Implementing Autonomous Reactive Agents by Using Active Objects

Giuliano Armano
DIEE
Dipartimento di Ingegneria Elettrica ed Elettronica,
University of Cagliari, Piazza d'Armi, I-09123, Cagliari, Italy,
email: armano@diee.unica.it

Eloisa Vargiu
CRS4
Centro Ricerche e Sviluppo Studi Superiori Sardegna,
VI Strada OVEST Z.I. Macchiareddu I-09010 UTA (CA), Italy
email: eloisa@crs4.it

Abstract

This paper briefly outlines the main characteristics of an ongoing research aimed at developing a support library for implementing systems based on autonomous agents. The library is built upon active objects, whose concurrent behavior is disciplined –for each controlled object– by a corresponding controller, according to the dictate of aspect-oriented programming. So far, we implemented a preliminary release of the library, with initial operational capabilities; in particular, agents are able to exhibit only a reactive behavior. To this end, a suitable architecture has been defined that embodies four subsystems, each one being able to deal with a different aspect: social ability, communication, behavior, and task execution. Agents' interaction and communication is performed according to the standard FIPA ACL. CLOS (Common Lisp Object System) has been adopted as the target language for implementing the library, according to the constraints imposed by an ongoing project, and for the sake of rapid prototyping.

1. Introduction

In the last few years, agents have been proposed as a new promising and pervasive technology (see, for example, [1], [2], [3], [4]). Interactions among agents typically are performed according to standardized protocols, which allow them to communicate, collaborate and negotiate within a common environment. Moreover, from a software engineering perspective, agents could be seen as a further improvement in software development technology [5], possibly aimed at replacing objects and components in complex software architectures; in particular when requirements are rapidly shifting over time.

Object-oriented programming is apparently the most appropriate technology to be used for implementing agents, although there are differences between objects and agents that should be pointed out in advance. The first one lies in the degree to which agents and objects are autonomous. An object can be thought of as being autonomous over its state, not over its behavior. The second one lies in the fact that the standard object model does not exhibit a flexible and autonomous behavior, while agents do.

In this paper we describe a support library for developing agent-oriented systems built upon active objects. It is worth pointing out that active objects are needed to give agents the capability of having their own thread of control. Nevertheless, in our opinion, an agent cannot be characterized by a unique active object, as an autonomous agent must be able to deal with communication, autonomy, and flexible behavior. To put together all these features, we deem that an agent has to be implemented as a set of collaborating active objects. In fact, an agent should embody (at least) the following interacting subsystems: social ability, communication, behavior, and task-execution. After discussing such basic issues, we will briefly outline the main features of the library we implemented as a support for building "agent-oriented" systems. CLOS (Common Lisp Object System) has been selected as the target programming language, according to the constraints imposed by an ongoing project, and for the sake of rapid prototyping.

2. Concurrent Programming and Active Objects

Object-oriented programming (OOP) has been presented as a technology that can greatly improve software-engineering activities, as the object model provides a better fit with real domain problems. Nevertheless, there are many problems where OOP techniques are not sufficient to capture all those important design decisions a complex system must implement. In particular, synchronization problems always arise while trying to give a system the capability of executing concurrent threads. To solve these problems, one must be able to exploit additional programming constructs, able to ensure synchronization. According to [6], synchronization may be associated with objects and with their communication means (i.e., message passing) through various sub-levels of identification. Message passing, in a concurrent environment, requires synchronization between sender and receiver, as the classical blocking policy cannot exploit the capabilities offered by a concurrent environment. That is why, some asynchronous types of transmission must be defined; e.g. "future", "past", together with the classical one (i.e. "present") [7]. In fact, object-oriented synchronization schemes can be derived from classical concurrent programming, and a general distinction can still be made depending on whether synchronization specifications are centralized or not. Such constructs, however, suffer from several characterizing drawbacks; e.g., the "inheritance anomaly" [8] and the so-called "code tangling" problem [9]. Solutions to the former problem have been proposed by [10], [11], and [12], whereas the latter could be faced by adopting an approach that separates synchronization issues from operational issues, according to the proposal of [9].

We decided to adopt a synchronization policy that helps avoiding the code-tangling problem. To this end, we enforced a clean separation between a method body and the constructs needed to perform synchronization. In particular, for each object to be synchronized, a separate object is used (as a controller) to take care of any aspect related with thread and concurrency control. We implemented each active object by actually using a pair of objects, i.e., a controlled and a controller object. The former is responsible for performing exported operations, whereas the latter is devoted to deal with any synchronization-related issue. In this way, the problem of dealing with the inheritance anomaly can be suitably faced adopting a solution based on guards (conceptually, a suitable guard -whose semantics is kept separated from the method body itself- embodies each synchronized method).

