Building Agents as an Extension of Active Objects - Application to the Simulation of Economic Models

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, Jean-Pierre.Briot}@lip6.fr

http://www.lip6.fr/oasis/ {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.

Keywords: active object, agent, implementation, meta-behavior, modularity, reusability, simulation.

Résumé

Cet article étudie une extension du concept d'objets actifs en une structure d'agents autonomes. Nous présentons d'abord les caractéristiques des agents autonomes qui ne sont pas assurées par de simples objets actifs. Nous proposons ensuite une extension du comportement simple d'un objet actif en un ensemble de comportements et un méta-comportement pour superviser leurs interactions. Comme application concrète de ces idées, nous décrivons l'extension d'une plate-forme d'objets actifs (appelée *Actalk*), en une plate-forme multi-agents générique (appelée *DIMA*). Nous discutons de l'implémentation de cette extension. Finalement, nous rapportons des expérimentations de *DIMA* sur la simulation d'agents économiques.

Mots-clés: objet actif, agent, implantation, méta-comportement, modularité, réutilisabilité, simulation.

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" [27] [28], 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] [21]). 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 multi-agent systems [7] [8] [15] and 2) the implementation and modelisation facilities and techniques provided by OOCP [16] 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 multi-agent implementation structure [16]?

This paper is an attempt to give answers to these two questions. It deals with 1) the modelisation requirements of multi-agent 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 multi-agent systems problems by incorporating into OOCP specific representations and computing structures driven by multi-agent systems needs. The paper presents an implementation structure that supports multi-agent systems 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 [9]. More concretely, in this paper we will describe how to extend a model of active object (named the *Actalk* framework) [3] [4], towards a generic and modular architecture 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 [20] 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 [32]).

2.1 Actalk

Actalk is a framework implementing various kinds of active object models [32] into a single programming environment based on *Smalltalk*. Asynchronism, a basic principle of active-

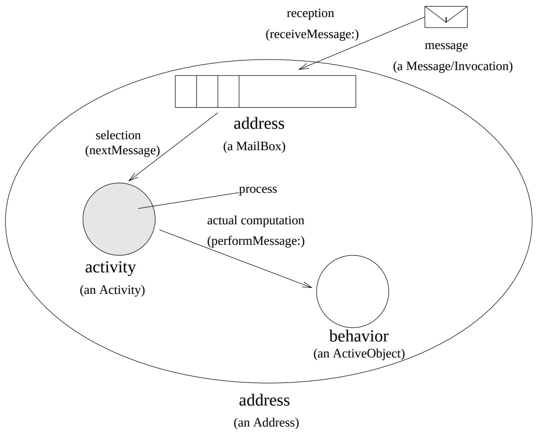

Figure 1: Components of an *Actalk* active object.

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.

Customizing the *Actalk* framework 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:

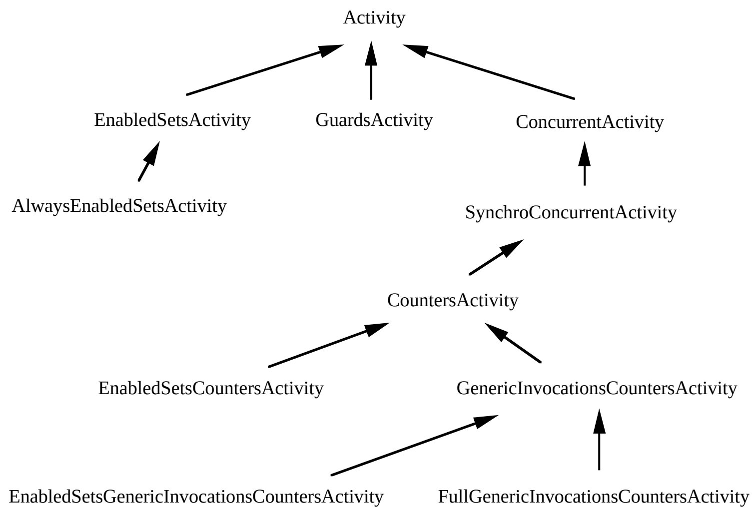

Figure 2: A sample hierarchy of the activity/synchronization classes.

- T. Maruichi proposed concepts to add, to active objects in order to provide autonomous agents [21]. 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 [27] [28].

- J. Bryson and B. Brendan defined an agent architecture as a demonstration of an interaction object-oriented and agent design [5].

