DESIGN PATTERNS FOR MULTI-AGENT SIMULATIONS

Ştefan Boronea,
Florin Leon,
Mihai Horia Zaharia
Gabriela M. Atanasiu
„Gh. Asachi" Technical University of Iaşi

Abstract. *The advent of mobile agent technology has brought along a few difficulties in designing a stable, efficient and scalable system for a certain problem. Agent-based simulations prove to be powerful tools for economic analyses. In this paper we aim at describing a set of design patterns which were specifically built for agents and multi-agent systems. The details of each design pattern discussed are presented and the possible applications and known issues are noted. In order to aid the software designers, we provide some examples of the basic implementation of these patterns using the JADE multi-agent framework.*

Keywords: intelligent agent, multi-agent design, multi-agent simulation.

1. Introduction

Agents are a fairly new programming paradigm that introduces a societal view of computation. An agent can decide its next step without the interference of a human user, or can serve as an intermediary between the user and another device or agent. According to Wooldridge (2000), an agent is a computer system that is *situated* in its environment and is capable of *autonomous* action in order to meet its design objectives. Intelligent agents retain the properties of autonomous agents, and in addition show a so-called "flexible" behavior (Wooldridge & Jennings, 1995):

▪ *reactivity:* the ability to perceive their environment, and respond in a timely manner to changes that occur in it;

▪ *pro-activeness:* the ability to exhibit goal-directed behavior by taking the initiative;

▪ *social ability* to interact with other agents and possibly humans.

Probably the most important difference between traditional object-oriented programming and agent-based programming is the freedom of an agent to respond to a request. When an object receives a message, i.e. one of its methods is called, the control flow automatically moves to that method. When an agent receives a message, it can decide whether it takes a corresponding course of action or not.

2. Agent-based economic simulations

Recently, multi-agent systems have begun to play an important role in the development and analysis of theories in economic and social sciences. Agent-based simulations are much closer to natural processes, and therefore they often ensure better results than those provided by classical methods, such as systems of differential equations. Agent-based modeling is fit to the analysis of complex dynamic processes because it allows a rich representation of the characteristics and actions of the agents, which could not be supported by the formalism of other computational algorithms.

The most studied systems of this kind belong to the economic, social and ecological fields, where finding a mathematical model of evolution is very difficult. However, the representation of system components by autonomous agents is more natural, and the study of the group behavior can indicate important general properties.

In any modeling, the proper ratio between *fidelity* and *abstractization* must be determined. Fidelity assumes the integration into the system of a number of realistic details, and abstractization helps to generalize the experimental results from a system to other systems as well.

Agent-based simulation is applicable to different economic activities. *Economic agents* are suitable to the analytical study of electronic commerce because they make possible the modeling of the complex interconnections that characterize electronic markets. The most important structures in this field are the constraints related to the technological standards, the legislation of electronic commerce and the existing trading relations. The Internet markets differ from traditional markets. Together with the development of the technology for electronic commerce, new transaction models will appear that can lead to the decrease of costs by the intermediation of the selling-buying process by the agents.

The main contribution of artificial intelligence in this field is related to a fundamental theoretic problem – the link between *micro* and *macro* levels, concerning the possibility that unconscious, unplanned forms of cooperation and organization may appear, i.e. the *emergence* of spontaneous order (Leon, 2006).

The fundamental differences between agents and traditional objects, added to the fact that agents can "physically" move within a local network or even the Internet, have led to a new set of challenges in designing agents and multi-agent systems. However, it seems like some experience has already been accumulated in the development of multi-agent systems, and can now be formalized into patterns.

3. Multi-agent design patterns

Patterns were first introduced by Alexander (1979) in the context of civil engineering, and have been adopted by the object-oriented software design community since the „Gang of Four" book (Gamma, Helm et al., 1995). Agent-based systems have become increasingly complex, posing new challenges when designing the

architecture of an application. Agents are best characterized by their mobility and ability to solve certain tasks by communicating and working together with other specialized agents on smaller tasks. It is thus important that in order for an agent system to become proficient, it should provide a large number of small agents that can easily go about a network and maintain the inter-agent communications to a minimum.

The new problems that emerge and the existing problems in agent-based systems will require more patterns to be discovered. In accordance to Schelfthout et al. (2002), the development of agent software has been until now based primarily on a more theoretical approach, with little or no implementation of the concepts presented. Our paper is mostly directed towards the implementation of the existing patterns and finding solutions to possible problems rather than finding new design patterns on a higher abstraction level.

