SADIO Electronic Journal of Informatics and Operations Research

http://www.dc.uba.ar/sadio/ejs

vol. 3, no. 1, pp. 1-12 (2000)

A Java Framework for Multi-agent Systems

Henri Avancini[1,2] Analía Amandi[1]

[1]Universidad Nacional del Centro, Facultad de Ciencias Exactas
ISISTAN Research Institute
Campus Universitario Paraje Arroyo Seco, (7000) - Tandil - Buenos Aires, Argentina
E-mail: {henri, amandi}@exa.unicen.edu.ar

[2] Universidad Nacional del Comahue, Departamento de Informática y Estadística
Buenos Aires 1400, (8300) - Neuquén, Argentina
E-mail: havancin@uncoma.edu.ar

Abstract

The complexity of distributed systems (multi-agent systems in particular) involves the design and coordination of many entities (called agents). This activity is not trivial and there is not a standard way to do this. So, it is desirable to have tools for reusing previous designed components and relations between them. Frameworks allow design and code to be reused. Intelligent agents cover a big amount of application domains but we can detect an important set of common characteristics that can be abstracted for reusing. This paper presents a framework for multi-agent systems (FraMaS) composed by these common characteristics. FraMaS is used to design agents that have capabilities for perceiving environment, communicating, and deliberating about what to do next.

Keywords: Framework, Multi-agent System, Software Agent, Java, Internet Programming

1 Introduction

The agent revolution represents an extraordinary opportunity for business, because they must be capable of managing and organizing information, recognizing personal tastes, and making increasingly important decisions on behalf of their owners.

Multi-agent Systems (MAS) are a particular case of distributed systems. The development of this kind of systems is not trivial. To avoid the task of designing each new system from scratch, we need tools to help in the MAS construction.

A *multi-agent system* is an instance of distributed systems composed by agents [O´Ha96]. An *agent* is an autonomous entity that has behavior conduced by objectives. Each agent can perceive and act over an environment. Agents are built to solve problems in several application domains. Examples include electronic commerce, intelligent tutoring systems, Internet searchers and several kinds of personal assistants. Surprisely, these so different agents have common characteristics that could be reused.

A *framework* is a reuse object-oriented technique, which is used to develop applications framed in a given context, and which allows the design and code to be reused. Each framework is composed by a set of classes and relations among them.

A *Framework for agent systems* allows developers to built multi-agent systems by using both composition and inheritance of a set of restricted but still general enough agent parts. There is a considerable research effort in the development of agent design techniques. Some related works allow researchers to identify patterns in the agent development. However, agent technology is still a beginning field.

With the aim to solve the MAS development from scratch we propose a framework for multi-agent systems, called FraMaS. Next section briefly describes what frameworks and multi-agent systems are. Our framework for multi-agent systems is introduced in section 3. Then we present Agents (section 4) and Multi-agent systems (section 5) under FraMaS. In section 6, experiences of instantiations are described. Section 7 is about related work. Lastly, conclusions and future work are presented.

2 Background

A *framework* is an object-oriented technique for reusing design and code [Lew95]. It is specified by a set of classes and schematic algorithms that show how their instances can interact. All this information is defined in a high level of abstraction. Such level of abstraction is usually achieved by developing experiences in the framework's application domain. A framework will become a skeleton of an application domain [Joh97].

Design reuse is achieved since a framework divides system into interacting objects in a schematic way. Also, a framework allows code reuse because it defines classes, inheritance and dynamic relationships in a programming language.

Each framework allows the construction of applications for a particular domain. This is possible since frameworks specify generic classes that define the domain structure, allowing the application customization. There are four kinds of methods that support such generalization: (a) abstract methods, (b) template methods, (c) hook methods and (d) base methods.

Abstract methods define only an interface, without any implementation. These methods must be re-implemented in each subclass. *Template methods* define the common control flow among objects of the system. *Hook methods* define a default implementation that developers can re-implement in subclasses, i.e. a communication protocol. *Base methods* are the complete and generic behavior of the systems belonging to the framework domain. It has not been re-implemented in any subclass.

