Communicating Process Architectures 2003
Jan F. Broenink and Gerald H. Hilderink (Eds.)
IOS Press, 2003

Agents for Concurrent Programming

Enrique GONZALEZ, César BUSTACARA, Jamir AVILA
Research Groups SIDRE-SIRP
Pontificia Universidad Javeriana - Facultad de Ingeniería
Bogotá, Colombia
egonzal@javeriana.edu.co

Abstract. This paper aims to demonstrate that concepts from Distributed Artificial Intelligence are very useful to design concurrent systems. The BESA framework and the AOPOA methodology are introduced as tools to achieve this goal. The Behavior-oriented, Event-driven and Social-based Agent (BESA) framework combines the concepts of MultiAgent Systems with the design of concurrent systems: an agent can be constructed as a set of behaviors; the notion of behaviors can be directly applied to concurrent systems design using the Agent Oriented Programming paradigm. The internal architecture of a BESA agent integrates two important features: a modular composition of behaviors and an event dispatcher based in a *select* like mechanism. The Agent Oriented Programming based in an Organizational Approach (AOPOA) methodology provides a systematic procedure to build complex system based in three concepts: a hierarchical recursive decomposition of the system, a goal-oriented role identification, and an evolution of the cooperation relationships linking the system components.

1 Introduction

In the last decade, the field of MultiAgent Systems (MAS) has evolved quickly. Applications of MAS based systems include many diverse domains such as information retrieval, robotics, education and training, social simulation, e-commerce, and many others [5][2][3][7]. FIPA, the Foundation for Intelligent Physical Agents, is an organisation aimed at producing standards for the interoperation of heterogeneous software agents [9]. The FIPA architecture includes basic communication services and specifies the Agent Communication Language (ACL). Several FIPA compliant platforms have emerged, some of the most significant Java based ones are JADE, ZEUS, FIPA-OS and April [10]. As MAS based systems are inherently parallel, MAS platforms can fusion elements from the artificial intelligence and from the concurrent system fields; the Behavior-oriented Event-driven and Social-based Agent (BESA) framework is an attempt to accomplish this fusion [13].

This paper aims to demonstrate that concepts from distributed artificial intelligence are very useful to design concurrent systems. In particular, MAS provide a framework inherently concurrent, as they are composed of a set of autonomous entities called agents, working in parallel. In a MAS, agents cooperate to achieve the system goals, the global behavior of the system emerges from the interactions between agents [6]. An agent is an entity that encapsulates data and behavior, in a similar way objects from the OOP paradigm do. Objects are passive, agents are active, agents are continuously acting to achieve their goals in a proactive and autonomous way. An agent is situated in an environment, owns the means to sense and modify its environment, thus agent behavior is closely related to it [3].

Agent concepts can be used to build parallel entities with a high degree of abstraction.

The BESA architecture [13] is the basis of an agent framework that aims to combine the concepts of MAS, as presented in the next section, with the design of concurrent systems. In BESA, an agent is composed of concurrent modules, called behaviors, and includes a channel that supports a *select* like mechanism (cf. section 2.4). Concurrence is present, not only at system level, but also inside agents. In order to design complex systems supported by the BESA platform and following a distributed artificial intelligence approach, an Agent Oriented Programming based in an Organizational Approach (AOPOA) methodology is proposed [12] [11].

This paper is focused in the concurrent design aspects; it presents how the concepts from MAS are implemented and used for concurrent system design. Once the relation between artificial intelligence and concurrent design is analyzed, the fundamental concepts of the three layers of the BESA architecture are introduced. Then a closer look of the concurrent internal architecture of a BESA agent is presented, followed by the main aspects of the AOPOA methodology and their implications for concurrent programming. Finally, some implementation issues and results are discussed.

2 Artificial Intelligence and Concurrent Design

The AI notions can be used to build concurrent systems based in a higher abstraction approach. In particular, agents could be seen as the basic concurrent entities composing the system; interaction between agents provide a clear semantic to the communication between entities; an organizational decomposition allows to deal with complexity; the behavior concept is directly mapped to the *select* mechanism used in concurrent systems.

