ARTÍCULO

E-commerce using an agent oriented approach

Suna Alexandru, Lemaitre Christian, El Fallah Seghrouchni Amal

LIP6 - CNRS UMR 7606 University of Paris 6
8, Rue du Capitaine Scott
75014, Paris
{Alexandru.Suna, Amal.Elfallah}@lip6.fr

Abstract

Electronic commerce is a domain continually evolving. However, the potential of the commerce on the Internet is far from being completely exploited today. There are many elements that could be easily automated and endowed with intelligence, especially using agents. This paper presents the process of developing an e-commerce application modelling the coffee market in Veracruz, Mexico, using intelligent and mobile agents implemented in the CLAIM agent-oriented programming language and deployed on the SyMPA platform.

Keywords: Multi-agent systems, e-commerce, agent-oriented programming, mobility.

1. Motivations

Electronic commerce (e-commerce) is one of the most important aspects of the Internet to emerge nowadays. It allows people and companies to exchange services and goods with no limits of time or distance. At a simplistic glance, the main actors are the *buyers* who want to purchase goods or services, the *sellers* who offer them and the *producers* who create them. A producer is always a seller as well. The e-commerce contains all these actors and other additional important elements (*e.g.* secure payment) in the Internet context.
The e-commerce is a domain continually evolving during the last years. Statistics show that the business amount of the world-wide e-commerce evolved from 60 billions dollars in 2000 to 250 billions dollars in 2003 and was expected to reach in 2004 an amount of 430 billions dollars. Everything can be bought today on the Internet, but the most sold range of products are the computer products (hardware, software or accessories), books, documents, financial services, electronics, travel services, etc.
In the context of the electronic commerce, an important role is played by the Web Services [1]. They propose a credible support allowing applications to expose their functionalities using standard interfaces, favoring a service-oriented architecture composed of complex, distributed and cooperative heterogeneous systems.

However, even with Web Services, the e-commerce is still lacking the flexibility of human based traditional commerce. The reason for this is that no actor able to take knowledge based decisions is used in the electronic value chain. It is clear that the next generation of e-commerce would need to deal with the flexibility issue. The most promising technology to deal with this is the intelligent agent technology. The main elements of an intelligent agent useful in this context are the cognitive components (*e.g.* beliefs, goals, plans) and his ability to interact with other agents allowing a powerful autonomous reasoning. Another additional useful feature of agents for e-

[1] http://www.w3.org/2002/ws/

commerce applications is the mobility (*e.g.* agents can move through the Internet to collect information about products or prices), in order to support the geographic distribution of complex systems, their computation over the net and to reduce the bandwidth and the data traffic [22]. Therefore we claim that intelligent and mobile agents are extremely useful for designing e-commerce applications.

1.1. Why an agent-oriented programming approach

Unfortunately, while the main focus of the multi-agent systems (MAS) community has been on the development of informal and formal tools for MAS engineering, concepts or techniques, the design of declarative languages and tools which can effectively support MAS programming remained at an embryonic stage. Therefore, we argue that for a successful use of MAS paradigm, specific high-level programming languages are needed. With this objective in mind we have proposed an agent oriented programming (AOP) language called CLAIM [11] that helps the designer to reduce the gap between the design and the implementation phases (*i.e.* the designer should think and implement in the same paradigm, namely through agents), allows the representation of intelligent elements (*e.g.* knowledge, beliefs, goals, reasoning) and meets the requirements of mobile computation. In addition, the CLAIM agents can invoke methods implemented using other programming language (*e.g.* Java) and can invoke Web Services. We consider that there is a difference of abstraction between agents and objects and consequently between CLAIM and Java (even if the agents will do most of the computation using Java and the underlying platform is implemented in Java). The language's operational semantics [12] should allow the verification of the built multi-agent systems. The language is supported by a distributed platform called SyMPA [26] that offers all the necessary mechanisms for agents' management, communication, mobility, security and fault tolerance.

