Towards an Algebra of Actors

Mauro Gaspari

Technical Report UBLCS-96-9

April 1996

Department of Computer Science
University of Bologna
Piazza di Porta S. Donato, 5
40127 Bologna (Italy)
The University of Bologna Department of Computer Science Research Technical Reports are available in gzipped PostScript format via anonymous FTP from the area ftp.cs.unibo.it:/pub/TR/UBLCS or via WWW at URL http://www.cs.unibo.it/. Plain-text abstracts organized by year are available in the directory ABSTRACTS. All local authors can be reached via e-mail at the address last-name@cs.unibo.it. Questions and comments should be addressed to tr-admin@cs.unibo.it.

Recent Titles from the UBLCS Technical Report Series

94-20  Dynamic Allocation of Signature Files in Multiple-Disk Systems, P. Ciaccia, August 1994.
94-21  Parallel Independent Grid Files Based on a Dynamic Declustering Method Using Multiple Error Correcting Codes, P. Ciaccia, November 1994.
95-13  Optimal Multi-Block Read Schedules for Partitioned Signature Files, P. Ciaccia, August 1995.
96-5   The Shape of Shade: a Coordination System, S. Castellani, P. Ciancarini, D. Rossi, March 1996.
96-7   Using Bayesian Belief Networks for the Automated Assessment of Students’ Knowledge of Geometry Problem Solving Procedures, M. Roccetti, P. Salomoni, March 1996.
96-9   Towards an Algebra of Actors, M. Gaspari, April 1996.
Towards an Algebra of Actors

Mauro Gaspari

Technical Report UBLCS-96-9

April 1996

Abstract

We present a study of the interaction properties of objects in concurrent object oriented programming. We identify a set of basic interaction mechanisms: object identity, asynchronous message passing, implicit receive primitive, which are close to those of the actor model. Then, we define a minimal algebra of actors as a basic formalism for representing the semantics of concurrent object oriented programming. Finally, we define a notion of observation equivalence between the actor terms of the algebra which has been proved to be a congruence with respect to the parallel composition operator.
1 Introduction

A common semantic framework for sequential computation is given by the notion of mathematical function which can be formally expressed by means of a functional calculus, notably the Alonzo Church’s $\lambda$-calculus. For concurrent programming and interactive systems in general we have nothing comparable. Indeed, as Robin Milner stated in his Turing Award Lecture, “. . . a theory of concurrency and interaction requires a new conceptual framework not just a refinement of what we find natural for sequential computing . . . so concurrency requires a fresh approach not merely an extension of the repertoire of entities and constructions which explain sequential computing . . .”. Process algebras like CCS and CSP have been developed in order to represent the semantic of concurrent programs, but they are not suitable to model the semantic of all the possible paradigm of concurrent computation. For instance, they do not allow to model the behavior of parallel object oriented programs.

Two are the main approaches which have been followed trying to define a common semantic framework for concurrent object oriented programming. The first is based on $\pi$-calculus [16, 23, 13, 22], 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 [12, 1], 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).

The actor model provides a more abstract view of concurrent object-oriented programming with respect to $\pi$-calculus. In particular the actor model allows a direct representation of the central concept of object-oriented programming, the object, as an actor has structural and interaction properties which are close to those of an object, while $\pi$-calculus does not provide entities which may be considered comparable with objects and the representation of an object involves a large number of processes [23]. On the other hand, the advantages of the approaches based on $\pi$-calculus depends mainly on the strong theoretical foundations of the calculus which is simple elegant and tractable [18, 19].

The aim of our work has been to develop a fresh approach to the semantics of concurrent object-oriented programming as a compromize between the $\pi$-calculus and the actor model, toward the formulation of a concurrent object algebra.

In this paper, we investigate the basic communication mechanisms of concurrent object-oriented languages, which are close to those of the actor model, with the goal of defining an actor algebra following the style of CCS [15]. We define an algebra of actors (AA for short) which is an extension of the Static Actor Algebra Presented in [11]. Finally, we define a notion of observation equivalence between the actor terms of the algebra which has been proved to be a congruence with respect to the parallel composition operator.

2 Concurrent Object Oriented Programming

Objects are the basic run-time entities in an object-oriented system. Objects have a local memory, a set of attributes and, a behaviour, a set of procedures and/or functions (methods) that defines the meaningful operations.

