

Agents are not part of the problem, agents can solve the problem

Danny Weyns, Alexander Helleboogh, Elke Steegmans, Tom De Wolf,
Koen Mertens, Nelis Boucké and Tom Holvoet

AgentWise, DistriNet, Department of Computer Science,
K.U.Leuven, B-3001 Heverlee, Belgium

{danny.weyns, alexander.helleboogh, elke.steegmans, tom.dewolf,
koenraad.mertens, nelis.boucke, tom.holvoet}@cs.kuleuven.ac.be

Abstract. In this paper, we discuss the position of multi-agent systems (MASs) in the software development process. Basically, MASs provide an approach for solving software problems by decomposing a system into a number of autonomous entities, embedded in an environment, which cooperate in order to achieve the functional and non-functional requirements of the system. As such, MASs are in essence a family of *software architectures* and hence enter the software development picture in the design phase.

Coverage and abstraction are identified as two important dimensions of the architectural design space for MASs. Based on this, we outline a good practice for architectural design with MAS. Given this perspective, we conclude with a critical reflection on state-of-the-art agent-oriented methodologies.

1 Introduction and Position Statement

In [25], M. Wooldridge and N. Jennings agree with O. Etzioni [6] that *Agents are 99% computer science and 1% AI* and conclude that developers should exploit conventional software technologies and techniques wherever possible to engineer the conventional 99%. The trend in agent-oriented software engineering however is to devise new entire methodologies, neglecting or even ignoring existing practice and research in software engineering. This evolution suggests that multi-agent systems (MASs) are radically different ways of developing systems, unrelated to existing practice and research in engineering complex software. This image of MASs could well be one important reason why MASs are not widely adopted yet. We are strongly convinced that the research community of MASs would benefit a great deal from allocating a correct role for MASs within mainstream software engineering, rather than positioning MASs as a radically new paradigm for software development.

In essence, what is the purpose of using MASs? MASs basically provide a approach for solving software problems by decomposing a system into a number of autonomous entities, embedded in an environment, which cooperate in order to achieve the functional and non-functional requirements of the system. As such, MASs are in essence a family of *software architectures*, which play a prominent role in the software development process. Based on a problem analysis that should not be biased by any solution

strategy, the designer may or may not choose to build a solution based on MASs. Requirements related to adaptability, distribution, openness of the system may be arguments for the designer to pick a multi-agent system software architecture. This architecture then is the backbone for the further development of the software.

This paper is structured as follows: section 2 elaborates on mainstream software development practice. In section 3, we discuss the position of MAS in the software development process. Section 4 reflects on state-of-the-art agent-oriented methodologies. Finally, we conclude in section 5.

2 Mainstream Software Development Practice

A methodology consists of a *modeling language* and a *software development process* [1]. A modeling language is a visual syntax that can be used to construct models. The most well-known modeling language is the Unified Modeling Language (UML). A software development process on the other hand, defines the who, what, when and how of developing software. In general, a software development process can be split up in four phases: requirements analysis, design, implementation, and testing. These four phases can be dealt with sequentially (e.g. the *waterfall* methodology [18]), but recent methodologies make iterations over (parts of) the phases (e.g. the *unified software development process* [11] or even the *extreme programming* methodology [3]). The focus of this paper is on the development process, in particular on its requirements analysis and design phases.

To clarify the difference between requirements analysis and design, we now elaborate on both these phases.

2.1 Requirements analysis phase

During requirements analysis, the analyst investigates the problem domain and the various requirements, independent of a solution. Thus, the analyst focusses on *what* the problem *is*.

The purpose of requirements analysis is first of all to discover and reach an agreement on what the system should do. It is a process of eliciting, prioritizing and organizing the *requirements* that the different stakeholders (i.e. the types of users, engineers, salespeople, managers, end users etc.) have for the system. According to [11], requirements are capabilities and conditions to which the system must conform. Requirements can be categorized as functional or non-functional. Functional requirements are statements on what the system must do and can be captured by use cases. Non-functional requirements are statements about the constraints on the system, such as performance, reliability or security. It is important to emphasize that requirements should *only* be statements on *what* a system should do and what behavior a system should exhibit, without saying anything about *how* this functionality may be realized.

Next to the identification of requirements, another important aim of requirements analysis is to derive a *domain model*. A domain model is a visual representation of all relevant concepts or real-world entities in a particular domain of interest [16, 8]. Next to the concepts, it shows attributes of the concepts and associations between them.