3. From Active Objects to Autonomous Agents

According to [13], active objects can be considered as the basic structure for building agents. In fact, an active object offers a basic autonomy level, characterized by its reactive behavior. [1] In its simplest form, an agent could be implemented as an active object. Nevertheless, although the concept of active object provides some degree of autonomy (as, to be activated, it does not rely on some external resources), its behavior still remain procedural in reaction to message requests.

On the other hand, agent activities are not limited to receiving and sending messages. To be autonomous, agents must be able to repeatedly perform a number of operations without external intervention. To make the difference between active objects and autonomous agent clearer, we will quickly summarize some agents' characteristics, not provided by active objects:

- an agent may exhibit several behaviors, that can be further decomposed into several elementary ones;
- an agent exhibits several kinds of responsibilities; from simple computations to various reasoning capabilities (some agent behaviors must incorporate AI structures, to integrate a representation formalism and a reasoning mechanism);
- an agent must be able to interact with other agents, thus exhibiting a proper social ability.

Let us consider, in greater detail, the above problems from a behavioral and a communication perspective.

3.1. Behavior

A distinguishing property of an agent is its autonomy. To model complex systems, agents need to combine cognitive abilities to reason about complex situations, and reactive abilities to meet deadlines. Therefore an agent may have two kinds of behaviors: (i) reactive, i.e., based on stimuli-response, and (ii) deliberative, i.e., cognitive.

A reactive agent has only a collection of simple computing schemata, used to react on the environment's

[1] An object is reactive in the sense that it reacts to events consisting of incoming messages. In fact, at least conceptually, the only way to activate an object is by sending a message to it.

change. Thus, it acquires information about the underlying world only by means of its sensors. On the contrary, a deliberative agent has an explicit symbolic representation of the world it has to interact with, in order to take decisions about the action to perform, in a "rational" way.

3.2. Communication

A community of agents is built upon a set of autonomous entities showing communication and cooperation capabilities. To this end, three important questions arise:

- how to adopt a common protocol to implement agents interaction;
- how to conceive a common communication language that they will be able to understand;
- how to find an agreement on the list of terms that can be used in the content of a message (and on their meaning).

To give a response to all above questions, we need a communication language that allows one to specify, (as the standard FIPA ACL does [14]) encoding, semantics and pragmatics of messages. Since, since different agents might run on different platforms and use different networking technologies, FIPA specifies that the messages transported between platforms should be encoded in a textual form.

4. The agents library

First, we implemented active objects, built in accordance with the dictate of aspect-oriented programming. Then, we started developing a support library, devoted to provide facilities for developing agent-oriented systems. It is worth pointing out that we were not interested in building an environment for agent-oriented programming; that is why, we decided to implement only a support library. Nevertheless, this choice does not rule out the possibility of customizing CLOS (at the meta-level); thus giving it the capability of interpreting LISP-like declarations devoted to cope with agents-related structures, policies and mechanisms at a higher level of abstraction.

A preliminary release of the support library has been implemented, with initial operational capabilities that allow building systems of reactive agents. The underlying architecture is a modified version of the classical one, described in [15]. The library will be used within an ongoing project aimed at developing a computer game (mainly in C++). The game lets the user play within a virtual city populated by two different kinds of entities, i.e., physical and network "avatars". Whereas the former ones represent virtual inhabitants, the latter ones live within a virtual computer network (e.g., computer viruses). The first release of the library has been devoted to implement network avatars with a reactive behavior. A prototype of the system has already been implemented, using CLOS as a target language for the sake of rapid prototyping. The second release will be concerned on giving avatars full autonomous capabilities, i.e., planning, reasoning, and learning capabilities. As far as planning is concerned, we are currently defining an extension of the UCPOP planner. Considering that UCPOP is written in LISP, and that PDDL (i.e., the de-facto standard for expressing the domain knowledge for planning algorithms) has a LISP-like syntax, we decided to keep using CLOS as a target language. A suitable DLL will be used to integrate the CLOS code within the C++ code.

4.1. Implementing Active Objects

We decided to implement active objects as a pair of separate, but strictly interconnected, units (i.e., a controlling and a controlled object). Any controlled unit must implement the abstract class `Synchronized`, whereas its controller must implement the abstract class `ProcessController` (see Figure 1). The core of the synchronization model is the class `ProcessController` that contains a semaphore `mutex` that ensures mutual exclusion, and two methods; i.e., `test` and `notify`. The controlled object is equipped with an input and an output queue, devoted to deal with incoming and outgoing messages, respectively. The controlled object is made active by associating a process to it.

