list)))
        ['return-var (var)]::
             ((send-to θ-var (M[x/var], saved-list))
              (unlock (var)))
```

Note: `M[x/v]` denotes the variable `v` substituted in for `x` in the expression `M`.

There are other ways, in which we can encode and show that the interaction of the agents correspond to the lazy reduction. In fact, we can also encode starting from the leftmost so that all the other agents would send the rightmost evaluated terms. The correct arrival ordering is essential; therefore we could use the *assign* semantics to perform a total replacement of state parameters and the *hibernate* operator. Instead of describing the mapping, we consider the translation of a typical term $(((\lambda x.\lambda y.M)N)P)$ as follows.

Example of the Agent Thread for Lazy Evaluation

Input Expression: λ x.y(x + y)(3)(2)



1. Send the lambda expression to an apply node. We assume there are at least two terms.
2. Send M to be evaluated and Send N to be evaluated into a constant. M arrives at another apply node λx.y(x+y)(3). The value of N waits at the variable continuation (a1) until its value is requested.
3. Send M to be evaluated and N to evaluated into a constant node (a2). This time M is an abstraction.
4. This abstraction node requests the value for the bound variable, when it arrives (in the 'return-var) the β-reduction is performed. The expression, in this case, is λx.y(x+y). When the 'return-var message arrives, the value for x arrives (3), and the new expression becomes λ.y(3+y).
5. The new M expression matches once again with a lambda abstraction and it subsequently requests the value for y. Once this value arrives in the 'return-var message, the new expression to be evaluated becomes (3 + 2). Thus (5) gets propagated to the (car list) the root.

Actor Representation	Deductive Step
$\{\![\{\![\langle t_1, \mu xy.M\rangle, \langle t_2, \overline{\mu}(t_1)N\rangle]\!\}, \langle t_3, \overline{\mu}(t_1)P\rangle]\!\}$	Original semantic definition
$\{\![\{\![\langle t_1, \mu y.M[N/x]\rangle\ ]\!\}\langle t_3, \overline{\mu}(t_1)P\rangle]\!\}$	Actor, message composition
$\{\![\langle t_1, \mu y.M[N/x]\rangle, \langle t_3, \overline{\mu}(t_1)P\rangle]\!\}$	Rule 9, Out of hibernation
$\{\![\langle t_1, \mu M[N/x][P/y]\rangle]\!\}$	Actor, message composition
$\{\langle t_1, \mu M[N/x][P/y]\rangle\}$	Rule 9, Out of hibernation

From the last item, it can be easily seen that the substitutions are exactly the same that obtained in the λ-calculus through lazy reduction. We have also been able to give an encoding of evaluation with *futures* where the body of an abstraction also gets evaluated without using the *hibernate* operator. The Variable, Application and Abstraction rules for evaluation with futures are:

```
Θ (): [expr,list]::
           (if (= (parse expr) Var-expr)
              (send-to (car list) (ans: [[ expr ]]))
              (if (= (parse expr) Abs-expr)
                  (let ((d (create Abstraction)))
                        (send-to d (expr, list)))
                  (let ((d (create Application) (self)))
                        (send-to d (expr, list)))))
           (unlock (expr,list))

Application (θ-var): [≡ MN,list]::
            (let ((c (create Continuation) (θ-var)))
                  (send-to θ-var (M, (append c list)))
                  (send-to θ-var (N, (append c list)))
                  (unlock (≡ MN,list)))

Continuation (θ-var): [ans,M,x,list]::
            ((send-to θ-var (M[x/ans], list))
             (unlock (ans,M,x,list)))

