From Interoperability to Cooperation: Building Intelligent Agents on Middleware.

Bruno Dillenseger
CNET, BP 98, F-38243 Meylan cedex, france
`bruno.dillenseger@cnet.francetelecom.fr`

Abstract. As agent technologies are increasingly being involved in telecommunication-related applications, the need for open standards is becoming critical. During the past years, different scientific communities gave birth to different standardization actions, such as the Foundation for Physical Intelligent Agents (FIPA) and the Object Management Group's MASIF (Mobile Agent System Interoperability Facilities). Although they finally share some major targets, the OMG and FIPA current results show their distinct origins, respectively with a Distributed Artificial Intelligence and Multi-Agent Systems awareness on the one hand, and a telecommunication and information technologies background on the other hand. In a context where these two actions think about joining their achievements to upgrade each other, this article reports several experiments, carried out during the five past years in the agent platforms field, mixing both the intelligence and the middleware aspects.

1 Context

1.1 The Story

As agent technologies are increasingly being involved in telecommunication-related applications, the need for open standards is becoming critical. During the past years, different scientific communities gave birth to different standardization actions, such as the Foundation for Intelligent Physical Agents (FIPA) and the Object Management Group's MASIF (Mobile Agent System Interoperability Facility). Although they finally share some major targets, the OMG and FIPA current results show their distinct origins, respectively with a Distributed Artificial Intelligence (DAI) and Multi-Agent Systems awareness on the one hand, and a telecommunication and information technologies background on the other hand.

Reporting our work in this context is a sort of historical counterfeiting, but is a striking way of explaining and justifying a hybrid approach that we have been carrying on for more than five years. In fact, we were inspired by several needs:

– a middleware to fit the distributed applications requirements (typically in the Computer Supported Cooperative Work and groupware fields);
– a higher level layer to support knowledge representation, distributed problem solving, cooperation…

As a result, we got involved in both the middleware and the DAI communities, and we started to mix their techniques.

1.2 Multi-Agent Platforms

Many multi-agent platforms offer modelling and implementation solutions to the distribution of intelligence. To a certain extent, one may consider multi-expert systems and Distributed Artificial Intelligence as parts of the origins of the multi-agent systems. Following this point of view, we can see through the introduction, in typical centralized Artificial Intelligence languages, of low-level communication features (e.g. TCP/IP sockets in Lisp or Prolog), or more elaborate communication structures (e.g. blackboards and *Linda Interactor* in [20]), the emergence of the first multi-agent platforms. Then, sophisticated models (*actors* [1]) and techniques (constraints, reflexivity [10]) have been merged to enhance multi-agent platforms.

But, as the enhancements are going on, the resulting diversity and heterogeneity makes it difficult for a standard to emerge, besides the AI classics, which industry is just beginning to appropriate to itself. Moreover, when it comes to distribution and communication, specific solutions are often applied, sometimes through simulation. But, the more these platforms integrate to standards, in real applications, the more striking is the proof of their accuracy. Then, it seems useful to find a way of defining a standards-based bridge between the real applications and multi-agent platforms, without preventing their evolutions, but enforcing reusability and interoperability.

1.3 Mobile Agents Platforms

Whereas Sect. 1.2 was dealing with multi-agent platforms springing from the DAI community, another type of agent platform is becoming more and more popular today. These platforms deal with mobile agents, and come with a strong telecommunication and information technology background. What can be called more generally “mobile computing” addresses various issues:

– access and locally process (so-called *remote programming*) big amounts of distributed information (e.g. distributed data mining);
– the active networks field, which aims at providing networks with autonomous, intelligent and dynamic setup features;
– management and access to distributed services and electronic marketplaces;
– load balancing and fault tolerance.

First introduced by General Magic’s Telescript for electronic commerce matters, mobile agents technology now benefits from the popularity of Java, which brings a precious transparency to heterogeneity. As a result, most of these platforms[1] consist in Java packages, offering an Agent class with communication and mobility features, and more or less sophisticated architectures and services. There also exists other languages-based platforms, such as Agent/TCL [8] (now called *D’agent*, as it will support other languages than just TCL in a near future), or hybrid platforms (e.g. the Tube [9], based on a Scheme interpreter written in Java).