2. Context

In this section we present the developed e-commerce application, the steps and the choices made during the design phase. This application was developed in the context of a French-Mexican collaboration project [2]. The coffee producers in Veracruz are usually small, with a reduced financial power, but their number is important. In 2001 they created together with the government of the Veracruz state the *Consejo Regulador del Café Veracruz* (The Veracruz Coffee Regulator Council - CRCV) whose role is to verify the coffee producers' facilities that satisfy specific norms of the origin denomination of "Café-Veracruz" and to certificate the lots of coffee reaching the quality stated by the norm. The main objective of this strategy is to reach a new emerging specialty coffee market where the prices of quality coffee attain prices several time higher that the stock exchange prices for coffee. In 1999, the first e-auction for specialty coffee took place in Brazil. Since then, several coffee buyers and consumer associations decided to work on different e-commerce applications for specialty coffee markets. In this context we propose an agent-based application able to deal with the different types of transaction negotiations and covering the entire value chain of coffee. We claim that our model can easily be applied to other domains.

2.1. Identify the actors

A first step toward the final application is to identify all the actors involved in the considered scenario, their functionalities, interactions and all the delicate issues, independent of any implementation aspect.

First of all, we have the *Coffee Producer*. The certification of his coffee is done by the *CRCV* presented above. In the real environment, belonging to the CRCV is the *Coffee Verifier* that physically goes to the producer's facilities and verifies the quality of the production chain and of the final product and identifies the best coffee lots. In function of his appreciation, the CRCV emits certificates for the producer's coffee facilities lots and for the green coffee. Another actor is the *Market* where the producers "publish" their offers in order to be found by the interested *Buyers*. A buyer searches specific types of coffee on the market. It is actually the market that matches the coffee offers and the buyers' requests. When a buyer's demand corresponds to a producer's offer, a direct interaction between them begins. One of the main issues in commerce in general and specially

[2] Lafmi = French-Mexican Informatics Laboratory, http://lafmi.imag.fr

in e-commerce is the trustfulness of the players. The buyer must be convinced that the seller has the right certificates and the seller needs to be sure that the buyer corresponds to a known company and has the financial disposability. To do that, he asks for a letter of credit from the buyer's *Bank* (another actor). As soon as these requirements are satisfied, the producer must find a *Transport* company and an *Insurance* service for the transported goods. After the successful arrival of the goods at the destination, the buyer's bank is notified in order to send the money transfer to the producer's bank and to send the customs documentation to the buyer. This last phase is a delicate one and requires very precise and synchronized protocol.

This scenario gives a general view of the coffee market commerce in Veracruz. Additional elements can be added to this basic model. An interesting extension, that is naturally achieved using intelligent agents, is the producers' cooperation. Taken individually, a producer may not have enough financial power, but together with other producers he can form stronger groups for sharing a better technical infrastructure, covering a larger fragment of the market, answering to a greater number of client requests or sharing means of conveyance. Joining their efforts, they can reduce their costs and increase their benefits. Interesting cooperation protocols can be analyzed in practice and programmed using agents.
Another extension can be done using the advantages of the mobile agents. For instance, the buyers can send mobile agents to other buyers in order to locally analyze their data about different producers (*e.g.* the satisfaction level, the fairness of the quality-price ratio, etc.). Mobile agents can also be used by producers to gather data from other producers about the buyers' history.

2.2. Modelling the actors

Once all the actors of the application and their interactions are identified, the next step is to model them. For this stage we have adopted the methodology presented in [20]. Nevertheless, any other design methodology, appropriate for the developed application, can be used (*e.g.* AALAADIN [13]).

Even if we are using a multi-agent approach, it may not always be the best solution to represent all the actors as agents. In function of the intelligence, autonomy and mobility level of each actor, one must carefully decide if it should be represented as an agent (either mobile or stationary), as an independent application running on a site, as a Web page (dynamic or not) or as a Web Service (accessed through a special agent or directly). We do not propose a general rule for determining the chosen representation of an entity, but the designer's common sense and experience should make this choice adequate. We will consider below each actor and justify the chosen representation.

The **CRCV** answers to producers' requests for certification. He sends verifiers to their sites and he manages and interprets the gathered observations. He also answers the buyers' questions about the quality of the coffee of different producers. Therefore we represent CRCV as an intelligent stationary agent.

A **Coffee Verifier** agent is more difficult to represent as a software entity. In our application, a verifier is a mobile agent created by CRCV when a request for certification arrives. He migrates to the producers and give qualitative appreciations in function of the data received from them. This choice can be motivated by security reasons. The producers may not want to send important and private date over the network but they accept verifiers on their site. Being the only mobile agent in our application, he also illustrates how mobile agents can be used in the language.

The **Producers** ask for certification at CRCV, send their coffee offers to the market, interact with the buyers, with the bank, must hire a transport company and must insure the transported goods. They may also use cooperation protocols for creating coalitions of producers or may create mobile agents for verifying the buyers' history. So it is clear that a producer will be an agent with powerful intelligence and reasoning abilities.

