Actors as a Coordinating Model of Computation

N. Raja and R.K. Shyamasundar
School of Technology & Computer Science
Tata Institute of Fundamental Research
Mumbai 400 005, India
Email: {raja, shyam}@tifr.res.in

Keywords: Actors, agents, pi-calculus

Received: November 24, 2004

This paper relates two prominent models of concurrent computation, namely Actors *and the* π-calculus. *We build on a thesis that proclaims – Actors enact the role of a* coordinating *model of computation. We enrich the Actor model by defining a mechanism for achieving a higher level of abstraction. This helps in reasoning with collections of Actors termed* Actor Troupes. *We identify a notion of* interaction equivalence *between Actor Troupes; and provide a semantic foundation for the enriched Actor model, in terms of the* π*-calculus – which has emerged as the canonical process calculus for the semantic analysis of object-based concurrent systems. Furthermore, we show that the algebraic notion of barbed bisimilarity in the* π*-calculus, corresponds precisely to interaction equivalence of the corresponding Actor Troupes.*

Povzetek: Predstavljena sta dva računska modela - z akterji in π*-računi.*

1 Introduction

There has been an exponential increase in the number of new paradigms that are being proposed to model concurrent distributed computation. This number far exceeds the corresponding figure for sequential computation. (This is understandable as the basic sequential architecture, the von Neumann model, has essentially remained unchanged.) Despite this fact, concurrency is less well understood than sequential computation. It would be an understatement to say that there is an urgent need for a *coordinating* model of computation which can interconnect and unify the varied paradigms. The solution to the problem of finding such a model should commence with a search among the existing models, rather than in the immediate proposal of yet another novel paradigm. The factors guiding us in this search should be at least threefold – expressive power of the model; relative efficiency of execution of the model with respect to paradigms based on orthogonal features; and the demonstrated existence of a sound semantic basis for the model.

This paper is based on a thesis that proclaims, *Actors* enact the role of a coordinating model of computation. Existing evidence corroborating such a thesis is the following: Actors have been shown to be an expressive medium which can easily mimic other paradigms [2, 3]; and the execution efficiency of Actors has been shown to be as efficient [7] as the shared memory models (which are orthogonal to the message passing paradigm of Actors). The only factor that remains is the provision of a semantic foundation. This paper aims to further the above thesis by taking a step in the direction of providing a semantic basis to Actors.

Among the various process-calculus approaches to the algebraic analysis of concurrency, the π-calculus is conspicuous by its success. With a well-developed body of theoretical work supporting it, the π-calculus has attained a canonical status in the semantic frameworks of object-based concurrent systems, analogous to the λ-calculus in sequential programming. The semantic power of the π-calculus has been demonstrated in many ways – by encoding the λ-calculus [26]; by embedding various data-types [23]; by translating higher-order primitives [23]; and by using it as a semantic domain for various object-based concurrent languages [31]. All these facts provide a compelling basis to choose the π-calculus as a semantic foundation for Actors.

This paper relates two prominent models of concurrent computation – Actors and the π-calculus – in a precise way, and has the following significant contributions:

1. It argues that Actors can play the role of a coordinating model of computation, due to the simplicity and inherent flexibility of the Actor primitives.

2. It enriches the Actor model by defining the notion of an *Actor Troupe* – which is a mechanism to achieve a higher level of abstraction – by restricting the visibility of some Actors.

3. It provides a semantic foundation to the enriched Actor model by mapping it to the π-calculus – which has emerged as the canonical process calculus for the semantic analysis of object-based concurrent systems.

4. It identifies an equivalence relation, *interaction equivalence* on Actor Troupes, and shows that under the translation this corresponds to the algebraic notion of barbed bisimilarity in the π-calculus semantics.

The rest of this paper is organized as follows: Section 2 gives an introduction to the Actor model of computation; Section 3 introduces the basic π-calculus notions required for the purposes of this paper; Section 4 develops a higher level of abstraction called Actor Troupe and defines a notion of equivalence between them; Section 5 demonstrates the translation process from actor systems to the π-calculus; Section 6 shows that the embedding is semantics preserving; Section 7 reviews related work; and finally Section 8 examines avenues for further research.

2 Actor Model of Computation

Actors [15] form one of the earliest proposed models of concurrent distributed computation. They include very few primitive constructs, but serve as a framework for studying various issues in computation. An *actor system* consists of a finite set of three basic entities – *actors, messages,* and *behavior definitions*.

The formal abstract syntax is shown in Figure 1. (Note that in Figure 1 the following non-terminals: $< actorName >$,$< behaviorName >$, $< method >$, and $< var >$ are all identifiers for which there are no corresponding production rules. Furthermore, although $< acqList >$ and $< parameters >$ are defined by the same production rule, they are distinguished for the sake of clarity of exposition.)

Actors embody the spirit of objects. Every actor has a unique name, and a unique mailbox address which remains unaltered throughout the lifetime of the actor. In other words, there is an implicit injective function mapping actor names to mailbox addresses. (We shall make this function explicit in the translation we provide.) It is at this address that the actor receives messages. The body of an actor consists of a *state,* an *acquaintance list,* and a collection of *methods* with relevant *actions*. The *state* – encapsulated, persistent, and private – is made up of variables, which in turn contain references to other actors. The *acquaintance list* is a collection of names of actors which are known to the present actor at the time of its creation. It is important to note that, the name of an actor may not be known to all other actors in the system. Conversely, an actor may not be aware of the names of all the other actors. *Messages* can be sent only to those actors whose addresses are known. Apart from the addresses that make up the acquaintance list initially, an actor might receive the addresses of more actors through the contents of incoming messages. Furthermore, every actor is also aware of its own mailbox address. Thus every actor can send messages to itself. The address of an actor is contained in its own acquaintance list. In particular, we stipulate that it occurs as the last element of its acquaintance list. Thus the acquaintance list of an actor is never empty, and always contains at least a single element namely its own address. Each *method* has a set of parameters, which is received along with the method name to be serviced. Corresponding to each method, is a set of *actions* that the actor performs. We shall explain the actions after we deal with *behavior definitions*.

Behavior definitions are parametric actor definitions, parameterized over the state variables and acquaintance names. Behavior definitions by themselves are not actors – they provide templates for the creation of new actors. The parameters corresponding to the state and the acquaintances have to be specified and instantiated at the time of creating new actors.

Messages are the driving force of an actor system. This is due to the fact that computation in an actor system is carried out in response to messages received by the actors of the system. Every message has two distinct parts – *destination address* and *message contents*. The destination address is the mailbox address of an actor in the system to which the message is to be delivered. The message content comprises a *method name* and corresponding *parameters*. The actor at the destination is known to respond to this method name. The parameters comprise names of other actors. Message passing in actors is point to point and asynchronous. Message delivery is guaranteed but the despatch order need not be preserved even when we consider the arrival order of a sequence of messages addressed to the same actor. The guarantee of message delivery forms a type of fairness assumption [2].

