Zahia Guessoum and Jean-Pierre Briot
Objects and Agents for Simulation and Information Systems (OASIS)

The authors describe how they extended a framework of active objects, named Actalk, into a generic multiagent platform, named DIMA. They discuss how they implemented this extension and report on one DIMA application that simulates economic models.

From Active Objects to Autonomous Agents

Object-oriented concurrent programming (OOCP) is the most appropriate and promising technology for implementing agents. Combining the agent concept and the object paradigm leads to the notion of agent-oriented programming.[1] The uniformity of objects' communication mechanisms provides facilities for implementing agent communication, and the concept of encapsulating objects lets us combine various agent granularities. Furthermore, the inheritance mechanism enables knowledge specialization and factorization.

The concept of an active object—integrating an object and activity (namely a thread or process)—provides some degree of autonomy in that it does not rely on external resources for activation. Thus, it provides a good basis for implementing agents. However, an active object's behavior still remains procedural and only reacts to message requests. More generally, we consider that to be autonomous, agents must be able to perform numerous functions or activities without external intervention over extended time periods. To achieve autonomy, several researchers have proposed adding to an active object a function that controls message reception and processing by considering its internal state.[2,3]

There are two basic questions regarding how to build a bridge between implementing and modeling multiagent-system requirements[4,5] and the facilities and techniques OOCP provides:[6]

- How can a generic structure define an autonomous agent's main features?
- How do we accommodate the highly structured OOCP model in this generic structure?[6]

Here we attempt to answer these two questions. We deal with multiagent-system modeling requirements by providing a generic and modular agent architecture. We also discuss extending OOCP's implementation and modeling facilities. More concretely, we describe how to extend an active object's model (the Actalk—*act*ors in small*talk*—framework)[2] towards a generic and modular agent architecture (the DIMA—Development and Implementation of MultiAgent systems—platform).

Active objects

Carl Hewitt introduced the active-object (or actor) concept to describe a set of entities that cooperate and communicate through message passing. This concept brings the benefits of object orientation (for example, modularity and encapsulation) to distributed environments and provides

1092-3063/99/$10.00 © 1999 IEEE

object-oriented languages with some of the characteristics of open systems.[7] Based on these characteristics, various active-object models have been proposed,[8] and to facilitate implementing active-object systems, several frameworks have been proposed. Actalk is one example.

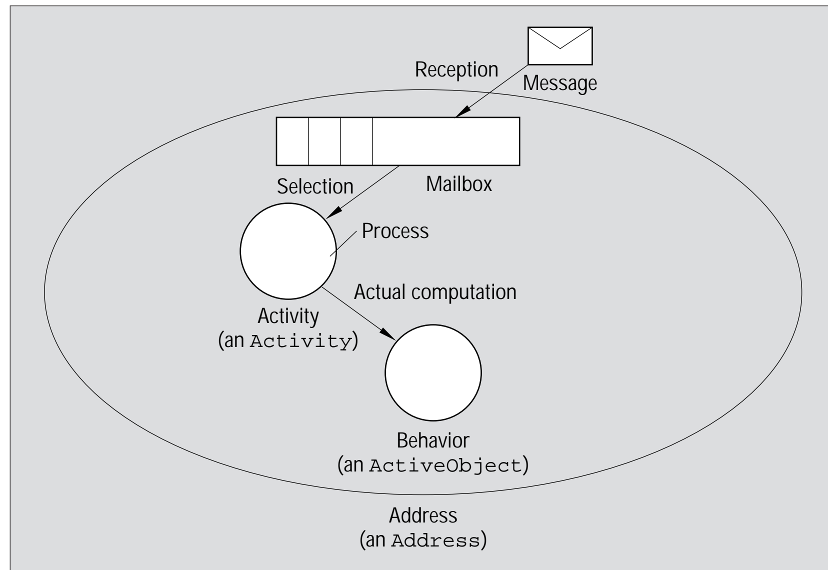

Figure 1. Components of an Actalk active object.

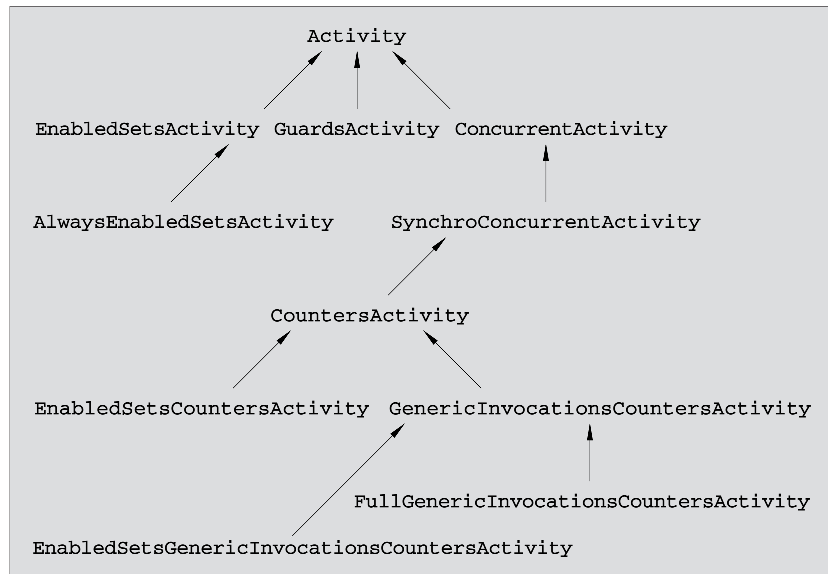

Figure 2. A sample hierarchy of the activity and synchronization classes.