- etc.

These are very interesting proposals, however they do not offer a generic agent structure which matches the whole spectrum of multi-agent systems requirements [16]. To make this more clear we will quickly summarize some needed agent properties [31] 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 control over its actions and its internal state [6] [7] [9]. 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 behaviors in accordance with its internal state and its environment. We call environment a representation of the world that the agents perceive and act upon. Environment is made of non agent entities.

- 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 (its acquaintances) by exchanging messages. It handles the other agents requests and generates adequate responses to these requests. For example, cooperating agents are considered to form a group, but this notion of a group is not provided by active objects.

- An agent is an **adaptive** entity. The agent's structure may change over time. The agent has to monitor and to adapt itself. This property is mainly important when agents evolve in a dynamic environment.

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

3 A Generic Agent Architecture

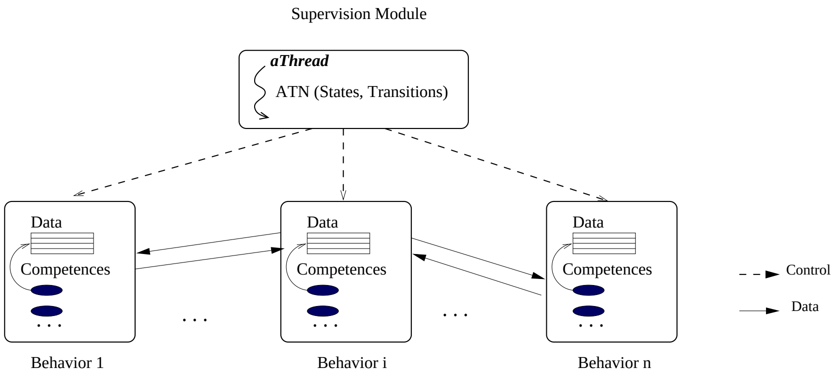

Figure 3: The proposed architecture.

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. These modules represent the different concurrent agent behaviors such as communicating, reasoning and perceiving. They can 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 is necessary to make the agent sociable and the perception module makes its reactive.

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

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: reactive (stimulus-response) and cognitive behaviors. These two kinds will be called respectively procedural and knowledge-based behaviors.

Each behavior encapsulates a set of data and a set of competences (see Figure 3). These competences can be either procedural or knowledge-based. We define a reactive behavior to be one that owns only simple procedural competences, it does not use symbolic reasoning. Moreover, we define a cognitive behavior to be one that contains both kinds of competences (knowledge-based and/or procedural) [31].

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).

These three modules seem sufficient to several application domains (see, for example, section 5). Agents often own a deliberation behavior, and a communication behavior and/or a perception behavior. Moreover, the use of a modular approach facilitates the integration of new modules such as a learning module.

3.1.1 The perception module

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 acquaintances and its environment. The obtained universe model (believes) is used by the deliberation module to reason about the other agents.

3.1.2 The deliberation module

The deliberation module represents beliefs, intentions and knowledge of the agent. It is responsible 1) for generating adequate responses to the messages transmitted by the communication module, or to the changes detected by the perception module, and 2) for achieving the agent goal(s). To do this it relies on a set of competences. There are two kinds of competences: know-how (operative competences) and/or knowledge (cognitive competences). The first one is represented by the standard behavior of the associated objects (example: calculation of a new budget for a firm), and the second one is embodied in a production system [18] [24](example: the decision process which chooses a strategy in a given context).

3.1.3 The communication module

The communication module manages the interactions between the agent and the other agents of its group(s), no matter what machine they are running on. It defines the mail box of the agent and 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] where the received message gives the method to be executed. In this case, it is reactive.

However, agents often use high-level and inter-operable communication language such as those proposed by KQML (Knowledge Query Manipulation Language) [29] which is based on ideas from speech acts. The KQML communication module that we have implemented in *DIMA* is provided with capabilities to reason on the agent state, on its acquaintances state and on the received performative and its various parameters. So, the communication module is cognitive.

3.2 The Agent Meta-Behavior

3.2.1 Issues

Several hybrid architectures were proposed (see [13] and [22]) to build agents out of two or more components which can be either reactive or cognitive. The reactive components are given some kind of precedence over the cognitive ones and the scheduling of these several components is static. Moreover, each component can be viewed as invoking a service in the underlying system which blocks the agent until the action has been completed.

