An Object Based Algebra for Specifying A Fault Tolerant Software Architecture

Nicola Dragoni **Mauro Gaspari**

Technical Report UBLCS-2003-9

June 2003

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)

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

Recent Titles from the UBLCS Technical Report Series

2002-4 *Specification and Analysis of Stochastic Real-Time Systems* (Ph.D. Thesis), Bravetti, M., March 2002.

2002-5 *QoS-Adaptive Middleware Services* (Ph.D. Thesis), Ghini, V., March 2002.

2002-6 *Towards a Semantic Web for Formal Mathematics* (Ph.D. Thesis), Schena, I., March 2002.

2002-7 *Revisiting Interactive Markov Chains*, Bravetti, M., June 2002.

2002-8 *User Untraceability in the Next-Generation Internet: a Proposal*, Tortonesi, M., Davoli, R., August 2002.

2002-9 *Towards Adaptive, Resilient and Self-Organizing Peer-to-Peer Systems*, Montresor, A., Meling, H., Babaoglu, O., September 2002.

2002-10 *Towards Self-Organizing, Self-Repairing and Resilient Distributed Systems*, Montresor, A., Babaoglu, O., Meling, H., September 2002 (Revised November 2002).

2002-11 *Messor: Load-Balancing through a Swarm of Autonomous Agents*, Montresor, A., Meling, H., Babaoglu, O., September 2002.

2002-12 *Johanna: Open Collaborative Technologies for Teleorganizations*, Gaspari, M., Picci, L., Petrucci, A., Faglioni, G., December 2002.

2003-1 Security and Performance Analyses in Distributed Systems (Ph.D Thesis), Aldini, A., February 2003.

2003-2 Models and Types for Wide Area Computing. The calculus of Boxed Ambients (Ph.D. Thesis), Crafa, S., February 2003.

2003-3 MathML Formatting (Ph.D. Thesis), Padovani, L., February 2003.

2003-4 Performance Evaluation of Mobile Agents Paradigm for Wireless Networks (Ph.D. Thesis), Al Mobaideen, W., March 2003.

2003-5 Synchronized Hypermedia Documents: a Model and its Applications (Ph.D. Thesis), Gaggi, O., March 2003.

2003-6 Searching and Retrieving in Content-Based Repositories of Formal Mathematical Knowledge (Ph.D. Thesis), Guidi, F., March 2003.

2003-7 Intersection Types, Lambda Abstraction Algebras and Lambda Theories (Ph.D. Thesis), Lusin, S., March 2003.

2003-8 Towards an Ontology-Guided Search Engine, Gaspari, M., Guidi, D., June 2003.

An Object Based Algebra for Specifying A Fault Tolerant Software Architecture

Nicola Dragoni[1] **Mauro Gaspari**[1]

Technical Report UBLCS-2003-9

June 2003

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. 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.

1. Dipartimento di Scienze dell'Informazione, University of Bologna, Via Mura Anteo Zamboni, 7, 40127 Bologna, Italy. E-mail: {dragoni,gaspari}@cs.unibo.it

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 concern the interaction among the components of a software architecture. In fact formalisms for object oriented design have often components to model the dynamics of a system, for example the UML activity diagrams[12]. 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 [14] 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 [15] or the π-calculus [16] which do not provide a direct representation of objects as first class entities. In these formalisms processes are stateless entities which communicate exploiting synchronous message passing and the representation of an object involves a large number of processes. 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.

In order to address this issue, Gaspari and Zavattaro [8] have proposed a process algebra based on a distributed object-oriented model (the Actor model [1]). The main result of this effort is the development of a formalism (the Algebra of Actors) which represents a compromise between the standard process algebras and the Actor model. Therefore, this formalism allow us to reuse standard results of the theory of concurrency in a context where object-based features (such as *object identity*, *asynchronous message passing*, *implicit message acceptance* and *dynamic object creation*) are provided.

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 the formal framework used for specifying a software architecture should provide abstract mechanisms for modelling failures and for reasoning about them.

In this paper we provide an extension of the actor algebra to model crash failures of actors and their detection. Then 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 managed with the standard tools provided by process algebra such as bisimulation or equality laws. 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 re-

quirement. We conclude the paper by discussing alternative agent architectures and highlighting our future research directions.

2 An Algebra of Actors

Actors are self-contained reactive processes with state whose behaviour 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 behaviour of an actor [1].

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 are 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 $C \stackrel{\text{def}}{=} P$ where P is a program defined as follows:

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

We allow recursive behaviours 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:

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

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

- *send*:

The program $send(e_1, e_2).P$ sends a message with contents e_2 to the actor 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 actual 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 behaviour 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.

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

$$ {}^{a}[e_1 : P_1 + \ldots + e_n : P_n]_s \xrightarrow{\tau} {}^{a}[P_i]_s \quad \text{if } [\![e_i]\!]^a_s = true \tag{4} $$

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).

Note that actors don't have an explicit receive primitive, which is instead *implicit*. Therefore, the receive operation does not correspond to an operation in the programming language and it is performed implicitly at certain points of the computation: only idle actors receive messages, and so become activated.