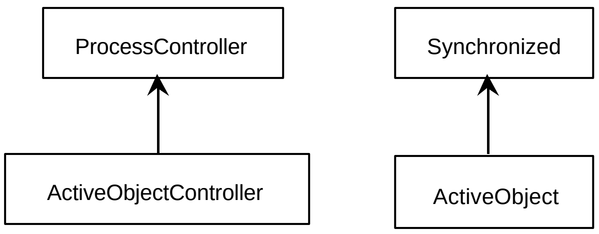

Fig. 1 - *The architecture of an active object.*

When a synchronized method is invoked, a check on method-entry is performed, so that the underlying method body can be executed only when an appropriate guard evaluates to true. On method exit, a notification is performed, to inform all listeners that a concurrent thread of control has been completed. It is worth pointing out that the above check and notification do not need to be explicitly asserted within the method body because `apply-generic` has been suitably redefined, in order to provide synchronization capabilities at the CLOS meta-level.

4.2. Implementing Autonomous (Reactive) Agents

Basically, to define a reactive architecture, one must provide a task-execution and a selection mechanism.

The former is typically realized through a set of behavior modules, continuously taking perceptual inputs and mapping them to an action to perform. Each of these behavior modules is intended to achieve some particular task. On the other hand, the latter usually relies on a set of precedence rules that establish a total order among all available behaviors. In this way, although several behaviors may be “fired” simultaneously, a mechanism able to choose among the different suggested actions is given.

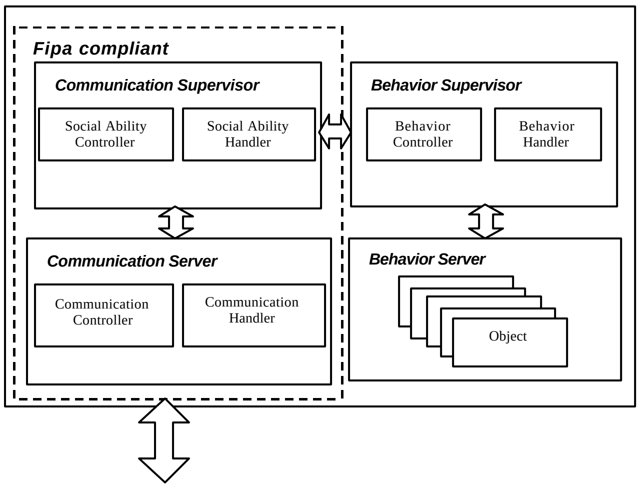

Fig. 2 – *The proposed architecture for implementing reactive agents.*

From our point of view, a reactive agent is an entity that encompasses four distinct subsystems, that embody behavioral and communication capabilities (see Figure 2). In particular:

1. the Behavior Supervisor is devoted to handle the switching among different behaviors;
2. the Behavior Server embodies the objects that actually perform operations;
3. the Communication Supervisor is devoted to handle the cooperation between agents;
4. the Communication Server is devoted to handle the interactions with other agents according to the standard FIPA ACL.

4.3. Behavior Supervisor

The behavior supervisor handles the reactive behavior of the agent, and is implemented using an active object. Its components are instances that implement the classes `BehaviorController` and `BehaviorHandler`, respectively. To specialize a behavior supervisor the above classes must be suitably derived.

The behavior supervisor decides what actions are to be performed and, in case of conflicts, it chooses the most suitable one, according to the precedence rules that have been specified on available actions. After consulting the causal constraints (i.e., precedence rules) that hold among actions, and after selecting an action to be performed, the behavior supervisor sends a request to the behavior server. When an agent must interact with other agents, the behavior supervisor exploit the communication supervisor capabilities; in particular, the latter decides what kind of ACL interaction policy has to be adopted.

4.4. Behavior Server

The behavior server collects the whole functionality embodied by the agent as a collection of objects. Here, objects can be active or passive, depending on the underlying need for synchronization.

The behavior server interacts only with the behavior supervisor. When the latter must perform an action, it sends a request to the former. The behavior server embodies a collection of objects, each one being able to perform all “atomic” actions provided by the corresponding agent.

