Integrating Agent Communication Languages in Open Services Architectures

Nicola Dragoni    Mauro Gaspari

October 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.

Recent Titles from the UBLCS Technical Report Series

Integrating Agent Communication Languages in Open Services Architectures

Nicola Dragoni\textsuperscript{1}  Mauro Gaspari\textsuperscript{1}


October 2003

Abstract

An Open Service Architecture is a software infrastructure that makes an open set of services available to users and agents. The role of agents is to retrieve, execute and compose available services providing more sophisticated instances of them. In order to achieve this goal, agents need to communicate with and use services provided by other agents. Although several Agent Communication Languages (ACLs) have been developed in the last few years providing high level communication primitives, for example based on speech acts, they are still not fully integrated in open service architectures. In this paper we address this issue studying the integration of Agent Communication Languages based on speech acts in these services oriented and geographically distributed architectures. We present the design of a knowledge-level fault tolerant ACL which concerns with the use, request and supply of knowledge, despite possible crashes of remote sites. We provide a formal definition of ACL primitives and a formal specification of the underlying architecture using an algebra of actors. We argue that, to integrate current ACLs in service-oriented geographically distributed architectures, they should be extended with: 1) knowledge-level one-to-many primitives, 2) support for anonymous (contents based) interaction, 3) an hidden unreliable failure detection mechanism.

\textsuperscript{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

Agents are one of the main building blocks of the World Wide Web under the Semantic Web activity. The Web is evolving toward an open, service-oriented architecture, which is a software infrastructure that makes an open set of information services available to users and agents. The role of agents is to retrieve, execute and compose available services providing more sophisticated instances of them. Knowledge is playing an increasingly important role in this scenario, services are being extended with semantic information and agents will provide complex problem solving capabilities to perform their tasks. Although there are many small-scale examples which exploit knowledge services on the Web, several issues concerning the integration of agents in the Semantic Web are still open. For example, Agent Communication Languages (ACLs) have been developed to provide high level communication primitives, but their integration in service oriented and geographically distributed infrastructures, which are subject to possible failures or malfunctions of nodes, is still an open issue.

In this paper we address this issue presenting the design of a knowledge-level fault tolerant ACL for an open service architecture. A knowledge-level ACL should concern with the use, request and supply of knowledge and not with symbol-level issues such as the reliability, the management of possible crashes of remote sites, synchronization of competing requests, the allocation of resources or the physical allocation of agents on a network [8].

We argue that to build such an ACL we need to extend current approaches with: 1) knowledge-level one-to-many primitives, 2) support for anonymous (contents based) interaction, 3) an hidden unreliable failure detection mechanism.

The paper is organized as follows. In Section 2 we present an abstract Open Services Architecture where both services and agents are considered at the knowledge level and we outline its main features and requirements. In Section 3 we introduce an Algebra of Actors modelling crash failures of actors and an unreliable failure detector mechanism. Then, in Section 4, we exploit the algebra to specify a Fault Tolerant ACL (FT-AACL) and the underlying agent architecture. Finally, in Section 5 we briefly discuss related work and in Section 6 we draw some conclusions and open issues of this research.

2 A Knowledge-level Open Services Architecture

Open services architectures define standard mechanisms for creating, naming, discovering and integrating service instances. These architectures can be defined at the knowledge-level when services are semantic web services and agents perform complex problem solving operations based on their semantics. However the complete exploitation of agents in general and, more specifically, of agent communication languages based on speech acts is still not clear in this scenario. Here we present an abstract service architecture which is useful to discuss these issues. Agents are the main building block of this architecture; they provide semantic web services to the outside world and they are also able to retrieve and use services published by other agents. Agents are defined at the knowledge level, following the approach of [8]: they should concern with the use, request and supply of knowledge without dealing with symbol level issues. Services are associated to virtual knowledge bases of agents they can be queried or activated by other agents and by users. In the following we outline the main features we assume for our abstract architecture:

- **Open architecture**: new services can be activated creating new agents which deal with them or extending the virtual KB of an agent.

- **Agent Communication Language**: agents access services and communicate using an agent communication language based on speech acts.

- **Asynchronous Communication**: agents 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).
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).

- **Competence of Agents**: each agent can deal with a set of services which constitutes its capabilities. These capabilities are expressed as a set of propositions and we assume an abstract matching operation that given a request of knowledge is able to match the competence of the agent matching propositions\(^2\).

- **Crash Failures**: agents are subject to possible crash failures, a faulty agent stops prematurely and does nothing from that point on. Before stopping, however, it behaves correctly\(^3\).

![Diagram](image)

**Figure 1.** A) Example of a simple Web Service request: agent A asks an agent P for a Web Service S. The agent P is able to provide S and it replies to A (of course, the reply depends on the service S). B) Example of a complex Web Service request: to execute the Web Service S, agent P needs to execute other services, such as S\(_1\), S\(_2\) and S\(_3\). If P is not able to solve these tasks, it requests the services to other agents (on the Web) which are able to solve them. When it receives all the replies then it execute the service S and then it replies to agent A.

**2.1 Knowledge-Level Requirements**

The notion of knowledge-level in the context of multi-agent systems has been discussed in detail by Gaspari in [8]. 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. In [8] conditions are postulated which require a careful analysis 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 communication faults explicitly.

\(^2\) In real open services architectures the semantics of services can be expressed with more complex pattern, for example the UPML model [5] also includes pre and post conditions on service execution.

\(^3\) Note that more severe types of failures can occur in these architectures, a well known classification of process failures in distributed systems [18].
The failure model we consider in this proposal is characterized by crash failures of agents in a fully asynchronous system. 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 [3] 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 detectors for systems with crash failures. The 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 has crashed even though it is still running. If this module later believes that suspecting a process was a mistake, it can remove it 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's 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.

2.2 Dealing with Crash Failures of Agents: Failure Detectors

The failure model we consider in this proposal is characterized by crash failures of agents in a fully asynchronous system. 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 [3] 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 detectors for systems with crash failures. The 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 has crashed even though it is still running. If this module later believes that suspecting a process was a mistake, it can remove it 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's 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.

3 An Algebra of Actors Dealing with Crash Failures

We present an algebra of actors which extends the formalism presented in [8] with dynamic primitives and facilities to model crash failures. 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: \( a, b, c, a_i, b_i, \ldots \) 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., \( L, L', L'' \subseteq \mathcal{A} \)). Let \( \mathcal{V} \) be a set of values (with \( \mathcal{A} \subseteq \mathcal{V} \)) 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 sel \( f \), 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 \( v, v', v'', \ldots \) when they appear as contents of a message and with \( s, s', s'', \ldots \) when they represent the state of an actor. \( [e]_c \) gives the value of \( e \) in \( \mathcal{V} \) assuming that \( a \) and \( s \) are substituted for sel \( f \) and state inside \( e \); e.g. \( [[\text{sel} f]]_s = a \) and \( [\text{state}]_s = s \). 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 $C$ be a set of actor behaviours identifiers: $C, C', \ldots$ will range over $C$. We suppose that every identifier $C$ is equipped with a corresponding behaviour definition $C \stackrel{df}{=} P$ where $P$ is a program defined as follows:

$$P \ := \ \text{become}(C,e).P \ | \ \text{send}(e_1,e_2).P \ | \ create(b,C,e).P \ | \ e_1 : P_1 + \ldots + e_n : P_n \ \sqrt{\cdot}$$

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

$$C \stackrel{df}{=} \text{become}(C,\text{state}).\sqrt{\cdot}.$$ 

Actor terms are defined by the following abstract syntax:

$$A \ ::= \ aC_s \ | \ a[P]_s \ | \ a0 \ | \ \langle a,v \rangle \ | \ A|A \ | A|a \ | 0$$

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

In order to model a crash failure in the algebra, we need to extend the standard behaviours of actors defined in [10] with a syntactic symbol $a0$ for each actor $a \in A$, which indicates that actor $a$ has crashed.

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

- **send:**
  The program $\text{send}(e_1,e_2).P$ sends a message with contents $e_2$ to the actor indicated by $e_1$:

$$a[\text{send}(e_1,e_2).P]_s \xrightarrow{\tau} a[P]_s \ | \ \langle e_1 \rangle^a, [e_2]^a \rangle$$

where $\tau$ represents an internal invisible step of computation.

