# An Actor Algebra for Specifying Distributed Systems: the Hurried Philosophers Case Study*

Mauro Gaspari and Gianluigi Zavattaro

Department of Computer Science, University of Bologna
Mura Anteo Zamboni 7, 40127 Bologna, Italy
E-mail: {gaspari,zavattar}@cs.unibo.it

**Abstract.** In this paper, we introduce an actor language following a "process algebra" notation. The idea is to define a formalism that enjoys a clean formal definition allowing the reuse of standard results of the theory of concurrency in a context where an high level object oriented specification style is preserved. To illustrate the expressive power of the language, we provide a specification of the Hurried Philosophers case study.

## 1 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 the object-oriented standard for integrating applications running in heterogeneous distributed environments developed by OMG, and Java, the new internet language developed by Sun Microsystems, are all examples of such efforts.

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 [15] or the $\pi$-calculus [16] which do not provide a direct representation of objects as first class entities. In these formalisms processes are stateless entities which communicate exploiting synchronous message passing and the representation of an object involves a large number of processes [21].

As a consequence of this situation there is a big gap between theory and practice: results developed from the theory of concurrent systems, such as the theories of equivalence for process algebras, can hardly be applied to real object oriented distributed systems.

The formal framework presented in this paper addresses this issue. In particular we introduce a process algebra based on an object-oriented model which

* This paper has been partially supported by the Italian Ministry of Universities (MURST).

has a clean formal definition, like the *pi*-calculus, while preserving basic object-oriented features, such as object identity, asynchronous message passing, implicit message acceptance and dynamic object creation. We exploit our framework showing how other object-oriented features such as attributes, methods and inheritance can be encoded in the algebra of actor and providing a specification of the Hurried Philosophers case study.

The paper is organized as follows. First, we investigate the basic communication mechanisms of distributed object-oriented languages, then we define a process algebra based on such mechanisms. Finally, to illustrate the expressive power of the language, we provide a specification of the Hurried Philosophers case study.

## 2 Distributed Object Oriented Programming

Objects are the basic run-time 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. Moreover, there is a notion of object-identity [13]. Identity is that property of an object which distinguishes each object from all others.

In distributed object-oriented systems objects are autonomous units executing concurrently and interacting by message-passing. Several models for distributed object-oriented programming have been proposed over the past years. Two main approaches have been followed:

- The first is based on $\pi$-calculus [16, 21, 12, 20], which can be seen as an extension of process algebras with the notion of naming and dynamic process creation.
- The second approach is based on the actor model [11, 2], which for a long time has been the unique model of concurrent computation able to deal with dynamic aspects such as process creation and transmission ($\pi$-calculus was designed about 10 years later).

In the following, we try to individuate the communication mechanisms which are common to most of these models and, consequently, which can be considered basic interaction mechanisms in this context.

### 2.1 Object Identity

Support for object-identity can be achieved by associating to each object a unique name or address. This notion of object-identity based on names has a natural mapping in the actor model [2]. Actors are named objects with 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. Conversely, object-identity is not easily embeddable in formalisms such as CCS [15] or $\pi$-calculus [16], 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.

### 2.2 Synchronous vs Asynchronous Message Passing

Objects can communicate by means of synchronous or asynchronous message-passing mechanisms: the synchronous mechanisms do not allow the buffering of messages, thus a message to be delivered needs a "hand-shake" between the sending object and the receiving object; this is the communication mechanism adopted by agents in CCS [15]. The asynchronous mechanism assumes that the buffering of messages is possible, thus it is not required an agreement with the receiving object to send a message; this is the approach adopted by actors [3] and by asynchronous process algebras [9].

Since our goal is to select a set of primitive interaction mechanisms, we need to specify which kind of message passing should be considered as primitive in this context. We claim that asynchronous message passing is the most suitable basic interaction mechanism for distributed object-oriented programming. This because synchronous communication can be modelled providing an adequate "rendez-vous" protocol, while, vice versa, if we want to model asynchronous message passing with a synchronous language, we need to introduce an extra entity to deal with the buffering of messages. However, the buffering of messages requires unbounded buffers; this assumption seems reasonable, expecially in all those sytems in which the storage media have a capacity which is largely greater than the average amount of pending messages.