Abstraction (): [≡λx.M,list]::
            ((send-to (car list)(M: M, x: x, list: (cdr list)))
             (unlock (≡λx.M,list)))
```

Multi-Party Interaction

Consider a system of more than two robots. Assume that an action (such as lifting a heavy object) has to be performed in a cooperative manner by a number of robots; in case the lifting actions of the participating robots do not coincide, the results will not be desirable. In a sense, we have to arrive at some common knowledge. Since without a global clock we can at best arrive at ϵ-common knowledge (see, for example, [Mos86]), we should provide sufficient flexibility so that the critical actions may be reliably performed in synchrony. Note that a correct implementation of this example needs further specification of constraints on the underlying system, for example, constraints on such factors as the time delays in a communication between robots. To address the problem, we split it into two levels:

- First, define a synchronization for multi-party interaction; this can be augmented with time limits if the response has to be provided within time-limits.

- Once a given set of actors has been synchronized for a multi-party interaction, we may change the communication model to support, for example, a synchronous deterministic behavior until the task is executed.

The hibernate operator provides a mechanism for addressing the first of these two problems. Consider a pool of identical robots which can do assorted tasks with cooperation from several of them. Each robot can receive a request for a given task (and thus, becomes the leader for that interaction); it then coordinates a dynamic subset of robots to execute the given task. In the following, we describe the necessary transitions for an actor that behaves as a robot. The set of identical robots is contained in the list *all* and is known to each robot. The `(set-diff A B)` function takes sets `A` and `B` and returns the set difference `A - B`. The function *respond* takes a relative time interval and puts it into a absolute time in the future (i.e. now + interval). The function *satisfy* determines whether timing constraints are satisfied. Finally note, this procedure assumes that task names are unique.

Step 1 *A coordinator is selected from the outside by issuing an* `initiate-task` *message. A request for enrollment within the given* `response-time` *is issued. The set-of-enrolled is initialized to the leader and the number-enrolled is initialized to 1. Task specific parameters which arrived in the message are also saved. The structure of the code for a* `Robot` *is described below with the* `initiate-task` *method filled in.*

```
Robot (all, set-of-enrolled, curr-task,
       required-no, number-enrolled, response-time, ready-time):
   (methods
      ['initiate-task (task-name, num-in-task, res-time, red-time)]::
           (if ( = curr-task Nil)
                (send-to all
                      ('enroll (task-name, self, (respond response-time))))
                ((assign (set-of-enrolled, self))
                 (assign (curr-task, task-name))
                 (assign (required-no, num-in-task))
                 (assign (number-enrolled, 1))
                 (assign (response-time, res-time))
                 (assign (ready-time, red-time))))
           (unlock (task-name, num-in-task, res-time, red-time))
       < other methods>)
```

Step 2 *If the robot is not enrolled in another task then, respond to the request for enrollment positively and become enrolled in the requesting task.*

```
      ['enroll (task-name, leader, time-limit)]::
            (if (and (satisfy time-limit) (= curr-task Nil))
                (send-to leader ('willing-to-enroll (task-name, self)))
                (assign (curr-task, task-name)))
            (unlock (task-name, leader, time-limit))
```

Step 3 *If you are the leader and the number of enrollments has not reached the required number, then you will be able to accept an enrollment message. However, if the number of enrollments has just reached the required number with the current message then, do not accept any messages other than from the enrolled actors. Send the enrolled actors a ready message and for all of the other actors, send a* `not-needed` *message, where the other actors are obtained via* `set-diff`.

```
['willing-to-enroll (task-name, actor-name)]::
    (if (= task-name curr-task)
        (if (< number-enrolled  (- required-no 1))
            ((assign (set-of-enrolled, (append actor-name set-of-enrolled)))
              (assign (number-enrolled, (+ number-enrolled 1))))
            (if (= number-enrolled (- required-no 1))
                   ((assign (set-of-enrolled, (append actor-name set-of-enrolled)))
                    (assign (number-enrolled, (+ number-enrolled 1)))
                    (hibernate (set-of-enrolled))
                    (send-to set-of-enrolled
                               ('ready (task-name, self, (respond ready-time))))
                    (send-to (set-diff all set-of-enrolled)
                                      ('not-needed (task-name))))))
     (unlock (task-name, actor-name))
```

Step 4 *Upon receiving a* `not-needed` *message from the task in which an actor is enrolled, an actor has no current task (i.e. enough participants have enrolled in the task.)*

```
['not-needed (task-name)]::
       (if (= task-name curr-task)
              (assign (curr-task, Nil)))
       (unlock (task-name))
```

Step 5 *When the leader receives the required number of enrollment messages, it sends a* `ready` *message to each member of the* `set-of-enrolled` *in order to signify that the transaction is about to take place. When they receive this message the* `set-of-enrolled` *is hibernating together and executes the given task.*