A **Buyer** looks for certain types of coffee and sends requests to the market. After receiving matching offers of producers, he verifies the certification of the chosen one at CRCV, negotiates with him and receives the notification of the goods' arrival. We can also endow the buyers with capacity to create mobile agents that migrate in order to check the satisfaction level of other clients. Consequently, they will be represented as agents.

The **Market** is acting mainly as a mediator between the producers' offers and the buyers' requests. When a new offer arrives, the market will

find and send to the corresponding producer a list of interested buyers. On the other side, when a client sends a request, the market will send him the list of corresponding producers. The market can be represented as a Web Service or as an agent. Since we intend to endow him with intelligent capabilities, he is represented as an agent.

When an agreement is reached between a buyer and a producer for a coffee order, the producer must contact a **Transport** company for delivering the coffee and an **Insurance** company for insuring the transport. As the main role of these actors is to answer requests concerning the price of their services, we will represent them as Web Services.

Finally, the **Bank**'s services can be used by producers and buyers. The buyers' bank emits credit letters proving that a buyer is financially able to pay a coffee order. Also, after the goods' arrival, a money transfer takes place between the two sides' banks. We consider that the banks must have reasoning abilities (*e.g.* for treating the clients differently in function of their financial power) so we represent them as agents.
The figure 1 presents a high level architecture of our application, independent of the chosen programming language, framework or environment.

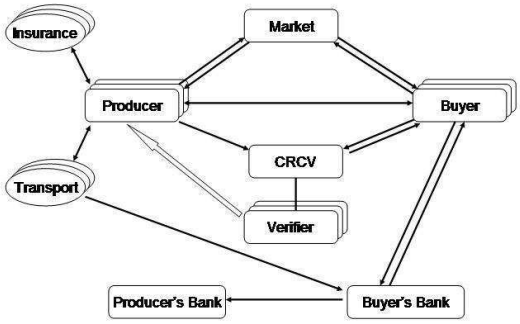

Figure 1. Coffee e-commerce architecture

3. Design

This section presents the design of the application using the CLAIM language.

3.1. CLAIM at a glance

In order to better understand the agents' definitions resumed in the next sub-section, we present here a brief description on the language.

CLAIM is a high-level declarative language which combines cognitive elements in order to endow agents with intelligent skills, and computational aspects such as mobility. The language's specifications were presented in [10] and its operational semantics in [12]. It can be used to define agents or classes of agents. A multi-agent system in CLAIM is a set of hierarchies of agents located on different computers in a network. A CLAIM agent is an autonomous, intelligent and mobile entity uniquely identified in the MAS, located at a node of a hierarchy. Thus, an agent has a parent and may have several sub-agents. In addition, he has intelligent components such as *knowledge*, *goals* and *capabilities* allowing a reactive or a goal-driven behavior.

An agent has a *knowledge* base containing pieces of information about the other agents' classes and capabilities or user-defined pieces of information about the environment they are acting in, represented as propositions containing a name and a list of arguments.

Agents' behavior is dictated by the received messages or by his *goals*. An agent can have several goals, represented as propositions. In order to achieve a goal, he tries to execute capabilities whose effects correspond to the current goals.

Therefore, the *capabilities* are the main elements of an agent, allowing to execute actions or to achieve goals. A capability has a name, a message of activation, a condition, the process to execute and eventual effects (post-conditions):

capability ::= capabilityName {
  message=(null | m);
  condition=(null | condition);
  do { p_i }
  effects=null; | { $e_1; e_2; ...; e_n$ }
}

To execute a capability, the agent must receive the activation message and verify the condition. A condition can be a Java function that returns a *boolean*, or a condition about the agent's knowledge, sub-agents or about his achieved effects.