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

$$a[\text{become}(C,e).P']_s \xrightarrow{\tau} (a[P'\{a/\text{sel}f\}]_s)s \ | \ aC_{[d]} \ | \ d$$

where $d$ is fresh (3)

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. $e = \text{state}$). The continuation $P'$ is executed by the new actor $d[P'\{a/\text{sel}f\}]_s$. 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 sel $f$, 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 $\text{create}(b,C,e).P'$ creates a new idle actor having state $s$ and behaviour $C$:

$$a[\text{create}(b,C,e).P']_s \xrightarrow{\tau} (a[P'\{d/b\}]_s \ | \ dC_{[d]} \ | \ d)$$

where $d$ is fresh (3)

The new actor receives a fresh named $d$. 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 \textit{true} or \textit{false}. The branch $P_i$ can be chosen only if the value of the corresponding expression $e_i$ is \textit{true}:

$$a[e_1 : P_1 + \ldots + e_n : P_n] \overset{a}{\rightarrow} \beta[e_i] : \Gamma \quad \text{if} \ [e_i] = \text{true}$$

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

Note that actors don’t have an explicit receive primitive, which is instead \textit{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.

### 3.1 Modelling Crash Failures in the Actor Algebra

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 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 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.

In the following we denote a correct actor by $^aC_a$, which means that the actor $a$ is idle ($^aC_a$) or active ($^a[P_a]$), but not faulty. Any correct actor in the system can crash and consequently become a faulty actor, as described in the following transition rule (\textit{Crash}):

$$^bC_a \overset{\tau}{\rightarrow} ^b\mathbf{0}$$

We model a crash by means of a transition from a correct actor, say $^bC_a$, to the respective faulty actor $^b\mathbf{0}$. When this transition 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, only correct actors are able to send and receive messages.

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: $\text{pin}(\text{actor}_\text{name}, x)$, where $\text{actor}_\text{name} \in A$ and $x \in \mathcal{X}$, and we assume the following behaviour$^4$. Given an actor $b$:

- If $b$ is crashed then $\text{pin}(b, x)$ binds $x$ to $f$.

  This property means that if an actor $b$ is really crashed, then it is permanently suspected by every correct actor.

- If $b$ is alive then:

  - $\text{pin}(b, x)$ binds $x$ to $t$ ($b$ is alive) OR
  - $\text{pin}(b, x)$ 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.

---

$^4$. We abbreviate \textit{true} and \textit{false} with $t$ and $f$ respectively.
Based implementation is shown in Figure 5. Thus the above definition of the implementation of failure detectors based on a “time-out” mechanism (an example of a time-out behaviour is correct.

The rules actor term. Given the actor term represented by the triple of some restriction, in the label of labels, where, respectively represent the receiving and the emission of the message with receiver and contents. The second and the third rules implement the unreliable behaviour of the primitive ping: if rule (8) fires the actor is considered too slow, otherwise if rule (7) 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 5). Thus the above definition of the ping primitive allows us to model an unreliable failure detector which is close to a possible implementation.

3.2 Operational Semantics

Definition 3.1. - Structural congruence. A structural congruence is the smallest congruence relation over actor terms (≡) satisfying:

\[ a \equiv [a] \equiv 0 \]

\[ A|B ≡ B|A \]

\[ 0|a ≡ 0 \]

\[ (A|B)|a ≡ A|(B|a) \quad \text{where } a \not\in \text{fn}(A) \]

\[ A\{a\}b ≡ A\{b\}a \quad \text{where } b \text{ is fresh} \]

Definition 3.2. - Computations. A transition system modelling computations in the actor algebra is represented by the triple \( (\Gamma, T, \{\phi \mid \alpha \in T\}) \). \( T = \{\tau\} \cup \{av, \tau L \mid a \in A, v \in \mathcal{V}, L \subseteq A\} \) is a set of labels, where \( \tau \) is the invisible action standing for local autonomous steps of computation; \( av \) and \( \tau L \) respectively represent the receiving and the emission of the message with receiver \( a \) and contents \( v \). The set \( L \) in the label \( \tau L \) 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. \( \phi \) 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 \( \{a, v\} \) representing a message \( v \) sent to the actor \( a \) is able to deliver its contents to the receiver by performing the action \( \tau L \). 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 \( \pi \)-calculus.

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
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 restriction is defined by the rule:

\[ a \mathbf{C}_s \xrightarrow{\tau} a[P]_s \]

in Table 1. Operational Semantics of the Actor Algebra. For the sake of readability we denote a correct actor by \( ^bC_s = ^bC, \) or \( ^b[P]_s, \) which means that the actor \( a \) is idle or active but not faulty.

\[ ln(A) \] (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 \( \text{act}(A) \) to denote the set of the names of the actors in \( A. \)

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 \( \text{Res} \)). The only way to pass throw a restriction is defined by the rule \( \text{Open}: \) 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 $m$. The process of extending the scope of the restriction terminates only when the message is received (rule $m_\text{inc}$), 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 $mL$), only if $B$ does not contain the target actor (i.e., $a \not\in \text{act}(B)$) 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 formal framework of the algebra of actors including actor equivalence and bisimulation and 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 [10, 11, 12].

### 3.3 Modelling an unreliable Failure Detector

An unreliable failure detector can be modelled in the actor algebra using the $\text{ping}_g$ primitive. Typically a failure detector is a distributed program: each site in a distributed system has its own failure detector module.

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 $(\text{dnames}, \text{failures})$, where $\text{dnames}$ is the list of all the detector actors in the system and $\text{failures}$ is the list of suspected actors. We also assume that $a^i$ has crashed $\iff a^j_d$ has crashed.

The behaviour of a detector-actor is shown in Figure 3 and is realized as a guarded program of the form:

$$C^d \overset{def}{=} e_1 : P_1 + \ldots + e_n : P_n$$

(9)

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

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

When a detector $a^i_d$ receives an initialisation message from the actor $a^i$ (1) it updates its state and starts checking all the actors in the system (1.1)(2). In order to check an actor the detector executed the primitive $\text{ping}$ (2.1). After each check the detector updates the list $\text{failures}$ with the result of the check (2.2)(2.3). The detector executes this program forever (3): when it finishes to...
check all the actors in the list dnames it restarts from the beginning of the list (3.1).

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 \( b \) 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 Ping\( b, y \) 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\( b, y \). 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\( b, y \) 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.

4 A Fault Tolerant ACL for Open Services Architectures

In this Section we provide the specification of a Fault-Tolerant ACL (FT-ACL) which supplies high level communication mechanisms supporting anonymous interaction in Open Services Architectures. We present the underlying agent architecture which is composed of three main components: a \( kb\)-actor encodes the services of the agent, while a facilitator and a distributed failure-detector module support high level communication primitives. We list the primitives of the language and we briefly discuss their fault-tolerant behaviour. Then we provide a formal specification using the extended algebra of actors previously discussed in detail.

4.1 A Fault-Tolerant Agent Architecture

Following the style of [8], an agent in the system has a symbolic (logical) name and a virtual knowledge base (VKB). The communication actions are asynchronous performatives based on the speech act theory, allowing buffering of messages and supporting non blocking ask primitives. We also assume that each communication action contains information in a given knowledge representation formalism.

Let \( \mathcal{A}_{ACL} \) be a countable set of agent names ranged over by \( \hat{a}, \hat{b}, \hat{c}, \ldots \). Let \( VKB_a \) be the virtual knowledge base of agent \( \hat{a} \). Agent terms \( A_g, A_g', A_g'', \ldots \), are defined with the following abstract syntax: \( A_g := \hat{a} \mid A_g' \mid A_g'' \).