Agents who try to achieve particular goal states compose a multi-agent system [Jen98]. We define an *agent* as a computer program that has an internal knowledge that guide his behavior with autonomy. An agent has sensors to perceive changes in his environment. It acts sometimes as a reactive program when it performs actions conduced by stimulus-response and, sometimes, it deliberates to decide what to do next.

For example, a personal assistant that helps users to administrate meetings, a meeting-scheduling agent, can learn the user's preferences by observing the user-application interactions and then it can make suggestions related to user tasks by using those learned user preferences. For example, when the user inputs a

new appointment specifying the subject and participants, the agent search and suggest the free time slot in the calendar. Moreover, it could negotiate with the participants the meeting time and place. In general, we say the user delegates some tasks to the agent, and the agent helps the user in the application use (suggesting, searching alternatives, etc.).

Agents live inside an environment or world. Two or more agents could reach a conflicting state (for example when two robots collision in a common path). To solve this kind of problems, each agent could have communication capacities (for negotiating in our case example). Moreover, the agent autonomy allows it to control his internal state and his actions without direct human intervention.

When we combine MAS domain with frameworks, we are looking for a reusable structure that allows agent developers to instantiate different agent systems.

3 Framework for Multi-agent Systems (FraMaS)

3.1 Purpose of FraMaS

FraMaS stands for Framework for Multi-agent Systems. FraMaS is used to design agent-based systems, avoiding do it from scratch. Each agent has the capability to perceive his environment; communicate with other agents, users and applications; and deliberate about what action to do next. This agent helps user to reduce work and information overload.

Examples of multi-agent systems are: meeting scheduling agents, which can learn the user's preferences from experience and suggest appointments according to this knowledge; interface agents, which recommend music, movies or other forms of entertainment; agents for electronic news filtering; intelligent tutors; etc.

3.2 General design

Environments compose the proposed abstraction to model multi-agent systems (MAS). Each environment holds agents. These agents interact with each other without the MAS intervention, intending to satisfy their objectives.

Agents have the capability to move from one environment to another. To do this, the framework provides agent primitives and environment primitives [Ava99]. The first ones are like *start* (standard execution) and *stop* (secure state, where the code and data are saved). The second ones are like *requestToTransfer* (the local environment notifies the remote environment of his desire to transfer an agent) and *beginTransfer* (the remote environment marks the beginning of the transfer).

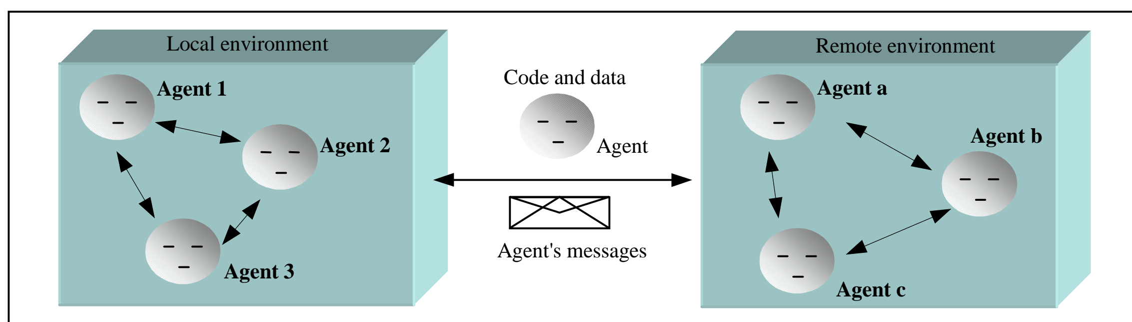

Figure 1 – Multi-agent system model

The agent design, using FraMaS, decorates the "basic behavior" (like the data base consulting or the *move* action in a robot) with "advanced behavior" wrappers (like autonomously search according to the agent

knowledge of the user or planning strategy to arrive to the target point). We will briefly describe the most important classes in the agent class structure that allow perception, deliberation, communication and next action selection under FraMaS.