Once a capability activated, the corresponding processes are executed (concurrently with the already running processes of the agent). So an agent (*e.g.* α) contains a set of concurrent running processes, $P_\alpha = p_i \mid p_j \mid ... \mid p_k$. One of these concurrent processes can be a (possibly empty) sequence of processes, a message transmission,

the creation of a new agent (instance of an already defined class), a mobility operation, a variable instantiation, a function defined in another programming language (in this version it is possible to invoke only Java methods), a Web Service invocation (a known Web Service; we are currently working on an extension of the platform where agents dynamically search and invoke Web Services), or the execution of a process for all the agent's elements from the knowledge base or his sub-agents that verify a certain criteria.

$$\begin{aligned} p_i ::= \quad & 0 \mid p_j.p_k \mid send(\alpha, m) \mid \\ & newAgent\ name : class(args) \mid \\ & in(\beta) \mid out(\beta) \mid move(\beta) \mid \\ & open(\beta) \mid acid \mid kill(\beta) \mid \\ & ?x = (value \mid Java(object.method)) \mid \\ & ?x = WebService(address, method) \mid \\ & Java(object.method(args)) \mid \\ & forAllKnowkedge(k)\{p_j\} \mid \\ & forAllAgents(\alpha_i)\{p_j\} \end{aligned}$$

The *send* primitive allows a message transmission to another agent, to all the agents belonging to a class (multicast) or to all the agents in the system (broadcast). There are pre-defined messages, used during the mobility protocols for asking and granting permissions, or used by agents to exchange information about their capabilities and knowledge bases. The users can also define their own messages, represented as propositions and utilized to activate capabilities.

The mobility operations are inspired from the ambient calculus [5]. Using *in*, an agent can enter another agent from the same level in the hierarchy (*i.e.* having the same parent) and using *out*, an agent can exit his parent. Unlike the ambient calculus, where there is no control, we added an asking/granting permission mechanism. The *move* mobility operation is a direct migration to another agent, without verifying a structure condition. Nevertheless, the operation is subject to permissions. Taking full advantage of the hierarchical representation of the agents, using the *open* primitive, an agent can open the boundaries of one of his sub-agents, thus inheriting, as in the ambient calculus, the later's running processes and sub-agents, but also his knowledge and capabilities. The *acid* primitive is similar to *open*, but it is an agent that decides to open his own boundaries, and as a consequence, his components are inherited by his parent. So in both cases, an agent dynamically gathers new capabilities and enriches his knowledge base.

These intelligent elements allow two types of reasoning for the CLAIM agents: forward reasoning (or reactive behavior) - activate capabilities when the corresponding messages arrive and the conditions are verified; backward reasoning (or goal-driven behavior) - execute capabilities in order to achieve goals.

CLAIM (in conjunction with the SyMPA platform resumed in the section 4.1) has already been used for several applications such as an information search on the Web [11], another e-commerce application [10], a network of digital libraries [20] and a resource sharing and load transfer application using mobile agents [21].

3.2. Agents' definitions in CLAIM

In this sub-section, starting from the agents' descriptions presented above, we will summarize the implementation process using CLAIM. For each agent we present only the most important cognitive aspects and capabilities. There are elements of the language that were not presented but that appear in the definitions (*e.g.* variables noted $?x$ can be used).

The **CRCV** agent's knowledge base contains pieces of information about the quality of the producers' coffee. An entry in the knowledge base is represented as: $Producer(name, type, quality)$ (*e.g.* $Producer(P_1, oro, 8)$). CRCV has capabilities for creating a *Verifier* when a producer asks for certification and can answer the buyers' questions about the quality of the producers' coffee.

```
Agent CRCV {
  ...
  capabilities {
   certifyProducer {
    message=askForCertification(?type);
    condition=null;
    do{newAgent Ver:Verificator().
     send(Ver,verify(sender,?type))}
    effects=null;
   }
   verifyProducerOK {
    message=verifyProd(?pr,?t,?q);
    condition=hasKnowledge(Producer(?pr,?t,?q));
    do{send(sender,prodOK(?pr,?t,?q))}
    effects=null;
   } ... }   ...
}
```

A **Verifier** agent is created by the CRCV agent each time a producer asks for certification. He is instantiated from a class whose definition contains a capability for migrating to producers, for

giving a qualitative appreciation to the coffee (calling a Java method) and for giving this information to CRCV.

```
AgentClass Verifier() {
  ...
  capabilities {
   verify {
    message=verify(?pr,?type);
    condition=null;
    do{move(?pr).
     ?q=Java(Verificator.verifyQuality(?type)).
      send(?producer,quality(?type,?q)).
      move(authority).
      send(authority,tell(Producer(?pr,?type,?q))).
      acid}
    effects=null;
   } }   ...
}
```

