A Process Algebraic Specification of the New Asynchronous CORBA Messaging Service*

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. CORBA (The Common Object Request Broker Architecture) has to continually evolve in order to cope with the changes of requirement of applications which become larger and more distributed. For this reason new features are being added to the CORBA specification, for instance the last proposal for a revised CORBA Messaging Service includes two new asynchronous models of request invocation. Since these new features will be added in the next CORBA implementations a relevant issue is to study their operational behaviour from different perspectives in order to facilitate the task of implementors. This paper addresses this issue providing an analysis of the CORBA Messaging Service which includes the new asynchronous features. In particular we illustrate how CORBA models for request invocation can be mapped into a message passing architecture based on the actor model. For this purpose we exploit an algebra of actors which supports some of the main features of the abstract Object Model of the Object Management Group, such as object identity and an explicit notion of state. This approach allows us to discuss and compare different models of request invocation in a standard process algebraic perspective for instance we show how different notions of equivalence, such as standard and asynchronous bisimulation, can be adapted to reason about CORBA.

1 Introduction

CORBA (The Common Object Request Broker Architecture) is the object-oriented standard for integrating applications running in heterogeneous distributed environments developed by the Object Management Group (OMG). One of the main limitations in the CORBA specification concerning the ability to cope with large scale distributed systems has been the lack of support for asynchronous models of request invocation. This hole has been recognized by OMG in the Messaging Service RFP [13] and in the next version of CORBA [32] new features will be provided in order to overcome this limitation. The proposal for a revised CORBA messaging service was submitted one year ago by a group of companies which are members of the OMG consortium [14]. This new proposal

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

extends the synchronous models of request invocation which were supported in CORBA 2.2 [16] with two new asynchronous models: callback and polling.

Since these new features will be added in the next CORBA implementations a relevant issue is to study their operational behaviour from different perspectives in order to facilitate the task of implementors. This paper addresses this issue providing an analysis of the CORBA Messaging Service which includes the new asynchronous features. In particular we illustrate how CORBA models for request invocation can be mapped into a message passing architecture based on the actor model. For this purpose we exploit an algebra of actors [10, 11, 9] which supports some of the main features of the abstract Object Model of the Object Management Group, such as object identity and an explicit notion of state.

This approach allows us to discuss and compare different models of request invocation in a standard process algebraic perspective where a nice, easy-to-understand specification style is supported. Thus we can reuse standard results of the theory of concurrency for reasoning on CORBA models of request invocation. For instance we show how different notions of equivalence, such as standard and asynchronous bisimulation, can be adapted to reason about CORBA.

The paper is organized as follows. Section 2 briefly describes the CORBA messaging facilities and the underlying object model. In Section 3 we introduce our algebra of actors. In Section 4 we specify the CORBA models for request invocation in the algebra of actors and we introduce a framework for reasoning on CORBA requests. In Section 5 we illustrate our approach for reasoning on a simple CORBA program. We conclude the paper by discussing related works and drawing conclusions.

2 The New CORBA Messaging Service

CORBA has been designed to support the integration of a wide variety of object systems providing solutions to the many technical problems that arise in this context. For this reason the CORBA architecture is quite complex and includes several components to deal with open systems and interoperability, for instance, the Interface Definition Language (IDL), the Dynamic Invocation Interface (DII), and, the Object Adapters [32].

CORBA clients can make requests exploiting an OMG IDL typed stub (a procedure which executes a single operation - the request - depending on the interface of the target object) or the dynamic invocation interface (an interface which is independent from the target object interface). Requests are sent through the Objects Request Broker (ORB) which is responsible for all of the mechanisms needed for transmitting them to the right destination. In particular the Object Adapter mediates between CORBA objects and the programming language used for the server implementation.

Although the CORBA architecture deals with all the interoperability issues presented above, a client exploits an interface which is completely independent concerning these details. Clients making their requests have only to consider the address of the server, the interface of the server (the operation supported by the server object) and the selection of an adequate protocol for request invocation. Moreover the semantics of the dynamic or the stub interface for invoking a request is the same and the server receiving a message cannot tell how the request was invoked.

All these considerations suggest that a more abstract model can be used for reasoning on CORBA messaging facilities where all the low level aspects are hidden. This is the role of the CORBA concrete object model [16] based on the abstract OMG object model [12]. The abstract OMG object model defines basic concepts such as objects, operations, types and interfaces and it is common to all the OMG technologies. The CORBA concrete object model is summarized below.

2.1 The CORBA Object Model

The concrete object model of CORBA [16] distinguishes between clients (requestors of services) and servers (providers of services). In the following we adopt this concrete model for reasoning on messaging facilities of CORBA with the main restriction of considering an untyped world. This implies that we will not deal with subtyping and inheritance issues but we will only concentrate on dynamic issues concerning the execution of requests.

We summarize here some of the main aspects of this model considering an untyped scenario where we also abstract away from the concepts of exceptions and request context. The reader interested in complete specification of the CORBA concrete object model can refer to [12].

- Each object has an identifier (its name) that provides a means to refer to the object and it is used for dispatching requests. The name (identity) is immutable and persists as long as the object exists.
- Clients (the requestors of services) are conceptually distinguished from servers (the providers of services).
- Server objects have a local memory and a behaviour that defines the meaningful operations. They are encapsulated entities accessible through a set of operations: the object interface.
- Clients send requests to server objects.
- Requests include an operation, a target object, a request identifier and zero or more parameters. One possible outcome for a request is returning to the client the result.
- When a client issues a request a method of the target object is called. A method specifies how a server executes the requested service. The input parameters are those of the request and the result is sent back to the requestor. The execution of a method may change the state of a server.
- From a client's point of view objects can only be created and destroyed as an outcome of issuing requests.