2.1 Identification of Concurrent Entities

When designing a concurrent system, one of the critical points is to identify entities or modules that can run in parallel to achieve a task in an efficient way. For instance, the methodology presented by Foster performs a four steps process: partition, communication, agglomeration and mapping [1]; the obtained entities when applying this process are meaningless in the context of the task. In opposition, based in the MAS paradigm, a system can be seen as a set of agents, each of them having a well-defined role. A role defines the responsibilities of the agent in terms of its goals, and also identifies unambiguously its relations with the environment and other agents. Several instances of a role can run in parallel in a natural fashion; each of them owning a clear semantics in the context of the problem that the system is dealing with.

From the point of view of software design, it is very valuable to build a system composed of entities with well-defined semantics. The system can be developed and maintained more easily, and also the obtained components can be reused as building blocks in other applications or expanded to enlarge their functionality.

2.2 Semantics of the Interactions

Rationality is one of the most important concerns when dealing with agents from the artificial intelligence perspective [4]. However, when dealing with the design of concurrent systems, what is interesting is to study how agents can interact in a well-defined fashion. The relationships between agents have a high degree of abstraction, thus the model of interactions among the entities of the system posses a well-defined semantics. In fact, there are multiple MAS techniques designed to foster the cooperation between agents.

Cooperation in MAS is achieved by incorporating three essential components: collaboration, which is in charge of task allocation; coordination, that includes synchronization and planning; and conflict resolution which is needed when concurrent resource access or incompatible objectives appear. Ferber [6] and Huhns [2] compile some of the more important techniques to deal with these three problems.

When a cooperation technique is applied, an interaction protocol is established between the cooperating agents. Interaction protocols are well-formed conversations that define communication patterns between the participants; the exchanged messages are considered as speech acts; they contain not only data, but also an intentional semantics [8]. From the perspective of concurrent design, it is very valuable to build a system where the relationships between its concurrent components have a clear semantics: it is easier to construct the system and to verify its correctness.

2.3 Complexity Management

Dealing with complexity is one of the major concerns of software system design; actual systems are large and their requirements are huge, so the relationships among its components become very difficult to manage. The n-tier architectures and the use of software patterns have permitted to construct large systems in a more systematic and robust way [24][25]. The MAS approach offers further concepts and tools to deal with complexity. As it was pointed out before, MAS design provides techniques to identify the set of entities that compose a system. In particular, a MAS can be studied as being an organization [6]; thus, it can be analyzed in a hierarchical recursive way: an organization is decomposed into simpler ones.

This approach is very useful to deal with complexity; if there is a very complex entity, it can be decomposed into simpler ones. The goals and relations of the original organization are delegated to the simpler ones; additional interactions between the new entities can be needed to make them work together. Building complex systems based in a decomposition strategy is systematic and produces well-structured and modular software.

2.4 Non Determinism Management

Minsky [14] proposed an agency model to build intelligent systems: the idea that a mind can be constructed from simple concurrent modules was introduced. This notion of modular components was exploited, mainly in the robotic research field, and complemented with biological inspired principles. The result of this evolution is the concept of behavior as the fundamental building block of intelligence. A behavior is a mapping of sensory inputs to a pattern of motor actions, which are used to achieve a task [15]. Thus, an agent can be constructed as a set of behaviours and the overall behavior of the agent emerges from them; the recursive organizational decomposition is applied again inside the agent.

The notion of behavior can be directly applied to concurrent systems design. A concurrent entity can be seen as being composed of several modules, similar to agent behaviors, each module having well-defined semantics, purpose and responsibility. An entity can be built using a *select* like mechanism, similar to the guard concept implemented in Joyce [22] and the CSP alting-channel [23]. Each module is associated to a different option/guard of the *select* mechanism. When a new stimulus arrives to the entity, it is redirected to the appropriate module by the *select* mechanism. Once again artificial intelligence notions facilitate parallel systems design; in fact, the abstraction provided by the behavior approach makes it easier to develop systems applying the traditional mechanisms for dealing with non-determinism.

