Living design for open computational systems

Jean-Pierre Georgé, Gauthier Picard, Marie-Pierre Gleizes, Pierre Glize

▶ To cite this version:

Jean-Pierre Georgé, Gauthier Picard, Marie-Pierre Gleizes, Pierre Glize. Living design for open computational systems. 11th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2003), 2003, Linz, Austria. pp.389-394, ⟨10.1109/ENABL.2003.1231442⟩. ⟨hal-01205630⟩

HAL Id: hal-01205630

https://hal.science/hal-01205630v1

Submitted on 17 Oct 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Living Design for Open Computational Systems

Jean-Pierre Georgé, Gauthier Picard, Marie-Pierre Gleizes, Pierre Glize
Institut de Recherche en Informatique de Toulouse, 118, route de Narbonne,F-31062 Toulouse Cedex
{george, picard, gleizes, glize}@irit.fr
http://www.irit.fr/SMAC

Abstract

Since Open Computational Systems are complex and dynamic structures with a great (but unknown) number of autonomous interacting entities, designers face up to a difficult problem: how can they completely specify such systems? We call Living Design the biologically inspired solution we expound here and which consists of the observation and manipulation during the design phase of the system being built as it "lives". More formally, in terms of design methodology, we propose an Object-Agent Overlapping Processes Shifting which is an extension and modification of the classical design phases. This is to be realized in our work-in-progress Adaptive Multi-agent System methodology called ADELFE, which is centered on the AMAS theory also briefly expounded here.

1. Introduction

When facing up to complexity, distribution and openness, designers have to develop tools to model applications for which the requirements are not so clear. In this kind of applications, designers cannot predict all situations the system will encounter during its running time because the environment and the system are dynamical, e.g. an application consisting in the simulation of a naval scenario [9]. Multi-agent systems seem to fit well to these applications but the current techniques in MAS are not always very adequate to support applications in real open and complex environment [21].

The first usual definition of open systems concerns the information exchanges between a system and its environment. This sense is given in [12]: "Distributed Artificial Intelligence (henceforth called DAI) deals with issues of large-scale open systems (i.e. systems which are always subject to unanticipated outcomes in their operation and which can receive new information from outside themselves at any time)". Another sense, developed with multi-agent systems [4], is about functional change of a system faced up to inputs or outputs of components (agents). Open in the sense that it is impossible to know at design time exactly what components the system will be comprised of and how these components will be used to interact with one another [20]. Therefore, we have to deal with the design of this kind of systems. The AMAS (Adaptive Multi-Agent System) theory provides one solution to design them.

Whereas humans can make quite good mechanistic or mathematical models of biological systems at their level of functioning, they are not able to integrate the complexity of the dynamic which occurs during the self-structuring ontogeny of a biological system [15]. Similarly, designing open computational systems is of the same degree of complexity, and thus designers face a very difficult problem leading to misconceptions. Commonly speaking, this refers to the broad use of complexity as an intuitive notion to speak about intractable systems. Many definitions are found in the literature [6], but our purpose concerns only its consequence during the design phase of artificial systems. Because complex systems are notoriously difficult to study and there are many problems with predicting the emergent behavior of large numbers of interacting entities, designers could be usefully helped by observing incompletely specified components interacting in a simplified environment.

Considering these facts, a solution we propose in this paper might be to design Open Computational Systems (OCSs) as biological systems. We claim that OCS can't be designed like usual systems because of the inability to define during this phase what the system will be in the future and what function is will have to achieve: in short, similar to living systems any form of finalism is irrelevant for an OCS. Thus, OCSs must be supported by theories of emergence (not the focus of the paper, see for example [8]) and we propose here to derive benefit of this for a new design approach. By making them evolve at a large scale and by elaborating theories according to their "living" behaviors, systems could be designed at "run-time": we call this *Living Design*.

Because this notion of *Living Design* is the paper main topic and it is centered on the AMAS theory, the section 2 briefly expounds this theory. In a second time, to illustrate this, a sample design is discussed in section 3 about a classical problem: ant foraging [5]. By analyzing this example, the need of a new approach to design OCSs emerges and is expounded in section 4. Beside, the fast living prototyping solution is proposed as a solution to concretize the notion of *Living Design* in section 5. Finally, the notion of *Coached Living Design* is projected as a long-term perspective and the logical consequence of the *Living Design*.