Definition 2.1. - **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*

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

Table 1. Operational Semantics

Definition 2.2. - **Computations.** *A transition system modelling computations in the actor algebra is represented by the triple* $(\Gamma, T, \{\stackrel{\alpha}{\rightarrow} \mid \alpha \in T\})$. $T = \{\tau\} \cup \{av, \overline{av}L \mid a \in \mathcal{A},\ v \in \mathcal{V},\ L \subseteq \mathcal{A}\}$ *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.* $\stackrel{\alpha}{\rightarrow}$ *is the minimal transition relation satisfying the axioms and rules presented in Table 1*

The rules $Send$, $Become$, $Create$ and $Guard$ have been already discussed. Rule $Deliver$ states that the term $\langle a, v \rangle$ representing a message v sent to the actor a is able to deliver its contents to the receiver by performing the action $\overline{av}\emptyset$. The corresponding receiving action labelled with av can be performed by the actor a when it is idle (rule $Receive$). The other rules are simply adaptation to our calculus of the standard laws for the π–calculus.
The 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 throw a restriction is defined by the rule $Open$: an actor can send restricted actor names in order to make them know 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.

The full algebra of actors, including a discussion about the main differences with respect to the formal semantics of actors and about different notions of equivalence of actor terms, can be found in [8, 9, 10].

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 [17]. 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 [17]. Then we recall the notion of *unreliable failure detector* for asynchronous distributed systems [2] 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 behaviour 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 behaviour whatsoever. For example, it can change state arbitrarily.
- **Arbitrary with message authentication**: faulty actors can exhibit arbitrary behaviour 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.

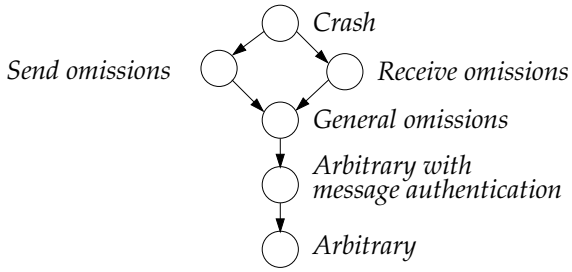

Figure 1. Classification of failure models. An arrow from type B to type A indicates that A is more severe than B.

Note that these failures can be classified in terms of "severity". Model A is *more severe* than model B if the set of faulty behaviours 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 behaviour of a faulty actor. Crash failures are the least severe failures listed above. The classification of models is illustrated in Figure 1.

The failure model we consider in this proposal is characterised by *crash* failures of actors in a *fully asynchronous* system. 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 don't 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 [2] 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 Modelling 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 modelling 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, despite the transition system has been extended modelling crashes, actors will not be able to

detect them using their standard primitives. In fact the behaviour of an actors 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 they 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 behaviours 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:

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

In the following we denote a correct actor by ${}^a\mathbb{C}_s$, which means that the actor a is idle (${}^aC_s$) or active (${}^a[P]_s$), but not faulty. Any correct actor in the system can crash and consequently become a faulty actor, as described in the following transition rules:

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

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

The first rule deals with a crash of an idle actor ${}^bC_s$. 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, consistent 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 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 behaviour[2]. Given an actor b:

- If b has crashed then $ping(b, x)$ binds x to f.
  This property means that if an actor b really crashes, then it is permanently suspected by every correct actor.
- If b is alive then:
  - $ping(b, x)$ binds x to t (b is alive) OR
  - $ping(b, x)$ binds x to f (b is very slow).

  This is the property of uncertainty: 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:

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

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

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

2. From now on we abbreviate $true$ and $false$ with t and f respectively.

The first rule ensures that if b is really crashed ($^b\mathbf{0}$), then $ping$ detects the failure (the variable x assumes the value f). Observe that the transition doesn't change the behaviour of the faulty actor, which remains crashed. The second and the third rules implement the unreliable behaviour of the primitive $ping$: if rule (9) fires the actor is considered too slow, otherwise if rule (8) fire the behaviour is correct.

It's important to observe that such unreliability is the same we can find, for instance, in some implementation of failure detectors based on a "time-out" mechanism (an example of a time-out based implementation is shown in Figure 2). Thus the above definition of the $ping$ primitive allows us to model an unreliable failure detector which is close to a possible implementation.

Figure 2. A TIME-OUT FAILURE DETECTOR. 1) Every actor a periodically sends a "a-is-alive" message ($\rightarrow$) to all the actors in the system. If an actor d times-out on some actor a, it adds a to its list of its suspects ($\Box$ symbolizes this list). 2) If d later receives an "a-is-alive" message, d recognises that it made a mistake by prematurely timing out on a: d removes a from its list of suspects, and increases the length of its timeout period for a in an attempt to prevent a similar mistake in the future.

In Table 2 the new set of transition rules which define the labelled transition system of the algebra is summarized.