For our study, we have chosen some of the most important design patterns used for mobile agents today. We have followed the pattern categories proposed by Aridor and Lange (1998). The method of dividing the design patterns into these categories leads to a better understanding of what it should do and how it reacts with the environment in which it resides. It also allows the users to plan the agent application as a component-based system, having the components interact to build the application logic.

Traveling patterns are used to recreate a more natural agent movement according to rules specific to each pattern and application. At a high abstraction level, these patterns help to plan ahead routes through the nodes of a system that an agent must follow in order to achieve its goals.

Task patterns are used to split down larger tasks into smaller ones and to distribute them to the appropriate agents in order to provide a continuous system workflow. In a multi-agent environment, a given task can be accomplished either by a single agent or by multiple agents working in parallel and cooperating to accomplish it.

As the system grows in complexity, the need for *interaction patterns* becomes increasingly obvious in order to achieve a good communication between the agents. These patterns describe ways in which agents can interact to solve the given tasks.

4. Traveling patterns

4.1. Itinerary

The *itinerary* design pattern is a very simple and useful pattern that is trying to make use of an important property of an agent: mobility. An agent usually travels through a network on a specific path in order to solve the task it was built for. Thus, a mechanism to describe and validate this movement is needed. *Itinerary* ensures that an agent travels according to a predefined (and possibly dynamic) path and that this path

is a safe one, verifying at every step the existence of its next location. How this happens is described in figure 1.
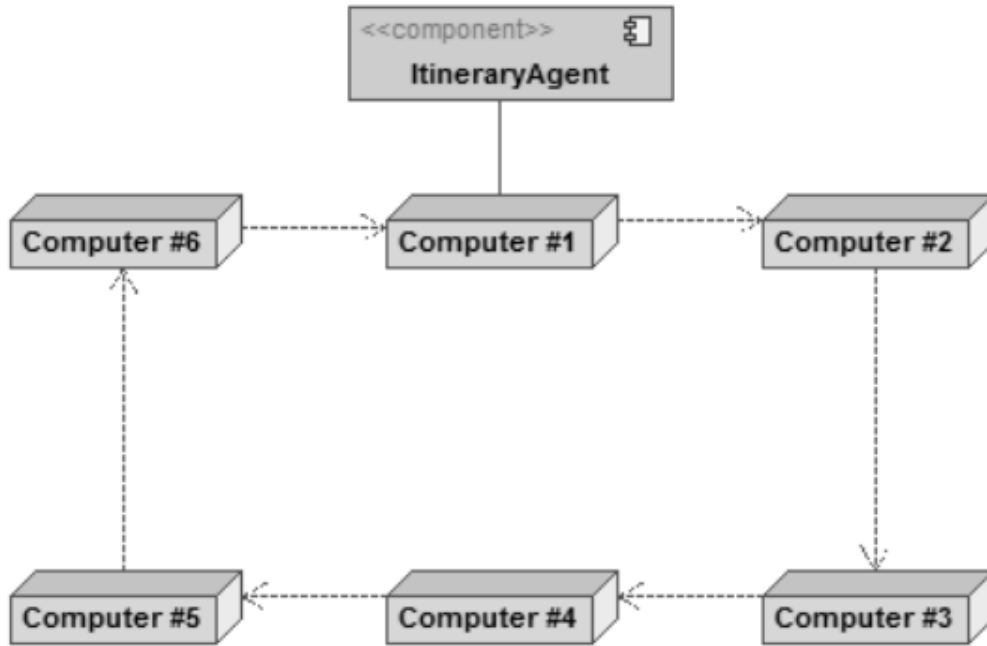

Figure 1. Movement of the *ItineraryAgent* in a network

From a design point of view, its structure consists of a small number of components: the ItineraryAgent and Itinerary classes, the ItineraryAgentJobBehaviour, the GetNextNodeBehaviour and the ItineraryBehaviour behaviors. The Itinerary class consists of a list of destinations in the previously specified manner. On initialization, an agent receives an instance of this class and immediately launches the ItineraryAgentJobBehaviour to do the work it was called for in the current node. When this job is done, the next node present in the itinerary is verified. If it exists, the ItineraryBehaviour is called and the agent migrates to the newly found location. Otherwise, the node is removed from the itinerary and the next node is verified until there are no more nodes available for the itinerary. This process is described for a node in figure 2.

A version of this pattern is when the visited nodes are added in the itinerary, determining a cyclic itinerary. Another variation also routes the agent in a distributed environment according to a routing scheme, having to choose the next node from various (and possibly structurally identical) nodes depending on their properties at migration, thus allowing a load balancing between these nodes or any other application logic of such type.