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

In object-oriented programming there is a notion of object-identity [14]. Identity is that property of an object which distinguishes each object from all others. Support for object-identity can be achieved by associating to each object a unique name or address. The name of the object is also used as basic dispatching mechanisms in message passing. This notion of object-identity based on
names has a natural mapping in the actor model [1]. 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 (π-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 “rendez-vous” between the sending object and the receiving object; this is the communication mechanism adopted by agents in CCS [15]. The asynchronous mechanisms 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 [2] 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 concurrent object-oriented programming. This because synchronous communication can be modelled providing adequate synchronization constraints, 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.

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 concurrent 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 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 concurrent object-oriented programming. In fact, exploiting implicit message acceptance, we can assume that the state of an object changes only when a message is received. This enable us to model the evolution of the state of an object and how this is related to interaction. On the other hand, approaches which adopt explicit receive primitives such as POOL [4], do not provide such a strict relationship between state modifications and communications. 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 concurrent 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.

The actor model was introduced by Carl Hewitt about 20 years ago [12] and for a long time it has been the unique model of concurrent computations able to deal with dynamic aspects, (π-calculus was designed about 10 years after) but 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.

2.5 Discussion

Other models and theories have been followed trying to give an appropriate semantics to concurrent object-oriented systems, but the main goal of most of these works is different from ours. As an
example, the approach based on denotational semantics presented in [5], and the approach based on Petri-nets [10], which appears in the same volume, provide different semantic descriptions for a particular object oriented language: POOL [5]. On the other hand our aim is to individuate and study the basic interaction mechanisms which are more suitable to model concurrent object-oriented languages in general.

We claim that these mechanism are those of the actor model, and that the actor model should be considered the basic model for concurrent object-oriented programming. In fact, it captures all the basic interaction mechanisms we have pointed out above providing: support for object identity, asynchronous message passing, an implicit receive mechanism and support for dynamicity.

As a consequence of this, we started to analyze actors in details. One problem of the current approaches to the semantic description of actors is that the formalisms considered are quite complex, for instance [2] describes the semantics of a functional language which embeds actor based communication primitives. In this context is difficult to individuate and study the basic interaction mechanisms. For instance, if we consider CCS we know that it is possible to translate a calculus which include values in messages into a more basic calculus which deals only with synchronization [15]; it would be nice to prove similar results for actors.

The analysis we performed is based on a hierarchy of languages, each language is designed as an extension of the previous one [8]. For each language, we studied the operational semantics, the expressive power and the equivalence theory. The actor algebra we are describing in this paper is the minimal language, able to model an actor system, which has been individuated as a result of this analysis.

An interesting result is that a basic algebra include also values in messages, namely, it is not possible to translate the value passing actor algebra into an algebra which model synchronization only, i.e., a message contains the name of the recipient and an actor name. In particular, the technique based on the definition of different port names associated to different contents for messages, described in [15] cannot be applyed in the case of actors. Intuitively, because actors have associated a unique address and a unique state which depends from the contents of messages. Thus, it is not possible translate one actor in a set of actors dealing with different messages and maintain a consistent global state.

3 An Algebra of Actors

We introduce an algebra of actors as a basic formalism to model object oriented systems. In the actor model the behaviour of an object is a function of incoming communications. Actors are self-contained agents that communicate by asynchronous and reliable message passing making use of three basic primitives: create, to create new actors; send, to send messages to other actors; and become, to change the behaviour of an actor [1].

Let \( \mathcal{A} \) be a countable set of actors names: \( a, b, c, a_i, b_i \ldots \) will range over \( \mathcal{A} \). Let \( \mathcal{V} \) be a set of values \( \mathcal{V} \subseteq \mathcal{V}: \text{NIL, true, false, } k, k', k'', \ldots \) will be value constants; and \( x, y, z, \ldots \) range over \( \mathcal{V} \). We assume value expressions \( e \) built from actor names, value variables, value constants, the expression \( \text{self any operator symbol we wish.} \) As a consequence \( \mathcal{V} \) includes structured values such as \( (a, k) \), which we will denote with \( v, v', v'', \ldots \) when they appear in messages and with \( s, s', s'', \ldots \) when they represents the state of an actor. \( [e] \) gives the value of \( e \) in \( \mathcal{V} \); if \( a \in \mathcal{A} \), then \( [a] = a; [\text{self}] \in \mathcal{A} \) returns the name of the current actor.

