From Active Objects to Autonomous Agents

Zahia Guessoum, Jean-Pierre Briot

From Active Objects to Autonomous Agents

Zahia Guessoum and Jean-Pierre Briot

Thème Objets et Agents pour Systèmes d'Information et de Simulation(OASIS)

Laboratoire d'Informatique de Paris 6 (LIP6), Case 169, 4 place Jussieu, 75252 PARIS Cedex 05

e-mail : {Zahia.Guessoum, ean-Pierre.Briot}@lip6.fr

http://www-poleia.lip6.fr/ {guessoum,briot}

Abstract

This paper studies how to extend the concept of active object into a structure of agent. It first discusses the requirements for autonomous agents that are not covered by simple active objects. We propose then the extension of the single behavior of an active object into a set of behaviors with a meta-behavior scheduling their activities. To make a concrete proposal based on these ideas we describe how we extended a platform of active objects, named *Actalk*, into a generic multi-agent platform, named *DIMA*. We discuss how this extension has been implemented. We finally report on the application of *DIMA* to model economic agent evolution.

1 Introduction

Object-oriented concurrent programming (OOCP) is the most appropriate and promising technology to implement agents. The concept of active object may be considered as the basic structure for building agents. Further-more, the combination of the agent concept and the object paradigm leads to the notion of "agent-oriented programming" [23] [24], which is the context of the present paper. The uniformity of communication mechanism of objects provides facilities to implement communicating agents and the concept of encapsulation of objects enables combination of various granularities of agents. Further-more, the inheritance mechanism enables specialization and factorization of knowledge.

An agent is easily implemented by an active object. Although the concept of an active object provides some degree of autonomy, in that it does not rely on some external resources to be activated, its behavior still remains procedural in reaction to message requests. To achieve autonomy, several researchers have proposed to add to this active object some function to control the messages reception and processing by considering its internal state (see for example [4] [18]). Therefore the agent activity is not limited to receiving and sending messages. More generally, we consider that, to be autonomous, agents must be able to perform a number of functions or activities without external intervention, over extended time periods.

The basic questions to build a bridge between: 1) the implementation and modelisation requirements of DAI systems [6] [7] [12] and 2) the implementation and modelisation facilities and techniques provided by OOCP [13] are:

- what is the most appropriate and the most generic structure to define the main features of an autonomous agent?
- how to accommodate the highly-structured OOCP representation machinery into some relatively generic DAI implementation structure [13]?

This paper is an attempt to give answers to these two questions. It deals with 1) the modelisation requirements of DAI systems by providing a generic and modular agent architecture and 2) the extension of the implementation and modelisation facilities of OOCP. We would like to make OOCP more relevant to DAI problems by incorporating into OOCP specific representations and computing structures driven by DAI needs. The paper presents an implementation structure that supports DAI applications and enables to model the activities of an autonomous agent in a multi-agent world. Agents may range from simple processing entities to complex "intelligent "entities [8]. More concretely, in this paper we will describe how to extend a model of active object (named the *Actalk* platform) [3] [4], towards a generic and modular model of agent (named the *DIMA* platform).

The paper is structured as follows. Section 2 presents briefly active objects, the platform *Actalk*, some limitations of these objects to represent agents and some requirements to build a generic agent structure. Section 3 describes the proposed generic agent architecture. Section 4 describes this architecture implementation with Actalk. Section 5 reports on the application of the proposed autonomous agents architecture to modelisation of economic agents evolution. Finally, we discuss the advantages of our architecture to design and implement multi-agent systems and we describe some future work.

2 Active Objects

The concept of active object (also named actor) has been introduced by C. Hewitt [17] to describe a set of entities which cooperate and communicate by message passing. This concept brings the benefits of object orientation (modularity) to distributed environments and it provides object-oriented languages with some of the characteristics of open systems [1]. Based on these characteristics several languages have been proposed (see for instance in [28]).

