ALBA: a Generic Library for Programming Mobile Agents with Prolog

Benjamin Devèze, Caroline Chopinaud and Patrick Taillibert

Thales Airborne Systems
2 avenue Gay-Lussac, 78851 Elancourt - France
{firstname.lastname}@fr.thalesgroup.com

Abstract. This paper presents ALBA, a generic library dedicated to the commissioning of mobile agents written in Prolog. This library offers a handful of mechanisms for autonomous agent creation, execution, communication and mobility, whose implementation strongly respects the principles of robustness, decentralization of data, flexibility and genericity. In this perspective, the following paper mainly focuses on ALBA architecture and implementation with an emphasis on the technical choices which were made to provide these essential features. It therefore presents an innovative migration protocol, a research algorithm of agents solely identified by their names. It exposes some considerations about communication handling in a fully decentralized environment and some ideas towards a distributed modularity of systems. It also highlights an agent model, called Reasoning Threads, that is being used on top of ALBA to program cognitive agents.

1 Introduction

Since the emergence of multiagent systems (MAS), the corresponding research community has grown considerably and has been hard at work to provide agent communication standards mainly based on the speech act theory [2]. Important efforts have been made to formalize the main characteristics of agents, focusing on agent-oriented languages able to describe the behaviour of an intelligent agent. An abundant literature can be found about MAS related concepts like social attitudes, organization, cooperation or autonomy.

Unfortunately, the design of practical tools that can effectively support MAS programming and deployment appears to miss the necessary maturity to be widely adopted and used for large-scale industrial applications. A remaining gap persists between theories and concrete implementations that prevents us from taking the full benefits of this technology.

In order to demonstrate the added-value of the multiagent paradigm and to convince the remaining sceptic researchers and industrials, it is necessary to provide efficient programming constructs that facilitate the implementation of the essential concepts used in MAS. It is now admitted that MAS deal with flexibility, robustness, decentralization, modularity and scalability [21]. This should

be kept in mind when developing new tools in this domain, so as not to alter these valuable qualities.

Despite the numerous approaches and platforms architectures that have been proposed for agents commissioning, there is no general agreement on a particular method that would combine all the advantages of the agent paradigm. Platforms are often too centralized and often rely on imperative object-oriented languages, like Java, for agents implementation, which are not so well suited for this task [18]. It is especially the case for the kind of applications we have in mind in our group, that could be characterized as an attempt to apply the multiagent methodology to what is generally called real-time applications built on multitasked operating systems. These applications, which in our case concern mission systems embedded in aircrafts (sea or ground surveillance, coordinated observation missions by UAV -Unmanned Air Vehicles-, etc.) are generally complex since they not only manage a lot of tasks simultaneously but also rely upon complex algorithms whose duration cannot always be predicted (Artificial Intelligence approaches are more and more often necessary to implement the requirements of the new mission systems in preparation). To explore the various possible ways to make these systems evolve from a multitask to a multiagent perspective, a powerful implementation language capable of rapid agent model experimentation and AI algorithms development was needed. The most simple infrastructure was also needed in order to be able to easily merge our agents in an existing system and prove, without a complete redesign, that the multiagent approach was an alternative to present practices. That is the reason why ALBA has been designed as a library rather than a platform as is most often the case. Mobility was also a point since it makes it really easier to commission our agents on changing environments (all agents can be created on one computer -whose access is easier or devoted to our experiments- and then dynamically moved to the available computers at the time of the experiment).

Section 2 exposes the main reasons that led to the development of ALBA. Section 3 gives a general overview of the main aspects of our system that, in a way, put it apart from the majority of other tools. In section 4 we go thoroughly into some practical considerations about communications handling and in section 5 a dynamic agent search algorithm is detailed. Section 6 explains in depth the migration protocol and offers some views about agents mobility. Section 7 introduces a specific agent model, called Reasoning Threads, that is being used on top of ALBA to program cognitive agents illustrating a possible usage of the library. Section 8 describes some industrial applications already implemented using ALBA and the Reasoning Threads. Finally, sections 9 and 10 draw the main lessons of our proposals and discuss related and future works.

2 Why a New Platform?

Recent years have seen a considerable growth in the number of platforms, with a current total of over 100 products. It is then legitimate to ask why it has been

necessary for us to develop a new one. The first exigence we had was that the platform had to allow the commissionning of agents written in Prolog.

2.1 Why Prolog?

Without exhaustively listing all the qualities of Prolog, the main reasons that naturally led us to use it to implement ALBA and our agents are stated here.

First of all, thanks to its two main mechanisms of unification and resolution and thanks to its efficiency in manipulating tree structures, Prolog is very well suited to deal with artificial intelligence problems and has already proved it in the past. As a declarative language benefiting from the first-order logic expressiveness, it seems to be the perfect candidate to serve as a basis for new agent-oriented programming languages. Moreover, Prolog allows to dynamically modify source code and offers a good environment to implement introspection capabilities. It is also a good choice for incremental verification of systems which are constructed with provability in mind.

Another essential argument, in our concern, was the natural efficiency of Prolog. It permits to develop and test very quickly some new prototypes and ideas. Its inherent productivity constitutes a great benefit in research activities without affecting at all the readability or the maintainability of source codes.

Prolog is an interpreted language and so as with Java, the same source code can run on various platforms and operating systems which is important to fulfill portability requirements at minimum cost.

Though, it can be argued that the Prolog language is not well equipped to deal with some specific tasks like real-time processing, modern graphical user interface development or efficient implementation of naturally imperative algorithms. Solutions can be found using the bidirectional interfaces to C, C++, Java which are provided with most mature Prolog implementations.