4.3 Modelling an unreliable Failure Detector

An unreliable failure detector can be modelled 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 $a^1_d, a^2_d, \ldots, a^n_d$ 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 and $failures$ is the list of suspected actors. We also assume that a^i has crashed $\Leftrightarrow$ a^i_d has crashed.

The behaviour of a detector-actor is shown in Figure 3.

$C'^d \stackrel{\text{def}}{=}$
(i) message=init(dnames):
send(self, pingall(dnames)).become(C'^d, addnames(dnames)) +
(ii) message=pingall(nl) $\wedge$ $\neg$empty(nl):
(ii1) send(self, pingall(rest(nl))).ping(1st(nl), y).
(ii2) (y=true): become(C'^d, updnofail(1st(nl))) +
(ii3) $otherwise$: become(C'^d, updfail(1st(nl))) +
(iii) message=pingall(nl) $\wedge$ empty(nl):
send(self, pingall(dnames)).become(C'^d)

Figure 3. A simple failure detector service.

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 $dlist$.

Send	${}^a[send(e_1,e_2).P]_s \xrightarrow{\tau} {}^a[P]_s \mid \langle [\![e_1]\!]^a_s, [\![e_2]\!]^a_s \rangle$	
Deliver	$\langle a, v \rangle \xrightarrow{\overline{a}v\emptyset} \mathbf{0}$	
Become	${}^a[become(C,e).P']_s \xrightarrow{\tau} ({}^d[P'\{a/self\}]_s)\backslash d \mid {}^a C_{[\![e]\!]^a_s}$	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 $C \stackrel{\text{def}}{=} P$
Guard	${}^a[e_1 : P_1 + \ldots + e_n : P_n]_s \xrightarrow{\tau} {}^a[P_i]_s$	if $[\![e_i]\!]^a_s = true$
Crash	${}^b\mathbb{C}_s \xrightarrow{\tau} {}^b\mathbf{0}$	
Ping1	${}^b\mathbf{0} \mid {}^a[ping(b,x).P]_s \xrightarrow{\tau} {}^b\mathbf{0} \mid {}^a[P\{f/x\}]_s$	
Ping2	${}^b\mathbb{C}_s \mid {}^a[ping(b,x).P]_s \xrightarrow{\tau} {}^b\mathbb{C}_s \mid {}^a[P\{t/x\}]_s$	
Ping3	${}^b\mathbb{C}_s \mid {}^a[ping(b,x).P]_s \xrightarrow{\tau} {}^b\mathbb{C}_s \mid {}^a[P\{f/x\}]_s$	
Res	$\dfrac{A \xrightarrow{\alpha} A'}{A\backslash a \xrightarrow{\alpha} A'\backslash a}$	$a \notin n(\alpha)$
Open	$\dfrac{A \xrightarrow{\overline{a}vL} A'}{A\backslash b \xrightarrow{\overline{a}vL\cup\{b\}} A'}$	$a \neq b \ \wedge \ b \in n(v)$
Par	$\dfrac{A \xrightarrow{\alpha} A'}{A \mid B \xrightarrow{\alpha} A' \mid B}$	if $\alpha = \overline{a}vL \ then \ a \notin act(B) \wedge L \cap fn(B) = \emptyset$
Sinc	$\dfrac{A \xrightarrow{av} A' \quad B \xrightarrow{\overline{a}vL} B'}{A \mid B \xrightarrow{\tau} (A' \mid B') \setminus L}$	
Cong	$\dfrac{B \equiv A \quad A \xrightarrow{\alpha} A' \quad A' \equiv B'}{B \xrightarrow{\alpha} B'}$	

Table 2. The new operational rules of the Actor Algebra. For the sake of readability we denote a correct actor by ${}^b\mathbb{C}_s = {}^bC_s$ or ${}^b[P]_s$, which means that the actor a is idle or active but not faulty.

- *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^i_d receives an initialisation message from the actor a^i (i) it starts checking all the actors in the system by means of the primitive $ping$ (ii1); after each execution of $ping$, the detector updates the list $failures$ according to the result of the check (which is stored in y)(ii2 and ii3). The detector executes this program forever (iii).

Let us show that the above failure detector satisfies the following properties:

1. if an actor b really crashes, then it is permanently suspected by every correct actor;

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

1) 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 fired). As a consequence, the variable y is bound to f and the failure detector updates its state adding the failure to the list $failures$ (ii3).
2) This property follows directly from the unreliable behaviour 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 $Ping3$ fires. Consequently the detector updates its state adding the actor b to its list of suspects (ii3). Note that if the detector later discovers the actor is correct, then it updates its state removing b from the list $failures$ (ii2). 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. 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. The requests of knowledge are anonymous and independent from low level issues, such as management of agent names, routing and agent reachability. Following the style of [6] an agent in the system has a symbolic (logical) name and a *virtual knowledge base* (VKB), which is a set of first order formulas. Let $\mathcal{A}_{\mathcal{ACL}}$ be a countable set of agent names ranged over by $\hat{a}$, $\hat{b}$, $\hat{c}$,. . .. Let $VKB_{\hat{a}}$ be the virtual knowledge base of agent $\hat{a}$.

