AN ALGEBRA OF ACTORS

Mauro Gaspari Gianluigi Zavattaro

Department of Computer Science, University of Bologna,
Mura Anteo Zamboni 7, 40127 Bologna, Italy.

gaspari,zavattar@cs.unibo.it

Abstract: We introduce an object-oriented language following a "process algebra" style. The idea is to define a formalism that enjoys a clean formal definition allowing the reuse of the rich algebraic theory typical of the process algebras in a context where an high level object oriented programming style is preserved. We provide an operational semantics based on a labelled transition system which allows to discuss, e.g., how different notions of equivalence, such as standard and asynchronous bisimulation, can be adapted to reason about our language. Finally, we illustrate the framework showing that an explicit receive primitive expressing a synchronization constraint or an update operation on the state of an object can be implemented in the language preserving a notion of observation equivalence among objects.

INTRODUCTION

The object-oriented research community developed techniques, tools and environments that have been applied to several software development projects in the context of a wide range of application domains. In particular, distributed object-oriented programming is one of the most promising candidate paradigms to build large scale distributed systems. OMG the Object Management Group consortium, CORBA [28] the object-oriented standard for integrating applications running in heterogeneous distributed environments developed by OMG, and Java [30, 12], the internet language developed by Sun Microsystems, are all examples of such efforts.

Objects are the basic entities in an object-oriented system. Objects have a local memory, a set of attributes, a behaviour, and a set of procedures and/or functions (methods) that defines the meaningful operations. In distributed object-oriented systems objects are autonomous reactive units executing concurrently and interacting by message-passing, which is typically asynchronous and unordered.

On the other hand, most of the theoretical computer science efforts in the theory of concurrency are oriented to study process algebras such as CCS [17] or the π-calculus [18] which do not provide a direct representation of objects as first class entities.

In these formalisms processes are *stateless* entities (i.e. entities without an explicit local memory) which communicate exploiting synchronous message passing and the representation of an object involves a large number of processes [29].

As a consequence of this situation there is a big gap between theory and practice, and whether or not results developed from the theory of concurrent systems, such as the theories of equivalence for process algebras, can be successfully applied to real object-oriented distributed systems, is still an open issue.

The aim of this paper is to provide a step in this direction showing that it is possible to define a process algebra based on a distributed object oriented model. The main result is the development of a formalism that enjoys a clean formal definition and a rich algebraic theory, like the π-calculus, while preserving a high level object oriented programming style. This formalism allows us to reuse standard results of the theory of concurrency in a context where object identity, asynchronous message passing, an implicit receive mechanism and support for dynamic object creation, are provided. In particular, we show that the language can be extended with primitives expressing state updating operations and synchronization constraints, preserving observation equivalence.

Objects as Actors

Process algebras, like CCS [17] and CSP [14], have been developed as formalisms for the study of concurrent systems. Initially, process algebras allowed interprocess communication via a static structure of channels between processes. Mobility, one of the basic features of modern object oriented systems (where new objects can be created at run time and/or moved in different locations), was not easily representable in these formalisms.

The π-calculus [20] can be considered the main attempt in order to overcome these limitations. In fact, it has been introduced as a calculus for *mobile* processes, *i.e.*, processes with a dynamically changing linkage structure. The π–calculus has been developed taking into account a synchronous handshake communication mechanism between processes. More recently [15, 8] also an asynchronous fragment of the π–calculus has been studied in order to analyze also the asynchronous communication mechanism and its similarities/differencies with the synchronous one [22].

There have been several attempts to adopt the π-calculus and its asynchronous version, for modelling interaction in the context of concurrent object oriented programming languages [18, 29, 23], but these approaches seem not completely satisfying mostly because they do not provide the concept of an object as a first class entity.

On the other hand, the actor model [13, 2] directly deals with many features of object oriented systems, such as object identity, asynchronous message passing, an implicit receive mechanism, and support for object creation; an actor has the same structural and interaction properties as an object.

The Actor Model

The actor model was introduced by Carl Hewitt about 20 years ago [13]. Actors are self-contained agents with a state and a behaviour which is a function of incoming communications. Each actor has a unique name (mail address) determined at the time of its creation. This name is used to specify the recipient of a message supporting object identity, a property of an object which distinguishes each object from all others. Object identity is a typical feature of object-oriented programming languages and it is used as basic dispatching mechanism in message passing. This property is not easily embeddable in formalisms such as CCS [17] (or asynchronous π-calculus [15, 8]), where message dispatching is performed by means of channels. In these formalisms the association address-process is not unique: a process may have several ports (channels) from which it receives messages and the same channel can be accessed by different processes.

Actors communicate by asynchronous and reliable message passing, *i.e.,* whenever a message is sent it must eventually be received by the target actor. Actors exploit an implicit receive mechanism. A receive operation is explicit when it appears in programs, while it is implicit when it does not correspond to an operation in the programming language and it is performed implicitly at certain points of the computation. An implicit receive mechanism is common in object-oriented programming where objects can be seen as passive entities which react to messages or to method invocation.