```
['ready (task-name , leader, deadline)]::
       ((if (satisfy deadline)
            ( execute the coordinated task)
            (send-to leader ('finish (task-name,leader))))
         (unlock (task-name, leader, deadline)))
```

Step 6 *On finishing the task, the leader sends* `finish` *messages to each of the participants and comes out of hibernation. After coming out of hibernation, both the leader and the participants have no current task.*

```
['finish (task-name,leader)]::
     ((if (= leader self)
         (send-to set-of-enrolled ('finish (task-name, leader))))
      (unhibernate)
      (assign (set-of-enrolled, Nil))
      (assign (curr-task, Nil))
      (unlock (task-name, leader)))
```

In the transitions given above, note that *executing the coordinated task* in Step 5 corresponds to saying that here we can assume that the *set-of-enrolled* processes can communicate only within themselves and hence, one can use other assumptions such as synchronous or strong synchronous or lock-step synchrony and derive the real-time properties. Further, in line 5, *deadline* is the result of the expression `(respond ready-time)` denotes some global time so that exact timing information can be arrived (perhaps, knowing the clock drifting, one can arrive at the timing properties).

5 CONCLUSIONS

In this section, we give a brief comparison between the Actor model and two other models of concurrency, namely the Chemical Abstract Machine (CHAM) and the π-calculus. We also describe some open research questions. Berry and Boudol have defined CHAM which they believe is based on a chemical metaphor. The model has been used to simulate other models of concurrency such as π-calculus and γ-calculus. CHAM is specified through a set of laws which can be said to provide an operational semantics.

We describe the actor analogs of these laws. The *reaction law* supports composing an actor and a message in a way similar to our rule of composition 6. The *chemical law* essentially corresponds to the fact that composition of one actor and a message targeted to it does not affect the possibility of composition of another actor and a message targeted to that actor. The purpose of the *membrane law* is to allow an actor to freely perform internal actions as well as send messages to other *ions* floating in the main solution. In a sense, this provides a way of hierarchical structuring solutions (or processes).

To allow global communication, an *airlock law* is introduced which states that a molecule may be taken out from a solution; in this case, the rest of the solution continues to have the internal actions. In our framework, the *membrane law* follows from the hibernate operator. Note that the hibernate operator permits sending messages: in particular, it is possible to send messages to the external world. Because the destination of a message is explicitly specified, there is no need for an explicit analog of the *airlock law* in our framework. In our formalism, one can remove either an actor or a message from a partition without any adverse affect – thus making the structuring symmetrical with respect to an actor and a message. The *hatching law* is accomplished trivially through successive application of the unhibernate operation.

To model external nondeterminism, for example, as defined in CCS, CHAM introduces a pairing operator and a tagging (labeling) mechanism. In our framework, such external nondeterminism is achieved without the need of tagging due to the explicit naming mechanism. As is apparent in our encoding of λ-calculus, there is no explicit prefixing or sequencing in our formalism. Our encoding of the λ-calculus suggests that the framework proposed has the abilities of concurrent λ-calculus [dB91]. For example, instead of specifying communication between a function and an argument, as in the classical λ-calculus, nondeterministic communication between several functions and several arguments can be specified using asynchronous communication. Furthermore, we do not compromise the higher-order characteristics of the λ-calculus.

It is of interest to note Milner ([MPW91], [Mil90]) has encoded the linear λ-calculus using π-calculus without replication and full lazy reduction (which is determinate) and call-by-value (which is nondeterministic) strategies in Mobile processes using replication. In the translation of a call-by-value strategy, due to the prefixing, parallel evaluations can take place for $\lambda x\text{-}M_1 M_2$ only when M_1 and M_2 are free of variable x. However, it is not clear whether the eager evaluation (wherein the body in an abstraction can be evaluated) can in fact be encoded in the π-calculus. It appears that to encode eager evaluation, it is necessary to ensure, first, that the same value is passed to all the subexpressions of the body of the abstraction, and second, that simultaneous substitution in the subexpressions is essential. Whether such an encoding can be done without the use of a mechanism such as *membrane* as in CHAM or an operator such as *hibernate* in our formalism is not clear. As we described earlier, without using the hibernate operator, we can encode the λ-calculus so that the body of an abstraction is concurrently evaluated. However, the relation of this encoding with the one using the hibernate operator is not clear in terms of λ-definability. Milner has encoded the *call-by-value* semantics in [Mil90]. In summary, although the π-calculus has some of the lambda-calculus power, it is not clear whether the power of λ-calculus can be derived in it.