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

- $ask\text{-}everybody(\hat{a}, p)$ asks all agents interested in p for an instantiation of p which is true in their VKB[3].
- *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 requirement. The notion of *knowledge-level* in the context of multi-agent systems has been discussed in detail by Gaspari in [6]. 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 they both have propositional contents. However, this is not enough to guarantee a correct knowledge-level behaviour. In [6] additional conditions are postulated which require an accurate specification of the underlying agent architecture in order to ensure knowledge-level behaviour. We recall these conditions below.

- The programmer should not have to handle physical addresses of agents explicitly.
- The programmer should not have to handle agent crashes and communication faults explicitly.

3. Note that the *ask-everybody* primitive includes the name of the sender agent. This is necessary because the recipient agents needs to know the name of the sender to send it the replay. As discussed in [6] agents modelling 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 [4, 5].

- 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.
- 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 [19].

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

2 Open multi-agent 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. In such systems, which are referred as *open multi-agent systems*, it is usually impossible that agents possess complete built-in information about the other agents in the system, simply because such information will initially be unavailable. As was already pointed out by Hewitt and de Jong ([13]) the only thing that holds the components of an open system in common, is their ability to communicate. This means that an important ingredient of an open multi-agent system will be the agents' ability to communicate about each other, especially about features like their capabilities and their expertise. Our requirement is that the anonymous interaction protocol should function in open multi-agent systems. Thus both *ask-everybody* and *all-answers* should be designed to deal with a dynamic system of agents where new agents are added dynamically to cooperate with the existing ones, and agents can leave the community when their tasks terminate.

5.1 An Agent Architecture for Anonymous Interaction

Each agent has a knowledge-level (KL) component which implements the VKB of the agent and its reactive behaviour (see Figure 4)). 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 (Figure 4). 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 unavailable suddenly 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 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 (Figure 4). 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 Figure 5. An agent $\hat{a}$ is composed of three actors (a, a_f, a_d) which run in parallel

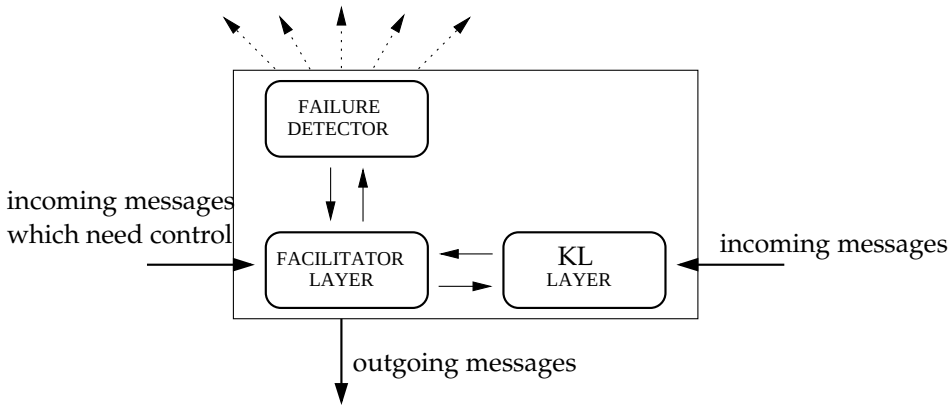

Figure 4. Fault tolerant agent architecture for anonymous interaction

$(a \mid a_f \mid a_d)$ and which implement the behaviour of the components of the general agent architecture (Figure 4):

- **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. Thus we don't require a centralized facilitator agent to deal with all the registration requests, but each agent can accept a register message.
- **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.

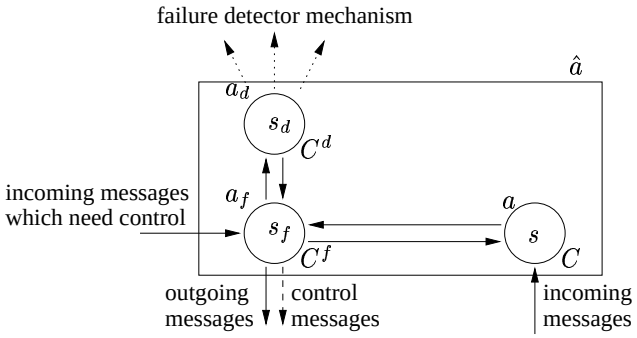

Figure 5. Actor-based agent architecture

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 needs 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 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 behaviour:

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

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 [6, 7].

An agent $\hat{a}$ uses the $ask\text{-}everybody(\hat{a}, p)$ 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[4]:

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

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

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

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

where $otherwise$ is a boolean expression which is true if all the previous guards are false (it is defined more formally in [10]).
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 $all\text{-}answers$ query, all the other messages are delayed (iii).

5.3 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;

answers contains information on the active conversations involving multicasting;

In Figure 6 we present a specification of a facilitator service which support the *ask-everybody* and *all-answers* primitives. We assume that a facilitator is able to translate agent names into physical actor names. The specification of a complete facilitator program for a knowledge-level Agent Communication Language can be found in [7].

4. 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 [6] for more details on this translation function).