### 2.3 Explicit vs Implicit Message Acceptance

Given that the basic message passing mechanism is asynchronous, and that the name of an object is used to specify the recipient of a message, we have still to establish which is the form of receive primitive which is more suitable to distributed object-oriented programming.

The receive primitive may be explicit or implicit. A receive operation is explicit when it appears in the program, 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: typically in actor systems the receive is implicit and is performed only when the actor is idle.

We think that the receive should be implicit in a basic calculus for distributed object-oriented programming: in this way, the communication has the form of an asynchronous object method invocation. Hence, each object has several methods, each one managing possible received messages. By the way, asynchronous message passing with an implicit receive mechanism is a basic feature of the actor model.

### 2.4 Staticity vs Dynamicity

An aspect which is central in distributed object-oriented programming is dynamicity, i.e., the possibility of creating new objects at run time. Most of the proposals for new languages [7, 6] handle dynamic issues by providing rigorous laws which regulate the creations of new objects and the transmission of channel and process identifiers.

### 2.5 Summary

As a result of the analysis presented above the basic interaction mechanisms, which are suitable to model distributed object-oriented, are close to those of the actor model. In fact, it provides support for object identity, asynchronous message passing, implicit receive mechanism and support for dynamicity. Thus, the second approach seems to be more adequate for our purposes and we have chosen the actor model as a model for the process algebra presented in this paper.

### 2.6 The Actor Model

The actor model was introduced by Carl Hewitt about 20 years ago [11] . 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. Actors communicate by asynchronous and reliable message passing exploiting an implicit receive mechanism.

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].

The current formalizations of actors, such as [3, 4, 19], do not exploit a "process algebra" style, for this reason it is difficult to import in the actor model techniques and results developed for other asynchronous calculi [8, 12, 5]. Unfortunately most of current efforts on the semantics of concurrent languages do not consider the actor model at all. Although recently Robin Milner [17] suggested that it may worth to investigate in this direction.

## 3 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 $\mathcal{A} \subset \mathcal{V}$ 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. 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 subsituted 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 countable set of *actor behaviours*: $C$, $D$, ... will range over $\mathcal{C}$. We suppose that every behaviour $D$ is equipped with a corresponding definition $D \stackrel{def}{=} P$ where $P$ is a program, that is a term defined by the following abstract syntax:

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

Observe that we allow also the recursive definition of behaviours, e.g,
$C \stackrel{def}{=} become(C, state).\surd$.

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). The state $s$ is sometimes 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 $\mathbf{0}$ stands for an empty actor.

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 described by $e_2$ to the actor 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 $\tau$ 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 unique 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

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

**Table 1.** Operational semantics.

expression $self$, which is the unique one that returns the value $d$, is changed in order to refere 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 $C$:

$${}^a[create(b,C,e).P']_s \xrightarrow{\tau} ({}^a[P'\{d/b\}]_s \mid {}^dC_{\llbracket e \rrbracket^a_s})\backslash d \qquad \text{with } d \text{ fresh}$$

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 $true$:

$$ {}^{a}[e_1 : P_1 + \ldots + e_n : P_n]_s \xrightarrow{\tau} {}^{a}[P_i]_s \qquad \text{if } [\![e_i]\!]^a_s = true $$

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 $\Gamma$ to denote the set of well formed terms ($A$, $B$, $D$, $E$, $F$,... will range only over $\Gamma$).

We model the operational semantics of our language following the approach of Milner [16] 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 [18].

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

| | | | | |
|---|---|---|---|---|
| $(i)$ | ${}^{a}\surd_s \equiv {}^{a}[\surd]_s \equiv \mathbf{0}$ | $(v)$ | $\mathbf{0}\backslash a \equiv \mathbf{0}$ | |
| $(ii)$ | $A\|\mathbf{0} \equiv A$ | $(vi)$ | $(A\backslash a)\backslash b \equiv (A\backslash b)\backslash a$ | |
| $(iii)$ | $A\|B \equiv B\|A$ | $(vii)$ | $(A\|B)\backslash a \equiv A\|(B\backslash a)$ | where $a \notin fn(A)$ |
| $(iv)$ | $(A\|B)\|D \equiv A\|(B\|D)$ | $(viii)$ | $A\backslash a \equiv A\{b/a\}\backslash b$ | where $b$ is fresh |

**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 $\tau$ 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 $\pi$–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.,* $a \notin act(B)$).