An actor term represents a set of actors running in parallel, let \( \Gamma \) be the set of all the actor terms, we assume that \( A, B, D, E, F, \ldots \) range over \( \Gamma \). Actor terms are defined by the following abstract syntax.

\[
A ::= a\,(P)^C_j \mid a\,C_j \mid [a, v] \mid A \| A \mid A[f] \\
P ::= \text{become}(C, e).P \mid \text{send}(e_1, e_2).P \mid \text{create}(b, C, e).P \\
\mid e_1 : P + e_2 : P + \ldots + e_n : P \mid \sqrt{\ }
\]

An actor is represented by \( \text{a}(P)^C_j \) where \( a \) is the actor name, \( s \) the state of the actor, \( P \) is the program which must still be executed, and \( C \) the behaviour defining the program that must be executed.
Definition 3.1  - Composable actor terms. Two actor terms $A$ and $B$ are composable with respect to the parallel composition operator $||$, if $\text{Int}(A) \cap \text{Int}(B) = \emptyset$.

Definition 3.2 - Adequate relabelling. A relabelling function $f$ is adequate with respect to an actor term $A$ if $\forall a \in A$, $a(A[f])$ has a unique value.

Proposition 3.3 Given two composable actor terms $A$ and $B$ then:

- $\text{Int}(A \| B) = \text{Int}(A) \cup \text{Int}(B)$
- $\text{New}(A \| B) = \text{New}(A) \cup \text{New}(B)$
- $\text{Ext}(A \| B) = (\text{Ext}(A) \cup \text{Ext}(B)) \setminus ((\text{Int}(B) \cap \text{Ext}(A)) \cup (\text{Int}(A) \cap \text{Ext}(B)))$. 

when the actor will receive a message. When the state is empty it can be omitted. An actor becomes idle and ready to receive messages when $P = \sqrt{1}$; an idle actor is also represented as $C_0$. A behaviour defining actor programs $C(x, y)$ takes two arguments which represent respectively: the contents of the message and the state of the actor. When arguments are not significant they will be omitted. $||$ is the parallel composition operator; only actors with different names can be composed, this notion of composable actors will be defined formally below. $f$ is a relabelling function defined over actor names following the style of [15], for instance the actor term $A[b/a]$ correspond to the actor term $A'$ where all the occurrences of the actor name $a$ are replaced with $b$.

In the case of actors a relabelling functions may be not adequate, because it is not possible to have two actors with the same name in an actor term, the formal definition of adequate relabelling function is given below.

We assume that $P, P', P'', \ldots$ range over programs and $C, C', C'', \ldots$ range over behaviours. A program is a sequence of basic operations or a guarded program. Basic operations are: $\text{send}(e_1, e_2)$ sends the value of the expression $e_2$ to the actor $e_1$, we assume that $\langle e_1 \rangle \in A$; $\text{become}(C, e)$ changes the behaviour and the state of an actor; and $\text{create}(a, C, e)$ binds $a$ to a new actor name and creates a new actor with behaviour $C$ and state $\langle e \rangle$. $e_1 : P + e_2 : P + \ldots + e_n : P$ is a guarded program and $e_1, e_2, \ldots, e_n$ are disjoint conditions on the state of the actor and/or the incoming message, such that there is always only one condition which is true.

The following restrictions hold in the calculus:

- We assume fair message delivery, thus it is guaranteed that a message eventually will reach its destination.
- We assume that all possible sequences of basic operations do not contain more than one $\text{become}$ primitive.
- Recursion is not allowed in programs, thus actors always terminate their internal computation.
- We do not model a restriction operator, thus we assume that all the actor names are known from the outside world.
- The name of actors created dynamically contains an index which is used to distinguish between them and the other actors.

The notation $a(A)$ indicates the actor named $a$ in the actor term $A$, if such an actor does not exist $a(A) = \text{NIL}$. Given an actor term $A$ the set of actor names $A$ is partitioned into three subsets $\text{Ext}(A), \text{Int}(A)$ and $\text{Rest}(A)$ as follows: $\forall a \in A$, if $A' = \text{NIL}$ then $a \in \text{Int}(A)$; else if $a(A) = \text{NIL}$ and $a$ is the recipient of a send primitive of the programs in $A, a \in \text{Ext}(A)$; otherwise $a \in \text{Rest}(A)$. The notation $\text{Int}(a)$ will be an abbreviation of $\text{Int}(\{a\})$. Moreover, we define the set $\text{New}(A) \subseteq \text{Int}A$ including all the actors which have been created dynamically.

