ELSEVIER

Available online at www.sciencedirect.com

SCIENCE DIRECT®

The Journal of Logic and Algebraic Programming 63 (2005) 271–297

THE JOURNAL OF LOGIC AND ALGEBRAIC PROGRAMMING

www.elsevier.com/locate/jlap

An object based algebra for specifying a fault tolerant software architecture

Nicola Dragoni *, Mauro Gaspari

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

Abstract

In this paper we present an algebra of actors extended with mechanisms to model crash failures and their detection. We show how this extended algebra of actors can be successfully used to specify distributed software architectures. The main components of a software architecture can be specified following an object-oriented style and then they can be composed using asynchronous message passing or more complex interaction patterns. This formal specification can be used to show that several requirements of a software system are satisfied at the architectural level despite failures. We illustrate this process by means of a case study: the specification of a software architecture for intelligent agents which supports a fault tolerant anonymous interaction protocol.

Keywords: Process algebra; Software architectures; Actors; Fault tolerance; Object-oriented design

1. Introduction

The object-oriented paradigm has been successfully used in many fields of computer science influencing methodologies, techniques, programming languages and tools. Among them object-oriented design can be considered a standard approach for the design phase in the development of software systems. Object-oriented design provides a methodology to organize the main building blocks of a software system exploiting objects, classes and inheritance. This also holds for the first phase of the design process which concerns the specification of a software architecture. A software architecture is an abstract view of a software system distinct from the details of implementation, algorithms, and data representation. The object-oriented approach allows a software designer to efficiently characterize and organize the main components of a software architecture providing a clean and reusable formalization. However, in the design phase another dimension of complexity arises which concerns the interaction among the components of a software architecture. In fact

* Corresponding author.
E-mail addresses: dragoni@cs.unibo.it (N. Dragoni), gaspari@cs.unibo.it (M. Gaspari).

1567-8326/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2004.05.006

formalisms for object-oriented design have often components to model the dynamics of a system, for example the UML activity diagrams [1]. The object-oriented paradigm recommends object identity, methods and message-passing to govern object interaction, but it is not trivial to extend all these concepts to a distributed scenario [2] and to encapsulate them in an abstract and compositional specification language. On the other hand, due to their compositional and abstract nature, process algebras have been widely adopted for the specification of software systems, especially those with communicating, concurrently executing components. However, most of the efforts are oriented to study process algebras such as CCS [3] or the π-calculus [4] which do not provide a direct representation of objects as first class entities. In these formalisms processes are stateless entities (although a notion of state can be simulated using value passing combined with recursion or parallel composition) which communicate using synchronous message passing and the representation of an object involves a large number of processes. As a consequence of this situation it is difficult to import the compositional properties of standard process algebras to an object oriented specification, because most of the laws concern stateless processes and not objects (or complex components).

An additional problem arises when software architectures deal with distributed systems which are often subject to failures of some of their components. A reasonable property which could be expressed at the architectural level for these systems is to guarantee some degree of fault tolerance. To achieve this goal a process algebra used for specifying a software architecture should provide abstract mechanisms for modeling failures and for reasoning about them. Despite distributed software systems have achieved a dominant role in computing in the last decade, currently there are only a few proposal which extend process algebras with mechanisms to model failures and to deal with them. A brief survey of the literature is reported in [5].

We argue that both an object-oriented specification style and an adequate support for modeling failures and their detection are fundamental advances in process algebra needed to improve its capabilities in designing complex system at the architectural level.

In this paper we address this issue extending a process algebra based on a distributed object-oriented model (the Actor model [6]) with mechanisms to model crash failures of actors and their detection. The Algebra of Actors [7] represents a compromise between standard process algebras and the Actor model, providing standard concurrency features in a context where object-based mechanisms (such as *object identity*, *asynchronous message passing*, *implicit message acceptance* and *dynamic object creation*) are supported. The extended formalism allow us to model usual components of fault tolerant systems, for example failure detectors, as actors.

We show how this extended algebra of actors can be successfully used to specify distributed software architectures. The main components of a software architecture can be specified following an object-oriented style and then they can be composed using asynchronous message passing or more complex interaction patterns. The result of a formal specification is a set of actor terms which represent the main building blocks of a software architecture. These terms can be used to demonstrate that several requirements of a software system are satisfied at the architectural level despite failures.

We illustrate this process by means of a case study: the specification of a software architecture for intelligent agents which supports a fault tolerant anonymous interaction protocol.

The paper is organized as follows. In Section 2 we recall the algebra of actors. In Section 3 we provide a classification of failures in distributed systems and we introduce the concept

of unreliable failure detector. In Section 4 we present an extension of the actor algebra to formalize crash failures of actors and failure detectors. In Section 5 we present the specification of an agent architecture which supports an anonymous interaction protocol outlining the main design requirements, and subsequently, in Section 6, we show that the specification satisfies these requirements. We conclude the paper by discussing related work and highlighting our future research directions.

2. An algebra of actors

Actors are self-contained stateful reactive processes whose behavior is a function of incoming communications. Each actor has a unique name (e.g. mail address) determined at the time of its creation. This name is used to specify the recipient of a message supporting object identity, a property of an actor which distinguishes each actor from all others. 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 make use of three basic primitives which are asynchronous and non-blocking: *create*, to create new actors; *send*, to send messages to other actors; *become*, to change the behavior of an actor [6].

Let $\mathcal{A}$ be a countable set of *actor names*: $a, b, c, a_i, b_i, \ldots$ will range over $\mathcal{A}$ and $L, L', L'', \ldots$ will range over its (finite) power set $\mathcal{P}_{\text{fin}}(\mathcal{A})$ (i.e., $L, L', L'' \subseteq_{\text{fin}} \mathcal{A}$). Let $\mathcal{V}$ be a set of values (with $\mathcal{A} \subset V$) containing, e.g., *true*, *false*, and let $\mathcal{X}$, ranged over by $x, y, z, \ldots$, 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: 1*st*, 2*nd*, *rest*, *empty*. We will denote values by $v, v', v'', \ldots$ when they appear as contents of a message and with $s, s', s'', \ldots$ when they represent the state of an actor. $[\![e]\!]_s^a$ gives the value of e in $\mathcal{V}$ assuming that a and s are substituted for *self* and *state* inside e; e.g. $[\![self]\!]_s^a = a$ and $[\![state]\!]_s^a = s$. The special expression *message* represents the contents of the last received message. Whenever a message is received, its contents are substituted for each occurrence of the expression *message* in the receiving actor.

Let $\mathcal{C}$ be a set of *actor behaviors* identifiers: $C, C', \ldots$ will range over $\mathcal{C}$. We suppose that every identifier C is equipped with a corresponding behavior definition $C \stackrel{\text{def}}{=} P$, where P is an *actor program* defined as "a sequence of actor primitives (*become*, *send* and *create*) and guarded choices $e_1 : P_1 + \cdots + e_n : P_n$ terminating in the null program $\surd$". Formally:

$$P ::= become(C, e).P \mid send(e_1, e_2).P \mid create(b, C, e).P \mid e_1 : P_1 + \cdots + e_n : P_n \mid \surd$$

We allow recursive behaviors to be defined. For example, we could have

$$C \stackrel{\text{def}}{=} become(C, state).\surd$$

Actor terms are defined by the following abstract syntax:

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

An actor can be idle or active. An idle actor ${}^aC_s$ (composed by a behavior C, a name a, and a state s) is ready to receive a message. When a message is received, the actor becomes active.

An active actor is denoted by ${}^a[P]_s$ where P is the program that the actor is executing. The actor a will not receive new messages until it becomes idle (by performing a *become* primitive).

An actor term is the parallel composition of (active and idle) actors and messages. A message is denoted by a term $\langle a, v\rangle$ where v is the contents and a the name of the actor the message is sent to.

A restriction operator $A\backslash a$ is used in order to allow the definition of local actor names ($A\backslash L$ is used as a shorthand for $A\backslash a_1\backslash\cdots\backslash a_n$ if $L = \{a_1, \ldots, a_n\}$) while $\mathbf{0}$ is the usual empty term.

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

- *send*:
  The program $send(e_1, e_2).P$ sends a message with contents e_2 to the actor indicated by e_1:
  $$ {}^a[send(e_1, e_2).P]_s \xrightarrow{\tau} {}^a[P]_s \mid \langle [\![e_1]\!]^a_s, [\![e_2]\!]^a_s\rangle \tag{1}$$
  where τ represents an internal invisible step of computation.
- *become*:
  The program $become(C, e).P'$ changes the state of the current actor from active to idle:
  $$ {}^a[become(C, e).P']_s \xrightarrow{\tau} ({}^d[P'\{a/self\}]_s)\backslash d \mid {}^a C_{[\![e]\!]^a_s} \quad \text{with } d \text{ fresh} \tag{2}$$
  The primitive *become* is the only one that permits the state to change according to the expression e; we sometimes omit e if the state is left unchanged (i.e. $e = state$). The continuation P' is executed by the new actor ${}^d[P'\{a/self\}]_s$. This actor will never receive other messages (i.e. it is unreachable) as its name d cannot be known to any other actor. Indeed, the expression *self*, which is the only one that returns the value d, is changed in order to refer to the name a of the initial actor.
- *create*:
  The program $create(b, C, e).P'$ creates a new idle actor having state s and behavior C:
  $$ {}^a[create(b, C, e).P']_s \xrightarrow{\tau} ({}^a[P'\{d/b\}]_s \mid {}^d C_{[\![e]\!]^a_s})\backslash d \quad \text{with } d \text{ fresh} \tag{3}$$
  The new actor receives a fresh name d. This new name is initially known only to the creating actor. In fact, a restriction on the new name d is introduced.
- $e_1 : P_1 + \cdots + e_n : P_n$:
  In the agent $e_1 : P_1 + \cdots + e_n : P_n$, the expressions e_i are supposed to be boolean expressions with value *true* or *false*. The branch P_i can be chosen only if the value of the corresponding expression e_i is *true*:
  $$ {}^a[e_1 : P_1 + \cdots + e_n : P_n]_s \xrightarrow{\tau} {}^a[P_i]_s \quad \text{if } [\![e_i]\!]^a_s = true \tag{4}$$

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 terms, 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 [8] which consists in separating the laws which govern the static relation among actors

(for instance $A|B$ is equivalent to $B|A$) from the laws which define their interaction. This is achieved defining a static structural equivalence relation over syntactic terms and a dynamic relation by means of a Labelled Transition System (LTS) [9].