Each instance of an actor can receive only one message. In response to a message received, which requires one of the methods of an actor to be processed, an actor may change its state, and may also perform a finite number of the following actions:

Send a message to another actor whose mail address is known;

Create a new actor using a behavior template, by providing all the parameters required for initialization;

Become an actor, which specifies the replacement behavior to come into effect, when the next message is processed.

It may be noted that the next incoming message at the same mail address is processed by another instance of the actor with the specified replacement behavior. The processing of the current message need not be completed before the replacement is specified. During start up time, an actor system consists of a collection of behavior definitions, together with a declaration of initial actors and messages. The actor system evolves in response to the messages sent to the system.

Example 2.1 (Actor R)**.** Actor R has an empty state, an acquaintance list comprising three actor names, and it processes the method name f. It takes requests for possibly transforming data by the method name f, and sends the result to the first actor on its acquaintance list by the method name g. It specifies an identical replacement behavior to replace itself. The template of actor R is given by the following behavior definition, which is parametrized by the two actor names on its acquaintance list:

$$
\begin{array}{rcl}
< System > & ::= & \{< actorName > \ \leftarrow \ < behaviorDef >\}^* \\
< behaviorDef > & ::= & \texttt{Bdef} < behaviorName > \texttt{with} < state > \texttt{and} < acqList > \\
 & & \quad < method > (< parameters >) \ \rightarrow \ \{< actions >\}^* \\
 & & \qquad \vdots \\
 & & \texttt{endBdef} \\
< state > & ::= & \{< var > \ \leftarrow \ < actorname >\}^* \\
< acqList > & ::= & \{< actorName >\}^* \\
< parameters > & ::= & \{< actorName >\}^* \\
< actions > & ::= & \quad \texttt{become} < behaviorName > \texttt{with} < state > \texttt{and} < acqList > \\
 & & | \ \texttt{create} < behaviorName > \texttt{with} < state > \texttt{and} < acqList > \\
 & & | \ \texttt{send} < method > (< parameters >) \texttt{to} < actorName > \\
 & & \quad \text{where} < actorName > \in < acqList >
\end{array}
$$

Figure 1: An Abstract Syntax for a System of Actors.

Bdef
R with $<>$ and a', x', r'
$f(d) \ \rightarrow$ send $g(d)$ to a'
become R with $<>$ and a', x', r'
endBdef
An instance of actor R can be created by the following message:
create R with $<>$ and a', x', r'

As mentioned before, the state is made up of variables which in turn are represented by actors. The state may contain integers or other data-types. However for the sake of simplicity we consider only empty state configurations in the examples in this paper.

Example 2.2 (Actor A)**.** Actor A has an empty state, an acquaintance list comprising two actor names, and it processes the method name g. It takes requests for possibly transforming data by the method name g, and sends the result to the first actor on its acquaintance list by the method name h. It specifies an identical replacement behavior to replace itself. The template of actor A is given by the following behavior definition, which is parametrized by the actor name on its acquaintance list:
Bdef
A with $<>$ and b', a'
$g(d) \ \rightarrow$ send $h(d)$ to b'
become A with $<>$ and b', a'
endBdef
An instance of actor A can be created by the following message:
create A with $<>$ and b', a'

Example 2.3 (Actor A_1)**.** Actor A_1 has an empty state, an acquaintance list comprising two actor names, and it processes the method name g. It accepts requests for possibly transforming data by the method name g, and sends the result to the first actor on its acquaintance list by the method name m. It specifies an identical replacement behavior to replace itself. The template of actor A_1 is given by the following behavior definition, which is parametrized by the two actor names on its acquaintance list:
Bdef
A_1 with $<>$ and x', a'
$g(d) \ \rightarrow$ send $m(d)$ to x'
become A' with $<>$ and x', a'
endBdef
An instance of actor A_1 can be created by the following message:
create A_1 with $<>$ and x', a'

Example 2.4 (Actor B)**.** Actor B has an empty state, an acquaintance list comprising two actor names, and it processes the method name h. It accepts data by the method name h, and sends the untransformed data to the first actor on its acquaintance list by the method name m. It specifies an identical replacement behavior to replace itself. The template of actor B is given by the following behavior definition, which is parametrized by the two actor names on its acquaintance list:
Bdef
B with $<>$ and x', b'
$h(d) \ \rightarrow$ send $m(d)$ to x'
become B with $<>$ and x', b'
endBdef
An instance of actor B can be created by the following message:
create B with $<>$ and x', b'

Example 2.5 (Actor X)**.** Actor X has an empty state, an acquaintance list comprising two actor names, and responds to the method name m. It accepts data by the method name m, does nothing with the data, and specifies an identical replacement behavior to replace itself. The template of actor X is given by the following behavior def-

inition, which is parametrized by the two actor names on its acquaintance list:

`Bdef`
X `with <> and` r', x'
$m(d)$ $\rightarrow$ `become` X `with <> and` r', x'
`endBdef`

An instance of actor X can be created by the following message:

`create` X `with <> and` r', x'

3 The Polyadic π-calculus

In this section, we include a brief review of the polyadic π-calculus (PPC) [24, 23, 25] and also introduce the specific syntax that we use for it in this paper.

Following Milner's idea [22], a number of calculi for concurrent computation have been proposed, where the communication mechanisms are similar. Communication consists of synchronously sending and receiving messages through a shared labeled channel. PPC [23, 25, 9] is a model of concurrent computation that supports process mobility by naming and passing channels. It consciously forbids the transmission of processes as messages. One of its goals is to demonstrate that in some sense it is sufficiently powerful to allow only channel names to be the content of communications. PPC has two kinds of entities – *names* (*channels*) and *processes* (*agents*).

Definition 3.1 (Names and Processes)**.** Names $(x, y, \ldots \in \mathcal{X})$ *are atomic entities while* Processes $(P, Q, \ldots \in \mathcal{P})$ *have the following structure:*

$$P \ ::= \ N \ \mid \ (P|Q) \ \mid \ !P \ \mid \ (\nu x)P$$

where, Normal Processes $(M, N, \cdots \in \mathcal{N})$ *are defined as:*

$$N \ ::= \ \pi.P \ \mid \ 0 \ \mid \ M + N$$

and, the Prefix (π)*, is given by:*

$$\pi ::= x(\tilde{y}) \ \mid \ \overline{x}[\tilde{y}]$$

where, $\tilde{y}$ refers to a finite sequence of names.