\( VKB \) and \( handler \) function. Agents provide services to other agents and to the user. Each agent has an associated \( handler \) function which maps the received message into the corresponding services. Services are represented by appropriate performatives and implemented as queries on the VKB or as compositions of other services. \( H_{\hat{a}} \) will be the \( handler \) function of agent \( \hat{a} \). The \( handler \) function is enclosed in the \( VKB \) of an agent: for each agent \( \hat{a} \in \mathcal{A}_{ACL}, H_{\hat{a}} \subseteq VKB_{\hat{a}} \) and \( VKB_{\hat{a}} - H_{\hat{a}} \) is a set of first order formulas; \( w, w', w'' \) will range over VKB. The \( handler \) function is expressed by a set of Prolog-like rules \( \{ r_1, r_2, \ldots, r_n \} \) having the form:

\[
\text{handler(service}(\hat{a}, \hat{b}, p)) \leftarrow \text{body} \quad (10)
\]

where \( body \) is a sequence of literals \( h_1 \land h_2 \ldots \land h_n \) in which each \( h_i \) can be a communication action, a dynamic primitive or a predicate on the VKB of the agent. There are no explicit receive primitives inside such predicates. Idle agents repeatedly look for messages: they perform pattern
matching between the incoming message and the head of rules and, when a matching succeeds, the body of the selected rule is executed.

This knowledge-level characterization of agents is mapped in a symbol-level architecture which is hidden at the agent level and is formalized in the algebra of actors. We follow the approach of [9] representing the internal components of agents as actors. We assumes a distributed facilitator mechanism which is hidden at the agent level. The facilitator is used to register new agents and services and to deliver service requests to agents which provide the adequate capabilities. Thus the Open Services Architecture does not require a centralized facilitator agent to deal with all the services requests, but it assumes that each agent can register services locally. The local facilitator component implements a distributed algorithm and forwards the relevant information to all the sites of the Open Services Architecture. To deal with crash failures we extend the agent architecture with a distributed failure detector assuming that each agent has its own failure detector module.

Agent architecture. The architecture of an agent is illustrated in Figure 4. An agent \( \hat{a} \) is composed of three actors \((a, a_f, a_d)\) which run in parallel \((a | a_f | a_d)\) and which have different behaviours:

- **kb-actor** \( a \): this actor implements the VKB and the handler function of the agent \( \hat{a} \), which specifies the services provided by the agent.

- **facilitator-actor** \( a_f \): this actor implements a distributed facilitator mechanism which provides anonymous (contents based) facilities to retrieve and request services.

- **detector-actor** \( a_d \): this actor implements a distributed failure detector mechanism. It monitors all the agents in the system and communicates to the facilitator-actor those that it currently suspects to have crashed.

![Figure 4. Agent architecture](image-url)

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 or in a new name server component. In general, incoming requests of service are handled by the kb-actor which uses an encoding of the handler function, but if services needs some control operations then they are sent through the facilitator layer. The facilitator-actor deals with the outgoing service requests 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 sends a message to the facilitator when it finds a crashed agent.
4.2 Primitives of the FT-AACL

The FT-AACL provides a set of communication primitives based on speech acts, following the approach of KQML. The language supports an anonymous interaction protocol which has been developed for Open Services Architectures and which is integrated with standard agent-to-agent performatives. This allows an agent to perform a request of services based on contents without knowing the name of the recipient agent. If required they can also continue the cooperation using agent-to-agent communication primitives to implement more specific services. Another feature FT-AACL provides to Open Services Architectures is a support for agent creation and cloning. Thus new agents and new services can be created dynamically and become part of the problem solving activity.

The primitives of the language can be divided into four categories as follows:

<table>
<thead>
<tr>
<th>Category</th>
<th>Performatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Conversation performatives</td>
<td>insert((a, b, p)), ask-one((a, b, p)), tell((a, b, p))</td>
</tr>
<tr>
<td>Contents based services requests (one-to-many performatives)</td>
<td>ask-everybody((b, p)), ask-first((b, p)), ask-best((L, b, p)), all-answers(p)</td>
</tr>
<tr>
<td>Registration of new services</td>
<td>register((b, p)), unregister((b, p))</td>
</tr>
<tr>
<td>Support for open architectures</td>
<td>create((b, w)), clone((b)), bye</td>
</tr>
</tbody>
</table>

Contents based services requests are realized as one-to-many performatives: whenever an agent needs a given service \(p\) it can execute these multicast primitives. We propose three different one-to-many performatives, which are defined as follows:

- **ask-everybody(\(\hat{b}, p\))**: Agent \(\hat{b}\) asks all agents interested in a service which matches \(p\) (except \(\hat{b}\)) for an instantiation of \(p\) which is true in their VKB. When an agent has executed this primitive, then it can use the predicate all-answers(\(p\)) to verify whether all the answers have been received or not. This predicate succeeds when an agent receives all the answers relative to a given service request \(p\) despite failures, where \(p\) must be exactly the same term (modulo variable renaming) occurring in the associated request. Despite failures means that the answers of fault agents are not waited.

- **ask-first(\(\hat{b}, p\))**: Agent \(\hat{b}\) asks all agents interested in a service which matches \(p\) (except \(\hat{b}\)) for an instantiation of \(p\) which is true in their VKB, and it gets the first one. This primitive does not consider crashed agents, it just gets the first answer to the query and discards all the other.

- **ask-best(\(L, \hat{b}, p\))**: Agent \(\hat{b}\) asks the first agent in the list \(L\) despite failures, say \(\hat{c}\), for an instantiation of \(p\) which is true in the VKB of \(\hat{c}\). Despite failures means that if \(\hat{c}\) fails before \(\hat{b}\) has received the answer, then \(\hat{b}\) doesn’t wait the reply of \(\hat{c}\) but instead asks the second agent in the list \(L\) and so on. In the current specification we assume that the list \(L\) is computed by the agent that performs the request, possibly as a result of a previous ask-everybody performeative.

The ask-everybody and ask-first primitives implement anonymous contents based requests for services: an agent which execute them does not need to know the names (or internet address) of all the agents which provide services related to the query.

Note that we assume a simple competence matching operation: the competence of an agent is expressed as a set of propositions which should match the requests for services. However, our approach can be easily extended with more complex competence specifications providing an adequate competence matching function. More specific one-to-many primitives can be defined if we provide a more complex matching operation, for example if it is possible to order the competence...
level of different agents with respect to a given service. In this case an ask-best(\(\hat{b}, p\)) primitive could be defined (removing the list \(L\)) which contacts the agent having the best competence despite failures.

The primitives create, done and bye are provided to support an open and dynamic architecture: new agents can be created from other agents in the system (for example to cooperate with the existing ones providing new services) and agents can leave the community when their tasks terminate. Agents declare the services which they provide through the register primitive, and the requests for services are forwarded to agents on the basis of these declarations.

The meaning of conversation performatives is standard and as been discussed in details in previous papers of one of the authors [8] [9]. In Table 2 we summarize the informal definitions of all the supported primitives.

In order to provide a formal specification of \(\mathbb{FT-AcL}\), we encode its primitives in the actor algebra. As we show in following Sections, this encoding includes the specification of the behaviours of the three actors which compose an agent (Figure 4).

### 4.3 The kb-actor

The encoding of the handler function of an agent follows the approach of [8]: we use the guarded programs of the actor algebra to implement the rules of the handler function. Each clause of the handler function is associated with a condition of the guarded program which implements the behaviour of the kb-actor for a given service. The encoding is defined by means of a function \([\text{A}_g]\) which translates agent terms into actor terms. We adopt the following translation for an agent \(\hat{a}\) (we use the notation \([\text{A}_g]^a\) when we translate an agent \(\hat{a}\)):

\[
[H_a]^a = \sum_{r \in H_a} [r]^a \tag{11}
\]

\[
[r]^a = [\text{handler}(msg) \leftarrow \text{body}]^a = (\text{message} = msg) : [\text{body}]^a \tag{12}
\]

As a result the behaviour of a kb-actor \(a\) is realized as a guarded program which implements the handler function:

\[
[H_a]^a = C_a \overset{\text{def}}{=} e_1 : P_1 + \ldots + e_n : P_n. \tag{13}
\]

The agent primitives are translated into actor messages and the dynamic primitives create new actors, as shown in Table 3.

As an example, let us consider a very simple Web Service executed by an agent \(\hat{a}\): a Counter. The state of the kb-actor could be an Integer number \(\text{count}\) which represents the current value of the Counter. When the service is invoked by an agent, say \(\hat{b}\), then the agent \(\hat{a}\) increments \(\text{count}\) and sends the new value to \(\hat{b}\) as a reply.