$C'^f \stackrel{def}{=}$
(i) message=init((e_1, e_2, e_3)):
become(C'^f, (e_1, e_2, e_3)) +
(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)) $\wedge$ alltag(p):
send(kb-local, tell(self, s, p)).become(C'^f, updalltag(p, s)) +
message=updandfrw(tell(self, s, p)): become(C'^f) +
(iv) message=allanswers(self, p) $\wedge$ testalltag(p):
send(kb-local, allanswersyes).become(C'^f, cleanalltag(p)) +
message=allanswers(self, p) $\wedge \neg$testalltag(p):
send(kb-local, allanswersno).become(C'^f) +
(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)) +
(vb) 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), [], [])).
become(C^f, updfnames(b_f))

Figure 6. A facilitator service which support the ask-$everybody$ and *all-answers(p)* agent primitives.

When a facilitator receives an *init* message from a kb-actor (i), it updates its state by means of a *become* primitive. When a facilitator receives an *ask-everybody(s, p)* message, it consults its database and forwards an *ask-one* message to all the interested agents (ii). Then the facilitator intercepts all the answers (*updandfrw* message), registering them in its local state and forwarding them to the associated kb-actor (iii). This 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 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 (vb). The protocol in (vi) supports the dynamic creation of new agents. We'll explain in detail it in Section 6.2.

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 our language. 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 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:

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

and the operational semantics by adding the axiom:

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

Our idea for implementing the program $forward(x, nl, m).P$ in a term $[\![forward(x, nl, m).P]\!]$ of the initial algebra is to create a new actor whose behaviour (FWD) is to execute the forward of a

Function	Operates on	Descritpion
$getcomp(p)$	*fnames*	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.

Table 3. Functions which operate on the state of the facilitator.

message (see Figure 7). When this actor finishes to send all the messages it terminates (correctly) its execution (*i.e.* becomes the empty term **0**).

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

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

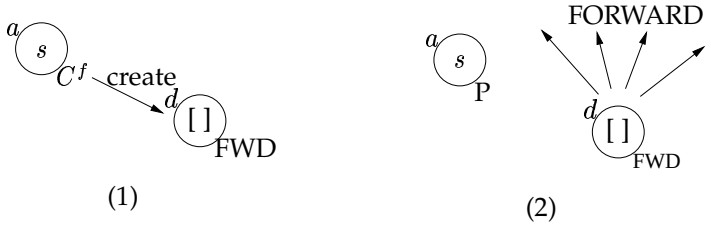

Figure 7. THE FORWARD PRIMITIVE. (1) In order to forward a message a new actor d is created. (2) The behaviour of this new actor is to execute the forward of a message. When it finishes to send all the messages it terminates correctly its execution. Note that in the meantime the actor a doesn't suspend its execution but it continues the program P.

5.4 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 behaviour C''^f of the facilitator actor is formally defined as

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

which means that the program C''^f is exactly the program C'^f where each occurrence of the expression $kb\text{-}local$ 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 Figure 6 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 *testall-tag(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 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 behaviour of a detector-actor when integrated in the agent architecture is shown in Figure 8.

$C''^d \stackrel{\text{def}}{=}$
(i) message=init(dnames):
send(self, pingall(x, dnames)).become(C''^d, addnames(dnames)) +
(ii) message=pingall(x, nl) $\wedge$ ¬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) $\wedge$ 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:** $\surd$

Figure 8. The program of the failure detector actor integrated in the agent architecture. Note that this program is similar to the one in Figure 3 except for the lines presented in bold.

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:

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

When a detector receives an initialisation 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 (iii) or deleting (iv) d to/from the list $dnames$ respectively. This functionality has been added to ensure the open multi-agent systems requirement. In Section 6.2 we'll show in detail an example of dynamic agent creation.

The second step of this integration is the extension of the facilitator program to include the list of suspected agents. In Figure 9 we show the new facilitator program C^f obtained after the integration of the facilitator component with the kb-actor and the failure detector.

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

(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)) +

(ivb) 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=updfail(x): become(C^f, addfail(x))** +
message=updnofail(x): become(C^f, remfail(x))

Figure 9. 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.

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 Figure 6 except for the lines presented in bold. The function $deltag(p, failures)$ updates the tags related to the query p cancelling 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 protocol in (vi) supports the dynamic creation of new agents and will be explained in detail in Section 6.2. The behaviour of the facilitator is extended with the protocol in (vii) 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 summarised.

Function	Operates on	Descritpion
$addfail(c_f)$	*failures*	Adds the actor c_f to the list.
$remfail(c_f)$	*failures*	Removes c_f if it is in the list.

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

6 Analysis of Requirements

The aim of this Section is to show that our agent architecture satisfies the specification requirements for the anonymous interaction protocol discussed in Section 5. To address this issue we proceed as follows. First, we focus on the knowledge-level programming requirement showing that the anonymous interaction protocol always succeeds. This means that deadlocks are not possible, also in presence of agent failures. We'll see that this result is possible thanks to the failure detector component. Then we focus on the open-ended requirement showing that the anonymous interaction protocol supports dynamic agent creation.