2.2 CORBA Models of Request Invocation

According to the new proposal for the CORBA Messaging Service, clients perform requests exploiting synchronous or asynchronous operations. CORBA requests are objects that include all the information that should be transmitted to the server. In the CORBA DII requests must be created explicitly before being transmitted to the server object. In the following we do not consider all the details concerning the creation of new requests and we provide an abstract representation of them. We represent requests as records and we extract some of the most significant parameters, making them directly available in the operation.

Usually a request contains additional flags specifying different semantics for the associated operation. For instance a flag specifies if an operation is blocking or not blocking. For the limited scope of this paper we will not consider all the possible combinations of requests and flags and we will restrict our study to the operational behaviour described below. We will use a notation based on the Dynamic Invocation Interface which is more adequate for our operational purposes.

Models of Request Invocation of CORBA 2.2 The models of request invocation of the current version of CORBA are:

- **Synchronous:** *INVOKE(Server,Request,Result)*
  A client performing this request waits until the server sends the answer. The result is placed in the *Result* argument. This is similar to a remote procedure call.
- **Oneway:** *ONEWAY(Server,Request)*
  This invocation returns the control to the caller object, the request is transmitted and there is no response.
- **Deferred Synchronous:** *SEND(Server,Request)*
  This invocation transmits the request to the server and returns the control to the caller object without waiting for the operation to finish. To get the result the calling object can perform the *get_response* operation having the same request identifier as an argument.
  GET_RESPONSE(Server,Request,Result) this operation gets the first answer of the request, if it is ready otherwise it blocks waiting for the result. The result is placed in the *Result* argument. CORBA also includes a GET_NEXT_RESPONSE primitive to retrieve other possible results; this feature will not be modelled in this paper.

New Asynchronous Models of Request Invocation The proposal for a new CORBA messaging facility extends CORBA 2.2 with two new asynchronous models of request invocation.

- **Callback** *SENDC(Server,Request,Callback)*
  When a client invokes this operation it sends the request to a server object and it creates a reply handler object. The answer will be processed by this object and *Callback* represents its behaviour.

– **Polling** *SENDP(Server,Request,Poller)*
When a client invokes this operation the address of a local poller object is returned in *Poller*. This object will be used to retrieve the answer exploiting the *POLL_GET_RESPONSE* operation. A poller object is also able to execute an *IS_READY* operation which returns true if and only if the answer is available in the poller object.

2.3 A Process Algebraic Framework for CORBA

We provide a formal framework for all the above models of request invocation following a process algebraic approach. Process algebras, like CCS [22] and CSP [18], 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. Note that the CORBA callback and polling models of request invocation need a more dynamic structure of channels.

The π-calculus [25] can be considered as 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, in [19, 7] also an asynchronous fragment of the π–calculus has been studied in order to analyze also the asynchronous communication mechanism and its similarities/differences with the synchronous one [27].

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 [23, 33, 28], but these approaches do not provide the concepts of state of a process or of object identity as a first class entities.

On the other hand, the actor model [17, 2] directly deals with many features of the OMG Core Object Model such as object identity, state and operations associated to objects that characterize their behaviour; an actor has the same structural and interaction properties as a CORBA object if we restrict to an untyped world.

For this reason we exploit a new formalism for our modelling purposes, an algebra of actors [11, 10, 9], which has been designed as a compromise between the standard process algebras, such as the π-calculus [24], and the actor model. This process algebra supports an high level specification style which allows the user to describe how CORBA models of request invocation can be mapped into sequences of message-passing primitives.

2.4 The Actor Model

The actor model was introduced by Carl Hewitt about 20 years ago [17]. 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 [20], a property of an object which distinguishes this object from all the others. Object identity is a typical feature of object-oriented programming languages and it is used as a basic dispatching mechanism in message passing. This property is not easily embeddable in formalisms such as CCS [22] (or asynchronous π-calculus [19, 7]), 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].

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 (finite) power set $\mathcal{P}_{fin}(\mathcal{A})$ (*i.e.,* L, L', $L'' \subseteq_{fin} \mathcal{A}$). Let $\mathcal{V}$ be a set of values (with $\mathcal{A} \subset \mathcal{V}$) containing, e.g., *true*, *false*, and let $\mathcal{X}$, ranged over by x, y, z,..., be a set of value variables that are bound to values at run-time. 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 examples we will use standard operators on sequences: $1st$, $2nd$, *rest*, *empty*. We will denote values by 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* identifiers: C, C', ... will range over $\mathcal{C}$. We suppose that every identifier C is equipped with a corresponding behaviour definition $C \stackrel{def}{=} P$ 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 recursive behaviours to be defined, for example we could have $C \stackrel{def}{=} become(C, state).\surd$.

Actor terms are defined by the following abstract syntax:

$$A ::= {}^{\mathbf{a}}\mathbf{C}_{\mathbf{s}} \mid {}^{\mathbf{a}}[\mathbf{P}]_{\mathbf{s}} \mid \langle \mathbf{a}, \mathbf{v} \rangle \mid \mathbf{A}|\mathbf{A} \mid \mathbf{A}\backslash\mathbf{a} \mid \mathbf{0}$$

An actor can be idle or active. An idle actor ${}^{\mathbf{a}}\mathbf{C}_{\mathbf{s}}$ (composed by a behaviour $\mathbf{C}$, a name $\mathbf{a}$, and a state $\mathbf{s}$) is ready to receive a message. When a message is received the actor becomes active. Active actors are denoted by ${}^{\mathbf{a}}[\mathbf{P}]_{\mathbf{s}}$ where $\mathbf{P}$ is the program that is executed. The actor $\mathbf{a}$ will not receive new messages until it becomes idle (by performing a **become** primitive). Sometimes the state $\mathbf{s}$ is omitted when empty (i.e. $\mathbf{s} = \emptyset$). A program $\mathbf{P}$ is a sequence of actor primitives (**become**, **send** and **create**) and guarded choices $\mathbf{e}_1{:}\mathbf{P}_1 + \ldots + \mathbf{e_n}{:}\mathbf{P_n}$ terminating in the null program $\surd$ (which is usually omitted). An actor term is the parallel composition of (active and idle) actors and messages. A message is denoted by a term $\langle \mathbf{a}, \mathbf{v} \rangle$ where $\mathbf{v}$ is the contents and $\mathbf{a}$ the name of the actor the message is sent to. Also a restriction operator $\mathbf{A}\backslash\mathbf{a}$ is used in order to allow the definition of local actor names ($\mathbf{A}\backslash\mathbf{L}$ is used as a shorthand for $\mathbf{A}\backslash\mathbf{a}_1\backslash\ldots\backslash\mathbf{a_n}$ if $\mathbf{L} = \{\mathbf{a}_1, \ldots, \mathbf{a_n}\}$) while $\mathbf{0}$ is the usual empty term.