The mobile agent community is very involved in security matters, as it is a specific and critical issue. Interoperability support between heterogeneous agent platforms,

[1] IBM’s *Aglets*, General Magic’s *Odyssey*, IKV++’s *Grasshopper*, ObjectSpace’s *Voyager*, The Open Group’s *MOA*, Mitsubishi’s *Concordia*, Fujitsu’s *Kafka*…

agents, and the need for intelligence are still needed, but nothing really exists at the present time. In this context, the FIPA and OMG standards are interested in playing a key-role to provide agents with interoperability and cooperation abilities.

1.4 Object-Oriented Communication Architectures

Operating systems, or layers between them and applications (*middleware*), increasingly integrate distribution and communication features. Standards from ODP (Open Distributed Processing) and OMG (Object Management Group) aim at designing a standard system environment which transparently cope with open systems issues, such as heterogeneity, interoperability, portability, distribution.

ODP brings a set of standards springing from independent organisations: ISO, IEC, ITU-T, AFNOR… The Reference Model (RM-ODP) [12] provides a conceptual framework for specifying an open distributed architecture. One key-point of RM-ODP is the use of five specification languages, bound to five points of view: enterprise, information, computation, engineering, technology. Strongly object-oriented, RM-ODP's concepts aim at improving interoperability and portability, while making distribution and heterogeneity transparent.

To a certain extent, the OMG's approach is more pragmatic, as it consists in building an open and non-proprietary object-oriented communication architecture by standardizing available technologies. Since its foundation by 11 organizations in 1989, the OMG gathered more than 500 members, among the main computing industry vendors. The CORBA standard [17] is being permanently refined and extended by the OMG, and several commercial implementations are available. The OMG's standards specify:

– an object-oriented model (*Core Object Model*);
– a reference architecture for objects management, based on the definition of the *Object Request Broker*;
– a *Common Object Request Broker Architecture* (CORBA);
– interfaces to generic objects and services: *Common Object Services* (e.g. naming service, event service), *Common Facilities*…

1.5 Our Approach

Object-oriented distributed systems come with communication architectures supporting generic functions related to distribution. A middleware such as CORBA gives agent platforms an actual distribution facility, while abstracting from heterogeneity and relying on basic communication primitives and common services. Moreover, the integration of intelligent agents to CORBA could lead to new intelligent common services, and bring advanced solutions to generic issues of CORBA applications.

Thus, an agent platform based on a standard communication layer:

– can concentrate on its specific jobs (information finding, knowledge representation, problem solving, cooperation…);
– is likely to benefit from, as well as to enrich, the applications and services of the environment it integrates to;

– builds the *bridge* we deal with in Sect. 1.2.

Today, some on-going work in the *agent community* and the OMG tends to encourage such a bridge. FIPA, for instance, underlines that "*agents need to be able to integrate with, and where appropriate use for themselves, existing and emerging computational infrastructure. Examples include: TCP/IP networking, CORBA, TINA-C, http and OLE.*" [4]. As far as the OMG is concerned, the MASIF [16] include a set of definitions and interface specifications related to agents systems interoperability. An agent is described as *"a computer program that acts autonomously on behalf of a person or organization. Most agents today are programmed in an interpreted language (for example, Tcl and Java) for portability. Each agent has its own thread of execution so that it can perform tasks on its own initiative"*. As we'll see in this article, these two last sentences have also something to do with our work (cf. "interpreted language" and "thread of execution").

2 The First Experiments

2.1 PUMA, the Forerunner

PUMA (Prolog Upgrade for Multi-Agent world) is our first experiment [5]. It results from the integration of a Prolog interpreter in a C++ object of the COOLv1 object-oriented communication layer [14]. The Prolog kernel and dialect have been extended to integrate COOL-based communication primitives: message and mailbox based communication, including group features, local synchronous communication, naming service, mobility, COOL object creation. Each agent has its own thread of activity, and can explicitly move and transparently resume its activity at the next programme goal.

PUMA have been enriched with many C++ development classes and Prolog modules allowing the incremental and modular composition of the agent's behaviour: activity step, cooperation structures and protocols, knowledge representation, specialised services… The system has been used to implement a distributed multi-agent application for meeting-rooms booking, accordingly to a multi-agent approach for office information systems [6]. Cooperation was based on communication groups with specific management and invocation protocols, dynamically created mobile agents for sub-contracted tasks, and constraints-based negotiation.