The study brings to light another question: what is an equivalent notion of bisimulation in an asynchronous formalism? In the context of an asynchronous formalism, it is very important to arrive at the notion of equivalence as it would provide methods for specifying and reasoning about concurrent programs. Recently, [dB91] has provided techniques for reasoning about POOL programs based on techniques for reasoning CSP programs; however, the communication model in POOL is synchronous. In practice, we regard two actor actor systems as equivalent with respect to an environment if the ordering between messages received from and sent to the environment is the same.

Note that the semantics we have defined does not provide any abstraction over the details of a computation. A more abstract semantics would make all transitions which do not involve messages to or from the environment, internal or silent transitions. Modeling *open systems* requires explicit representation of *receptionists* and *external actors* [Agh86]. External actors are actors whose name is known but whose behavior and local state is unknown in a configuration. Note that both the set of receptionists and the set of external actors is dynamic. We have to appropriately modify our state function and allow messages from outside the system to be sent to receptionist actors, and in turn, messages from inside to be sent to external actors.

At a superficial level, one can see that the strong encapsulation of local state in the Actor formalism provides support for modular specification. However, there is as yet no satisfactory formalization of the composition of open actor systems. We believe that research in this direction will provide an understanding of both asynchrony and object-based programming.

References

[Abr88] S. Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming.* Addison Wesley, 1988.

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

[Agh89] G. Agha. Supporting multiparadigm programming on actor architectures. In *Proceedings of Parallel Architectures and Languages Europe, Vol. II: Parallel Languages (PARLE '89)*, pages 1–19. Espirit, Springer-Verlag, 1989. LNCS 366.

[Agh90a] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[Agh90b] G. Agha. The structure and semantics of actor languages. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, pages 1–59. Springer-Verlag, 1990. LNCS 489.

[AS88] W. Athas and C. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, pages 9–23, August 1988.

[BB90] B. Berry and G. Boudol. The chemical abstract machine. In *17th ACM Annual Symposium on Principles of Programming Languages*, 1990.

[Bou89] B. Boudol. Towards a lambda-calculus for concurrent and communicating systems. *Lecture Notes in Computer Science*, 1989.

[BVN91] F. Baude and G. Vidal-Naquet. Actors as a parallel programming model. In *Proceedings of 8th Symposium on Theoretical Aspects of Computer Science*, 1991. LNCS 480.

[Cli81] W. Clinger. Foundations of actor semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory, May 1981.

[CS84] G. Costa and C. Stirling. A fair calculus of communicating systems. *Lecture Notes in Computer Science*, 1984.

[dB91] F.S. de Boer. A proof system for the language pool. *Lecture Notes in Computer Science*, 1991.

[Hew77] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[Mil90] R. Millner. Functions as processes. Technical report, INRIA, 1990.

[Mos86] Y. Moses. Knowledge in a distributed environment. Technical Report STAN-CS-86-1120, Stanford University, 1986.

[MPW91] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 1991. to appear.

[SNP87] R. K. Shyamasundar, K. T. Narayana, and T. Pitassi. Semantics of nondeterministic asynchronous broadcast networks. In *ICALP*, 1987. Full version to appear in Information and Computation.

[SVN91] Y. Sami and G. Vidal-Naquet. Formalisation of the behaviour of actors by colored petri nets and some applications. In *Proceedings of Parallel Architectures and Languages Europe, (PARLE '91)*, 1991.

[TS89] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In *OOPSLA Proceedings*, 1989.

[Wal90] D. Walker. Pi-calculus semantics of object-oriented programming languages. In *LFCS*, 1990.