The term 0 represents an inactive process, which cannot perform any action. We shall omit the trailing ".0" from process terms. Basic actions in PPC constitute *sending* or *receiving* names on channels. The construct $x(y)$ (called an *input prefix*) represents an atomic action, where name x *binds* name y. The process term $x(y).P$ waits for a name to be transmitted along channel x, *substitutes* the *received* name for all free occurrences of y in P, and then triggers P. The construct $\overline{x}[y]$ (representing an atomic action) *outputs* the name y along x, but *does not bind* name y. The form $P|Q$ denotes that P and Q are *concurrently* active, independent, and can *communicate*. The form $M + N$ means that the process can indulge in precisely one of the alternatives, given by M and N, for communication. Operator "!" is called *replication*, and $!P$ denotes $P|!P$. Finally, $(\nu x)P$ *restricts* the use of name x to P. Apart from input prefix, "ν" is another mechanism for *binding* names within a process term in API. Operator "ν" may also be thought of as *creating* new channels. As both mechanisms – input prefix and ν – bind names, we define $BoundNames(P)$ as those names with a bound occurrence in P, and $FreeNames(P)$ as those with a not bound occurrence in P. The basic rule of computation in PPC is provided by the *parallel composition* of processes which communicate along the same channel.

The operational semantics of PPC is given in two stages. A structural congruence is first defined over processes, as shown below; and then a reduction relation is defined as shown in Figure 2. Note that the rules do not allow reduction under prefix, sum, or replication.

Definition 3.2 (Structural Congruence)**.** The relation '$\equiv$' is the smallest congruence relation over processes such that the following laws hold:

1. Processes are identified if they only differ by a change of bound names.
2. $(\mathcal{N}/\equiv, +, 0)$ is an abelian monoid.
3. $(\mathcal{P}/\equiv, |, 0)$ is an abelian monoid.
4. $!P \equiv P|!P$
5. $(\nu x)0 \equiv 0, (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$
6. If $x \notin FreeNames(P)$ then $(\nu x)(P|Q) \equiv P|(\nu x)Q$

Furthermore, the synchronous π-calculus outlined above, can be suitably modified to yield the asynchronous π-calculus [10, 17]. The word 'asynchrony' in this calculus, means that message output is non-blocking. This is ensured by restricting the formation of a term $\overline{x}[\tilde{y}].P$ in the π-calculus to the case where P is a 'nil' process. However, it has been shown that under certain natural assumptions, the asynchronous version is strictly less expressive than the synchronous one [29].

PPC allows for the definition of a variety of equivalences between processes. Following Milner [27, 23], we define the notion of barbed bisimulation for PPC:

Definition 3.4 (Unguarded Process)**.** *A process Q occurs* unguarded *in P if it has some occurrence in P which is not under a prefix.*

Definition 3.5 (Observable Action)**.** *A process P can perform an* observable action *at x, written $P \downarrow_x$, if for some $x, \tilde{y}$, either the input prefix $x(\tilde{y}).Q$ or the output prefix $\overline{x}[\tilde{y}].Q$ occurs unguarded in P with x unrestricted.*

Let "$\rightarrow^*$" denote the transitive reflexive closure of "$\rightarrow$". We shall use $Q \rightarrow^* \downarrow_x$ to denote "$Q \rightarrow^* Q'$ for some Q', and $Q' \downarrow_x$".

Definition 3.6 (Barbed Bisimulation)**.** *A relation R_w over processes, is a* barbed simulation*, if $P \ R_w \ Q$ implies:*

Definition 3.3 (Reduction Relation)**.** The reduction relation $\rightarrow$ over processes is the smallest relation satisfying the following rules:

$$\text{Comm} \quad (\ldots + x(\tilde{y}).P) \mid (\ldots + \overline{x}[\tilde{z}].Q) \;\rightarrow\; P\{\tilde{y} \leftarrow \tilde{z}\} \mid Q$$

$$\text{Par} \quad \frac{P \rightarrow P'}{(P|Q) \rightarrow (P'|Q)}$$

$$\text{Struct} \quad \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

$$\text{Res} \quad \frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'}$$

Figure 2: Reduction Relation in PPC

1. *For each x, $P \downarrow_x$ implies $Q \rightarrow^* \downarrow_x$.*
2. *If $P \rightarrow P'$ then $Q \rightarrow^* Q'$ and $P'\ R_w\ Q'$;*

The relation R_w is a barbed bisimulation *if R and R^{-1} are* barbed simulations*. Processes P and Q are* barbed-bisimilar*, if $P\ R_w\ Q$ for some* barbed bisimulation R_w.

4 Actor Troupes: A Higher Level of Abstraction

An important requirement of a potential coordinating paradigm, which seeks to unify diverse models of concurrent computation, is the inherent support for various levels of abstraction. A desirable level of abstraction would be one which helps in dealing with bigger collections of Actors as if they were a single unit. Such a higher level of abstraction on actor systems can be defined by restricting and specifying the interface of actor systems (rather than individual actors) with the external world. We introduce the abstraction of *Actor Troupes* and also formally define a notion of *Interaction Equivalence* with respect to this abstraction.

Definition 4.1 (Actor Troupe)**.** *An* Actor Troupe *comprises actors, behavior definitions, and messages which satisfy the following conditions:*

1. *Certain actors within the troupe, declared* Receptionists*, are the only components whose existence is visible to the external world. Furthermore, only the mail addresses and the method names of Receptionists are visible outside.*
2. *Conversely, the components comprising the troupe are aware of the mail addresses and method names of a certain collection of* External actors *(fixed a priori) which are not members of the troupe. (Actor A is said to be aware of Actor B, if the acquaintance list of A contains the mail address of B).*

The notion of Actor Troupes helps in the modular development and composition of Actor programs, since it specifies the interface of a collection of actors with the external world. This can be observed by the fact that in any Actor Troupe only the Receptionists are capable of receiving messages from the external world. Furthermore, the *External Actors* are the only ones which can potentially receive messages from any member comprising the Actor Troupe. By adapting Milner's idea of experiments [22], we define a notion of *Interaction Equivalence* between actor troupes.

Definition 4.2 (Interaction Equivalence)**.** *Actor Troupes T_1 and T_2 are said to be interaction equivalent, when there is a nonempty relation R_a over Actor Troupes such that $T_1\ R_a\ T_2$ implies:*

1. *The* Receptionists *of T_1 and T_2 have the same locations, and respond to the same set of messages.*
2. *The* External Actors *that are known to T_1 and T_2 have the same locations, and respond to the same set of messages. (An external actor X is said to be known to Troupe T if the mail address of X is contained in the acquaintance list of any of the actors comprising Troupe T).*
3. *For every input message I sent to T_1 and T_2, where the message content of I does not contain the mail address of any component external to the troupe; the set of output messages O emanating from T_1 and T_2 are the same, and the message contents of O do not contain the mail address of any component internal to the troupe.*

4. *The Actor Troupes T_1' and T_2' which result from T_1 and T_2 respectively after they process message I, are in the relation $T_1'\ R_a\ T_2'$.*

The purpose of the definition is to ensure that if the above notion of equivalence is satisfied by the Actor Troupes T and T', then any occurrences of the Troupe T in an Actor program, may be replaced by the Troupe T' without any change in the meaning of the program. The above notion of equivalence does not permit the mail addresses of components to be carried in the messages because such communications violate and destroy the encapsulation of Actor Troupes.