Agent class is responsible for the common characteristic of all agents. The Agent class uses the *reflection* capabilities provided by Java. In this way dynamic messaging can be used, i.e. it allows an agent to inspect itself to find out what operations it supports, and to dynamically invoke them. Other classes that inherit from Agent are *BasicAgentActions* and *AdvancedBehavior*:

- The first is responsible for the basic-actions information, and has the methods for adding and deleting actions. As an agent can change the operations that it can perform, this information is necessary, for example, to create a plan. An action is called *basic* when it involves neither planning nor negotiation or deliberation; in general, any kind of artificial intelligence technique.
- The second class, AdvancedBehavior, is responsible for the wrappers over the basic actions. The deliberation mechanisms are Case Based Reasoning [Lea96], used in our examples to learn the user's preferences; Planning [Wel98], used to decide what action to do next; etc. The communication (and negotiation) is performed by classes that inherit from AdvancedBehavior and *decorate* the deliberation wrapper.

Users, agents, MAS environment, and applications compose the agent context. Using sensors, each agent can perceive his context. *AgentContext* and *AgentListener* FraMaS classes are responsible for the required functionality.

Nowadays a large amount of agents exists. However, we have detected an important set of common characteristics that can be abstracted for reusing. As a result, to design a particular MAS using FraMaS we need the systems' requirements for each kind of agent and for each environment. We divide the development into three stages: (i) identification of the agent's basic functionality, (ii) the required advanced behavior and (iii) the services that the agent will need.

We present in section 4 how an agent designed under FraMaS can perceive his environment, deliberate, communicate, and decide what action to do next. We introduce in section 5 the multi-agent systems services and, in section 6, we present examples of multi-agent systems designed using FraMaS.

4 Agents under FraMaS

FraMaS uses the pattern Decorator [Gam95], which allows adding functionality to objects dynamically. This pattern has been incorporated to dynamically *decorate* the basic functionality of each agent with the advanced behavior. In this way new layers are incorporated over the previous ones, each layer improving the agent with new behavior.

