Software Engineering Approaches for Design of Multi-agent Systems

Lect. Gheorghe Cosmin SILAGHI, PhD
Babeş-Bolyai University Cluj-Napoca, gsilaghi@econ.ubbcluj.ro

This paper presents 2 software engineering approaches for design of multi-agent systems. The goal of the paper is to evaluate the usefulness of such methodologies regarding agent systems design. For each method, the methodology map is described, together with its most important features. We conclude that software engineering approaches are worth for consideration when moving the agent system into in-house development, as the most used agent development technologies are still object-oriented.
Keywords: *agent systems, design, agent-oriented software engineering.*

1 Introduction

Five outgoing trends have marked the history of computing, during the last decade: ubiquity, interconnection, intelligence, delegation and human orientation. Computer systems no longer standalone, but are networked into large distributed systems. Internet is an obvious example, but networking is spreading its ever-growing tentacles. Since distributed and concurrent systems have become the norm, some researchers are putting toward theoretical models that portray computing as primarily a process of interaction. Delegation means what computers are doing for us without our intervention. Humans are giving control to computers even in safety and critical tasks. Delegation requires some sort of intelligence in the software and hardware entities that acts for us, with or without our request.

In this context, a new emergent field arose in artificial intelligence: agent systems. An agent is a computer system that is capable of independent (autonomous) action on behalf of its user or owner [Wooldridge 2002], figuring out what needs to be done to satisfy the design objectives, rather than constantly being told. Therefore, the study of how we can build multi-agent systems becomes of great interest. Design of multi-agent systems will represent the study goal of this paper.

[Silaghi 2004] presented the most important knowledge-engineering approaches for design of multi-agent systems. We concluded that such methods give a good formalization and put the agent system under construction on a sound foundation. Knowledge engineering approaches are worth for consideration for prototyping reasons, to validate and verify concepts and features of the new system. But, for the design and development of large-scale agent systems, other agent-based software engineering approaches are still needed. This paper will review software engineering approaches for multi-agent system design. We will describe and evaluate two software-engineering based methodologies: MaSE and AUML. Our efforts try to answer the question regarding which are the design methodologies that should be used when designing large-scale, commercial agent systems.

The paper develops as it follows. Section 2 will introduce some basic concepts and requirements about software engineering of multi-agent systems. Section 3 and 4 will present each methodology under study, together with a short evaluation according with some well-accepted principles. We will conclude in section 5 with our opinion about developing agent-based systems with the analyzed methodologies.

2. Agent-oriented software engineering

Software engineering is “the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software” [IEEE 1990]. It is concerned with developing large applications, covering not only the technical aspects of building software systems, but also management issues. As agent-based computing introduces novel abstractions; agent – oriented

software engineering is required to sustain them.

In the strong artificial intelligence point of view, a multi-agent system is a society of individuals (agents) that interact by exchanging knowledge and by negotiating with each other to achieve either their own interest or some global goal [Wooldridge 2002]. From the weak software engineering viewpoint, a multi-agent system is a software system made up of multiple independent and encapsulated loci of control (agents) interacting with each other in a specific application context [Singh 1994]. The software-engineering viewpoint focuses on the characteristics of agents that have impact on software development, like concurrency, interaction, loci of control. Intelligence can be seen as a peculiar form of control independence, conversation as a peculiar form of interaction. This viewpoint is more general, does not exclude the strong AI opinion. There are software systems that were not conceived as agent-based systems but can be interpreted and characterized in terms of the weak notion of multi-agent system.

Agent-oriented software engineering should deal with abstractions like agents, environment, local context etc. These abstractions will be translated in concrete entities of the software system. Methods of agent-oriented methodology should focus on realizing the properties of agent systems, as other methodologies do not give specific tools for these.

Agent-oriented methodologies could be divided in two main categories: knowledge engineering approaches and software engineering approaches.

Knowledge engineering is the process of eliciting, structuring, formalizing and operationalizing information and knowledge. The main advantage of this approach is that it provides techniques for modeling the agent's knowledge. The main drawback is the fact that it does not address software engineering criteria. Examples of knowledge engineering methodologies are GAIA [Wooldridge 2000], DESIRE [Brazier 1997], MAS-CommonKASD [Iglesias 1996] etc. We analyzed GAIA and DESIRE in [Silaghi 2004].

Software engineering approaches are focused on the object-oriented paradigm. Some authors claim that agents are active objects [Shoham 1991] and therefore, object-oriented methodologies are suitable for building agent systems. Software engineering approaches are very popular, using the same tools like the most software engineering methodologies. Therefore, these types of methods are used for building commercial agent systems, as they provide with well-known development patterns. Examples of software engineering methodologies are AUML [Odell 2000], MESSAGE/UML – EURESCOM[1] project [EURESCOM 2001], MaSE [DeLoach 1999], OPM/MAS [Sturm 2003], etc. In this paper we will detail MaSE and AUML methodologies.