The **Producer** agents' definition is more complex and takes into account all the interactions with the CRCV agent, with the Market, with the potential buyers and with the transport and insurance companies. We present only two of his capabilities. First, after certifying his coffee, a producer sends his offer to the Market and receives a list of interested buyers (represented in the knowledge base as $Buyer(name, type, quality, quant)$, *e.g.* $Buyer(B_1, oro, 7, 500)$). He uses a Java function to select a buyer from the list. In the second capability presented, after signing an agreement with a buyer, a producer invoke Web Services of the known Transport companies (and similarly for insurance) for finding their prices and for choosing later the best price.

```
AgentClass Producer(?bank) {
  ...
  capabilities {
   chooseBuyer {
    message=knowInteresBuyers(?t,?qual,?price);
    condition=hasKnowledge(Buyer(?n,?t,?qual,?q));
    do{?b=Java(Prod.selectBuyer(this,?t,?qual)).
     send(?b,offer(?t,?qual,?price))}
    effects=null;
   }
   findTransport {
    message=findTransport(?quant);
    condition=null;
    do{forAllKnowledge(Transport(?n,?ad,?prM)) {
       ?pr=WebService(?ad,?prM)).
       send(this,tell(TransportPrice(?n,?pr))) }.
     ?bestTr=Java(Prod.selBestTrPrice(this)).
     send(?bestTr,transportOrder(?quant)) }
    effects=null;
   } ... }   ...
}
```

The **Buyer** agents have complementary but similar capabilities with the producers' ones. They send requests to the market, receive a list of producers, select one of them in function of different criterions (*e.g.* best price, other buyers' opinions), verify the certification of the chosen producer, negotiate with him and finally receive the coffee after the bank transfer is done.

The main functions of the **Market** agent is to match the producers' offers with the buyers' requests stored in his knowledge base (the pieces of information about the producers' offers have the format $Producer(name, type, quality, price)$, *e.g.* $Producer(P_1, oro, 8, 15)$ and those about the buyers' requests have been presented before, *e.g.* $Buyer(B_1, oro, 7, 500)$).

```
Agent Market {
  ...
  capabilities {
   offersFromProd {
    message=prodOffer(?t,?q,?pr);
    condition=null;
    do{send(this,tell(Producer(sender,?t,?q,?pr))).
     forAllKnowledges(Buyer(?b,?t,?q,?qt)) {
      send(sender,tell(Buyer(?b,?t,?q,?qt))) }.
     send(sender,knowInteresBuyers(?t,?q,?pr)) }
    effects=null;
   } ... }
  ...
}
```

The **Bank** has information about the clients and their accounts. It also emits credit letters for buyers. At the end, a money transfer is made from the buyer's account to the producer's bank.

4. Implementation

Once the agents and the classes of agents are defined, we deployed the application on the SyMPA platform. Its main features are presented in the next sub-section.

4.1. Overview of SyMPA

SyMPA [26] (System Multi-Platform of Agents) is a distributed platform that offers all the necessary mechanisms for a safe execution of a distributed MAS implemented using CLAIM. SyMPA is implemented using the Java language and its architecture is compliant with the specifications of

the MASIF [24] standard from the OMG.

SyMPA consists of a set of connected computers. On each computer there is an agent system that provides mechanisms for agents' management, authentication, authorization, resources access control and fault tolerance. There is also a Central System that has management functions. SyMPA's architecture is presented in the figure 2.

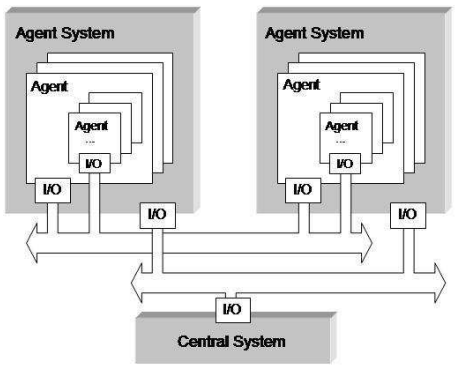

Figure 2. SyMPA's Architecture

An agent system is deployed on each computer connected to the platform. It provides (figure 3) a graphical interface for defining and creating agents and for visualizing their execution, a CLAIM compiler, mechanisms for agents's deployment, communication, migration and management, all of these in a secure and fault tolerant environment.