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

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

Definition 2 (*Computations*). A transition system modeling computations in the actor algebra is represented by the triple $(\Gamma, T, \{\overset{\alpha}{\rightarrow} \mid \alpha \in T\})$. $T = \{\tau\} \cup \{av, \overline{av}L \mid a \in \mathcal{A},\ v \in \mathcal{V},\ L \subseteq \mathcal{A}\}$ is a set of labels, where τ is the invisible action standing for local 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, i.e. names which are made available to actors which were outside their initial scope. $\overset{\alpha}{\rightarrow}$ is the minimal transition relation satisfying the axioms and rules presented in Table 1.

Note that actors do not have an explicit receive primitive, which is instead *implicit*. Therefore, the receive operation does not correspond to an operation in the programming

Table 1
Operational semantics of the actor algebra

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	$\langle a, v \rangle \xrightarrow{\overline{av}\emptyset} \mathbf{0}$	
Become	${}^a[become(C, e).P']_s \xrightarrow{\tau} ({}^d[P'\{a/self\}]_s)\backslash d \mid {}^aC_{[\![e]\!]^a_s}$	d fresh
Create	${}^a[create(b, C, e).P']_s \xrightarrow{\tau} ({}^a[P'\{d/b\}]_s \mid {}^dC_{[\![e]\!]^a_s})\backslash d$	d fresh
Receive	${}^aC_s \xrightarrow{av} {}^a[P\{v/message\}]_s$	if $C \overset{\text{def}}{=} P$
Guard	${}^a[e_1 : P_1 + \cdots + e_n : P_n]_s \xrightarrow{\tau} {}^a[P_i]_s$	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	$\dfrac{A \xrightarrow{\alpha} A'}{A|B \xrightarrow{\alpha} A'|B}$	$if\ \alpha = \overline{av}L$ *then* $a \notin act(B) \wedge L \cap fn(B) = \emptyset$
Sinc	$\dfrac{A \xrightarrow{av} A' \quad B \xrightarrow{\overline{av}L} B'}{A|B \xrightarrow{\tau} (A'|B') \setminus L}$	
Cong	$\dfrac{B \equiv A \quad A \xrightarrow{\alpha} A' \quad A' \equiv B'}{B \xrightarrow{\alpha} B'}$	

language and it is performed implicitly at certain points of the computation: only idle actors receive messages, and so become activated.

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*). The other rules are simply adaptations to our calculus of the standard laws for the π-calculus.

The restriction operator allows to define local names, hence only actions which does not include restricted actor names can be executed by the agent $A\backslash a$ (rule *Res*). The only way to pass through a restriction is defined by the rule *Open*: an actor can send restricted actor names in order to make them known to actors external to the restriction. In this case the names sent to the outside are no more restricted and they are stored in the set L of the label $\overline{av}L$. The process of extending the scope of the restriction terminates only when the message is received (rule *Sinc*), here the restriction on the actor names in the set L is reintroduced.

The rule *Par* states that the actor term $A|B$ can deliver a message inferred by A (i.e. execute an emission action $\overline{av}L$), only if B does not contain the target actor (i.e. $a \notin act(B)$) and the names exported (i.e. names in L) are free names in B. In this way, an actor a, which is initially out of the scope of an actor name b, will be able to send a message to the actor b only if the name b is explicitly communicated to it.

2.1. Discussion

There are several differences with respect to the formal semantics of actors of Agha et al. in [10–13] 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 and communication related aspects and not with issues concerning the sequential execution of programs inside actors.
- The operational semantics of the algebra of actors is defined by means of a Labelled Transition System LTS instead of a simple reduction system as in [10] or the rewriting rules in [13]. This allows to use standard observational equivalences of process algebras, such as bisimulation, without defining explicit observers (as we show in [7]).
- We have introduced the guarded choice as an alternative to the conditional which is present in previous formalization of actors [10].
- We provide an explicit representation of the state of an actor while in Agha et al. the state is part of the behavior of an actor.
- We have introduced a mechanism to model termination of actors. Actors are not perpetual processes with a default behavior as usual, but they can terminate: an actor terminates whenever it finishes its internal computation. This is not a limitation because a perpetual actor can always be obtained performing an explicit *become* operation for each internal computation. Note that this mechanism slightly modifies the reliability assumption of the actor model. In fact, we guarantee that a given message will always reach its destination only if the receiving actor is alive.
- In the algebra of actors, the creation of an actor is performed by a single basic primitive (*create*), while in the semantics of Agha et al. is composed of two basic operations: the creation of an empty actor and the initialization of its behavior. The main advantage of

our approach is that we do not need to restrict the possible computations to guarantee an atomic create operation.
- We introduce a restriction operator similar to the one of the π-calculus [4]. This operator is more tractable with respect to the approach of [10], which is based on the specification of the sets of receptionists and external actors in actor configurations. On the other hand, the calculus presented in [12] uses the inverse operator indicating the actors which are reachable from the outside world explicitly.

The full algebra of actors, including a discussion about different notions of equivalence of actor terms, can be found in [7,14,15].

3. Failures and failure detectors

Classifying failures and understanding their nature is fundamental if one wants to design an architecture of a distributed system which is able to tolerate and/or continue service despite malfunctions. Hence failures must be considered essential aspects of distributed systems. Problems in fault-tolerant distributed computing have been studied in a variety of computational models [16]. Such models fall into two broad categories, *message-passing* and *shared-memory*. In the former, processes communicate by sending and receiving messages over the links of a network; in the latter, they communicate by accessing shared objects, such as registers, queues, etc. Since actors communicate by means of an asynchronous message passing mechanism, in this paper we focus only on message-passing models. The parameters which characterise a particular message-passing model may be the following: synchrony of processes and communication, types of actor failures, types of communication failures, network topology, and deterministic versus randomized processes.

In this section we present the failure model that we consider for actors based on a well known classification of process failures in distributed systems [16]. Then we recall the notion of *unreliable failure detector* for asynchronous distributed systems [17] which will be used as a starting point to model a failure detector primitive in the actor algebra.

3.1. Actor failures

An actor is *faulty* in an execution if its behavior deviates from that prescribed by the algorithm it is running; otherwise, it is *correct*. A *model of failure* specifies in what way a faulty actor can deviate from its algorithm. The following is a list of models of failures that have been studied in the literature:
- **Crash**: a faulty actor stops prematurely and does nothing from that point on. Before stopping, however, it behaves correctly.
- **Send omission**: a faulty actor stops prematurely, or intermittently omits to send messages, or both.
- **Receive omission**: a faulty actor stops prematurely, or intermittently omits to receive messages sent to it, or both.
- **General omission**: a faulty actor is subject to send or receive omission failures, or both.
- **Arbitrary** (sometimes called **Byzantine**): a faulty actor can exhibit any behavior whatsoever. For example, it can change state arbitrarily.

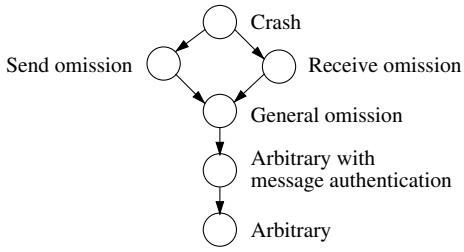

Fig. 1. Classification of failure models. An arrow from type B to type A indicates that A is more severe than B.

- **Arbitrary with message authentication**: faulty actors can exhibit arbitrary behavior but a mechanism for authenticating messages using unforgeable signature is available. With arbitrary failures, a faulty actor may claim to have received a particular message from a correct actor, even though it never did. A message authentication mechanism allows the other correct processes to validate this claim.

Note that these failures can be classified in terms of "severity", as shown in Fig. 1. Model A is *more severe* than model B if the set of faulty behaviors allowed by B is a proper subset of the set of those allowed by A. Thus, an algorithm that tolerates failures of type A, also tolerates those of type B. Arbitrary failures are the most severe failures, since they do not place any restrictions on the behavior of a faulty actor. Crash failures are the least severe failures listed above.

The failure model we consider in this proposal is characterised by *crash* failures of actors in a *fully asynchronous* system. Note that considering only crash failures is a common fault assumption in distributed systems, since several mechanism can be used to detect more severe failures and to force a crash in case of detection. Actors communicate by asynchronous and reliable message passing, i.e. whenever a message is sent it must be eventually received by the target actor (thus we do not handle communication failures, such as send or receive omission). The asynchrony of the system implies that there is no bound on message delay, clock drift or the time necessary to execute a step (so we omit all timing-based failures).

3.2. Failure detectors

Since impossibility results for asynchronous systems stem from the inherent difficulty of determining whether a process has actually crashed or is only "very slow", Chandra and Toueg [17] propose to augment the asynchronous model of computation with a model of an external failure detection mechanism that can make mistakes. In particular, they model the concept of *unreliable failure detector* for systems with *crash* failures.