2. A Theory for OCSs

In the AMAS theory [7], a multi-agent system is a system composed of autonomous entities interacting in a common environment. But the MAS itself also an environment and it has to reach a behavioral or a functional adequacy in this environment. For example, in a simulation, reaching a behavioral adequacy is to reproduce the behavior of the simulated entity; a functional adequacy is to perform the right task, the task for which the system had been built. We are specifically interested in Adaptive MAS which are classically defined by the fact that they are able to change their behavior to react to the evolution of their environment. Such a system has the following characteristics:

- the system is evolving in an environment with which it interacts,
- the system is constituted by parts : the agents,
- each agent has its own function to compute.

Moreover, our approach induces two other characteristics:

- the global function of the system is not implemented in the agents and there is no global control,
- the development is a bottom-up development : agents are defined first.

The AMAS theory is an example of collaborative emergence, in which the goal is not to obtain a given end state but a never ending adaptation process because the systems are plunged in a dynamic environment. We think that it is an important class of MAS simulating hard problems and having specific characteristics (dynamic environment, non termination, huge space search).

This theory has given many interesting results in domains like flood forecast, electronic commerce, training in information retrieval, mechanism design, telecommunication network management (for further information see also http://www.irit.fr/SMAC)

2.1. Openness and Self-Organization

In our vision, the important notion is the collective; the AMAS theory must therefore lead to a coherent collective activity that realizes the right task. We name this property "functional adequacy" and we proved the following theorem [10]: "For any functionally adequate system, there is at least a cooperative interior medium system which fulfills an equivalent function in the same environment". Therefore, we focused on the design of cooperative interior medium systems in which agents are in cooperative interactions.

The specificity of our AMAS theory resides in the fact that we do not code the global function of the system within the agents. Due to the agents' self-organization ability, the system is able to adapt itself and realize a function that it is not coded in the agent, i.e. emerging from the interactions between components. If the organization between the agents changes, the function which is realized by the collective changes. Each agent possesses the ability of self-organization i.e. the capacity to locally rearrange its interactions with others depending on the individual task it has to solve. Changing the interactions between agents can indeed lead to a change at the global level and this induces the modification of the global function. This capacity of self-organization at the lowest level enables to change the global function without coding this modification at the upper level of the system. Self-organization is founded on the capacity an agent possesses to be locally "cooperative", this does not mean that it is always helping the other ones or that it is altruistic but only that it is able to recognize cooperation failures called "Non Cooperative Situations" (NCS, which could be related to exceptions in classical programs) and to resolve them.

2.2. Cooperation for OCSs

The local treatment of NCS is a means to build a system that does the best it can when a difficulty is encountered. Such a difficulty is primarily due to the dynamical nature of the environment of the system, as well as to the dynamics of the interactions between agents. More precisely an agent can detect three kinds of NCS:

- when a signal perceived from its environment is not understood without ambiguity;
- when the information perceived does not induce the agent to an activity process;
- when concluding results lead to act in a useless way in the environment.

Briefly, in order to design an AMAS, designers have to design agents by giving them a cooperative behavior – see example in the next section.

3. Example: Designing an Anthill

Natural selection has acted upon ant species to produce a large range of foraging strategies [13][17]. At one extreme one can find species that forage almost exclusively solitarily. At the other extreme one can find species such as "army-ants" in which almost blind workers forage exclusively collectively by traveling on a network of chemical trunk-trails radiating from their nest [11].

Thus, ant foraging seems to be a good domain for testing the efficiency of the AMAS theory. In order to do that, suppose we want to develop a simulation platform and, as a software engineer, we currently specified entities: the nest and an ant class. At this stage, the designer has embedded ants with

Figure 1. Two examples of non co-operative situations ants can detect during exploration: at the left the uselessness situation and at the right the concurrency situation.

classical capabilities such as explore, pick up food, etc… Living design simply allows observing, on the platform, ants having a random behavior in the environment because they are incompletely specified. The designer is then an observer of the running system. Therefore, he may find new behaviors to add to ants.
The two following sections show the interest of the living design approach by examples (Figure 1).

3.1. Avoiding Uselessness During Exploration

Even if the platform is not completely designed, ants are able to explore the environment in order to find resources during the living design. The designer can observe that two ants meet (see figure 1) and he has not yet taken into account this situation. The basic behavior would be to avoid each other and continue exploration in the same direction (from evidence this is useless). Therefore, the designer is facing an interesting case for the AMAS theory: the co-operation reasoning implies that they must deviate to explore unexplored space. *Living Design* enabled the designer to see a situation he may not have thought of and thus specify more precisely the system he is building.