2.1 Actalk

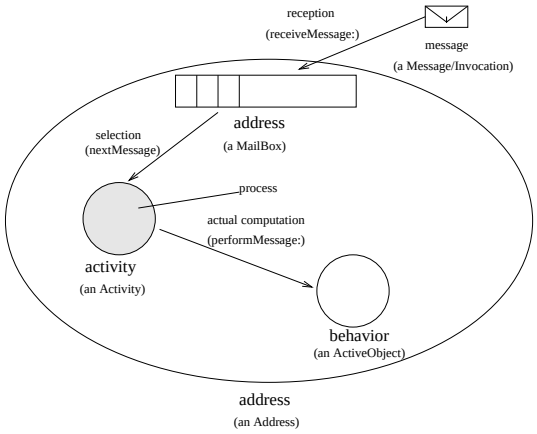

Figure 1: Components of an *Actalk* active object.

Actalk is a platform implementing various kinds of active object models [28] into a single programming environment based on *Smalltalk*. Asynchronism, a basic principle of active-object languages, is implemented by enqueuing the received messages into a mail box, thus dissociating message reception from its interpretation. In *Actalk*, an active object is composed of three components (see Figure 1):

- an instance of class `Address` represents the mail box of the active object. It defines the way messages will be received and enqueued for later interpretation;
- an instance of class `Activity` represents the internal activity of the active object. It provides autonomy to the actor. It owns a *Smalltalk* process which continuously removes messages from the mail box and launches their interpretation by the behavior component;
- an instance of class `ActiveObject` represents the behavior of the active object, i.e. the way individual messages will be interpreted.

To build an active object with *Actalk*, one has to describe its behavior as a standard *Smalltalk* object. The active object using that behavior is created by sending the message `active` to the behavior.

```
active
 "Creates an active object by activating its behavior"
 ^self activity: self activityClass address: self addressClass
```

Methods `activityClass` and `addressClass` represent the default component classes for creating the activity and address components.

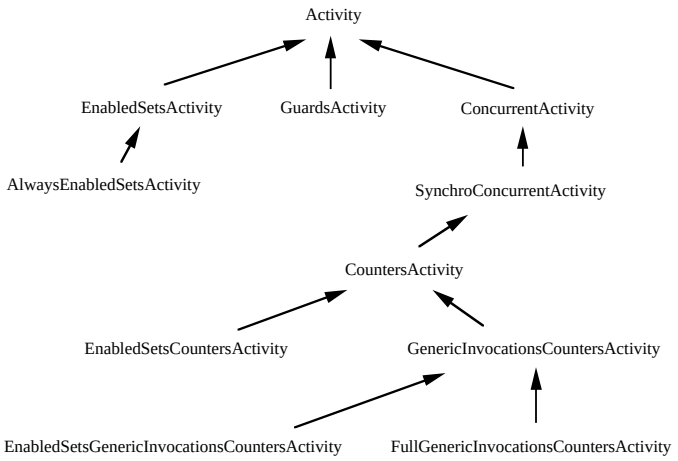

Figure 2: Hierarchy of the activity/synchronization classes.

Customizing *Actalk* means defining subclasses of the three component classes: `Address`, `Activity` and `ActiveObject`. This allows to define specific models of active objects, e.g. various communication protocols as subclasses of class `Address`, various models of activity and synchronization as subclasses of class `Activity` (see Figure 2).

2.2 Limitations of Active Object to Build a Generic Agent Structure

OOCP provides us with powerful foundations for modeling and implementing agents. However, these powerful and useful foundations do not really provide a generic agent structure. Active objects are monolithic and they have a procedural behavior. In spite of their communicating subjects appearance, active objects do not reason about their behavior, about their relations and on their interactions with other active objects. Also if an active object does not receive messages from other objects, it stays inactive. To cope with these limitations, several researchers have enriched the concept of active object to define a generic agent structure:

- T. Maruichi proposed concepts to add, to active objects in order to provide autonomous agents [18]. He introduced a message interpreter to realize autonomy and the notion of environment to form groups of agents.

- T. Bouron added a speech-act theory to improve the communication between active objects [2].

- Y. Shoham introduced mental states to have an interaction-based behavior [23] [24].

- etc.