Last but not least, these implementations come with all the functionalities needed for the system to work: TCP/IP sockets, processes handling, Input/Output, etc. Finally, they provide advanced debuggers, efficient garbage collectors, constraint solvers and all the facilities programmers can expect nowadays.

2.2 Related Works

Obviously this part is mainly focused on platforms based on or providing logic programming facilities.

QU Prolog [6] and Ciao Prolog [14] both are Prolog extensions which offer multithreading and multi-machine execution of Prolog code. Agent behaviour programming is done thanks to production rules but other models can be implemented. Both offer also a blackboard for memory sharing or synchronization.

Jinni [24] is a platform allowing the programming of agents in BinProlog and Java. Jinni is based on a simple Things, Places, Agents ontology. Things are Prolog term, Places are processes running on computers and Agents are collections

of threads executing a set of goals. The threads and the agents can communicate by using a blackboard and term unification. The threads can migrate between Places to communicate with the other threads and particularly to accelerate the resolution. Jinni can be used, for example, to simulate stock market, with the blackboard allowing agents coordination. Jinni is an interesting platform to program Prolog agents but the blackboard oriented communication is a limitation we prefered to avoid in our context.

Eel [7] also deserves a mention since it implements communicating processes with an original point to point communication through term unification. But asynchronous processes were looked for, as carried out in ALBA.

tuProlog [9] might have been a good candidate since its design enforces interaction which is essential when agent implementation is concerned. Its close integration to Java is also an interesting feature, let alone for programming man-machine interfaces. The TuCSoN architecture [8] which was developped from tuProlog is a good example of what is looking for with ALBA: a programming environment facilitating the implementation of various agent or coordination models adapted to our needs such as the coordination artifact for time-aware agents presented in [11].

Thus, it looks as if several opportunities were offered for Prolog agent commissioning. So, why an industrial as Thales chose exploring new tracks rather than exploiting the existing solutions ? One of the reason was that not all features we had in mind were gathered in a unique platform (mobility, decentralized agent search) but the main point was our need for a robust Prolog basis such as the one offered by SICS with Sicstus Prolog and the existence in the company of a lot of legacy code for artifical intelligence tools (interval constraint propagation, for example) or applications that simply could not be neglected just for the sake of agent programming.

When non-Prolog platforms implementing mobility are concerned, they generally rely on imperative object-oriented languages for agents implementation (and not on Prolog) and are often dedicated to one specific or a limited set of agent models which was not satisfying. To our knowledge, all mobile agent platforms offer a centralization (from a server providing agents management in a same context or machine, to a central server managing agents in different computers). In every case, the migration, the agent creation, the communication are done through a dedicated entity which knows the local agents and its remote equivalents. These principles mainly exist for scalability and security reasons because the platforms are often used in the Web context. Our MAS approach is quite different. Our main objective is to distribute the agents of a given MAS over several computers in order to reduce and adapt the system workload throughout the execution. Moreover, robustness and functioning simplicity are essential. So, we tried to distribute platform specifications and services into the agent through the ALBA library.

3 Overview of ALBA

3.1 Main features

ALBA is a Prolog library dedicated to the commissioning of agents written in Prolog. It uses SICStus Prolog [1] which is a mature and complete Prolog implementation with high performance and industrial qualities.

ALBA offers the basic functionalities expected from a multiagent platform. It brings the necessary mechanisms for agents creation, execution, communication and mobility.

The first noticeable point about ALBA is its complete *decentralization*. It means that the code implementing the platform functionalities is embedded in each agent. In this perspective, neither any central program nor any kind of data centralization are required for it to work. Hence, as it has already been mentioned, ALBA can better be described as a predicate library rather than as a platform. Of course, decentralization raises a lot of problems, especially related to communications handling. A substantial part of this paper is devoted to the practical proposals we have implemented to tackle these issues.

ALBA is also about *genericity*. That means that no assumption is made on the agent models used. Therefore, ALBA can be used with any kind of agent models (Agent-0[22], AgentSpeak[25], BDI[19], etc.). Since, at this stage, the research community has not agreed on a specific universal model which can be used for any kind of applications, and since it is even doubtful that such a model exists, it seems to be the best way to proceed when industrial applications are concerned. Moreover, this approach greatly facilitates experimentations on various models and on the way they can be combined to reach our expected goals. In the same range of ideas, no assumption is made on the language used by the agents to communicate.

Another point of interest is *flexibility*. As a generic low-level tool, the library assures, purposely, a restricted range of basic functionalities. It is a core tool that can be extended at will to provide higher level functionalities, as mentionned in sections 7 and 8.

3.2 General Overview

An ALBA agent is constituted of two parts, the embedded library in charge of all the basic services (messaging, contacts, etc.) and the behaviour of the agent itself that is coded in Prolog. Each agent is an independent Prolog process and has a unique name that fully identifies it in the system. Therefore, in all the paper, the agents will be represented as in Figure 1. All agents can communicate via asynchronous messages.

ALBA has been developed in a multi-machine perspective and, of course, our systems can be distributed on a pool of computers over a network. For the remaining of the paper, the term computer will refer to any computer in a network where an *ALBA daemon* is running. These daemons are in charge of executing on remote machines, the creation or migration functions called by an

Fig. 1. An ALBA Agent

agent and the associated data transfers (see section 6 for more details). Therefore, agents can be created locally or on remote computers and can migrate from a computer to another. A general overview is presented in Figure 2.