The actor primitives and the guarded choice are described below.

- **send**:
  The program $\mathbf{send}(\mathbf{e}_1, \mathbf{e}_2).\mathbf{P}$ sends a message with contents $\mathbf{e}_2$ to the actor indicated by $\mathbf{e}_1$:

  $${}^{\mathbf{a}}[\mathbf{send}(\mathbf{e}_1, \mathbf{e}_2).\mathbf{P}]_{\mathbf{s}} \xrightarrow{\tau} {}^{\mathbf{a}}[\mathbf{P}]_{\mathbf{s}} \mid \langle [\![\mathbf{e}_1]\!]^{\mathbf{a}}_{\mathbf{s}}, [\![\mathbf{e}_2]\!]^{\mathbf{a}}_{\mathbf{s}} \rangle$$

  where τ represents an internal invisible step of computation.

- **become**:
  The program $\mathbf{become}(\mathbf{C}, \mathbf{e}).\mathbf{P}'$ changes the state of the actual actor from active to idle:

  $${}^{\mathbf{a}}[\mathbf{become}(\mathbf{C}, \mathbf{e}).\mathbf{P}']_{\mathbf{s}} \xrightarrow{\tau} ({}^{\mathbf{d}}[\mathbf{P}'\{\mathbf{a}/\mathbf{self}\}]_{\mathbf{s}})\backslash\mathbf{d} \mid {}^{\mathbf{a}}\mathbf{C}_{[\![\mathbf{e}]\!]^{\mathbf{a}}_{\mathbf{s}}} \qquad \text{with } \mathbf{d} \text{ fresh}$$

  The primitive **become** is the only one that permits to change the state according to the expression $\mathbf{e}$; we sometimes omit $\mathbf{e}$ if the state is left unchanged (i.e. $\mathbf{e} = \mathbf{state}$). The continuation $\mathbf{P}'$ is executed by the new actor ${}^{\mathbf{d}}[\mathbf{P}'\{\mathbf{a}/\mathbf{self}\}]_{\mathbf{s}}$. This actor will never receive other messages (i.e. it is unreachable) as its name $\mathbf{d}$ cannot be known to any other actor. Indeed, the expression **self**, which is the only one that returns the value $\mathbf{d}$, is changed in order to refer to the name $\mathbf{a}$ of the initial actor.

- **create**:
  The program $\mathbf{create}(\mathbf{b}, \mathbf{C}, \mathbf{e}).\mathbf{P}'$ creates a new idle actor having state $\mathbf{s}$ and behaviour $\mathbf{C}$:

$$\mathbf{{}^{a}[create(b, C, e).P']_s} \xrightarrow{\tau} (\mathbf{{}^{a}[P'\{d/b\}]_s} \mid \mathbf{{}^{d}C_{[\![e]\!]^{a}_{s}}})\backslash\mathbf{d} \qquad \text{with } \mathbf{d} \text{ fresh}$$

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.

– $\mathbf{e_1{:}P_1 + \ldots + e_n{:}P_n}$:
In the agent $\mathbf{e_1{:}P_1 + \ldots + e_n{:}P_n}$, the expressions $\mathbf{e_i}$ are supposed to be boolean expressions with value **true** or **false**. The branch $\mathbf{P_i}$ can be chosen only if the value of the corresponding expression $\mathbf{e_i}$ is **true**:

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

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 [23] which consists in separating the laws which govern the static relation among actors (for instance $\mathbf{A|B}$ is equivalent to $\mathbf{B|A}$) from the laws which rules their interaction. This is achieved by defining a static structural equivalence relation over syntactic terms and a dynamic relation by means of a labelled transition system [29].

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

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

Definition 2. - **Computations.** *A transition system modelling computations in the actor algebra is represented by the triple* $(\Gamma, T, \{\stackrel{\alpha}{\rightarrow} \mid \alpha \in T\})$. $T = \{\tau\} \cup \{av, \overline{av}L \mid a \in \mathcal{A},\ v \in \mathcal{V},\ L \subseteq_{fin} \mathcal{A}\}$ *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.* $\stackrel{\alpha}{\rightarrow}$ *is the minimal transition relation satisfying the axioms and rules presented in Table 1.*

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

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 labelled with av can be performed by the actor a when it is idle (rule *Receive*). Note here the use of the expression *message* which is replaced by the content of the incoming message v in the program P. 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 (see side condition $a \notin act(B)$).

The fact that an output label can be performed by an agent only if it does not contain the target actor introduces in our calculus an interesting strong form of *unique receptiveness* property. Indeed, the operational semantics not only forces that a message can be consumed uniquely by its receptor, but also ensures that external actors are neither able to observe the presence of a pending message sent to another actor. We will show in the remainder of the paper nice features of our calculus related to this property.

3.1 Equivalence of Actor Terms

As already stated, one of the advantages of having introduced a semantics for actors based on a labelled 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* [22] (only bisimulation in the following) and the *asynchronous weak bisimulation* [19, 5] (only asynchronous bisimulation in the following) which is the corresponding equivalence for languages based on asynchronous communication.