Actalk

Actalk is a framework for implementing and computing various active-object models into a single programming environment based on Smalltalk, an object-oriented programming language. Actalk implements asynchronism, a basic principle of active-object languages, by queuing the received messages into a mailbox, thus dissociating message reception from interpretation. In Actalk, an active object is composed of three component classes (see Figure 1), which are instances of the classes.

- `Address` encapsulates the active object's mailbox. It defines how to receive and queue messages for later interpretation.
- `Activity` represents the active object's internal activity and provides autonomy to the actor. It has a Smalltalk process and continuously removes messages from the mailbox, and the behavior component interprets the messages.
- `ActiveObject` represents the active object's behavior—that is, how individual messages are interpreted.

To build an active object with Actalk, we must 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
    with self as corresponding
    behavior"
  ^self activity: self
    activityClass address:
    self addressClass
```

The `activityClass` and `addressClass` methods represent the default component classes for creating the activity and address components (along the *factory method* design pattern).

Customizing the Actalk framework means defining subclasses of the three component classes. This lets the user define specific models of active objects—various communication protocols as subclasses of the `Address` class and various models of activity and synchronization as subclasses of the `Activity` class (see Figure 2).

Limitations

OOCP provides powerful foundations for modeling and implementing agents. However, these foundations do not really provide a generic agent structure. Active objects are monolithic and have procedural behavior. In spite of their communicating subjects' appearances, active objects do not reason about their behavior, relations, or 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 active-object concept to define a generic agent structure.[1,3] These are very interesting proposals; however, they do not offer a generic agent structure that matches the whole spectrum of multiagent-system requirements.[6] To clarify this, we quickly summarize required agent properties that active objects fail to provide.[9]

Various behaviors

An agent is not monolithic. It can have

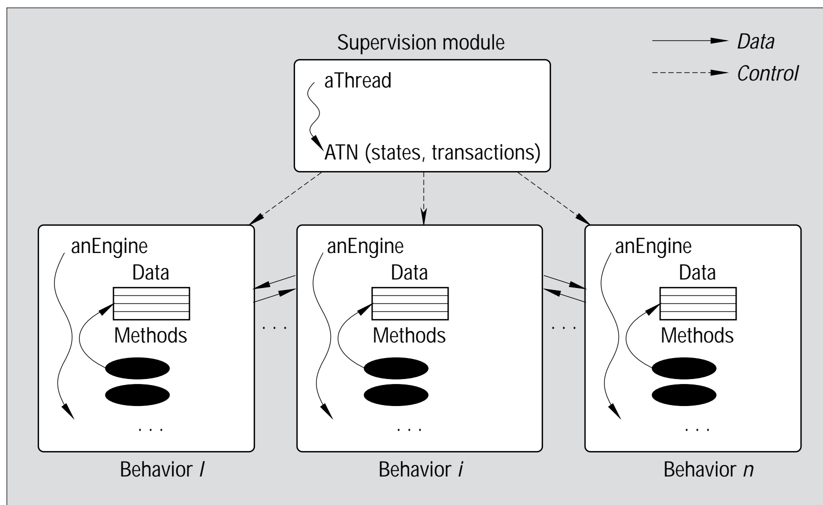

Figure 3. The proposed architecture.

several behaviors: communication, perception, deliberation, and so forth. Each behavior can be procedural but can also include symbolic representations and substantial reasoning mechanisms. For example, the communication behavior can integrate knowledge about agent acquaintances (other agents) and a reasoning mechanism for its interactions with these acquaintances.

Autonomy

An agent operates without direct human or agent intervention. Therefore, it must have some control over its behaviors and internal state[4]—each agent needs autonomous behaviors. For example, the communication behavior can reuse the active object communication mechanism but can integrate a message interpreter to control message acceptance.

The following example shows an interpretation rule that tests the agent's mailbox state and the received message priority before performing this message:

```
rule1
  |Message message|
conditions
  self mailBox size < self
    threshold1.
  message priority >= self
    threshold2.
  actions
  self perform: message
```

Proactivity

An agent has a goal; it does not simply act in response to messages from other agents. For example, in the application we describe later, an agent does not communicate with other agents. It interacts with its environment (made of nonagent entities) and deliberates over the most appropriate action in the current context. Thus, it has two behaviors: perception and deliberation. The agent's proactivity manages these two behaviors.

Sociability

Agents are considered to form groups, but active objects rarely provide a notion of *group*. Moreover, agents can interact by speaking different languages. For example, in agent communication languages such as KQML (Knowledge Query and Manipulation Language),[10] the message's language is indicated in an attribute of the received performative. This language can be object oriented (such as Smalltalk or Java) or an interchange representation language (for example, KIF).

Adaptability

The agent's world (its environment and other agents) continuously evolves. So, some behaviors must enable the agent to sustain performance. For example, in our experiments, the results show that it is difficult to improve a firm's performance without predicting the competitor's performance. Furthermore, our results indicate that using case-based reasoning to predict the competition-model changes improve agent performance.

A generic agent architecture

In an attempt to define a generic architecture that addresses the previously mentioned properties, we propose extending an active object's single behavior into a set of behaviors. This architecture (see Figure 3) relies on a first layer made of interactive modules.

These modules represent different concurrent agent behaviors, such as communicating, reasoning, and perceiving. They can provide the agent with the required properties described earlier. For example, the communication module manages the agent's interaction with the system's other agents. A higher-level supervision module represents agent metabehavior. It lets the agent manage the different behaviors.

Agent behavior

To model complex systems, agents must combine cognitive abilities to reason about complex situations and reactive (stimulus-response) abilities. So, an agent can have two kinds of behaviors: reactive and cognitive.

Each behavior encapsulates a set of data that defines the behavior's state and a set of methods (see Figure 3). The behavior methods or some asynchronous events can update the data. For example, the method scan of the perception behavior manages the interactions between the agent and its environment. It monitors, sensors, and translates sensed data to define a set of beliefs, which can represent a model of the agent's acquaintances and environment.

The methods can be either procedural (such as standard object-oriented methods) or knowledge-based (a method that executes a knowledge base—for example, production rules). A behavior that only contains procedural methods is called a reactive behavior. Otherwise, it contains knowledge-based methods and is called a cognitive behavior. Each behavior has its own engine controlling the activation of various methods. A procedural-behavior engine is a simple automaton. For example, the agent firm's perception-behavior engine defines an infinite loop of activating this behavior's single method scan.

A knowledge-based-behavior engine is an inference engine. To implement the knowledge bases, we use the framework NéOpus,[11] which realizes a neat integration of rule-based programming with Smalltalk. One of its prominent features is declarative specification of con-

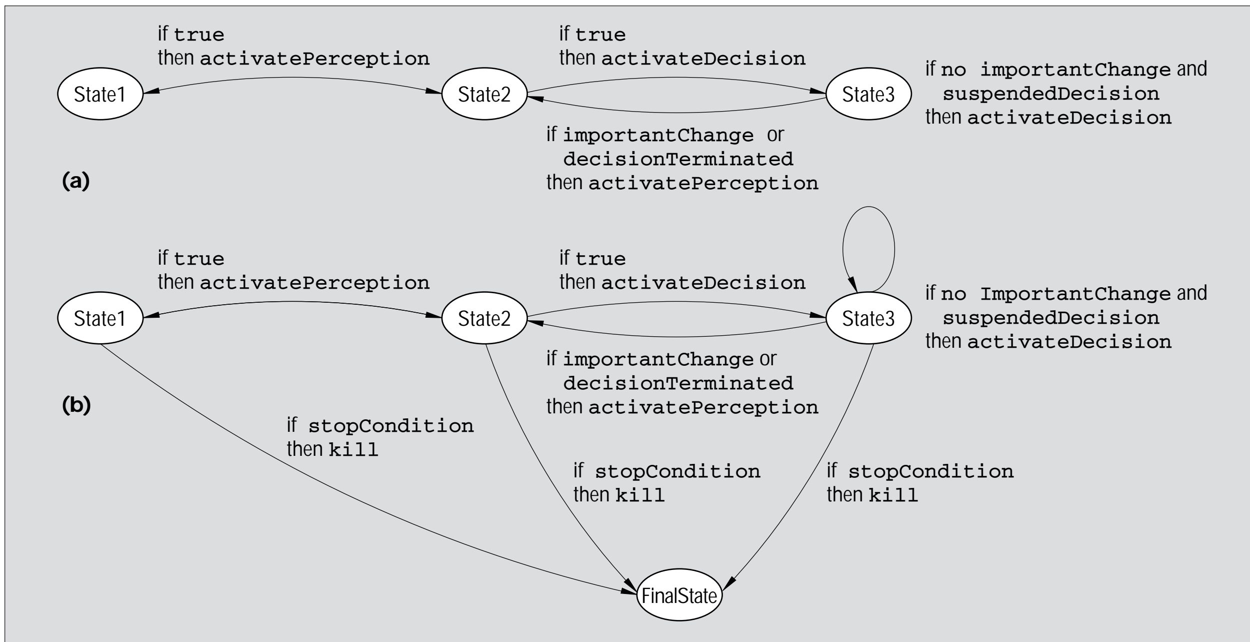

Figure 4. An example of an ATN (a) describing an economic agent metabehavior and (b) using `stopCondition` and `kill`.

trol with metarules. Metarules are rules that operate on so-called control objects. A metabase that controls the firing of its rules is associated with a rule base. NéOpus users have designed several metabases to define standard types of control. We have adapted some of them to our needs by changing a few individual metarules to control firing the set of procedural and knowledge-based methods (we later give an example).

Agents often own a deliberation behavior, and a communication or perception behavior. Moreover, using a modular approach facilitates the integration of new modules such as a learning module. We list three examples of modules that seem sufficient for several application domains (discussed later):

- a *perception module* (procedural behavior) manages the interactions between the agent and its environment.
- a *deliberation module* (knowledge-based behavior) 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 the perception module detects, and for achieving the agent's goals.
- a *communication module* (which can be either procedural or knowledge-based) manages the interactions between the agent and other agents in its group, no matter what machine they are running on. It defines the agent's mailbox and how messages are received and queued for later interpretation.

Agent metabehavior

Several hybrid architectures have been proposed to build agents out of two or more components, which can be either reactive or cognitive.[13,14] The reactive components are given precedence over the cognitive ones, and the scheduling of these components is static. Moreover, each component can invoke a service in the underlying system, blocking the agent until the action has been completed.