6.1 Analysis of the Anonymous Interaction Protocol

Let us consider a multi-agent system composed of four agents $\{\hat{a}, \hat{b}, \hat{c}, \hat{d}\}$. Suppose that the agent $\hat{a}$ uses the $ask\text{-}everybody(\hat{a}, p)$ 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.

Scenario 1. Let us suppose that the agents $\hat{b}, \hat{c}, \hat{d}$ are correct (and never fail) and they reply to $\hat{a}$ at different times ($t_0 \leq t_1 \leq t_2$ are the receiving times respectively) by means of the agent primitive *tell*, as shown in Figure 10. In the following we show that in this scenario the primitive *all-answers(p)* always succeeds.

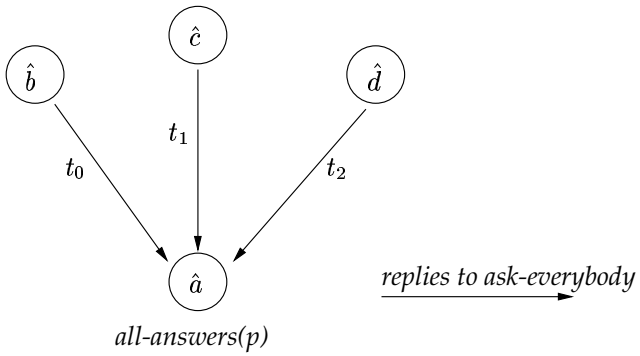

Figure 10. Scenario 1

If the facilitator actor a_f receives the *allanswers(a,p)* message (from the kb-actor) and $testalltag(p)$ is true (which means that all the replies have been received), then the facilitator send $allanswersyes$

to the kb-actor and thus the predicate *all-answers(p)* succeeds (see (iv) in the program in Figure 9). If the facilitator-actor a_f receives the *allanswers(a,p)* message and $\neg testalltag(p)$ is true, then the facilitator removes each agent $\hat{g} \in failures$ which has to answer about p (by means of the function deltag(p, failures)). For instance, suppose that an *all-answers* predicate is executed at the time t_*, where $t_0, t_1 < t_* < t_2$. The facilitator-actor a_f has already received the answers of the actors $\hat{b}$ and $\hat{c}$, while the answer of the actor $\hat{d}$ hasn't been received. Then the facilitator checks whether or not the actor $\hat{d}$ has crashed. Since $\hat{d}$ is correct, the facilitator continues to wait the answers of $\hat{d}$. If the *all-answers(p)* is executed at a time $t_3 > t_2$, then the primitive succeeds.

Scenario 2. Let us consider the same domain of the previous scenario and suppose that the agent $\hat{d}$ really crashes before answer to $\hat{a}$, as shown in Figure 11.

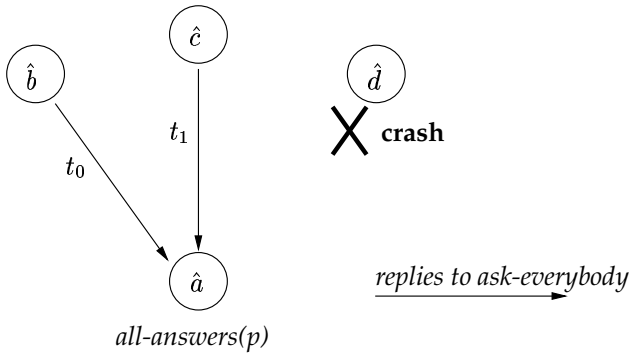

Figure 11. Scenario 2

In this scenario, when the facilitator a_f receives the message *allanswers(a,p)*, it executes (ivb) because $\neg testalltag(p)$ is $true$. Now two different situations are possible:

- **2.1** The detector has already discover the failure ($\hat{d} \in failures$): then the agent $\hat{d}$ is deleted from the list *answer* (in this example, the list is only composed of the actor $\hat{d}$). Suppose that this update is made at the time t_{up}. Thus, if the predicate *all-answers(p)* is executed at the time $t_3 > t_{up}$, then the predicate succeeds since now $testalltag(p) = true$.
- **2.2** The detector hasn't already discover the failure ($\hat{d} \notin failures$): then the facilitator continues to wait the answers of $\hat{d}$. Sooner or later the detector will discover the failure and communicate it to the facilitator (which will update its state executing (vi)). Thus the *all-answers(p)* succeeds as in 2.1.

Scenario 3. Let us consider the same domain of the scenario 2 and suppose that the agent $\hat{d}$ sent the answer to $\hat{a}$ and then it (really) crashes. Suppose also that when $\hat{a}$ executes the *all-answers* predicate, the facilitator hasn't received the message from $\hat{d}$ (thus $\neg testalltag(p) = true$). This scenario is shown in Figure 12.