These are very interesting proposals, however they do not offer a generic agent structure which matches the whole spectrum of DAI requirements [13]. To make this more clear we will quickly summarize some needed agent properties [27] which are not provided by active objects:

- An agent is an **autonomous** entity. It operates without direct intervention of humans or other agents. It must have some kind of control over its actions and internal state [5] [6] [8]. The autonomy notion needs the resolution of several problems: 1) how to "understand" and "adapt itself" to "reality"? 2) how to elaborate action plans? and 3) how to deal with perturbations in the perception and actions on the environment? To be autonomous each agent needs a self-control mechanism to manage its behavior in accordance with its internal state and its external universe one.

- An agent is a **pro-active** entity. It does not simply act in response to the received messages from the other agents. For example, it interacts with its environment and reasons to determine the most appropriate action in the current context.

- An agent is a **reactive** entity. It scans asynchronously and concurrently its environment to perceive data and responds in real time to these changes.

- An agent is a **sociable** entity. It interacts with other agents by exchanging messages. It handles the other agents requests and generates adequate responses to these requests.

In the following section, we present a generic architecture of agent that addresses such requirements.

3 A Generic Agent Architecture

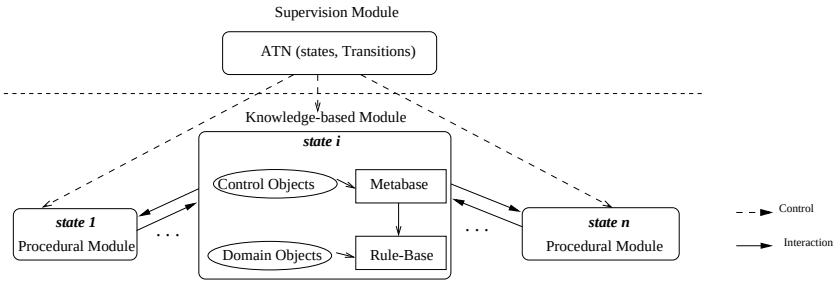

Figure 3: The proposed architecture.

In attempts to define a generic architecture which owns the main properties of an agent, we propose the extension of the single behavior of an active object into a set of behaviors. This

architecture (see Figure 3) relies on a first layer made up of interactive modules that can describe procedural behaviors and/or knowledge-based behaviors. These modules represent the different concurrent agent behaviors such as communicating, reasoning and acting. They provide the agent with the properties described in Section 2.2. For example, the communication module manages the interaction between the agent and some other agents of the system. Therefore, it makes the agent sociable and the perception modules makes its reactive.

A higher level supervision module represents the agent meta-behavior. It allows the agent to elaborate the different behaviors activation plan.

The first layer modules interact directly without the intervention of the supervision module. However, the activation of these modules is supervised by the latter.

3.1 The Agent Behaviors

To model complex systems, agents need to combine cognitive abilities to reason about complex situations, and reactive abilities to meet deadlines. So, an agent may have two kinds of behaviors: stimulus-response and deliberative behaviors. These two kinds will be called respectively procedural and knowledge-based behaviors.

In this section, we give three examples of modules: the perception module (procedural behavior), the reasoning module (knowledge-based behavior) and the communication module (which can be either procedural or knowledge-based by using, e.g. speech acts).

- **The perception module** manages the interactions between the agent and its environment. It monitors sensors and translates sensed data to define a set of believes which can represent a model of the agent universe (the other agents and the environment). The obtained universe model (believes) is used by the deliberation module to reason about the other agents.

- **The deliberation module** represents beliefs, intentions and knowledge of the agent. It is responsible for generating adequate responses to the messages transmitted by the communication module, or to the changes detected by the perception module. To do this it relies on two kinds of abilities: know-how (operative abilities) and/or knowledge (cognitive abilities). The first one is represented by the standard behavior of the associated objects (example: calculation of the new budget), and the second one is embodied in a production system [15] [21](example: the decision process which chooses a strategy in a given context).

- **The communication module** defines the mail box of the agent. It defines the way the messages are received and enqueued for later interpretation. It can reuse the low-level communication mechanism of the active-object model [4]. In this case, it is reactive. However, agents often use high-level communication mechanism such as those based on speech acts [25]. So, the communication module can be deliberative.

These three modules seem sufficient to several application domains (see, for example, section 5). Moreover, the use of a modular approach facilitates the integration of new modules such as a learning module.