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

$$P \stackrel{\tau}{\Longrightarrow} P' \text{ iff } P(\stackrel{\tau}{\longrightarrow})^\star P'$$
$$P \stackrel{\alpha}{\Longrightarrow} P' \text{ iff exists } P'' \text{ and } P''' \text{ s.t. } P \stackrel{\tau}{\Longrightarrow} P'' \stackrel{\alpha}{\longrightarrow} P''' \stackrel{\tau}{\Longrightarrow} P' \quad (\text{for } \alpha \neq \tau)$$

Observe that given $P \stackrel{\tau}{\Longrightarrow} P'$ P' can be simply P if no τ transition is performed.

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 $A \stackrel{\alpha}{\longrightarrow} A'$ then there exists B' such that $B \stackrel{\alpha}{\Longrightarrow} B'$ 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 $(A, B) \in \mathcal{R}$.*

Observe that we are dealing with the standard *weak* bisimulation in which a τ step can be simulated by simply making no action. This kind of bisimulation is not a congruence in the presence of a general choice composition operator; in our case, the unique composition operators for actor terms are the parallel $A|A'$ and the restriction $A \setminus a$ operators, and it is not difficult to see that the bisimulation equivalence we have defined is a congruence w.r.t. these operators (*i.e.*, 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. 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:

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

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

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

where the behaviour *Forward* is:

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

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

In order to make the actor ${}^{b}Double$ unreachable from the outside, we can think to introduce the following restriction on the actor name b:

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

As the action bv is no more observable, we have that $B' \approx A$. In order to prove this equivalence, it is enough to see that the forward operation is composed of only unobservable τ labelled steps.

For languages based on asynchronous communication a new notion of *asynchronous bisimulation* has been introduced in [19] and formally analyzed in [5]. 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. This difference reflects the idea that an observer cannot synchronise with the observed system but can only send messages and look at what may come out of it.

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 $B \stackrel{\alpha}{\Longrightarrow} B'$ and $(A', B') \in \mathcal{R}$.*
- *if $A \xrightarrow{av} A'$ then there exists B' such that $B \stackrel{av}{\Longrightarrow} B'$ and $(A', B') \in \mathcal{R}$ or $B \stackrel{\tau}{\Longrightarrow} B'$ 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, it is not difficult to prove that also the asynchronous bisimulation is a congruence.

It is clear from the definition that the asynchronous bisimulation is coarser with respect to the standard bisimulation, *i.e.,* if $A \approx B$ then also $A \approx_a B$. For example, the equivalences described in the example 1 hold also if we take into account $\approx_a$ instead of $\approx$.

The following example shows the need to move to the asynchronous bisimulation in order to formally analyze interesting aspects of the actor model.

Example 2. We consider two actors implementing two different communication medias: a queue and an ether, *i.e.,* an unordered set (mailbox) of messages [22]. The behaviours of the two actors are defined as follows:

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

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

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.

The property of arrival-order non-determinism in message delivery holding in the actor model, suggests that there is no way to distinguish between the behaviour of the $ETHER$ and the $QUEUE$. Indeed, an external observer has no way to force a particular order of reception of the data to introduce in the buffer, so we can always consider that data are received by the $QUEUE$ in the same order as they are stored in the $ETHER$.

Formally, we have that this equivalence is not captured by the standard bisimulation as it requires that each input operation should be matched exactly by an equivalent input operation, that is, the $ETHER$ and the $QUEUE$ are forced to consume data in the same order. In order to capture the above equivalence we need to move to the asynchronous bisimulation; indeed, it is not difficult to prove that ${}^bETHER \approx_a {}^bQUEUE$ for any actor name b. We do not report the complete proof for this equivalence, because standard, but we only point out some interesting related aspects.

When we exploit the asynchronous bisimulation we can assume that an input operation consuming a message with contents (put, d) performed by ${}^bETHER$ could be matched by ${}^bQUEUE$ by performing no action, provided that the agent ${}^bQUEUE|\langle b, (put, d)\rangle$ is equivalent to the term reached by the actor ${}^bETHER$. In order to prove this, we observe that ${}^bETHER_{\{d\}} \approx_a {}^bQUEUE|\langle b, (put, d)\rangle$. It is interesting to observe that this equivalence holds due to the previously described *unique repectiveness* property, indicating that no external actor can observe the presence of the pending message $\langle b, (put, d)\rangle$.

Example 3. The actor model does not provide an explicit primitive for receiving only data having a certain structure or a certain contents. In fact an actor can read a message only when it is idle and each available message can be read independently from its sender or its contents. On the other hand, in real applications it is often necessary to specify the kind of data that are needed to be received in a particular point of the computation.

In this example we use the asynchronous bisimulation to prove the correctness of the implementation of a new command that explicitly permits to receive only messages containing a particular identifier. As we want to have more information than the identifier inside the contents of the message, we consider that the messages are records with a field id, containing the identifier, and possibly other fields that we left unspecified.

In general we denote a record r by $(f_1{:}e_1, \ldots, f_n{:}e_n)$ and the selection of the value in the field f_i is written as $r.f_i$.

We consider a new primitive $receive(e, x)$ which forces to receive only a message with the value e in the field id. Formally, we extend the syntax of the language by allowing also programs of the following kind:

$$P ::= \ receive(e, x).P$$

having the following operational semantics:

$$^{a}[receive(e, x).P]_s \xrightarrow{av} {}^{a}[P\{v/x\}]_s \qquad \text{if } v.id = [\![e]\!]^a_s$$

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 any *receive* commands.