A domain model is *always* described using the terminology of the problem domain, i.e. the abstractions used in the domain model should form part of the vocabulary of the problem domain.

Summarizing, during requirements analysis, the analyst investigates the problem, (carefully) omitting decisions with respect to the solution. This (1) is a pre-requisite for an objective choice for the most suitable architectural solution, and (2) ensures that the requirements and the domain model are concise and simple statements of the system's behavior and structure independent of a particular solution. To quote Gruia-Catalin Roman at SELMAS 2004: "The analysis of the problem must last for a hundred years" [21].

2.2 Design phase

During the design phase the designer creates a solution to the problem driven by the outcome of the requirements analysis phase. Thus, the designer focusses on *how* the problem can be solved.

It is common practice to make a distinction between two design levels in the design phase: architectural design and detailed design respectively.

Architectural design During architectural design, an architecture is constructed that has to satisfy both functional and non-functional requirements. Architecture is high-level design; it is about making decisions about how the system will be built. Bass, Clements, and Kazman [2] define a software architecture as follows:

The *software architecture* of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

We further clarify this definition:

- *Structures*. An architecture is defined by one or more structures. Each structure is an abstraction of the system from the viewpoint of one or more stakeholders. A structure consists out of a set of elements and their relationships.
- *Software elements*. Software elements are the basic building blocks of a structure. Each structure uses specific kinds of elements, for example process, module, component.
- *Externally visible properties*. Externally visible properties of an element represent the assumptions other elements can make about that element, such as provided services, performance characteristics and shared resource usage. This implies that an architecture omits the internals of each element and only considers information about how it uses, is used by, relates to, or interacts with other elements.
- *Relationships*. Elements of a structure are related to other elements by means of relationships. There are different kinds of relationships such as "a module uses another module" or "a process synchronizes with another process."

Common Practice. Building an architecture involves designing different structures of the system. The process of designing a specific structure of the architecture is an iterative process [2], where the architect decomposes the system into elements and relationships in a top-down fashion. First the system as a whole is decomposed, then each element identified at that level is further decomposed, and so on, until a convenient level is reached to start detailed design.

Building an architecture is not done from scratch. Over the years, a number of *architectural styles* [22] have been developed that express patterns of structural organization and that can be used by the designer. In [2], an architectural style is defined as a description of elements and relation types together with a set of constraints on how they may be used. Examples are layered systems, pipe-and-filter and blackboards.

Importance. As stated in [2, 22], “it is very important for a system to have a good architecture first, before completing the design.” Architectural decisions have the most far-reaching effects and are the hardest to change later. Therefore, designing an architecture should be well-considered since it establishes the main structure of the solution.

Detailed design During detailed design, a model of the system that can actually be implemented, is produced. Therefore, detailed descriptions and diagrams of the inner workings of all elements in the architecture are necessary. These descriptions and diagrams indicate which software artifacts (classes, methods, associations, . . .) need to be implemented. For example, choosing for a specific data structure that will be encapsulated in a particular element of the architecture, is a decision made during detailed design and not an architectural decision.

3 Software development with MAS

According to M. Wooldridge and N. Jennings in [25], developers should exploit conventional software technologies and techniques wherever possible to engineer their MASs. This rises the question what the position of MAS is with respect to the mainstream software engineering practice.

In essence, MASs provide an approach for decomposing a system into a number of autonomous entities, embedded in an environment, which cooperate in order to achieve the functional and non-functional requirements of the system. As such, the primary contribution of research and practice in MASs is proposing one particular, yet large family of *ways to solve problems*. Because a MAS is in essence an approach to solve problems, it should enter the software development picture in the (architectural) design phase. Based on an objective analysis of the problem (i.e. not biased by any solution strategy), the designer may or may not choose for building a solution based on MASs. Requirements related to adaptability, distribution, openness of the system may be arguments for the designer to pick MASs as a solution strategy. The designer then defines a complete software architecture *explicitly* in terms of MAS related concepts such agents, an environment, collaborations, and so on, and further refines it towards implementation.

Obviously, MAS researchers are convinced that MASs offer important advantages for building software for various application domains in distributed problem solving, collective robotics, agent-based simulations, and so on. We are strongly convinced that the MAS research community would benefit a great deal from allocating a correct place for MASs in mainstream software engineering, rather than positioning MASs as a radically new paradigm for software development. This does not mean that MASs should lose their specific properties and research tracks. It does mean that MASs deserve to be considered as one valuable (family of) way(s) to solve problems in a large spectrum possible ways to solve problems.