3 BESA Model

The abstract model of BESA is based in three fundamental concepts: a modular behavior-oriented agent architecture, an event-driven control approach implementing a *select* like mechanism, and a social-based support for cooperation between agents [13]. In this section, the concept of event used and the layers of the BESA architecture are introduced.

3.1 Event-Driven Control

When building agents it is fundamental to consider the external information coming from the environment, agent behavior derives from it. Normally, the agent reacts to the stimulus provided by the environment. From the concurrent system perspective, stimulus can be modelled as incoming events; in BESA all interactions between agents are modeled by events. Since the agent is cooperating with others, messages coming from them must be also considered as events. In this case, the event includes also the data associated to the message, and represents an speech act. A speech act is more than a communication exchange because it includes a well-defined intention that yields to an action; it constitutes a way to achieve a goal.

In the BESA model, an agent is activated only when a fact, concerning it, occurs. Thus, this event-driven control model avoids CPU busy-wait, agents do not consume processor while waiting for events.

3.2 BESA Architecture

The BESA architecture is composed of three levels: agent level, social level and system level.

Agent Level: an agent is composed by a set of behaviors, a channel acting as an event selector mechanism, and a state. A detailed explanation of the agent architecture will be presented in the next section.

Social Level: it is intended to define and manage hierarchical organizations. These organizations are composed by a set of agents with a common goal. In the BESA model, a facilitator agent could be included in the organization to provide cooperative support to the other agents. The services provided by the facilitator agent include task and resource allocation, conflict resolution, distributed planning and synchronization. This level offers tools to design semantic interactions between the components (agents) of the concurrent systems.

System Level: it comprises the execution environment for agents conformed by BESA containers that could reside in different processing units. In order to comply with the FIPA standard, this level includes a unique entry port that receives ACL request coming from other external MAS [9]. In figure 1, the components of the system level are presented. The container manages the life cycle of the agents; the local administrator supports the agent communication; directory facilitators provided naming and lookup services. Local administrators are synchronized and include automatic replication facilities; it also includes the coordination mechanism to achieve agent mobility.