### 3.1 Discussion

There are several differences with respect to the formal semantics of actors in [3, 4] and in [19] which is worth to point out.

- We do not assume a fair message delivery mechanism as in [3, 4] and in [19].
- 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 interagent 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 [3] or the rewriting rules in [19]. 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 [3].
- 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 $\pi$-calculus. This operator is more tractable with respect to the approach of [3] based on the specification of the sets of receptionists and external actors in actor configurations. On the other hand, the calculus presented in [19] uses the inverse operator indicating the actors which are reachable from the outside world explicitly.
- Finally, in the operational semantics of Agha et al. a receiving rule that is reminiscent of the rule **IN** of [12] is used. This rule (as discussed in [5]) has the disadtvantage to give rise to *infinitely 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 allows (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 [5], where it is shown that the same oservational semantics can be obtained by eliminating the problem of infinitely branching by slighty modifying the usual (synchronous) observational semantics.

## 4 Hurried Philosophers

In this section we present a specification of the Hurried Philosophers case study, which aims to establish the expressive power of object-oriented formalisms, testing features such as dynamic object creation and termination; dynamic binding of communication channels (*i.e.,* the partner of a communication is defined when the communication occurs); inheritance (a class of objects may inherit some features from other classes of objects).

To improve the presentation, we introduce some further notations which will be useful in the specification, giving a direct representation of the standard object-oriented concepts of methods, inheritance and attributes. Then, we provide a specification of the traditional (static) version of dining philosophers problem [14]. Finally, we show how dynamic aspects concerning the creation of new philosophers and the deletion of them, can be added to the specification exploiting inheritance.

### 4.1 Methods, Inheritance and Attributes

To provide a more modular specification of the problem, we add some further syntactic notations and we show how the usual object-oriented concepts of methods, inheritance and attributes can be represented in the algebra of actors.

First of all we define a new boolean expression *otherwise* which can be used as possible guard in a choice:

$$e_1 : P_1 + e_2 : P_2 + \ldots + e_n : P_n + otherwise : P_{n+1}$$

Such a new program is the same as:

$$e_1 : P_1 + e_2 : P_2 + \ldots + e_n : P_n + \neg(e_1 \vee e_2 \vee \ldots \vee e_n) : P_{n+1}$$

The second extension we consider helps in writing programs incrementally, exploiting a form of inheritance. Given a behaviour of an actor defined as follows:

$$Beh \stackrel{def}{=} e_1 : P_1 + e_2 : P_2 + \ldots + e_n : P_n$$

we can consider programs $P_1, P_2, \ldots, P_n$ as methods, where their activation depends on the conditions $e_1, e_2, \ldots, e_n$ on the incoming message. In an object oriented scenario it could be useful to add new features to it, *i.e.,* more branches

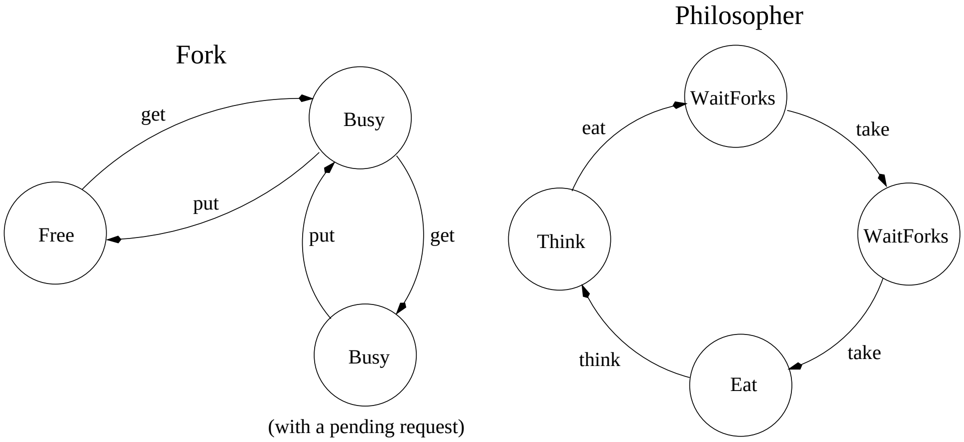


**Fig. 1.** Behaviours of forks and philosophers.

to the choice. This feature is obtained introducing a new operator for defining behaviours:

$$Beh \stackrel{ext}{=} e_{n+1} : P_{n+1} + e_{n+2} : P_{n+2} + \ldots + e_{n+m} : P_{n+m}$$

which redefines the initial behaviour in the following way:

$$Beh \stackrel{def}{=} e_1 : P_1 + e_2 : P_2 + \ldots + e_{n+m} : P_{n+m}$$

When a behaviour with an *otherwise* guard is extended as above, we suppose no new *otherwise* guards can be added, and the meaning of the *otherwise* is updated taking into account also the new conditions.

Since objects are usually modelled as entities having several attributes, we introduce a notation to represent records as possible structured values in actor expressions. A record $r$ is denoted with $(f_1 : e_1, \ldots, f_n : e_n)$ and the selection of the value in the field $f_i$ is written as $r.f_i$. An operator $\uplus$ which allows to add fields to records is assumed; it is defined as follows:

$$(f_1 : e_1, \ldots, f_n : e_n) \uplus (f_{n+1} : e_{n+1}) \stackrel{def}{=} (f_1 : e_1, \ldots, f_{n+1} : e_{n+1})$$

### 4.2 Static dining philosophers

In this section we present the specification of a static version of dining philosophers in which no new philosophers can be added to the table, neither philosophers can leave or change their place on the table. In order to avoid both deadlock and starvation, we use the right-left algorithm presented in [14]. It consists of having two categories of philosophers: *right* and *left* ones, asking for the forks in a different order. A right philosopher asks for the fork on his right first, he waits until the fork is available, and then he asks for the fork on his left. A

| | |
|---|---|
| **Free** $\stackrel{def}{=}$ | |
| $message.con = get:$ | $become(Busy).$ |
| | $send(message.act, (act: self, con: take))$ |
| **Busy** $\stackrel{def}{=}$ | |
| $message.con = get:$ | $become(Busy, message.act)+$ |
| $message.con = put \wedge state = null:$ | $become(Free)+$ |
| $message.con = put \wedge state \neq null:$ | $become(Busy, null).$ |
| | $send(state, (act: self, con: take))$ |
| **Think** $\stackrel{def}{=}$ | |
| $message.con = eat:$ | $become(WaitForks).$ |
| | $send(state.fork1, (act: self, con: get))$ |
| **WaitForks** $\stackrel{def}{=}$ | |
| $message.con = take \wedge$ | |
| $message.act = state.fork1:$ | $become(WaitForks).$ |
| | $send(state.fork2, (act: self, con: get))+$ |
| $message.con = take \wedge$ | |
| $message.act = state.fork2:$ | $become(Eat)$ |
| **Eat** $\stackrel{def}{=}$ | |
| $message.con = think:$ | $become(Think).$ |
| | $send(state.fork2, (act: self, con: put)).$ |
| | $send(state.fork1, (act: self, con: put))$ |

**Table 2.** Static dining philosophers.

left philosopher makes the same operations but inverts the order of the requests (*i.e.,* he asks for the left fork first). The philosophers are hurried: everytime a philosopher asks for a fork, if the fork is not free, his request is stored in order to give him the fork as soon as possible.

This algorithm ensures that neither deadlock nor starvation can occur, if at least one right and one left philosopher are present at the table. In fact, under this condition, no circular waiting chains can be reached during the computation (see [14] for more details on the proof of the correctness of the algorithm).

The way we implement the right-left algorithm consists in representing each philosopher and each fork as an actor. We assume a fixed structure for messages which are represented using records containing two fields: *act* indicating the name of the sender of the message, and *con* representing the contents of the message.