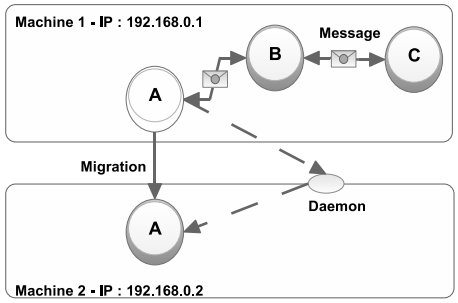

Fig. 2. ALBA Overview

3.3 Towards Distributed Modularity

In ALBA, agents are created from *proto-agents* using the predicate:

`create_agent(+Host, +Name, ?Param, +Contacts, +ProtoAgent, -FullName).`[1]

Proto-agents are to agents what classes are to objects in the object-oriented paradigm. Each agent running in a MAS can be viewed as a specific instance of a proto-agent. For the sake of reusability, proto-agents aim to be as generic as possible. They consist of the Prolog source code describing the "to-be-instantiated" agents behaviour and of any resource files they could need. Each proto-agent is stored as a directory or as an archive file and takes its name from its corresponding directory name or filename. Each newly created agent is given a specific workspace initialized from the content of the proto-agent it is based on. When an agent creates another agent from a proto-agent A, ALBA automatically searches A following the order given in its *proto-agent path*, querying remote daemons when necessary. The proto-agent is then automatically retrieved to localhost as a compressed archive and can be used to launch the new agent.

[1] A few predicates of the library are introduced but generally parameters are not described for being quite self-explicit

To accomplish their tasks, agents may need specific libraries, for example libraries dedicated to image analysis, interval computations, etc. Including these generic libraries in each proto-agent would be costly and nearly unmaintainable, that is why ALBA use quite the same mechanisms as for proto-agents to automatically find and retrieve required libraries which are shared at MAS level. To do so, ALBA offers the following predicates which are encapsulations of their well-known Prolog homologues: `alba_consult(+LibName)`, `alba_compile(+LibName)`, `alba_use_module(+LibName)`.

Now, let us suppose we want to run a MAS to simulate a mission with boats and planes using various libraries. We have nothing in our machine but we know the address of remote servers hosting required data. Provided we just have a daemon running on our computer, we can build a complete customized MAS using proto-agents and libraries coming from various machines, serving as distributed banks of generic agents and resources.

4 Communications Handling

Communications handling is a hard task that, at low-level, needs some knowledge about network protocols that seems far from intelligent agents problematics. However, communication is a fundamental aspect of multiagent paradigm, as being the only way for cognitive agents to share information. Indeed, it is by prohibiting the usage of complex sharing methods (shared memory, semaphores, etc.) that MAS can pretend to reduce the structural complexity of systems.

As stated before agents are able to communicate asynchronously through messages. In this view, ALBA offers direct point-to-point communications with `send_message(+Message, +Recipient)` predicate. Communications relied on TCP/IP sockets which can seem inappropriate for local communications but is required for remote transmissions.

Agents are identified by their names, whose uniqueness is ensured by ALBA using the following naming scheme, which only exploits information locally saved in each agent: *AgentName/SonName/etc.* Note that names are stored and manipulated as Prolog terms, allowing us to deduce immediately from an agent's name the identity of all his ancestors. Another interesting feature of this naming scheme is that it allows the merging of completely distinct MAS into a single one. Indeed, provided that the seed agents of each MAS to merge have a unique name -which is not a severe requirement-, it is clear that there won't be any name clash issues. Therefore, relying on the search method described in section 5, merging two distinct MAS can be done by simply linking one agent from each MAS to each other.

Exchanged messages are Prolog terms which is extremely convenient when it comes to parsing and analyzing their contents. No other assumptions are made about messages contents and, of course, every classical communication languages (KQML[16], FIPA ACL, etc.) can be used.

ALBA users can build up their systems on the following postulate: for the same pair of agents, messages ordering is preserved. More formally, if *m1* and

m2 are two messages sent from A to B, *m1* being transmitted before *m2*, then B will receive *m1* before *m2*. This is ensured by TCP protocol and the library internal mechanisms.

Error treatment is an important aspect of communications handling. ALBA acts as a layer on top of TCP/IP to manage every detail related to communications, such as connections, transmissions, proper disconnections and so on. It also has to deal with any potential low-level errors that may occur. Agent programmers work at a higher abstraction level and must not have to be preoccupied about these kind of considerations. Communications handling in ALBA, can be compared to ordinary postal service. It is, therefore, up to agents to prevent any possible loss of essential information thanks to specific protocols. For example, ALBA comes with acknowledgments facilities, which can be used for synchronous communications if it becomes necessary.

It can be useful for an agent to use some appropriate messages treatment strategies. It becomes nearly inevitable for very solicited agents so as to rationally handle the mass of received messages. Strategies can give the precedence to some specific senders or to a given kind of messages that need to be processed in priority. That is why, in addition to the classical `read_message(-Message, ?Sender, +Timeout)` routine, ALBA provides the predicate `read_all_messages(-Messages, ?Senders)` that returns all the messages available at call time. It is possible to instantiate the variable *Senders* in order to get only the messages sent by given senders. Note that an agent messagebox is stored in memory using Prolog terms, allowing to easily handle advanced requests by unification.

5 Search Method

5.1 Introduction

All agents of the system are identified by their unique name. This is the only information accessible to the end user of ALBA. All communications being based on TCP/IP sockets, the library has to provide internal mechanisms to recover the IP address and port number of an agent from its name. In order to achieve this goal, we refused to use any kind of name servers or matchmaker agent (respectively white and yellow pages) or, more generally, to assign this task to any form of central system that would constitute potential drawbacks for our applications. Relying on a centralized approach would affect the robustness of ALBA because a single fault in this central organ could paralyse all the system. Moreover, excessive centralization constitutes a major bottleneck, as the central entity has to stay aware of every changes occurring in the MAS (agent creation, migration, etc.) and to answer all the queries of the agents willing to communicate. Hence, the central entity has to deal with a very important amount of messages with an overload risk. These issues can be partially solved using several matchmaker entities communicating with each other, ensuring the integrity of their names database and implementing mechanisms of data redundancy to prevent the system to fully depend on the existence of a single entity. This solution

has, though, an important cost and that is the reason why an alternative way was chosen.