The insights on the positioning of MASs with respect to software development partly arose from our cooperation with an industrial partner, an expert in automating warehouse systems using automated guided vehicles (AGVs). The requirements analysis for their software describes the customer's functional requirements, i.e. the jobs the warehouse system should take care of, and non-functional requirements, e.g. on overall performance, robustness and scalability. Today, the design of their software systems is based on an architecture which is similar for different customers. This architecture is a centralistic setting, using one central planner which controls all AGVs. In a joint project with our team, we envisage to develop a decentralized system using a particular type of MASs, situated MAS, as these promise to be scalable, and to be resilient to failure or changes in the physical environment. In this project, we should not and will not change the analysis of the problem. We mainly aim to investigate the possible advantages of applying another way to solve the problem of warehouse management, i.e. by using a different architecture.

In the remainder of this section, we focus on architectural design. First, we discuss architecture in the context of MASs. Second, we propose a good practice for architectural design with MASs.

3.1 Architecture and MASs

Architecture in the context of MASs is typically associated with agent architectures. A whole set of agent architectures has been proposed over time. Well-known examples are the BDI agent architecture and the subsumption architecture. P. Maes [15] defines agent architecture as:

[An agent architecture] specifies how ... the agent can be decomposed into ... a set of component modules and how these modules should be made to interact. The total set of modules and their interactions has to provide an answer to the question of how the sensor data and the current internal state of the agent determine the actions ... and future internal state of the agent.

More recently, J. Ferber [7] defines agent architecture as:

An agent's architecture characterizes its internal structure, that is, the principle of organization which subtends the arrangement of its various components.

However, a MAS does not only consist of agents. In the MASs research community, there is a growing awareness that apart from the agents, a lot of other entities in MAS

are essential and have to be dealt with explicitly. Examples are the explicit environment in which the agents are situated or dynamic objects such as evaporating pheromones used for agent coordination. The fact that a MAS comprises more than just agents, imposes architectural challenges that go beyond the agents' internal architecture. Since architecture in MAS has not yet been addressed to its full extent, a lot of confusion and misunderstanding exist about what comprises architecture for MASs.

A first problem is the lack of *coverage* in architectures for MASs. We define coverage as the degree to which an architectural description takes into account all constituent parts of the system. Thus, a lack of coverage means that only a subset of all essential parts of a MAS are considered for architectural description, neglecting other important ones. This is a problem because these neglected architectural parts are extremely difficult to tackle in later design stages. By analogy, you do not design a house and build in the bathroom afterwards. The most striking example of this is the way the environment is often dealt with [23].

The second problem is that architectures generally only provide a high-level problem decomposition. Architectural descriptions are typically limited to a logical description in terms of interacting agents. However, architectures should come down to an abstraction level appropriate for detailed design. This is necessary to bridge the gap between the architectural description of the MAS and its implementation [19].

3.2 Good practice for architectural design with MAS

In this section, we outline an approach to address the problems with respect to architecture for MAS as described in the previous section. By studying the problems, we distinguish between two important dimensions that need to be considered when developing architecture for MASs: coverage and abstraction.

In figure 1, we focus on coverage and abstraction as orthogonal dimensions of the architectural design space. Three different architectural descriptions are depicted. *A* represents an architecture which is characterized by a high level of abstraction on the one hand, and a wide-ranging coverage on the other hand. An example is an architecture in which an environment, a number of agents, their responsibilities and inter-relationships are described as a logical decomposition of the problem. *B* is characterized by a lower level of abstraction and a limited coverage. An example is an architectural description of a tuple-space and its interface as coordination medium. *C* finally represents an architecture with a high level of abstraction and an intermediate coverage.

We now address the question: "What is a good practice in the architectural design for MASs with respect to the two-dimensional design space we focus on?"

Our outline for good practice is based on the following elements:

1. First mainstream software development principle: it is very important for a system to have a good architecture first, before completing the design [2, 22].
2. Second mainstream software development principle: refine your architecture to an abstraction level appropriate for detailed design [2].
3. Important dimensions of the design space: coverage and abstraction.