The Counter service can be described with the following handler function:

\[
H_a : \quad \text{handler}(\text{ask-one}(\hat{a}, \hat{b}, \text{counter})) \leftarrow \text{tell}(\hat{b}, \hat{a}, \text{count} + 1) \text{become}\{\text{count} + 1\}
\]

The translation generates a kb-actor \(\\otimes[C]_{\text{count}}\) which has the following behaviour:

\[
C \overset{\text{def}}{=} \text{message} = \text{ask-one}(\hat{a}, \hat{b}, \text{counter}) : \text{send}(a_f, \text{tell}(\hat{b}, \hat{a}, \text{count} + 1)) \text{become}\{C, \text{count} + 1\}
\]

In this proposal we don’t specify the behaviour of agents with respect to their reactions towards specific services. Thus we don’t show a particular kb-actor program or handler function (the above behaviour is only an example). In fact, our opinion is that all the reactions towards incoming services requests are agent-dependent and hence should not be part of the semantics of agent primitives. They should be only specified in the agent program. For instance, in the above handler function we may implement an incorrect behaviour: the agent could reply count instead of count+1 to a counter request. We think that this contradictory behaviour is part of a higher level semantics, such as for example a “semantics of conversation” rather than a semantics of communication primitives.
<table>
<thead>
<tr>
<th>Agent primitives</th>
<th>Knowledge-level behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert($\hat{a}, \hat{b}, p$)</td>
<td>Agent $\hat{b}$ asks agent $\hat{a}$ to insert $p$ in its VKB.</td>
</tr>
<tr>
<td>ask-one($\hat{a}, \hat{b}, p$)</td>
<td>Agent $\hat{b}$ asks agent $\hat{a}$ for an instantiation of $p$ which is true in the VKB of $\hat{a}$.</td>
</tr>
<tr>
<td>tell($\hat{a}, \hat{b}, p$)</td>
<td>Agent $\hat{b}$ sends agent $\hat{a}$ an instantiation of $p$ which is true in the VKB of $\hat{b}$.</td>
</tr>
<tr>
<td>ask-everybody($\hat{b}, p$)</td>
<td>Agent $\hat{b}$ asks all agents in the system interested in a service which matches $p$ (except $\hat{b}$) for an instantiation of $p$ which is true in their VKB.</td>
</tr>
<tr>
<td>ask-first($\hat{b}, p$)</td>
<td>Agent $\hat{b}$ asks all agents in the system interested in a service which matches $p$ despite failures (except $\hat{b}$) for an instantiation of $p$ which is true in their VKB, and it gets the first one.</td>
</tr>
<tr>
<td>ask-best($L, \hat{b}, p$)</td>
<td>Agent $\hat{b}$ asks the first agent in the list $L$ despite failures, say $\hat{c}$, for an instantiation of $p$ which is true in the VKB of $\hat{c}$.</td>
</tr>
<tr>
<td>all-answers($p$)</td>
<td>This predicate succeeds when an agent receives all the answers relative to a given service request $p$ despite failures, where $p$ must be exactly the same term (modulo variable renaming) occurring in the associated request.</td>
</tr>
<tr>
<td>register($\hat{b}, p$)</td>
<td>An existing agent $\hat{b}$ specifies that is interested in propositions having the form $p$.</td>
</tr>
<tr>
<td>unregister($\hat{b}, p$)</td>
<td>An existing agent $\hat{b}$ specifies that is not interested in propositions having the form $p$.</td>
</tr>
<tr>
<td>create($\hat{b}, w$)</td>
<td>Creates a new agent with a new (fresh) name $\hat{b}$ and a new VKB $w$. The name of the new agent is known only by the agent that creates it.</td>
</tr>
<tr>
<td>clone($\hat{b}$)</td>
<td>When an agent $\hat{a}$ executes this primitive it creates a clone agent with a new (fresh) name $\hat{b}$ and $\text{VKB}_b$ as $\text{VKB}$ (if the agent $\hat{a}$ was registered in the system for knowledge $p$, then $\hat{b}$ will also be registered);</td>
</tr>
<tr>
<td>bye</td>
<td>Terminates the execution of the agent which performs the primitive. This information is sent to all the agents in the systems.</td>
</tr>
</tbody>
</table>

Table 2. Knowledge-level specification of the primitives of $\text{FT-ACL}$. 

In the following we briefly discuss the encoding of the $\text{FT-ACL}$ agent primitives. We focus only on all-answers, create and clone. We omit a discussion on the rest of the primitives because their encoding is very simple: they are forwarded to the local facilitator actor.
a detector-actor with fresh behaviour names 

answer, all the other messages are delayed. 

program and starts waiting for other messages. Note that while the kb-actor is waiting for a positive answer then it continues the execution of the rest of the program. Otherwise it stops the (which stores all the incoming messages) and waiting for its answer. If the kb-actor receives a All-answer primitive.

The detector-actor is initialised by 

create 

callone 

by 


All-answer primitive. This primitive is implemented by sending a request to the local facilitator (which stores all the incoming messages) and waiting for its answer. If the kb-actor receives a positive answer then it continues the execution of the rest of the program. Otherwise it stops the program and starts waiting for other messages. Note that while the kb-actor is waiting for an answer, all the other messages are delayed.

Create primitive. This primitive creates a new agent, including a kb-actor, a facilitator-actor and a detector-actor with fresh behaviour names \( C^a, C^b, C^d \), respectively. The facilitator-actor obtains all the information needed to start its activity from the facilitator of agent \( \tilde{a} \). To implement this behaviour, a start message is sent to the facilitator-actor \( a_f \). The detector-actor \( b_d \) is initialised by the facilitator \( b_f \) when it starts its execution.

Table 3. Specification of the \( \mathcal{F}^+\)-ACL primitives in the actor algebra. Here \( p \) represents a first order predicate.
Clone primitive. This primitive clones agent \( \hat{a} \) including the kb-actor, the facilitator-actor and the detector-actor; the new facilitator obtains all the information needed to start its activity from the facilitator of the agent \( \hat{a} \). To implement this behaviour, a clone message is sent to the facilitator-actor \( a_f \). The detector-actor is initialised by the facilitator \( b_f \) when it starts its execution.

4.4 The facilitator-actor

Our open services architecture is based on a distributed facilitator service. Each agent has its own local facilitator process which executes a distributed algorithm: it forwards control information to all the other local facilitators, and delivers requests of services to their destinations. The distributed facilitator is formally specified as a dynamic set of local facilitator actors \( \{a_f, a'_f, a''_f, \ldots \} \) which run the same actor program. This set may evolve dynamically whenever a new agent is created or an agent terminates its computation.

The state of the facilitator actor. The state of a local facilitator is a quadruple \( (f_{names}, competence, answers, failures) \), where \( f_{names} \) 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 and \( failures \) is the list of all the agents which are suspected to have crashed. This list is dynamically updated as information on suspected agents is received from the detector-actor. An agent in \( failures \) is only suspected to have crashed because the failure detector we use is unreliable. In fact an agent can be in the \( failures \) list even if it hasn’t really crashed. Thus the list \( failures \) represents “a list of suspected agents”.

The behaviour of the facilitator. The behaviour of the facilitator-actor is realized as a guarded program of the form:

\[
C^f 
= e_1 : P_1 + \ldots + e_n : P_n
\]  

(14)

The program executed by the facilitator is shown in Appendix A. It is based on the following two assumptions:

- a local facilitator is able to translate agent names into physical actor names;
- a local facilitator knows the names of its kb-actor and detector-actor.

In the following we describe the behaviour of the facilitator. To do this we discuss in detail the program it runs, showing how it deals with all the incoming messages. For the sake of readability we assume that the program is executed by a facilitator \( \hat{a}_f \). Thus the local kb-actor is \( a \), the local detector is \( a_d \) and the expressions \( self, sel_f \) indicate \( \hat{a}, a_f \) respectively.