A key problem in such architectures is what kind of control can be used to manage the interactions between these fundamentally different components. Let's consider a simple firm with two behaviors: perception and decision components. Static scheduling implicitly used in most existing architectures[12] involves agent activity that can be summarized by a perception-then-decision loop.

To underline this problem of control, consider the following scenario:

> At time t, Firm 1 has a model of its competitor Firm 2, which has a higher price and lower quality. Firm 1 activates its decision process. Parallel to this process, Firm 2 decreases its price and improves the quality of its product. If Firm 1 does not deal rapidly with these modifications, its product will not be consumed.

To avoid this problem, we can dynamically focus control. Each agent must rapidly adapt its scheduling to changes in its world (other agents and the environment). So, the agent scheduler cannot be static, as most existing architectures propose.

Many systems have emphasized the need for explicit and separate control representation or the reflexive aspect of metalevel architectures. Following this traditional representation of control, we propose a metabehavior in our agent architecture that lets each agent make appropriate decisions about control or adapt its behaviors over time to new circumstances. It provides the agent with a self-control mechanism to dynamically schedule its behaviors in accordance to its internal state and internal representation of its world.

Procedural behaviors have a time granularity of a single procedural method call, and knowledge-based behaviors have a time granularity of a single rule firing. In other words, at the knowledge-based methods level, scheduling is performed after each rule firing. This solution is acceptable in many industrial applications.[14]

Our proposed metabehavior relies on two fundamental notions: states and transitions, which naturally build up an *Augmented Transition Network*. Figure 4a

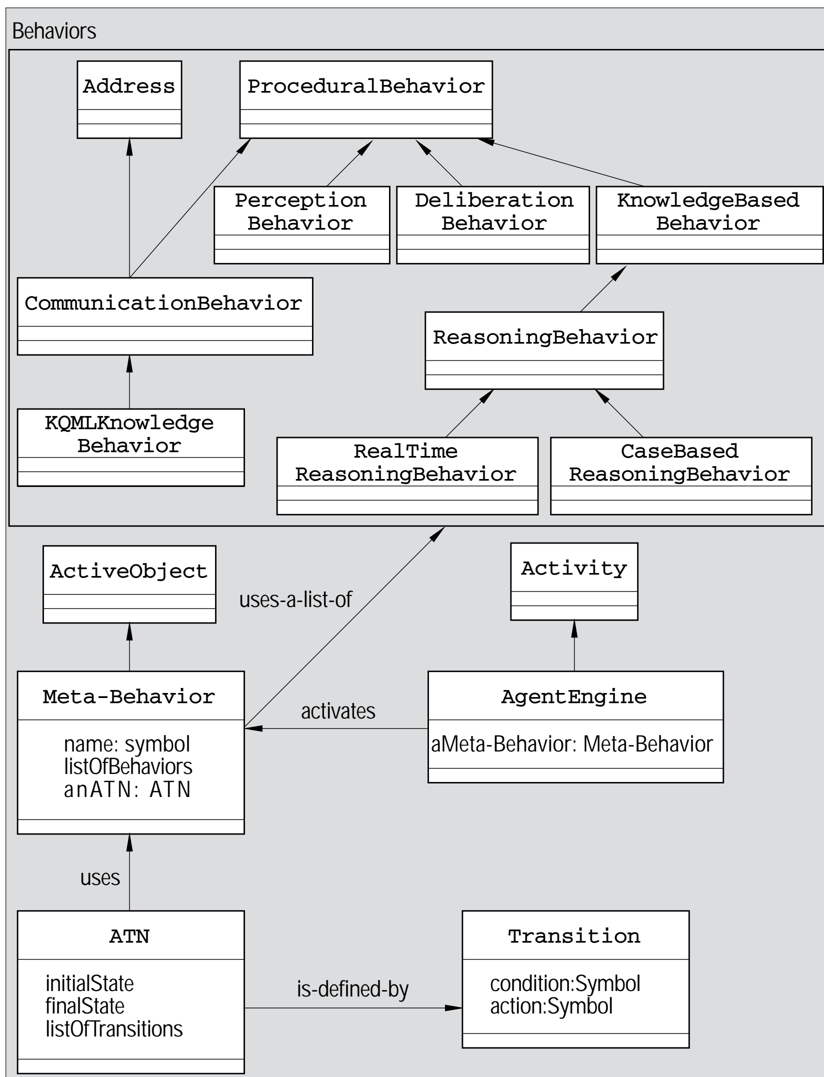

Figure 5. Classes describing the agent components.

gives an example of an ATN that can describe an economic agent metabehavior. States represent decision points, and they are used to choose the next transition among the associated transitions. Each transition is labeled: *if* condition *then* action. A transition represents a step of the global scheduling of the agent and links an input state with an output state.

The transition conditions test the occurrence of an asynchronous event (urgent message reception, new data, and so forth). These asynchronous events often operate changes on the behaviors data. For example, `importantChange` tests for new data to see if it provides an important change according to the previous data. Transition actions let the metabehavior manage the first-layer behaviors (activate reasoning, terminate reasoning, activate communication, activate perception, and so on), so the metabehavior must choose which behavior to activate.

Moreover, the metabehavior can evaluate its behaviors and stop it when needed. It has a (virtual method) condition `stopCondition` and an action `kill` (the whole agent). Figure 4b gives an example of an ATN that uses `stopCondition` and `kill`. The metabehavior also has `aListofBehaviors`, which it can update. For example, it can remove a behavior or add a new one.

At each state, the ATN-based metabehavior evaluates the transition conditions (representing new events) to select the most appropriate behavior. After verifying these conditions, the metabehavior executes the associated transition's actions and modifies the agent state. The metabehavior thus represents the agent's self-control mechanism. This self-control provides the agent with control over its behaviors and internal state. It defines agent proactivity, which is not restricted to receiving and sending messages. Therefore, it makes the agent autonomous by allowing it to operate without direct intervention by humans or other agents. Moreover, it makes the agent adaptive by allowing it to rapidly deal with new events.

Implementing DIMA with Actalk

In DIMA, a multiagent system is a set of agents and possibly a set of objects representing the agents' environment. To implement a multiagent system, we must implement the environment—a collection of simple Smalltalk objects—and implement the agents by customizing DIMA.

Using an object-oriented language provides the inheritance mechanism's benefits, so customizing DIMA means using or subclassing the class hierarchies (see Figure 5). This lets us define the agent's behavior, metabehavior, and engine.

Agent metabehavior

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

As the agent metabehavior represents the analog of an active object's behavior, it is implemented as class `Meta-Behavior` and defined as a subclass of Actalk class `ActiveObject`. The agent engine represents the agent's activity and thus is implemented by class `Agent Engine`, defined as a subclass of Actalk class `Activity` (see Figure 5).

The class `Meta-Behavior` has an attribute `anATN` that defines the metabehavior and an attribute `aListofBehaviors` describes the defined agent's collection of behaviors. This class implements the ATN transitions' conditions, actions, and one or several methods for creating agents.

The main steps to describe an agent are to

1. determine agent behaviors;
2. implement the classes describing its behaviors by using or subclassing existing classes (see Figure 5);
3. implement the agent ATN by instantiating the class `ATN`; and
4. create the agent by using an appropriate method defined in the class `Meta-Behavior`:

```
agent := Meta-Behavior
            newAgent: aSymbol
          listofBehaviors:
          (OrderedCollection
                   with:
        aDeliberationBehavior
                   with:
         aPerceptionBehavior)
          atn: anATN.
```

After completing steps 1 through 4, we can activate the agent engine (`agent resume`).

Agent engine

In Actalk, the class `Activity` represents the active object's internal activity. The instance method `body`, used by `createProcess`, defines the active object's basic loop, which serially processes messages of the mailbox:

```
!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 (see Figure 6).

In agent-oriented programming, as defined by Yoav Shoham,[1] the cycle of knowledge inference implements the cycle of message acceptance. Agent activity is limited to receiving and sending messages. In DIMA, an ATN interpreter represents the agent activity and makes it explicit to the programmer.

```
!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]
```

Figure 6. Implementation of the agent egnine.

Advantage of the proposed agent architecture

Thanks to its modularity, the proposed architecture helps decompose an agent's arbitrarily complex behavior into a collection of small specialized behaviors, and helps supervise these various behaviors with a metabehavior.

This architecture's advantages match the agent properties discussed earlier. The architecture can express agents of various granularities (size, internal behaviors, knowledge), and multigranularity is very important in the design of complex systems. It goes further than the classical dichotomy between reactive and cognitive agents.

Also, agents can be dynamically created or killed. They can integrate new behaviors and change their acquaintances along with the information they receive or perceive. The agent metabehavior can create and integrate behaviors according to required new skills. In addition, our architecture implements an agent-based model of reflection in which each agent has its own metabehavior that governs its various behaviors—that is, to make appropriate decisions about control or adapt its behaviors over time to new circumstances.

Finally, heterogeneity is a very important issue in multiagent systems. However, most existing systems do not accept heterogeneous agents such as agents implemented with or speaking different languages. The proposed architecture is mainly characterized by separating the agent communication behavior from the deliberation behavior. Also, each communication behavior has a message interpreter to translate the received message's language to an internal agent language. These two characteristics allow more than one language in the same multiagent system.

Experiments

To validate the operational platform (DIMA), we developed several applications—for example, a manufacturing process simulator[15] and a multiagent system to control mechanical ventilation.[14] Here we report on an application that simulates a population of firms within a shared market.