4.5. Communication Supervisor

The communication supervisor handles the agent interaction policies (i.e., its social ability), and is implemented using an active object. The components of the communication supervisor are instances that implement the classes `SocialAbilityController` and `SocialAbilityHandler`, respectively. To specialize the behavior of a communication supervisor, the above classes must be suitably derived.

After choosing the type of interaction to be adopted (e.g., using a broker, adopting a given kind of streaming, etc.), the communication supervisor delegates the communication server to actually perform the communication according to the standard imposed by FIPA ACL.

4.6. Communication Server

The communication server handles FIPA ACL performatives, and is implemented using an active object. The components of the communication server are instances that implement the classes `CommunicationController` and `CommunicationHandler`, respectively. To specialize a communication server the above classes must be suitably derived.

This server interacts with the communication supervisor and with the external "agentified" environment. The communication server receives from the communication supervisor a request to send a performative to another agent and vice-versa. In other words, the communication server acts as a bridge between the agent and its environment.

5. Conclusions and Future Work

The paper briefly outlines our implementation of a support library for implementing systems based on autonomous agents. The library is built upon active objects, whose concurrent behavior is disciplined -for each controlled object- by a corresponding controller, according to the dictate of aspect oriented programming. So far, agents exhibit only a reactive behavior, and are able to communicate and interact through a subset of the standard FIPA ACL. Further support facilities, able to improve the usability and flexibility of the above library, are still under implementation.

As far as future work is concerned, we are currently studying the characteristics (and implementation details) of more complex architectures, such as vertically-layered architectures (e.g., [17]). In particular, the vertical architecture we are currently working on is a two-pass architecture that defines a behavior layer, a two-tiered plan layer (that encompasses a strategic and a situated planner), and a cooperation layer. Medium-term goal, to be tackled in the next future, will be giving agents reasoning and learning capabilities.

6. References

[1] M. R. Genesereth and S. P. Ketchpel, "Software agents", *Communications of the ACM*, Vol. 37 (7), pp. 48-53, 1994.

[2] N. R. Jennings and M. Wooldridge, "Applying agent technology", *Applied Artificial Intelligence*, Vol. 9(4), pp. 351-361, 1995.

[3] M. Wooldridge, "Intelligent Agents", in G. Weiss, editor, *Multiagent Systems: Modern Approach to Distributed Artificial Intelligence*, pp. 1-51, 1999.

[4] G. Weiss, *"Multiagent systems: A Modern Approach to Distributed Artificial Intelligence"*, 1999.

[5] M. Wooldridge, "Agents and software engineering", *AI*IA Notizie*, Vol. 11(3), pp. 31-37, 1998.

[6] J-P. Briot, R. Guerraoui and K-P. Lohr, "Concurrency and Distribution in Object-Oriented Programming", *ACM Computing Surveys*, Vol. 30(3) pp. 291-329, 1998.

[7] A. Yonezawa and M. Tokoro, " Object-Oriented Concurrent Programming: An Introduction", in A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pp. 1-7, 1987.

[8] A. Yonezawa, "ABCL: An Object-Oriented Concurrent System", 1990.

[9] G. Kiczales, "Aspect Oriented Programming", *ACM SIGPLAN Notices*, Vol. 32(10), pp. 162-162, 1997.

[10] S. Matsuoka and A. Yonezawa, "Analysis of inheritance anomaly in object-oriented concurrent programming languages", in G.Agha, P. Wegner, and A. Yonezawa, editors, *Research directions in Concurrent Object-Oriented Programming*, pp. 107-150, 1993.

[11] S. Ferenczi, "Guarded methods vs. inheritance anomaly inheritance anomaly solved by nested method calls", *ACM SIGPLAN Notices*, Vol. 30(2), p. 49-58, 1995.

[12] A. Poggi and G. Rimassa, “An efficient and flexible C++ library for concurrent programming”, *Software Practice & Experience*, Vol. 28(13), pp. 1437-1463, 1998.

[13] Z. Guessoum and J-P. Briot, “From Active objects to autonomous agents”, *IEEE Concurrency*, 7(3), p. 68-76,1999.

[14] Foundation for Intelligent Physical Agents. Specifications. Available from http://www.fipa.org, 1997.

[15] R.A. Brooks, “A robust layered control system for a mobile robot”, *IEEE Journal of Robotics and Automation*, Vol. 2(1) pp. 14-23, 1986.

[16] J-P. Muller, M. Pischel and M. Thiel, “Modelling reactive behavior in vertically layered agent architectures”, in M. Wooldridge and N.R. Jennings, editors, *Intelligent Agents: Theories, Architectures and Languages*, Vol. 890, pp. 261-276, 1995.