The predicate *all-answers(p)* also succeeds in this scenario. Indeed, if $\hat{d} \notin failures$ and the facilitator receives the message $< \hat{a}, tell(\hat{a}, \hat{d}, p) >$, then *all-answers(p)* succeeds as in scenario 1. If $\hat{d} \in failures$ then the facilitator acts as in 2.1 and therefore *all-answers(p)* succeeds. In such a case the message sent from $\hat{d}$ hasn't effect on the current execution of the predicate *all-answers(p)*.

Scenario 4. In this scenario the agent $\hat{d}$ is erroneously suspected by the detector of $\hat{a}$, *i.e.* $\hat{d} \in failures$ but $\hat{d}$ is not really crashed. In this situation the facilitator acts as in the scenario 2.1

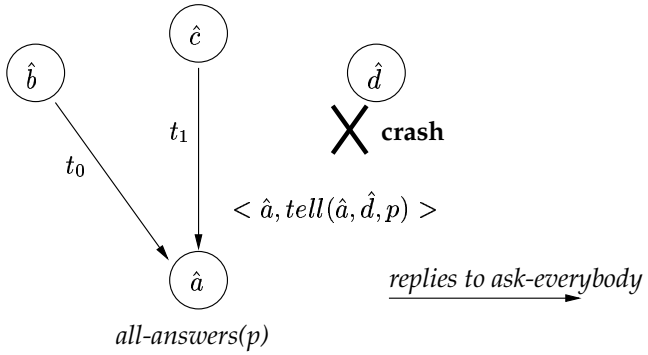

Figure 12. Scenario 3

because it's impossible to know if $\hat{d}$ has crashed or is only very slow. Thus the *all-answers(p)* primitive succeeds and the message sent from $\hat{d}$ hasn't effect on the current execution of the primitive *all-answers(p)*.

As we have showed above, the anonymous interaction protocol always succeeds. This means that if an agent executes an *ask-everybody* primitive, we are sure that sooner or later the agent will get an answer executing the *all-answers* primitive. No deadlocks or infinite waiting are possible, also in presence of crash failures of agents. Moreover, we have not to deal explicitly with physical names of actors and with crashes of agents. Thus the knowledge-level programming requirement is satisfied.

6.2 Analysis of the Dynamic Behaviour

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: $\hat{a}$ doesn't handle directly the queries sent from other agents, but instead it creates new 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 more adequate new agents.
In order to create a new agent $\hat{b}$, an existing agent can use the primitive $create(\hat{b}, w)$ which creates a new 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.
The following is the translation of an agent primitive $create(\hat{b},\ w)$ into the actor algebra (we suppose that the primitive is executed by an agent $\hat{a}$, as shown in Figure 13):

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

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

The creation of a new 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. After the creation of these actors, the kb-actor delegates their initializations to its local facilitator a_f. This is done by means of a *start* message. When the local facilitator a_f receives the $start$ message it behaves as follows:

message=start(b_f):

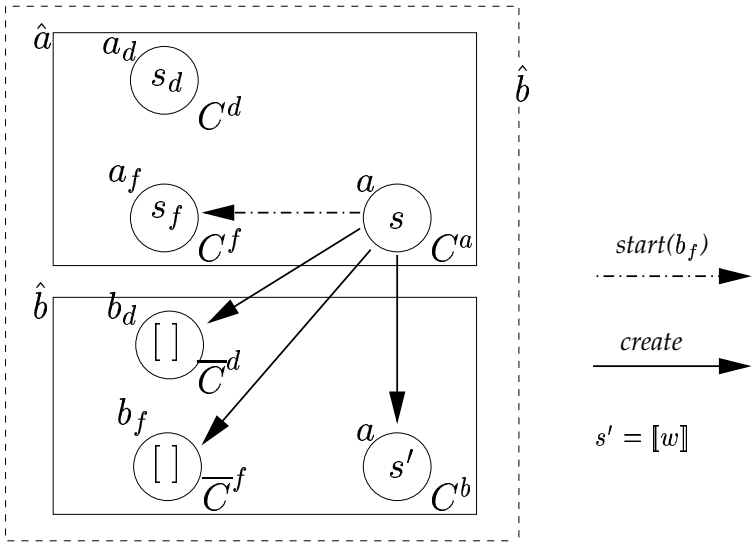

Figure 13. Dynamic agent creation. The dashed line shows the scope of the name of the created agent $\hat{b}$: the new agent is accessible (known) only by its creator $\hat{a}$.

(i) send(b_f, init((updfnames(self), [], [], failures)).
(ii) send(a_d, addname(b_d)).
(iii) become(C^f, updfnames(b_f))

The function *updfnames*(f) returns a new list of facilitators obtained inserting the facilitator f in the $fnames$ list. When a_f receives the *start*(b_f) message it initialises the state of the new facilitator b_f sending an *init* message (i). Then it updates the state of the local detector-actor a_d sending an *addname* message (ii): when the detector will receive this message it will update its state adding the name of the new detector b_d (see (iii) in the failure detector program in Figure 8). Finally the facilitator a_f executes a *become* primitive to update its state (iii) with the storage of the new facilitator name b_f.
The new facilitator b_f reacts to the *init* message sent from a_f by executing the following behaviour:

message=init((e_1, e_2, e_3, e_4)):
(i) send(b_d, init(e_1)).
(ii) become(C'^f, (e_1, e_2, e_3, e_4))

When b_f receives the *init* message, it initializes its local detector actor b_d sending the list of detector actors in the system (which is stored in the parameter e_1) (i). Then it updates its state by means of a *become* primitive (ii).