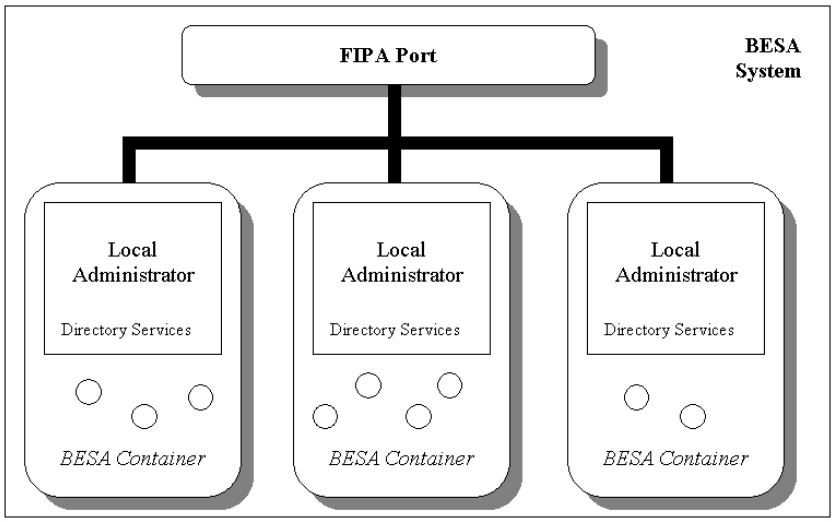

Figure 1. BESA System Level Model

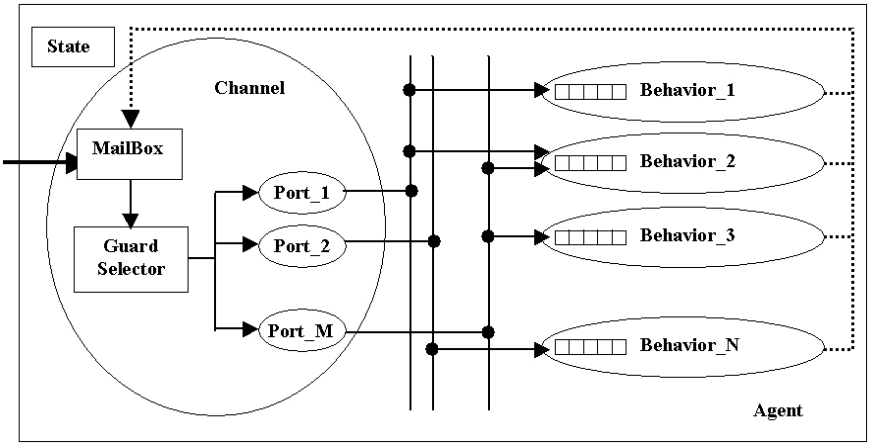

Figure 2. Internal Architecture of an Agent

4 Architecture of a BESA Agent

The internal architecture of an agent integrates three important features: a modular composition of behaviors, a channel implementing an event selector mechanism and an agent internal state. In figure 2 the relations between components are illustrated. In BESA, channel and behaviors run in parallel providing high level of concurrency. This approach allows to easily manage the natural non-determinism present in complex systems. In fact, the channel includes a set of ports to provide a *select* like mechanism. BESA is implemented in the Java language in order to exploit features such as portability, object oriented approach, support for multi-platform and multi-threading, and communication facilities.

4.1 Channel Model

The channel receives all events directed to the agent from the environment or other agents. The channel represents the only entry point to the agent, ensuring that the agents are always ready to receive events in asynchronous way. The incoming events are transferred to the concerned behaviors using the selector mechanism. The channel components are a mailbox, a guard selector and a set of ports.

The mailbox is a temporary mechanism used to store events; it is implemented using a queue. As the channel includes an unique entry point, while an event is picked out and processed, it is possible that new concurrent events arrive; the mailbox assure that these events are not missed.

Once an event arrives, the guard selector receives and processes it. An event includes a tag that defines its semantics; the selector uses this tag to place the incoming event in the appropriated port. There is a unique port for each defined tag in order to avoid ambiguous behavior activation.

The abstract model of a *select* mechanism includes a set of guards that can be concurrently fired in a non-deterministic pattern. A guard is fired when a specified type of event arrives and a logical condition is satisfied. Once the guard is fired, the appropriated actions are executed in response to the incoming event. In the BESA channel, for each guard exists an associated port. When an event is placed in a port, if the respective logical condition is satisfied, the event is transferred to the interested behaviors; otherwise, the event is held in the port. When the logical condition becomes true, if there is any event in the port, the first one must be transferred immediately; in order to assure this requirement, the logical condition is associated to a predefined object; when this object is modified, an attempt to fire the guard is performed automatically.

A channel is implemented as a thread that performs four steps:

- The channel *setup* method is automatically called in order to initialize the local data structures of the agent and to create its behaviors.
- A barrier synchronization of the behaviors is performed in order to assure that all the components of the agent are completely operational before processing any event.
- The heart of the channel run method is an infinite loop that picks out events from the mailbox, in a blocking receive statement, transfer them to the appropriate port, and tries to fire the corresponding guard.
- Finally, when the *kill* method is called, the infinite loop is terminated and the channel *setdown* procedure is executed; it includes behavior destruction and house keeping tasks.

This technique to handle events allows the agent to be ready to react to multiple simultaneous incoming events. As the run method is fixed in advance, the programmer must provide only the methods that are called automatically; which assures the channel robustness and allows to implement the *select* mechanism in a proper way.

4.2 Behavior Model

In the BESA model, an agent is composed by a set of behaviors. Behaviors are concurrent processes that encapsulate the actions that must be executed in response to the received events. A behavior handles at least one type of event; in order to obtain automatically the events of interest, it must bind itself to the corresponding guard. When analyzed from the abstract view, a behavior encapsulates the data and the code required to manage a set of guards.

The behavior has a priority queue, where events are transferred by the channel. All external events have the normal priority level; only some special system events have a higher priority. This design avoids starvation problems, while allowing to deal with special situations.

A behavior is implemented as a thread that performs three steps:

- The behavior *setup* method is automatically called in order to initialize the local data structures of the behavior and to create and bind its associated guards.