Actors make use of three basic primitives which are asynchronous and non-blocking: *create*, to create new actors; *send*, to send messages to other actors; and *become*, to change the behaviour of an actor [2].

There are four main differences between the asynchronous π-calculus and the actor model:

- The asynchronous π-calculus does not support first class object identity, while this is a basic feature of the actor model.
- In the asynchronous π-calculus processes are stateless entities while actors have an associated state.
- The asynchronous π-calculus is based on an explicit receive primitive, while actors exploit an implicit receive mechanism, which does not appears in programs.
- Finally, the asynchronous π-calculus does not assume a fair (reliable) message delivery mechanism, while the actor model assumes reliability.

Results

In the past few years, several advances have been achieved on the semantics of actors, dealing with aspects of communication and concurrency [5, 4, 26, 25, 16], but these papers do not investigate the relationships of the actor model with traditional process algebras, even though recently Robin Milner [19] suggested that it may be worthwhile to work in this direction. Thus, the question whether some of the results that have been proved in the context of process algebras can be imported in the actor model and in general in object-oriented distributed systems is still an open issue.

The main results presented here concern this issue. In particular, we provide a process algebra based on the actor model, we discuss how standard notions of equivalence can be formulated in this context, and we exploit the framework illustrating the encoding of an update operation on the state of an actor and of an explicit receive primitive expressing a synchronization constraint which preserves a notion of observation equivalence among actors. Our process algebra captures all the main features of the actor model except the reliability assumption.

AN ALGEBRA OF ACTORS

Let $\mathcal{A}$ be a countable set of *actor names*: a, b, c, a_i, b_i,... will range over $\mathcal{A}$ and L, L', L'',... will range over its power set $\mathcal{P}(\mathcal{A})$ (*i.e.*, $L, L', L'' \subseteq \mathcal{A}$). Let $\mathcal{V}$ be a set of values (with $\mathcal{A} \subset \mathcal{V}$) containing, e.g., NIL, $true$, $false$. We assume value expressions e built from actor names, value constants, value variables, the expressions $self$, $state$, and $message$, and any operator symbol we wish. In the example we will present we will use standard operators on sequences: $1st$, $2nd$, $rest$, $empty$. We will denote values with v, v', v'',... when they appear as contents of a message and with s, s', s'',... when they represent the state of an actor. $[\![e]\!]^a_s$ gives the value of e in $\mathcal{V}$ assuming that a and s are substituted for $self$ and $state$ inside e; e.g. $[\![self]\!]^a_s = a$ and $[\![state]\!]^a_s = s$. The special expression $message$ represents the contents of the last received message. Whenever a message is received, its contents is substituted for each occurrence of the expression $message$ in the receiving actor.

Let $\mathcal{C}$ be a set of *actor behaviours*: C, D, ... will range over $\mathcal{C}$. We suppose that every behaviour D is equipped with a corresponding definition $D \overset{def}{=} P$ where P is a program, that is a term defined by the following abstract syntax:

$$\begin{array}{rcl} P & ::= & become(C,e).P \mid send(e_1,e_2).P \mid create(b,C,e).P \mid \\ & & e_1 : P_1 + \ldots + e_n : P_n \mid \surd \end{array}$$

Observe that we allow recursive behaviours to be defined, for example we could have $C \overset{def}{=} become(C, state).\surd$.

Actor terms are defined by the following abstract syntax:

$$A \;\; ::= \;\; {}^aC_s \mid {}^a[P]_s \mid \langle a, v\rangle \mid A|A \mid A\backslash a \mid \mathbf{0}$$

An actor can be idle or active. An idle actor ${}^aC_s$ (composed by a behaviour C, a name a, and a state s) is ready to receive a message. When a message is received the actor becomes active. Active actors are denoted by ${}^a[P]_s$ where P is the program that is executed. The actor a will not receive new messages until it becomes idle (by performing a $become$ primitive). Sometimes the state s is omitted when empty (i.e. $s = \emptyset$). A program P is a sequence of actor primitives ($become$, $send$ and $create$) and guarded choices $e_1 : P_1 + \ldots + e_n : P_n$ terminating in the null program $\surd$ (which is usually omitted). An actor term is the parallel composition of (active and idle) actors and messages, each one denoted by a term $\langle a, v\rangle$ where v is the contents and a the name of the actor the message is sent to. Also a restriction operator $A\backslash a$ is used in order to allow the definition of local actor names ($A\backslash L$ is used as a shorthand for $A\backslash a_1\backslash \ldots \backslash a_n$ if $L = \{a_1, \ldots, a_n\}$) while $\mathbf{0}$ is the usual empty term.

The actor primitives and the guarded choice are described as follows.

- *send*:
  The program $send(e_1, e_2).P$ sends a message with contents e_2 to the actor

Table 1 Operational semantics.