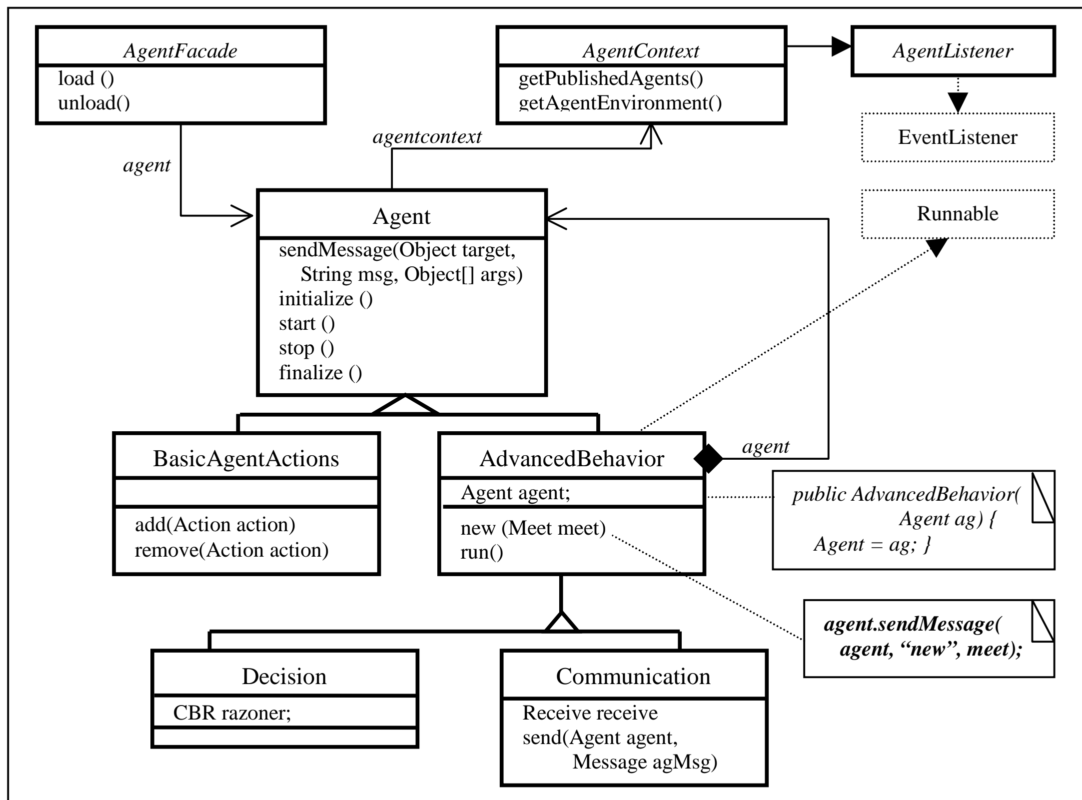

Figure 2 – Agent's design class schema

The general class structure of an agent is represented in figure 2. We show an agent with two wrappers: one for learning the user's preferences using Case Based Reasoning; an other for communication. AgentContext class and AgentListener abstract class are used to manage the agent context.

Initializing, Executing, Stopped and Finalized are the agent states. These are reached by the Agent messages: *initialize* (agent's structure initialization), *start* (the agent work), *stop* (saves internal data, intermediate results, etc. to reach a safe state), and *finalize* (kills the agent). Initialize and finalize take place only once in the agent's life. See figure 3.

The states' change depends on each agent's movement from host to host. Moreover, concurrent access could occur in a multi-thread environment. In this way, the framework implements a common interface using the Facade pattern [Gam95]. The agent access is simplified using this interface because it makes the subsystem more reusable and easier to customize.

AgentFacade class can provide a simple default view of the *agent.* The agent's clients (the local environment for example) communicate with the subsystem by sending requests to the AgentFacade class, which forwards them to the appropriate subsystem object(s). It insolates the clients from subsystem components. Moreover it reduces the number of objects that the client deals with, and makes the subsystem easier to use.

The AgentFacade class is the agent host-facing interface to an agent. It provides thread safe access to an agent and has the functionality to load an agent to the system. To do this, it uses a class loader called AgentClassLoader, which inherits from ClassLoader (the default java class loader that uses the JVM -Java Virtual Machine- to execute bytecodes).

FraMaS needs a special agent-loading behavior to start an agent in the environment. It works according to the following list:

1. Create a new agent class loader
2. Create a new agent facade, using the agent class loader
3. Initialize the agent facade

4. Load the agent into the agent facade
5. Initialize and/or start the agent

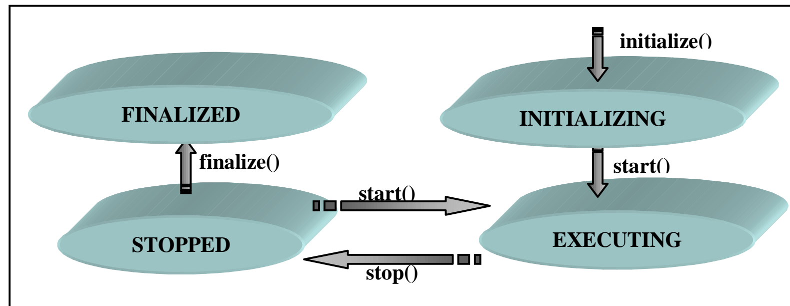

Figure 3 – Agent's state diagram

4.1 Perception

One agent characteristic is the ability to perceive the environment, composed by other agents, users and applications. Now we will explain how an agent designed using FraMaS can support this characteristic.

The Agent class implements the AgentListener interface to observe other agents. This interface contains a common defined protocol that can be used by the agent. Following this design way, each agent has a list of observers; if an event occurs then the agent will notify the interested agents in the list (the notification will be carried out only when the appropriate conditions are satisfied, such as security issues).

AgentListener interface defines five abstract methods: the addAgentListener(Agent agent) and the removeAgentListener(Agent agent) which add and delete an agent from the observers list; the notifyAgentListeners() and the notifyAgentListeners(AgentEvent agEvt) which notify other agents that an event has occurred; and the eventFired(AgentEvent e) which notifies to agent's interface (if it exists) that an event has occurred.

We said that an agent environment is composed for applications too. FraMaS allows agents to perceive Java applications (applet and standalone applications). However, it is mandatory that the application implements the AgentListener interface described above. It allows the agent to be registered in the application. A registered agent will be notified of interesting changes in the application.