The uses of this pattern can be multiple, starting with monitoring and logging agents that repeatedly gather information from all (or some) nodes in the application structure, maintenance agents, or simply all agents that are following a route.
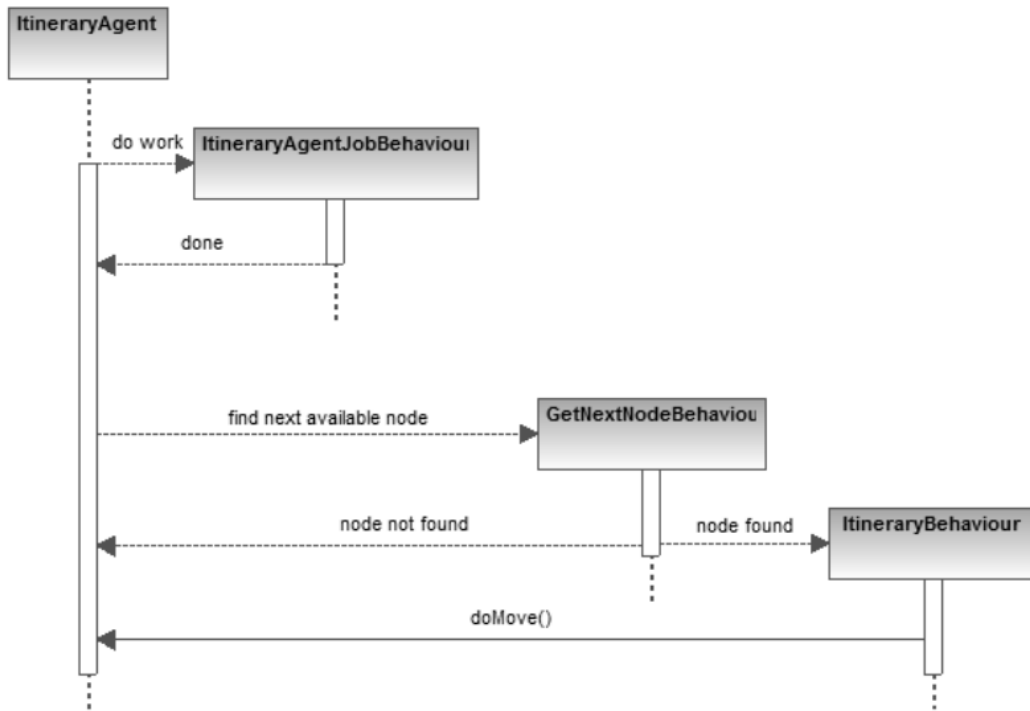

Figure 2. *Itinerary* **pattern sequence diagram**

We can easily discover a useful collaboration between the *Itinerary* design pattern and the *Ticket* pattern presented below. This would allow an agent to store security credentials necessary for it when accessing secure nodes in the network.

4.2. Star-Shaped

The *Star-Shaped* pattern can be considered a particular case of the *Itinerary* pattern, where after visiting each node in the given path, the agent returns to the initial node. In figure 3, one can see how the StarShapedAgent travels back and forth from the central node (consisting of Computer #1) to the other computers in the specified path.

This pattern can be particularly useful for agents that have to gather information from a large number of nodes, information that is very likely to be of a substantial size. Using *Itinerary* in this case would become very inefficient as the agent passes through more and more nodes, collecting information that would make