3.2 The Agent Meta-Behavior

To be autonomous, each agent needs a self-control mechanism to elaborate activities in accordance to its internal state and its external universe one. The first level modules define

the agent internal state, the other agents states and the environment one. So, an agent state relies on the first layer modules states.

In the agent architecture that we propose, the self-control is managed by the supervision module. This module is a generic scheduler of the agent activities which are defined in the first-layer modules. It allows the agent to dynamically adapt its behaviors to its universe changes. Usually, the modules which describe the interactions between the agent and its universe, are reactive. So, the separation between deliberative behaviors (such as reasoning) and reactive behaviors (such as perception), as well as the use of concurrent processes to represent these behaviors provide reactivity to the defined agents.

The supervision module relies on two fundamental notions: states and transitions which naturally build up an Augmented Transition Network (ATN [26]).

States qualify the context as perceived by some first-layer modules. Each module has its own state. The combination of these states defines the global agent state. Changes in the context are reflected as transitions between states. Each transition links an input state with an output state. The various signals received by the agent's modules (urgent message reception, perception of new data, ...) represent the conditions of transition. The actions of transition are the activation of the first layer behaviors (activate reasoning, terminate reasoning, read mail box, scan the environment, ...).

At each transition, the ATN-based supervision module evaluates the conditions of transitions (representing new events) to determine the most appropriate behaviors. When these conditions are verified, the transition actions are executed and the agent state is modified.

Indeed, at each transition, the agent can adapt its deliberative behaviors to the reactive ones, so to say, it adapts its behaviors to its universe changes (An example of ATN is given in section 5).

4 Implementation

To implement the proposed agent architecture, the *Actalk* kernel (see Section 2.1) has been reused.

We have enriched this kernel to implement the proposed generic agent architecture. In this architecture, an agent is defined by a meta-behavior and a set of behaviors. In the agent-oriented programming [23] [24], the cycle of knowledge inference is merged with the cycle of message acceptance. In our implementation, the cycle of agent activity is merged with the agent behaviors activation described by the ATN.

Moreover, we dissociate the declarative part of the agent meta-behavior (ATN states and transitions) from its procedural part (ATN interpreter). This declarative representation makes the implementation more flexible.

Therefore, the agent architecture is composed of:

- an object (instance of `AgentActivity` sub-class of `Activity`) which implements the ATN interpreter,
- an object (instance of `Meta-Behavior-ATN` sub-class of `ActiveObject`) which implements the ATN declarative part,
- a set of objects describing the agent behaviors. They implement the different modules of the first layer (perception, reasoning, communication/action, ...). These objects are defined as instance variables in the class `Meta-Behavior-ATN`.

4.1 Agent Activity

In *Actalk*, an object `Activity` manages and transmits the received messages which are interpreted by another object describing the active object behavior.

The instance method `body` used by `createProcess` creates a process to take out continuously the messages present in the active-object mail box.

```
!Activity methodsFor: 'activity setting'!
body
 [true] WhileTrue: [self acceptNextMessage]

createProcess
 [self body] newProcess
```

In our architecture, the agent activity is described by an ATN which schedules the different behaviors (perception, reasoning, communication, etc.).

```
!AgentActivity methodsFor: 'activity setting'!
 body
self atnInterpreter

!AgentActivity methodsFor: 'atn'!
atnInterpreter
"The agent engine"
|atn state|
atn := self metaBehavior atn.
state := atn initialState.
state = atn finalState whileFalse:[state := atn transitionAt: state]
```

4.2 Implementation of the Agent Behavior and Meta-Behavior

To build intelligent control systems for real-life applications (control of mechanical ventilation, manufacturing process, etc.), we need to design agents which range from simple entities to complex entities. These entities own deliberative abilities to reason about complex situations and reactive abilities to meet deadlines.

The use of an active object language brings the benefit of the inheritance mechanism. Thus, we have defined several hierarchies of classes to describe the agent meta-behavior and behaviors.

Figure 4 gives an example of hierarchy of meta-behaviors. We have considered three behaviors: perception, deliberation and communication. `CommunicatingPerceivingDeliberatingAgent` supervises the three behaviors.

At each class is associated an ATN to define the meta-behavior. Each class implements:

- the conditions and actions of the ATN transitions,
- one or several methods for creating agents.