Failure detectors are *distributed*: each process has access to a local *failure detector module*. Each local module monitors a subset of the processes in the system, and maintains a list of those that it currently suspects to have crashed. Each failure detector module can make mistakes by erroneously adding processes to its list of suspects: i.e., it can suspect that a process p has crashed even though p is still running. If this module later believes that suspecting p was a mistake, it can remove p from its list. Thus, each module may repeatedly add and remove processes from its list of suspects. Furthermore, at any given time the failure detector modules at two different processes may have different lists of suspects. It is important to note that the mistakes made by an unreliable failure detector

should not prevent any correct process from behaving according to specification even if that process is (erroneously) suspected to have crashed by all the other processes.

4. Modeling crash failures and failure detectors

In this section we present an extension of the actor algebra to formalize crash failures of actors and failure detectors. Our aim is to extend the computational model of the algebra with rules for modeling possible crashes of actors. We assume that any given actor can crash at any time and we introduce specific (crash) transitions to model these events. Crash transitions will be always enabled in the transition system and they will fire for both idle and active actors. However, although the transition system has been extended for modeling crashes, actors will not be able to detect faulty actors using their standard primitives. In fact the behavior of an actor only depends on its local state and on the incoming messages. An actor (and in general a process) is not aware of the state and properties of other actors, unless it will be explicitly notified by appropriate messages. For this reason we extend the algebra with a specific *ping* primitive which will be the basis to realize an unreliable failure detector.

4.1. Crash failures in the actor algebra

In order to model a crash failure in the algebra, we need to extend the standard behaviors of actors. As we have seen in Section 2, an actor can be idle (when is ready to receive a message) or active (when it receives a message). To model a crash, we provide a syntactic symbol ${}^{a}\mathbf{0}$ for each actor $a \in \mathcal{A}$, which indicates that actor a has crashed. Consequently the set of actor terms of the algebra is updated with this new term:

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

Any correct actor in the system can crash and consequently become a faulty actor, as described in the following transition rules:

$${}^{b}C_s \xrightarrow{\tau} {}^{b}\mathbf{0} \tag{5}$$

$${}^{b}[P]_s \xrightarrow{\tau} {}^{b}\mathbf{0} \tag{6}$$

The first rule deals with a crash of an idle actor ${}^{b}C_s$. We model this failure by means of a transition from the *idle* actor to the respective *faulty actor*. The second rule is analogous to the first one and deals with a crash of an *active* actor ${}^{b}[P]_s$.

When one of the previous transitions fires, then an actor becomes faulty and therefore will not be able to do nothing from that point on. Note that, consistently with the rules *Send* and *Receive* (Table 1), only correct actors are able to send and receive messages.

4.2. Detecting failures in the actor algebra

To detect crashes of actors we need to extend the algebra with an appropriate primitive that is usually called *ping* in distributed systems. The task of this primitive is inherently difficult in asynchronous distributed systems, because in general it is not possible to detect if a certain site has crashed or is only very slow. Our aim is to model this uncertainty in

Table 2
Fault tolerant transition rules

Crash	${}^{b}\mathbb{C}_s \xrightarrow{\tau} {}^{b}\mathbf{0}$
Ping1	${}^{b}\mathbf{0} \mid {}^{a}[ping(b,x).P]_s \xrightarrow{\tau} {}^{b}\mathbf{0} \mid {}^{a}[P\{false/x\}]_s$
Ping2	${}^{b}\mathbb{C}_s \mid {}^{a}[ping(b,x).P]_s \xrightarrow{\tau} {}^{b}\mathbb{C}_s \mid {}^{a}[P\{true/x\}]_s$
Ping3	${}^{b}\mathbb{C}_s \mid {}^{a}[ping(b,x).P]_s \xrightarrow{\tau} {}^{b}\mathbb{C}_s \mid {}^{a}[P\{false/x\}]_s$

the actor algebra. We introduce a primitive having the form: $ping(a, x)$, where $a \in \mathcal{A}$ and $x \in \mathcal{X}$, and we assume the following behavior. Given an actor b:

- If b has crashed then $ping(b, x)$ binds x to *false*.
  This means that if an actor b really crashes, then it is suspected by every correct actor which executes $ping(b, x)$.
- If b is alive then $ping(b, x)$ binds x to *true*, but it can also make a mistake:
  - $ping(b, x)$ binds x to *true* (b is alive) OR
  - $ping(b, x)$ binds x to *false* (b is assumed to be crashed but is only very slow).

  As a consequence, if an actor b is correct then it can be erroneously suspected by any correct actors.

To formalize these features we add three transition rules:

$$ {}^{b}\mathbf{0} \mid {}^{a}[ping(b,x).P]_s \xrightarrow{\tau} {}^{b}\mathbf{0} \mid {}^{a}[P\{false/x\}]_s \tag{7} $$

$$ {}^{b}\mathbb{C}_s \mid {}^{a}[ping(b,x).P]_s \xrightarrow{\tau} {}^{b}\mathbb{C}_s \mid {}^{a}[P\{true/x\}]_s \tag{8} $$

$$ {}^{b}\mathbb{C}_s \mid {}^{a}[ping(b,x).P]_s \xrightarrow{\tau} {}^{b}\mathbb{C}_s \mid {}^{a}[P\{false/x\}]_s \tag{9} $$

For the sake of readability we denote a correct actor by ${}^{b}\mathbb{C}_s = {}^{b}C_s$ or ${}^{b}[P]_s$, which means that the actor a is idle or active but not faulty. The first rule ensures that if b is really crashed (${}^{b}\mathbf{0}$), then *ping* detects the failure (the variable x assumes the value *false*). Observe that the transition does not change the behavior of the faulty actor, which remains crashed. The second and the third rules implement the unreliable behavior of the primitive *ping*. If rule (8) fires the behavior is correct and *ping* binds x to *true*. Otherwise, if rule (9) fires then *ping* binds x to *false*, that is, it detects a faulty actor even if this actor is not really crashed but only very slow[1]. In summary, the labelled transition system of the extended algebra is obtained adding the rules in Table 2 to the ones of Table 1.

4.3. Modeling an unreliable failure detector

An unreliable failure detector can be modeled in the actor algebra using the *ping* primitive. Typically a failure detector is a distributed program: each site in a distributed system has its own failure detector module.

In the algebra of actors if we consider a system composed of a fixed set of actors $a^1, a^2, \ldots, a^n$, we can specify a distributed failure detector as a set of local detector-actors $a_d^1, a_d^2, \ldots, a_d^n$ which run the same actor program. The state of a local detector consists of a pair $(dnames, failures)$, where *dnames* is the list of all the detector-actors in the system

[1] In the algebra of actors we have no formal notion of time and thus we cannot formalize the concept of "actor (or system) slow". However, we consider only asynchronous systems and we know that, in such systems, a process may appear failed because it is slow or because the network connection to it is slow. In our proposal we are not interested in formalizing the concept of "slow", but only in modeling this property of uncertainty.

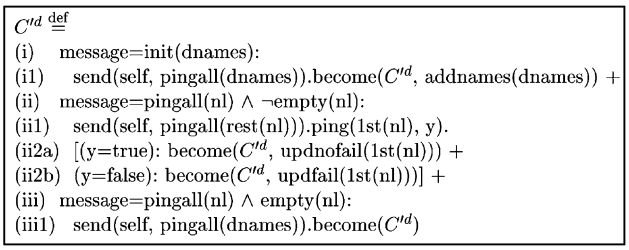

Fig. 2. A simple failure detector service.

and *failures* is the list of suspected actors. We also assume that a^i has crashed $\Leftrightarrow$ a_d^i has crashed. The behavior of a detector–actor is shown in Fig. 2.

In order to provide a more compact notation we use the following functions which operate on the state of the detector:

- *addnames(dlist)*: updates the field $failures$ of the state with *dlist*.
- *updfail*(b_d): adds the actor b_d to the list *failures*.
- *updnofail*(b_d): removes the actor b_d from the list *failures*.

When a detector a_d^i receives an initialization message from the actor a^i (i) it stores in its state the list of actors to be checked (*dnames*) (i1). Then the detector starts checking all the actors in the system by means of the primitive *ping* (ii1). Since in the algebra of actors there are no loop statements, we implement these checks by means of the following behavior: an actor sends to itself a message (*pingall*) with content the list of actors to be checked (*dnames*) (i1); when it receives this message (ii and iii), it checks the first actor of the list ($1st(nl)$) and it sends to itself the rest of the list (ii1). Observe that after each execution of $ping$, the detector updates the list *failures* according to the result of the check (which is stored in y) (ii2a and ii2b). The detector executes this program forever (iii): after the check of the last actor in the list *dnames*, the detector restarts its work with the first actor of the list. This is made by means of the message *pingall(dnames)* in (iii1).

Though the program can receive *init* messages more than once, we are assuming that only one *init* message is sent to it. Moreover, we do not model a termination protocol. The main reason for these choices is for the sake of simplicity and readability of the actor program. In fact, it is very simple to modify the program in order to restrict the detector to receive only one *init* message and to terminate in accordance with a particular protocol. However we think these adjustments are out of the scope of the paper and thus, in our opinion, they only complicate the actor program and its understanding.

Let us show that the above failure detector satisfies the following two properties.

Property 1. *If an actor b really crashes, then it is permanently suspected by every correct actor.*

Proof. In order to prove this property, we proceed as follows. First (a) we show that a really crashed actor is suspected by every correct actor. Then (b) we show that this actor is also *permanently* suspected by every correct actor.

(a) If an actor b really crashes, then it becomes a faulty actor ${}^b\mathbf{0}$ by means of a *Crash* transition. If a detector, say a_d, later checks the actor b using the $ping(b, y)$ primitive (ii1), then it discovers the failure because the transition $Ping1$ fires (and no other transitions can

fire). As a consequence, the variable y is bound to *false* and the failure detector updates its state adding the failure to the list *failures* (ii2b).