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 1. *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 2). Indeed, this primitive allows to hide the name of a newly created actor by means of the restriction operator $\backslash$. 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 actor primitives *create* 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 agent creator. Then we are sure that the new agent is only accessible (known) by its creator. The only way for an agent, say $\hat{b}$, to unhide its name is to send a message to an agent which is outside the scope of $\hat{b}$. For example, if an agent $\hat{b}$ is created from an agent $\hat{a}$ and $\hat{b}$ sends a message to an agent $\hat{c}$, then the restriction disappears and $\hat{b}$ becomes reachable from the outside.

Now it's simple to show that the anonymous interaction protocol supports dynamic agent creation. Since a newly created agent is unreachable from the external world, it cannot receive messages sent from other agents. The only way for an agent to receive a message sent by means of an *ask-everybody* primitive is register its interests. This can be done, for example, by means of a *register* primitive[5]. For insatnce, if a newly created agent $\hat{b}$ executes $register(\hat{b}, p)$, then that agent will receive all the messages sent by means of *ask-everybody* primitives which ask about p.

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 [18], 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.

With respect to the integration of new agents into an existing multi-agent system, we distinguish two different situations. First, there is the integration of an agent that already exists outside the system, in which case we refer to the integration as an *agent introduction*. An example of agent introduction can be found in [7]. Secondly, in the situation a newly integrated agent constitutes a previously nonexistent entity, we refer to the integration as an *agent creation*. In Section 6.2 we discuss in detail an agent primitive that can be used to create a new agent from an existing one. A particular agent creation is the act of agent cloning, which is typically performed in situations in which an agent with limited resources, faces an overload of tasks that need to be accomplished. To obviate this overload, the agent might then produce a clone of itself and subsequently delegate several of its tasks to this newly created agent. An example of agent cloning can be found in [7].

8 Conclusions

In this paper 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 specify a fault tolerant software architecture and to discuss its requirements. We have presented simple connectors which allows us 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. 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. This technique

5. A detailed example of a *register* primitive can be found in [7].

has been described in Section 5.4 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.

References

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

[2] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of ACM*, 43(2):225-267, 1996.

[3] Rogier M. van Eijk, Frank S. de Boer, Wiebe van der Hoek, John-Jules Ch. Meyer. Open multi-agent systems: Agent Communication and Integration. ATAL 1999: 218-232.

[4] T. Finin, Y. Labrou and J. Mayfield. KQML as an agent communication language. In Jeffery M. Bradshaw, editor, *Software Agents*. MIT Press, 1996.

[5] FIPA Communicative Act Library Specification. Foundation for Intelligent Physical Agents, 2001. *http://www.fipa.org/specs/fipa00037*.

[6] M. Gaspari. Concurrency and knowledge-level communication in agent languages. In Artificial Intelligence, 105 (1-2): 1-45, 1998.

[7] M. Gaspari. An ACL for a Dynamic System of Agents. Computational Intelligence, 18(2): 102-119, 2002.

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

[9] M. Gaspari and G. Zavattaro. A process algebraic specification of the new asynchronous corba messaging service. In *Proc. European Conf. on Object Oriented Programming (ECOOP)* number 1628 in Lecture Notes in Computer Science, pages 495–518. Springer-Verlag, Berlin, 1999.

[10] M. Gaspari and G. Zavattaro. An actor algebra for specifying distributed systems: the hurried philosophers case study. In G. Agha, F. Decindio, and G. Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, volume 2001 of *Lecture Notes in Computer Science*, pages 428–444. Springer-Verlag, Berlin, 2001.

[11] M.R. Genesereth, N. Nilsson. Logical Foundation of Artificial Intelligence. Morgan Kaufmann, Palo Alto, CA, 1986.

[12] P. Harmon and M. Watson. Understanding UML. Morgan Kaufmann, Palo Alto, CA, 1998.

[13] C. Hewitt and P. de Jong. Analyzing the roles of descriptions and actions in open systems. In Proceedings of 3rd National Conference on Artificial Intelligence (AAAI-83), pages 162-167, Washington, D.C., 1983.

[14] B. Meyer. Systematic Concurrent Object-Oriented Programming. Journal of CACM, 36(9): 56-80, 1993.

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

[16] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Journal of Information and Computation, 100:1-77, 1992.

[17] S. Mullender. Distributed Systems. ADDISON-WESLEY, 1993.

[18] Marian H. Nodine, Amy Unruh. Facilitating Open Communication in Agent Systems: The InfoSleuth Infrastructure. In *Agent Theories, Architectures, and Languages*, 281-295, 1997.

[19] M. Singhal. Deadlock Detection in Distributed Systems. *IEEE Computer*, 22(11): 37-48, 1989.