Send	$^a[send(e_1,e_2).P]_s \xrightarrow{\tau} {}^a[P]_s \mid \langle [\![e_1]\!]^a_s, [\![e_2]\!]^a_s \rangle$	
Deliver	$\langle a,v\rangle \xrightarrow{\overline{a}v\emptyset} \mathbf{0}$	
Become	$^a[become(C,e).P']_s \xrightarrow{\tau} ({}^d[P'\{a/self\}]_s)\backslash d \mid {}^aC_{[\![e]\!]^a_s}$	d fresh
Create	$^a[create(b,C,e).P']_s \xrightarrow{\tau} ({}^a[P'\{d/b\}]_s \mid {}^dC_{[\![e]\!]^a_s})\backslash d$	d fresh
Receive	$^aC_s \xrightarrow{av} {}^a[P\{v/message\}]_s$	if $C \overset{def}{=} P$
Guard	$^a[e_1 : P_1 + \ldots + e_n : P_n]_s \xrightarrow{\tau} {}^a[P_i]_s$	if $[\![e_i]\!]^a_s = true$
Res	$\dfrac{A \xrightarrow{\alpha} A'}{A\backslash a \xrightarrow{\alpha} A'\backslash a}$	$a \notin n(\alpha)$
Open	$\dfrac{A \xrightarrow{\overline{a}vL} A'}{A\backslash b \xrightarrow{\overline{a}vL\cup\{b\}} A'}$	$a \neq b \wedge b \in n(v)$
Par	$\dfrac{A \xrightarrow{\alpha} A'}{A \mid B \xrightarrow{\alpha} A' \mid B}$	if $\alpha = \overline{a}vL$ then $a \notin act(B) \wedge$ $L \cap fn(B) = \emptyset$
Sync	$\dfrac{A \xrightarrow{av} A' \quad B \xrightarrow{\overline{a}vL} B'}{A \mid B \xrightarrow{\tau} (A' \mid B') \backslash L}$	
Cong	$\dfrac{B \equiv A \quad A \xrightarrow{\alpha} A' \quad A' \equiv B'}{B \xrightarrow{\alpha} B'}$	

indicated by e_1:
$^a[send(e_1,e_2).P]_s \xrightarrow{\tau} {}^a[P]_s \mid \langle [\![e_1]\!]^a_s, [\![e_2]\!]^a_s \rangle$
where τ represents an internal invisible step of computation.

- *become*:
  The program $become(C,e).P'$ changes the state of the actual actor from active to idle:
  $^a[become(C,e).P']_s \xrightarrow{\tau} ({}^d[P'\{a/self\}]_s)\backslash d \mid {}^aC_{[\![e]\!]^a_s}$ with d fresh
  The primitive $become$ is the only one that permits to change the state according to the expression e; we sometimes omit e if the state is left unchanged (i.e. $e = state$). The continuation P' is executed by the new actor $^d[P'\{a/self\}]_s$. This actor will never receive other messages (i.e. it is unreachable) as its name d cannot be known to any other actor. Indeed, the expression $self$, which is the

only one that returns the value d, is changed in order to refer to the name a of the initial actor.

- *create*:
  The program $create(b, C, e).P'$ creates a new idle actor having state s and behaviour C:
  $^a[create(b, C, e).P']_s \xrightarrow{\tau} (^a[P'\{d/b\}]_s \mid {}^dC_{[\![e]\!]^a_s})\backslash d$ with d fresh
  The new actor receives a fresh name d. This new name is initially known only to the creating actor, in fact a restriction on the new name d is introduced.

- $e_1 : P_1 + \ldots + e_n : P_n$:
  In the agent $e_1 : P_1 + \ldots + e_n : P_n$, the expressions e_i are supposed to be boolean expressions with value $true$ or $false$. The branch P_i can be chosen only if the value of the corresponding expression e_i is $true$:
  $^a[e_1 : P_1 + \ldots + e_n : P_n]_s \xrightarrow{\tau} {}^a[P_i]_s$ if $[\![e_i]\!]^a_s = true$

The function n returns the set of the actor names appearing in an expression, a program, or an actor term. Given the actor term A, the set $n(A)$ is partitioned in $fn(A)$ (the free names in A) and $bn(A)$ (the bound names in A) where the bound names are defined as those names a appearing in A only under the scope of some restriction on a. We use $act(A)$ to denote the set of the names of the actors in A. An actor term is well formed if and only if it does not contain two distinct actors with the same name. In the following we will consider only well formed agents, and we will use Γ to denote the set of well formed terms (A, B, D, E, F,... will range only over Γ).

We model the operational semantics of our language following the approach of Milner [18] which consists in separating the laws which govern the static relation among actors (for instance $A|B$ is equivalent to $B|A$) from the laws which rules their interaction. This is achieved defining a static structural equivalence relation over syntactic terms and a dynamic relation by means of a labelled transition system [24].