(b) We have to show that no transition can cancel actors from the list *failures*. An actor has only one way to remove an actor b from the list *failures*: to execute the primitive *become* in (ii2a). This is possible only if $y = true$, that is, if the execution of *ping* in (ii1) has bound y to *true*. But this is impossible because b is really crashed (for hypothesis) and thus *ping* can only bound y to $false$ (rule *Ping1* in the LTS). □

Property 2. *If an actor b is correct, then it can be erroneously suspected by any correct actors.*

Proof. This property follows directly from the unreliable behavior of the primitive *ping*. If an actor b is correct, it can be erroneously suspected by a detector, say a_d, which executes $ping(b, y)$ (ii1) and the transition *Ping*3 fires. Consequently the detector updates its state adding the actor b to its list of suspects (ii2b). Note that if the detector later discovers the actor is correct, then it updates its state removing b from the list *failures* (ii2a). Thus at any given time a detector can correct its mistakes. □

5. A case study: an agent architecture for anonymous interaction

The algebra of actors we have presented above is a powerful tool for specifying software architectures. A software architecture is described specifying its main components as actors and connecting them. Given this specification, the properties of the architecture can be discussed and proved at the architectural level. We illustrate this process by means of a case study: the design of an agent architecture for supporting anonymous interaction. Agents provide services to the outside world and request services to other agents. Following the style of [18], we assume a description of agents based on two levels: a **knowledge level** which focuses on agents' competences and abstracts from implementation details, and a **symbol level** (our architectural level) which specifies how these competences are realized. This approach has several advantages which allows us to illustrate the utility and the expressive power of our formalism. Firstly, there is a clear distinction between the knowledge level and the architectural level. Agents are abstract entities which operate at the knowledge level executing high level communication primitives, while the architectural level concerns actors and their interaction, synchronization and management of failures. Secondly, it's possible to define a set of requirements that should be satisfied at the agent level and that can be proved at the architectural level. Finally, it is not trivial to provide a direct operational characterization of the agent behavior (for example defining an ad hoc transition system for agents) well integrated with standard mechanisms to model communication and concurrency in process algebra (for example handshake communication). This approach was focused in a previous work of one of the authors where a specific transition system was defined and successfully exploited to give an operational definition of a simple agent communication language [19]. However we have realized that this approach does not scale when we need to model more complex communication primitives. In fact, it is difficult to integrate standard low level communication and concurrency mechanisms, which are necessary to describe the real behavior of a system, in a transition system describing high level multi-party communication primitives, providing a tractable formalism. The

reader interested in this issue will find in [18] an extended discussion on the motivations of using an actor algebra for this class of architectures.

5.1. Knowledge level description

An agent in the system has a symbolic (logical) name and a *virtual knowledge base* (VKB), which is a set of first order formulas. Agents only interact with requests of knowledge which is stored in VKBs of other agents. The requests of knowledge are anonymous and independent from low level issues, such as management of agent names, routing and agent reachability.

Let $\mathcal{A}_{ACL}$ be a countable set of agent names ranged over by $\hat{a}, \hat{b}, \hat{c}, \ldots$. Let $VKB_{\hat{a}}$ be the virtual knowledge base of agent $\hat{a}$ and $p, p', p'', \ldots$ first order formulas.

The anonymous interaction protocol which we study is realized by means of the following two agent primitives:

- *ask-everybody*$(\hat{a}, p)$ asks all agents interested in p for an instantiation of p which is true in their VKB.[2]
- *all-answers*(p) allows to know if all the replies concerning the proposition p have been received.

These primitives should satisfy the following specification requirements.

1 Knowledge-level programming requirements. The notion of *knowledge-level* in the context of multi-agent systems has been discussed in detail by Gaspari in [18]. Following that approach, we require a knowledge-level model for agents: that is, they should provide communication primitives which support the use, request and supply of knowledge independently from implementation-related aspects. Syntactically both *ask-everybody* and *all-answers* can be considered at the knowledge-level, since both have propositional contents. However, this is not enough to guarantee a correct knowledge-level behavior. In [18] additional conditions are postulated which require an accurate specification of the underlying agent architecture in order to ensure knowledge-level behavior. We recall these conditions below.

(1) The programmer should not have to handle physical addresses of agents explicitly.
(2) The programmer should not have to handle (2.1) communication faults and (2.2) agent crashes explicitly.[3]
(3) The programmer should not have to handle starvation issues explicitly. A situation of starvation arises when an agent's primitive never gets executed despite being enabled.
(4) The programmer should not have to handle communication deadlocks explicitly. A communication deadlock situation occurs when two agents try to communicate, but they do not succeed; for instance because they mutually wait for each other to answer a query [22].

2 Open system requirement. In the research on multi-agent systems there is an increasing emphasis on the open-ended nature of agent systems, which refers to the feature of supporting the dynamic integration of new agents into an existing agent system. The anonymous

[2] Note that the *ask-everybody* primitive includes the name of the sender agent. This is necessary because the recipient agents need to know the name of the sender to send it the replay. As discussed in [18] agents modeling real world situations in many cases need to know the name and the beliefs of a partner agent in order to perform an agent-to-agent interaction. This name is usually included in the primitives of Agent Communication Languages [20,21].

[3] Condition (2.2) has been added here to consider possible crash failures.

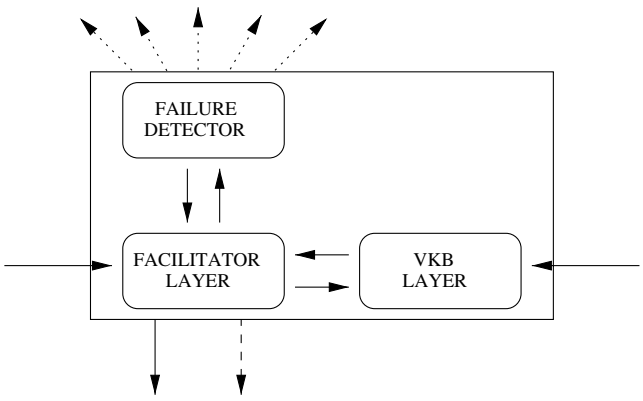

Fig. 3. Fault tolerant agent architecture for anonymous interaction.

interaction protocol should function correctly in open multi-agent systems where agents are added dynamically to cooperate with the existing ones. To this purpose we establish the following requirement:[4]

(5) Only registered agents are involved in the anonymous interaction protocol. This means that only registered agents can be reached through the *ask-everybody* primitive.

The informal definition of *ask-everybody* and *all-answers* we have given above is not sufficient to show that these requirements hold. To achieve this goal we have to specify the details of the concurrent behavior of these primitives and of the underlying agent architecture.

5.2. Architectural level description

Each agent has a knowledge-level (KL) component which implements the VKB of the agent and its reactive behavior (see Fig. 3)). This component only deals with knowledge-level operations and it is able to answer requests from other agents. To realize the anonymous interaction protocol we exploit a *distributed facilitator service* which is hidden at the knowledge-level and provides mechanisms for registering capabilities of agents and delivering messages to the recipient actors.

Facilitators are distributed and encapsulated in the architecture of agents. Each agent has its own *local facilitator* component which executes a distributed algorithm: it forwards control information to all the other local facilitators, and delivers messages to their destinations. Since the facilitators are encapsulated in the agent architecture, they are not visible at the knowledge-level. Therefore, although facilitators deal with some low-level issues, we do not violate our knowledge-level requirement.

An additional difficulty which the specification of the anonymous interaction protocol should take into account is that multi-agent systems are prone to the same failures that can occur in any distributed software system. An agent may become suddenly unavailable due to various reasons. The agent may die due to unexpected conditions, improper handling of exceptions and other bugs in the agent program or in the supporting environment. The

[4] Of course, many other requirements can be discussed for open multi-agent systems. However, here we prefer to focus only on the requirement which we consider the most interesting in the context of the paper.

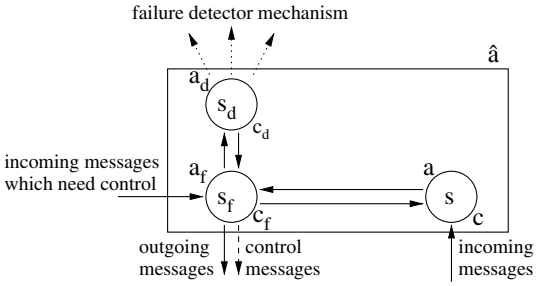

Fig. 4. Actor-based agent architecture.

machine on which the agent process is running may crash due to hardware and software faults.

Since we have postulated that to ensure knowledge-level programming the programmer should not have to handle agent crashes and communication faults explicitly, it is important to guarantee that the *ask-everybody* and *all-answers* primitives are fault tolerant in some way. In order to address this issue we provide a *failure detector component* which is also encapsulated in the agent architecture (Fig. 3). The aim of this component is to check all the agents in the system trying to discover the ones which have crashed.

Observe that this is a generic agent architecture: the failure detector and the facilitator components are standard for all the agents in a multi-agent system, while the KL component can be instantiated with different VKB.

This architecture is formally specified in the actor algebra. The architecture of an agent is illustrated in Fig. 4. An agent $\hat{a}$ is composed of three actors (a, a_f, a_d) which run in parallel $(a \mid a_f \mid a_d)$ and which implement the behavior of the components of the general agent architecture (Fig. 3):

- **kb-actor** a: this actor implements the agent at the knowledge-level, it contains the VKB of the agent $\hat{a}$.
- **facilitator–actor** a_f: this actor implements the distributed facilitator mechanism and the anonymous interaction protocol.
- **detector–actor** a_d: this actor implements the distributed failure detector mechanism. It monitors all the agents in the system and individuates those that it currently suspects to have crashed.