Now we can define formally when two actor terms are composable and when a relabelling function is adequate.

Definition 3.1  - Composable actor terms. Two actor terms $A$ and $B$ are composable with respect to the parallel composition operator $||$, if $\text{Int}(A) \cap \text{Int}(B) = \emptyset$.

Definition 3.2 - Adequate relabelling. A relabelling function $f$ is adequate with respect to an actor term $A$ if $\forall a \in A$, $a(A[f])$ has a unique value.

Proposition 3.3 Given two composable actor terms $A$ and $B$ then:

- $\text{Int}(A \| B) = \text{Int}(A) \cup \text{Int}(B)$
- $\text{New}(A \| B) = \text{New}(A) \cup \text{New}(B)$
- $\text{Ext}(A \| B) = (\text{Ext}(A) \cup \text{Ext}(B)) \setminus ((\text{Int}(B) \cap \text{Ext}(A)) \cup (\text{Int}(A) \cap \text{Ext}(B)))$. 

UBLCS-96-9
3.1 Operational Semantics

We model the operational semantics of our actor algebra following the approach of Milner [16] which consists in separating the laws which govern the static relation among actors (for instance $A \parallel B$ is equivalent to $B \parallel A$ from the laws which rules their interaction. This is achieved defining a static structural equivalence relation over syntactic terms and a dynamic reduction relation by means of a labelled transitions system [20].

**Definition 3.4** - **Structural congruence**, is the smallest equivalence relation over actor terms ($\equiv$) defined by:

- $i$ $A \parallel B$ whenever $A$ is $\alpha$-convertible to $B$ with respect to names appearing in New($A$) and New($B$).
- $ii$ $A \parallel B \equiv B \parallel A$
- $iii$ $(A \parallel B) \parallel D \equiv A \parallel (B \parallel D)$
- $iv$ $A \parallel _{a} (\sqrt{\_}) \equiv A$.
- $v$ $\sqrt{A} \equiv \sqrt{\_} \equiv C_{s}$.

Structural congruence states that the parallel composition operator ($\parallel$) is associative and commutative (ii,iii). The actor term $\sqrt{A}$ denotes an actor with an empty behaviour which has terminated its computation, the reason of introducing such an actor term will become clear below.

**Definition 3.5** - **Reduction.** A transition system modelling reduction in the actor algebra is represented by a triple $(\Gamma, T, \{\alpha : \alpha \in T\})$, $\Gamma$ is a set of labels and $\alpha \rightarrow$ is the minimal transition relation satisfying the axioms and rules presented in Table 1.

<table>
<thead>
<tr>
<th>Send</th>
<th>$\alpha($send$(e_{1}, e_{2}), P)<em>{C} \equiv</em>{scon} d_{(1,2)}$</th>
<th>$\alpha(P)<em>{C} \equiv (\llbracket e</em>{1}\rrbracket, \llbracket e_{2}\rrbracket)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bec</td>
<td>$\alpha($become$(C', e), P)<em>{C} \equiv</em>{become(a)}$</td>
<td>$\alpha(P)<em>{C} \equiv \alpha(C'</em>{[_]}$</td>
</tr>
<tr>
<td>Create</td>
<td>$\alpha($create$(b, C', e), P)<em>{C} \equiv</em>{create(d_{i})}$</td>
<td>$\alpha(P[d/b]C')<em>{C} \equiv d</em>{i}C'<em>{[_]}$ where $d</em>{i}$ is a new name</td>
</tr>
<tr>
<td>Rec</td>
<td>$\alpha C_{s} \equiv [a, v]_{\sqrt{_}}$</td>
<td>$\alpha(P[v/x, s/y])<em>{C} \equiv C'</em>{[_]}$ where $C(x,y) \Rightarrow P$</td>
</tr>
<tr>
<td>Par</td>
<td>$A \rightarrow A'$</td>
<td>$A \rightarrow A'$</td>
</tr>
<tr>
<td></td>
<td>$A \parallel B \rightarrow A' \parallel B$</td>
<td>$A[f] \Rightarrow A'[f]$</td>
</tr>
<tr>
<td>Sum</td>
<td>$\llbracket e_{i} \rrbracket = \text{true}$</td>
<td>$\alpha$</td>
</tr>
<tr>
<td></td>
<td>$\alpha(e_{1} : P_{1} + e_{2} : P_{2} + \ldots + e_{n} : P_{n})<em>{C} \equiv</em>{sum(a)} (P_{i})_{C}$</td>
<td></td>
</tr>
<tr>
<td>Con</td>
<td>$B \equiv A$</td>
<td>$A \rightarrow A'$ $A' \equiv B'$</td>
</tr>
<tr>
<td></td>
<td>$B \rightarrow A'$ $B \equiv B'$</td>
<td></td>
</tr>
</tbody>
</table>