To address this problem[2], we aim at taking the best advantage of the natural distributivity of MAS and to exploit information stored locally in each agent. The problem can be viewed as a graph search problem where nodes represent agents and where arcs stand for connections between agents, i.e. an agent A is linked to an agent B if and only if A knows the correct IP address and port of B at a given time T. The problem is more complex than a traditional graph search because, here, the topology of the graph, since it is a model of the MAS, can evolve dynamically during the search. It is also important to understand that, even if agent A has some information about B, we cannot be sure that these data are up-to-date.

Therefore, the search algorithm need to be able to successfully retrieve agents in an unstable graph whose arcs may be wrong, by propagating a wave of messages in the MAS. It needs to fulfill the three following objectives:

1. the wave of messages generated by the algorithm must come to an end, no matter what is the configuration of the MAS
2. if agent A searches agent B, which is in the same connected component, the algorithm must be able to find B
3. the amount of messages sent during the search need to be limited as much as possible

A Naive Algorithm The general idea is very simple. When agent A wants to communicate with agent B and does not know how to reach it, A sends a search message to all its contacts. The search message contains: some necessary information to reach the search initiator (SI), a blacklist of already visited contacts (BL) and the name of the target agent (TA). Of course the algorithm works and fulfills the two first objectives but generates too many messages to be used. Indeed, in the worst case, i.e. N agents interconnected (complete graph) with a search for an agent absent from the MAS, it is straightforward to see that the algorithm induces a wave of $(N-1)!$ messages.

5.2 An Improved Algorithm

The idea, to limit the number of sent messages, is to exploit local information stored in each agent, i.e. its contacts, in order to anticipate a step forward. Thus, when an agent receives a search message it adds itself *and* all its non already visited contacts in the blacklist. Unfortunately, this algorithm does not ensure anymore that an agent will find any agent in its connected component. It is illustrated in Figure 3.

In this example, A wants to reach agent D. A sends a search message to its contacts B and C with A, B and C in the BL. A cannot effectively send the message to C because it has outdated localization information concerning C.

[2] So far, only the replacement of centralized "white pages" is achieved

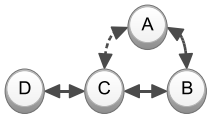

Fig. 3. Error in search

Note that B holds updated position of C and could reach it, but of course it does not even try to do so because C is in the BL. Even if D and A belong to the same connected component, A cannot find D anymore.

It is therefore necessary to add a local error treatment mechanism. An automatic ping pong procedure could have been used to ensure that each contacts are reachable before sending them the search message. It is not reliable and too heavy to be used in large MAS, especially when it appears that, in the majority of cases, there won't be any error to handle. That's why an alternative method has been proposed, described in algorithm 1, which we have called Waves-Search algorithm.

In this method, agents automatically take into account communication errors and resend their search messages with an appropriate corrected blacklist.

Algorithm 1 Waves-Search algorithm

```
1: if OwnName = Target Agent (TA) then                    ▷ I am the searched agent!
2:     Get in contact with the Search Initiator
3: else
4:     if TA ∈ OwnContacts and search message successfully sent to TA then
5:         return
6:     end if
7:     Forwards ← OwnContacts − Blacklist (BL)
8:     UpdatedBL ← BL + OwnName + Forwards
9:     for all Agent ∈ Forwards do
10:        Send updated message with UpdatedBL to Agent
11:    end for
12:    Errors ← List of agents in Forwards that could not receive the search message
13:    if Errors ≠ ∅ then
14:        NewForwards ← Forwards − Errors
15:        NewUpdatedBL ← UpdateBL − Errors
16:        for all Agent ∈ NewForwards do
17:            Send updated message with NewUpdatedBL to Agent
18:        end for
19:    end if
20: end if
```

This algorithm works well for applications involving a reasonable number of agents organized in favorable interconnected topologies. It is also important to understand that the algorithm is launched only a limited number of times

to interconnect two agents at first or as an alternative procedure if an agent has lost some contacts. Though, it suffers some scalability issues and would not be suitable for massive MAS with all possible topologies. It can be viewed as a first step towards a fully effective dynamic search method in a decentralized environment.

The problem is very close to search processes in Gnutella-like unstructured and fully decentralized peer to peer networks and to classical application layer routing protocols, which are active fields of research [10]. A possible improvement would be to adapt Distributed Hash Table based methods like Chord, CAN, Pastry or mobile ad hoc network routing protocols like DSR or AODV to the specificities of the problem. It may imply to soften the second objective offering only guarantees in probability to find an existing agent. Another very promising approach would be to exploit the agents genealogy, which can be deduced from agents name, in order to direct the search very quickly and with bound guarantees on the number of messages sent for all topologies.

6 Migration Protocol

Mobility, i.e. the support to the network transport of agent code and execution state, has become one of the fundamental feature any modern platform should provide. Mobile agent advantages, which are stressed in several papers, such as [4, 17], explain this imperative requirement. Listing only a few of them, agent mobility allows network traffic reduction, dynamic MAS reconfiguration, load balancing and is of great support to improve scalability and fault-tolerance. This section describes the migration protocol used by ALBA and discusses its main characteristics.