- The main part of the behavior run method is an infinite loop that picks out events from its queue, in a blocking receive statement, and depending on the event tag calls the method corresponding to the associated guard.
- Finally, when the *kill* method is called, the infinite loop is terminated and the behavior *setdown* procedure is executed, which includes guard destruction and house keeping tasks.

This technique to build an agent allows it to process several events in parallel; however, notice that events associated to the same behavior are processed sequentially. Since the run method is fixed in advance, the programmer must provide only the methods that are called automatically; which assures the behavior robustness and the execution of actions in a proper way. These actions include changes on the state, generation of events, or modifications of the environment.

4.3 Internal Communication and Mobility

The agent state serves to store its internal state or the model of the environment, which may include its beliefs concerning other agents. The state is unique and shared by all the behaviors, thus a mutual exclusion mechanism is implemented using the monitor supplied by the Java language. The state can be used as a communication mean between behaviors.

In order to respect the semantics of the *select* abstract model, this shared memory mechanism does not allow to directly exchange events between behaviors. Therefore, when a behavior needs to synchronize with another one, it must send an event to the channel mailbox.

Agent mobility is one important feature that can be implemented using the BESA agent architecture. In fact, as the behavior and channel run methods are pre-coded, a synchronization of all the active components can be achieved; thus, allowing to safely stop and restart an agent in order to be moved.

When it is required to move an agent, the agent's channel sends a high priority special event to its behaviors, causing them to terminate its infinite loop without calling the *setdown* method and to stop in a well-defined and stable state. Once this point is reached, the behavior informs the channel, which using a synchronization barrier mechanism detects when the agent is ready to be moved. At this point, the actual BESA container transfers the agent instance to the desired destination container, where a clone of the moving agent is created and started in the adequate point to continue its normal operation. The local administrator retains the events arriving to the agent while the agent moving procedure is performed and sends them to the agent once it has been restarted.

5 Agent Oriented Programming

Shoham introduced the agent oriented programming (AOP) approach [20]. The AOP can be seen as an extension of the object oriented programming (OOP) model; in fact, AOP is a new paradigm to analyze and design engineering systems. The world is inherently concurrent, the agent abstraction allows to map it in a more realistic fashion: the model is closer to the natural parallelism of the problem being analyzed. AOP can incorporate into unique framework notions from the OOP approach and from the concurrent programming techniques, enriched by the artificial intelligence concepts of rational agents. AOP methodologies must be used in order to develop modular, flexible and extensible systems, which can be implemented on agent platforms like the BESA framework.

The construction of flexible systems exhibiting complex behaviors, emerging form the interaction of modular components, can be achieved in a structured fashion [17]. Most of

the known AOP methodologies include, sometimes with different denominations, the following steps: requirements analysis, role identification and task allocation, agent type specification, agent system architecture design, agent internal architecture design, and final system integration. The artifacts produced and the way these steps are performed are particular for each methodology. For instance, Boer argues that five models must be constructed: agent/role, skills, knowledge, organization and interaction [16]. Tropos is supported by two basic ideas: the notion of modeling an agent based in its mental states and a well-defined set of environment requirements [18]. The Gaiga approach analyzes the system in a macro level, the agent society, and in a micro level, the individual agent [21]. Miles approach is centered in the agent interaction analysis, roles as means to achieve goals allow representing the social aspects of the system [19]. Ferber proposes that a MAS can be seen as an organization, which has qualities not present in its individual components [6]. An agent-oriented programming based in an organizational approach (AOPOA) methodology is proposed by Ahogado [12].