Example 4.1 (Troupe T)**.** Comprises actors – R, A, and B, where R is the *Receptionist* – with a reference to an *External Actor* X. Actor R takes requests for transforming data by the message f, and sends the result to actor A by the message g. Actor A transforms the data further and sends the result to B by the message h. Actor B passes it on unchanged to the external actor X by the message m.

Example 4.2 (Troupe T_1)**.** Similar to Troupe T except for actor A which is replaced by actor A_1, which returns its results directly to the external actor.

5 Semantic Foundation for Actors

In this section, we present a semantic foundation for Actors in terms of the polyadic π-calculus (PPC). We shall concentrate on constructs that are unique to the Actor model. The translation uses, for each syntactic category, a mapping $[\![.]\!]$ from the Actor grammar to PPC process-terms. The translation also employs a set of auxiliary functions for "book-keeping" purposes, namely for maintaining correspondences between entities in the Actor formalism and PPC-names. Figure 3 is a formal description of the semantic function. The translation is explained in detail below.

Recall that in actor systems, behavior definitions are not actors. They are templates which enable creating actors. However, in the translation, we shall associate PPC processes with behavior definitions, and also with actors. The translation of *Behavior definitions* is given by:

$$[\![\texttt{BehaviorDef}]\!] \ = \ !\, l(\vec{s}, \vec{a}, i).([\![\texttt{Actor}]\!])$$

Behavior definitions are mapped to process terms containing the *Bang* operator, in order to model a resource which can create a new actor instance, every time it is requested. The PPC name l represents the location of the process term, and is statically determined by the function

$$\beta \ : \ BehaviourName \longrightarrow \text{PPC-}\textit{Name}$$

However, the association of actor addresses with PPC-names will be modeled by a dynamic mechanism. The tuples $\vec{s}$ and $\vec{a}$ are formal place holders for the *state* and *acquaintance* parameters which are supplied with each `create` and `become` request. The parameter i, again supplied by incoming requests, represents the address at which the newly created actor instance (or the replacement behavior) is to be located.

In the actor model the `create` primitive is endowed with an implicit capability of generating globally unique actor addresses on a purely local basis. This mechanism is modeled using the properties of the operator "ν" of PPC:

$$[\![\texttt{create}\ BehaviorName\ \texttt{with}\ state\ \texttt{and}\ acqList]\!]$$

$$= \ (\nu\, i)(\bar{l}[\vec{s}, \vec{a}, i])$$

The PPC-name l represents the location of the process term corresponding to the behavior definition which receives and services the `create` action. It is given as before by $\beta(BehaviourName) = l$. The function

$$\sigma \ : \ state \longrightarrow \text{PPC-}\textit{Names}$$

which maintains the correspondence between the *state* and PPC-names, gives $\sigma(state) = \vec{s}$. Similarly the correspondence between *acqList* and PPC-names is maintained by the function

$$\tau \ : \ acqList \longrightarrow \text{PPC-}\textit{Names}$$

which gives $\tau(acqList) = \vec{a}$.

The newly generated name i is guaranteed to be globally unique by the semantics of PPC [25, 23]. This can be explained as follows. Consider a PPC process term, $(\nu y)\ Q$, where the name y is bound by the restriction operator. The restriction mechanism combines two distinct roles in one operator. Firstly, it hides all interactions on the name y within Q, thus preventing external processes from interfering on communications along channel y. In effect, it declares a local name y, for use exclusively within Q. In this role it is similar to the '*let-in*' construct in programming languages, and the *hiding* mechanism of CCS. Secondly, the restriction operator also ensures that the name y is distinct from all external names too [25, 23]. This follows from the fact that PPC allows local names to be communicated to external processes. A term of the form $(\nu y)(\overline{x}[y].Q)$ can be viewed as simultaneously creating and transmitting a new name. The name y is at first local to Q and becomes active after the transmission. Thus the operator "ν" is a mechanism which creates globally unique channel names.

The "book-keeping" associated with the dynamic creation of addresses is managed in the semantic domain by the function

$$\alpha \ : \ \text{Actor} \longrightarrow \text{PPC-}\textit{Name}$$

whose definition enlarges after every creation of an actor instance.

The onus of providing a globally unique address for the newly created actor lies with the actor which issues the create command. This feature of the semantics guarantees the requirement of actor systems that at first the location of a

newly created actor is known only to its parent. Any other actor in the system becomes aware of the new arrival only on receiving the address of the newcomer. As we shall see later, this is a powerful mechanism to achieve modularity in actor systems by keeping certain actors hidden from the view of certain other actors.

The translation of `become` action is similar to that of `create` action:

$$[\![\texttt{become}\ BehaviorName\ \texttt{with}\ state\ \texttt{and}\ acqList]\!]$$
$$= \bar{l}[\vec{s}, \vec{a}, i]$$

but more simple. In the case of `become`, no new actor address needs to be generated because, the replacement is to be at the same location as its parent (namely '*i*'), even though both may exist concurrently. However, the parent cannot accept messages any longer. It is worthwhile to point out that we have modeled the `become` action exactly as envisaged by the pioneering work on Actors [15]. We place absolutely no restriction on the type of replacement behavior an actor instance might specify. For example, an actor instance created from a behavior definition A could specify its replacement to be created from the behavior definition Z.

The translation of the *actor instance* created by the *behavior definition* is:

$$[\![\texttt{Actor}]\!] = (\nu\ \vec{s})(\nu\ \vec{m})(\bar{i}[\vec{m}].\Sigma\vec{m}(\vec{p})[\![actions]\!])$$

The *actor instance* resides at location i, and its *state* $\vec{s}$ is encapsulated as shown by the restriction operator. The *actor instance* creates a tuple of channels $\vec{m}$ – which has as many elements as the number of messages the actor responds to. The newly created channel names are accessible at its location i, and are used for receiving parameters corresponding to each of the messages that are available. However, a single instance of an actor can service only one message – as indicated by the summation operator of PPC. The translation of the `send` action:

$$[\![\texttt{send}\ method\ (parameters)\ \texttt{to}\ Actor]\!]$$
$$= i(\vec{m}).\overline{m_j}[\vec{p}]$$

shows that in order to execute the method m_j of actor i, the parameters $\vec{p}$ have to be sent on the corresponding channel name. The function

$$\mu_i\ :\ methods \longrightarrow \text{PPC-}\textit{Name}$$

which maintains the correspondence between the *methods* and PPC-names of actor i, gives $\mu_i(method) = m_j$. Also $\vec{m}$ is a list of all the PPC-names which can be used to access the different methods provided by the actor. Quite naturally we have $m_j \in \vec{m}$. Similarly the correspondence between *parameters* and PPC-names is maintained by the function

$$\rho\ :\ parameters \longrightarrow \text{PPC-}\textit{Names}$$

which gives $\rho(parameters) = \vec{p}$. Notice that the `send` action is serviced by actor instances, while the `create` and `become` actions are serviced by the behavior definitions.