An actor representing a fork can be free or busy. The state of the fork $f_i$ can be *null* or $p_j$ indicating that the philosopher $p_j$ is waiting for that fork (hence when the fork $f_i$ will be freed it will be given to $p_j$) while *null* says that no philosophers is waiting for the fork $f_i$. When a fork receives a *get* request and it is free it becomes busy, otherwise if it is busy it stores such a request. This is needed to implement a fair policy. See Figure 1 for a graphical representation of

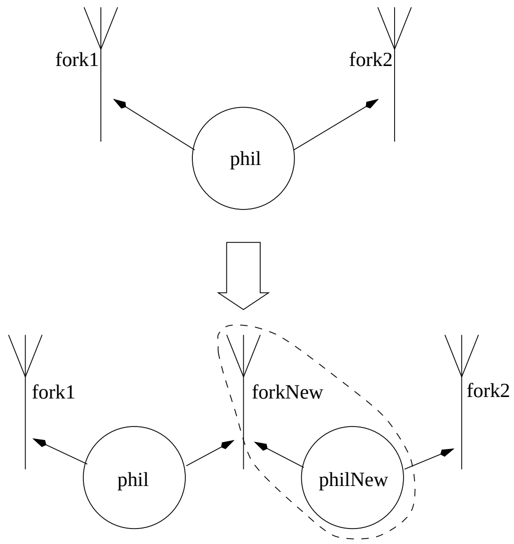


**Fig. 2.** Insertion of a new philosopher.

the possible behaviours of fork actors.

Initially all the philosophers are thinking. When a philosopher receives a message having contents *eat*, he starts asking for the forks he needs to eat, and, eventually, he starts eating. A philosopher eats until he receives a message having contents *think*, in this case he frees the forks and starts thinking. Note that, we do not consider the source of the messages *eat* and *think*, we only define how the philosophers react to them. The state of an actor representing a philosopher is a record having the form: $(fork1 : f_i, fork2 : f_j)$ where $f_i$ is the name of the first fork that the philosopher asks for, and $f_j$ the name of the second one. In our solution actors do not know the names of their neighboring philosophers: a philosopher only knows his first and his second fork. See Figure 1 for a graphical representation of the possible behaviours of philosopher actors.

The actor term which specifies the static dining philosophers is the following one:

$$\begin{aligned}\mathbf{DinPhil} \stackrel{def}{=} & {}^{f_0}Free_{null} \mid \ldots \mid {}^{f_{n-1}}Free_{null} \mid \\ & {}^{p_0}[Think]_{(fork1:f_0, fork2:f_1)} \mid \ldots \mid {}^{p_{n-1}}[Think]_{(fork1:f_0, fork2:f_{n-1})}\end{aligned}$$

where the actors ${}^{f_i}Free_{null}$ and ${}^{p_i}[Think]_{(fork1:f_k, fork2:f_l)}$ represent the $i$-th fork and the $i$-th philosopher, respectively.[2] We explicitly require that the philosophers $p_0$ and $p_{n-1}$ share the first fork, in this way the condition of having at

[2] The state *null* of each fork indicate that no philosopher is waiting for it, while the fields $fork1$ and $fork2$ of the state of each philosopher contain the name of the first and of the second fork that the philosopher needs to eat.

least one right and one left philosopher is satisfied. The behaviours $Free$ and $Think$ are defined in Table 2 where in every behaviour is omitted the choice:

$$otherwise : send(self, message)$$

which resends to the same actor a message which can not be managed from the actual active message handlers.

### 4.3 Adding new philosophers

| | |
|---|---|
| **Heap** $\stackrel{def}{=}$ | |
| $message.con = newFork \wedge$ | |
| $state > 0 :$ | $become(Heap, state - 1).$ |
| | $create(d, Free, null).$ |
| | $send(message.act, (act : d, con : takeNew))+$ |
| $message.con = newFork \wedge$ | |
| $state = 0 :$ | $become(Heap).$ |
| | $send(message.act, (act : self, con : noNew))$ |
| **Think** $\stackrel{ext}{=}$ | |
| $message.con = create :$ | $become(WaitNewFork).$ |
| | $send(heap, (act : self, con : newFork))$ |
| **WaitNewFork** $\stackrel{def}{=}$ | |
| $message.con = takeNew :$ | $become(Think, (fork1 : message.act,$ |
| | $fork2 : state.fork2)).$ |
| | $create(d, Think, (fork1 : message.act,$ |
| | $fork2 : state.fork1))+$ |
| $message.con = noNew :$ | $become(Think)$ |

**Table 3.** Adding new philosophers.

In Table 3 we show how it is possible to extend the specification of static dinining philosophers of Table 2 introducing a protocol for creating new philosophers. First of all, we introduce a new heap actor:

$$^{h}Heap_{m}$$

which has the responsability of creating a new fork, whenever a new philosopher is added to the table. The maximum number of new forks that can be created is $m$ (the initial state of the heap actor). Thus, if $n$ is the number of starting philosophers, no more than $n + m$ philosophers can be present around the table at the same time. In fact, when the heap creates a new fork, the state of the heap is decremented, and when it reach 0 no further fork can be added.