The dynamic, modular and declarative programming of agents, the activity-safe mobility, the constraints-based negotiation language, the group organization for cooperation, but also the quick prototyping opportunity, are the key-points brought to light by the PUMA experiment. Its career was interrupted by a big evolution of COOL, which evolved towards a CORBA conformance. However, other experiments were necessary, not only to follow this evolution, but also to try other combinations with other interpreted languages. Another practical aspect about the interpreter kernel is that the integration we did in PUMA needed much development time on the Prolog kernel source code, not only to add new primitives, but also to encapsulate it (break the interpreter loop and build a control function, extract and encapsulate the global variables to create several independent Prolog objects in a multi-threaded process).

2.2 The Successors

The immediate successors to PUMA followed the evolutions of COOL. This layer gained more and more distance from the underlying micro-kernel specificities (Chorus), and progressively became CORBA compliant. COOLv2 figures an intermediate state, mixing typical COOL features (persistence, mobility, communication groups, activity and concurrency management) with a CORBA-like architecture. COOL-ORB [3], the latest evolution, comes with a fully CORBA compliant platform. Beyond the middleware concerns, we also tried different kinds of integration of other logics-based interpreted languages, featuring extensions such as constraints.

Compared to PUMA, ROZACE (Remote execution server for OZ Agent in CORBA Environment) figures a complementary approach [15], by keeping the interpreter kernel apart from the agent and making it available as a script execution server. This architecture illustrates the case of a really big interpreter kernel, which offers many powerful features, but needs a large amount of computing resources. As a result, agents are *light*, but can perform advanced tasks by submitting scripts to the remote execution server[2], either synchronously or asynchronously. ROZACE was built on COOLv2 and Oz [19], an interpreted language that combines the logic, constraints, concurrent, high-order functional and object-oriented programming paradigms. Unlike PUMA, no primitive has been added to the language, and scripts are pure Oz. The Oz constraints features have been applied to a few parts of the room booking application formerly developed with PUMA.

CHOCOLAT (CHorus COol & Life Agents Tool) [11] has been developed in parallel with ROZACE, to explore a PUMA-like integration on COOLv2 (i.e. integrated interpreter, extended dialect), but relying on an extended logic programming language. Derived from Prolog, LIFE (Logic, Inheritance, Functions and Equations) offers functional programming, concurrency and constraints features, with types (cf. *sorted logics*) and inheritance [2]. The quality of LIFE's C language interface allowed a quick integration, with fewer source transformation than in the case of PUMA. Thanks to the underlying COOLv2 middleware, it has been possible to introduce a dynamic invocation mechanism, allowing the CHOCOLAT's interpreted kernel to invoke any method on any COOLv2 object, with no preliminary link edition. CHOCOLAT has been used to implement a representation and revision model for the beliefs of agents [18].

2.3 Conclusion

These experiments taught us some tips about middleware-based agent platforms. Not wanting to start from scratch, most of the work often consisted in adapting high-level interpreted languages from the source code, which is not convenient, neither efficient. In fact, we realized that very few such languages are designed as ready-to-integrate components.

[2] In fact, this approach can be mixed with PUMA's: a small interpreter kernel may be integrated in the agent to implement a minimal autonomous behaviour and knowledge representation.

As far as the agent primitives are concerned, those dealing with message sending, mailbox management and address publication (cf. naming service) appear to be essential and straightforward to implement. Group features, including broadcasting and functional sending, are of great interest for agents organization and cooperation matters. The dynamic creation of agents is also very useful, as it makes it possible for an agent to sub-contract a task to an autonomous extra activity.

Synchronous communication, which typically consists in having another agent executing a script in a synchronous way, generally causes concurrency problems at the embedded interpreter kernel level. As a result, this kernel has to be protected by mutual exclusion, which causes a deadlock in case of cyclic synchronous calls. So, this can be a convenient feature, but it must be used very carefully.

The main difficulty springs from mobility. PUMA was the only platform supporting a transparent agent state and activity mobility feature. It relied on COOLv1's migration feature and on Prolog's save/1 predicate, which creates a complete state file. But this mobility was limited to homogeneous environments.