Figure 5 gives an example of hierarchy of behavior. `KnowledgeBasedBehavior` uses a rule base implemented with the rules based-framework *NéOpus*.

The class describing the communication module has a function to control message execution while considering the agent internal state. It has a variety of operations which allow to model the notion of group as proposed by [18]. Each group is reified as a simple agent. Its main function is to forward messages to the related agents.

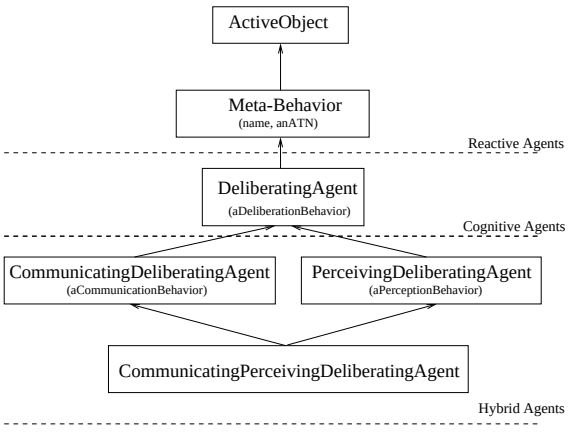

Figure 4: Examples of classes describing agent meta-behavior.

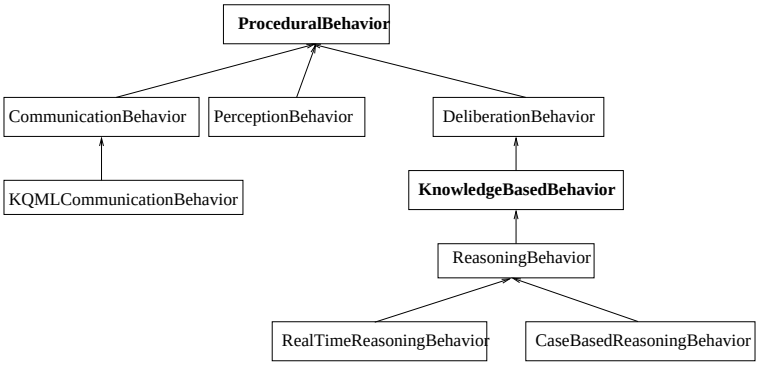

Figure 5: Examples of classes describing the agent behaviors.

4.3 Use of DIMA to Implement Multi-Agent Systems

In *DIMA*, a multi-agent system (MAS) is a set of agents and also possibly a set of objects representing the agents environment. To implement a MAS, one has to implement the objects, which are simple Smalltalk objects, and then to implement the agents.

The main steps to implement an agent are the followings:

1. Determination of the agent class, so to say, the choice of the class describing the agent meta-behavior (see Figure 4).
2. Implementation of the class describing its behaviors by sub-classing or using existing classes (see Figure 4),
3. Implementation of the agent ATN by instantiating the class `ATN`.
4. Creation of the agent by using a method defined in the class selected in the first step. Example of method:

```
PerceivingDeliberatingAgent newAgent: aName
            DeliberationBehavior: aDeliberationBehavior
            PerceptionBehavior: aPerceptionBehavior
            atn: anATN.
```

5. Activation of this agent by using the method `resume`.

5 Experiments

To validate the operational platform (*DIMA*), we have developed several applications [9] [14] [15]. In this section, we report on some application to model the evolution economic agents.

The economic model that we chose is the result of an extensive research [10] which was conducted on a representative sample of the French manufacturing firms. The database is collected by the *Bank of France* and contains information on almost 3000 firms.

This economic model is based upon a twofold conception [20]. First, it defines a firm by its intrinsic assets. A firm is a collection of resources. Second, it introduces some dynamics among competitors. It contends that the path followed by a firm depends on its past trajectories. These paths are the strategic choices made by the firm, either technological investments or market commitments. Thus, the resource endowment of a firm is not a random variable. It is both constrained and constraining.

In this application, we consider a set of economic agents in completion with each other in a market. The economic agents have two behaviors: perception and deliberation. The perception behavior allows the firm to observe the market and to build a competition model which is updated in real-time.