Insert and tell messages. When a facilitator receives an insert message from its local kb-actor (1), it directly forwards the message to the recipient actor (1.1). Instead, if a facilitator receives a tell message (2), it sends an updandfrw message to the facilitator of the recipient agent (2.1). When this message is received, it is forwarded to the local kb-actor, as shown in (3).

(1) message=insert(\( \hat{b}, \hat{a}, p \)):
(1.1) send(\( \hat{b}, \hat{a}, p \)).become(\( C^f \)) +
(2) message=tell(\( \hat{b}, \hat{a}, p \)):
(2.1) send(\( \hat{b}_f, \text{updandfrw}(\text{tell}(\hat{b}, \hat{a}, p)) \)).become(\( C^f \)) +
(3) message=updandfrw(tell(self, \( \hat{b}, p \))): send(a, tell(self, \( \hat{b}, p \)).become(\( C^f \)) +

Ask-one message. The message ask-one (4) is directly sent to the recipient kb-actor \( \hat{b} \) (4.1). After this delivery, the facilitator stores in its state the information that the recipient agent \( \hat{b} \) has to answers about \( p \). This is done by means of the primitive become which uses a function settag
to update the facilitator state (4.2). When the facilitator receives the answer of the query (5), it forwards the message to the local kb-actor (5.1) and changes its state deleting the tag (\(\hat{b}, p\)) from the field \textit{answers} (5.2). This update is necessary to store that the facilitator doesn’t wait for an answer of \(\hat{b}\) any more.

(4) \texttt{message=ask-one}(\(\hat{b}, \hat{a}, p\)):
   (4.1) \texttt{send}(b, \texttt{ask-one}(\(\hat{b}, \hat{a}, p\))).
   (4.2) \texttt{become}(C', \texttt{settag}(\(\hat{b}, p\))) +

(5) \texttt{message=updandfrw}(\texttt{tell}((\texttt{self}, \hat{b}, p)) \land \texttt{tag}(\hat{b}, p)):
   (5.1) \texttt{send}(a, \texttt{tell}((\texttt{self}, \hat{b}, p))).
   (5.2) \texttt{become}(C', \texttt{deltag}(\hat{b}, p)) +

<table>
<thead>
<tr>
<th>Function</th>
<th>Operates on</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{settag}((\hat{b}, p))</td>
<td>\textit{answers}</td>
<td>Returns a new facilitator state where the field \textit{answers} includes a record with the agent (\hat{b}) and the query (p).</td>
</tr>
<tr>
<td>\texttt{deltag}(L, p)</td>
<td>\textit{answers}</td>
<td>Deletes the tag ((L, p)), where (L) is a list of agents.</td>
</tr>
</tbody>
</table>

\textit{Ask-everybody and ask-first} messages. When a facilitator receives these messages, it consults its database and forwards an \textit{ask-one} message to all the interested agents (6.1)(7.1). In order to forward a message the facilitator uses the \texttt{forward} primitive which implements a multicast interaction mechanism (see next paragraph). If the message is \textit{ask-everybody} (6) then the facilitator will intercept all the answers, registering them in its local state and forwarding them to the associated kb-actor (\texttt{updandfrw} message) (8). This is needed to implement the \textit{all-answers} predicate. On the other hand, if the message is \textit{ask-first} (7) then the facilitator forwards the first answer he receives and discards all the others (9). Both \textit{ask-everybody} and \textit{ask-first} update the \textit{answers} data structure (by means of \texttt{become} primitive (6.1)(7.1)) to deal with incoming \texttt{tell} messages.

(6) \texttt{message=ask-everybody}(\(\hat{a}, p\)):
   (6.1) \texttt{forward}(x, \texttt{getcomp}(p), \texttt{ask-one}(x, \(\hat{a}, p\))).\texttt{become}(C', \texttt{setalltag}(p)) +

(7) \texttt{message=ask-first}(\(\hat{a}, p\)):
   (7.1) \texttt{forward}(x, \texttt{getcomp}(p), \texttt{ask-one}(x, \(\hat{a}, p\))).\texttt{become}(C', \texttt{setfirsttag}(p)) +

(8) \texttt{message=updandfrw}(\texttt{tell}((\texttt{self}, \hat{b}, p)) \land \texttt{alltag}(p)):
   (8.1) \texttt{send}(a, \texttt{tell}((\texttt{self}, \hat{b}, p))).\texttt{become}(C', \texttt{updalltag}(p, \(\hat{a}\))) +

(9) \texttt{message=updandfrw}(\texttt{tell}((\texttt{self}, \hat{b}, p)) \land \texttt{firsttag}(p)):
   (9.1) \texttt{send}(a, \texttt{tell}((\texttt{self}, \hat{b}, p))).\texttt{become}(C', \texttt{delfirsttag}(p)) +

<table>
<thead>
<tr>
<th>Function</th>
<th>Operates on</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{getcomp}(p)</td>
<td>\textit{fnames}</td>
<td>Retrieves the list of all the agents which are able to deal with proposition (p) and returns the list of the associated facilitators.</td>
</tr>
<tr>
<td>\texttt{alltag}(p)</td>
<td>\textit{answers}</td>
<td>Returns true if an alltag on (p) has been set.</td>
</tr>
<tr>
<td>\texttt{setalltag}(p)</td>
<td>\textit{answers}</td>
<td>Returns a new facilitator state where the field \textit{answers} includes a record with the query (p) and the list of all the agents which have this competence.</td>
</tr>
<tr>
<td>\texttt{updalltag}(p, (\hat{a}))</td>
<td>\textit{answers}</td>
<td>Returns a new facilitator state where \textit{answers} contains the fact that a reply concerning proposition (p) has been received.</td>
</tr>
<tr>
<td>\texttt{setfirsttag}(p)</td>
<td>\textit{answers}</td>
<td>Returns a new facilitator state where the field \textit{answers} includes a firsttag with the query (p).</td>
</tr>
<tr>
<td>\texttt{firsttag}(p)</td>
<td>\textit{answers}</td>
<td>Returns true if a firsttag on (p) has been set.</td>
</tr>
<tr>
<td>\texttt{delfirsttag}(p)</td>
<td>\textit{answers}</td>
<td>Deletes a firsttag on (p).</td>
</tr>
</tbody>
</table>
The multicast protocol. 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:

\[ P := \text{forward}(x, nl, m).P \]

and the operational semantics by adding the axiom:

\[ a[\text{forward}(x, nl, m).P] \xrightarrow{T} a[P] \cup \{a_1, m\} \cup \ldots \cup \{a_n, m\} \]

Our idea for implementing the program \text{forward}(x, nl, m).P in a term \llbracket \text{forward}(x, nl, m).P \rrbracket of the initial algebra is to create a new actor whose behaviour (FWD) is to execute the forward of a message (see Figure 5). When this actor finishes to send all the messages it terminates (correctly) its execution (i.e., becomes the empty term \(\textbf{0}\)).

\[
\llbracket \text{forward}(x, nl, m).P \rrbracket \overset{\text{def}}{=} \text{create}(d, \text{FWD}, \{\}).\text{send}(d, \text{frw}(x, nl, m)).P \quad d \text{ fresh}
\]

\[
\text{FWD} \overset{\text{def}}{=} \begin{array}{ll}
\text{message}=\text{frw}(x, nl, m) \land \neg \text{empty}(nl): & \\
\text{send}(\text{self}, \text{frw}(x, \text{rest}(nl), m)).\text{send}(\text{1st}(nl), m[\text{1st}(nl)/x]). & \\
\text{become}(\text{FWD}) & \\
\text{message}=\text{frw}(\_, nl, m): & \sqrt{
\end{array}
\]

\begin{figure}[h]
\centering
\begin{align*}
\text{(1)} & \quad \text{create} & \quad \text{send} & \quad \text{become} \\
\text{FWD} & \quad \begin{array}{ll}
\text{message}=\text{frw}(x, nl, m) \land \neg \text{empty}(nl): & \\
\text{send}(\text{self}, \text{frw}(x, \text{rest}(nl), m)).\text{send}(\text{1st}(nl), m[\text{1st}(nl)/x]). & \\
\text{become}(\text{FWD}) & \\
\text{message}=\text{frw}(\_, nl, m): & \sqrt{
\end{array}
\end{align*}
\caption{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 \(P\).}
\end{figure}