3 Building a Multi-Agent Application on CORBA

3.1 What is "CIDRIA Générique"?

CIDRIA Générique is a generic workflow system, built on CORBA and intranet technologies, according to a multi-agent model [7]. *Workflow* systems consist in executing predefined or on-line generated scenarios, in order to trigger and synchronize a set of tasks in the information system, while achieving information gathering, circulation and tracing. CIDRIA Générique aims at managing procedures combining any kind of service, involving any type of resource: software, hardware, users… To cope with this real world heterogeneity, a homogeneous multi-agent virtual world is created by mapping each resource to a dedicated agent. Inside the multi-agent world, resources are represented by their *skills*, and they *cooperate* via *service requests* accordingly to these skills.

3.2 The Multi-Agent Platform

While making CIDRIA Générique, we wanted to produce as reusable as possible developments. We chose an on-the-shelf CORBA compliant middleware, and a Edinburgh-type Prolog interpreter (SWI-Prolog [21]). The high quality of its C interface made it possible for us to embed it in a C++ class without modifying the source code. This way, any interpreter or middleware update has a minor impact on our work. This class has been used to integrate a Prolog kernel in a CORBA server. As a result, we could implement every server operation in Prolog, through the C++ *mapping*[3]. This integration is minimal, as we didn't extend the Prolog dialect with communication

[3] A CORBA server interface is declared independently from the implementation language, in IDL (Interface Definition Language). Applying this abstract definition to a given programming language, to make server or client software, needs a specific *mapping*.

primitives. In current version, agents are concentrated on the same server, and share, for practical reasons (e.g. persistence, resources mobility transparency), the same Prolog kernel. Nevertheless, agents cooperate and communicate via message handling (no shared memory is used), and the architecture is ready to be extended with actual agents distribution, with several servers.

3.3 Conclusion

Built following modularity and reusability principles, CIDRIA Générique fits evolution. When developing and maintaining it, we really appreciated the comfort of Prolog programming, propitious to quick prototyping. As the agents are fully represented in Prolog, saving their states as a set of Prolog clauses fully describes the overall system. We used this opportunity for debugging, and also to introduce persistence features. Moreover, the Prolog declarative programming makes it possible to dynamically modify the operations implementation, without interrupting the server functioning. One just have to modify a Prolog file, and "reconsult" it from the server; the agents' behaviour is updated while their states are not affected (unless desired).

From a performance point of view, one could believe that the CORBA+Prolog association is too heavy, but it doesn't appeared to be the case. Although we didn't perform any load test, demonstrations made through the french network RENATER, between Paris (INRIA) and Caen (CNET), didn't reveal any particular response delay when invoking the server's operations; i.e. it looked like a local application[4].

To conclude, if our previous experiments tend to prove that object-oriented middlewares can bring an accurate solution to the communication needs of agent platforms, CIDRIA Générique legitimates the use of a CORBA-based multi-agent approach for the conception and implementation of distributed applications.

4 Recent Work

4.1 An "Interpreter" CORBA Interface

In order to extract the generic aspects of developing multi-agent platforms by integrating interpreted languages to CORBA, we have defined an "interpreter" CORBA interface, named `Interp`. It specifies two operations:

– `execute(in string script, out string result)` makes the server object execute `script` and give the `result` of the execution;
– `receive(in string message)` makes the server object add `message` to its mailbox. No message format is specified. It may consist of a script to run asynchronously, or any kind of data.

Messages, scripts and results are generically represented by strings. If a problem occurs during a call, these operations raise an exception. There are 4 exceptions:

[4] The client machine was a portable PC running Windows 95, and the server machine was a Sun UltraSparc 1 with 64 Mo of RAM, and wasn't dedicated to the demonstration (it run other server processes). Client software is written in Java, and server software in C++ and Prolog.

– FULL_MAILBOX indicates that the message couldn't be added to the mailbox;
– SCRIPT_ERROR means that the script couldn't be properly executed, because of a syntax or execution error;
– NOT_IMPLEMENTED is raised when the operation isn't available in the server object;
– _UNKNOWN informs that the operation couldn't be performed, but provides no diagnostic.