We assume a simple mapping between the logical names of agents and the physical names of actors. Given an agent name $\hat{a}$, the corresponding physical name of the kb-actor is obtained removing the hat (thus it is a), its facilitator–actor is a_f and its detector–actor is a_d. This mapping is known by all the facilitators. In a more general architecture the translation between logical and physical names of agents can be embedded in the facilitator process. In general, incoming messages are handled by the kb-actor, but if incoming messages need some control operations then they are sent through the facilitator layer. The facilitator–actor deals with the outgoing messages and it also receives control information from other facilitators. The detector–actor implements the local *unreliable* failure detector mechanism: it checks all the agents in the system and it manages the list of suspected agents.

A formal specification of an unreliable failure detector component has been presented in Section 4.3. In the following we describe the kb-actor and the facilitator components and then we discuss their integration with the failure detector component.

5.2.1. Kb-actor component

The kb-actor implements the knowledge-level agent which performs requests for knowledge and it is able to answer requests from other agents. The kb-actor is realized exploiting the guarded program of the actor algebra to implement its reactive behavior:

$$C^a \stackrel{\text{def}}{=} e_1 : P_1 + \cdots + e_n : P_n. \tag{10}$$

All the outgoing primitives are sent to the facilitator. We provide here the encoding of the *ask-everybody* and *all-answers* primitives which are relevant for the scope of this paper. The reader interested to the full specification of a kb-actor can refer to [18,23].

An agent $\hat{a}$ uses the *ask-everybody*$(\hat{a}, p)$ primitive to ask all agents interested in p for an instantiation of p which is true in their VKB. Subsequently, $\hat{a}$ can execute the *all-answers*(p) primitive to know if all the replies concerning the proposition p have been received.

The two agent primitives are translated into actor messages sent from the kb-actor a to the facilitator a_f, as follows:[5]

$$[\![\textit{ask-everybody}(\hat{a}, p)]\!]^a = \textit{send}(a_f, \textit{ask-everybody}(\hat{a}, [\![p]\!]^a))$$

$$[\![\textit{all-answers}(p)]\!]^a = \textit{send}(a_f, \textit{allanswers}(\hat{a}, p)).\textit{become}(C'^a)$$

where $C'^a \stackrel{\text{def}}{=}$

(i) message=allanswersyes: $[\![\textit{Rest of the program}]\!]$ +
(ii) message=allanswersno: become(C^a) +
(iii) *otherwise*: send(self, message).become(C'^a)

where *otherwise* (iii) is a boolean expression which is true if all the previous guards are false (it is defined more formally in [15]).

We briefly discuss the encoding of *all-answers*. This primitive is implemented by sending a request to the local facilitator–actor and waiting for its answer. If the kb-actor receives a positive answer (i) then it continues the execution of the rest of the program which follows the *all-answers* call; otherwise (ii) it stops the program C'^a and starts waiting for other messages (become(C^a)). Note that while the kb-actor waits for the answer to the *all-answers* query, all the other messages are delayed (iii).

5.2.2. Facilitator component

The distributed facilitator is formally specified as a dynamic set of local facilitator actors $\{a_f, a'_f, a''_f, \ldots\}$ which run (in parallel) the same actor program. This set may evolve dynamically whenever a new agent is created or an agent terminates its computation.

The state of a local facilitator is a triple (*fnames*, *competence*, *answers*), where *fnames* is the list of all the local facilitators, *competence* is a data structure which stores the competence of the agents in the system and *answers* contains information on the active conversations involving multicasting. In Fig. 5 we present a specification of a facilitator service which supports the *ask-everybody* and *all-answers* primitives. The specification of a complete facilitator program for a knowledge-level Agent Communication Language can be found in [23].

[5] The encoding is defined by means of a function $[\![Ag]\!]$ which translates agent terms into actor terms. We use the notation $[\![Ag]\!]^a$ when we translate an agent $\hat{a}$ (see [18] for more details on this translation function).

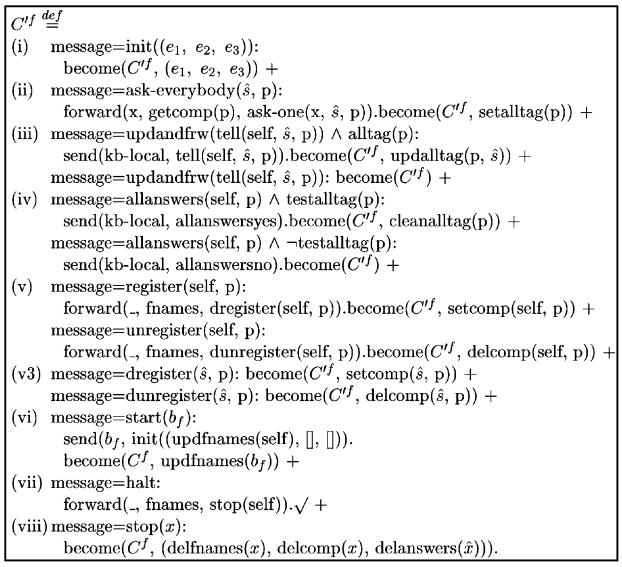

Fig. 5. A facilitator service which supports the *ask-everybody* and *all-answers(p)* agent primitives.

When a facilitator receives an *init* message from a kb-actor (i), it updates its own state by means of a *become* primitive. This initialization depends on the parameters (e_1, e_2, e_3) received in the *init* message. When a facilitator receives an *ask-everybody*$(\hat{s}, p)$ message, it consults its database (the field *competence* of its state) by means of the function *getcomp*(p). According to the information retrieved from its database, the facilitator forwards an *ask-one* message to all the agents interested in p (ii). The program (iii) deals with all the replies concerning p: when a facilitator receives the message *updandfrw*(*tell*(*self*, $\hat{s}$, p)) (which stores the information that the agent $\hat{s}$ is able to deal the proposition p) and it is waiting this answer (function *alltag*(p)), then it sends this information to the kb-actor and updates its own state. The program (iii) is needed to implement the *all-answers* primitive. Indeed, when a facilitator receives the message *all-answers*(*self*, p) it checks if all the replies concerning proposition p have been received and then communicates the result of this control to the local kb-actor (iv). The *register*(*self*, p) and *unregister*(*self*, p) messages are forwarded to all the other facilitators in the system (v). When a facilitator receives these messages from another facilitator it updates the *competence* data structure (v3). The protocol in (vi) supports the dynamic creation of new agents. The protocol in (vii) and (viii) supports the termination of a facilitator. When a facilitator receives a *halt* message (vii) it forwards a *stop* message to all the facilitators to communicate its own termination. Then the facilitator terminates. When another facilitator receives the *stop* message (viii), it updates its own state removing the name of the sender (in the program: when the facilitator receives the *stop* message the variable x is bound to this name). Table 3 summarizes the functions which operate on the fields of the state of the facilitator.

In order to forward a message the facilitator uses the *forward* primitive which implements a multicast interaction mechanism. We introduce it because the algebra of actors does not provide an explicit primitive for forwarding a message to a set of known actors. In the following we show that this explicit *forward* primitive can be simply implemented in the algebra. In order to prove this, we extend the algebra with a new primitive *forward*(x, nl, m) which allows actors to forward a message m to all the actors in a list

Table 3
Functions which operate on the state of the facilitator

Function	Operates on	Description
$getcomp(p)$	*competence*	Retrieves the list of all the agents which are able to deal with proposition p and returns the list of the associated facilitators
$alltag(p)$	*answers*	Returns true if an alltag on p has been set
$setalltag(p)$	*answers*	Returns a new facilitator state where the field $answers$ includes a record with the query p and the list of all the agents which have this competence
$updalltag(p, \hat{a})$	*answers*	Returns a new facilitator state where $answers$ contains the fact that a reply concerning proposition p has been received
$testalltag(p)$	*answers*	Returns true if all the replies concerning proposition p have been received
$cleanalltag(p)$	*answers*	Returns a new facilitator state where the tags concerning proposition p have been removed
$deltag(p, \hat{b})$	*answers*	Removes agent $\hat{b}$ from the list of agents which have to answer about p. To remove a *list* of agents we can use $deltag(p, list)$
$setcomp(\hat{a}, p)$	*competence*	Adds to the competence data structure the information that agent $\hat{a}$ is able to deal with proposition p
$delcomp(\hat{a}, p)$	*competence*	Removes agent $\hat{a}$ from the competence list of p

nl. If the variable x is in m then it will be instantiated with the elements of nl. Hence, we extend the syntax by allowing also:

$$P ::= forward(x, nl, m).P$$

and the operational semantics by adding the axiom:

$$^{a}[forward(x, nl, m).P]_s \xrightarrow{\tau} {}^{a}[P]_s \mid \langle a_1, m\{a_1/x\}\rangle \mid \ldots \mid \langle a_{nl}, m\{a_{nl}/x\}\rangle$$

Our idea for implementing the program $forward(x, nl, m).P$ in a term of the algebra $[\![forward(x, nl, m).P]\!]$ is to create a new actor whose behavior (FWD) is to execute the forward of a message (see Fig. 6). When this actor finishes to send all the messages it terminates (correctly) its execution (i.e. becomes the empty term **0**). Formally:

$$[\![forward(x, nl, m).P]\!] \stackrel{\text{def}}{=} \text{create(d, FWD, []).send(d, frw(x, nl, m))}.P$$

where d is fresh and

$$\text{FWD} \stackrel{\text{def}}{=}$$

$$\text{message=frw(x, nl, m)} \wedge \neg\text{empty(nl)} :$$
$$\quad \text{send(self, frw(x, rest(nl), m)).send(1st(nl), m[1}\hat{st}(nl)\text{/x]).}$$
$$\quad \text{become(FWD)} \; +$$
$$\text{message=frw(\_, nl, m)} : \surd$$

5.2.3. Integrating the components of the architecture

The components of the agent architecture previously discussed have been specified individually and their integration is not always trivial. The integration of the kb-actor and the facilitator–actor is simple and requires just an agreement on their respective names. In Section 5.2 we have already bound the kb-actor to the facilitator because in the encoding of the primitives the kb-actor a sends the messages to the local facilitator a_f. Therefore, in order to integrate these two components we only need to bind the facilitator with the kb-

actor. This can be done substituting the expression *kb-local* in the facilitator program with the name of the local kb-actor a. The new behavior C''^f of the facilitator-actor is formally defined as

$$C''^f \stackrel{\text{def}}{=} C'^f[\, a/\textit{kb-local}\,]$$

which means that the program C''^f is exactly the program C'^f where each occurrence of the expression *kb-local* is substituted with a, i.e. with the local kb-actor name.