AOPOA aims to provide a systematic and general procedure to build complex systems based in three concepts: a hierarchical recursive decomposition of the system, a goal-oriented role identification, and a evolution of the cooperation relationships linking the system components. The resulting MAS is composed by a set of active entities that aim to accomplish a well-defined set of objectives. The organizational approach makes it easier to perform an iterative and recursive decomposition of complex entities into simpler ones based in a goal split and merging procedure; and at the same time to identify the interactions between the entities composing the system. The design process is iterative, an organization level is developed at each iteration. In the analysis phase, tasks and roles are detected; during the design phase, the interactions between roles are specified and managed by cooperation links. At the final iteration, the role parameterization is performed, which allows to specify the events and actions associated to each agent. The abstraction level of the cooperation links evolve: first they represent abstract relations, then a cooperative mechanism is associated to deal with the specific interaction situation, a well-formed interaction protocol is used to implement it, finally an appropriate communication pattern is obtained. Figure 3 illustrates the iterative role decomposition process; the new identified roles must achieve the goals of the original ones; the cooperative links present in *i-th* iteration remain the same; notice that these links are assumed by the new inner roles.

The AOPOA final model includes a set of artifacts that specify and describe in a complete and structured way the roles, the cooperation links and the agent parameterization of the MAS. This model is independent of the platform where the system will be implemented; it contains a set of entities linked by well-defined message exchanges (interaction protocols), and a statement of what actions must be taken in response to each message.

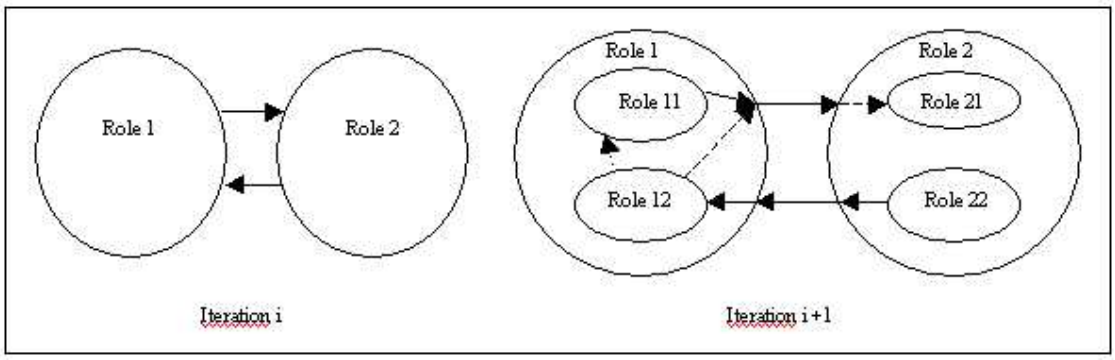

Figure 3. AOPOA role decomposition and cooperative link inheritance.

The BESA platform is very well suited to implement this kind of model. The messages are seen as events; for each event an associated guard is in charge of performing the specified actions. The guards of an agent that aim to assure the same goal and that can run sequentially are grouped into the same behavior. Finally, the obtained agents are deployed in one or several BESA containers.

6 Results and Discussion

The agent and system levels of the BESA architecture have been implemented successfully using the Java programming language; this choice aims to run BESA systems in a heterogeneous, hardware and software, platform. The BESA architecture was evaluated by the implementation of two applications, which include most of the challenging situations that MAS must deal with. The first application case concerns the cooperative robotics field; a multirobot simulator was built, and then extended to control a physical multirobot system; the simulator has been used intensively in artificial intelligence courses. The second study case was used for refining the AOPOA methodology; the simulation of a restaurant was developed following the steps of AOPOA. The obtained model was easily implanted in BESA. In both cases, the operation of the systems was satisfactory, a large number of agents interact concurrently in a correct fashion. In the near future a more detailed evaluation of BESA, including quantitative results, will be performed.

The BESA framework and the AOPOA methodology complement each other. AOPOA provides an iterative systematic procedure to generate a MAS abstract model that fulfills the specific application requirements. BESA incorporates concepts and mechanism from the artificial intelligence and concurrent systems fields to supply a framework that is well suited to implement MAS. When both are combined, a complete development tool for concurrent system design is obtained, which includes identification of concurrent entities, specification of semantic interaction patterns, organization of system complexity, and management of non-determinism. Besides, the system is composed of modular components with increased cohesion and reduced coupling; the inherent high level of abstraction of the agent approach makes the system easy to extend, scale and maintain.