Register and unregister messages. The register (10) and unregister (12) messages are forwarded to all the other facilitators in the system (10.1)(12.1). The names of the forwarded messages are dregister and dunregister. When a facilitator receives dregister (11) it updates its state adding the competence stored in the message. Analogously, when a facilitator receives dunregister (13) it removes from its state the competence stored in the message.

\begin{align*}
(10) & \quad \text{message}=\text{register}(\hat{b}, p): \\
(10.1) & \quad \text{forward}(\_, \text{frames}, \text{dregister}(\hat{b}, p)).\text{become}(C', \text{setcomp}(\hat{b}, p)) + \\
(11) & \quad \text{message}=\text{dregister}(\hat{b}, p): \text{become}(C', \text{setcomp}(\hat{b}, p)) + \\
(12) & \quad \text{message}=\text{unregister}(\hat{b}, p): \\
(12.1) & \quad \text{forward}(\_, \text{frames}, \text{dunregister}(\hat{b}, p)).\text{become}(C', \text{delcomp}(\hat{b}, p)) + \\
(13) & \quad \text{message}=\text{dunregister}(\hat{b}, p): \text{become}(C', \text{delcomp}(\hat{b}, p)) +
\end{align*}
**Function** | **Operates on** | **Description**
--- | --- | ---
setcomp(\(a, p\)) | competence | Adds to the competence data structure the information that agent \(a\) is able to deal with proposition \(p\). Removes agent \(a\) from the competence list of \(p\).

defcomp(\(a, p\)) | competence | 

---

**All-answers message.** The request all-answers(\(\dot{a}, p\)) is sent from the kb-actor to the facilitator to know whether the replies relative to a given request \(p\) have been received or not. When the facilitator receives this message it checks the replies about \(p\) and then communicates this control to the kb-actor. If all the replies have been received (14), it sends an allanswersyes message (14.1), otherwise (15) it sends an allanswersno message (15.1).

Note the fault tolerant behaviour of the facilitator. If it hasn’t received all the answers (15), it updates its state removing from the list of all the agents which have to answer about \(p\) the agents suspected by the failure detector. This is done by means of a become primitive which uses the function cleanag(\(p, failures\)) to update the state (15.1).

(14) message=allanswers(\(\dot{a}, p\)) \& testalltag(\(p\)):
(14.1) send(\(a\), allanswersyes).become(C\(I\), cleanalltag(\(p\))) +
(15) message=allanswers(\(\dot{a}, p\)) \& ¬testalltag(\(p\)):
(15.1) send(\(a\), allanswersno).become(C\(I\), cleantag(\(p, failures\))) +

**Function** | **Operates on** | **Description**
--- | --- | ---
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.
cleantag(\(p, \hat{b}\)) | 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 cleantag(\(p, list\)).

---

**The creation protocol.** When a facilitator receives the start(\(b_\dot{f}\)) message (16) it initialises the state of the new facilitator \(b_\dot{f}\) sending an init message (16.1). Then it updates the state of the detector sending an addname message (16.2): when the detector will receive this message it will update its state adding the name of the new detector. Finally the facilitator executes a become primitive to update its state adding \(b_\dot{f}\) to the list fnames (16.3).

(16) message=start(\(b_\dot{f}\)):
(16.1) send(\(b_\dot{f}\), init((updfnames(self), [], [], failures))).
(16.2) send(\(a_\dot{d}\), addname(\(b_\dot{d}\))).
(16.3) become(C\(I\), updfnames(\(b_\dot{f}\))) +

**Function** | **Operates on** | **Description**
--- | --- | ---
updfnames(\(b_\dot{f}\)) | fnames | Returns a new list of facilitators obtained inserting the facilitator \(b_\dot{f}\) in fnames.

---

**The cloning protocol.** The behaviour of a facilitator that receives a clone message is to initialise the states of the facilitator and detector of the newly created agent. This is done sending an init message to the facilitator (17.1) and an addname message to the detector (17.2). Subsequently it updates its state adding the cloned facilitator to fnames and storing the competencies of the new agent (17.3).

(17) message=clone(\(b_\dot{f}, \dot{a}\)):
(17.1) send(\(b_\dot{f}\), init((updfnames(self), updcomp(\(b_\dot{f}, \dot{a}\), [], failures))).
(17.2) send(\(a_\dot{d}\), addname(\(b_\dot{d}\))).
The termination protocol. When a facilitator receives an *halt* message it first forwards a *stop* message to all the facilitators it knows and then terminates (18.1). When another facilitator receives a *stop* message (19) it deletes the agent $\hat{b}$ from its state (19.2). A message to the detector actor is also sent to update its state (19.1).

(18) \hspace{1cm} \text{message}=\text{halt}:
(18.1) \hspace{1cm} \text{forward}(\_\text{fnames}, \text{stop}(\text{self})).\text{send}(a_d, \text{halt}).\sqrt{+}
(19) \hspace{1cm} \text{message}=\text{stop}(b_f):
(19.1) \hspace{1cm} \text{send}(a_d, \text{delname}(b_d)).
(19.2) \hspace{1cm} \text{become}(C^f, (\text{delfnames}(b_f), \text{delcomp}(b_f), \text{delanswers}(\hat{b}))). +

The initialization protocol. In the initialisation protocol the facilitator starts the activity of the local detector actor (20.1) and then changes its own state using the parameters received in the *init* message (20.2). We suppose that the translation between the name of a facilitator, say $b_f$, and the name of a detector $b_d$ is automatic and each facilitator is able to perform this translation. Thus if the parameter $e_1$ is a list of facilitator names then they are translated in detector names when the list is sent to the detector.

(20) \hspace{1cm} \text{message}=\text{init}(e_1, e_2, e_3, e_4):
(20.1) \hspace{1cm} \text{send}(a_d, \text{init}(e_1)).
(20.2) \hspace{1cm} \text{become}(C^f, (e_1, e_2, e_3, e_4)) +

Ask-best message. When a facilitator receives an *ask-best*($L, \hat{a}, p$) message (21), it sends an *ask-one* message to the first agent in the list $L$ and updates its state by means of a *become* primitive (21.1). This update is necessary to store that the facilitator has to wait an answer from that agent. If the list $L$ is empty (22) the message *askbestno* is sent to the local kb-actor to inform the unsuccessful of the *ask-best* primitive (22.1). Instead, when a facilitator receives an *updandfrw* message from the first agent of the list $L$ (23), then it informs the kb-actor of the answer and it updates its state removing the tag $(L, p)$ from the *answer* data structure (23.1).

(21) \hspace{1cm} \text{message}=\text{ask-best}(L, \hat{a}, p) \land \neg\text{empty}(L):
(21.1) \hspace{1cm} \text{send}(\text{1st}(L), \text{ask-one}(\text{1st}(L), \hat{a}, p)).\text{become}(C^f, \text{setbesttag}(L, p)) +
(22) \hspace{1cm} \text{message}=\text{ask-best}(L, \hat{a}, p) \land \text{empty}(L):
(22.1) \hspace{1cm} \text{send}(a, \text{askbestno}).\text{become}(C^f, \text{deltag}(L, p)) +
(23) \hspace{1cm} \text{message}=\text{updandfrw}(\text{tell}(\text{self}, b, p)) \land \text{firstbest}(b, (L, p)):
(23.1) \hspace{1cm} \text{send}(a, \text{tell}(\text{self}, b, p)).\text{become}(C^f, \text{deltag}(L, p)) +
setbesttag(L,p)
firstbest(\b,L,p))

Function | Operates on | Description
--- | --- | ---
setbesttag(L,p) | answers | Adds the tag (L, p) to the answer data structure. Returns true if \b is the first agent in the list L of the tag (L, p). Otherwise the function returns false.