A key problem in such architecture is what kind of control can be used to manage the interactions between these fundamentally different components: reactive and cognitive components.

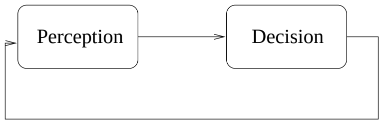

Figure 4: A static scheduler

Let's consider a simple economic agent with two behaviors: a perception and a decision components. Figure 4 gives an example of scheduling that is implicitly used in most existing architectures (see for example [13]). The agent activity may be summarized within a "perception and then decision" loop.

To underline the previous problem, we give the following scenario:

At time t, Firm 1 has a model of its competitive Firm 2 which has an above price and a lower quality. Firm 1 activates its decision process. Parallel to this process, Firm 2 decreases the unit price and increases the quality of its product. If Firm 1 does not deal with these modifications in real-time, its product would not be consumed.

To avoid this problem, a solution is to focus control attention without prescribing a static scheduling. Each agent has to adapt in real-time its scheduling to changes in its surrounding world (the other agents and the environment). So, the agent scheduler cannot be static.

3.2.2 Our solution principles

Many systems have emphasized the need for explicit and separate representation of control or the reflexive aspect of meta-level architecture [25]. Following this tradition of explicit and separate representation of control, we propose a meta-behavior in our agent architecture. This meta-behavior gives each agent the ability to reason about its own behaviors, e.g. in order to make appropriate decisions about control or to adapt its behaviors over time to new circumstances. It provides the agent with a self-control mechanism to dynamically schedule behaviors in accordance to its internal state, and the other agents and the environment states.

This solution has the following foundations:

- behaviors are explicitly and dynamically scheduled by the meta-behavior,
- behaviors are explicitly interrupted to allow the meta-behavior to adapt its scheduling according to new events. Knowledge-based behaviors are provided a time granularity of a single rule firing. In other words, at the knowledge-based competences, scheduling is performed after each rule firing.

3.2.3 Meta-Behavior Representation

The proposed meta-behavior relies on two fundamental notions: states and transitions which naturally build up an Augmented Transition Network (ATN) [30]. Figure 5 gives an example of ATN that may describe an economic agent meta-behavior.

States represent decision points. They are used to choose the next transition beyond the associated transitions.

Each transition is labeled <condition>/<action>. It represents a step of the global scheduling of the agent and links an input state with an output state. The various signals received by the agent's modules (urgent message reception, perception of new data, ...) and

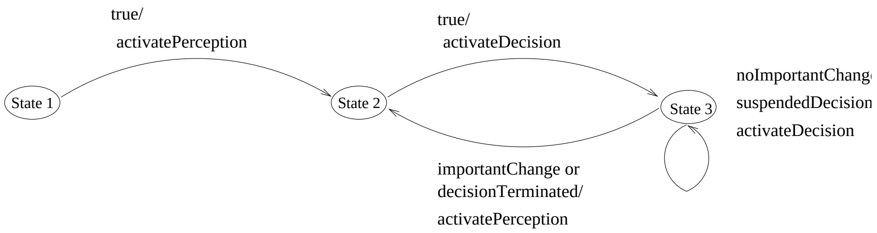

Figure 5: An example of ATN of an economic agent

its own data represent the conditions of transition. The actions of transition allow the management of the first layer behaviors (activate reasoning, terminate reasoning, read mail box, activate perception, ...), the meta-behavior has to choose which behavior it should activate.

Moreover, the meta-behavior may evaluate its behaviors and stop its behaviors when needed. So, each agent has a condition `stopCondition` and an action `kill` (the whole agent). Figure 6 gives an example of an ATN which uses `stopCondition` and `kill`.

The meta-behavior has also a data `aListofBehaviors` which can be updated. For example, it can remove a behavior or add a new one.

At each state, the ATN-based meta-behavior evaluates the conditions of transitions (representing new events) to select the most appropriate behaviors. When these conditions are verified, one transition is executed and then the agent state is modified.