For the last, to perceive users in the agent's environment, we propose observe the user-application interaction and the user-agent interaction.

4.2 Deliberation

Many wrappers add responsibilities to each agent. Next, we will show a learning process using Case Based Reasoning (CBR) [Lea96]. The Deliberation class (figure 2) uses CBR to learn the user's preferences. The reasoner maintains a case base, and each case is an interaction between the user and the system. When a new (probably incomplete or conflicting) case is presented (situation) the reasoner searches for the *most similar* ones and generates a proposal (solution) to the user. An example of this presented in Section 6.1 Meeting-scheduling Agent.

Using hook methods and abstract methods the CBR class is defined (figure 4) and the reasoner interface is specified. This class is responsible for the case base (caso), i.e. the user's preferences. Moreover, this class could be re-implemented using this structure to decide what action to do next (based on the past training examples saved in a case base)

Figure 4 shows the CBR class that maintains a case base (AgentCase is the type of each case) that is generated using the *save* method.

The interface is defined in the hook method *searchSolution* (see pseudo-code) to look for the most similar cases in the case base according to the new situation. It uses a matching algorithm to rank over the case base.

At last, the abstract method *composeSolution* composes the solution using the ranking vector generated by the *searchSolution* method. The solution is presented to the user. If it is not accepted, the cycle starts again (re-ranking - compose solution - present solution).

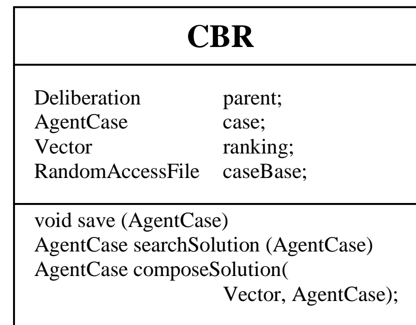

Figure 4 – CBR class

```
public AgentCase searchSolution (AgentCase situation) {
     case              = new AgentCase();
     case.id           = new Date().toGMIString();
     case.situation    = situation;
     Matching matching = new Matching (CaseBase, situation);
     Ranking = matching.algMatching1();
     if (ranking.size() > 0)
       case.solution = composeSolution(ranking, situation);
     else
       case.solution = situation //when similar cases are not found
}
```

A new case is generated by the *searchSolution* method. Then, an instance of a matching class is initialized. The matching class can be formed by many strategies; each one has the functionality to match cases (in the case base) with the new situation. So, we obtain a solution made up by the most similar cases. If the case base is small or if the new case is completely different (for example, a new user's preference) the ranking vector will be empty and the solution will be the original situation.

4.3 Communication

As we said, each wrapper covers the basic agent actions improving the agent, i.e. adding new behavior dynamically. The responsibility for the communication between agents was implemented in the FraMaS agent design. The framework gives a set of common primitives that allow agents to send and receive messages. The communication is performed without the MAS intervention, avoiding a bottleneck of messages to/from the MAS [Jen98].

The Java mechanism of RMI (Remote Method Invocation) is used by FraMaS to send/receive messages. So, it is necessary to re-define the AgentMessage class to design the communication between agents. To send/receive messages the framework provides a predefined set of hook methods.

For each new agent, an internal FraMaS structure and relations are initialized, and the communication is allowed. In fact, a hook method receiveMsg(AgentMessage) -common to all agents- is called. The functionality to receive messages is implemented in this method. Another hook method is initialized analogously to send messages.

There are other FraMaS classes that support the communication: Receive class, a Communication class component that initializes a thread to accept the incoming messages; SendMsg class, a new instance is created to send messages.

A natural extension to the communication is KQML (Knowledge Query and Manipulation Language) [Fin95]. Shared knowledge is mandatory in any intelligent interaction, i.e. the mutual understanding of the knowledge and the communication of the knowledge. Some authors, like Genesereth, emphasize that an entity is a software agent if and only if it communicates correctly in an agent communication language.

After all, it is hard to think about an agent community that exists and grows only in isolation; it would go against our perception of a decentralized and interconnected MAS.