Definition 1 - Structural congruence, *is the smallest congruence relation over actor terms ($\equiv$) satisfying:*

$$\begin{array}{llllll}
(i) & ^a[\surd]_s \equiv 0 & (v) & 0\backslash a \equiv 0 & \\
(ii) & A|0 \equiv A & (vi) & (A\backslash a)\backslash b \equiv (A\backslash b)\backslash a & \\
(iii) & A|B \equiv B|A & (vii) & (A|B)\backslash a \equiv A|(B\backslash a) & \textit{where } a \notin fn(A) \\
(iv) & (A|B)|D \equiv A|(B|D) & (viii) & A\backslash a \equiv A\{b/a\}\backslash b & \textit{where } b \textit{ is fresh}
\end{array}$$

Definition 2 - Computations. *A transition system modelling computations in the actor algebra is represented by the triple $(\Gamma, T, \{\xrightarrow{\alpha} \mid \alpha \in T\})$. $T = \{\tau\} \cup \{av, \overline{av}L \mid a \in \mathcal{A},\ v \in \mathcal{V},\ L \subseteq \mathcal{A}\}$ is a set of labels, where τ is the invisible action standing for internal autonomous steps of computation; av and $\overline{av}L$ respectively represent the receiving and the emission of the message with receiver a and contents v. The set L in the label $\overline{av}L$ represents the set of actor names in the expression v which were initially under the scope of some restriction. $\xrightarrow{\alpha}$ is the minimal transition relation satisfying the axioms and rules presented in Table 1.*

The rules *Send*, *Become*, *Create* and *Guard* have been already discussed. Rule *Deliver* states that the term $\langle a, v\rangle$ (representing a message v sent to the actor a) is able to deliver its contents to the receiver by performing the action $\overline{av}\emptyset$. The corresponding receiving action labeled with av can be performed by the actor a when it is idle (rule *Receive*). The other rules are simply adaptation to our calculus of the standard laws for the π–calculus. The most interesting difference is due to the fact that in our calculus, more than one restriction can be extended by one single delivering operation. In fact, in our case the contents of a message is an expression instead of a unique name. This is the reason why we have added the set L to the label $\overline{av}L$. Another difference is in the rule *Par*: the actor term $A|B$ can deliver a message inferred by A (*i.e.*, execute an emission action $\overline{av}L$), only if B does not contain the target actor (*i.e.*, $a \notin act(B)$).

Discussion

There are several differences with respect to the formal semantics of actors in [5, 4] and in [26] which is worth to point out.

- We do not assume a fair message delivery mechanism as in [5, 4] and in [26].
- The algebra of actors describes only communication and synchronization primitives, while in the semantics of Agha et al. actor primitives are embedded in a functional language. This enables us to focus on concurrency and inter-agent communication related aspects and not deal with issues concerning the sequential execution of programs inside actors.
- The operational semantics of the algebra of actors is defined by means of a labelled transition system instead of a simple reduction system as in [5] or the rewriting rules in [26]. This allows to use standard observational equivalences of process algebras *e.g.*, bisimulation, testing, failure or trace, without defining explicit observers.
- We have introduced the guarded choice as an alternative to the conditional which is present in previous formalization of actors [5].
- We provide an explicit representation of the state of an object while in Agha et al. the state of an actor is represented as part of its behaviour.
- We have introduced a mechanism to model termination of actors. Actors are not perpetual processes with a default behaviour as usual, but they can terminate: an actor terminates whenever it finishes its internal computation. This is not a limitation because a perpetual actor can always be obtained performing an explicit become operation for each internal computation.
- In the algebra of actors, actors are created exploiting a single basic primitive, while in the semantics of Agha et al. the creation process is composed of two basic operations, the creation of an empty actor and the initialization of its behaviour. The main advantage of our approach is that we do not need to restrict the possible computations to guarantee an atomic create operation.

- We introduce a restriction operator similar to the one of the π-calculus. This operator is more tractable with respect to the approach of [5] based on the specification of the sets of receptionists and external actors in actor configurations. On the other hand, the calculus presented in [26] uses the inverse operator indicating the actors which are reachable from the outside world explicitly.

- In the operational semantics of Agha et al. a receiving rule that is reminiscent of the rule IN of [15] is used. This rule (as discussed in [7]) has the disadvantage to give rise to *infinite branching*: the transition system allows each term (containing at least one receptionist) to activate an infinite number of transition, at least one for each possible message that can be sent to one of the receptionists. If, for example, a receptionist will be no more able to receive a message (e.g., it is executing an infinite computation) or external actors never send messages to a receptionist, the transition system make possible (infinite) useless transitions. One of the most important advantages of the rule IN is that it allows the definition of observational semantics (*e.g.,* bisimulation) that capture interesting aspects of asynchronous communication. Instead we follow the approach of [7], where it is shown that the same observational semantics can be obtained by eliminating the problem of infinite branching by slightly modifying the usual (synchronous) observational semantics.

- Finally, here we define only equivalences for actor terms while Agha et al. [6] consider equivalences for both actor expressions and actor configurations. However, it is not difficult to define equivalences for processes also in our setting. For example, we could consider two expressions equivalent whenever they are interchangeable in each possible actor term.

EQUIVALENCE OF ACTOR TERMS

As already stated, one of the advantages of having introduced a semantics for actors based on a labeled transition system is that standard observational semantics for process algebras can be used. In this section we investigate two of them based on the notion of bisimulation: the *weak bisimulation* [17] (only bisimulation in the following) and the *asynchronous weak bisimulation* [15, 7] (only asynchronous bisimulation in the following) which is the corresponding equivalence for languages based on asynchronous communication.