4.2 A Generic ORB Class

As we were to extend an interpreted dialect with communication primitives based on the middleware, accordingly to the Interp interface, we decided to generically (i.e. independently from the actual interpreted language) encapsulate these primitives in a C++ class, named Generic. We call it Generic because it isolates the interpreter kernel from the actual middleware. But this class is middleware dependent, and needs to be adapted from one middleware to another. Nevertheless, adaptation is clearly straightforward between CORBA compliant platforms[5].

Public methods of the Generic class include:
– initialization and destruction methods, which manage, among others, the link with the underlying ORB (e.g. access to the naming service, the group service…);
– two *pure virtual* methods[6], run() and call(), respectively representing an interactive interpreter loop, and a script execution method. These methods will have to be implemented when deriving the class for a given interpreted language;
– generic communication primitives, which are likely to extend the embedded interpreted language: naming service, message sending, mailbox management, communication groups, synchronous remote script execution, object (agent) creation;
– two methods implementing the Interp operations (see Sect. 4.1).

4.3 ORB-fying an Interpreter Component

Theory. Suppose we find a C++ class interpreter component, that we can derive, and that makes it possible to create several objects in the same multi-threaded process (typically one object and one thread per agent). This class also provides:
– a way to add primitives, by adding or overriding one or several methods;
– facilities to handle the entities of the embedded language (e.g. Prolog terms, or Lisp expressions);
– methods for external interpreter loop control and script execution, with error handling;

[5] Group communication primitives are specific to the ORB we chose (COOL-ORB), and should be implemented on other CORBA platforms via specific servers.
[6] A *pure virtual* method consists in a method declaration without implementation. Such a method can be called, but the class it belongs to is *abstract*, i.e. it cannot be instantiated. This class has to be derived so as to actually define the method, and thus be able to create objects.

– conversion methods between the string and internal representations of the language entities, which really helps write the message sending and script execution primitives.

Provided the fact that we find this ideal component, the integration task is easy to do through multiple inheritance, as shown by Fig. 1, and we get a CORBA-based agent platform. The port to another communication layer just involves the Generic class, and the port to another operating system is really minimal, while achieving interoperability.

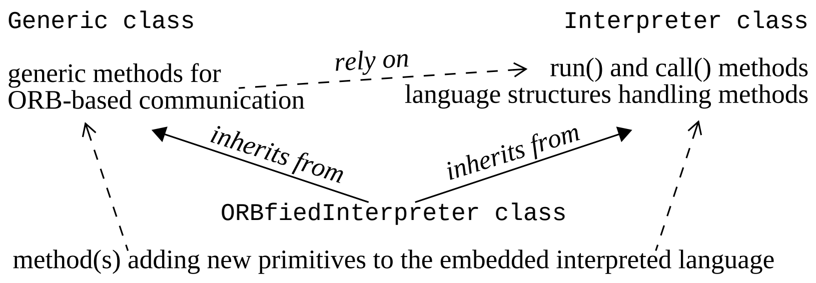
Generic class
Interpreter class
generic methods for ORB-based communication
rely on
run() and call() methods
language structures handling methods
inherits from
inherits from
ORBfiedInterpreter class
method(s) adding new primitives to the embedded interpreted language

Fig. 1. The ORBfiedInterpreter class defines new primitives based on the inherited ORB-based communication methods (Generic class) and interpreted language structures handling methods (Interpreter class). The Interpreter class must define two specific methods to complete the Generic class' implementation (run() and call()).

Practice. Confrontation to reality rapidly destroys this dream. According to our investigations, "AMZI! Prolog + Logic Server" is the only available interpreter component which matches our main requirements. Most of the others come with poor C interfaces which really turns the integration task into a nightmare. Obviously, global and static variables forbid multi-threading and multiple independent interpreter kernels in the same process. The worst C interfaces keep the main activity in the interpreter loop, which makes it very hard to implement a synchronous call primitive. Sometimes, the string conversion functions for the language entities aren't directly available, whereas they always exist in some form in any interpreter.

Finally, we decided to build by ourselves the libraries and their C++ encapsulation from existing interpreter C sources. We believe that on-the-shelf C++ components will be soon available, following AMZI! Prolog, Ilog products, or a few *extension languages*[7] way. Then, we'll just have to change our adapted kernels for these ready-to-integrate components.

[7] The extension language principle consists in integrating an interpreted language into an application in order to provide users with a powerful configuration and customization tool. Choosing a standard language prevents users from learning a new language for each application. The Emacs Lisp and the Guile extension language, from the Free Software Foundation, illustrate this idea. Elk (see Sect. 4.5) is another example of extension language which shows Scheme's popularity.