4.4 Threads

Agents have the capacity to decide what action to do next in each time stamp. The proposed model uses a thread of the AdvancedBehavior class. The Java Runnable interface is implemented in this class, so that a template method *run* is defined.

A decision strategy can be implemented, for example, invoicing an abstract method that returns an action (or set of actions) that must be executed for the agent in the next time stamp. The decision algorithms are abstracted in an abstract class that can be sub-classified, so the protocols are respected and the concrete algorithms are defined. An alternative decision strategy can be implemented as a planning algorithm.

According to our adopted definition, an agent has an independence of the user, i.e. it operates autonomously. A common agent control structure has been defined in FraMaS (template method run). As we see in the next pseudo-code example, a protocol to select an action and to execute an action is established.

```
public abstract class AdvancedBehavior
                        extends Agent implements Runnable {
   public Agent agent;
   public AdvancedBehavior(Agent ag) {
      agent = ag;
      if (actionThread == null) {
      actionThread = new Thread(this, "Action");
      actionThread.start();  }
  }
  public void run() {
      Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
      /* Next action selection */
      action = select_action();
      do_action(action);
      // other agent actions
  }
}
```

5 Multi-agent System Services

The Multi-agent System was defined as a set of *environments*, each one providing functionality that allows agent inter-operation. Any agent is registered in one environment (according to his necessities) and probably interacts with other agents.

In the MAS (environment) the most important class is MASServices. It is responsible for the common environment that allows agents to share information, to communicate, to negotiate, to move from one host to another. Agents have a unique identity. Each agent can communicate with other agents in his (local) environment, or with agents in a remote environment, directly. The communication is archived using the Remote Method Invocation (RMI) provided by Java (See Communication in section 4.3).

Agents can translate from host to host In that way, the environment must provide a mechanism to get to know the physical address of a given agent knowing the agent identity. We are talking about a kind of name server, but this depends on the MAS policies, such as the mobility.

The *createNewAgent* message in MASServices class allows to create an agent from the resource file (bytecode file) and the data file (when an agent has stopped his execution and has migrated from one host to another).

We have already described the basic MAS functionality. Future classes that inherit from MASServices can improve each environment adding specific characteristics, for example, to allow agents to offer and request products in an electronic commerce system.

Lastly, inherited classes from MASServices could consider security aspects like authentication, privacy, safety rules to guide the social interaction (especially important for mobile agents), etc. As these issues depend on each MAS, they could be designed when we use the framework to construct that system.

6 Examples of Instantiations

6.1 Meeting-scheduling Agent

There is a Multi-agent System composed by meeting-scheduling agents. Each one belongs to a specific user and has the functionality to learn the user's preferences and to negotiate the appointments with other users' agents. Moreover, the basic functionality is to add/remove appointments to/from the agenda and to consult the saved appointment.

An appointment is composed by the date, time, place, duration, subject and a (probably not empty) list of invitees which is used to communicate (negotiate) with other agents (users' invitees agents) to send the meeting request.

The user's preferences (like date/time to attend subject, relations between people and subjects, etc.) are learned from examples. These examples are obtained automatically from the user-agenda interaction. They are saved in a case base and retrieved using Case Based Reasoning. When the user adds a new (incomplete) appointment, i.e. the user specifies only the subject and invitees, or a conflicting appointment, the CBR looks for the most similar cases in the case base and -adapting this- presents a solution to the user.

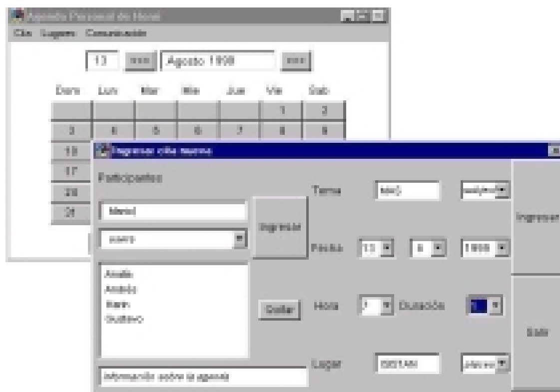

Figure 5 - Meeting-scheduling interface

In figure 5 we show the meeting-scheduling interface to add an appointment to the agenda. The agent is "watching" the user-application interaction, and presents alternatives when exist conflict or the required information is incomplete.