3.2. Avoiding Concurrency During Exploration

Later, the designer adds resources in the environment since the resource class has been defined. A new unforeseen situation occurs when an ant perceives two patches where one is currently exploited by colleagues. The designer has to take into account this situation by adding an adequate behavior to ants. Thus, according to the AMAS theory, this ant must preferably exploit the patch where concurrency is avoided (i.e. where there are less partners).

3.3. Contribution of Living Design

Some years ago, we developed a platform for ant foraging has been developed in order to test different behaviors. We have shown that co-operative foraging based on the AMAS theory is at least as good as that of natural ants [18]. In order to do that we have slightly modified the natural behavior in adding the following rules:

1. An AMAS ant preferably explore a part of the environment which is not currently occupied by other ants (section 4.2.2).
2. When an AMAS ant sees two patches of food simultaneously, it chooses to exploit the patch where the concurrence with other ants is the lower (section 4.2.1)
3. When an AMAS ant comes back to the nest, it lays pheromone only if there is still some food left on the patch just exploited
4. The amount of pheromone dropped by an AMAS ant is a function of the total number of food items encountered while returning to the nest.
5. The recruitment of nestmates depends on the amount of pheromone dropped, not on the amount of food brought to the nest.

We have really spent a lot of time to find these five rules during development, and no doubt that *Living Design* would have greatly improved this activity, by helping us to find these rules.

The emergence isn't in any way located in the ants but in the efficiency of the collective behaviour. Thus, living design is relevant for components (here individual ants) which are fully specified during the design phase without any knowledge of an evaluation function related to the global anthill performance. Generally speaking, living design uses the basic property of an OCS constituted of autonomous components as a help for their design.

4. The Need to Shift Artificial System Development Processes to Living Phases

In this section, a new approach to artificial system designing is discussed over biologists' experiences [15]. Beside, former artificial system design methods are confirmed not to be sufficient and adequate to apprehend open systems modeling.

4.1. The Gap between Present Methods and Required Methods

Today trend to model and specify software is to use the object paradigm and associated notation and methods. Considering the agent concept as a way to design artificial systems as biological systems, we can admit yet a huge gap between classical software design method and OCS design as this gap exists between agent and object notions [16]. Fundamental discrepancies are:

- There is no more control on the complete system during run-time;
- All the situations the system may encounter cannot be specified;
- The function of the system may evolve during run-time in comparison to original requirements.
- The size of the system is not constant: components may appear or disappear.

Thus, methods to design such systems must be different from existing ones. With the complexity increasing, the huge distribution and the environment dynamics of such systems, the classical software engineering methodologies are not sufficient. The specifications are not complete, the system has to evolve because it is open, and the control is distributed because agents are autonomous in the system.
However, such consideration need to provide some methodological helps, methods and associated tools.

4.2. Object-Agent Overlapping Processes Shifting

One way to apply this idea may be to use and to shift processes in terms of their abstraction concepts: object or

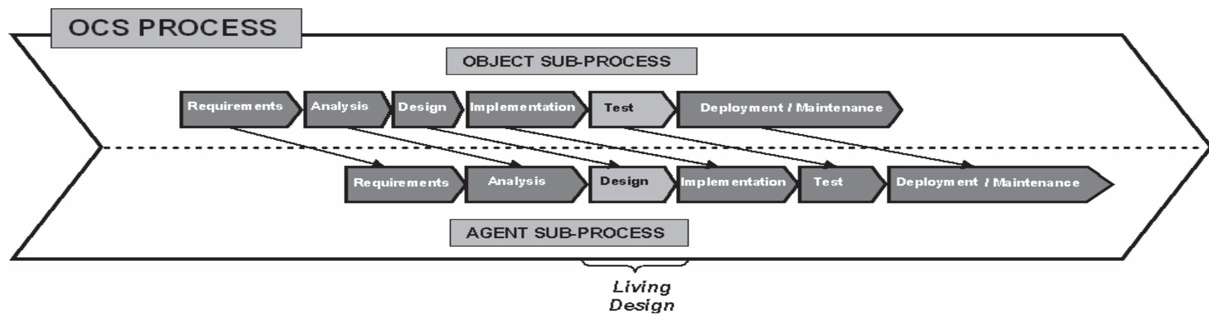

Figure 2. Object-Agent Overlapping Processes Shifting. The whole OCS Process is composed of two parallel sub-processes: the object sub-process on top and the agent sub-process on bottom. The agent process is shifted to the last phases of the object process.

agent. We call this *Object-Agent Overlapping Processes Shifting*.