Our idea for implementing the program $receive(e, x).P$ by a term of the algebra denoted by $[\![receive(e, x).P]\!]$ is to define a behaviour which executes the program P only if the message has the desired id; otherwise it resends the received message and becomes idle waiting for another one:

$$[\![receive(e, x).P]\!] \stackrel{def}{=} become(REC, state)$$

where:

$$\begin{aligned} REC \stackrel{def}{=} \ & (message.id = e)\colon P\{message/x\} + \\ & (message.id \neq e)\colon send(self, message). \\ & \qquad\qquad\qquad\qquad become(REC, state) \end{aligned}$$

The correctness of our mapping is proven by the fact that the equivalence $^{a}[receive(e, x).P]_s \approx_a {}^{a}[[\![receive(e, x).P]\!]]_s$ holds 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 the sender is 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 introduce a busy waiting; for example, if only messages without the desired id are received by 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.

4 A Specification of the CORBA Messaging Service

The actor model and the algebra presented in the previous sections provide an abstract representation of the CORE OMG object model in an untyped scenario. The further extension to our algebra is to add the distinction between clients and servers objects and an adequate abstract syntax for the CORBA models of requests invocation.

4.1 CORBA clients

CORBA clients are represented as actors performing sequences of requests. To this aim we extend the syntax for the behaviour of our actor algebra by introducing also:

$$\begin{aligned} P ::= \; & \mathit{INVOKE}(a, e, x).P \mid \mathit{ONEWAY}(a, e).P \mid \mathit{SEND}(a, e).P \mid \\ & \mathit{GET\_RESPONSE}(a, e, x).P \mid \mathit{SENDC}(a, e, P').P \mid \\ & \mathit{SENDP}(a, e, b).P \mid \mathit{IS\_READY}(b, e, x).P \mid \\ & \mathit{POLL\_GET\_RESPONSE}(b, e, x).P \end{aligned}$$

Several kinds of parameters are considered: a indicates the name of the actor representing the server the client is asking for the service, e is a record that describes the request, x is the value variable that will contain the results returned by the server, P' is the program that the callback object will perform after having received the result of the request in the case of *SENDC*, and b is the name of the poller object in the case of *SENDP*.

According to this extended syntax an actor can perform the CORBA primitives specified above and the primitives of the actor algebra. We assume that a CORBA client can only perform CORBA primitives.

4.2 CORBA servers

CORBA servers are represented as actors that reply to the client requests according to the specification of the provided services. As already stated, requests are described as records; we suppose that these records contain at least three fields: *id* that contains an identifier that uniquely characterizes the request, *to* that indicates the name of the object to which the server has to send the results of the request, and *client* which is filled with the name of the client asking for the service. It is interesting to note that the fileds *to* and *client* are not in general the same. For example, in the case of the asynchronous invocations *SENDC* and *SENDP* the client will ask to the server to send the results to the callback or to the poller object respectively and not to itself.

We assume the existence of a function $f(a, e)$ that, given a server a and a request e, returns the results that the server will compute. We suppose that $f(a, e)$ is a record that contains the field *id* filled with the identifier of the current request.

Usually servers are idle waiting for requests: whenever a request is delivered to a server, it deterministically computes the answer, sends the result to the correct destination and possibly modifies its state.

Idle servers are denoted by ${}^{a}S_s$ where a is the name of the server, S its behaviour, and s its state. The behaviour of a server can be defined formally as follows:

$$\langle a, e \rangle \mid {}^{a}S_s \stackrel{\tau}{\Longrightarrow} \langle b, v \rangle \mid A \quad \text{iff } b = e.to,\ v = f(a, e) \text{ and } A = {}^{a}S_{s'}$$

where s' is the new state of the server which is a function of the previous state s and of the request e.

In this way we provide a specification of CORBA servers which is independent from the model of request invocation used by clients. We also abstract from the server interface, and we assume that a server is able to execute a set of operations and that the name of the method and its arguments are transmitted in the request.

4.3 CORBA Models of Request Invocation

We present a characterization of the possible request invocations, giving the indication of how a client behaves in each possible case.

- **INVOKE**. The program $\mathit{INVOKE}(a, e, x).P$ implements the synchronous model of request invocation of CORBA, basically it is a remote procedure call where a is the name of the server object, e is the request containing the operation and the arguments and x is the result that can be used in the rest of the program P.
  We implement this primitive by a term $[\![\mathit{INVOKE}(a, e, x).P]\!]$ which is defined by exploiting the *send* and the *receive* operations of the actor algebra. When the answer (which is a record with the corresponding field id) is received, the program P can be executed.

  $$[\![\mathit{INVOKE}(a, e, x).P]\!] \stackrel{def}{=} \mathit{send}(a, e).\mathit{receive}(e.id, x).P$$

  Here we suppose that $e.to = \mathit{self}$ (i.e. the name of the client actor) in order to be sure that the results are sent back to the correct client.
- **ONEWAY.** According to this model of request invocation the answer of the server will never be consumed by the client. We model this by creating a new empty actor to which the server will send the results of the request.

  $$[\![\mathit{ONEWAY}(a, e).P]\!] \stackrel{def}{=} \mathit{create}(b, \surd).\mathit{send}(a, e).P$$

  Here we suppose that $e.to = b$ in order to ensure that the server will send to the new empty actor its results. In this way the produced results will be consumed by the new object that performs a sort of garbage collection.
- **SEND.** This is the deferred synchronous model of request invocation. The request $\mathit{SEND}(a, e)$ is transmitted to the server object through the ORB and then the client retrieves the answer exploiting $\mathit{GET\_RESPONSE}(a, e, x)$ which binds the variable x to the result of the request.

  $$[\![\mathit{SEND}(a, e).P]\!] \stackrel{def}{=} \mathit{send}(a, e).P$$

Here we suppose that $e.to = self$. The answer will be consumed at the moment the client performs a $GET\_RESPONSE$; in order to ask for the result of the correct request e, we use its unique identifier $e.id$ as handler:

$$[\![GET\_RESPONSE(a, e, x).P]\!] \stackrel{def}{=} receive(e.id, x).P$$

- **SENDC.** This is an asynchronous model of invocation: $SENDC(a, e, P')$ creates a callback object that will manage the answer executing the program P' after having received the result of the request, and sends the request and the address of the callback object to the server object through the ORB. Finally it executes the rest of the program.