**Table 1. Reduction Relation of AA.**

The effect of a send primitive (rule *Send*) is to create an actor term representing a message. The rule *Bec* modifies the behaviour of an actor, a new actor with the same name is created and the rest of the computation is executed by an actor with an empty behaviour, which will be destroyed when terminates its computation (see the structural equivalence). The rule *Create* binds $b$ to a new actor $d_{i}$ and creates a new idle actor with name $d_{i}$, behaviour $C'$ and state $\llbracket e \rrbracket$. The implicit receive (rule *Rec*) is performed only when an actor $a$ is idle (*i.e.*, when an actor ends its computation).
and there is a message which has \( a \) as a target. The rules \( Par \) describes the behaviour of the parallel composition operator and the rule \( Sum \) allows to infer guarded commands. Rule \( Rel \) is the equivalent of the relabelling rule of CCS [15]: \( f(\alpha) \) denotes the application of the relabelling function to the label \( \alpha \).

3.2 Discussion

There are several differences with respect to the formal semantics of actors in [2] which is worth to point out.

- 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-related aspects and not deal with issues concerning the sequential execution of programs inside actors, as an example: termination.
- We have introduced a guarded command as an alternative to the conditional which is present in previous formalization of actors [2].
- We provide a 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. Our choice allows us to model easily and in a rigorous way the fact that the a behaviour (an actor program) depends on the state of an actor (See the rule \( Rec \) in 1).
- In the actor algebra 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.
- Finally, we define a new notion of observation equivalence which is finer with respect to those made in previous formalizations of actors such as [2, 3], this will be discussed in details in Section 4.

In summary, we believe that our approach is complementary to previous approaches to the semantics of actors, providing a new framework to discuss concurrency-related aspects.

3.3 An Example: The Contract Net Protocol

We illustrate the expressive power of the language by modelling the Contract Net Protocol [21], a protocol which allows an agent to distribute tasks among a set of agents by means of negotiation. We only model a restricted version of the protocol where a single manager object sends task announcements to a set of workers which evaluate them, bidding only on those of interest. The manager evaluates bids to select the most appropriate worker which will execute the task.

We associate an actor to each agent in the contract net, thus the set of actor names is \( A = \{\text{man}, w_1, \ldots, w_n\} \). We also suppose to have three fixed set of values \( \text{Bids} \cup \text{Tasks} \cup \{\text{reject}, \text{start}\} \subseteq \mathcal{V} \) representing possible tasks, bids and protocol keywords, where \( \text{Bids} = \{\text{bid}_1, \ldots, \text{bid}_b\} \) and \( \text{Tasks} = \{\text{task}_1, \ldots, \text{task}_t\} \). A contract net can be defined by the following actor term:

\[
\text{man} \triangleleft \text{Do} \parallel (w_1 \text{WaitTask}) \ldots \parallel (w_n \text{WaitTask})
\]

where \( \text{Do}(\alpha) \) and \( \text{WaitTask}(\alpha) \) define respectively the behaviours of the manager agent and of the workers as in Figure 1. The manager object sends the workers the task to be executed; then it executes \( \text{become}(\text{WaitBid}, (0, \text{NIL}, \text{task})) \) and it starts waiting for bids. The state of the waiting manager is stored in a tuple which contains: the number of workers which have already sent a bid, the list of received bids and the task to perform. The second condition of the \( \text{Do}(\alpha) \) constant expresses a synchronization constraint i.e., if the actor receive a message different from start, it send it again to the same address. The fair message delivery assumption in the actor model guarantees that the right message will be eventually executed. We assume a set of functions to manage a stack: \( \text{push}, \text{1st}, \text{rest}, \text{empty} \); two functions which operate on bids: \( \text{best}_\text{bid} \) and \( \text{other}_\text{bids} \); and a function \( \text{execute} \) which executes tasks. The symbol \( \text{self} \) indicates the actor which perform the send primitive.
Manager Agent:

\[
D_0(x) \overset{def}{=} (x = \text{start}): \text{send}(d_1, \text{task}), \ldots, \text{send}(d_n, \text{task})..
\]

\[
\text{becomes}(\text{WaitBid}, (0, \text{NIL}, \text{task})), \\
(x \neq \text{start}): \text{send}(\text{self}, x).
\]

\[
\text{WaitBid}(x, y) \overset{def}{=} \\
[2n \times d(x) \in \text{Bids} \land 1st(y) \neq n - 1]: \text{becomes}(\text{WaitBid}, (1st(y) + 1, \text{push}(x, 2n \times d(y)), \text{3rd}(y))), \\
[2n \times d(x) \in \text{Bids} \land 1st(y) = n - 1]: \text{send}(\text{bestBid}, \text{push}(x, \text{2nd}(d(y)), \text{task})), \\
\text{becomes}((\text{Broadcast}, \text{otherBids}(\text{push}(x, \text{2nd}(y)))), \\
\text{send}(\text{self}, \text{start})), \\
(2n \times d(x) \notin \text{Bids}): \\
\text{send}(\text{self}, x).
\]

\[
\text{Broadcast}(x, y) \overset{def}{=} \\
x = \text{start} \land \text{empty}(y)): \text{becomes}(D_0, x), \\
(x \neq \text{start} \land \neg \text{empty}(y)): \text{send}(d_1(y), \text{reject}), \\
(x \neq \text{start}): \text{send}(\text{self}, \text{start}).
\]

Worker Agents:

\[
\text{WaitTask}(x) \overset{def}{=} \\
x \in \text{Tasks}: \text{send}(a, (\text{self}, \text{bid}_{i, j})), \text{becomes}(\text{WaitAnswer}), \\
x \notin \text{Tasks}: \text{send}(\text{self}, x).
\]

\[
\text{WaitAnswer}(x) \overset{def}{=} \\
x \in \text{Tasks}: \text{send}(d, \text{execute}(x)), \text{becomes}(\text{WaitTask}), \\
x \notin \text{Tasks} \land \neg \text{reject} \land (x \notin \text{Tasks}): \text{send}(\text{self}, x).
\]

Figure 1. The Contract Net Protocol

4 Equivalence of Actor Terms

Intuitively, two actor terms are equivalent if they cannot be distinguished by an external actor term interacting with each of them. Given an actor term \(A\) two types of events are observed: when an actor \(a \in \text{Int}(A)\) receives a message from an actor in \(b \in \text{Ext}(A)\), and when an actor in \(a \in \text{Int}(A)\) sends a message to an actor in \(b \in \text{Ext}(A)\). Since for actors there is arrival-order non-determinism in message delivery, the equivalence does not depend on the order in which send operations are performed. To illustrate these notions we consider an instance of the contract net protocol presented in the previous Sect. 3.3.

Example 4.1 A contract net including two workers can be defined by the following actor term:

\[
A \overset{man}{=} D_0 \mid w_1 \text{WaitTask} \mid w_2 \text{WaitTask}
\]

where the behaviour of the manager is:

\[
D_0(x) \overset{def}{=} (x = \text{start}): \text{send}(w_1, \text{task}), \text{send}(w_2, \text{task}). \\
\text{becomes}(\text{WaitBid}, (0, \text{NIL}, \text{task})), \\
(x \neq \text{start}): \text{send}(\text{self}, x).
\]

If we give an alternative definition of behavior of the manager, as follows:

\[
D_0'(x) \overset{def}{=} (x = \text{start}): \text{send}(w_2, \text{task}), \text{send}(w_1, \text{task}). \\
\text{becomes}(\text{WaitBid}, (0, \text{NIL}, \text{task})), \\
(x \neq \text{start}): \text{send}(\text{self}, x).
\]

we can define a new actor term \(B \overset{man}{=} D_0' \mid w_1 \text{WaitTask} \mid w_2 \text{WaitTask}\) representing a similar contract net.

The actor terms \(A\) and \(B\) cannot be distinguished by an external actor term interacting with them. In fact, whenever the external actor sends \(\text{man} \ a \ \text{start} \) message, \(\text{man} \) will send back two messages: one to an actor named \(w_1\), and another to an actor named \(w_2\). The order in which the send operations are performed is not relevant because the actor model does not guarantee ordered message delivery. As a

3. Since here we are considering only the manager actor \(w_1, w_2 \in \text{Ext}^{(\text{man}D_0)}\).
including all the observable send operations performed.