Update from the failure detector. The list failures is dynamically updated when the facilitator receives the updfail (26) and updnofail (27) messages from the detector-actor. The programs (24) and (25) characterize the fault tolerant behaviour of the facilitator. If it is waiting for an answer from an agent and that agent is suspected to have crashed, then the facilitator doesn’t wait that answer any more. This behaviour avoids infinite waitings of answers from really crashed agents. So, if the facilitator receives an updfail(c_f) from the detector and it is waiting an answer from the agent c (24), then it informs the kb-actor of the crash (24.1). Then it changes its state adding the new failure and updating the answers data structure (24.2). If the facilitator is waiting for an answer of an ask-best primitive and the agent which has to reply has crashed (25), then it sends the query to the next agent in the list L, according to the semantics of the agent primitive (25.1). Successively it informs the kb-actor of the crash and it updates its state (25.2).

(tag(c, p))
addfail(c_f)
remfail(c_f)

Function | Operates on | Description
--- | --- | ---
tag(c, p) | answers | Returns true if a tag (c, p) has been set.
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.

4.5 The detector-actor

The distributed failure detector is formally specified as a dynamic set of local detector-actors \{a_d, b_d, c_d, \ldots\} which run the same actor program. This set may evolve dynamically whenever a new agent is created or an agent terminates its computation.

We extend the failure detector program presented in Section 3.3 to deal with a dynamic set of agents. Instead of storing the list of suspected actors in its own state it sends the relevant information to the local facilitator. Thus, the state of a local detector consists only of a list dnames, which is the list of all the detector actors in the system. Moreover, the facilitator is also responsible of the initialization of the detector.

The failure detector mechanism. The distributed failure detector mechanism is based on the following assumptions:

- \(a_f\) has crashed \(\iff a\) has crashed \(\iff a_d\) has crashed \(\iff \hat{a}\) 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 \(a_d\) is shown in Figure 6 and is realized as a guarded program of the form:

\[
C^d = e_1 : P_1 + \ldots + e_n : P_n
\]  

UBLCS-2003-12
When a detector receives an initialisation message from the local facilitator (1) it updates its state and starts checking all the agents in the system (1.1)(2). The result of each check is sent to the local facilitator to update the list of failures (2.2)(2.3). The detector executes this program until it receives a halt message from the local facilitator. If this event occurs then the detector stops its execution forever (6). The state of the detector is dynamically updated when a new agent is created or cloned. Indeed, when a detector receives the messages addname (4) and delname (5) from the local facilitator it updates its state adding or deleting the new detector respectively.

\[ C^d \overset{def}{=} \]

(1) message=init(dnames):
   send(self, pingall(x, dnames)).become(C', addnames(dnames)) +
(1.1) message=pingall(x, nl) \land \neg empty(nl):
   send(self, pingall(x, rest(nl))).ping(1st(nl), y).
(2) (y=t): send(a_f, updnofail(1st(nl))).become(C') +
(2.2) (y=tf): send(a_f, updfail(1st(nl))).become(C') +
(2.3) empty(nl):
   send(self, pingall(x, dnames)).become(C') +
(3) message=pingall(x, nl) \land empty(nl):
   send(self, pingall(x, dnames)).become(C') +
(4) message=addname(b_d): become(C', addnames(b_d)) +
(5) message=delname(b_d): become(C', delnames(b_d)) +
(6) message=halt:

Figure 6. The failure detector program.

<table>
<thead>
<tr>
<th>Function</th>
<th>Operates on</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addnames(b_d)</td>
<td>dnames</td>
<td>Adds the actor b_d to the list dnames. If the input is a list of actors dlist, then the function updates the state of the detector with dlist.</td>
</tr>
<tr>
<td>delnames(b_d)</td>
<td>dnames</td>
<td>Removes b_d from dnames if it is in the list.</td>
</tr>
</tbody>
</table>

5 Related Work and Applications

The research presented in this paper deals with issues which concern both agent communication languages and open multi agent systems. The most promising application are related to the development of a knowledge layer for the web or the semantic GRID. In this section we discuss the impact of our work in these research areas.

5.1 Agent Communication Languages

Agent communication languages (ACLs) allow agents to effectively communicate and exchange knowledge with other agents despite differences in hardware platforms, operating systems, architectures and programming languages. In the past few years many Agent Communication Languages have been proposed for Multi-Agent Systems (MAS), incorporating specific mechanisms of agent communication. Many of these communication mechanisms are based on speech act theory, which has originally been developed as a basic model of human communication [21]. The more promising ACLs that have adopted the speech act theory are KQML [6] and the FIPA ACL [7]. The goal of these languages is to support high-level, human like, communication between intelligent agents, exploiting knowledge-level features rather than symbol-level ones. They should support knowledge-level programming of MAS [8]: agents should concern with the use, request and supply of knowledge and not with symbol level issues such as the reliability, synchronization of competing requests, the allocation of resources or the physical allocation of agents on a network.

Despite these efforts, an important issue in the research on ACLs is still open and concerns how to deal with possible failures of agents. Indeed, 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. Agent communication languages should provide mechanisms to deal with these events maintaining a knowledge-level characterization of the communication primitives. At least two approaches are possible: to design fault-tolerant communication primitives, or to reflect failures at the knowledge-level providing a set of predicates which allows an agent to know if another agent is still active and reachable.

An additional difficulty depends on the asynchronous nature of the multi-agent systems we are considering. The central assumption about such systems is that we cannot detect the death of an agent.

The main advantage of our approach with respect to current ACLs is that we provide a set of fault tolerant communication primitives which are well integrated at the knowledge-level. On the other hand most of the current ACLs do not provide a clear distinction between conversation and network primitives, these are often considered at the same level. Moreover, failures crashes and fault tolerance are often not present in the specifications.

5.2 Open multi-agent systems

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 to allow for the dynamic integration of new agents into an existing agent system. In such systems, which are referred to 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 is initially unavailable. As was already pointed out by Hewitt and de Jong ([14]) 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.

In our approach we propose a set of communication primitives which implements an anonymous interaction protocol at the knowledge level. This protocol is fully integrated with the dynamic nature of Open Services Architectures. The main advantage of our approach is to demonstrate that different issues, such as high-level communication and fault tolerance, can be successfully integrated in an open system maintaining a clean design of the architecture.

5.3 Applications: the semantic GRID

The GRID [20] is evolving toward an open, service-oriented architecture which provides the basic infrastructure for the integration of web services distributed across the Internet in heterogeneous and dynamic virtual organizations. The Open GRID Services Architecture [19] is an example of infrastructure which defines standard mechanisms for creating, naming, discovering and integrating GRID service instances.

Although several ACLs have been recently developed and there is a proposal for a current standard [7], several foundational issues concerning the integration of ACL with the service oriented view of the semantic GRID are still open. As a result, the design of this knowledge-level layer for the GRID is still an open issue which faces several challenges in integrating knowledge and distributed systems technologies. In particular, it is still not clear how current ACLs can be integrated with semantic web services in a geographically distributed infrastructure.

Our proposal addresses these fundamental issues by providing a first attempt to answer to the following questions:

- can a knowledge-level layer be individuated in the semantic GRID infrastructure?

---

5. Informally, a distributed system is asynchronous if there is no bound on message delay, clock drift, or the time necessary to execute a step. Thus, to say that a system is asynchronous is to make no timing assumptions whatever. Therefore in such systems we cannot distinguish a dead agent from a merely slow one.
is it possible to design a knowledge-level ACL which enable agents to operate in a geographically distributed semantic GRID despite failures or malfunctions of nodes?

In our opinion, the answers to these two questions are both positive. In fact, the design of a knowledge-level layer for a semantic GRID infrastructure should be based on the specification of a fault tolerant knowledge-level ACL. This high-level language will enable agents to retrieve knowledge and to operate despite possible crash failures of geographically distributed semantic web services.

As remarked in this paper, a knowledge-level ACL should concern with the use, request and supply of knowledge and not with symbol level issues such as the reliability, the management of possible crashes of remote sites, synchronization of competing requests, the allocation of resources or the physical allocation of agents on a network [8]. As a consequence, we argue that, to provide a knowledge-level layer for the semantic GRID, we need to extend current ACLs with three basic knowledge-level requirements:

1. knowledge-level one-to-many performatives
2. support for anonymous interaction
3. an hidden unreliable failure detection mechanism.

As shown in this paper, our knowledge-level ACL satisfies all the above requirements and thus it can be considered a first approach to define a knowledge-level infrastructure for the semantic GRID.