The economic model we chose is the result of extensive research conducted on a representative sample of manufacturing firms. The French National Bank (*Banque de France*) collected the database. In this application, we consider a set of firms in competition with each other within a shared market. Again, these agents have two behaviors: perception and deliberation.

The firm's perception behavior

The perception behavior lets the firm observe the market and build a competition model. The observable data are the ones shown in the market such as the data representing the performances of the various firms.

To represent this behavior, we implemented a class `FirmPerceptionBehavior` (a subclass of `PerceptionBehavior`). This behavior has only one method (called `scan`) that aims to build a competition model by observing data changes in the market.

```
scan
  "yCollection is a collection
      of performances of the
      other firms"
  yCollection := self scanMarket.
   self updateMemory.
   self buildCompetitionModel
```

Therefore, an elementary competition

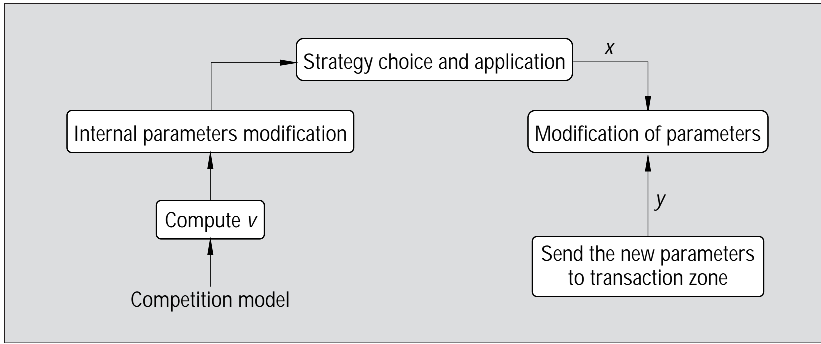

Figure 7. Firm deliberation behavior.

```
!FirmMetaBase methodsFor: 'firm engine'!
  initNoObjects
  | Evaluator e |
  conditions e status = #init.
  e context exists not.
  actions
  self computeV.
  e status: #modifyInternalParameters.
  e modified
(a)

!FirmMetaBase methodsFor: 'firm engine'!
  strategyChoiceEnd
  | Evaluator e|
  conditions
  e status = #strategyChoice.
  e hasSucceeded. actions e strategyChoice
  e status: #modifyParameters.
  e modified
(b)
```

Figure 8. Two examples of metarules: (a) the first activates the method ComputeV before all the other methods, so it tests that no other method is active; (b) the second tests that the method `strategyChoice` has been activated and activates `modifyParameters`.

model can define data such as the best, average, and worst performances:

```
buildCompetitionModel
  ymax := yCollection max.
  yave := yCollection average.
  ymin := yCollection min.
```

This class also implements a set of conditions that are used in the ATN. For example, the method `importantChange` (see Figure 4b) means that the new observed collection of data is different from the previous one.

```
noimportantChange
^self testSimilarityOfData
```

This similarity is defined by

$$\frac{\sum_{i=1}^{n} W_i \left| Y_{t,i} - Y_{t-1,i} \right|}{\sum_{i=1}^{n} \left| Y_{t,i} - Y_{t-1,i} \right| < \epsilon},$$

where ϵ is a constant (~ 0.05) and W_i are weights associated with each Y_i performance variable. An economist expert from HEC (Ecole des Hautes Etudes Commerciales) in Paris with whom we worked defined these weights. Note that

$$\sum_{i=1}^{n} W_i = 1.$$

The firm's deliberation behavior

The following main properties define a firm:

- The state variables (**x** vector) represent the different types of resources (for example, funds, people, or equipment) the firm owns.
- The *y* variables represent the firm's performances. They are directly influenced by the **x** vector.
- A firm is characterized by the strategy it follows to allocate its resources. In our model, a strategy is an order of priority for changing resources (such as hiring new people). For instance, the *cost strategy* concentrates on the **x** vector variables that are related to the production resources.

Figure 7 describes the firm deliberation behavior methods—each box describes a method. For example, `ComputeV` is a procedural method. On the other hand, `StrategyChoiceAndApplication` is a knowledge-based method that chooses a strategy. This method is implemented as a simple knowledge base with NéOpus.[11] The following is an example rule:

```
!FirmRuleBase methodsFor:
      'init'!
  chooseStrategy1
  conditions
  |FirmDeliberationBehavior b|
  b v decreases.
  actions
  b applyStrategy1
```

Class `FirmDeliberationBehavior` is a subclass of `ReasoningBehavior`. As we underlined earlier, this behavior's engine is the inference engine of the system that we use to implement the knowledge base. This system is based on a set of metarules. Figure 8 gives two examples of metarules of this behavior.

Simulation experiments

The aim of these experiments is to underline some properties of the proposed agent architecture. The first property we try to underline is that agents can be added dynamically and can leave the system. So, an agent metabehavior (see Figure 4b) can have one or more transitions with the condition `stopCondition`, which is related to the application domain (for an economic agent, it is defined by

the expression: `capital <= 0`) or by the action `kill`, which kills all the agent behaviors and its metabehavior.