Applying the first software development principle to the first dimension, coverage, imposes the need for a full coverage by the architecture from the beginning of the design

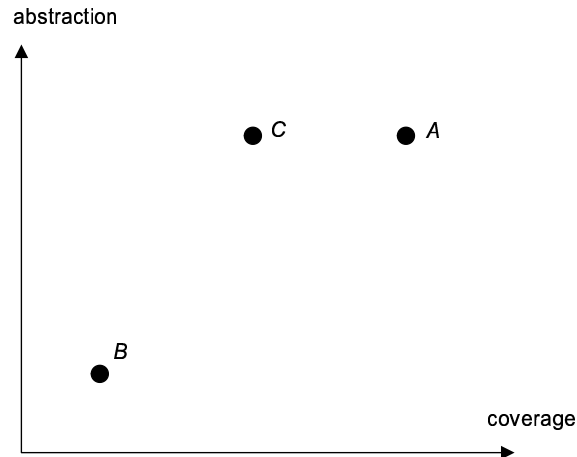


Fig. 1. Architectural design space in terms of the dimensions coverage and abstraction

phase. This is articulated in the first law of our good practice for architectural design with MAS:

Law 1: *Get the full architecture of your MAS straight from the beginning.*

Figure 2 illustrates our outline for good practice of architectural design with MAS. The first law is reflected by *D*, depicting an architecture with wide-range coverage which has to be assembled in early design.

Analogously, applying the second software development principle to the second dimension, abstraction, imposes that architectures for MASs have to be refined down to an abstraction level appropriate for detailed design. This is articulated by the second law of our good practice for architectural design with MAS:

Law 2: *Refine the architecture of your MAS towards implementation.*

In figure 2, the second law is reflected by the evolution of *D* towards *E*, which represents a process of architectural refinement. For example, whereas *D* is described in terms of agents as logical entities able to perceive and perform actions in an environment, *E* is described at a convenient level of abstraction to start detailed design, where the agents interact with a tuple-space via a well defined interface, and their internals are structured as a set of interconnected components.

3.3 Discussion

This section raises some important issues and questions with respect to architectural design for MASs. First, architecture has to be dealt with explicitly. What are possible motivations that could convince designers to choose for MASs as a solution? What are

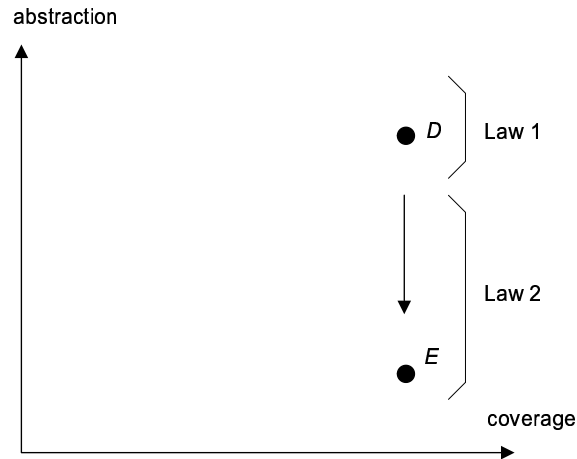


Fig. 2. Good practice for architectural design

possible contra-motivations? Second, architecture of a MAS comprises more than the architecture of its agents. What are possible architectures for an explicit environment? Which other parts of a MAS have to be incorporated in an architectural description?

4 A reflection on state-of-the-art agent-oriented methodologies

In this section we reflect on a number of the state-of-the-art agent-oriented methodologies. We focus on two issues: the scope of the methodology with respect to the software development process and the position of architecture in that methodology.

Gaia v.2 Gaia v.2 [27, 5] is an extension of the Gaia methodology [26]. According to [27] Gaia v.2 is “suitable for the development of open MASs in complex environments.”

Scope. Gaia v.2 covers the analysis and the design phase of the software development process: (1) in the *analysis phase* the requirements are organized into an environmental model, a preliminary role and interaction model, and a set of organizational rules, for each of the (sub-)organizations composing the overall system; (2) in the *architectural design phase* the specifications of the analysis phase are used to structure the MAS organization, and to complete the preliminary role and interaction models; (3) in *detailed design phase* the agent model and the services model are identified, which act as guidelines for implementation of agents and their activities.

A requirement of the Gaia methodology is that the analysis phase yields a preliminary role model. However, the goal of problem analysis is to deliver a problem description that is not biased towards particular ways to solve it. A strong focus on roles in the problem analysis could endanger a free and objective choice for the most suitable solution during the design phase.

Architecture. In [27] it is stated that “the organizational metaphor of Gaia v.2 ... leads implicitly to a general architectural characterization of the MAS.” However, according to law 1 of our good practice in architectural design, the system’s architecture should be *explicited* straight in the beginning of design.