5.1 The Firm Deliberation Behavior

A firm is mainly defined by the following properties:

- The state variables (X vector) represent the different levels of resource owned by the firm;
- The Y variables represent the performances of the firm. They are directly influenced by the X vector.
- A period of action (p) indicates the rhythm of the decision process of the firm;
- The capital (K) gives the firm size and the means it can allocate to investment;
- The budget of investment (B) is the amount of money a firm allocates to improve its X vector;
- Finally, a firm is characterized by the strategy it follows. In our model, a strategy is an order of priority for modification of the X values. For instance, the cost strategy concentrates on the X variables related to the production resources.

Figure 6 describes the firm deliberation behavior. At each period, the firm computes the value V which is an expression combining the firm performances indicators and the competition performances. This value estimates the past strategy. It then calculates its capitalization K, how much it can invest in B, and how to allocate the money according to a chosen strategy.

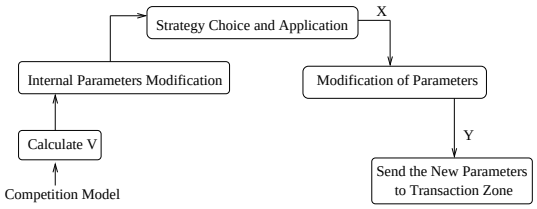

Figure 6: A firm decision process.

To choose a strategy, we have two kinds of decision:

- *simple decision* : the firm uses a fixed decision table,

- *case-based decision* : the firm uses case-base reasoning [22] to choose a strategy by adapting solutions that were used and that have shown good performance. A case is a set of the firm parameters and a set of the competition parameters. The latter defines the competition model which is built by the firm perception module. In the case base, each case has some additional information such as the chosen strategy and the obtained performances.

These modifications of the X vector imply new performances Y, which are sent to the market area.

5.2 The Firm Meta-Behavior

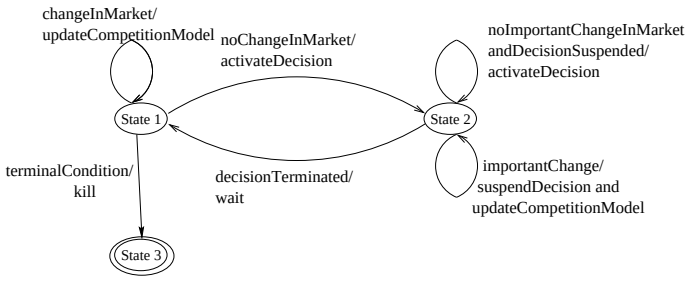

Figure 7: Example of ATN.

The firm meta-behavior is instance of the class `PerceivingDeliberatingAgent`. Figure 7 gives the associated ATN. The latter manages two first-level modules (perception and deliberation). This ATN gives priority to important data. In state 1, the condition "changeInMarket" leads to the action "updateCompetitionModel" and to stay in state 1. The condition "importantChange" leads to the actions "suspendDecision" and "updateCompetitionModel" whatever the state may be.

5.3 The Simulation Experiments

In a first sample version of the economics agent model the main goal of the autonomous agents (firms) is to win over their competitors.

5.3.1 Firm Population Evolution

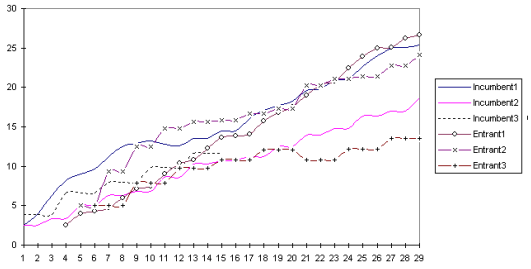

Figure 8: Performance evolution.

In the first series of simulation, we considered three firms (incumbent1, incumbent2, incumbent3) with the same capital and the same initial resource sets. We have tested the effects

of entry on a market. Three new firms have entered the simulation. In this experimentation, the first entrant is as rapid as incumbent1, and the last entrant is as slow as incumbent3. We present the comparison of the performances of the six firms. It appears that the slower firms were the worst performers. incumbent3 was obliged to exit, and entrant3 arrived at the last position. entrant3 has the smaller budget, and the worst performances of the survivors. Reciprocally, incumbent1 and entrant1, the best reactive firms, lead market. They exhibit the best performances, just followed by entrant2.