In this first series of simulation, we considered three firms with the same capital and initial resource set. We tested the effects of entry on a market; three new firms entered the simulation and only one has left the market (see Figure 9).

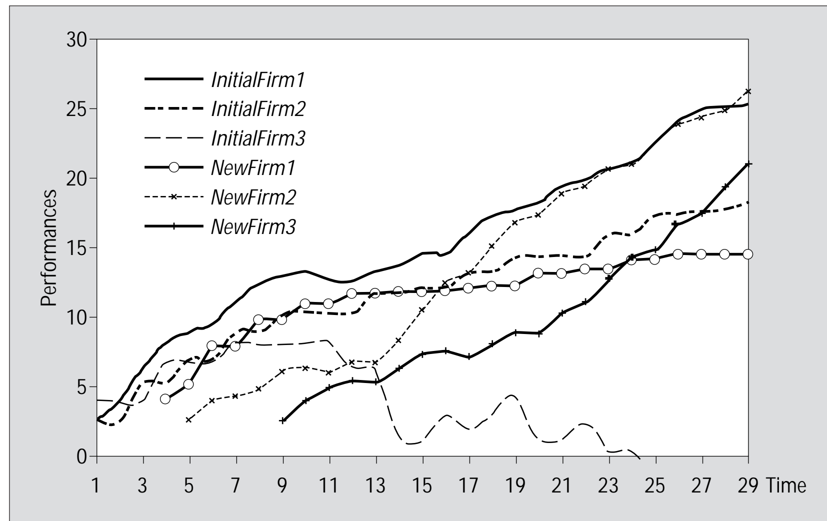

Figure 9. Market evolution

In this second simulation series, we show that in DIMA, we can define heterogeneous sets of agents. We defined a second kind of firm that uses case-based reasoning to choose a strategy (behavior `CaseBasedReasoningBehavior`). So, we have now two kinds of agents: those with a fixed decision process implemented as a knowledge base and those that build their case base by studying the evolutionary paths. The fixed decision process uses a set of rules to choose a strategy, and the reasoning process cannot modify these rules. However, in the case-based decision process, the set of cases is dynamically built by selecting the best cases. The set of cases is thus dynamically modified. To choose a strategy in a given context, the case-based reasoning chooses the most similar case of the base and then it adapts this case to the new context.

These simulations show that the firms with the case-based decision are more efficient than the ones with a fixed decision process (see Figure 10). These results show the merits of adaptive behaviors.

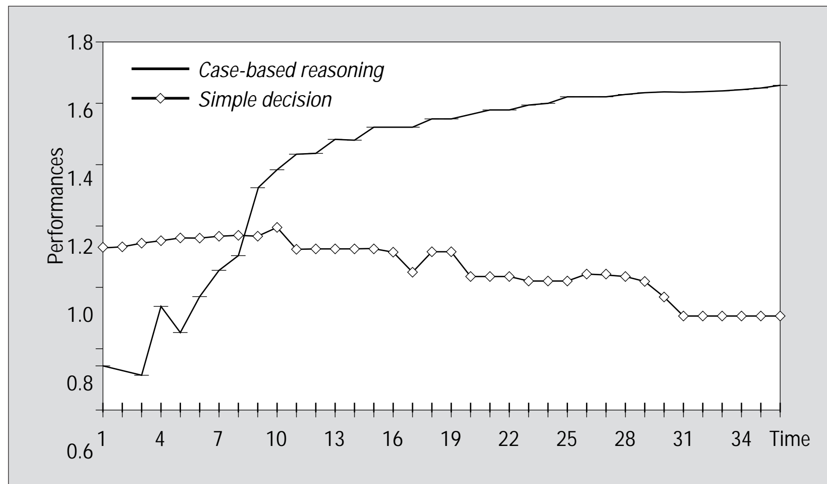

Figure 10. Performance evolution of firms with learning abilities.

DIMA PROVIDES THE USER with several facilities to implement multiagent systems. These facilities improve the development time. For example, the firms' simulation systems were implemented in a few days. Furthermore, using the inheritance mechanism lets us specialize existing classes to introduce new behaviors. For example, to implement agents with learning abilities, we reused the class `ReasoningBehavior` describing simple decisions (see Figure 5). In the new class (`CaseBasedReasoning-Behavior`), we redefined the method `activateDecision` that implements the strategy choice to use case-based reasoning. Implementing the agents with case-based decisions has not required any other change. For instance, the ATN of the new agents is the same as the old one (see Figure 4b).

Our architecture offers an interesting framework for studying multiagent problems. For instance, to describe real-time agents, we're studying an *anytime* reasoning technique. The realized experiments 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 multiagent systems. We hope to study how the agent's society cooperates to solve a global problem in real time.