$$\begin{aligned}[\![SENDC(a, e, P').P]\!] \stackrel{def}{=} \; & create(d, CALLBACK_{eP'}). \\ & send(a, e).P\end{aligned}$$

Here we suppose that $e.to = d$ and that the behaviour $CALLBACK_{eP'}$ is defined as follows:

$$\begin{aligned}CALLBACK_{eP'} \stackrel{def}{=} \; & (message.id = e.id)\colon P'\{message/answer\} + \\ & (message.id \neq e.id)\colon send(self, message). \\ & \qquad\qquad become(CALLBACK_{eP'}, state)\end{aligned}$$

where $answer$ is a special expression used inside the program P in order to denote the content of the message that the callback object will receive from the server. Note that in this case the *to* field of the request is set to the address of the callback object d.

- **SENDP.** The polling model of request invocation creates a poller object that will be used by the client to retrieve the answers.

$$[\![SENDP(a, e, b).P]\!] \stackrel{def}{=} create(b, POLLER_e, (empty, empty)).send(a, e).P$$

Here we suppose $e.to = b$ as in this case the server will have to send the response to the poller actor. The state of the poller actor is a pair, initialized to $(empty, empty)$, that contains two kinds of information. The first is the name of the client actor that is introduced in the state of the poller whenever the client asks for the results that are not yet received by the poller object. The second element is used to store the results produced by the servers.
The client has the possibility to interact with the poller object by means of two primitives that are compiled as follows:

$$\begin{aligned}[\![IS\_READY(b, e, x).P]\!] \stackrel{def}{=} \; & send(b, is\_ready). \\ & receive(e.id, x).P \\ [\![POLL\_GET\_RESPONSE(b, e, x).P]\!] \stackrel{def}{=} \; & send(b, poll\_get\_response). \\ & receive(e.id, x).P\end{aligned}$$

Finally, we can describe the behaviour of the poller as follows:

$$
\begin{array}{l}
POLLER_e \stackrel{def}{=} \\
\quad (message.id = e.id) and (1st(state) = empty): \\
\qquad become(POLLER_e, (empty, message)) + \\
\quad (message.id = e.id) and (1st(state) \neq empty): \\
\qquad send(1st(state), message) + \\
\quad (message = is\_ready) and (2nd(state) = empty): \\
\qquad send(e.client, false).become(POLLER_e, state) + \\
\quad (message = is\_ready) and (2nd(state) \neq empty): \\
\qquad send(e.client, true).become(POLLER_e, state) + \\
\quad (message = poll\_get\_response) and (2nd(state) = empty): \\
\qquad become(POLLER_e, (e.client, empty)) + \\
\quad (message = poll\_get\_response) and (2nd(state) \neq empty): \\
\qquad send(e.client, 2nd(state))
\end{array}
$$

This is a static specification of the poller object. A poller object always sends answers to the original client, *e.g.,* the client which sent the *SENDP* request. This feature can be used to prove properties of this model of request invocation, for instance the equivalence presented in the next section. A more dynamic specification of the poller object is also possible and it will be the subject of future work.

4.4 Some Properties

After having mapped the different kinds of request invocation to our algebra of actors, we can use the equivalences we have introduced in the previous section in order to prove formally some interesting, even if simple, properties.

First of all we consider the standard synchronous *INVOKE* and the deferred synchronous *SEND*. The difference between these two requests is that *INVOKE* blocks a client until a result is received, while *SEND* does not block a client and the result will be received at the moment the corresponding *GET_RESPONSE* operation is executed. The blocking behaviour of the *INVOKE* can be simply simulated also in the case of the deferred synchronous invocation by performing the *GET_RESPONSE* primitive after the *SEND* primitive. This is formalized by the following equivalence:

$$
\begin{array}{c}
{}^b[INVOKE(a,e,x).P]_s \\
\approx_a \\
{}^b[SEND(a,e).GET\_RESPONSE(a,e,x).P]_s
\end{array}
$$

Also a stronger equivalence result holds: the two terms obtained by substituting each CORBA primitive according to the corresponding definition are syntactically equals.

More interesting is to show that the synchronous *INVOKE* mechanism can be simulated also by using the asynchronous primitives *SENDC* and *SENDP*.

In the case of *SENDC*, the idea is that the client blocks after having asked for the service, until the callback object forwards the results. To this aim we force the callback object to perform the operation $send(e.client, x)$ where $e.client$ is the name of the client and x is the variable that will be bound to the results produced by the server. This is formalized by the following equivalence:

$$
\begin{gathered}
{}^{b}[INVOKE(a, e, x).P]_{s} \mid {}^{a}S_{s'} \\
\approx_{a} \\
{}^{b}[SENDC(a, e, send(e.client, answer)).receive(e.id, x).P]_{s} \mid {}^{a}S_{s'}
\end{gathered}
$$

In this case we have to take into account configurations composed of both the client and the server. In this way, the communication between them is not observable; indeed, if an external actor can observe the contents of the exchanged messages, it is easy to discriminate between the *INVOKE* and the *SENDC* mechanisms as the contents e of the two kinds of request are different (e.g. the contents of the field $e.to$).

The above equivalence can be proven by recalling the equivalence in the Example 1, and observing that the *CALLBACK* object, which is dynamically created, behaves like the *Forward* actor defined in that example. Moreover, the *CALLBACK* object is not visible to the outside because the name of new created actors is local to the actor itself and the parent actor. A new feature with respect to the Example 1 is that in this case the restriction is on the name of the object performing the forward operation and not on the client name. Note also that the above equivalence does not hold exploiting the CORBA primitives only. We have to resort to the low level message passing operations of the actor algebra when we specify the behaviour of the callback object.