5.3.2 Learning

In these second series of simulation, we enriched firms with a case-base reasoning mechanism. So, we defined two kinds of agents :

- Agents with a fixed decision process which is implemented as a knowledge base. So, the deliberation behavior of these agent is instance of the class `KnowledgeBasedBehavior` (see Figure 5).
- Agents which build their rule base by studying the evolutionary paths and eventually modify the their strategy set by introducing a new strategy or deleting an existing one. So, the deliberation behavior of these agent is instance of the class `CaseBasedReasoningBehavior` (see Figure 5).

We consider a set of three firms (Firm 4, Firm 5 and Firm 6) with a fixed decision process, and three other firms (Firm 1, Firm 2 and Firm 3) with the same initial characteristics but with learning abilities. These simulations show that the most efficient firm is often a firm with learning abilities (see an example of simulation in Figure 9).

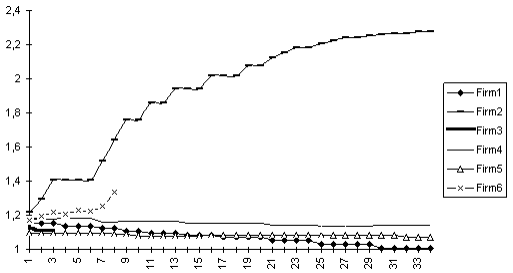

Figure 9: performances evolution of firms with learning abilities.

5.3.3 Discussion

DIMA provides the user with several facilities to implement multi-agent systems. These facilities improve the development time. For example, the economic agents system was implemented in a few days. The number of agents is fixed at the beginning by the user. However, agents may leave and new agents are created dynamically.

Further-more, the use of the inheritance mechanism enables specialization of existing classes to introduce new behaviors. For example, to implement agents with learning abilities, we have reused the class `ReasoningBehavior` describing simple decision (see Figure 5). In the introduced class (`CaseBasedReasoning`), we have defined the method which implements the simple decision. The new method uses case-based reasoning. The implementation of the agents with learning abilitiesdecision has not required any other change. For instance, the ATN of the new agents is the same as the old one (see Figure 7).

6 Conclusion

Several hybrid architectures were proposed (see [11] and [19]) to build agents out of two or more components which can be either deliberative or reactive. The reactive component is given some kind of precedence over the deliberative one. A key problem in such architecture is what kind of control can be used to manage the interactions between these fundamentally different components.

Our architecture proposes a meta-behavior to decompose the behavior of an agent into an organization of behaviors and it uses a meta-behavior to manage the interaction between these different (reactive or deliberative) behaviors. This architecture provides the main property of an agent: autonomy, pro-activity, reactivity and sociability.

The paper presented an extension of active objects to implement a structure of autonomous agents. The developed platform (*DIMA*) includes different frameworks. In addition to the active-object framework (*Actalk*), it uses the rule-based framework *NéOpus* to implement the agent knowledge bases. It also uses the discrete-event simulation framework to represent and to manage the temporal evolution of the implemented agents. On the other hand, the use of a modular architecture and the inheritance mechanism facilitates the introduction of new classes to describe new behaviors.

We validated the platform *DIMA* on several applications: manufacturing process simulator [14]; NéoGanDi [15]: a multi-agent system to control mechanical ventilation [9]; Meveco [16]: a multi-agent system to model economic agents evolution, etc.

DIMA offers an interesting framework for studying multi-agent problems. For example, the implemented agents are mainly characterized by reactivity and adaptability to changes of their environment. To have real-time agents, we are currently studying an *anytime* reasoning technique [29]. The realized experiments [15] offers promising results. However, we have limited our study of real-time aspects to the agent level. Real-time agents are necessary to most real-life applications but they are not sufficient to build real-time multi-agent systems. It seems very interesting to study how the agents society cooperate to solve a global problem in real-time.

References

[1] G. Agha and C. Hewitt. Concurrent programming using actors: Exploiting large scale parallelism. In *5th Conference On Foundations of Software Technology & Theoretical Computer Science*, pages 19–41, 1985.