6 Conclusions

In this paper we have analyzed the fundamental issue of integrating agent communication languages in open system architectures.

We have proposed a Fault Tolerant Agent Communication Language (FT-ACL) which is based on knowledge-level primitives and which provides an anonymous, content based, communication protocol. FT-ACL includes primitives to retrieve and execute services which are defined as requests of knowledge. We have also provided a formal semantics of the language and a formal specification of the underlying agent architecture.

The formal approach is based on an Algebra of Actors, which is substantially a process algebra based on the actor model. The actor algebra has been extended to model crash failures and their detection (with an unreliable failure detector). The algebra has been used to specify the communication primitive of FT-ACL providing one to many communication patterns and an anonymous interaction protocol which constitutes an enhancement with respect to the current ACL proposals, such as the FIPA ACL (which only exploits one-to-one communication pattern).

A promising application is the use of these concepts to design a knowledge-level layer for the GRID which is evolving towards an open services architecture.

A lot of work is left for future research, both in the field of the semantics of Fault Tolerant ACLs and in the research concerning the algebra of actors. One of our main goals is the formalization of other groups of failures. Following the classification in [18], we could study how to model classes more severe than crash failures. The main aim of this study could be to formalize arbitrary failures and to study their detection and recovery.

An additional benefit (but also a future work) of our actor based approach is that we can import into agent communication research, techniques and results developed from the theory of concurrency, for instance criteria establishing equivalence of agents with respect to communication and concurrency issues.

Finally, we want to provide an implementation of the ACL integrated with the IRS-II system [17], an internet reasoning tool which is able to publish, locate, compose and execute semantic web services.
A The facilitator program

\[ C^f \triangleq \]

\begin{align*}
1 & \quad \text{message} = \text{insert} (\hat{b}, \tilde{a}, p) : \\
(1.1) & \quad \text{send} (b, \text{insert} (\hat{b}, \tilde{a}, p)). \text{become} (C^f) + \\
2 & \quad \text{message} = \text{tell} (\hat{b}, \tilde{a}, p) : \\
(2.1) & \quad \text{send} (b_f, \text{updandfrw} (\text{tell} (\hat{b}, \tilde{a}, p))). \text{become} (C^f) + \\
3 & \quad \text{message} = \text{updandfrw} (\text{tell}(\text{self} f b, p)) : \text{send} (a, \text{tell}(\text{self} f b, p)). \text{become} (C^f) + \\
4 & \quad \text{message} = \text{ask-one} (\hat{b}, \tilde{a}, p) : \\
(4.1) & \quad \text{send} (b, \text{ask-one} (\hat{b}, \tilde{a}, p)). \\
(4.2) & \quad \text{become} (C^f, \text{settag} (\hat{b}, p)) + \\
5 & \quad \text{message} = \text{updandfrw} (\text{tell}(\text{self} f b, p)) \land \text{tag} (\hat{b}, p) : \\
(5.1) & \quad \text{send} (a, \text{tell}(\text{self} f b, p)). \\
(5.2) & \quad \text{become} (C^f, \text{deltag} (\hat{b}, p)) + \\
6 & \quad \text{message} = \text{ask-everybody} (\hat{a}, p) : \\
(6.1) & \quad \text{forward} (x, \text{getcomp} (p), \text{ask-one} (x, \tilde{a}, p)). \text{become} (C^f, \text{setalltag} (p)) + \\
7 & \quad \text{message} = \text{ask-first} (\tilde{a}, p) : \\
(7.1) & \quad \text{forward} (x, \text{getcomp} (p), \text{ask-one} (x, \tilde{a}, p)). \text{become} (C^f, \text{setfirsttag} (p)) + \\
8 & \quad \text{message} = \text{updandfrw} (\text{tell}(\text{self} f b, p)) \land \text{alltag} (p) : \\
(8.1) & \quad \text{send} (a, \text{tell}(\text{self} f b, p)). \text{become} (C^f, \text{updalltag} (p, \tilde{a})) + \\
9 & \quad \text{message} = \text{updandfrw} (\text{tell}(\text{self} f b, p)) \land \text{firsttag} (p) : \\
(9.1) & \quad \text{send} (a, \text{tell}(\text{self} f b, p)). \text{become} (C^f, \text{delfirsttag} (p)) + \\
10 & \quad \text{message} = \text{register} (\hat{b}, p) : \\
(10.1) & \quad \text{forward} (\_ \text{fnames}, \text{dregister} (\hat{b}, p)). \text{become} (C^f, \text{setcomp} (\hat{b}, p)) + \\
11 & \quad \text{message} = \text{dregister} (\hat{b}, p) : \text{become} (C^f, \text{setcomp} (\hat{b}, p)) + \\
12 & \quad \text{message} = \text{unregister} (\hat{b}, p) : \\
(12.1) & \quad \text{forward} (\_ \text{fnames}, \text{dunregister} (\hat{b}, p)). \text{become} (C^f, \text{delcomp} (\hat{b}, p)) + \\
13 & \quad \text{message} = \text{unregister} (\hat{b}, p) : \text{become} (C^f, \text{delcomp} (\hat{b}, p)) + \\
14 & \quad \text{message} = \text{allanswers} (b, p) \land \text{testalltag} (p) : \\
15 & \quad \text{message} = \text{allanswers} (\tilde{a}, p) \land \text{testalltag} (p) : \\
(14.1) & \quad \text{send} (a, \text{allanswersyes}). \text{become} (C^f, \text{cleanalltag} (p)) + \\
16 & \quad \text{message} = \text{start} (b_f) : \\
(16.1) & \quad \text{send} (b_f, \text{init} ((\text{updnames} (\text{self} f), [\_], [\_], \text{failures}))). \\
(16.2) & \quad \text{send} (a_d, \text{adname} (b_d)). \\
(16.3) & \quad \text{become} (C^f, \text{updnames} (b_f)) + \\
17 & \quad \text{message} = \text{clone} (b_f, \tilde{a}) : \\
(17.1) & \quad \text{send} (b_f, \text{init} ((\text{updnames} (\text{self} f), \text{updcomp} (b_f, \tilde{a}), [\_], \text{failures}))). \\
(17.2) & \quad \text{send} (a_d, \text{adname} (b_d)). \\
(17.3) & \quad \text{become} (C^f, (\text{updnames} (b_f), \text{updcomp} (b_f, \tilde{a}))) + \\
18 & \quad \text{message} = \text{halt} : \\
(18.1) & \quad \text{forward} (\_ \text{fnames}, \text{stop} (\text{self} f)). \text{send} (a_d, \text{halt}) . \sqrt{+} \\
19 & \quad \text{message} = \text{stop} (b_f) : \\
(19.1) & \quad \text{send} (a_d, \text{delname} (b_d)). \\
(19.2) & \quad \text{become} (C^f, (\text{delfnames} (b_f), \text{delcomp} (b_f), \text{delanswers} (\hat{b}))). + \\
20 & \quad \text{message} = \text{init} ((e_1, e_2, e_3, e_4)) : \\
(20.1) & \quad \text{send} (a_d, \text{init} (e_1)). \\
(20.2) & \quad \text{become} (C^f, (e_1, e_2, e_3, e_4)) + \\
21 & \quad \text{message} = \text{ask-best} (L, \tilde{a}, p) \land \neg \text{empty} (L) : \\
(21.1) & \quad \text{send} (1\text{st} (L), \text{ask-one} (1\text{st} (L), \tilde{a}, p)). \text{become} (C^f, \text{setbesttag} (L, p)) + \\
22 & \quad \text{message} = \text{ask-best} (L, \tilde{a}, p) \land \text{empty} (L) :
\end{align*}
send(a, askbestno).become(C!, deltag(L, p)) +

message=updandfrw(tell(self, b, p)) ∧ firstbest(b, L, p):

send(a, tell(self, b, p)).become(C!, deltag(L, p)) +

message=updfail(c_i) ∧ tag(c, p):

send(a, tell(c, self, crashed(c, p))).

become(C!, (addfail(c_i), deltag(c, p))) +

message=updfail(c_i): become(C!, addfail(c_i)) +

message=updnofail(c_i): become(C!, remfail(c_i))

References