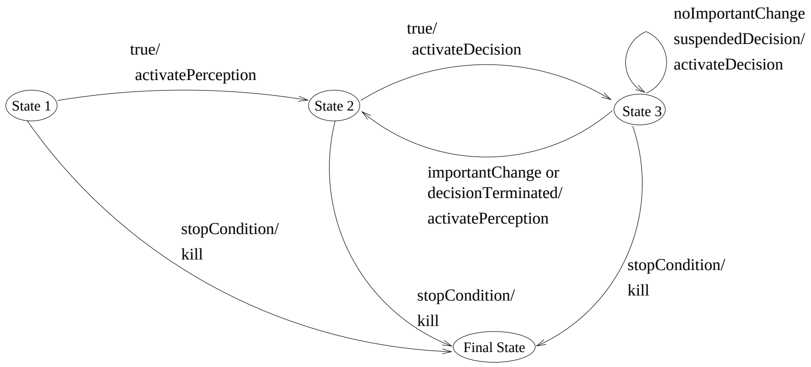

Figure 6: An example of meta-behavior

The meta-behavior represents the agent self-control mechanism. This self-control provides the agent with some kind of control over its actions and its internal state [6] [7] [9]. It defines the agent proactivity which is not restricted to message reception/sending. Therefore, it makes the agent autonomous by allowing him to operate without direct intervention of humans or other agents. Moreover, it makes the agent adaptive by allowing him to deal with new events.

4 Implementation of DIMA with Actalk

The use of an object-oriented language brings the benefit of the inheritance mechanism. So, *DIMA* offers several hierarchies of classes which implement:

- the agent engine,
- some agent behaviors,

- the agent meta-behavior.

Note that we decouple the agent meta-behavior (described by an ATN) and the agent engine (which interprets the ATN). As for expert systems which decouple knowledge bases with the inference engine, this achieves declarativity and modularity.

As the agent meta-behavior represents the analog of the behavior of an active object, it will be implemented as class `MetaBehavior`, defined as a subclass of *Actalk* class `ActiveObject`. The agent engine represents the activity of the agent and thus is implemented by class `AgentEngine`, defined as a subclass of *Actalk* class `Activity`. (See Figure 7.)

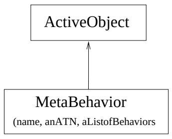

Figure 7: Classes describing the agent meta-behavior and the agent engine.

The class `MetaBehavior` has an attribute `anATN` which defines the meta-behavior, and an attribute `aListofBehaviors` which defines the set of behaviors of the defined agent. This class implements:

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

4.1 Agent Engine

In *Actalk*, the class `Activity` represents the internal activity of the active object.

The instance method `body`, used by `createProcess`, defines the basic loop of the active object which serially processes messages from the mail box.

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

 createProcess
     [self body] newProcess
```

To implement the agent engine, we have subclassed the class `Activity` of *Actalk* to define the class `AgentEngine`. In the latter, the method `body` has been redefined:

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

!AgentEngine methodsFor: 'atn'!
  atnInterpreter
     |atn state|
     atn := self metaBehavior atn.
     state := atn initialState.
     state = atn finalState whileFalse:[state := atn transitionAt: state]
```

Note that, in the agent-oriented programming [27] [28], the cycle of knowledge inference is merged with the cycle of message acceptance. So, the agent activity is limited to the message reception and sending.

In DIMA, the agent activity is described by a meta-behavior which manages the different behaviors (perception, deliberation, communication, etc.). This meta-behavior is described by an ATN. So, the cycle of agent activity is merged with the ATN interpreter.

4.2 Hierarchy of Behaviors

Figure 8 gives an example of hierarchy of behaviors. The class describing the deliberation behavior `DeliberationBehavior` defines a set of competences and a deliberation engine. The deliberation engine defines the scheduling of these various competences.

`KnowledgeBasedBehavior` uses some cognitive competences which are implemented with the rule-based programming framework *NéOpus* [24].