The integration of the facilitator component with the unreliable failure detector is more complex and requires a careful analysis and more complex connectors. A critical point of the facilitator program which is related to failures is the implementation of the *all-answers* primitive (see the facilitator program in Fig. 5 line (iv)). When a facilitator receives an *allanswers* message from the kb-actor and an answer from an agent interested in p is still not arrived, it is possible that the agent has crashed. This means that the *all-answers* predicate will never succeed (because *testalltag*(p) will be always false and thus the facilitator will always reply *allanswerno* to the kb-actor). Thus whenever such a situation arises it is reasonable to contact the failure detector component to verify whether that agent is suspected to have crashed or not.

A possible solution to this problem can be obtained contacting the failure detector component and asking it for the list of suspected actors. Unfortunately this solution requires a synchronization which may slow down the performance of the facilitator component. In fact, it should wait for the list of suspected agents before answering to other queries.

The solution that we have adopted exports part of the state of the failure detector component to the facilitator: the list *failures* which contains the suspected agents. The new failure detector just notifies the results of its checks to the facilitator. The state of the new local detector consists only of a list *dnames*, which is the list of all the detector actors in the system, while the list $failures$ is exported to the facilitator.

Since the new failure detector is part of the agent architecture, we also have the following assumptions:

- a_f has crashed $\Leftrightarrow$ a has crashed $\Leftrightarrow$ a_d has crashed $\Leftrightarrow$ $\hat{a}$ has crashed; thus we can say that "a detector checks the agent $\hat{a}$ in the system" to mean "a detector checks the detector a_d of the agent $\hat{a}$".
- The communication between a kb-actor a, a facilitator-actor a_f and a detector–actor a_d is reliable.

The behavior of a detector-actor when integrated in the agent architecture is shown in Fig. 7.

In a similar way to the integration between the kb-actor and the facilitator, we also need to insert the name of the facilitator in the program C''^d and the name of the failure detector in the program C''^f. This can be done by means of the following substitutions:

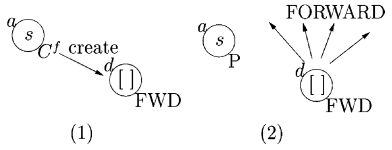

Fig. 6. THE FORWARD PRIMITIVE. (1) In order to forward a message a new actor d is created. (2) The behavior of this new actor is to execute the forward of a message. When it finishes to send all the messages it terminates correctly its execution. Note that in the meantime the actor a does not suspend its execution but it continues the program P.

$C''^{d} \stackrel{\text{def}}{=}$
(i) message=init(dnames):
send(self, pingall(x, dnames)).become(C''^{d}, addnames(dnames)) +
(ii) message=pingall(x, nl) ∧ ¬empty(nl):
send(self, pingall(x, rest(nl))).ping(1st(nl), y).
[(y=true): become(C''^{d}).**send(fac-local, updnofail(1st(nl)))** +
otherwise: become(C''^{d}).**send(fac-local, updfail(1st(nl)))**] +
message=pingall(x, nl) ∧ empty(nl):
send(self, pingall(x, dnames)).become(C''^{d}) +
(iii) **message=addname(d): become(C''^{d}, addnames(d))** +
(iv) **message=delname(d): become(C''^{d}, delnames(d))** +
(v) **message=halt: √**

Fig. 7. The program of the failure detector actor integrated in the agent architecture. Note that this program is similar to the one in Fig. 2 except for the lines presented in bold.

$$C^{f} \stackrel{\text{def}}{=} C''^{f}[\, a_d/\textit{fd-local}\,] \quad C^{d} \stackrel{\text{def}}{=} C''^{d}[\, a_f/\textit{fac-local}\,]$$

When a detector receives an initialization message from the local facilitator (i) it starts checking all the agents in the system (ii). The result of each check is sent to the local facilitator to update the list *failures*. The detector executes this program until it receives a *halt* message from the local facilitator. If this event occurs then the detector stops forever its execution (v).

Note that the state of the detector is dynamically updated when a new agent is created and when an existing agent terminates its computation. Indeed, when a detector receives a message *addname*(d) or *delname*(d) from the local facilitator it updates its state adding

$C^{f} \stackrel{def}{=}$
(i) message=init((e_1, e_2, e_3, e_4)):
send(a_d, init(e_1)).
become(C^{f}, (e_1, e_2, e_3, e_4)) +
(ii) message=ask-everybody(s, p):
forward(x, getcomp(p), ask-one(x, s, p)).become(C^{f}, setalltag(p)) +
(iii) message=updandfrw(tell(self, s, p)) ∧ alltag(p):
send(a, tell(self, s, p)).become(C^{f}, updalltag(p, s)) +
message=updandfrw(tell(self, s, p)) ∧ firsttag(p):
send(a, tell(self, s, p)).become(C^{f}, delfirsttag(p)) +
message=updandfrw(tell(self, s, p)): become(C^{f}) +
(iv) message=allanswers(self, p) ∧ testalltag(p):
send(a, allanswersyes).become(C^{f}, cleanalltag(p)) +
(iv2) message=allanswers(self, p) ∧ ¬testalltag(p):
send(a, allanswersno).**become(C^{f}, deltag(p, failures))** +
(v) message=register(self, p):
forward(_ , fnames, dregister(self, p)).become(C^{f}, setcomp(self, p)) +
message=unregister(self, p):
forward(_ , fnames, dunregister(self, p)).become(C^{f}, delcomp(self, p)) +
message=dregister(s, p): become(C^{f}, setcomp(s, p)) +
message=dunregister(s, p): become(C^{f}, delcomp(s, p)) +
(vi) message=start(b_f):
send(b_f, init((updfnames(self), [], [], **failures**)).
send(a_d, addname(b_d)).
become(C^{f}, updfnames(b_f)) +
(vii) message=halt:
forward(_, fnames, stop(self)).**send(a_d, halt).**√ +
(viii) message=stop(x):
send(a_d, delname(x)).
become(C^{f}, (delfnames(x), delcomp(x), delanswers($\hat{x}$))). +
(ix) **message=updfail(y): become(C^{f}, addfail(y))** +
message=updnofail(y): become(C^{f}, remfail(y))

Fig. 8. Integrating the facilitator program with the failure detector and the kb-actor. Note that all the occurrences of the expressions *kb-local* and *fd-local* have been replaced by a and a_d respectively.

Table 4
Functions which operate on the field $failures$ of the state of the facilitator

Function	Operates on	Description
addfail(*b*)	*failures*	Adds the actor *b* to the list
remfail(*b*)	*failures*	Removes *b* if it is in the list

(iii) or deleting (iv) d to/from the list $dnames$ respectively. This functionality has been added to ensure to support dynamic creation of new agents.

The second step of this integration is the extension of the facilitator program to include the list of suspected agents. In Fig. 8 we show the new facilitator program C^f obtained after the integration of the facilitator component with the kb-actor and the failure detector. Note that the properties of Section 4.3 are still satisfied by this new failure detector.

Note that all the occurrences of the expressions *kb-local* and *fd-local* have been replaced by the name of the local kb-actor a and the name of the local failure detector a_d respectively. This program is analogous to that presented in Fig. 5 except for the lines presented in bold. The function *deltag*(p, *failures*) updates the tags related to the query p canceling all the suspected agents. Thus, if an answer is still not arrived, but the agent which has to reply is suspected by the failure detector, then that answer is ignored and the *all-answers* primitive succeeds. The behavior of the facilitator is extended with the protocol in (ix) to deal with messages from the failure detector component and update the list of suspected actors. In Table 4 the functions which update the *failures* field of the state are summarized.

6. Analysis of requirements

The aim of this section is to show that our specification satisfies the requirements discussed in Section 5 at the architectural level. To address this issue we proceed as follows. First, in Section 6.1, we focus on knowledge-level programming requirements. Then, in Section 6.2, we focus on the open system requirement. We think that this analysis shows even more the benefits of distinguishing between a knowledge level description from an architectural level where we can reason about the (concurrent) behavior of the components of the agent's architecture.

6.1. Knowledge-level programming requirements

In the following theorem we prove that our architectural specification based on the algebra of actors satisfies the knowledge-level requirements (1), (2), (3) and (4) of Section 5.1. Since in [18] a similar Theorem is already proved for multi-agent systems in which no failures can occur, here we focus only on fault-tolerant issues of the Theorem.

Theorem 3 (Knowledge-level requirements). *The actor based specification of the agent architecture and of the anonymous interaction primitives satisfies the requirements for knowledge-level programming.*

Proof. Requirements (1), (2.1) and (3) are not influenced by crash failures of agents and thus we omit the complete proof because is the same as the one in [18].