[2] T. Bouron. *Structures de communication et d'organisation pour la coopération dans un univers multi-agents*. PhD thesis, Paris-6, Laforia, 1992.

[3] J.-P. Briot. Actalk: a testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *ECOOP'89*, 1989.

[4] J.-P. Briot. An experiment in classification and specialization of synchronization schemes. In Springer-Verlag LNCS, editor, *2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, pages 227–249, 1996.

[5] C. Castelfranchi. *Decentralized AI*, chapter A Point Missed in Multi-Agent, DAI and HCI, pages 49–62. Elsevier, Amsterdam, 1990.

[6] C. Castelfranchi. Guarantees for autonomy in cognitive agent architecture. In *Intelligent Agents: Theories, Architectures and Languages, LNAI Volume 890*, pages 56–70, 1995.

[7] P. R. Cohen and H. J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42(1):225–253, 1990.

[8] Y. Demazeau and J.-P. Müller. *Decentralized AI.* Elsevier, 1990.

[9] M. Dojat and C. Sayettat. NéoGanesh: an extensible knowledge-based system for the control of mechanical ventilation. In *14th IEEE-EMBS*, pages 997–1004, 1992.

[10] R. Durand. *Management Stratégique des ressources et analyse de la performance.* PhD thesis, HEC, Paris, 1997.

[11] I. A. Ferguson. *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents.* PhD thesis, University of Cambridge, Clare Hal, 1992.

[12] L. Gasser. *An Overview of DAI.* Kluwer Academic Publisher, Boston, 1992.

[13] L. Gasser and J.-P. Briot. *Distributed Artificial Intelligence: Theory and Praxis*, chapter Object-Oriented Concurrent Programming and Distributed Artificial Intelligence, pages 81–108. Kluwer Academic Publisher, 1992.

[14] Z. Guessoum and P. Deguenon. A multi-agent approach for distributed discrete-event simulation. In *DISMAS'95*, pages 183–190, Poland, November 1995.

[15] Z. Guessoum and M. Dojat. A real-time agent model in an asynchronous object environment. In *MAAMAW'96*, pages 190–203, Eindhoven, The Netherlands, January 1996. Springer Verlag.

[16] Z. Guessoum and R. Durand. Un système multi-agents pour la modélisation de l'évolution économique. In J.-P. Muller et J. Quinqueton, editor, *JFIADSMA'96*, pages 47–57. Hermès, 1996.

[17] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[18] T. Maruichi, M. Ichikawa and M. Tokoro. *Decentralized AI*, chapter Modeling Autonomous Agents and their Groups, pages 215–134. Elsevier Science, 1990.

[19] J.-P. Müller and M. Pischel. Modelling reactive behaviour in vertically layered agent architectures. In *ECAI'94*, pages 709–713, Amsterdam, (NL), 1994.

[20] R. R. Nelson and S. Winter. *An evolutionary theory of economic change.* Harvard University Press, Cambridge, 1982.

[21] F. Pachet. On the embeddability of production rules in object-oriented languages. *Journal of Object-Oriented Programming 8(4)*, pages 19–24, 1995.

[22] C. K. Riesbeck and R. C. Shank. *Inside Case-Base Reasoning.* Lawrence Erlbaum Associates, NY, 1989.

[23] Y. Shoham. Agent0: An agent-oriented programming language and its interpreter. In *AAAI-91*, pages 704–709, 1991.

[24] Y. Shoham. Agent-oriented programming. *AI*, 60(1):139–159, 1993.

[25] D. McKay T. Finin, R. Fritzson and R. McEntire. KQML as an agent communication language. In ACM Press, editor, *Third international conference on information and knowledge management*, November 1994.

[26] W. Woods. Transition network grammar for natural language analysis. *Communication of Association of Computing Machinery*, 10(13):591–606, 1970.

[27] M. J. Wooldridge and N. R. Jennings. Agent theories, architectures, and languages: A survey. *Knowledge Engineering Review*, 10(2), june 1995.

[28] A. Yonezawa and M. Tokoro. *Object-Oriented Concurrent Programming.* The MIT Press, 1987.

[29] S. Zilberstein and S. Russel. Optimal composition of real-time systems. *AI*, 82(1-2):181–213, 1996.