4.4 A Prolog Integration: BPorb

BinProlog is an Edinburgh-style Prolog implementation [20]. It is famous for its high performance, and the C source code is available at a low price academic licence. Its C++ encapsulation has been very tricky, for its C interface really didn't fit our needs. The result is interesting, however, as we succeeded in overcoming the fact that BinProlog keeps the main activity in its own interpreter loop. This was achieved thanks to a BinProlog primitive which creates a new engine to solve a goal, without destroying the current goal.

The resulting `BinProlog` class mainly supplies the `call()` and `run()` methods, and an `extension()` virtual method. This method is invoked by the Prolog kernel to implement the new_builtin/3 predicate. The new_builtin(+code, +in, ?out) goal is mapped to a call to `extension(code, in, out)`, where `code` figures a primitive number. A null return value from `extension()` means a goal failure.

Once this encapsulation is done, the final step consists in defining the `BPorb` class, which inherits from `Generic` and `BinProlog`. The `extension()` method is overriden, to map the communication primitives provided by `Generic` to the Prolog dialect, via the new_builtin/3 predicate.

4.5 A Scheme Integration: ElkOrb

Elk [13] is a free Scheme implementation, specially designed to be embedded (see *extension language*, note 7). Its C++ encapsulation resulted in two classes:

- the `ElkObject` class encapsulates the Scheme entities, with handling methods for every type;
- the `Elk` class encapsulates the language engine.

The `Elk` class mainly defines the `call()` and `run()` methods, required by the use of the `Generic` class. `Elk` also defines an `extension()` method, which is called when evaluating the (new-primitive arg1 arg2 ...) expression. The `extension()` method takes a vector of `ElkObject` objects (arg1, arg2...) as an argument, and returns an `ElkObject` object, which represents the evaluation result.

Once this encapsulation is done, the `ElkOrb` class is obtained by inheriting from `Generic` and `Elk`. `ElkOrb` overrides the `extension()` method to make the Generic communication primitives reachable from the Elk kernel.

4.6 Observations

The `ElkOrb` and `BPorb` classes have been used to create slave or interactive interpreter objects (slaves permanently look for incoming messages and execute them as scripts). At initialization time, these objects load a Scheme or Prolog file which contains their behaviours. They consist in CORBA server objects, registered in the naming service under chosen names which represent their addresses for messages and synchronous calls. Each communication group is also registered in the naming service, but under two names: one for message broadcasting, the other for *functional* (i.e. an arbitrary member of the group receives the message) synchronous or asynchronous calls.

Since communication primitives are based on the `Interp` CORBA interface, it is possible for different interpreted languages, integrated to CORBA via the `Generic` class, to communicate asynchronously or synchronously with each other. The main issue is to build a script for another language, and then to extract the result, via a string representation. It could be interesting to look at Xerox's ILU approach. The Inter-Language Unification platform formerly aimed at inter-language interoperability, but then took distribution into account, and finally offers a CORBA hook.

5 Conclusion

The experiments we have reported aim at proposing a bridge between distributed applications and multi-agent platforms, based on the adoption of a standard and non-proprietary middleware, CORBA. The reasons for such an approach are:

– CORBA platforms, services and distributed applications can be enhanced with intelligent features;
– Distributed Artificial Intelligence techniques may find a concrete investigation and application field;
– agent platforms may abstract themselves from the low-level distribution management primitives (basic communication, CORBA services), while adopting a standard communication architecture, propitious to their interoperability.

We have integrated typical Artificial Intelligence languages to CORBA platforms, either as remote execution servers, or by embedding an interpreter kernel in the objects, and extending its dialect with CORBA-based communication primitives. The last experiment resulted in extracting some generic features, thus isolating the interpreter part from the CORBA communicating part.

With the recent outcome of FIPA's and OMG's actions, and the will to combine the former's ACL (Agent Communication Language) with the latter's MASIF specifications, our experiments could be regarded as a way of combining a high-level approach with a middleware support.