Bisimulation

In order to define equivalences which does not take into account the τ steps, we recall the notion of *weak* transition which allows to contract successive τ-steps:

$$\begin{array}{ll} P \overset{\tau}{\Longrightarrow} P' & \text{iff } P(\xrightarrow{\tau})^{\star}P' \\ P \overset{\alpha}{\Longrightarrow} P' & \text{iff exists } P'' \text{ and } P''' \text{ s.t. } P \overset{\tau}{\Longrightarrow} P'' \xrightarrow{\alpha} P''' \overset{\tau}{\Longrightarrow} P' \quad (\text{for } \alpha \neq \tau) \end{array}$$

Observe that given $P \overset{\tau}{\Longrightarrow} P'$ also the case in which no steps are performed is permitted (in this case P' is the same as P).

Definition 3 - Bisimulation. *A symmetric relation $\mathcal{R}$ on actor terms ($\mathcal{R} \subseteq \Gamma \times \Gamma$) is a* bisimulation *if $(A, B) \in \mathcal{R}$ implies:*

- *if $A \xrightarrow{\alpha} A'$ then there exists B' such that $B \overset{\alpha}{\Longrightarrow} B'$ and $(A', B') \in \mathcal{R}$.*

Two actors A and B are bisimilar, *written $A \approx B$, if there exists a bisimulation $\mathcal{R}$ such that $(A, B) \in \mathcal{R}$.*

As for the asynchronous π-calculus [7], also in our language the bisimulation relation is a congruence; in fact, we have that if $A \approx B$ then for every actor term D and actor name a, $A|D \approx B|D$ and $A \setminus a \approx B \setminus a$.

Example 1 *Since for actors there is arrival-order non-determinism in message delivery, it is expected that the bisimulation equivalence does not depend on the order in which send operations are performed. To illustrate that the bisimulation captures this notion we consider a simple example of two actor terms: $A = {}^{a}BreakPair1$ and $B = {}^{a}BreakPair2$ which receive pairs and forward to the actor b the elements of the pair: $BreakPair1$ in the same order they appear in the pair, $BreakPair2$ in the inverse one.*

$$\begin{aligned} BreakPair1 &\overset{def}{=} send(b, 1st(message)).send(b, 2nd(message)). \\ &\quad\; become(BreakPair1) \\ BreakPair2 &\overset{def}{=} send(b, 2nd(message)).send(b, 1st(message)). \\ &\quad\; become(BreakPair2) \end{aligned}$$

We have $A \approx B$. In fact, the actors A and B cannot be distinguished because the sending order cannot be observed (the emission of a message consists of a local τ-step).

Example 2 *Consider the actor term $A = {}^{a}Double$ which receives messages represented as pairs (b, v) where the first argument is an actor name and the second argument is an integer, and sends to the actor b the integer $2 * v$. This behaviour is defined formally below:*

$$Double \overset{def}{=} send(1st(message), 2 * 2nd(message)).become(Double)$$

Suppose now that we want to build an interface that receives messages and forwards them to an actor which doubles them. This job is performed by the actor term:

$$B = {}^{a}Forward \mid {}^{b}Double$$

where the behaviour $Forward$ is:

$$Forward \overset{def}{=} send(b, message).become(Forward)$$

The actor terms A and B are not equivalent because the term B has two addresses that can be reached from the outside (the action bv cannot be observed in the term A).

The intuition of restriction is to make the restricted actors unreachable from the outside. Thus, if we add a restriction on actor b, the action bv can not be observed and the term:

$$B' = ({}^{a}Forward \mid {}^{b}Double)\backslash b$$

is equivalent to A. Note that we abstract away from details of internal communication: the synchronization of actor b, which receives a message from actor a, is an internal action labelled τ (rule $Sinc$ in Table 1) which is not observable (hence it does not have any effect on bisimulation).

Example 3 *We illustrate here a scenario similar to the previous example where actors have a significant internal state. Consider the actor term $A_1 = {}^{a}Sum_s$ (where s is an integer), which receives messages represented as pairs (b, v), where the first argument is an actor name and the second argument is an integer, updates the state to $s + v$ and sends b the integer $s + v$. This behaviour is defined formally below:*

$$\begin{aligned} Sum \overset{def}{=} \; & send(1st(message), 2nd(message) + state). \\ & become(Sum, 2nd(message) + state) \end{aligned}$$

The evolution of the state is modelled by the rule $Become$ in Table 1: a become operation updates the state of the actor, but the new state can be accessed only after the next receive operation.

Suppose now that we want to compose this actor with the interface actor defined in the previous example: we define the actor term: $B_1 = {}^{a}Forward \mid {}^{b}Sum_s$. As stated above actor terms A_1 and B_1 are not equivalent because the term B_1 has two addresses that can be reached from the outside. But, if we define $B_1' = ({}^{a}Forward \mid {}^{b}Sum_s)\backslash b$, we can prove the equivalence of the actors A_1 and B_1' abstracting away from details of internal communication.