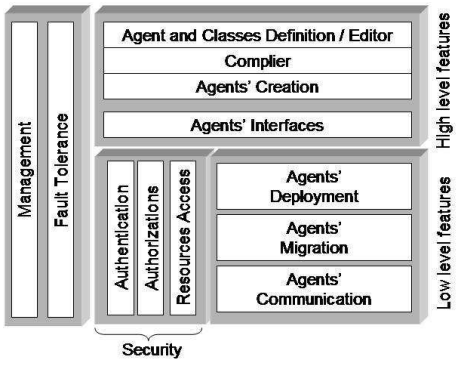

Figure 3. SyMPA's Features

An agent system offers visual tools and an editor where the agents' designer can define the agents or the classes of agents needed for the application, in a .adf (agents description) file. Nevertheless, any other text editor can be used. The definitions are then interpreted, the CLAIM syntax is verified and the agents (.agd) and classes (.cld) files are created in a format understandable by the Execution Engine. The interpreter was implemented using JavaCC (Java Compiler Compiler) [3]. The running agents are charged in memory and executed. There is a corresponding (optional) graphical interface for each running agent, where one can visualize agent's behavior, communication and migration.

The agent system is also in charge with the communication with other agent systems or with the central system and deals with the mobility, taking into account the security constraints. The communication and the mobility are implemented using Java above the TCP/IP protocol. As a result of our hierarchical representation of the agents we can distinguish two types of migration : local and remote migration. The local migration takes place inside a hierarchy, using the primitives that we have already presented. Thus, an agent moves with all his components in the local hierarchy. The remote migration is the migration between hierarchies (between different computers), using the *move* primitive. In CLAIM, the remote migration is strong at the CLAIM processes level, because the state of an agent is saved before the migration, it is transferred to the destination and the processes are resumed from their interruption point. At the Java level, we use the *Java Virtual Machine* migration facilities (*e.g.* Class Loader, object serialization), so there is a weak migration. A Java method begun before the migration will be reinvoked after the successful arrival at the destination.

The mobile agents are programs running in a distributed and insecure environment (*e.g.* the Internet) where there are possible different attacks against the host agent systems and against an agent during the migration or during his execution. Several solutions exist to these attacks [16], but they are outside the scope of this paper. In SyMPA, for the agent systems' protection, we are using agents' authentication, the control of the access to the system's resources in accordance with a set of permissions given to agents in function of their authority, and audit techniques. For the agents' protection, we are using encryption and also fault tolerance mechanisms. The reader can find in [26] a detailed description of these security aspects in SyMPA.

[3] JavaCC on-line at https://javacc.dev.java.net/

4.2. Implementation and tests

The classes of agents presented in the section 3.2. were edited and compiled using the SyMPA interface. The Transport and Insurance Web Services were implemented using the J2EE framework. We started a couple of different Transport and Insurance Web Services, with different prices for the offered services. Next, using SyMPA and the already defined agents and classes, we started the CRCV agent, the Market and a Bank (for both producers and buyers). After the "stable" stationary agents and services were deployed, we executed several producers on different computers, they asked for certification at CRCV; created by CRCV, Verifiers migrated to each new producer, verified the quality of the coffee and sent this information to CRCV. The producers sent their offers to the market, but there were no matching buyers at that moment. So we instantiated next several buyers whose requests matched some of the producers' offers. Successful protocols and transactions took place between several buyers and producers, the goods arrived at the destination and the money was transferred to the producers' bank. The figure 5 presents the graphical interface of the running agents.

The complexity of this application and the obtained results prove the strength and the expressiveness of the CLAIM language and the solidity of our platform. Using the same concepts for the design and for the implementation and using the agent-specific actions and mechanisms offered by CLAIM, the implementation duration was substantially reduced.

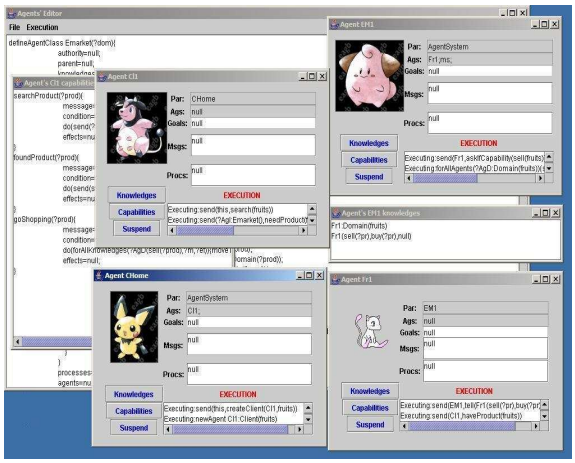

Figure 5. Application's graphical interface

5. Related Work

As we have seen, this paper presents a complex e-commerce application implemented using the agent-oriented programming language CLAIM supported by the SyMPA platform. This work is situated at the intersection of several domains and should be positioned with respect to all of them.