References

1. Y. Shoham, "Agent-Oriented Programming," *Artificial Intelligence*, Vol. 60, No. 1, 1993, pp. 139–159.
2. J.-P. Briot, "An Experiment in Classification and Specialization of Synchronization Schemes," *Lecture Notes in Computer Science*, No. 1107, Springer-Verlag, New York, 1996, pp. 227–249.
3. T. Maruichi, M. Ichikawa, and M. Tokoro, "Decentralized AI," *Modeling Autonomous Agents and Their Groups*, Elsevier Science, Amsterdam, 1990, pp. 215–134.
4. C. Castelfranchi, "A Point Missed in Multi-Agent, DAI and HCI," *Lecture Notes in Artificial Intelligence*, No. 890, Springer-Verlag, New York, 1995, pp 49–62.
5. L. Gasser, "An Overview of DAI," *Distributed Artificial Intelligence*, N.A. Avouris and L. Gasser, eds., Kluwer Academic, Boston, 1992, pp. 1–25.
6. L. Gasser and J.-P. Briot, "Object-Oriented Concurrent Programming and Distributed Artificial Intelligence," *Distributed Artifi-*

THE Best OF Both Worlds!

IEEE COMPUTATIONAL SCIENCE & ENGINEERING
+ COMPUTERS IN PHYSICS
= **COMPUTING IN SCIENCE & ENGINEERING**

Scientists, **engineers**, and **mathematicians** can learn much from each other about advances in computing. In a groundbreaking development, the IEEE Computer Society has joined forces with the American Institute of Physics to jointly produce ***Computing in Science & Engineering*** magazine. This new magazine bridges a broad range of scientific and engineering disciplines to explore both the scientific and the practical aspects of computers and computation.

This new bimonthly publication continues the strong tradition of focused theme issues and topical articles in the spirit of ***IEEE CS&E***, augmented by the comprehensive departments that have long been the hallmark of ***Computers in Physics***.

As a subscriber, you now get new coverage on
Lab Applications • Algorithms • Scientific Programming
Computer Simulations • Web Mechanics • Practical Visualizations
Essays from the Top • Education • Computing Perscriptions

To subscribe, check out our Web site at **http://www.computer.org**
for special pricing options or contact our Customer Service office at
membership@computer.org

1999 Editorial Calendar

January	Computation in Communication
March	Cosmology and Computation
May	Computational Biology
July	Massive Data Visualization
September	Dynamic Fracture
November	Computational Finance

Computing in SCIENCE & ENGINEERING

cial Intelligence, N.A. Avouris and L. Gasser, eds., Kluwer Academic, Boston, 1992, pp. 81–108.

7. G. Agha and C. Hewitt, "Concurrent Programming Using Actors: Exploiting Large Scale Parallelism," *Lecture Notes in Computer Science*, No. 206, S.N. Maheshwari, ed., Springer-Verlag, New York, 1985, pp. 19–41.
8. *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, eds., The MIT Press, Cambrige, Mass., 1987.
9. M.J. Wooldridge and N.R. Jennings, "Agent Theories, Architectures, and Languages: A Survey," *Knowledge Eng. Review*, Vol. 10, No. 2, June 1995, pp. 115–152.
10. T. Finin et al., "KQML as an Agent Communication Language," *Third Int'l Conf. Information and Knowledge Management,* ACM Press, New York, 1994.
11. F. Pachet, "On the Embeddability of Production Rules in Object-Oriented Languages," *J. Object-Oriented Programming*, Vol. 8, No. 4, 1995, pp. 19–24.
12. I.A. Ferguson, *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents*, PhD thesis, Computer Laboratory, Univ. of Cambridge, 1992.
13. J. Müller and M. Pischel, "Modelling Reactive Behaviour in Vertically Layered Agent Architectures," *11th European Conf. Artificial Intelligence* (ECAI'94), G. Cohen, ed., Wiley & Sons, New York, 1994, pp. 709–713.
14. Z. Guessoum and M. Dojat, "A Real-Time Agent Model in an Asynchronous Object Environment," *Agent Breaking Away*, W. Van de Velde and J. Perram, eds., Springer-Verlag, New York, 1996, pp. 190–203.
15. Z. Guessoum and P. Deguenon, "A Multi-Agent Approach for Distributed Discrete-Event Simulation," *DISMAS'95, Development and Implementation of MultiAgent Systems*, 1995, pp. 183–190.

Zahia Guessoum is an assistant professor at the University of Reims. Her research interests include hybrid multiagent architectures (reactive/deliberative integration), architectures for real-time agents, and applications that control dynamic systems (Process Control Man-Machine Systems) and simulate economics models. She received her PhD in computer science from University Paris 6, France. She is a member of the "Objects and Agents for Simulation and Information Systems" (OASIS) research team headed by Jean-Pierre Briot. Contact her at Zahia.Guessoum@lip6.fr; http://wwwpoleia.lip6.fr/oasis/~guessoum.

Jean-Pierre Briot is a senior researcher at the Centre National de la Recherche Scientifique (CNRS) in France. His general research interests include object-based and agent-based models, and architectures and techniques for high-level and adaptive concurrent and distributed computing. His application fields of interest include information systems, simulation, mobile computing, collective robotics, and computer music. He received his PhD in computer science from University Paris 6, France. He is a member of the "Laboratoire d'Informatique de Paris 6" (LIP6), where he heads the OASIS research team. Contact him at LIP6, Paris 6 - Case 169, 4 place Jussieu, 75252 Paris Cedex 05, France; Jean-Pierre.Briot@lip6.fr; http://www.lip6.fr/oasis/~briot.