If, for example, the user inputs a new appointment with A and B to discuss about environment, the meeting scheduling agent propose a date, time, duration and place for the meeting using his knowledge of the user.

Then, if the meeting is accepted, the meeting scheduling agent sends (negotiate) the invitations to A´s agent and B´s agent.

6.2 FORKLIFT Agent

This is another example of a multi-agent system, called Forks. It is an instance of a general problem of a set of autonomous agents interacting to achieve the system's objective. It is composed by forklifts, which have sensors to perceive their environment. There are five different instances of these MAS; each one depends on the forklift decision strategy to select what action to do next and how to resolve conflicts.

The aim of implementing this system is to confront the performance results with the ones showed in the bibliography. In order to do that, we have driven the experiment as similarly as possible to Fischer [Fis94]. Homogeneous agents communities are used in different tests with 4, 8 and 12 forklift agents in a loading dock of size 15 x 20 squares with six shelves and one truck. A schema is showed in figure 6.

Moreover, the Forks systems allow to test decision strategies (in the selection of what action to do next), planning (for example: use the previous experience to carry out a box from the truck to the shelves), and negotiation (when two or more agents are in a path conflict).

The different forklifts are:

- RWK, which selects next action at random.
- BCR, which uses a decision function based on priorities, and solves conflicts at random.
- BCH, which uses a decision function based on priorities, and solves conflicts heuristically.
- LCH, which selects next action using a local plan, and solves conflicts heuristically.
- LCC, which selects next action using a local plan, and solves conflicts cooperatively.

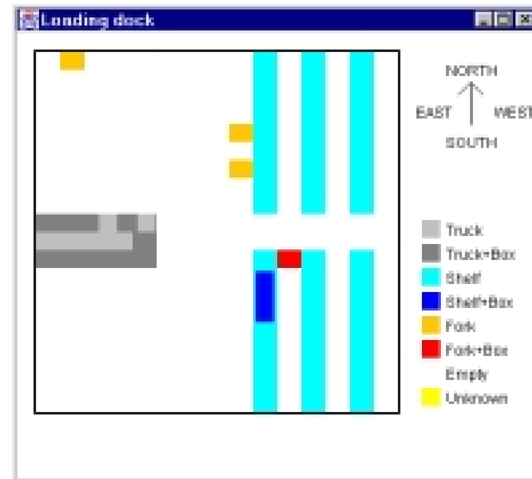

Figure 6 - Loading dock

7 Related Work

There is a considerable research effort in the development of agent design techniques, but only a few of these approaches will be discussed here. In particular, we will discuss some of them related to agent development methodology, patterns and frameworks for multi-agent systems.

The bottom-up approach to the design of agents provides a first-step in the agent development. [Kau94] describes the process of an agent that schedules a visitor to the laboratory. First he identifies feasible tasks for agents, then he implements and tests, and for last identifies crucial factors (like reliability, security) defining a platform for agent development. Our approach to framework construction is similar: we have planned a real agent for meeting scheduling, and then we have added some advanced behavior (like deliberation, using CBR) and identified common components and relations formalizing in a framework. To continue, we have used the framework to develop the Forks multi-agent system and to design an agent for electronic commerce.

An approach adopted in [Ari98] and [Ken98] provides a feasible way to develop agent-based systems using pattern for agents. A pattern catalog to agents will be useful, but not enough to agent system development because patterns are just ideas for specific design problems. To improve this, we can use patterns in integration frameworks.

[Cha98] describes a framework that provides a methodology for developing speech-act based on multi-agent systems. This framework allows representing and developing cooperation knowledge and protocols in multi-agent systems. We share a common goal, avoiding the multi-agent system development from scratch, providing tools that help the agent design. However, [Cha98] is oriented to interoperation, communication and cooperation mechanisms, while we are oriented to the set of common components and how to integrate them in an agent environment.