Our further work will focus on agent mobility, OMG's and FIPA's work, CORBA services and facilities, and CIDRIA générique's evolution (integration of our last results, extensions with actual distribution and agent communication protocols). As a matter of fact, this application will be our privileged integration and demonstration platform for truly standards-based mobile intelligent agents.

Acknowledgements

The work reported in this paper was carried out in the CNET[8] lab in Caen, by Christophe Trompette, Emmanuel Hym, Eric Malville, Stéphane Piolain, and Bruno Dillenseger, under the responsibility of François Bourdon and Anne Lille, in collaboration with Patrice Enjalbert and Jurek Karczmarczuk from the University of Caen.

[8] "Centre National d'Etudes des Télécommunications", France Télécom research laboratories.

References

1. Gul Agha. *Actors, a Model of Concurrent Computation in Distributed Systems.* MIT Press (1988).
2. Hassan Aït-Kaci, Bruno Dumant, Richard Meyer, Andreas Podelski, Peter Van Roy. *The Wild LIFE Handbook (prepublication edition).* Digital Paris Research Laboratory (1994).
3. *Chorus / COOL-ORB Programmer's Guide.* Documentation CS/TR-96-2.2, Chorus Systems (1997).
4. Ian Dickinson. *nyrequirements.* FIPA document notes, Yorktown (September 28th 1996).
5. Bruno Dillenseger, François Bourdon. *Supporting Intelligent Agents in a Distributed Environment: a COOL-based Approach.* TOOLS 16, Prentice Hall (1995) 235–246.
6. Bruno Dillenseger. *Une approche multi-agents des systèmes de bureautique communicante.* Thèse de Doctorat, Université de Caen (1996).
7. Bruno Dillenseger, François Bourdon. *Modélisation de la coopération et de la synchronisation dans les systèmes d'information (une expérience de workflow basée sur les nouvelles technologies).* Calculateurs Parallèles Vol. 9, No 2, HERMES (1997) 183–207.
8. Robert Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. *Mobile agents: The next generation in distributed computing.* In Proceedings of the Second Aizu International Symposium on Parallel Algorithms/Architectures Synthesis (pAs '97), Fukushima, Japan. IEEE Computer Society Press (1997) 8–24.
9. Dave Halls. *Applying Mobile Code to Distributed Systems.* Doctoral Dissertation Computer Laboratory University of Cambridge (1997).
10. Salima Hassas. *GMAL - Un modèle d'acteurs réflexif pour la conception de systèmes d'intelligence artificielle distribuée.* Thèse de doctorat, Université Claude Bernard - Lyon I (1993).
11. Emmanuel Hym. *Intégration et mise en œuvre d'un langage interprété sur la plate-forme répartie à objets Chorus/COOLv2.* Rapport de DEA, Université de Caen (1995).
12. ISO, ITU. *Reference Model of Open Distributed Processing.* ISO standard 10746, ITU-T recommandations X.900 (1995).
13. Oliver Laumann, Carsten Bormann. *Elk: the Extension Language Kit.* Technische Universität Berlin / Universität Bremen, Germany (1996).
14. Roger Lea, Christian Jacquemot, Eric Pillevesse. *COOL: System Support for Distributed Programming.* Communications of the ACM, Vol. 36, No 9 (1993).
15. Eric Malville. *Etude d'une architecture agent sur la plate-forme Chorus/COOLv2.* Rapport de DEA, Université de Caen (1995).
16. *Mobile Agent System Interoperability Facilities Specification.* Joint submision: GMD Fokus & IBM Corp., supported by Crystaliz Inc., General Magic Inc., The Open Group. OMG TC document orbos/97-10-05 (1997).
17. Object Management Group. *The common Object Request Broker: Architecture and Specification (revision 2.0).* OMG (1995).
18. Stéphane Piolain. *Expérimentation de l'approche Programmation Logique Etendue pour la réalisation d'agents nomades sur Systèmes Répartis à Objets.* Rapport de stage de DEA, Université de Caen (1996).
19. Gert Smolka. *An Oz Primer.* DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany (1994).
20. Paul Tarau. *Bin Prolog 5.75 User Guide.* Département d'Informatique, Université de Moncton, Canada (1997).
21. Jan Wielemaker. *SWI-Prolog 2.5.* Dept. of Social Science Informatics (SWI), University of Amsterdam, Roeterstraat 15, 1018 WB Amsterdam, The Netherlands (1996).