6.1 Description

At any time, agents can use the predicate `migrate(+Host)`, to keep on with their work on any remote computer. Note that ALBA provides only the necessary mechanisms for agents mobility. Each agent chooses its target host and the best moment to migrate relying upon migration strategies established by the developer. This is of course reasonable considering that these strategies are application-dependant and stand at a higher abstraction level than ALBA. To achieve the migration task, ALBA proceeds as described in Figure 4.

1. The migrant wants to move to a remote host.
2. A clone of the migrant is created on the remote host.
3. The clone creates a connection with the migrant contacts, the migrant stops its activity and only forwards messages to its clone.
4. The connections between the migrant and its old contacts are cut.
5. The migrant process destroys itself, the clone has replaced it on the remote target.

Now that only a general overview of the migration protocol has been given, it is necessary to describe what happens in each agent playing a part in the procedure.

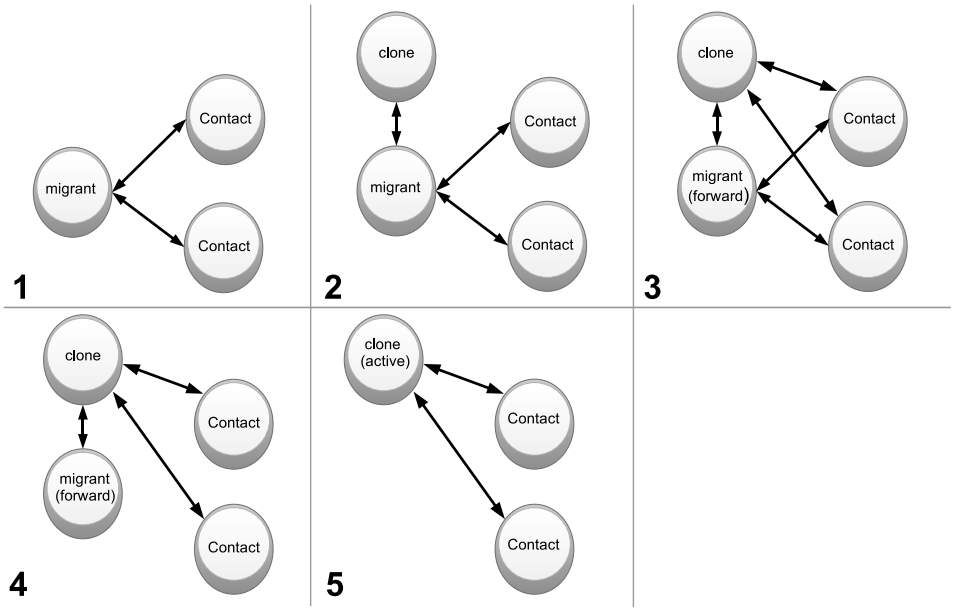

Fig. 4. Migration Protocol

From the migrant perspective The migrant first creates a clone of itself on the target computer. In practice, the workspace of the migrant is transferred by the ALBA layer in the migrant, to the remote ALBA daemon as a compressed archive file and the remote daemon launches the clone agent. The migrant reads all its pending messages and transfers all its messages and ALBA related internal data, such as its contacts, directly to its clone. All messages received during this phase are forwarded by the migrant to the clone. Note that all the forwarded messages are encapsulated properly to inform the clone of its real senders. Then, the migrant immediately sends an *end of migration* message to the clone and closes itself.

From the clone perspective The clone is launched from the same source code as the migrant. It first connects to its father, i.e. the migrant, which is the only agent of the system aware of its existence. As described in section 4, its internal name has the form *migrant_name/clone(X)*. It then initializes its internal data with those provided by the migrant. As soon as it receives migrant contacts information, it sends them a special internal update message stating that it is the new agent named *migrant_name*. This message is automatically interpreted by the ALBA layer which just replaces migrant information with clone address and port. Upon the reception of the *end of migration* message from the migrant, the clone changes its internal name to *migrant_name* and

calls the *restart* predicate that has to be written by the agent developer and define the first behaviour of the restarted agent.

From the migrant contacts perspective From the migrant contacts everything is transparent. There is only a hidden substitution in their internal data from the migrant address and port to the clone address and port.

6.2 Discussion

One of the main interest of this protocol is that the migrant and its clone are running together during a very short period of time and that no messages are lost during this transitory phase. Indeed, if one of the migrant contact sends the migrant a message when it is still running, it will forward it to the clone. If the migrant is already closed but the contact has not already received the special internal update message, the message will be queued and sent as soon as the clone contacts it.

At first, it seems that ALBA provides only weak mobility because no migration of execution state is involved, the migrating agents are explicitly restarted at their destination. However, ALBA comes with the two following routines: `put_into_luggage(+Name, +Value)` and `get_from_luggage(+Name, ?Value)`, allowing to save and restore data in a specific part of the memory which is automatically transferred during a migration. These predicates are callable at any time during agent execution and represent a convenient way to manage what can be viewed as a *migration luggage*. The agent model described in section 7 can be fully defined by its internal data. Thus, using their migration luggage properly, as described in section 7, agents implemented with this model are able to completely resume their execution after a migration, which becomes a transparent procedure. Therefore, the migration strength depends of the agent model which is used, the library offers strong migration at agent level if the model used is *migration compliant*, i.e. if the agent behaviour can be resumed by the sole knowledge of its *luggage*.

7 Reasoning Threads: a Model of Agency

As a low-level library dedicated to the commissioning of agents, ALBA is well suited to serve as the core of various agent model experimentations. Using directly the communication routines offered by ALBA would, of course, be inadequate to implement intelligent agents involved in complex interaction and coordination processes. This section illustrates the genericity of ALBA and shows how it can be used with a specific model based on Reasoning Threads (RTs) to deploy these agents.