In the pure actor formalism that we have considered, the entities *state*, *acqList*, and *parameters* refer to sequences of *Actor Names*. The translation of these entities is provided by the functions σ, τ, and ρ respectively. As explained earlier, these functions map the *Actor Names* pointed to by these entities to the corresponding PPC-names. Note that it is possible to add *integers* and other *data-types* to the actor formalism and translate them to PPC-processes [23, 26]. However, for the sake of simplicity we shall not consider the encoding of *data-types* in this paper.

Actor Troupes correspond to *Systems* of actors which satisfy the additional constraints imposed by Definition 4.1. These constraints can be easily imposed and verified by monitoring the messages from the troupe to the external world, and vice versa. The translation of *Actor Troupes* is then very similar to the translation of the *System* of actors explained till now in this section.

Example 5.1 (Translation of Troupe T)**.** Consider the Troupe T of example 4.1. Suppose that the templates of the actors A, B, R, and X are located at a, b, r, and x respectively. The PPC translations of the templates are as follows:

$$A \equiv !a(b', a').(\nu g)(\overline{a'}[g].g(d).b'(h).\overline{h}[d].\overline{a}[b', a'])$$
$$B \equiv !b(x', b').(\nu h)(\overline{b'}[h].h(d).x'(m).\overline{m}[d].\overline{b}[x', b'])$$
$$R \equiv !r(a', x', r').$$
$$(\nu f)(\overline{r'}[f].f(d).a'(g).\overline{g}[d].\overline{r}[a', x', r'])$$
$$X \equiv !x(r', x').(\nu m)(\overline{x'}[m].m(d).\overline{x}[r', x'])$$

The translations of the `create` operations which initialize the troupe T are as follows:

$$[\![\texttt{create}\ A\ \texttt{with}\ <>\ \texttt{and}\ b', a']\!] = (\nu\ a')\overline{a}[b', a']$$
$$[\![\texttt{create}\ B\ \texttt{with}\ <>\ \texttt{and}\ x', b']\!] = (\nu\ b')\overline{b}[x', b']$$
$$[\![\texttt{create}\ R\ \texttt{with}\ <>\ \texttt{and}\ a', x', r']\!]$$
$$= (\nu\ r')\overline{r}[a', x', r']$$
$$[\![\texttt{create}\ X\ \texttt{with}\ <>\ \texttt{and}\ r', x']\!] = (\nu\ x')\overline{x}[r', x']$$

As all the above four `create` operations are meant to initialize the same troupe T in the example under consideration, they can be combined with the translation of the behaviour definitions using parallel composition. Further evolution of the troupe is given in later examples.

Example 5.2 (Translation of Troupe T_1)**.** Consider the Troupe T_1 of example 4.2. The Troupe T_1 has been built by modifying Troupe T of example 4.1. The template A is replaced by the template A_1 at the same location as A; the template B is discarded; and the templates of R and X

$$[\![\texttt{System}]\!] = [\![\texttt{BehaviourDef}_1]\!] \mid \ldots \mid [\![\texttt{BehaviourDef}_n]\!]$$
with the auxiliary functions ($\alpha,\ \beta,\ \mu,\ \rho,\ \sigma$ and τ)

$$[\![\texttt{BehaviourDef}]\!] = !\, l(\vec{s}, \vec{a}, i).[\![\texttt{Actor}]\!]$$
where $\beta(BehaviourName) = l$

$$[\![\texttt{Actor}]\!] = (\nu\ \vec{s})(\nu\ \vec{m})(\bar{i}[\vec{m}].\Sigma\vec{m}(\vec{p})[\![actions]\!])$$

$$[\![\texttt{create}\ BehaviourName\ \texttt{with}\ state\ \texttt{and}\ acqList]\!] = (\nu i)(\bar{l}[\vec{s}, \vec{a}, i])$$
where $\beta(BehaviourName) = l;\ \sigma(state) = \vec{s};\ \tau(acqList) = \vec{a}$

$$[\![\texttt{become}\ BehaviourName\ \texttt{with}\ state\ \texttt{and}\ acqList]\!] = \bar{l}[\vec{s}, \vec{a}, i]$$
where $\beta(BehaviourName) = l;\ \sigma(state) = \vec{s};\ \tau(acqList) = \vec{a}$

$$[\![\texttt{send}\ method\ (parameters)\ \texttt{to}\ Actor]\!] = i(\vec{m}).\overline{m_j}[\vec{p}],\ \text{where}$$
$$\alpha(Actor) = i;\ \mu_i(method) = m_j,\ m_j \in \vec{m};\ \rho(parameters) = \vec{p}$$

Figure 3: Mapping Actors to PPC.

remain the same. The PPC process term corresponding to the template A_1 is given by

$$A_1 \equiv !a(x', a').(\nu g)(\overline{a'}[g].g(d).x'(m).\overline{m}[d].\overline{a}[x', a'])$$

The translation of the creation of an instance of actor A_1 is given by:
$[\![\texttt{create}\ A_1\ \texttt{with}\ <>\ \texttt{and}\ x', a']\!] = (\nu a')\overline{a}[x', a']$

5.1 Preserving fairness on message delivery

As we mentioned before, the guarantee of message delivery in the Actor model forms a type of fairness assumption [2]. Message delivery is guaranteed but the despatch order need not be preserved. In the semantic domain, the corresponding property which provides the means to preserve fairness, is given by the fact that the reduction rules of π-calculus ensure that allowed reductions do take place within an unbounded but finite number of reduction steps.

The semantics ensures the fairness condition on message delivery. This can be seen from the fact that the only cases where the *bang* ("!") operators arise in the semantic mapping are from the translations of behaviour definitions. Such infinitely replicating terms are always guarded by input prefix operators. However, infinitely replicating terms that are guarded by matching output prefixes never arise.

This suffices to rule out the occurrence of situations like the following:

$$P = (a(x).Q \mid \overline{a}[w]) \mid (!b(x) \mid !\overline{b}[z])$$

Situations similar to the above will never arise from our semantic mappings. It is important to note that if the translation allowed processes with behaviors similar to those above, then fairness is not ensured, since there is no guarantee that in the above process the following allowed reduction:

$$a(x).Q \mid \overline{a}[w]$$

would ever take place.

6 Semantic Correspondence

In this section we demonstrate that our embedding is a semantic preserving mapping from actor troupes to PPC processes. In particular, the semantic function defined by our embedding maps interaction equivalence of actor troupes to barbed bisimilarity of PPC processes.

Consider actor troupes T and T_1 whose definitions are given in examples 4.1, and 5.1 respectively. Not surprisingly, it turns out that Troupe T is *interaction equivalent* to Troupe T_1 (A detailed evolution of Troupes T and T_1 is provided in examples 6.1 and 6.2). So an actor program containing Troupe T can be transformed into an actor program which has Troupe T_1 in place of T. In the semantic domain this would correspond to the replacement of one PPC process with another. Such an operation would make sense only if the PPC processes corresponding to T and T_1 are equivalent in some way. In fact, our semantic mapping has precisely the required property of equivalence preservation. The equivalence in the semantic domain corresponds to *barbed bisimilarity* [23] of PPC processes (defined in Section 3).