Requirement (2.2) is trivially satisfied. Indeed, to guarantee that the interaction primitives are fault tolerant we provide a failure detector component. This component is encapsulated in the agent architecture and is hidden at the knowledge-level.

Requirement (4) is not trivial and requires a more careful analysis. Also this requirement is proved in [18] for system without crashes and thus here we focus only on fault tolerant issues. If we admit crashes, then a communication deadlock can occur when a correct agent waits for answers of crashed agents. Consider a set $\mathbb{S}$ of m agents $\hat{a}_1, \hat{a}_2, \ldots, \hat{a}_m$ and suppose without loss of generality that the agent $\hat{a}_i \in \mathbb{S}$ is waiting for replies about a proposition p from a subset $\overline{\mathbb{S}} \subset \mathbb{S}$ which has cardinality n, $1 \leqslant n \leqslant m-1$, and which does not include $\hat{a}_i$. To avoid a communication deadlock, the agent must wait only for answers of *correct* agents and must continue the execution of its program after it has received all these answers. Our architecture satisfies these properties as stated by the following two Lemmas.

Lemma 4. *An agent which executes the all-answers*(p) *primitive does not wait for replies of crashed agents forever.*

Proof. We have to prove that $\hat{a}_i$ does not wait the replies forever, even if some (or all the) agents in $\overline{\mathbb{S}}$ become crashed before to reply. This result holds. In fact, whenever an agent $\hat{a}_k \in \overline{\mathbb{S}}$ crashes, sooner or later the crash will be discovered and $\hat{a}_i$ will not wait for that answer anymore. This agent behavior follows directly from Property 1 of the failure detector component proved in Section 4.3. Indeed, when the detector a_{i_d} checks a_k executing $ping(a_k, y)$ ((ii) in Fig. 7), it discovers the crash because of the transition:

$$^{a_k}\mathbf{0} \mid {}^{a_{i_d}}[ping(a_k, y).P]_s \xrightarrow{\tau} {}^{a_k}\mathbf{0} \mid {}^{a_{i_d}}[P\{false/y\}]_s$$

Then the detector executes the branch *otherwise* of its program and communicate the crash to the facilitator by means of the $updfail(a_k)$ message. When the facilitator receives this message, it updates the list *failures* adding a_k ((ix) in the facilitator program of Fig. 8):

$$\text{message=updfail}(a_k) : \text{become}(C^f, \text{addfail}(a_k)) +$$

Therefore we are sure that sooner or later the facilitator will remove $\hat{a_k}$ from the list of agents which have to reply about p. Thus we have proved that $\hat{a}_i$ does not wait replies of crashed agents forever. □

Lemma 5. *If an agent has received all the replies of correct agents, then it is able to continue the execution of its program.*

Proof. Consider the encoding of the agent primitive *all-answers*(p) (executed by $\hat{a}_i$) into the algebra of actors:

$$[\![all\text{-}answers(p)]\!]^{a_i} = send(a_{i_f}, allanswers(\hat{a}_i, p)).become(C'^{a_i})$$

$$C'^{a_i} \stackrel{\text{def}}{=}$$
$$\text{message=allanswersyes: } [\![Rest\ of\ the\ program]\!] +$$
$$\text{message=allanswersno: become}(C^{a_i}) +$$
$$otherwise\text{: send(self, message).become}(C'^{a_i})$$

The kb-actor sends the message *all-answers*$(\hat{a}_i, p)$ to the local facilitator a_{i_f} and then it blocks the execution of the rest of the program until it receives the *allanswersyes* message

from a_{i_f}. So we have to prove that sooner or later the kb-actor will receive this message. To show this, consider the behaviour of the facilitator (program in Fig. 8):

$C^f \stackrel{\text{def}}{=} \ldots$

(iv) message=allanswers(self, p) $\wedge$ testalltag(p):
send(a, allanswersyes).become(C^f, cleanalltag(p)) +

(iv2) message=allanswers(self, p) $\wedge \neg$testalltag(p):
send(a, allanswersno).become(C^f, deltag(p, failures)) +

$\ldots$

The facilitator sends the *allanswersyes* message if and only if *testalltag(p)* returns $true$ (iv), which means that all the agents which are not crashed or suspected have replied. Thus there is to prove that sooner or later the predicate *testalltag(p)* returns *true*. This predicate returns *true* if and only if

(P1) all the replies of correct agents have been received AND
(P2) all the replies of crashed agents are not waited forever.

Property (P1) is satisfied for hypothesis (a proof can be found in [18]). Property (P2) is satisfied thanks to the previous lemma. In fact, if all the replies of correct agents have been received but *testalltag*(p) is still false because of crashed agents, we are sure that sooner or later all the crashes will be discovered and the predicate will return *true*. To show this, consider the above facilitator program. If *testalltag*(p) is *false* (iv2) then the facilitator uses the predicate *deltag*(p, *failures*) to remove all the agents in *failures* from the list of agents which have to reply about p. The list *failures* is dymanically and continuously updated by the local detector a_{id} and the previous Lemma assures us that sooner or later the crashed agents in $\overline{\mathbb{S}}$ will be added to *failures*. Therefore, whenever all crashed agents are discovered the predicate *testalltag*(p) returns *true* and thus the facilitator can reply *allanswersyes* to the local kb-actor a_i. □

Thanks to the above lemmas, our architecture satisfies the requirement (4) also for multi-agent systems in which agents can crash. Since all the knowledge-level requirements have been satisfied, the theorem is proved. □

6.2. Open system requirement

In this section we show that our agent's architecture supports the open system requirement which establish that only registered agents can be reached through the anonymous interaction protocol.

Suppose that an agent $\hat{a}$ is able to execute tasks which take a long time to solve. An efficient and intelligent behaviour of that agent would be the following: $\hat{a}$ does not handle directly the queries sent from other agents, but instead it creates new *working agents* which serve the requests in place of it. This is a reasonable situation: the role of agent $\hat{a}$ could be to filter the incoming requests and distribute them to the most adequate working agents. Although they are based on the same architecture, working agents must not be directly reachable through the anonymous interaction protocol because agent $\hat{a}$ should be the only responsible of the tasks allocation policy. In fact, working agents cannot explicitly

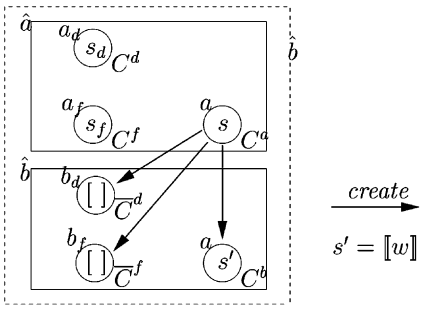

Fig. 9. Dynamic agent creation. The dashed line shows the scope of the name of the created agent $\hat{b}$: the new agent is accessible (known) only by its creator $\hat{a}$.

register their competence to the distributed facilitator because both their local facilitator and detector components are not initialized.

Assume to have defined an agent primitive $create(\hat{b}, w)$ which creates a new working agent with a new *fresh* name $\hat{b}$ and a new VKB w. We require a *fresh* name because we want to ensure that, at the time of creation, the name of the new agent is known only by the agent that creates it. If we suppose that the primitive is executed by an agent $\hat{a}$, as shown in Fig. 9, we can specify its behaviour in the actor algebra as follows:

$$[\![create(\hat{b}, w)]\!]^a = create(b_f,\ \overline{C}^f,\ []).create(b,\ C^b,\ [\![w]\!]).create(b_d,\ \overline{C}^d,\ [])$$

where $\overline{C}^f \stackrel{\text{def}}{=} C^f[\ b_d/\ \text{fd-local}, b/\ \text{kb-local}\ ]$ and $\overline{C}^d \stackrel{\text{def}}{=} C^d[\ b_f/\ \text{fac-local}\ ]$.

The creation of a new working agent is translated into the algebra with the creation of all the actors which compose the architecture of an agent. Note that the integration of these components is realized by means of dynamic substitutions of names in the actor programs. Note also that the facilitator actor b_f is created but its state is empty. This means that it is not fully integrated with the distributed facilitator because it does not know the names of the other facilitators. These names are transmitted only if the facilitator is initialised ((i) in the program of Fig. 8). However the facilitator is able to perform all its standard functions despite being disconnected from the distributed facilitator mechanism.

Now we want to show that the anonymous interaction protocol is not able to reach the working agent. As mentioned above, the informal semantics of the agent primitive *create* requires that, at the time of creation, the name of the new agent is known only by the agent that creates it. We call this property *hiding name creation*. The following Theorem states that our encoding satisfies this property.

Theorem 6 (Hiding name creation). *The encoding of the agent primitive* ***create*** *into the actor algebra satisfies the* ***hiding name creation*** *property.*

Proof. The *hiding name creation* property follows directly from the semantics of the actor primitive *create* (Table 1). Indeed, this primitive allows to hide the name of a newly created actor by means of the restriction operator \. This operator ensures that the name of a newly created actor is not reachable from the external world, but only from the creating actor. In our encoding we create a new agent by means of three *create* actor primitives which build the components of an agent architecture. Therefore the only actors which are able to

communicate with these new ones are those of the creator agent. Then we are sure that the new agent is only accessible (known) by its creator. □

Thus a new working agent is unreachable from the external world (as stated in the theorem), it cannot receive messages sent from other agents. Therefore the only way for a working agent to receive a message sent by means of an *ask-everybody* primitive is to record its interests. This cannot be done by the working agent directly because the local facilitator is not integrated with the distributed facilitator. Only the creator agent can integrate the new agent in the anonymous interaction protocol. Here we do not show how this integration can be realized and how an agent can register its interests because these issues are out of the scope of the paper. However, readers interested in a formal specification of a *create* primitive which allows new agents to register their interests can find it in [23,24].