4.2.1. A Two-Layered Process. At present, known methods and processes to model agent systems either manipulate new formalisms or reuse object formalisms, such as UML language (Unified Modeling Language), and associated processes, such as RUP (Rational Unified Process) [14]. For instance, TROPOS proposes a very new method with associated notations (i*) to express agent concepts, but does not manage object entities [3]. On the contrary, AOR (Agent-Object Relation) [19] or MESSAGE/UML [2] models agents by using UML formalism and by adding some agent-specific concepts to an already existing language. Beside, ADELFE [1], we are developing by now, takes part in the RUP by adding or modifying steps or activities. A first step to living design systems may be to manipulate the two paradigms, object and agent, in a two-layered process or *Object-Agent Overlapping Processes*. However, even if object or agent concepts might be expressed with the same language, it is clear that they require different views. A solution to link the two layers may be to use in parallel the two methods in which agent process is shifted to the latest phases of the object process (test, deployment and maintenance), called "living phases", during which the system is "living". To sum up, the OCS development process is now composed of two sub-processes: object and agent.

4.2.2. Shifting the Overlapping Processes to Living Phases. Figure 2 shows the two sub-processes (agent and object) which are shifted. The agent sub-process is shifted to the "living phases" of the object sub-process. In this paper, we only focus on the *Living Design*, i.e. light grey areas in the figure. The link between the two phases means that designing agents correspond to a test phase in the object sub-process. Briefly, regarding the other phases, the requirements for the agent sub-process are considered by the designer at the same time than the three first phases of the object sub-process: Requirements, Analysis and Design. This shifting between the two sub-processes appears in the ADELFE method when during the analysis work definition[1], a step, called *Verify the AMAS Adequacy*, determines whether the agent technology is necessary to develop the system [1]. At the end of the process, the Implementation, Test, Deployment and Maintenance phases of the agent sub-process are considered at the same time than Deployment and Maintenance phases of the object sub-process. Moreover, the design phase is not entirely at the responsibility of the designers as in classical software design but the system must have capabilities to react to situations non-predefined in the specifications. Therefore, systems may become more robust, more autonomous, and more complex. They may self-modify, self-repair or work in progressive degradation.

4.2.3. Living Design Pre-Requisites. *Living Design* is defined by the link between Design and Test phases of the two processes. Namely, *Living Design* means "construct agent during run-time". Therefore, the designer is like a biologist who studies the behavior of living creatures and who can modify its model according to his observations. It induces that tools exist to model and to simulate agents which lack aptitudes, skills or other components. Beside, we could have considered a kind of *Living Design*, during the deployment phases, i.e. the system is evolving and in construction in its real-life environment. Yet, it seems difficult by now for final users to accept their software as incomplete ones that cannot achieve all the possible tasks after the deployment.
In the next section, a solution to *Living Design* is suggested and an example is proposed.

5. A Solution: Fast Living Prototyping

As discussed above, to design open systems in dynamical environments, designers certainly need tools to test the components of the system during run-time. In this section we expound a solution to Living Design: fast living prototyping.

[1] A work definition is a set of activities in the Software Process Engineering Metamodel (SPEM) of the OMG.

The software OpenTool is a classical UML editor and verifier. In a near future, it will be AUML-compliant by taking into account AUML notation such as AIP protocols. This is the first reason why, the project ADELFE has chosen this tool to support the ADELFE method. The second reason for ADELFE to use OpenTool is its simulation functionality. This latter allows testing behaviors of classes by simulating state-machines from an initial configuration describe by a collaboration diagram.

ADELFE proposes in the fast prototyping activity to test interactively agents' behaviors by simulating their state-machines [1]. For example, it simulates interactions between agents by simulating concurrent state-machines corresponding to protocols. During this activity, designers may find agents lack skills, aptitudes, or even non co-operative situation detection rules. For example, by simulating artificial ants from a simple collaboration diagram, designers may detect that two ants exploit the same resource item by looking at their state status. Then, by simply adding a rule to avoid this situation, he can enhance his system. Moreover, he may need to modify ant perception to perceive resource items and ants at the same time. In a nutshell, the concurrency non co-operative situation has been found.

6. One Step Further: Coached Living Design

Let us now consider that we have a particularly complex system to design. We managed to identify some high level components/agents but we are unable to fully or formally describe their behavior and their competences. Very naturally, in a classic top-down problem approach applied to multi-agent problem solving, we will go down one level and describe the agent as a multi-agent system. These agents will be smaller and simpler agents but by cooperating as an organization, they will be able to execute the function expected from the "big" agent they are composing.