A similar idea can be used in the case of the *SENDP* primitive where the command *POLL_GET_RESPONSE* can be used in order to retrieve the results via the poller object. This is formalized by the following equivalence:

$$
\begin{gathered}
{}^{b}[INVOKE(a, e, x).P]_{s} \mid {}^{a}S_{s'} \\
\approx_{a} \\
{}^{b}[SENDP(a, e, b).POLL\_GET\_REQUEST(b, e, x).P]_{s} \mid {}^{a}S_{s'}
\end{gathered}
$$

In this case, the forward operation is performed by the *POLLER* object, which will receive a new restricted name at the moment the *SENDP* operation is executed.

5 A CORBA Programming Example

We illustrate our approach showing how a specification of a client behaviour can be extracted from a simple CORBA program. We assume to have defined a server which is able to compute the result of mathematical formulas. We also assume a display object which is able to show the answer to the user. We assume the following OMG IDL interfaces:

```
interface Compute {
  string compute (in string formula);
};
```

```
interface Display {
  void display (in string message);
};
```

Here we suppose that formulas and results are represented as strings. We define an user interface client which takes a formula from the user, asks the server to compute it, waits for the answer and finally sends the answer to the display object.

We illustrate our example exploiting the CORBA Scripting Language (CORBAScript) [8], an interpreted scripting language dedicated to CORBA environments. CORBAScript provides an high level interface to implement CORBA clients and tools, hiding some of the specific mechanisms of CORBA. For example in CORBAScript the access to an IDL interface is simply done by providing its IDL interface identifier as illustrated below.

```
server = Compute("IOR:....");
display = Display("IOR:.....");
answer = server.compute("Formula");
display.display(answer);
```

Here we suppose that *Formula* is the string representing the formula to compute. We use an IOR (Interoperability Object Reference) to access the CORBA objects. The IOR is a unique reference to a CORBA object and thus can be modelled as an actor name. The variables *server* and *display* are references to CORBA objects, and the operations *display.display* and *server.compute* are synchronous requests invocations (*i.e., INVOKE*).

This situation is modelled by the following client program in the algebra of actors where we assume that *server* and *display* are the names of actors implementing the two servers.

$$
\begin{array}{llll}
\mathit{UICLIENT}_1 \stackrel{def}{=} & \mathit{message} = \mathit{compute}: & \mathit{INVOKE}(\mathit{server}, \mathit{Formula}, x). \\
& & \mathit{INVOKE}(\mathit{display}, x, y). \\
& & \mathit{become}(\mathit{UICLIENT}_1, \mathit{state}) + \\
& \mathit{message} = \mathit{other}: & P
\end{array}
$$

A drawback of this kind of interaction between the client and the mathematical server, is that the client is blocked on the call *INVOKE(server,Formula,x)* waiting for the results. In this way, the user interface program is not able to accept the *other* input which activates the generic program P.

This problem can be solved by using the new asynchronous request invocation mechanisms. We represent this alternative approach in the algebra of actors only, because CORBAScript does not yet support the asynchronous messaging facilities.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathit{UICLIENT}_2 \stackrel{def}{=}} \\
\mathit{message} = \mathit{compute}: & \mathit{SENDC}(\mathit{server}, \mathit{Formula}, \mathit{INVOKE}(\mathit{display}, \mathit{answer}, y)). \\
& \mathit{become}(\mathit{UICLIENT}_2, \mathit{state}) + \\
\mathit{message} = \mathit{other}: & P
\end{array}
$$

Here the client performs an asynchronous request and thus it does not block and it is able to deal with the *other* inputs activating the generic program P. The display request is performed by the callback object created at the time the $SENDC$ primitive is executed.

Now, it is interesting to compare formally the two above client descriptions. The result we have obtained is that the two different approaches are equivalent:

$$
\begin{gathered}
{}^{b}[\mathit{UICLIENT}_1]_s \mid {}^{server}S_{s'} \mid {}^{display}D_{s''} \\
\approx_a \\
{}^{b}[\mathit{UICLIENT}_2]_s \mid {}^{server}S_{s'} \mid {}^{display}D_{s''}
\end{gathered}
$$

where the actors ${}^{server}S_{s'}$ and ${}^{display}D_{s''}$, representing the mathematical and the displaying servers respectively, are defined according to the indication reported in Section 4.2.

This equivalence can be proven following the same lines described in Section 4.4. The most interesting aspect to observe is that the second client is able to start the execution of P even if the request has not been considered by the mathematical server yet. The execution of P is started whenever an *other* input is received from the client. In order to simulate the same behaviour, also the first client has to perform the same input action. This can happen only if the two server invocations have been completely executed. Even if this behaviour is different, it does not permit to distinguish between the two above agents; indeed, the two reached actors are equivalent:

$$
\begin{gathered}
{}^{b}[P]_s \mid {}^{server}S_{s'} \mid {}^{display}D_{s''} \\
\approx_a \\
({}^{b}[P]_s \mid {}^{c}\mathit{CALLBACK}_{s'''} \mid \langle server, request \rangle) \setminus c \mid {}^{server}S_{s'} \mid {}^{display}D_{s''}
\end{gathered}
$$

This equivalence holds because the $CALLBACK$ object, the message containing the *request* for the server, as also the messages that will be exchanged among the servers and the $CALLBACK$ object, are not observable to the outside.

It is interesting to observe that in this case we have not followed the previous general way to implement the $INVOKE$ primitive using the callback mechanism. Indeed, here the callback object is not a forwarder, but it is responsible for the execution of the displaying request. The approach used in this case is more efficient, because the client does not block waiting for the answer of the server forwarded by the callback object, but it directly delegates the execution of the continuation to the callback.

6 Related Work

Najm and Stefani [26] present a notion of computational model for open distributed processing which can also be exploited for modelling the new features of CORBA for instance the quality of service [14]. Since their model is based on rewriting logic and they do not deal with the issue of modelling asynchronous requests, their work can be considered complementary to our work. An interesting point could be to analyse if their proposal for modelling quality of service can be successfully used in an asynchronous context.