the migration difficult because of serialization, deserialization and network transfer times. This comes into conflict with the requirement for an agent to be small in size, therefore to be able to travel very fast. The previously mentioned collaboration from the *Itinerary* pattern is also useful here in the context of security-enabled networks.
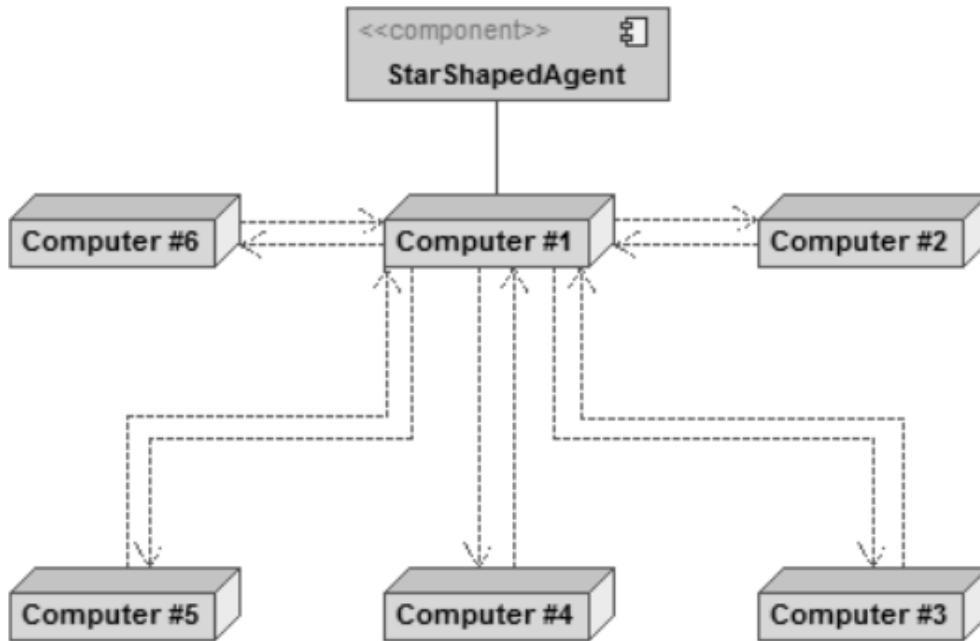

Figure 3. Movement of the *StarShapedAgent* in a network

4.3. Branching

For the *Branching* design pattern, the path defined for the *Star-Shaped* pattern is followed using a different method: at creation, a BranchingAgent clones itself in the central node and then transmits the clones at each location in the specified itinerary. By doing this, we basically have the same functionality described in the previous version, but at the same time we are allowed to execute each job at the remote locations at the same time, without having to wait for the same instance to return to the starting point and then be sent to the following node.

The implementation of this pattern is described in figure 4. Here we can observe that the agent cloning is done in the central node (Computer #1). Afterwards, all the clones are sent to resolve their tasks in a remote location at the same time. All agents return to the central node with the results following their execution. The synchronization of these agents depends on each application and should be taken into

account when we specifically require that some jobs be fired at a later time or only after running these tasks on just a part of the initial node list (when task order matters).
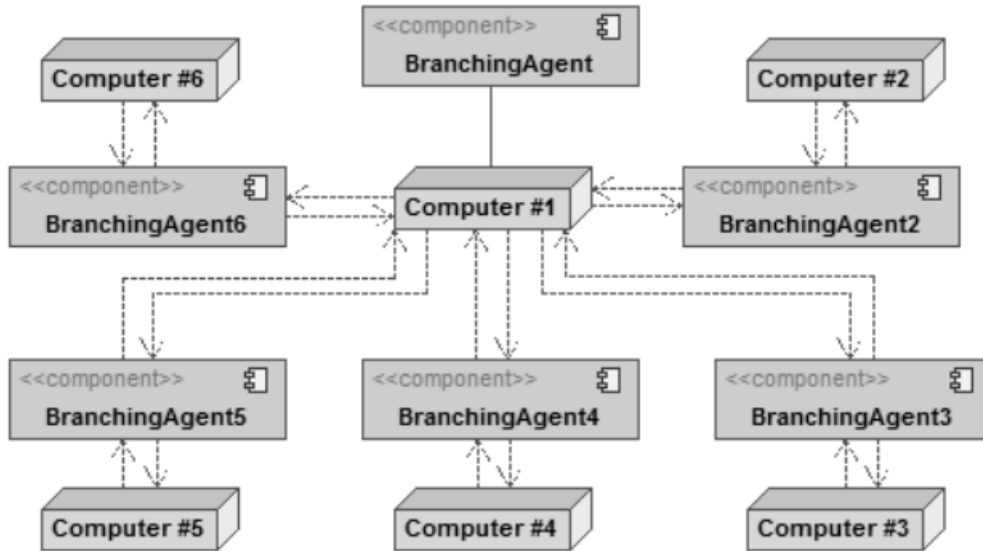

Figure 4. *BranchingAgent* **self-cloning mechanism**

This pattern can be successfully applied in applications consisting of tasks that have to run on a series of nodes. The tasks have to be independent from each other and therefore they can be parallelized in order to obtain an increase in speed and a lower network traffic.

4.4. Forwarding

This simple pattern offers only one functionality to a node, that of forwarding all the incoming agents to another location (or to a series of nodes, depending on the initial call properties). Therefore, that node becomes a router for the agents and their calls. This can be useful when assigning the task of external communication from a local agent network to only one agent, who has the necessary credentials to do so. The agent can act as a dispatcher for the external calls, deciding the priority and which agents have the right to communicate with the zone with a higher security clearance.