7.1 Basic Concepts

Autonomy is a central concept of agency. Following [12], we adopt an operational definition of autonomy stating that *an agent A is autonomous with regard*

to an agent B if and only if B cannot predict definitely A decisions. An agent can decide not to process a message e.g. because it has more important goals to satisfy or because of workload. Therefore, agents need to be able to react accordingly if they do not receive expected answers and must foresee commitments breaking. Agents can not predict the exact behaviour of other agents, but they can delimitate classes of alternative behaviours that can be expected. As a consequence, agents plans need to be conditional over possible actions/reactions of other agents. Thus, thinking agents in term of autonomous entities constitutes a way to improve fault tolerance at the source, dealing with loss of messages, death of other agents or machine overload. Another central concept to reduce the structural complexity of systems is the *limited dependency*, we assume that interactions with other agents and the environment only take place by exchanging messages, prohibiting memory or resource sharing without an agent mediation. These two principles influence a lot the agentification of systems and the implementation of agents.

Messages play a central role. They are the only information agents perceive about others activities. Therefore, it has been necessary to provide an architecture that makes messages analysis and handling easier. As an agent gets dynamically involved in many interaction processes with various agents it has also been necessary to provide some mechanisms to properly handle different contexts simultaneously. As a matter of fact, when a human receives his mail he generally roughly sorts it on the basis of the sender and content of each received item and then link each received letter with a previous context or create new contexts to handle further mail exchanges attached to new topics. In the same range of idea, threads in forum allow the sorting of messages according to their object and topic and can consequently handle multiple contexts simultaneously. These metaphores were of great influence for the RT approach.

7.2 Description

The RT library acts as a layer on top of ALBA encapsulating the low-level routines offered by the library which are not available anymore to the agent programmer (Cf. Fig. 5).

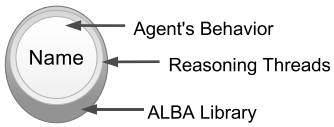

Fig. 5. An ALBA Agent based on RT

Each RT can be viewed as a context. A RT template is described as an extended finite state machine representing a procedural knowledge associated to a context. Thus, a RT conveys some explicit knowledge about the interaction

processes taking place among agents in order to reach a goal in a specific context. When a RT template is instanciated, a local memory is created to store the data relevant to this context. A global memory is accessible to all RTs. All messages arrive to a *switch* which is in charge of the messages routing to the relevant RTs according to a set of grammar rules that work on the sender, the syntax and the content of incoming messages (Cf. Fig. 6). If a message can not be filtered by the switch, it is automatically directed to the default RT which plays a crucial role to identify new contexts and to handle unknown messages.

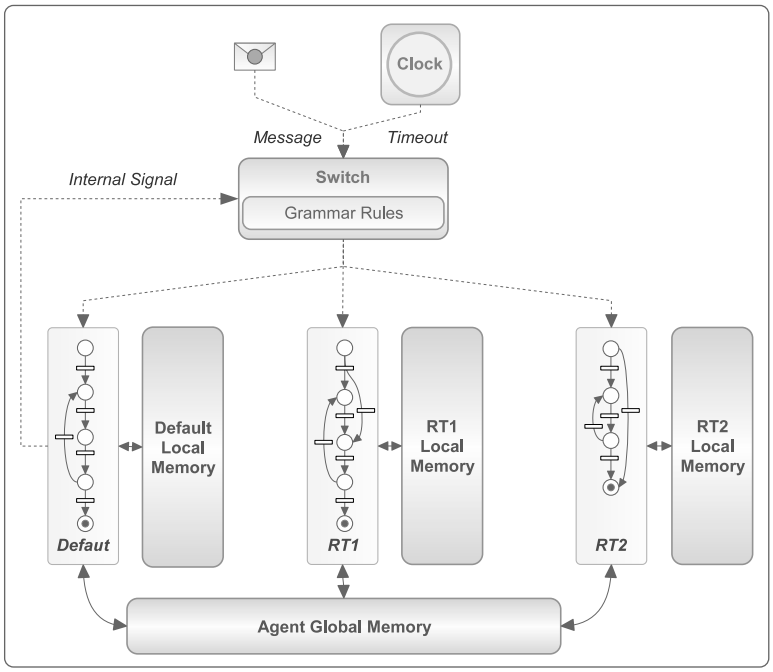

Fig. 6. Architecture Overview

RTs are dynamically instanciated and destroyed according to the evolution of the system. As an agent may be simultaneously involved in multiple interaction processes, multiple RTs can of course be instanciated at the same time. For example, if an agent answers to various proposals from different agents, it would possibly have multiple instances of RTs describing the Contract Net protocol [23] running at the same time to handle these interactions. Grammar rules, RT templates and currently instanciated RTs can be modified, added or deleted at run time allowing to dynamically adapt the behavior of running agents.

A RT consists of a set of states S_i, an initial state $begin \in S_i$, a final state $end \in S_i$, a set of state transition rules R_i, a local memory Mem_i and each state

$s_i \in S_i$ is associated with a timeout $t_{s_i} \in T_i$ (in seconds or *off*) modifiable at runtime; that is $RT_i = \langle S_i, T_i, R_i, Mem_i \rangle$.