The following theorem establishes that the semantic function preserves interaction equivalence of Actor Troupes.

Lemma 6.1. *If T_1, T_2 denote arbitrary actor troupes which are interaction equivalent, and $[\![T_1]\!]$, $[\![T_2]\!]$ denote their corresponding semantic mappings in* PPC*, then:* $[\![T_1]\!] \downarrow_x$ implies $[\![T_2]\!] \rightarrow^* \downarrow_x$

Proof: By simple induction over reduction rules.

Lemma 6.2. *If T_1, T_2 denote arbitrary actor troupes which are interaction equivalent, and $[\![T_1]\!]$, $[\![T_2]\!]$ denote their corresponding semantic mappings in* PPC*, then:* $[\![T_1]\!] \rightarrow^* \downarrow_x$ implies $[\![T_2]\!] \rightarrow^* \downarrow_x$

Proof: By simple induction over reduction rules.

Theorem 6.3 (Semantic Correspondence)**.** *For any two Actor Troupes T_1, T_2 and their corresponding semantic mappings $[\![T_1]\!]$, $[\![T_2]\!]$ in* PPC*, we have the following: If T_1 R_a T_2 where R_a is an interaction equivalence, then $[\![T_1]\!]$ R_w $[\![T_2]\!]$ where R_w is a barbed bisimulation.*

Proof: From the definition of Actor Troupes and from the definition of the semantic mapping it is easy to see that there is a one-to-one correspondence between the set comprising 'Receptionists and method names' and a certain 'subset of PPC channel names'. The channel names on which actions are observable, belong to this subset. Actions on all other channel names which do not correspond either to the Receptionists or their methods, are unobservable since they are bound by the restriction operator. By Lemma 6.1, for all x, $[\![T_1]\!] \downarrow_x$ implies $[\![T_2]\!] \rightarrow^* \downarrow_x$. Furthermore if $[\![T_1]\!]$ reaches a state $[\![T_1]\!]'$ through an unobservable action, then $[\![T_2]\!]$ can also reach a state $[\![T_2]\!]'$ through a series of unobservable actions such that the observable actions of $[\![T_1]\!]'$ and $[\![T_2]\!]'$ coincide (by Lemma 6.2) (where $[\![T_1]\!]'$ and $[\![T_2]\!]'$ denote PPC processes). Thus the semantic mapping preserves interaction equivalence by transforming it to bisimilarity. □

The following examples provide a concrete and detailed illustration of the fact that the troupes T and T_1 (of examples 4.1 and 4.2) are equivalent.

Example 6.1 (Evolution of Troupe T)**.** Consider the Troupe T of examples 4.1 and 5.1. In the following we shall use A, B, R, and X as abbreviations for the PPC terms of the corresponding templates. In the beginning, the configuration is

$(\nu\ a, b, r)(R \mid A \mid B) \mid X$

When the `create` messages are introduced for the sake of initializing troupe T, and the external actor X, we get

$(\nu\ a, b, r, a', b', r')(R \mid A \mid B \mid \overline{r}[a', x', r']$
$\mid \overline{a}[b', a'] \mid \overline{b}[x', b']) \mid X \mid \overline{x}[r', x']$

This causes transitions to occur on r, a, b, and x to give rise to

$(\nu\ a, b, r, a', b', r')$
$(R \mid A \mid B \mid R' \mid A' \mid B') \mid X' \mid X$

where R, A, B and X are as before; while

$$R' \equiv (\nu\ f)(\overline{r'}[f].\ f(d).a'(g).\overline{g}[d'].\ \overline{r}[a', x', r'])$$

$$A' \equiv (\nu\ g)(\overline{a'}[g].g(d').b'(h).\overline{h}[d''].\ \overline{a}[b', a'])$$

$$B' \equiv (\nu\ h)(\overline{b'}[h].h(d'').x'(m).\overline{m}[d''].\ \overline{b}[x', b'])$$

$$X' \equiv (\nu\ m)(\overline{x'}[m].m(d'').\ \overline{x}[r', x'])$$

R', A', B' and X' correspond to particular instances of actors, with addresses r', a', b', and x' respectively. They are created from the templates R, A, B and X respectively.

The next step of the evolution is caused as the result of the message, `send` $f(d)$ `to` r', being sent to the troupe. The above `send` operation translates to the PPC expression $r'(f).\overline{f}[d]$ which is placed in parallel with the existing configuration.

Thus we now have

$(\nu\ a, b, r, a', b', r')(R \mid A \mid B \mid R' \mid A' \mid B')$
$\mid X \mid X' \mid r'(f).\overline{f}[d]$
$\rightarrow (\nu\ a, b, r, a', b', r')(R \mid A \mid B$
$\mid a'(g).\overline{g}[d'].\overline{r}[a', x', r'] \mid A' \mid B') \mid X' \mid X$
$\rightarrow (\nu\ a, b, r, a', b', r')(R \mid A \mid B \mid \overline{r}[a', x', r']$
$\mid b'(h).\overline{h}[d''].\overline{a}[b', a'] \mid B') \mid X' \mid X$
$\rightarrow (\nu\ a, b, r, a', b', r')(R \mid A \mid B \mid R'$
$\mid \overline{a}[b', a'] \mid x'(m).\overline{m}[d''].\overline{b}[x', b']) \mid X' \mid X$
$\rightarrow (\nu\ a, b, r, a', b', r')(R \mid A \mid B$
$\mid R' \mid A' \mid x'(m).\overline{m}[d''].\overline{b}[x', b']) \mid X' \mid X$
$\rightarrow (\nu\ a, b, r, a', b', r')(R \mid A \mid B$
$\mid R' \mid A' \mid \overline{b}[x', b']) \mid \overline{x}[x'] \mid X$
$\rightarrow (\nu\ a, b, r, a', b', r')(R \mid A \mid B$
$\mid R' \mid A' \mid B') \mid X' \mid X$

Example 6.2 (Evolution of Troupe T_1)**.** Consider the Troupe T_1 of examples 4.2 and 5.2. The troupe T_1 has been built by modifying Troupe T, by replacing the template A by the template A_1 at the same location as A; the template B has been discarded; while the templates of R and X remain the same. The PPC process term corresponding to the template A_1 is given by

$A_1 \equiv (\nu\ g)(!a(x', a').\overline{a'}[g].$
$g(d').x'(m).\overline{m}[d''].\overline{a}[x', a'])$

After the appropriate create messages have been sent to the troupe, the configuration is as follows:

$(\nu\ a, r, a', r')(R \mid A_1 \mid R' \mid A_1') \mid X' \mid X$

where R, X, R', X' are as before; while

$A_1' \equiv (\nu\ g)(\overline{a'}[g].g(d').x'(m).\overline{m}[d''].\overline{a}[x', a'])$

We follow the changes in the troupe resulting from the arrival of the message, `send` $f(d)$ `to` r', which translates to $r'(f).\overline{f}[d]$.