From all the other traveling patterns, this one is the most entitled to use the *Ticket* pattern to allow messages to be sent to the upper level. It can also be used with the *Star-Shaped* and *Branching* patterns to allow a custom forwarding of the agents.

4.5. Ticket

As mentioned by Aridor and Lange (1998), the *Ticket* is used to objectify the address of a destination node and to encapsulate the quality of service (QoS) and permissions needed to dispatch an agent to a host address and to execute it there.

The actual implementation for this pattern can be very different according to the security needs and platforms for both the agents and the environment in which they reside. In order to still remain platform independent, these credentials should be accessible to all possible platforms or use *Forwarding* agents that act as authorities and have been delegated control of a local node group.

5. Task Patterns

5.1. Master-Slave

This first task pattern presented helps an agent create other agents for certain tasks that it may have, delegating parts of the work. This can greatly speed up the execution and facilitates parallel execution of tasks. Aridor and Lange (1998) have included an *aglet* implementation of this pattern in their work. The Master Agent is usually a static agent that controls how the tasks are split, issued and resolved in the remote nodes. The Slave Agents are agents with simple tasks that usually require a short amount of processing and which after finishing their work at the remote location turn back at the Master Agent and report back, changing the acquired information and subtask result.

This pattern is similar in structure to *Branching*, but the differences are obvious. The Master Agent does not clone itself to execute the same task in all the nodes, but instead tasks are distributed to slave agents in the system and executed in parallel, requiring that a logic for merging the results be implemented within the Master.

5.2. Plan

This pattern is intended to resolve complex tasks by implementing specific agent-controlled workflows. The uses of workflows in multi-agent systems are numerous and the domains in which they can be applied vary from the improvement of team cooperation and agent-based cognitive flow management (Zhuge, 2003) to collaborative system-on-chip (SoC) design (Trappey, Trappey et al., 2009). Here, the actual subtasks for the goal are distributed to a large number of agents, an approach similar to the *Master-Slave* pattern, but the assignment is usually dynamic and the system must include structures that synchronize the gathering of subtask results and oversee the issuing of subtasks.

Figure 5 displays a simple structure for the workflow consistency. First a WorkflowStateManager agent has to be created and initialized with the required phases for the problem to solve. The available tasks for the first phase are sent to a TaskDispatcherAgent whose sole purpose is to distribute the available tasks received corresponding to their priorities, order, potential for parallel execution for the subtasks and the current state of the workflow. These subtasks are dynamically solved by TaskSolverAgents. If the performance requirements demand it, a collection of previously initialized solver agents for each subtask can be specified to the TaskDispatcherAgent, so that subtasks can be passed without creating them each time. After finishing a subtask, the TaskSolverAgent sends the results of its execution back to a ResultsManager agent that receives, merges and processes all results from the solver agents. In order to maintain workflow integrity, it may decide when the workflow should pass to the next phase and what course of action the TaskDispatcherAgent should take corresponding to the events in the system (e.g a task could not be resolved). Then, a new set of tasks can be transmitted to the dispatcher and the cycle repeated until the results found in the ResultsManager can determine that the problem has been solved.
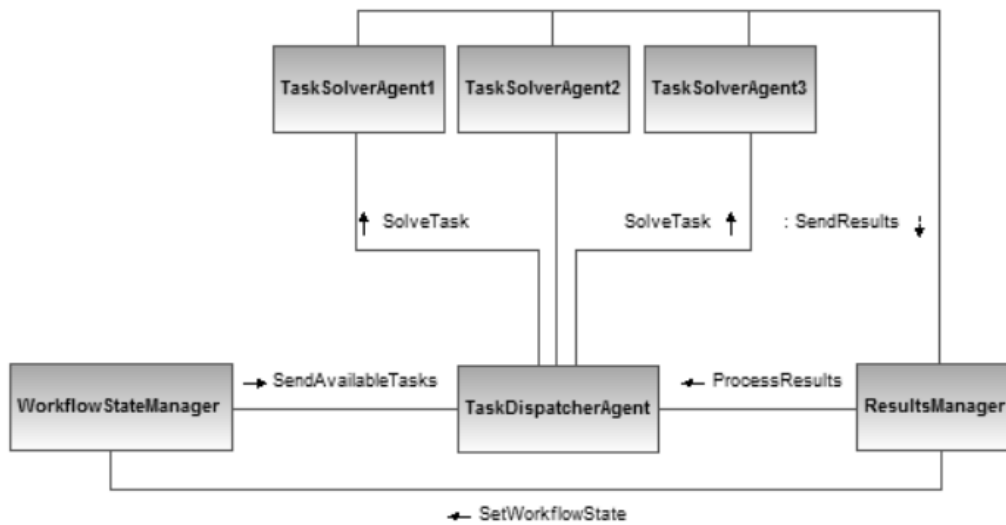