Definition 4.2 Strong Bisimulation.

A binary relation \( S \subseteq \Gamma \times \Gamma \) over actor terms is a strong bisimulation if \((A, B) \in S\) implies, for all \(B \in T\),

---

4. Remember that \(Int(a) = \{a\}\).
Table 3. The Transition Relation for Bisimulation.

1. Whenever $A \xrightarrow{\beta} A'$ then, $\exists B', B \xrightarrow{\beta} B'$ and $(A', B') \in S$.
2. Whenever $B \xrightarrow{\beta} B'$ then, $\exists A', A \xrightarrow{\beta} A'$ and $(A', B') \in S$.

Definition 4.3 Observation equivalence - $A$ and $B$ are observation equivalent ($A \equiv B$) if $(A, B) \in S$ for some strong bisimulation $S$.

Proposition 4.4 Given two actor terms $A$ and $B$ such that $A \equiv B$ then $\text{Int}(A) = \text{Int}(B)$ and $\text{Ext}(A) = \text{Ext}(B)$.

This proposition is a direct consequence of the definition of equivalence given above. Intuitively, if two actor terms are observation equivalent, they contain the same set of actor names. Note that the inverse of the above proposition does not hold. Two actor terms may be defined on the same sets of actor names but they result different with respect to an external observer.

4.1 Comparison with previous work

Our observations are different from those made in previous formalizations of actors such as [2, 3] and other asynchronous calculus such as [13], where two extra axioms are introduced to model the interaction with the outside world. For instance, the out and in rules defined in [3] specify when an external actor receives a message from the bag of incoming messages and when an external actor sends a message into the bag. In the case of our algebra these observations can be defined adding the two axioms defined in table 4 to the transition system in defined 1, and setting all the other labels to $\tau$.

Table 4. Interaction Equivalence.

The advantage of our approach is that the resulting equivalence finer because is more related to the structure of an actor rather then being mediated by the multiset of incoming messages. We illustrate this point with the following example.

Example 4.5 We consider two actors implementing two different communication media: a queue and an ether i.e., an unordered set (mailbox) of messages [15]. The programs of the two actors are defined as
follows:

\[
\text{QUEUE}(x, y) \overset{d}{=} \\
(1st(x) = get \land \text{empt}(y)) : \quad \text{send}(\text{sel} f, x), \sqrt{+} \\
(1st(x) = get \land \neg\text{empt}(y)) : \quad \text{become(QUEUE, rest(y))} \\
\hspace{1cm} \quad \text{send}(3rd(x), 1st(y)), \sqrt{+} \\
(1st(x) = put) : \quad \text{become(QUEUE, insert\_last(2nd(x), y))}, \sqrt{+}
\]

\[
\text{ETHER}(x, y) \overset{d}{=} \\
(1st(x) = get \land \text{empt}(y)) : \quad \text{send}(\text{sel} f, x), \sqrt{+} \\
(1st(x) = get \land \neg\text{empt}(y)) : \quad \text{become(ETHER, rest(y))} \\
\hspace{1cm} \quad \text{send}(3rd(x), 1st(y)), \sqrt{+} \\
(1st(x) = put) : \quad \text{become(ETHER, insert\_random(2nd(x), y))}, \sqrt{+}
\]

The two actor programs assume messages with the following structure
\((\text{operation, message, sender})\) where \text{operation} \in \{\text{put, get}\}. We also assume standard operators on sequences: \(1st, 2nd, 3rd, \text{rest}, \text{empty}\). The two programs only differs for the functions insert\_last and insert\_random, the first inserts a message at the end of a sequence and the second in a random position.

We now observe the computations of two actors \("\text{ETHER}\) and \("\text{QUEUE}\). If we consider the interaction equivalence presented in table \ref{tab:actor-eq} we cannot distinguish between the two actors because there is no relation between the order in which the messages are inserted in the multiset and the order in which the messages are received from the actor \(a\). In other words, when an external actor sends two \text{put} messages in the multiset of incoming messages: \(m_1\) and \(m_2\), we cannot observe if \(m_1\) is received before \(m_2\) or vice versa. Thus is not possible to distinguish the behaviour of an \text{ether} and a \text{queue} because we can not establish when the actors really receives messages.