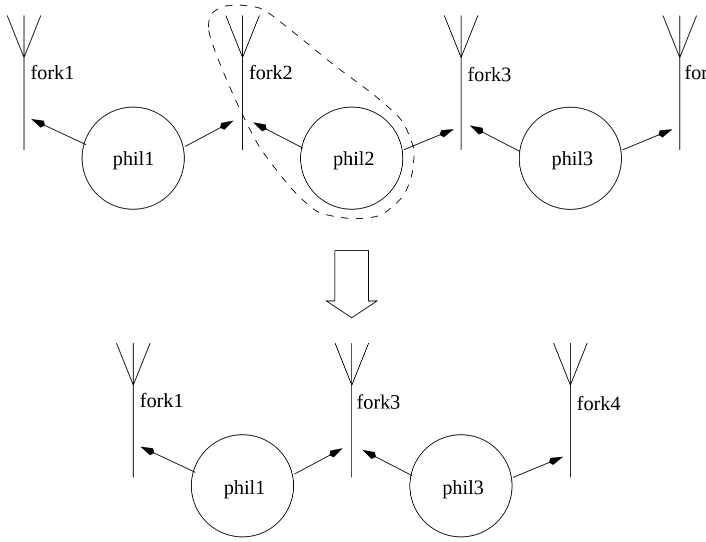


**Fig. 3.** Elimination of a philosopher.

The process of creation of a new philosopher starts when a philosopher who is thinking receives a message containing the *create* operation. A philosopher can manage this kind of messages only when he does not own any fork.

A philosopher, before creating his new neighbor, asks to the heap for a new fork. If the operation succeeds, the heap creates the new fork, and sends its name to the creating philosopher. Otherwise, the creation fails and the protocol terminates. When a philosopher receives from the heap the name of the new fork, he creates his new neighbor philosopher who is located in between himself and his first fork (see Figure 2). It is interesting to note that every philosopher is able to introduce new neighbors without knowing the name of the actual neighbors.

### 4.4 Eliminating philosophers

In Table 4 we show how it is possible to extend the specification of dining philosophers by introducing also the possibility of eliminating philosophers actually around the table. Also the process of elimination can start only when a philosopher is thinking. When a thinking philosopher receives a message containing the *kill* operation, he starts the protocol for leaving. To support this protocol a philosopher needs to know the names of his neighbors. Hence, we extend the state of each actor with two new fields $phil1$ and $phil2$ containing the name of the philosophers who share the first and the second fork, respectively.

When a philosopher starts the leaving protocol, he continues only if his neighbors are different, *i.e.,* the philosophers at the table are more than two. If the protocol continues, then the philosopher that is leaving the table asks the neighbors for the name of the forks they use in order to be able to define the new links.

| | |
|---|---|
| **Think** $\overset{ext}{=}$ | |
| $message.con = kill \wedge$ | |
| $state.phil1 \neq state.phil2:$ | $become(Leave).$ |
| | $send(state.phil1, (act : self, con : leave)) +$ |
| $message.con = kill \wedge$ | |
| $state.phil1 = state.phil2:$ | $become(Think) +$ |
| $message.con = leave:$ | $become(Halt).$ |
| | $send(message.act, (act : self, con : state))$ |
| **Leave** $\overset{def}{=}$ | |
| $message.act = state.phil1:$ | $become(Leave, state \uplus (stPhil1 : message.con)).$ |
| | $send(state.phil2, (act : self, con : leave)) +$ |
| $message.act = state.phil2:$ | $send(state.fork1, kill).$ |
| | $send(state.phil1, (act : self,$ |
| | $\quad con : compState(1, self, state$ |
| | $\quad \uplus(stPhil2 : message.con)))).$ |
| | $send(state.phil2, (act : self,$ |
| | $\quad con : compState(2, self, state$ |
| | $\quad \uplus(stPhil2 : message.con)))) +$ |
| $message.con = leave:$ | $become(Think)$ |
| **Halt** $\overset{def}{=}$ | |
| $message.act = state.phil1 \vee$ | |
| $message.act = state.phil2:$ | $become(Think, message.con)$ |
| **Free** $\overset{ext}{=}$ | |
| $message.con = kill:$ | $send(heap, (act : self, con : forkDies))$ |
| **Heap** $\overset{ext}{=}$ | |
| $message.con = forkDies:$ | $become(Heap, state + 1)$ |