First, CLAIM combines elements from the AOP languages, such as *AGENT-0* [25], *AgentSpeak* [27], *3APL* [19] (and their extensions), that are suitable for representing intelligent agents endowed with cognitive skills but lack clear mechanisms to deal with mobility and distribution, and elements from the concurrent languages, such as the *ambient calculus* [5], the *safe ambients* [23] or *Klaim* [8], that enable to represent concurrent processes, that can communicate and migrate in a distributed environment, but are not suitable to represent intelligent aspects of agents. *Telescript* [28] and *April* (with its object oriented extension *April++*) [7] focus on the mobility aspect of agents; nevertheless these languages have neither the reasoning capabilities of the AOP languages nor the formal solidity of the concurrent languages.

Several platforms supporting mobile agents exist nowadays, such as *Aglets* [1], *D'Agents* [15] or *Grasshopper* [2]. All of them offer mechanisms for the agents' creation, communication, migration and management, guaranteeing in the same time a high level of security. However, the supported agents are implemented using mainly Java, so the agents are actually mobile objects. *JADE* [3] is a Java-based platform, FIPA-compliant, that offers support for Java agents execution, communication and (recently) migration. Another platform that has common elements with our approach is *Jason* [4], an interpreter implemented in Java for agents designed in AgentSpeak.
A presentation of the AOP languages and of the concurrent languages can be found in [10] and of the mobile agents platforms in [26].

From the application point of view, agents have been used to develop many e-commerce environments, because of their advantages stressed in the first section. However, to our knowledge, there is no e-commerce application implemented in an agent-oriented language. Usually Web approaches (dynamic Web pages, ASP, JSP, Java Script, frameworks such as J2EE or .NET, etc), Java or

other object-oriented languages are used for implementing agent-oriented applications.
In [18] is presented a survey of the e-commerce systems using agents and of the roles agents play as mediators in different stages of a customer to buyer scenario. There are systems such as *Jango* [9] or *Tete-a-Tete* [17] whose agents are useful in the *Product Brokering* stage (*i.e.* information retrieval to help to determine what to buy). *Jango*, *Kashbah* [6] or *Tete-a-Tete* use agents during the *Merchant Brokering* phase (*i.e.* determine who to buy from). And during the *Negotiation phase*, *Kashbah* or *Tete-a-Tete* provide agents that establish the terms of the transactions. ABROSE [14] is a large scale project trying to create an agent-based brokerage service to provide user support for request, navigation, registration and propagation of information.
Many other applications and systems for e-commerce using agents that we did not cite here exist. They try to exploit the agents' advantages but do not use an agent-oriented approach as in our case. The developer uses agent concepts for the design phase and then must translate these concepts into an object-oriented framework. This gap is not present in our approach.

6. Conclusion

This paper presents the results of a project we developed in a French-Mexican collaboration context (LAFMI laboratory). This project consists of a generic framework for the design of distributed applications based on MAS technology and mobile computing.

The proposed framework is constituted of an agent oriented language for MAS design (*i.e.* CLAIM), a distributed platform to support such MAS (*i.e.* SyMPA) and a real-world international e-commerce scenario (coffee Market of Veracruz) that takes benefit from MAS concepts and have been implemented using CLAIM and SyMPA.

Face to the increasing interest in the e-commerce domain and the several developments that have been proposed to deal with, one of our project motivation was to fully exploit this domain's potential by introducing MAS paradigm. We argue here that MAS technology is useful to bring new interesting concepts to design intelligent applications/systems that require sophisticated interactions, intelligent behaviors and autonomy. These features enable to introduce more flexibility in the designed system, to sustain both individual and collective intelligent behaviors, and consequently to improve the global performance of the whole system. Hence, the approach presented in this paper takes benefit of the main advantages of the MAS technology. It goes on to use the power of mobile computation thanks to our language CLAIM and the SyMPA platform.

The complexity of the international e-commerce application and the first results of our prototype prove the strength and the expressiveness of CLAIM and the robustness of the platform.

Our future work aims to introduce several improvements such as the use of mobile agents to gather information about the producers' and buyers' history, or the coalitions' formation by the producers using cooperation protocols.

Referencias

[1] Aglets: http://www.trl.ibm.co.jp/aglets.

[2] Grasshopper: http://www.grasshopper.de.