A transition rule consists of a condition and an associated action. The invocability conditions of a rule consist of the RT current state, an incoming event and a facultative filter that has access to local and global memories. An event can be (1) an incoming message from another agent (`msg(Message, Sender)`), (2) an internal signal (`signal(Signal, RT_ID)`) or (3) a timeout. Internal signals allow communications between various RTs which can be useful for example to resolve conflicts between various contexts. Each rule specifies an action that is executed when it is fired and the next state of the RT. An action can consist of any Prolog code including calls to encapsulated ALBA routines (message sending, migration, agent creation) and calls to RT related routines (grammar rule modifications, RT modifications, RT invocations, memory modifications). Each rule is written as a Prolog clause:

```
rt(Type, State-in, State-out, Event, Action) :- Filter.
```

Agents plans may be incomplete or inaccurate and the knowledge to extend or correct them may become available only at runtime. Therefore, agents need to be able to extend and modify their existing plans and also to build new plans dynamically. For this reason, `State-out` can be a free variable in order to incrementally build a plan at runtime.

7.3 Execution Model

When the switch is initialized, the default RT and the agent global memory are created. The interpreter keeps up to date RTs related data like the names, current states, current timeouts and local memories of currently instanciated RTs. It also maintains a queue of pending events.

The execution cycle works as follows:

Algorithm 2 Execution Cycle

```
1: while there is an instanciated RT do
2:     compute the timeout to apply (minimum of the remaining timeouts of the currently instanciated RTs)
3:     read all pending messages during at most timeout seconds
4:     if there are messages to handle then
5:         apply the filtering_strategy routine to select the message m to process
6:         use the grammar rules on m to compute the RTs to trigger
7:     else
8:         wake up the RTs in timeout
9:     end if
10:    sequentially execute actions of the transitions that have been fired
11:    remove RTs in state "end"
12: end while
```

Since state transition rules are fired sequentially, it is not necessary to use mutual exclusion mechanisms to protect accesses to the global memory of the agent, memories will stay consistent while an RT action is executed. This approach reduces the systems complexity still allowing to handle multiple contexts simultaneously.

Note that the `filtering_strategy` predicate can be redefined to implement various strategies to give precedence to specific messages or senders according to the context. The default behaviour is a queue.

7.4 Mobility

As described in section 6, ALBA provides strong migration at agent level if the model used is *migration compliant* as is the case for the RT approach. Indeed, when the migration routine is called in the body of an action, the interpreter ensures that the migration procedure is the last operation executed just after the branching to the new state, leading to the following steps:

1. the states, memories and related data are stored in the agent migration luggage
2. the migration predicate of ALBA is explicitly invoked
3. the revelant data are restored on target host when migration is achieved
4. the timeouts of currently instantiated RTs are updated
5. the execution is resumed

7.5 Discussion

Related works can be found in the field of coordination languages with COOL [3] or AgenTalk [15] which respectively introduced the notions of *conversations* and *scripts* which share the same philosophy as the RTs. These two languages do not use grammar rules for messages filtering but propose to use user maintained or automatically maintained identifiers to route incoming messages to conversations. The grammar rules of the RT approach are much more flexible since they can be modified at runtime and allow to route a given event to multiple running RTs. It also contributes to the clear definition of contexts based on the syntax and semantics of incoming events. Moreover, provided the introspection and metaprogramming facilities offered by Prolog and the language used to describe RTs, we defend the idea that it will be easier to extend the mechanisms introduced by COOL and AgenTalk to develop the flexibility and the introspection abilities of our agents.

A *script*, a *conversation* or a *RT* can be viewed as a procedural knowledge for an agent to reach a certain goal. In this perspective, these approaches have much in common with Procedural Reasoning System (PRS) [13]. However, PRS is mainly focused on a single goal-directed agent whereas the coordination languages are focused on social aspects describing protocols among agents. Therefore, we now aim at combining these approaches and extending the RT model with explicit goals and planification abilities in order to improve the proactivity

of our agents which is so far limited to timeouts and internal signals processing. Thus, further works will naturally be focused on ways to achieve a good balance between goal-directed and reactive behavior in a timely fashion.

Looking at the execution cycle of the system, the question of actions duration is clearly of great importance to answer appropriately to incoming events in a timely fashion. In this perspective, [11] propose to delegate "long" computations to computational artifacts controlled by agents.

8 Applications

ALBA and the RT approach have already been used in various applications of our department. Only one of them is mentioned here: *Interloc*. Interloc is a software for mobile marine targets localization. In Interloc, planes seek to detect boats while remaining stealth, i.e. without using their own radar, but by exploiting the targets emissions to deduce their positions. The system is implemented as a MAS. Each boat is represented by an agent, just as each plane. Another agent manages the graphical interface, another one makes measures and an agent by plane is in charge of localization computations. Therefore, a substantial number of agents (10 to 25) are to run and interact at the same time. Interloc was a perfect testbed application to validate ALBA and especially its migration protocol. Indeed, considering the significant number of agents running simultaneously, migration was interesting for load balancing purposes.

As explained, as a low-level tool, ALBA provides purposely a very limited set of functionalities. Thus, to carry out these applications, a lot of useful tools have been developed on top of it, which proved the flexibility and extensibility of the library. For instance, an agent was developed to facilitate human agents interactions with active MAS via a graphical interface. It mainly allows to create or kill agents and comes with a console to easily communicate with running agents by sending them messages.

ALBA has been well tested in practice and has proved to be efficient in achieving its tasks. The RT approach has proved to be extremely easy and convenient to use thanks to the natural separation of contexts which allows the programmer to focus on local problems attached to specific contexts. The ALBA library and its related tools are now mature enough to be used in larger scale industrial applications.

9 Future Works

Even though, ALBA is already quite functional, several aspects have to be improved and new functionalities need to be added.

The increasing development of agents mobility and the distribution of MAS over heterogeneous networks raised the question of security. Therefore, to be deployed in untrustworthy environments, ALBA needs to support cryptography mechanisms to provide communications encryption and agents authentication. In the same perspective, all manipulated archives need to be encrypted too.