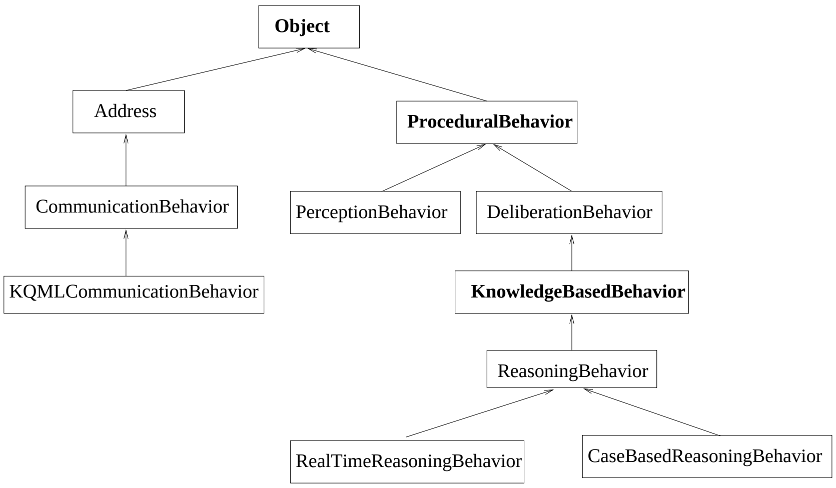

Figure 8: Examples of classes describing the agent behaviors.

The class describing the communication behavior (`KQMLCommunicationBehavior`) has a function to control message execution while considering the agent internal state. It implements a variety of operations which allow to model the notion of group as proposed in [21]. It implements a set of message passing primitives `send` such as:

- sendPerformative: aKQMLPerformative to: anAgentName
- sendPerformative-all: aKQMLPerformative
- sendPerformative: aKQMLPerformative toGroup: aGroupName

4.3 Use of DIMA to Implement Agents

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

Customizing *DIMA* means using or subclassing the hierarchies of classes (see Figure 8 and Figure 7) . This allows to define a set of behaviors and a meta-behavior and then to define the agent.

The main steps to implement an agent are the followings:

1. Determination of the agent behaviors.
2. Implementation of the class describing its behaviors by sub-classing or using existing classes (see Figure 7).
3. Implementation of the agent ATN by instantiating the class `ATN`.
4. Creation of the agent by using a method defined in the class `Meta-Behavior`.
   Example of method:

```
agent := MetaBehavior newAgent: aName
            listofBehaviors: (OrderedCollection with: aDeliberationBehavior
                                                with: aPerceptionBehavior)
            atn: anATN.
```

5. Activation of this agent (`agent resume`).

5 Advantage of the Proposed Agent Architecture

Thanks to its modularity, the proposed architecture helps at decomposing the complex agent behavior into a collection of small specialized behaviors, and supervising these various behaviors by a meta-behavior.

The advantages of this architecture match the agent properties discussed in section 2.2. They may be stated as following:

- **multi-granularity**: Agents of various granularities (size, internal behaviors, knowledge) may be found on the proposed architecture. This property is very important in the design of complex systems. It goes further than the classical dichotomy between reactive and cognitive agents.

- **dynamicity and openness**: Agents may be dynamically created and/or killed. They may integrate new behaviors and change their acquaintances along the information they receive and/or they perceive. Auto-organization may be dynamically reconfigured and new behaviors may be created and integrated by the agent meta-behavior along the needs for new skills.

- **reflection**: The proposed meta-behavior gives each agent the ability to reason about its own behaviors, e.g. in order to make appropriate decisions about control or to adapt its behaviors over time to new circumstances [14].

  Our architecture implements an agent-based model of refection in which each agent has its own meta-behavior which governs its various behaviors. Moreover, this architecture can be used to implement a group-based model of reflection. In this case, the meta-behavior implements a group which governs a set of agents.

- **heterogeneity**: Heterogeneity is a very important issue in multi-agent systems. However, most existing systems do not accept agents talking different languages [12]. The proposed architecture is mainly characterized by: 1) The separation of the agent communication behavior from the deliberation behavior and 2) Each communication behavior has a message-interpreter to translate the received-message language to an internal agent language. These two characteristics allow to have more than one language in the same multi-agent system.

6 Experiments

To validate the operational platform (*DIMA*), we have developed several applications [10] [17] [18]. In this section, we report on some application to model the economic agents evolution.

The economic model that we chose is the result of an extensive research [11] 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 [23]. 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 competition with each other in a market. The economic agents have two behaviors: perception and deliberation. The meta-behavior was described in Section 3.2 (see Figure 6).

6.1 The Firm Perception Behavior

The perception behavior allows the firm to observe the market and to build a competition model which is updated in real-time. The observable data are the ones shown in the market such as the data representing the performances of the various firms. Therefore, the competition model can define data such as:

- the best, worst and middle performances,
- the evolution of these data.

6.2 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.