7. Related work

In the subfield of agent research that focuses on agent architectures, various types of agents have been proposed that facilitate the communication process in a multi-agent system. These agents, referred to with terms like *routers*, *mediators*, *brokers* and so on [25], act as intermediaries between communicating agents by providing some services. Such facilitating activities are indispensable in the context of open multi-agent systems and in particular in the context of knowledge-level communication. As mentioned before, in our approach we encapsulate a distributed facilitator mechanism in the agent architecture providing both facilitating services and knowledge-level communication.

Regarding the formalization of failures and fault tolerance, we do not know works that explicitly address the issue of formally modeling failures and failure detectors in an actor based approach. Instead, there are many works on the possibility of modeling faults and fault tolerance mechanisms using standard process algebras. A brief survey of the related literature is reported in [5]. Several of these proposals share with our approach the explicit modeling of a fault as a transition. For example, a first work on the specification and proof of a simple fault tolerant system in CCS is done by Prasad in [26]. With the work [27] a research line which focuses on providing a direct modeling of fault tolerance issues started. More precisely, the focus is modeling certain relevant aspects of systems such as the distribution of processes on different *locations*, the impact of failures on the behavior of the system and their detection. In [28] Amadio pursues and improves this research line modeling location failures and detection in a fragment of the asynchronous π-calculus. As in our approach, Amadio follows the work of Chandra and Toueg [17] and enriches its model with a failure detector *ping* which eventually allows any process to know if a location runs or not. The differences with our approach consists in the unit of failure and in its detection. In Amadio's paper locations are the units of failure: a location can fail, entailing the failure of all the processes running at it. Moreover, Amadio focuses on a *perfect* failure detector, that is a failure detector that cannot make mistakes (like an oracle). Other works that follows the line of research based on the concept of locations are [29, 30]. In [29] the authors (Riely and Hennessy) define a distributed language with location failures which has much in common with the language developed by Amadio. The main difference is that the language of Riely and Hennessy has no value-passing, allowing the authors to concentrate on the effects of location failures and simplifying the statement of some results.

In a recent work of Nestmann and Fuzzati [31] failure detectors are modeled using the operational semantics. The formal framework has been successfully used to model distributed consensus algorithms with a process calculus [32]. Following this technique a single failure detector is directly hardcoded in the transition system, while in our approach several failure detectors can be specified in the algebra using the ping primitive as a basic building block. The main advantages of our approach are generality and flexibility. Firstly, given a basic mechanism that allows actors to detect failures (for example our ping primitive) we can specify all the failure detectors which use this mechanism in a single formal framework. Secondly, a failure detector is not a black box component and can be tailored to the needs of a specific software system or architecture, as we have shown in Section 5. We claim that these features of our approach are particularly relevant if the purpose of the formalism is to specify software architectures rather than discussing theoretical issues.

8. Conclusions

We have presented an algebra of actors extended with mechanisms to model crash failures and their detection. We have shown that this algebra can be used to describe a fault tolerant software architecture specifying its main components as actors and connecting them. We have presented simple connectors which allows to integrate actor specifications of architectural components linking their names. We have shown that these assembled components satisfy our design requirements assuming the correctness of the components when considered in isolation. All this process has been illustrated by means of a case study: the design of an agent architecture for supporting anonymous interaction.

Our future work will concern the study of more expressive connectors among components. For example we would like to express more formally a "state export" operation by means of an adequate connector, generalizing the technique described in Section 5.2.3 to integrate the failure detector with the facilitator component. Moreover we would like to explore more severe failure models to detect and single out communication failures among components.

Acknowledgement

The authors would like to thank Gianluigi Zavattaro for his contribution in the development and presentation of the algebra of actors.

References

[1] P. Harmon, M. Watson, Understanding UML, Morgan Kaufmann, Palo Alto, CA, 1998.
[2] B. Meyer, Systematic concurrent object-oriented programming, Journal of ACM 36 (9) (1993) 56–80.
[3] R. Milner, Communication and Concurrency, Prentice Hall, 1989.
[4] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, Information and Computation 100 (1) (1992) 1–77.
[5] L.S.C. Bernardeschi, A. Fantechi, Formally verifying fault tolerant system designs, The Computer Journal 43 (3) (2000) 191–205.
[6] G. Agha, Actors: a Model of Concurrent Computation in Distributed Systems, MIT Press, 1986.

[7] M. Gaspari, G. Zavattaro, An Algebra of Actors, in: Proceedings of IFIP Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS), Kluwer Academic Publisher, 1999, pp. 3–18.
[8] R. Milner, Functions as processes, Mathematical Structures in Computer Science 2 (2) (1992) 119–141.
[9] G. Plotkin, A Structural Approach to Operational Semantics, Tech. Rep. DAIMI FN-19, Department of Computer Science, Aarhus University, Denmark, 1981.
[10] G. Agha, I.A. Mason, S. Smith, C. Talcott, Towards a theory of actor computation, in: W.R. Cleaveland (Ed.), Proceedings of the 3rd International Conference on Concurrency Theory (CONCUR), Lecture Notes in Computer Science, vol. 630, Springer-Verlag, Berlin, Heidelberg, 1992, pp. 565–579.
[11] G. Agha, I.A. Mason, S.F. Smith, C.L. Talcott, A foundation for actor computation, Journal of Functional Programming 7 (1) (1997) 1–72.
[12] C. Talcott, Interaction semantics for components of distributed systems, in: Proceedings of Formal Methods for Open Object-based Distributed Systems (FMOODS), Chapman & Hall, 1996, pp. 154–169.
[13] C. Talcott, An actor rewriting theory, in: J. Meseguer (Ed.), First International Workshop on Rewriting Logic and its Applications, Electronic Notes in Theoretical Computer Science, vol. 4, Elsevier, Amsterdam, 1996, pp. 360–383.
[14] M. Gaspari, G. Zavattaro, A process algebraic specification of the new asynchronous Corba Messaging Service, in: Proceedings of European Conference on Object Oriented Programming (ECOOP), Lecture Notes in Computer Science, vol. 1628, Springer-Verlag, Berlin, 1999, pp. 495–518.
[15] M. Gaspari, G. Zavattaro, An actor algebra for specifying distributed systems: the hurried philosophers case study, in: G. Agha, F. Decindio, G.Rozenberg (Eds.), Concurrent Object-Oriented Programming and Petri Nets, Lecture Notes in Computer Science, vol. 200, Springer-Verlag, Berlin, 2001, pp. 428–444.
[16] S. Mullender, Distributed Systems, Addison-Wesley, 1993.
[17] T. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, Journal of ACM 43 (2) (1996) 225–267.
[18] M. Gaspari, Concurrency and knowledge-level communication in agent languages, Artificial Intelligence 105 (1–2) (1998) 1–45.
[19] M. Gaspari, E. Motta, Symbol-level requirements for agent-level programming, in: A.G. Cohn (Ed.), Proceedings of the 11th European Conference on Artificial Intelligence (ECAI), John Wiley, Amsterdam, 1994, pp. 264–268.
[20] T. Finin, Y. Labrou, J. Mayfield, KQML as an agent communication language, Software Agents, MIT Press, 1997, pp. 291–316.
[21] Foundation for Intelligent Physical Agents, FIPA Communicative Act Library Specification, 2001. Available from <http://www.fipa.org/specs/fipa00037>.
[22] M. Singhal, Deadlock detection in distributed systems, IEEE Computer 22 (11) (1989) 37–48.
[23] M. Gaspari, An ACL for a dynamic system of agents, Computational Intelligence 18 (2) (2002) 102–119.
[24] N. Dragoni, M. Gaspari, Integrating agent communication languages in open services architectures, Technical Report UBLCS-2003-12, Department of Computer Science, University of Bologna, Italy, 2003.
[25] M.H. Nodine, A. Unruh, Facilitating open communication in agent systems: the InfoSleuth Infrastructure, in: Agent Theories, Architectures, and Languages, 1997, pp. 281–295.
[26] K. Prasad, Specification and proof of a simple fault tolerant system in CCS, Technical Report CSR-178-84, Department of Computer Science, University of Edinburgh, Scotland, 1984.
[27] R. Amadio, S. Prasad, Localities and failures, in: Proceedings of the 14th Foundations of Software Technology and Theoretical Computer Science Conference, Lecture Notes in Computer Science, vol. 880, Springer-Verlag, 1994, pp. 205–216.
[28] R. Amadio, An asynchronous model of locality, failure, and process mobility, in: Proceedings of COORDINATION, Lecture Notes in Computer Science, vol. 1282, Springer-Verlag, 1997, pp. 374–391.
[29] J. Riely, M. Hennessy, Distributed processes and location failures, Theoretical Computer Science 266 (1–2) (2001) 693–735.
[30] C. Fournet, G. Gonthier, J. Levy, L. Maranget, D. Remy, A calculus of mobile agents, in: Proceedings of the 7th International Conference on Concurrency Theory (CONCUR), Springer-Verlag, 1996, pp. 406–421.
[31] U. Nestmann, R. Fuzzati, Unreliable failure detectors via operational semantics, in: V.A. Saraswat (Ed.), Proceedings of ASIAN Conference, Lecture Notes in Computer Science, vol. 2896, Springer Verlag, 2003, pp. 54–71.
[32] U. Nestmann, R. Fuzzati, M. Merro, Modeling consensus in process calculus, in: Proceedings of the International Conference on Concurrency Theory (CONCUR), Lecture Notes in Computer Science, vol. 2761, Springer Verlag, 2003, pp. 393–407.