**Table 4.** Eliminating philosophers.

When he receives these informations, he eliminates his first fork and he communicates to the neighbors that they have to share his second fork (see Figure 3). Moreover, in order to be sure that at least one right and one left philosopher stay at the table, the neighbors will share the same fork as first fork. The function $compState(i, self, state)$ computes the new state of the neighbor $i$ of the leaving philosopher:

$compState(i, self, state) \overset{def}{=}$
$rec := (fork1 : state.fork2, phil1 : state.phil_{\overline{i}})$
if $(state.stPhil_i.phil1 = self)$
then $rec := rec \uplus (fork2 : state.stPhil_i.fork2, phil2 : state.stPhil_i.phil2)$
else $rec := rec \uplus (fork2 : state.stPhil_i.fork1, phil2 : state.stPhil_i.phil1)$
return $rec$

where $\overline{i}$ is used to indicate 2 if $i = 1$, or 1 otherwise.

If a philosopher executing this protocol receives a message having contents *leave*, which means that also a neighbor philosopher is leaving, he aborts the protocol. This protocol guarantees that the definition of the new links between the remaining philosopher and forks is correct, *i.e.,* fairness is still preserved.

## 5 Conclusion

We have defined an algebra of actors that enjoys a clean formal definition and a nice specification style. We have presented the specification of the Hurried Philosopher case study which shows the expressive power of the algebra.

As we have already stated, the process algebraic definition that we have given to the actors, allows the reuse of standard techniques used in the field of asynchronous calculy. Actually, we are investigating the actor model by using the asynchronous bisimulation of [5] and we found interesting results presented in [10].

A number of additional research items still need to be carried out. Among them: a study of how typing and inheritance issues, such as in [1], can be addressed in our algebra; and the definition of a framework for formal reasoning about programs, *e.g.,* following the style of the Hennessy and Milner logic [15].

**Acknowledgements:** The authors would like to acknowledge discussions with Gul Agha, and to thank the anonymous referees for 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, I. 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.
4. G. Agha, I. Mason, S. F. Smith, and C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7:1–72, 1997.
5. R. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous $\pi$-Calculus. In *Proc.* CONCUR'96, volume 1119 of *Lecture Notes in Computer Science*, pages 147–162. Springer-Verlag, Berlin, 1996.
6. J. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. *New Generation Computing*, 9(3-4):445–473, 1991.
7. L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, January 1995.
8. P. Ciancarini, R. Gorrieri, and G. Zavattaro. Towards a Calculus for Generative Communication. In *Proc. FMOODS'96*, pages 283–297. Chapmann & Hall, 1996.
9. F. deBoer, J. Klop, and C. Palamidessi. Asynchronous Communication in Process Algebra. In *Proc. LICS'92*, pages 137–159. IEEE Computer Society Press, 1992.

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. C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.
12. K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *Proc. ECOOP'91*, volume 512 of *Lecture Notes in Computer Science*, pages 141–162. Springer-Verlag, Berlin, 1991.
13. S. Khoshafian and G. Copeland. Object Identity. In *Proc. OOPSLA'86*, pages 406–416, September 1986.
14. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1996.
15. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
16. R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
17. R. Milner. Elements of interaction. *Communications of the ACM*, 36(1):79–89, January 1993.
18. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Dep. of Comp. Science, Aarhus University, Denmark, 1981.
19. C. Talcott. Interaction Semantics for Components of Distributed Systems. In *Proc. FMOODS'96*, pages 154–169. Chapmann & Hall, 1996.
20. V. Vasconcelos. Typed Concurrent Objects. In *Proc. ECOOP'94*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1994.
21. D. Walker. Objects in the $\pi$-calculus. *Information and Computation*, 116(2):253–271, 1995.