[Li97] presents another framework for the control of multiple robots' arms with multi-agent technology focused on cooperative behavior. The agents are grouped into a team of a specific task. The proposed architecture reflects the results of the developed system and helps in the future construction of this kind of systems.

Another interesting approach is the layered structure of InteRRaP architecture combined with the BDI-style architecture described in [Fis96]. It provides architecture both to explain agent behavior and to support system development from a pragmatic perspective. The paper center is on individual agents rather than on cooperation and interaction. However, architectures are more abstract than frameworks that provide design and code reuse, allowing the fast system instantiation. Moreover, architecture can have more than one framework that implements it.

8 Conclusions and Future Work

Agent-based systems have been developed to the requirements of fault-tolerant distributed systems, open systems, personalized and customized user interfaces that are proactive in assisting the user, just to mention a few. Our work aims to achieve multi-agent system development without doing it from scratch. We propose a framework (FraMaS) that merges common design characteristics of the multi-agent systems.

We divide the MAS into environments, with agents composing each one. Each agent has capabilities to perceive his environment, communicate and deliberate about what action to do next. The functionally to interact between agents is provided by the MAS services.

The proposed idea is to think of the agent as a set of basic action functionality that are decorated (or wrapped) with a set of advanced behavior, like reasoning capabilities, negotiation, etc. In order to allow the agent's interaction and mobility (from one environment to another) the MAS provides the primitives that the agents will use.

We are working on FraMaS, specifically on: (i) the framework instantiation and (ii) some mental states issues related to negotiation and learning techniques (decisions trees) in order to improve the negotiation reducing it.

References

[Ari98] Aridor, Y. and Lange, D. Agent Design Patterns: Element of Agent Application Design. Proceedings of the Second International Conference on Autonomous Agents (1998).

[Ava99] Avancini, H. Experiencias en reuso de sistemas multi-agente en Java. In: Workshop de Investigadores en Ciencias de la Computación. San Juan, Argentina (1999).

[Cha98] Chauham, D. and Baker, A. JAFMAS: A Multiagent Application Development System. Proceedings of the Second International Conference on Autonomous Agents (1998).

[Fin95] Finin, T. et al. KQML as an agent communication language. Software Agent. Ed. Jeff Bradshaw, MIT Press, Cambridge (1995).

[Fis94] Fischer, K.; Müller, J.; Pischel, M. Unifying Control in a Layered Agent Architecture. Technical report TM-94-05 from the DFKI GmbH (1994).

[Fis96] Fischer, K.; Müller, J.; Pischel, M. A pragmatic BDI Architecture. In: Wooldridge , M.; Müller, J.; Tambe, M. (Eds). Intelligent Agents II. p.203-218.(LNAI, v.1037). Berlin: Springer-Verlang (1996).

[Gam95] Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995).

[Jen98] Jennings, N. et al. A Roadmap of Agent Research and Development. Autonomous Agents and Multi-Agent Systems, 1, pp.7-38 (1998).

[Joh97] Johnson, R. Components, Frameworks, Patterns. In: Symposium on Software Reusability Proceedings, pp.10-17 (1997).

[Kau94] Kautz, H. et al. Bottom-Up Design of Software Agents. Communications of the ACM, Volume 37, Number 7, pp.143-146 (July 1994).

[Ken98] Kendall, E. et al. Patterns of Intelligent and Mobile Agents. Proceedings of the Second International Conference on Autonomous Agents (1998).

[Lea96] Leake, David. Case-Based Reasoning: Experiences, Lessons, & Future Directions. Edited by David B. Leake. American Association for Artificial Intelligence (1996).

[Lew95] Lewis, T. et al. Object Oriented Application Frameworks. Prentice Hall, Manning (1995).

[Li97] Li, G.; Weller, J. and Hopgood, A. Shifting Matrix Management - A Framework for Multi-Agent Cooperation. In: Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology (1997).

[O´Ha96] O´Hare, G. and Jennings, N. (editors). Foundations of Distributed Artificial Intelligence. Willey-Interscience Publication (1996).

[Wel98] Weld, David. Recent Advances in AI Planning. Technical Report UW-CSE-98-10-01. Department of Computer Science & Engineering. University of Washington. (1998).