Thus we now have

$(\nu\ a, r, a', r')(R \mid A_1 \mid R' \mid A_1')$
$\mid X' \mid X \mid r'(f).\overline{f}[d]$
$\rightarrow (\nu\ a, r, a', r')(R \mid A_1$
$\mid a'(g).\overline{g}[d'].\overline{r}[a', x', r'] \mid A_1') \mid X' \mid X$
$\rightarrow (\nu\ a, r, a', r')(R \mid A_1 \mid \overline{r}[a', x', r']$
$\mid x'(m).\overline{m}[d''].\overline{a}[x', a']) \mid X' \mid X$
$\rightarrow (\nu\ a, r, a', r')(R \mid A_1 \mid R'$
$\mid x'(m).\overline{m}[d''].\overline{a}[x', a']) \mid X' \mid X$
$\rightarrow (\nu\ a, r, a', r')(R \mid A_1 \mid R'$
$\mid \overline{a}[x', a']) \mid \overline{x}[x'] \mid X$
$\rightarrow (\nu\ a, r, a', r')(R \mid A_1 \mid R' \mid A_1') \mid X' \mid X$

Notice that the PPC processes corresponding to the Actor

Troupes T and T_1, are *barbed bisimilar*. Thus, the replacement is valid in the semantic domain as well.

7 Related Work and Comparisons

The Actor model is one of the earliest proposed paradigms of object-based concurrent computation. It has been treated in a rather informal fashion by most of the papers which originally proposed the model [15][14]. There have been a few attempts to give a formal semantics to Actors, in the decades that have passed since it was proposed [15]. Research work related to formulating a semantic basis for Actors may be broadly classified under three main subheadings, as explained in detail in the following three subsections.

7.1 Process-Algebraic Semantics for (Non-Actor) Object-Based Systems

Work that has used process algebras to give semantics to object-based systems can in turn be classified into two major groups depending on the type of process algebra they use – higher-order or first-order. Higher-order process algebras like CHOCS [37] allow only processes to be transmitted as messages. First-order process algebras like the π-calculus [25] allow only channels to be passed in communication.

Among papers that use first order process algebras are those by – Walker [40] who maps POOL to the π-calculus; Jones [19] who maps on object-based design notation called $\pi o \beta \lambda$ to the π-calculus; Pierce and Turner [31] who use the π-calculus for the design of concurrent object-based programming languages; Vaandrager [38] who maps POOL to the process algebra ACP [8]; and Honda and Tokoro [16] who map an object-based calculus to the process calculus reported in [17].

Researchers who use higher-order process algebras are: Nierstrasz [28] who uses the higher-order π-calculus to model his object-based calculus; Sangiorgi [34] proposes the higher-order π-calculus as a rival semantic domain to the π-calculus. But Papathomas [30] and Walker [41] show that the higher-order calculi provide no more conceptual advantages over the first-order process calculi, while modeling object-based systems.

7.2 Non Process-Algebraic Semantics for Actors

The work reported in [13], [12], and [39], model only the concurrent execution of Actors while completely ignoring the object-based features of Actors like persistent state, encapsulation, dynamic creation and reconfigurability. Hewitt and Baker [13] define the notion of an *activation order*, which is a partial order on events. An event x is said to *activate* an event y when y is related to some message created by x. They also define an *arrival order* on messages (for each actor), which is a linear order. The linear order arises from the condition that the mailing queue associated with an actor can receive only one message at a time. Concurrency is modeled by the history order which is defined as the transitive closure of the activation and arrival orders. Clinger [12] develops the above work by creating *event diagrams* from the activation order. An *event diagram* is a historical record of all the computations right from the initial stage. The event diagram together with the set of pending messages is said to form a *powerdomain* which is used to describe the concurrency of Actors. Vasconcelos and Tokoro [39] refine this work further by weakening the condition on activation orders which no longer requires each event to have at most one immediate predecessor. Thus they use the notion of *traces* to model concurrent executions where an event might be immediately be preceded by many events.

The papers [5, 36] deal with the object-based features of Actors as well as with the concurrency notions, Agha et al. [5] formulates an Actor language as an extension of a simple functional language, and employs transition systems to provide an operational semantics for Actors. It further demonstrates that in the presence of fairness assumptions, the three notions of equivalences (*testing, may,* and *must*), collapse into two classes. Talcott [36] characterizes actor languages by defining a notion of *abstract actor structure*, and provides a semantics for them using transition rules that use properties of concurrent rewriting systems. Janssens and Rozenberg [18] model a restricted version of actors using graph grammars, and introduce the notion of an abstract actor structure. Kahn and Saraswat [20] model actors as a special case of concurrent constraint programming.

7.3 Process-Algebraic Semantics for Actors

The only other paper in this category, apart from our work, is by Agha [2]. It is a preliminary attempt using CCS [22] to provide a semantics for Actors. However, only the concurrency features of Actors is modeled, while key aspects like encapsulation, dynamic creation, and unrestricted replacement behaviors are completely ignored. This is because of the limitations in the expressive power of CCS – which deals with synchronous agents having a static interconnection topology, and lacks dynamic creation of new channels and processes.

In contrast with all the related work on Actors, our approach gives a comprehensive process-algebraic interpretation of all the basic features of the Actor Model for the first time. Also noteworthy is the fact that we use π-calculus – which is synchronous – to simulate an asynchronous system like Actors. This is possible due to the power of the *Bang* (!) operator of the π-calculus, which allows unbounded replication of processes and thereby provides the capability to model asynchronous behavior. This particular feature of the π-calculus has been recognized by other researchers in a different setting [10, 17, 33].

8 Conclusion

We have related two prominent models of concurrent computation, namely Actors and the π-calculus, by presenting an elegant semantic mapping of all the fundamental constructs of actors in terms of the polyadic π-calculus. We have enriched the Actor model by defining a higher level of abstraction, and also have provided a notion of equivalence between them. There are several interesting avenues which present themselves for further exploration. The object-based nature of the primitive constructs of actors, should be extended to include other object-oriented notions as well. In particular the varieties of inheritance [21, 42, 11], like self-based inheritance, and delegation-based inheritance, and also the notion of subtyping should be studied in this setting. The actor model allows various levels of granularity and abstraction. This flexibility of the actor model should be explored further by encompassing the framework given by [6]. A sound basis for typed object-based computing [1], can be explored using extensions to the formalism of this paper. Such extensions should include the notion of types in the setting of Actors. The Actor model has been extended in a different way by [4], using constructs which seem to have strong connections with the paradigm of concurrent constraint programming. It would be meaningful to explore connections between ActorSpace [4] and the model proposed in this paper.