MaSE In [24], it is stated that MaSE “supports the complete software development life cycle from problem description to realization and it is an environment for analyzing, designing and developing heterogeneous MASs”.

Scope. The *analysis phase* of MaSE consists of the following sub-phases: capturing goals, applying use cases and refining roles. In the *design phase* the following sub-phases are distinguished: creating agent classes, constructing conversations, assembling agent classes and system design.

Similar to Gaia v.2, the problem description in MaSE is biased towards particular ways to solve it, because roles (and also goals) are created in the analysis phase.

Architecture. MaSE supports architectural design by the selection of agent types and the definition of their interactions. As such, the coverage of architectural design in MaSE is limited to purely communicative agent systems. Furthermore, MaSE does a fairly poor job in dealing with agent architectures. The internal agent architectures, which have a severe impact on the overall design of the system, are unsupported and left to the designer. The general applicability of the provided component framework [20] is at least debatable.

Prometheus In [17], Prometheus is describes as “a methodology for the design of MASs. The methodology specifically supports the use of BDI-like agents.”

Scope. The Prometheus methodology consists of three phases: (1) the *system specification phase* focusses on identifying the basic functionalities of the system, together with how the agent system is going to interact with the environment; (2) the *architectural design phase* is used to determine the agents and how they will interact; (3) the *detailed design phase* looks at the internals of each agent and how it will accomplish its tasks within the overall system.

Although Prometheus is claimed to be a methodology for the *design* of MAS, it appears that the *system specification phase* boils down to analysis. Identifying the basic functionalities of the system investigates *what* the problem is, and as such is part of analysis.

Architecture. During architectural design, shared data objects and agents are identified, assigned functionalities and tied together by specifying their interactions. The choice for a specific BDI-like architecture of the agents is interwoven with detailed design. However, decisions about the agent architecture are part of architectural design and hence should be completed before entering the detailed design phase.

Tropos According to [10], Tropos is “an agent-oriented software development methodology based on two key ideas. First the notion of agent and the related mentalistic notions, such as goals and plans, are used in all phases of software development, from early analysis down to the actual implementation. Second, the methodology covers the

very early phases of the requirements analysis, thus allowing for a deeper understanding of the environment where the software-to-be will eventually operate.”

Scope. *Early requirements analysis* is concerned with the understanding of a problem by studying its organizational setting, mainly focussing on the intentions of the stakeholder modeled as goals. In *late requirement analysis* the system-to-be is described in its operational environment, along with relevant functions and qualities. During *architectural design* the system’s global architecture is defined in terms of interconnected subsystems. Finally, *detailed design* deals with the behavior of each component in further detail.

Tropos violates mainstream software development practice by introducing the use of agent-related mentalistic notions during requirements analysis. This severely biases the problem’s description towards a particular solution, endangering objective design choices as was the case with Gaia v.2 and MaSE.

Architecture. In Tropos, architectural design is based on a set of organizational architectural styles for MASs [10], which guide the design of the system architecture.

Summary To conclude, we list a number of general remarks with respect to agent-oriented methodologies. First, state-of-the-art agent-oriented methodologies fail to make a clear separation between *what* the problem *is* and *how* it is *solved*. As stated in section 3, biasing the analysis of what the problem is towards a particular way to solve it, should be avoided because this could endanger an objective choice for the most suitable architectural solution.

Second, in some methodologies a number of architectural decisions are made implicitly or are interwoven with detailed design. However, out of all design decisions, architectural decisions have the most far-reaching effects and are the hardest to change later. Therefore, any methodology should support all architectural issues explicitly and to their full extent in order to facilitate the construction of a well-considered architecture for a MAS.

Finally, none of the methodologies clearly lists the requirements of the *problems* for which their approach can be applied (or on the contrary: is not applicable) to build an favorable solution.

5 Conclusions

State-of-the-art agent-oriented methodologies suggest MASs to be a revolution in software development, neglecting or even ignoring existing practice and research in mainstream software engineering. In this paper, we argued that MASs are an evolution rather than a revolution. More precisely, MASs provide a particular approach to solve problems, and as such MASs enter the software development picture in architectural design. Do we have to conclude that agent-oriented methodologies are in fact design methodologies?

6 Acknowledgments

We would like to thank the members of the Networking task force at the DistriNet research group, K.U.Leuven for the valuable discussions that have contribute to the work presented in this paper. Also a word of appreciation goes to Jim Odell for his useful comments to improve this paper.