But these agents may be too difficult to code. So we will reiterate the previous step and go down one level more. If you do this for some steps, depending of the initial problem, you will come to the lowest possible level: the instructions of a programming language. Thus we have maximal expressivity: anything which can be done with a computer program will be able to be realized, as long as we can find a way to make the instructions interact in the right way.

The "Living" part of the design will be the most important here because it is through it that the system will be able to become what it has to become. And that which enables this "becoming" is the ability of the system to adapt.

6.1. Emergent Programming: Description of the Idea

Let us imagine that the whole set of instructions of a programming language is designed as agents. Each instruction (like an addition, an equality test, a variable) is an autonomous agent with a competence, the ability to communicate and work with the other agents, and a social behavior. The *competence* of an instruction agent is simply what the associated instruction is meant to do. Thus, the addition agent will receive two values from some other agents, sum them up, and send the result to some third agent. The *ability to communicate* and work with the other agents allows the instruction agents to make calls between one another, make requests, forward values. The whole calculus done in a classic computer program can thus take place. And the *social behavior* can be said to be the way the agents work together, which protocols they have to follow. For example, depending of its competence, an agent will have only a specific number of acquaintances with which it is working at a given time. And some synchronization will have to take place.

6.2. Self-organizing capability and openness of the system

So we have agents which are able to simulate a classic computer program. Now, since we want to build OCS, the agents have to self-organize so as to find organizations which are adequate when confronted to the environment. And the openness means that agents can be introduced in the system and some others disappear.

We want the system to be able to adapt to the environment and learn the right behavior. So we need to have agents able to produce this capability to adapt. Since instruction agents are very simple, they can't change their competences. But since what the system does depends from what agents are in the system an how they are working together (the topology of the organization), we only have to give the agents the ability to change their place in the organization, and let them introduce or suppress other agents when needed.

6.3. Coached Living Design

We can begin to build these agents in a very classical way and then confront the system to the environment in the Living Design phase we described earlier. Thus we can indeed fine tune the agents to make them work together and self-organize. But how can we obtain the right system at the end (meaning a system behaving adequately)?

Indeed, we cannot code the global goal into the instruction agents. As we said, if we could do that, we could write the program directly. But since the agents have the ability to self-organize, we can influence this self-organization toward the desired goal, like a coach of a team encouraging or reprimanding it. That's why we call it *Coached Living Design*.

In fact, during this phase, the designer is observing and judging the system while it is really *living*, i.e. interacting with the environment and self-organizing. But whereas in the Living Design the designer then modifies some agents to behave accordingly to what he wants, here he will have the

means to give specific feedbacks to the whole system. Of course the agents will have to be able to take into account theses feedbacks and use their self-organizing capabilities to respond to these feedbacks.

6.4 Living Design vs. Coached Living Design

Coached Living Design is necessary to guide the adaptation process of components when they are coded by the way of emergent programming. Nevertheless, it could even be relevant for higher levels of the overall system when the designer is unable to specify the organization and the social behavior of coarse-grained components which he considers already sufficiently specified.

So, to resume, the *Living Design* phase can be split into two parts:

1. The specification of the function of a component when the designer knows that an unexpected event occurring during simulation is due to the temporary lack of some parts of code.
2. The specification of the social behavior of a component when the designer knows that the unexpected event encountered during simulation is not relevant for the system function.

And the *Coached Living Design* regroups these two parts in one and goes up a level:

The "*Coaching"* of the whole system when the designer knows that an unexpected event occurred during simulation (the system has not the adequate behavior) but he doesn't want, for some reason, to modify the agents, or he may be unable to identify the agent which poses problem. In this case the designer specifies only the desired global behavior that the system should have, and the system will adapt itself to reach it.

This approach could be useful for any component sublevel, but we showed that it is essential for the Emergent Programming approach.

7. Conclusion

We have shown the need to define new approaches for designing OCSs. We propose *Living Design* as a way to take into account openness and dynamics as soon as their design phase. The fast prototyping planned in the ADELFE methodology for designing AMAS systems is the first step to a living design application.

Coaching, the further step is also involved in *Living Design* when the designer cannot further specify the system and its components but he knows the right behavior he wants the system to achieve.

From our experiments in the development of adaptive systems, we think that *Living Design* and coached living design will improve greatly their reliability and efficiency. This assertion cannot today be proved, before using and testing this approach by developing related tools in many applications.