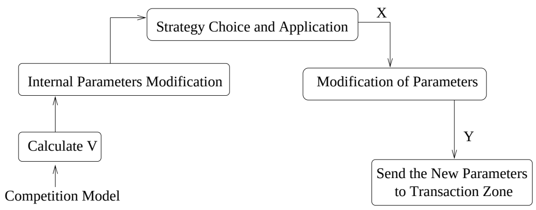

Figure 9: A firm deliberation behavior.

Figure 9 describes the firm deliberation behavior (each box describes a competence). 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. 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 [26] to choose a strategy by adapting the solutions that were used and that have shown good performance. A case is defined as both 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.

6.3 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.

6.3.1 Firm Population Evolution

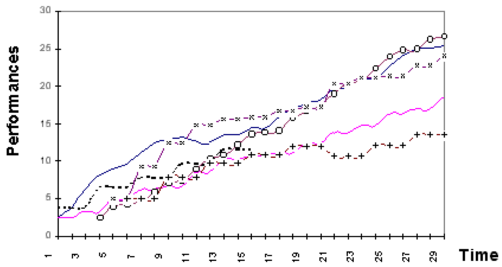

Figure 10: Performance evolution.

In this first series of simulation, we considered a set of firms with the same capital and the same initial resource sets. Each firm has some initial capital and may leave the market. We consider a set of agents with rudimentary behavior that consists in scanning the market to test the need to change their goal and to become real firms. The only difference between these different firms is their period of action.

Figure 11 gives an example of performances evolution of this dynamic set of firms. In this experimentation, it appears that the slower firms were the worst performers.

6.3.2 Learning

In these second series of simulation, we have defined a second kind of firms that use case-based reasoning to choose a strategy. So, we have two kinds of agents :

- Agents with a fixed decision process which is implemented as a knowledge base. The deliberation behavior of these agents is instance of the class `KnowledgeBasedBehavior` (see Figure 8).
- Agents which build their case 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 8).

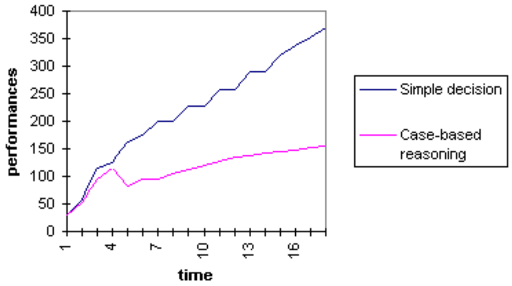

Figure 11: Performances evolution of firms with learning abilities.

We considered a set of firms with a fixed decision process, and another set of firms with the same initial characteristics but with a case-based decision process. These simulations show that the firms with case-based decision are most efficient (see Figure 11).

6.3.3 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.

Furthermore, 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 `KnowledgeBasedBehavior` describing simple decision (see Figure 8). In the new class (`CaseBasedReasoningBehavior`), we have redefined the method which implements the strategy choice. The new method uses case-based reasoning. The implementation of the agents case-based decision has not required any other change. For instance, the ATN of the new agents is the same as the old one (see Figure 5).

7 Conclusion

Several hybrid architectures have been proposed (see [13] and [22]) to build agents out of two or more components which can be either cognitive or reactive. The reactive components are given some kind of precedence over the cognitive ones. 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 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 cognitive) behaviors. This architecture provides the main property of an agent: autonomy, pro-activity, reactivity and sociability.

The paper presented also an extension of active objects to implement a the proposed architecture 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 [17]; intensive care patient monitoring [18] [10]; modelling of economic agents evolution [19], 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 [33]. The realized experiments [18] offer 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*, pages 109–129, 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] J. Bryson and B. McGonigle. Agent architecture as object oriented design. In Springer Verlag LNAI, editor, *ATAL'97*, pages 15–24, 1997.

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

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

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

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

[10] 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.

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

[12] J. Ferber and O. Gutknecht. Alaadin: a meta-model for the analysis and design of organizations in mutli-agent systems. In *ICMAS'98*, page to appear, Paris, 1998.

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

[14] L. Gasser. Conceptual modeling in distributed artificial intelligence. *Journal of the Japanese Society of Artificial Intelligence*, 5(4), July 1990.

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

[16] 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.

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

[18] 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.

[19] 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.

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

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

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

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

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

[25] R. Rao. From the board notion of reflection to the engineering practice of object-oriented metalevel architectures. In *OOPSLA-91 Workshop on Reflection and Metalevel Architectures*, pages 27–31, 1991.

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

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

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

[29] 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.

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

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

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

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