Acknowledgement

We wish to thank the anonymous referees for constructive comments which were of immense help in improving the content and presentation of this paper. Our thanks to Ms. Margaret D'Souza for typing and proofreading this paper.

References

[1] M. Abadi, and L. Cardelli (1995) A theory of primitive objects: Second-order systems, *Science of Computer Programming*, Vol. 25, pp. 81–116.

[2] G. Agha (1987) *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press.

[3] G. Agha (1989) Supporting Multiparadigm programming on Actor Architectures, *Proc. PARLE'89*, LNCS 366, Springer, pp. 1–19.

[4] G. Agha, and C. J. Callsen (1993) ActorSpace: An Open Distributed Programming Paradigm, *Proc. PPOPP'93*, ACM, pp. 23–32.

[5] G. Agha, I. A. Mason, S. Smith, and C. Talcott (1997) A Foundation for Actor Computation, *Journal of Functional Programming*, Vol. 7, pp. 1–72.

[6] J. M. Andreoli, H. Gallaire, and R. Pareschi (1995) Rule-Based Object Coordination, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, Springer, pp. 1–13.

[7] F. Baude, and G. Vidal-Naquet (1991) Actors as a Parallel Programming Model, *Proc. STACS'91*, LNCS 480, Springer, pp. 184–195.

[8] J. Bergstra, and J. Klop (1985) Algebra of communicating processes with abstraction, *Theoretical Computer Science*, Vol. 37, pp. 77–121.

[9] G. Berry, and G. Boudol (1992) The Chemical Abstract Machine, *Theoretical Computer Science*, Vol. 96, pp. 217–248.

[10] G. Boudol (1992) Asynchrony and the π-calculus, *Research Report*, Number 1702, INRIA Sophia-Antipolis.

[11] K. B. Bruce (1994) A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics, *Journal of Functional Programming*, Vol. 4, pp. 127–206.

[12] W. Clinger (1981) Foundations of Actor Semantics, *AI-TR*, Number 633, MIT.

[13] C. Hewitt, and H. Baker (1977) Laws for Communicating Parallel Processes, *Proc. IFIP*, pp. 987–992.

[14] C. Hewitt (1977) Viewing Control Structures as Patterns of Passing Messages, *Artificial Intelligence*, Vol. 8, pp. 323–364.

[15] C. Hewitt, P. Bishop, and R. Sterger (1973) A Universal Modulator Actor Formalism for Artificial Intelligence, *Proc. IJCAI*, pp. 235–245.

[16] K. Honda, and M. Tokoro (1991) A Small Calculus for Concurrent Objects, *OOPS Messenger*, Vol. 2, pp. 50–54.

[17] K. Honda, and M. Tokoro (1991) An Object Calculus for Asynchronous Communication, *Proc. ECOOP'91*, LNCS 512, Springer, pp. 133–147.

[18] D. Janssens, and G. Rozenberg (1989) Actor grammars, *Math. Systems Theory*, Vol. 22 (2), pp. 75–107.

[19] C. B. Jones (1993) A pi-calculus Semantics for an Object-Based Design Notation, *Proc. CONCUR*, LNCS 715, Springer, pp. 158–172.

[20] K. M. Kahn, and V. A. Saraswat (1990) Actors as a Special Case of Concurrent Constraint Programming, *Proc. ECOOP/OOPSLA'90*, ACM Press, pp. 57–66.

[21] B. Meyer (1993) Systematic Concurrent Object-Oriented Programming, *Communications of the ACM*, Vol. 36, pp. 56–80.

[22] R. Milner (1989) *Communication and Concurrency*, Prentice Hall.

[23] R. Milner (1999) *Communicating and Mobile Systems: The Pi Calculus*, Cambridge University Press.

[24] R. Milner (1993) Elements of Interaction, *CACM*, Vol. 36 (1), pp. 78–89.

[25] R. Milner, J. Parrow, and D. Walker (1992) A calculus of mobile processes (Parts I and II), *Information and Computation*, Vol. 100, pp. 1–77.

[26] R. Milner (1992) Functions as processes, *Mathematical Structures in Computer Science*, Vol. 2, pp. 119–141.

[27] R. Milner, and D. Sangiorgi (1992) Barbed Bisimulation, *Proc. ICALP*, LNCS 623, Springer, pp. 685–695.

[28] O. Nierstrasz (1992) Towards an Object Calculus, *Proc. Object-Based Concurrent Computing*, LNCS 612, Springer, pp. 1–20.

[29] C. Palamidessi (1997) Comparing the expressive power of the Synchronous and the Asynchronous pi-calculus, *Proc. POPL*, ACM, pp. 256–265.

[30] M. Papathomas (1991) A Unifying Framework for Process Calculus Semantics of Concurrent Object-oriented Languages, LNCS 612, Springer, pp. 53–79.

[31] B. C. Pierce, and D. N. Turner (1994) Concurrent Objects in a Process Calculus, *Proc. TPPP*, LNCS 907, Springer, pp. 187–215.

[32] B. C. Pierce, and D. N. Turner (2000) Pict: A Programming Language Based on the Pi-calculus, *Proof, Language and Interaction*: Essays in Honour of Robin Milner, MIT Press, pp. 455–494.

[33] N. Raja, and R. K. Shyamasundar (1996) Actors as a Coordinating Model of Computation, *Proc. Perspectives of System Informatics*, LNCS 1181, Springer, pp. 196–202.

[34] D. Sangiorgi (1993) From pi-calculus to higher-order pi-calculus and back, *Proc. TAPSOFT'93*, LNCS 668, Springer, pp. 151–166.

[35] D. Sangiorgi, and D. Walker (2001) *The Pi-Calculus - A Theory of Mobile Processes*, Cambridge University Press.

[36] C. Talcott (1997) Interaction Semantics for Components of Distributed Systems, *Proc. IFIP Workshop on Formal Methods for Open Object-based Distributed Systems*, Chapman & Hall, pp. 154–169.

[37] B. Thomsen (1993) Plain CHOCS: A Second Generation Calculus for Higher Order Processes, *Acta Informatica*, Vol. 30, pp. 1–59.

[38] F. W. Vaandrager (1990) Process algebra semantics of POOL, *Applications of Process Algebra*, Cambridge University Press, pp. 172–236.

[39] V. Vasconcelos, and M. Tokoro (1992) Trace Semantics for Actor Systems, *Proc. Object-Oriented Concurrent Computing*, LNCS 612, Springer, pp. 141–162.

[40] D. Walker (1991) π-calculus Semantics of Object-Oriented Programming Languages, *Proc. TACS'91*, LNCS 526, Springer, pp. 532–547.

[41] D. Walker (1995) Objects in the π-calculus, *Information and Computation*, Vol. 116, pp. 253–271.

[42] A. Yonezawa (1990) *ABCL: An Object-Oriented Concurrent System*, MIT Press.