Example 4 *Finally, we also show that the same example holds for actors which include create operations. Consider the actor term $A_2 = {}^{a}Fact$, which computes the factorial of a given integer. This actor takes as input messages having the form (a, v) where a is an actor name and v is an integer, and returns to the actor a the factorial of v.*

$$\begin{aligned} Fact \overset{def}{=} \; & (2nd(message) = 0) : send(1st(message), 1).become(Fact) + \\ & (2nd(message) > 0) : create(d, Mul, message). \\ & \qquad send(self, (d, (2nd(message) - 1))). \\ & \qquad become(Fact) \end{aligned}$$

$$Mul \overset{def}{=} send(1st(state), 2nd(state) * message)$$

This is a standard example for actors showing that recursion can be implemented exploiting the create operation [3]. The actor with behaviour $Fact$ creates an actor whose behaviour will be to multiply $2nd(message)$ with an integer it receives, to send the reply to $1st(message)$ and then to terminate. After, it requires itself to evaluate the factorial of $2nd(message) - 1$ and sends the answer to the new created actor d.

Again, we can compose the actor term A_2 with the actor which forwards messages obtaining the actors $B_2 = {}^{a}Forward \mid {}^{b}Fact$ and $B_2' = ({}^{a}Forward \mid {}^{b}Fact)\backslash b$. Finally, we can prove that actors A_2 and B_2' are equivalent: the two terms intuitively have the same input/output behaviour and the $Create$ rule in Table 1 guarantees that all the new actors are restricted and, thus, not reachable from the outside: all the internal actions are labelled τ thus are not observable.

Asynchronous Bisimulation

For languages based on asynchronous communication a new notion of *asynchronous bisimulation* has been introduced in [15] and formally analyzed in [7]. The basic difference between the asynchronous bisimulation and the standard (synchronous) one, is that in the asynchronous case, the action of removing a message and immediately reintroducing it, is considered as unobservable. In fact, an asynchronous observer, is supposed to be able to observe only the messages present in the communication medium without knowing if a certain actor is waiting or not for a message.

Definition 4 - Asynchronous bisimulation. *A symmetric relation $\mathcal{R}$ on actor terms ($\mathcal{R} \subseteq \Gamma \times \Gamma$) is an* asynchronous bisimulation *if* $(A, B) \in \mathcal{R}$ *implies:*

- *if* $A \xrightarrow{\alpha} A'$ *where* $\alpha \neq av$ *then there exists* B' *such that* $B \overset{\alpha}{\Longrightarrow} B'$ *and* $(A', B') \in \mathcal{R}$.
- *if* $A \xrightarrow{av} A'$ *then there exists* B' *such that* $B \overset{av}{\Longrightarrow} B'$ *and* $(A', B') \in \mathcal{R}$ *or* $B \overset{\tau}{\Longrightarrow} B'$ *and* $(A', B'|\langle a, v\rangle) \in \mathcal{R}$.

Two actors A and B are asynchronous bisimilar, *written* $A \approx_a B$, *if there exists an asynchronous bisimulation $\mathcal{R}$ such that* $(A, B) \in \mathcal{R}$.

As for the standard bisimulation, also the asynchronous bisimulation is a congruence. The asynchronous bisimulation allows us to formally analyze interesting aspects of the actor model.

Example 5 *We consider two actors implementing two different communication media: a queue and an ether,* i.e., *an unordered set (mailbox) of messages [17]. The behaviours of the two actors are defined as follows:*

$$
\begin{array}{l}
QUEUE \overset{def}{=} \\
(1st(message) = get \ \wedge \ empty(state)) : \ send(self, message). \\
\qquad\qquad become(QUEUE) + \\
(1st(message) = get \ \wedge \ \neg empty(state)) : become(QUEUE, rest(state)). \\
\qquad\qquad send(2nd(message), 1st(state)) + \\
(1st(message) = put) : \ become(QUEUE, insert\_last(2nd(message), state))
\end{array}
$$

$$
\begin{array}{l}
ETHER \overset{def}{=} \\
(1st(message) = get \ \wedge \ empty(state)) : \ send(self, message). \\
\qquad\qquad become(ETHER) + \\
(1st(message) = get \ \wedge \ \neg empty(state)) : become(ETHER, rest(state)). \\
\qquad\qquad send(2nd(message), 1st(state)) + \\
(1st(message) = put) : \ become(ETHER, insert\_rand(2nd(message), state))
\end{array}
$$

The two actor programs are assumed to receive messages with the following structure: $(put, item)$ *or* $(get, sender)$. *The two programs differ only for the functions* *insert_last and insert_rand, the first inserts a message at the end of a sequence and the second in a random position.*

We observe that the two actors ${}^{a}ETHER$ and ${}^{a}QUEUE$ are equivalent under asynchronous bisimulation, i.e., ${}^{a}ETHER \approx_a {}^{a}QUEUE$. *This result also follows from the property of arrival-order non-determinism in message delivery, which does not allow to know in what order the two actors implementing the two different channels read the requests sent to them. If we consider the standard bisimulation the two actors are instead distinguished.*

EXPLOITATION

Synchronization Constraints

The actor model does not provide an explicit primitive for receiving a certain kind of message, in fact an actor can read a message only when it is idle and each available message can be read independently from its contents. On the other hand, in real applications is often necessary to express synchronization constraints which restrict the set of messages that can be received at a certain point of the computation.