On the other hand, in our approach the two actors are not equivalent with respect to our definition of observations \("\text{ETHER} \not\equiv \text{QUEUE}\). This because we observe when the actor \(a\) receives a message, and thus we are able to compare the rest of the computation with the expected behaviour of the actor. As an example, suppose that actor \(a\) receives two \text{put} messages: first \(m_1\) and then \(m_2\). When it receive a \text{get} message \(m_3\), the contents of the answer message sent by \(a\) depends on the state and the behaviour of the actor which depends from the insert function used.

4.2 Congruence Result

Given the definition of observation equivalence, we would like to prove that the given equivalence is a congruence with respect to the parallel composition operator \(\parallel\). Thus, if two actor systems are proved to be equivalent also their compositions with any other composable actor system are equivalent. As an example, if we consider the actor terms \(A\) and \(B\) defined in the example 4.1, we would like to prove that if \(\text{man} D \overset{\text{man}}{\approx} \text{man} D'\) then \(A \overset{\text{man}}{\approx} B\).

Theorem 4.6 Congruence. The equivalence relation \(\overset{\text{man}}{\approx}\) over actor terms is a congruence with respect to the parallel composition operator.

Proof 4.7 We have to prove that if \((A \overset{\text{man}}{\approx} B)\) then \(\forall D \in \Gamma\) such that \(D\) is composable with both \(A\) and \(B\), we have \(A || D \overset{\text{man}}{\approx} B || D\). The main difficult arises when \(\text{Ext}(D) \cap \text{Int}(A) \neq \emptyset\) or vice versa when \(\text{Ext}(A) \cap \text{Int}(D) \neq \emptyset\); we give a sketch of the proof in this case. In all the other cases the actor terms proceed in parallel without interfering one another, thus, it is always possible to build a bisimulation.

Let \(E = A || D\) and \(F = B || D\), from the Propositions 3.3 and 4.4 we can prove that \(\text{Int}(A) = \text{Int}(B)\) and \(\text{Ext}(A) = \text{Ext}(B)\) and thus \(\text{Int}(E) = \text{Int}(F)\) and \(\text{Ext}(E) = \text{Ext}(F)\). Moreover, we have that \(\text{Ext}(A) \cap \text{Int}(D) = \text{Ext}(B) \cap \text{Int}(D)\) and \(\text{Ext}(D) \cap \text{Int}(A) = \text{Ext}(D) \cap \text{Int}(B)\). Suppose that \(\text{Ext}(A) \cap \text{Int}(D) \neq \emptyset\), we show that it is always possible to build a bisimulation.

Suppose that \(E \xrightarrow{\text{man}} E'\) (vice versa if \(F \xrightarrow{\text{man}} F'\) we can proceed in the same way):

- Case 1: \(a = \text{receive}(a, v)\) where \(a \in \text{Int}(E)\).
\[ \text{If } a \in \text{Int}(A) \text{ then } A \xrightarrow{\alpha} A', \text{ thus } B \xrightarrow{\alpha} B' \text{ with } A' \cong B'. \text{ This implies that } B \parallel D \xrightarrow{\alpha} B' \parallel D, \text{ thus we can take } F' = B' \parallel D \text{ and } F \xrightarrow{\alpha} F'. \]
\[ \text{If } a \in \text{Int}(D) \text{ then } D \xrightarrow{\alpha} D', \text{ thus we can take } F' = B \parallel D' \text{ as above.} \]
\[ \text{Case } 2: \alpha = \text{send}(a^1, a^1) \ldots \text{send}(a^n, v^n) \text{ where } a^i \in \text{Int}(A) \lor a^i \in \text{Int}(D). \text{ Since } \text{Ext}(F) = \text{Ext}(E) \text{ and } \text{Int}(B) = \text{Int}(A), \text{ we are always able to obtain the same set of observable actions in } F. \]

5 Conclusion

We have presented a study of the interaction properties of concurrent object-oriented systems with the aim to find out a set of basic mechanisms to model interaction. We have defined an algebra of actors that captures the highlighted basic interaction primitives in a static framework. We believe that this algebra can be assumed as basis for modelling static concurrent object-oriented programming systems and 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.

Acknowledgements: The authors would like to acknowledge discussions with Gul Agha, Angela Dalmonte, Roberto Gorrieri, Cosimo Laneve.

References