8. References

[1] Bernon C., Gleizes M-P., Peyruqueou S., Picard G. – ADELFE, a Methodology for Adaptive Multi-Agent Systems Engineering – *In Third International Workshop on Engineering Societies in the Agents World (ESAW-2002)*, Madrid, 16-17 September 2002.

[2] Caire G., Leal F., Chainho P., Evans R., Garijo F., Gomez J., Pavon G., Kearney P., Stark J. & Massonet P. – Agent Oriented Analysis using MESSAGE/UML – *In Agent-Oriented Softaware Engineering (AOSE'01)*, 2001.

[3] Castro J., Kolp M. & Mylopoulos J. – A Requirements-driven Development Methodology – *In Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, Stafford, UK, June, 2001.

[4] Costa A.C.R., Demazeau Y. – Toward a formal model of multi-agent systems with dynamic organizations – *Proceedings of the Second International Conference on Multi-Agent Systems,* AAAI Press/ The MIT Press, Kyoto, 1996.

[5] Drogoul A., Corbara B. et Fresneau D. – MANTA : New experimental results on the emergence of (artificial) ant societies – *In Artificial Societies: the computer simulation of social life*, Nigel Gilbert & R. Conte (Eds), UCL Press, London, 1995.

[6] Edmonds, B. (1999) – What is Complexity: the philosophy of Complexity per se with application to some examples in evolution – In *F. Heylighen & D. Aerts (eds.): The Evolution of Complexity*, Kluwer, Dordrecht, 1-18, 1999.

[7] Ferber J. – Les systèmes multi-agents : Vers une intelligence collective – InterEditions, ISBN-2-7296-0572-X, 1995.

[8] Forrest, S., – Emergent computation: Self-organizing, Collective, and cooperative phenomena in Natural and Artificial Computing networks – , *Special issue of Physica D* ,MIT Press / North-Holland, 1991.

[9] Fredriksson M., Gustavsson R. and Ricci A. – Sustainable Coordination – *In Klusch, M., Bergamaschi, S., Edwards, P., Petta, P. (eds.) Intelligent Information Agents: The AgentLink Perspective.* Lecture Notes in Artificial Intelligence, vol. 2586. Springer Verlag, 2003.

[10] Gleizes M-P., Camps V., Glize P. – A Theory of emergent computation based on cooperative self-organization for adaptive artificial systems – *Fourth European Congress of Systems Science*, Valencia, 1999.

[11] Gotwald W. H., Jr. – *Army ants: the biology of social predation* – Ithaca, New York: Cornell University Press, 1995.

[12] Hewitt C. – Open information systems semantics for Distributed Artificial Intelligence – *Special volume foundations of Distributed Artificial Intelligence*, Elsevier Science Publishers B.V. Vol.47, 1991.

[13] Hölldobler, B. & E. O. Wilson – *The Ants* – The Belknap Press of Harvard University Press: Cambridge, Mass., 1990.

[14] Jacobson I., Booch G. and Rumbaugh J. – The Unified Software Development Process: the complete guide to the Unified Process from original designers – Addison-Wesley, ISBN-0-201-57169-2, 1999.

[15] Maturana H.R., & Varela F.J., *The tree of knowledge*, Addison Wesley, 1994.

[16] Odell J. - *Agents and objects: How do they differ?* - Working Paper V2.2, 2000.

[17] Oster G.F., and E.O. Wilson – *Caste and ecology in the social insects* – Princeton University Press, 1978.

[18] Topin X., Fourcassié V., Gleizes M-P., Théraulaz G., Régis C., Glize P. – Theories and experiments on emergent behaviour : From natural to artificial systems and back – *In Proceedings on European Conference on Cognitive Science*, Siena, 1999.

[19] Wagner Gerd – Agent-Oriented Analysis and Design of Organizational Information System – *In Proceedings of the Fourth IEEE International Baltic Workshop on Databases and Information Systems*, Vilnius (Lithuania), May 2000.

[20] Wooldridge M. & Ciancarini Paolo – Agent-Oriented Software Engineering: The State of the Art – *In P. Ciancarini and M. Wooldridge (eds.) Proceedings of Agent-Oriented Software Engineering (AOSE 2000)*, Springer-Verlag Lecture Notes in AI Volume 1957, 2001.

[21] Zambonelli F. and Van Dyke Parunak H. – Signs of a Revolution in Computer Science and Software Engineering – *In Proceedings of Engineering Societies in Agent World (ESAW'02)*, Madrid, 2002.