Here we show that it is possible to program in our algebra a new primitive $receive(e)$ which forces to receive only a message with contents e. In particular, we present an implementation of the new primitive in the initial algebra which preserves the asynchronous bisimulation semantics; in other words, we prove that for every actor containing such a new primitive, there exists an equivalent term which does not contain $receive$ commands.

Suppose to extend the syntax of the language by allowing also programs of the following kind:

$$P ::= receive(e).P$$

having the following operational semantics:

$${}^{a}[receive(e).P]_s \xrightarrow{a[e]^{a}_{s}} {}^{a}[P]_s$$

Our idea for implementing the program $receive(e).P$ in a term $[\![receive(e).P]\!]$ is to define a behaviour which executes the program P only if a message with contents expressed by e has been received; otherwise it resends the received message and becomes idle waiting for another one:

$$[\![receive(e).P]\!] \overset{def}{=} become(RECEIVE, state)$$

where:

$$\begin{aligned} RECEIVE \overset{def}{=} &(message = e) : P + \\ &(message \neq e) : send(self, message). \\ &\qquad become(RECEIVE, state) \end{aligned}$$

The correctness of our mapping is proved by the fact that ${}^{a}[receive(e).P]_s \approx_a {}^{a}[[\![receive(e).P]\!]]_s$ for every a and s. On the other hand, the standard (synchronous) bisimulation is not preserved. This is because the implementation uses the technique of immediately reintroducing the received messages (when different from e) that, as stated above, is observed by the standard bisimulation and not by the asynchronous one.

One feature of this implementation is that the encoding of a *receive* command could introduces a busy waiting; for example, if only messages different from e are sent to the actor the messages are repeatedly received and resent. Even if the encoding could introduce this divergent behaviour, asynchronous bisimulation is preserved because it is not divergence sensitive.

Update Operation

In the actor model the state of an actor can be changed by a *become* primitive, but the updated state is not accessible from the part of the program following the *become* primitive (see rule *Become* in Table 1). This feature depends from the fact that the *become* primitive transforms an actor from active to idle and the updated state becomes active only when the actor receives another message. Thus, the actor model does not provide an explicit primitive that changes the state of an actor leaving it active on the updated state; however, such a primitive is often useful in programming parallel applications. For instance, this is the case if we need to register that a given message has been received, and we want to perform the rest of the computation taking this new information into account.

Here we show that a new primitive $newstate(e)$ which only changes the state can be implemented in our language. We first extend the syntax of the language by allowing also:

$$P \ ::= \ newstate(e).P$$

and the operational semantics by adding the axiom:

$$^{a}[newstate(e).P]_{s} \xrightarrow{\tau} {}^{a}[P]_{[\![e]\!]^{a}_{s}}$$

The program $newstate(e).P$ first changes the state of the actor, and then executes the program P. In order to have the same behaviour in the initial algebra, we first use the *become* primitive in order to change the state. After the execution of *become*, the actor becomes idle and waits for a new message. If the received message is different from go, then the message is reintroduced in the communication medium, otherwise (if go is received) the remaining program P is performed:

$$[\![newstate(e).P]\!] \stackrel{def}{=} become(WAIT, e).send(self, go)$$

where:

$$\begin{aligned} WAIT \stackrel{def}{=} & (message = go) : P + \\ & (message \neq go) : send(self, message).become(WAIT, state) \end{aligned}$$

This implementation preserves the asynchronous bisimulation semantics: in fact $^{a}[newstate(e).P]_{s} \approx_{a} {}^{a}[[\![newstate(e).P]\!]]_{s}$ holds for every a and s.

As we do not consider the fairness assumption, it could happen that the encoding introduces divergent behaviours. Indeed, a message different from go could be received (and then resent) infinitely many times before the message go is processed. As discussed above, asynchronous bisimulation is divergence insensitive; hence, the addition of this particular behaviour does not permit to distinguish one term from its encoding.

CONCLUSION

The main results presented in this paper concern the study of the relationship between the actor model and process algebras. We have defined an algebra of actors where the fairness assumption is relaxed. This algebra enjoys a clean formal definition and a nice programming style. We have presented several programming examples and discussed different notions of equivalence based on standard and asynchronous bisimulation. Finally, we have presented the encoding of an update operation on the state of an object and the encoding of an explicit receive primitive expressing a synchronization constraint and we show that these encoding preserve a notion of observation equivalence among objects. An extended version of this paper [10] contains more programming examples and the encoding of the asynchronous π-calculus into the algebra of actors.

We believe that our approach is complementary to previous approaches to the semantics of actors, providing a new framework to discuss concurrency related aspects in this context.

We have used our algebra of actors in two different directions: (i) to model interaction in multi-agent systems [9], (ii) as a basis for an object-oriented formalism which has been used to specify the hurried philosophers case study [11]. This demonstrates that our process algebra can be successfully used to formalize more complex protocols and systems.

Besides the approaches cited in the Introduction, concerning the actor model, several approaches have been followed trying to define a semantic framework for modelling interaction in concurrent object oriented programming. It is worthwhile to recall here some of them.