In our applications, agents are to interact with entities that are not really agents but just some runtime devices providing a specific kind of function or service. There is, for example, an entity that is used by agents to display graphical interfaces. This is not a common agent, it does not use ALBA and cannot fully interact with other agents. Therefore, ALBA needs to properly handle this kind of entities introduced in [20], as a first-class abstraction in MAS under the name "artifacts".

We have also begun to develop a generic library allowing to define events linked with any chosen predicates. Programmers can use these predefined events to inject their own code on specific points of interest. The library relies on Prolog introspection mechanisms. It would allow to easily customize ALBA with external files and without modifying its core. It could also be used to dynamically control agents as in [5]. More advanced studies are to be made to explore the potential benefits of this library.

10 Conclusion

We have presented in this paper a generic Prolog library called ALBA, dedicated to MAS deployment. We described thoroughly its architecture and implementation with an emphasis on the technical choices made to provide robustness, decentralization, flexibility and modularity. With a strong respect for these features, we introduced an innovative migration protocol, an agent research algorithm and some considerations about communications handling. We also highlighted some ideas to achieve a distributed modularity of agents. Relying upon the described mechanisms, it is already possible to merge completely distinct MAS, to tackle on-line repairing of agents or to stop any agent for some time and relaunch it later, minimizing the impact on the rest of the system. Part of our current work is focused on these experimentations, on ALBA improvements and on its applications.

Now that we have a usable library dedicated to MAS commissioning, our main concern is also to explore the best ways to express autonomous agents behaviour. A preliminary work has been presented in this paper with the description of an agent model based on Reasoning Threads. We now aim to extend this model and to propose a new declarative high-level agent-oriented programming language built on top of Prolog.

References

1. Available at `http://www.sics.se/isl/sicstuswww/site/index.html`.
2. John Langshaw Austin. How to Do Things with Words. *Clarendon Press*, 1962.
3. M. Barbuceanu and M. S. Fox. Cool: A language for describing coordination in multiagent systems. In Victor Lesser and Les Gasser, editors, *Proceedings of the First International Conference oil Multi-Agent Systems (ICMAS-95)*, pages 17–24, San Francisco, CA, USA, 1995. AAAI Press.
4. David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile Agents: Are They a Good Idea? Technical report, IBM Research Division Report, 1995.

5. Caroline Chopinaud, Amal El Fallah Seghrouchni, and Patrick Taillibert. Prevention of Harmful Behaviors within Cognitive and Autonomous Agents. In *Proc. of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 205–209, August 2006.
6. K. Clark, P.J. Robinson, and R. Hagen. Multithreading and message communication in Qu-prolog. *Theory and Practice of Logic Programming*, 1(3), 2001.
7. Torbjrn S. Dahl. The eel programming language and internal concurrency in logic agents. In *the Proceedings of the Workshop on Multi-Agent Systems in Logic Programming, (ICLP'99)*, Las Cruces, New Mexico, November 29 - December 4 1999.
8. Enrico Denti and Andrea Omicini. From tuple spaces to tuple centers. *Sci. Comput. Program.*, 41:277–294, 2001.
9. Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm java-prolog integration in tuprolog. *Sci. Comput. Program.*, 57(2):217–250, 2005.
10. Gang Ding and Bharat K. Bhargava. Peer-to-peer file-sharing over mobile ad hoc networks. In *PerCom Workshops*, pages 104–108, 2004.
11. Cédric Dinont, Emmanuel Druon, Philippe Mathieu, and Patrick Taillibert. Artifacts for time-aware agents. In *Fifth Int. conf. on Autonomous Agents and Multiagents Systems (AAMAS 06)*, Hakodate, Japan, 8 - 12 May 2006.
12. Mark d'Inverno and Michael Luck. Understanding autonomous interaction. In *ECAI*, pages 529–533, 1996.
13. M. Georgeff and A. Lansky. Procedural knowledge. *Proceedings of the IEEE (Special Issue on Knowledge Representation)*, 74:1383–1398, 1986.
14. Daniel Cabeza Gras and Manuel V. Hermenegildo. The ciao module system: A new module system for prolog. *Electr. Notes Theor. Comput. Sci.*, 30(3), 1999.
15. Kazuhiro Kuwabara, Toru Ishida, and Nobuyasu Osato. Agentalk: Coordination protocol description for multiagent systems. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi–Agent Systems*, page 455, San Francisco, CA, 1995. MIT Press.
16. Yannis Labrou and Tim Finin. A Proposal for a New KQML Specification. Technical Report TR CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, February 1997.
17. Danny B. Lange and Mitsuru Oshima. Seven Good Reasons for Mobile Agents. *Commun. ACM*, 42(3):88–89, 1999.
18. James Odell. Objects and Agents Compared. *Journal of object technology*, 1(1):41–53, 2002.
19. A. S. Rao and M. P. Georgeff. BDI-Agents: from Theory to Practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.
20. Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Programming MAS with Artifacts. In *Proc. of the Third International Workshop on Programming Multi-Agent Systems'05*, pages 163–178, July 2005.
21. Pierre-Michel Ricordel and Yves Demazeau. From Analysis to Deployment: A Multi-agent Platform Survey. *LNCS*, 1972:93–105, 2001.
22. Yoav Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1):51–92, 1993.
23. Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, 29(12), 1980.
24. Paul Tarau. Jinni: Intelligent mobile agent programming at the intersection of java and prolog. In *Proceedings of PAAM'99*, London, 1999.
25. D. Weerasooriya, A. Rao, and K. Ramamohanarao. Design of a Concurrent Agent-Oriented Language. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 386–402. Springer-Verlag: Heidelberg, Germany, 1995.