[3] F. Bellifemine, A. Poggi, and G. Rimassa. JADE - a FIPA-compliant agent framework. In *Proceedings of PAAM'99*, pages 97–108, London, U.K., 1999.

[4] R.H. Bordini and J.F. Hubner. Jason, a Java-based AgentSpeak interpreter used with Saci for multi-agent distribution over the net. On-line at http://jason.sourceforge.net.

[5] L. Cardelli and A.D. Gordon. Mobile ambients. *Foundations of Software Science and Computational Structures, LNAI*, 1378:140–155, 1998.

[6] A. Chavez, D. Dreilinger, R. Guttman, and P. Maes. A real-life experiment in creating an agent marketplace. In *Proceedings of PAAM '97*, London, UK, 1997.

[7] K.L. Clark, N. Skarmeas, and F. McCabe. Agents as clonable objects with knowledge base state. In *Proceedings of ICMAS96*. AAAI Press, 1996.

[8] R. de Nicola, G.L. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, pages 315–330, 1998.

[9] R. Doorenbos, O. Etzioni, and D. Weld. A scalable comparison-shopping agent for the world wide web. In *Proceedings of the First Interational Conference on Autonomous Agents*, 1997.

[10] A. El Fallah Seghrouchni and A. Suna. Claim: A computational language for autonomous, intelligent and mobile agents. *Proceedings of ProMAS'03 Workshop of AAMAS, LNAI*, 3067:90–110.

[11] A. El Fallah Seghrouchni and A. Suna. An unified framework for programming autonomous, intelligent and mobile agents. *Proceedings of CEEMAS'03, LNAI*, 2691:353–362, 2003.

[12] A. El Fallah Seghrouchni and A. Suna. Programming mobile intelligent agents: an operational semantics. In *Proceedings of IAT'04*, Beijing, China, 2004. IEEE Press.

[13] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS98)*, pages 128–135, Paris, Frace, 1998.

[14] M.-P. Gleizes and P. Glize. ABROSE: Multi agent systems for adaptive brokerage. In *Proceedings of the AIOS 2002 workshop of CAiSE*, 2002.

[15] R.S. Gray, D.Kotz, G.Cybenko, and D. Rus. D'agents: Security in a multiple-language, mobile-agent system. *Mobile Agents and Security, LNCS*, 1419:154–187, 1998.

[16] M.S. Greenberg, J.C. Buyington, and D.G. Harper. Mobile agents and security. *IEEE Comunications Magazine*, 36(7):76–85, 1998.

[17] R.H. Guttman and P. Maes. Agent-mediated integrative negotiation for retail electronic commerce. In *Proceedings of of AMET'98*, 1998.

[18] R.H. Guttman, A. Moukas, and P. Maes. Agent-mediated electronic commerce: A survey. *Knowledge Engineering Review*, 13(2):147–159, 1998.

[19] K.V. Hindriks, F.S. deBoer, W. van der Hoek, and J.J.Ch. Meyer. Agent programming in 3APL. *Intelligent Agents and Multi-Agent Systems*, 2:357–401, 1999.

[20] G. Klein, A. Suna, and A. El Fallah Seghrouchni. A methodology for building mobile multi-agent systems. In *Proceedings of the 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing SYNACS04*, Timisoara, Romania, 2004.

[21] G. Klein, A. Suna, and A. El Fallah Seghrouchni. Resource sharing and load balancing based on agent mobility. In *Proceedings of ICEIS'04*, 2004.

[22] D.B. Langeand and M. Oshima. Seven good reasons for mobile agents. *Communication of the ACM*, 42:88–89, 1999.

[23] F. Levi and D. Sangiori. Controlling interference in ambients. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 352–364. 2000.

[24] D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, and J. White. MASIF, the OMG mobile agent system interoperability facility. In *Proceedings of Mobile Agents*, pages 50–67. 1998.

[25] Y. Shoham. Agent oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[26] A. Suna and A. El Fallah Seghrouchni. A mobile agents platform: architecture, mobility and security elements. In *Proceedings of ProMAS'04 Workshop of AAMAS*, New-York, 2004.

[27] D. Weerasooriya, A. Rao, and K. Ramamohanarao. Design of a concurrent agent-oriented language. *Intelligent Agents: Theories, Architectures, and Languages, LNAI*, 890:386–402, 1995.

[28] J. White. Mobile agents. In *Software Agents*, MIT Press, 1997.