The calculus presented in [15] allows the authors to define a notion of observation equivalence among processes in an asynchronous framework. This notion of equivalence has been proved to be captured by asynchronous bisimulation in [7]. The main limitation of this calculus, as the π-calculus, is that it does not support object identity.

In [27] a typed name-passing calculus is introduced. This calculus provides a method invocation mechanism based on asynchronous message passing. But, also this calculus, as the previous one, does not support object identity: there is no correspondence between objects and names of channels, *i.e.,* there may be more than one object sharing the same channel. Finally, as in our approach objects are not persistent, *i.e.,* they do not survive the reception and the processing of messages, unless this is programmed explicitly. On the other hand, the pure actor model provides persistent objects, which become ready to receive new messages whenever their internal computations terminate.

Finally, a number of additional research items still need to be carried out in our algebra. For instance: an encoding of the CORBA [28] operational model, which has been recognized a common model for several existing distributed systems and languages [21]; a study of how typing and inheritance issues, such as in [1], can be addressed; the formulation of algebraic laws that characterize the equivalences of actor terms as for example an axiomatization for the asynchronous bisimulation; and the definition of a framework for formal reasoning about programs, *e.g.*, following the style of the Hennessy and Milner logic [17].

Acknowledgments

This research was partially supported by the Italian Ministry of Universities (MURST). We are also grateful to the anonymous referee for their helpful suggestions.

References

[1] M. Abadi and L. Cardelli. An Imperative Object Calculus. *Theory and Practice of Object Systems*, 1(3):151–166, 1995.

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

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

[4] G. Agha, I. Mason, S. F. Smith, and C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(1):1–69, January 1997.

[5] G. Agha, I.A. Mason, S. Smith, and C. Talcott. Towards a Theory of Actor Computation. In *Proc. of CONCUR'92*, volume 630 of *Lecture Notes in Computer Science*, pages 564–579. Springer Verlag, 1992.

[6] G. Agha, I.A. Mason, S. Smith, and C. Talcott. A Foundation of Actor Computation. Technical report, University of Illinois, 1993.

[7] R. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.

[8] G. Boudol. Asynchrony and the *π–calculus*. Technical Report INRIA-92-1702, INRIA Sophia-Antipolis., 1992.

[9] M. Gaspari. Concurrency and knowledge-level communication in agent languages. *Artificial Intelligence*, 1998. To appear.

[10] M. Gaspari and G. Zavattaro. An algebra of actors. Technical Report UBLCS-97-4, Comp. Science Laboratory, Università di Bologna, Italy, May 1997.

[11] M. Gaspari and G. Zavattaro. An actor algebra for specifying distributed systems: the hurried philosophers case study. In G. Agha and F. Decindio, editors, *Concurrent Object-Oriented Programming and Petri Nets*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1998. To appear.

[12] M. Hamilton. Java and the Shift to Net-Centric Computing. *IEEE Computer*, 29(8):31–39, 1996.

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

[14] CAR. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[15] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *The Fifth European Conference on Object-Oriented Programming*, volume 512

of *Lecture Notes in Computer Science*, pages 141–162. Springer-Verlag, Berlin, 1991.

[16] I.A. Mason and C. Talcott. A Semantically sound Actor Translation. In *Proc. of ICALP'97*, volume 1256 of *Lecture Notes in Computer Science*, pages 369–378. Springer Verlag, 1997.

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

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

[19] R. Milner. Elements of interaction. *Communications of the ACM*, 36(1):79–89, January 1993.

[20] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes I and II. *Information and Computation*, 100(1):1–40 - 41–77, 1992.

[21] E. Najm and JB. Stefani. Computational Models for Open Distributed Systems. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 157–176, Canterbury, UK, 1997. Chapmann & Hall.

[22] C. Palamidessi. Comparing the expressive power of the Synchronous and the Asynchronous pi-calculus. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 256–265, 1997.

[23] B. C. Pierce and D. N. Turner. Concurrent Objects in a Process Calculus. In *invited lecture at Theory and Practice of Parallel Programming (TPPP)*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215, Sendai, Japan, nov 1994. Springer-Verlag, Berlin.

[24] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark, 1981.

[25] C. Talcott. An actor rewriting theory. In *Workshop on Rewriting Logic*, number 4 in Electronic Notes in Theoretical Computer Science. Elsevier, 1996.

[26] C. Talcott. Interaction Semantics for Components of Distributed Systems. In *Proc. of Formal Methods for Open Object-Based Distributed Systems*, pages 154–169. Chapman & Hall, 1996.

[27] V.T. Vasconcelos. Typed Concurrent Objects. In *8th European Conference on Object Oriented Programming*, volume 821 of *Lecture Notes in Computer Science*, pages 100–117. Springer-Verlag, Berlin, 1994.

[28] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazines*, 14(2), February 1997.

[29] D. Walker. π-calculus Semantics of Object-Oriented Programming Languages. *Information and Computation*, 116(2):253–271, 1995.

[30] E. Yourdon. Java, the Web, and Software Development. *IEEE Computer*, 29(8):25–30, 1996.