BESA makes easy to develop concurrent applications because the framework hides the low-level details to the programmer, such as thread management and communications. You have to code only the business logic and map it to guards that are grouped in behaviors; the concurrent aspects of programming are automatically provided. However, in order to obtain a proper system, a coherent agent approach must be applied to analyze the specific problem and to design an adequte solution.

The MAS approach is not only the basis of a promising technology, but also it emerges as a new way of thinking: a paradigm for analyzing and designing inherently concurrent systems. In this paper, it has been demonstrated that concepts from distributed artificial intelligence are very useful to design concurrent systems. In particular, the synergy between BESA and AOPOA provides a set of interesting agent based tools to achieve this goal.

Acknowledgements

This work is part of the ASMA project supported by the Pontificia Universidad Javeriana. The authors gratefully acknowledge the contributions of the students working in ASMA related projects.

References

[1] Foster I., Designing and Building Parallel Programs. Addison-Wesley, 1995.
[2] Huhns M. et al., “Readings in Agents”, Morgan Kaufmann, 1998.
[3] Weiss G. (Editor),”Multiagent Systems”, MIT Press, cap. 1-2, 1999.
[4] Russell S., Norving P., "Inteligencia Artificial: Un Enfoque Moderno". Prentice Hall, 1996.
[5] CMU, “The MultiRobot Lab”. http://www-2.cs.cmu.edu/~multirobotlab.
[6] Ferber J., MultiAgent Systems: an Introduction to Distributed Artificial Intelligence, Addison Wesley, 1999.
[7] Knapik M. et al., “Developing Intelligent Agents for Distributed Systems”. McGraw Hill, 1998.
[8] Labrou Yannis, Finin Tim, Agent Communication Languages: The Current Landscape, IEEE Intelligent Systems, March/April 1999.
[9] FIPA Organization, “FIPA - Foundation for Intelligent Physical Agents”. http://www.fipa.org.
[10] FIPA, “Publicly Available Implementations”. http://www.fipa.org/resources/livesystems.html.
[11] Pontificia Universidad Javeriana, “PUJ_Ingeniería – Proyecto de Investigación ASMA”. http://ainsuca.javeriana.edu.co/asma.
[12] Ahogado D., Reinemer A.M., González E., “AOPOA: Aproximación Organizacional para Programación Orientada a Agentes”. Submitted to CLEI’03, 2003.
[13] González E., Avila J., Bustacara C., “BESA: Behavior-oriented, Event-driven and Social-based Agent Framework”. To appear in PDPTA’03, 2003.
[14] Minsky M., The Soceity of Mind, Touchstone Book, 1986.
[15] Murphy R.R., Introduction to AI Robotics, MIT Press, 2000.
[16] De Boer F., “Methodology for Agent-Oriented Software Design”, NOAG-i project, http://www.cwi.nl/~frb/.
[17] Caire G., Leal F., Evans R., “Agent Oriented Analysis using MESSAGE/UML”.
[18] Castro J., Kolp M., Mylopoulos J., “Towards Requirements-Driven Information Systems Engineering: The Tropos Project”, 2001.
[19] Miles S., Joy M. y Luck M., “Designing Agent-Oriented Systems by Analysing Agent Interactions”, Department of Computer Science, University of Warwick Coventry, CV4 7AL, United Kingdom.
[20] Shoham Y.,”Agent-Oriented Programming”, Artificial Intelligence, 60-1, pp. 51-92, 1993.
[21] Wooldridge M., Jennings M. Kinny D., “The Gaia Methodology for Agent-Oriented Analysis and Design”, Autonomous Agents and Multi-Agent Systems 3, Kluwer Academic Publishers, pp. 285-312, 2000.
[22] Hansen B., "A JOYCE Implementation", Software - Practice and Experience, Vol. 17 Nº 4, April 1987.
[23] Hoare C., "Communicating Sequential Processes", Comm. ACM, Vol. 21 Nº 8, Aug. 1978.
[24] Alur D. et al., “Core J2EE Patterns: Best Practices and Design Strategies”. Prentice Hall, 2001.
[25] Gamma E., Helm R., Jonson R., Vlissides. “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley. 1999.