Figure 5. The components of the *Plan* pattern and the interactions required for maintaining workflow consistency

6. Interaction Patterns

6.1. Meeting

Agents can usually communicate without taking note of their location and the location of a partner via a network communication protocol. The main problem occurs when security restrictions or business logic requires that the agents communicate within the same location. The *Meeting* pattern does this by first exchanging the locations of the agents involved, after which they can set a safe location where to meet (Lima, Machado et al., 2004).

6.2. Messenger

This pattern creates an agent that will be used as a messenger between two agents, passing a message directly to the location of the receiver. This can be useful when there is a need for communication between a stationary agent and another agent, and the conditions previously specified for the *Meeting* pattern apply, or when the number of messages sent would make a single-messenger approach inefficient.

6.3. Facilitator

The *Facilitator* acts as a discovery service, ensuring that other agents can inquire about the location of a specific service or agent that can solve a given task. This feature is present in the *JADE* multi-agent framework through the Directory Facilitator (DF) agent. This agent acts like a yellow pages service in which possible service provider agents can publish their services and where user agents search for a specific type of service (Bellifemine, Caire & Greenwood, 2007).

6.4. Organized Group

This pattern allows the creation of groups of agents that work together and are characterized by the ability to migrate together. This can be particularly useful when creating a system of agents that work together to solve a complex task and the performance and bandwidth costs of communicating over a network would be very high. Thus, a local communication approach is needed and the *Organized Group* solves this problem.

7. Conclusions

In this paper we presented a series of design patterns and their respective implementation in the *JADE* multi-agent framework. Because of the flexibility and independence from a specific platform, these patterns can be easily implemented on any other mobile agent platform. We have also discussed the difficulties that could emerge when applying these patterns and how we can overcome them. We have also mentioned how some of the patterns can work together and form a more complex structure with more tasks to resolve. We can thus conclude that the design patterns presented here can greatly ease the work of software designers and improve the performance of the system as a whole by lowering the communication costs between agents by providing direct, efficient solutions to some specific problems.

Acknowledgements

References

Alexander, C. (1979), *The Timeless Way of Building*. Oxford University Press, New York: University Press

Aridor, Y., Lange, D. B. (1998), Agent design patterns: elements of agent application design, in *AGENTS '98:* Proceedings of the second international conference on autonomous agents, New York, NY, USA, ACM, pp. 108-115

Bellifemine, F. L., Caire, G., Greenwood, D. (2007), *Developing multi-agent systems with JADE*, Wiley Series in Agent Technology, Wiley

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995), *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc, USA

Leon, F. (2006), *Intelligent agents with cognitive capabilities*, Tehnopress, Iaşi, România

Lima, E.F.A., Machado, P.D.L., Sampaio, F.R., Figueiredo, J. C. A. (2004), *An approach to modelling and applying mobile agent design patterns,* SIGSOFT Softw. Eng. Notes 29(3), pp. 1-8

Schelfthout, K., Coninx, T., Helleboogh, A., Holvoet, T., Steegmans, E., Weyns, D., Distrinet, A., Agent implementation patterns, in Proc. Workshop on Agent-Oriented Methodologies, 17th *Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications,* OOPSLA02, 2002, pp. 119-130

Trappey, C.V., Trappey, A.J.C., Huang, C.J., Ku, C.C., The design of a jade-based autonomous workflow management system for collaborative soc design. *Expert System Applications* 36(2), 2009, pp. 2659-2669

Wooldridge, M. (2000), Intelligent agents, in G. Weiss (ed.), *Multi-agent systems – A modern approach to distributed artificial intelligence.* The MIT Press, Cambridge, Massachusetts

Wooldridge, M., Jennings, N.R. (1995), *Intelligent agents: Theory and practice*. The Knowledge Engineering Review, Vol. 10(2), pp. 115-152

Zhuge, H. (2003), Workflow-and agent-based cognitive flow management for distributed team cooperation. *Information Management* 40(5), pp. 419-429