Bastide et al. [6] propose to extend the interface definition of CORBA distributed objects by a specification of their behaviour expressed exploiting high level petri nets. Although our goal was different, *e.g.,* to provide a formal representation of CORBA clients which interact exploiting the new CORBA asynchronous messaging service, the algebra of actors could also be used to provide a formal account of CORBA services.

In the past few years, several advances have been achieved on the semantics of actors, dealing with aspects of communication and concurrency [4, 3, 30, 31, 21], but these papers do not investigate the relationships of the actor model with traditional process algebras, even though recently Robin Milner [24] suggested that it may be worthwhile to work in this direction. 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. The reader interested in a more detailed comparison with other calculi can refer to [11, 10, 9].

7 Conclusion

The main result presented in this paper is the development of a process algebraic specification of the new CORBA models for request invocation.

The specification is based on an algebra of actors that enjoys a clean formal definition and a rich algebraic theory, inspired from the π-calculus, while preserving a basic object-oriented features such as object identity, asynchronous message passing, an implicit receive mechanism and support for dynamic object creation. This approach allows us to reuse standard results of the theory of concurrency for reasoning on CORBA models of request invocation.

The algebra of actors is well suited to specify most of the aspects of CORBA which concern interaction of objects, their creation and their deletion. Thus CORBA services which define these features [15], such as the Life Cycle Service, the Concurrency Service, the Transaction Service or the Naming Service are good candidates for being modelled with our algebra and they will be target of our future work. Moreover other CORBA services which describe internal aspects of the involved objects can also be modelled defining their interfaces and their functionalities in an abstract way. For example we have followed this approach in the formal definition of a CORBA server in Section 4.2.

CORBA Exceptions can be also modelled as messages of the process algebra provided that they are included in the server specification. Namely, the specification of the CORBA server that we have presented in Section 4.2 must be extended considering that one of the possible outcomes of a request invocation can be an exception.

On the other hand, aspects such as types [12], quality of service and objects by value [32] have not an immediate mapping in our algebra. Thus a formal specification of all the CORBA features is not feasible with the current version of our algebra, and a significant extension is required to achieve this ambitious goal. A less ambitious goal will be to provide a specification of the CORBAScript [8] language which is more abstract and hides most of the low-level mechanisms of CORBA. Nevertheless, we still claim that a more formal description of interaction aspects of the CORBA services will be useful both to CORBA implementors and programmers.

Finally, a number of additional research items still need to be carried out in this research. For instance: a study of how typing and inheritance issues, such as in [1] can be integrated in our framework to have a more real model of CORBA; the formulation of algebraic laws that characterize the equivalences of CORBA programs, for example an axiomatization for the asynchronous bisimulation and its application to CORBA; and the definition of a framework for formal reasoning about CORBA programs, *e.g.,* following the style of the Hennessy and Milner logic [22].

Acknowledgements

We would like to thank the anonymous referees for their detailed reports, and we acknowledge also interesting discussions with Carolyn Talcott as also her comments on a previous version of this paper.

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. F. Smith, and C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(1):1–69, January 1997.
4. 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.
5. R. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.
6. R. Bastide, O. Sy, and P. Palanque. Formal Specification and Prototyping of CORBA Systems. In *Proc. of ECOOP99*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1999. This volume.
7. G. Boudol. Asynchrony and the $\pi-$*calculus*. Technical Report INRIA-92-1702, INRIA Sophia-Antipolis., 1992.
8. Laboratoire d'Informatique Fondamentale de Lille and Object Oriented Concepts Inc. CORBA Scripting Language. Technical Report orbos/98-12-09, Object Management Group, 1998.
9. M. Gaspari. Concurrency and knowledge-level communication in agent languages. *Artificial Intelligence*, 105(1-2):1–45, 1998.

10. 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.
11. Mauro Gaspari and Gianluigi Zavattaro. An algebra of actors. In *Proc. 3nd IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 3–18. Kluwer Academic Publishers, Feb 1999.
12. Object Management Group. OMG Architecture Guide: The OMG Object Model. Technical report, Object Management Group, 1992.
13. Object Management Group. Messaging Service RFP. Technical Report orbos/96-03-16, Object Management Group, Framingham, MA, 1996.
14. Object Management Group. CORBA Messaging Joint Revised Submission. Technical Report orbos/98-05-05, Object Management Group, Framingham, MA, 1998.
15. Object Management Group. CORBA Services. Technical Report formal/98-12-09, Object Management Group, 1998.
16. Object Management Group. *The Common Object Request Brocker: Architecture and Specification*. Object Management Group, 1998.
17. C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.
18. CAR. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
19. 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.
20. S.N. Khoshafian and G.P. Copeland. Object Identity. In *Proc. OOPSLA '86*, pages 406–416, September 1986.
21. I.A. Mason and C. Talcott. A Semantically Sound Actor Translation. In *Proc. of ICALP'97*, Lecture Notes in Computer Science. Springer Verlag, 1997.
22. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
23. R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
24. R. Milner. Elements of interaction. *Communications of the ACM*, 36(1):79–89, January 1993.
25. 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.
26. 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.
27. 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.
28. 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.
29. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark, 1981.
30. C. Talcott. Interaction Semantics for Components of Distributed Systems. In *Proc. FMOODS'96*, pages 154–169. Chapmann & Hall, 1996.

31. C. L. Talcott. An actor rewriting theory. In J. Meseguer, editor, *Proc. 1st Intl. Workshop on Rewriting Logic and its Applications*, number 4 in Electronic Notes in Theoretical Computer Science, pages 360–383. North Holland, 1996.
32. S. Vinoski. New Features for CORBA 3.0. *Communications of the ACM*, 41(10), October 1998.
33. D. Walker. Objects in the π–*calculus*. *Information and Computation*, 116(2):253–271, 1995.