References

1. J. Arlow and I. Neustadt, *UML and the Unified Process*, The Object Technology Series, Addison Wesley, 2002.
2. L. Bass, P. Clements and R. Kazman, *Software Architecture in Practice*, SEI series in Software Engineering, Addison Wesley, 2003.
3. K. Beck, *Extreme programming explained: embrace change*, ISBN 0-201-61641-6, Addison-Wesley Longman Publishing Co., Inc., 200.
4. P. Bresciani and F. Sannicolo, *Requirements Analysis in Tropos: a self referencing example*, NetObject.Days, Erfurt, Germany, 2002.
5. L. Cernuzzi and T. Juan and L. Sterling and F. Zambonelli, *The Gaia Methodology: Basic Concepts and Extensions*. In Methodologies and Software Engineering for Agent Systems, Kluwer, 2004, to appear.
6. O. Etzioni, *Moving Up the Information Food Chain: Deploying Softbots on the World Wide Web*, 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference, 1996.
7. J. Ferber, *Multiagent Systems, An Introduction to Distributed Artificial Intelligence*, Addison Wesley, 1999.
8. M. Fowler, *Analysis patterns: reusable objects models*, ISBN 0-201-89542-0, Addison-Wesley Longman Publishing Co., Inc., 1997.
9. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns*, Addison-Wesley, 1995.
10. P. Giorgini and M. Kolp and J. Mylopoulos and M. Pistore, *The Tropos Methodology: An Overview*. In Methodologies and Software Engineering for Agent Systems, Kluwer, 2004, to appear.
11. I. Jacobson, G. Booch and J. Rumbaugh, *The unified software development process*, ISBN 0-201-57169-2, Addison-Wesley Longman Publishing Co., Inc., 1999.
12. P. Kruchten, *The Rational Unified Process: An Introduction*, Addison-Wesley, 2000.
13. C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, ISBN 0-13-092569-1, Prentice Hall, 2002.
14. M. Luck and R. Ashri and M. D'Inverno, *Agent-Based Software Development*. Artech House, 2004.
15. P. Maes, *The agent network architecture (ANA)*, SIGART Bulletin, 2(4), 1991.
16. J. Martin and J. Odell, *Object-Oriented Methods: A Foundation, UML Edition*, ISBN 0-13-905597-5, Prentice Hall, 1997.
17. L. Padgham and M. Winikoff, *Prometheus: A methodology for Developing Intelligent Agents*. Proceedings of the Third International Workshop on AgentOriented Software Engineering, at AAMAS 2002. July, 2002, Bologna, Italy.
18. W. Royce, *Managing the Development of Large Software Systems: Concepts and Techniques*, 9th International Conference on Software Engineering, Pittsburgh, PA, USA, ACM Press, 1989.

19. K. Schelfhout, T. Coninx, A. Helleboogh, T. Holvoet, E. Steegmans, and D. Weyns, *Agent Implementation Patterns*, Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies (Debenham, J. and Henderson-Sellers, B. and Jennings, N. and Odell, J., eds.), 2002.
20. D. Robinson, *A Component Based Approach To Agent*, citeseer.ist.psu.edu/632651.html
21. SELMAS 2004, Third International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, <http://www.teccomm.les.inf.puc-rio.br/selmas2004/>
22. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
23. D. Weyns, V. Parunak, F. Michel, T. Holvoet and J. Ferber, *Environments for Multi-agent Systems, State-of-the-art and Research Challenges*, Post-proceedings of First International Workshop on Environments for Multi-agent Systems, E4MAS, New York, 2004. To appear in Lecture Notes for Computer Science Series, Springer.
24. M. F. Wood and S. A. DeLoach, *An Overview of the Multiagent Systems Engineering Methodology*. Agent-Oriented Software Engineering, Volume 1957 of LNCS, Berlin: Springer, January 2001, 207-221.
25. M. Wooldridge and N. Jennings, *Software Engineering with Agents, Pitfalls and Pratfalls*, IEEE Internet Computing, 1999.
26. M. Wooldridge and N. Jennings and D. Kinny, *The Gaia Methodology for Agent-Oriented Analysis and Design*. Autonomous Agents and Multi-Agent Systems, vol.3(3), 2000.
27. F. Zambonelli, N. Jennings, M. Wooldridge, *Developing Multiagent Systems: the Gaia Methodology*. ACM Transactions on Software Engineering and Methodology, vol.12(3), 2003.