3. MaSE

MaSE (Multi-agent Systems Engineering) is an attempt on how to engineer practical multi-agent system. It provides a framework and a complete lifecycle methodology for analyzing, designing and developing heterogeneous multi-agent systems. MaSE is a further abstraction of the object-oriented paradigm where agents are at an even high level of abstractions than objects. MaSE addresses only closed systems; an agent that participates in the system communication protocols encapsulates all external interfaces [Wood 2001]. The methodology does not consider dynamic systems where agents can be created, destroyed or moved during execution. Inter-agent conversations are assumed to be one-to-one, as opposed to multicast. Systems designed with MaSE are not very large; the target is 10 or less software agent classes.

3.1. MaSE methodology map

Inspired from traditional object-oriented software engineering, MaSE approaches a cascading development model. Figure 1 draws the detailed methodology map of MaSE [Wood 2001].

MaSE is a goal-based methodology. The analysis is role-directed. Roles and tasks capture required organization, action and inter-

[1] http://www.eurescom.de/public/projects/P900-series/P907/default.asp

actions. Roles are played by agent classes that capture the organization. Agent design captures the roles and tasks. Therefore, conversations capture interaction and actions are captured via methods.
Capturing goals takes the initial system specification and transforms it into a structured set of system goals, building a Goal Hierarchy diagram. The goals are structured into a form (the diagram) that can be passed on and used in the design phase. In the Goal Hierarchy diagram, goals are organized by importance.

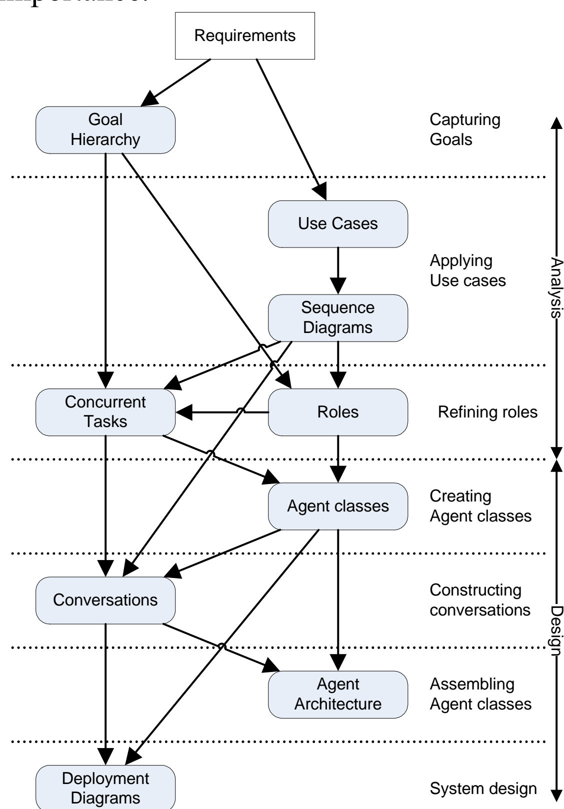

Figure 1. MaSE detailed methodology map

Use cases are drawn from the system requirements. They are narrative descriptions of a sequence of events that define desired system behavior. A sequence diagram is used to determine the minimum set of messages that must be passed between roles. Each message should have a corresponding communication path between the related roles. A communication path between roles played by separate agent classes means that a conversation must exist between the two agent classes, in order to pass the message. MaSE methodology [Wood 2001] recommends creating at least one sequence diagram from a use case. If there are several possible scenarios, multiple sequence diagrams should be created.
Roles are the building blocks used to define agent's classes and capture system goals during the design phase. A role is an abstract description of an entity's expected function and encapsulates the system goals that it has been assigned the responsibility of fulfilling. The general case of transformation of goals to roles is one-to-one: each goal maps to a role. Role definitions are captured in a traditional Role Model. In the role model, lines between roles denote possible communication paths. These paths are derived from the sequence diagram developed in the previous step. When documenting a role, the goal number is listed below the role name. Roles are denoted by rectangles, while role tasks are denoted by ovals. Lines between tasks denote communication protocols that occur between those tasks. Arrows denote the initiator / responder relationship. The concurrent tasks diagram shows the precedence of identified role tasks.
Agent classes are identified from component roles. The output of this phase is an Agent Class diagram, which depicts agent classes and the conversations between them. The boxes represent agent classes, containing the class name and the assigned roles. Lines with arrows denote conversations. The primary difference between the agent class diagram and a corresponding object diagram is the semantics of the relationship between components. In the agent class diagram, relationships define conversations, while in object diagrams lines means associations. As a design recommendation, the designer may combine multiple roles into a single agent class. It is desirable to combine two roles that share a high message traffic volume. When determining what roles to combine, size and frequency of communication are important, not only the number of communication paths.
Constructing conversations step is closely linked with the next one, assembling agents. A MaSE conversation defines a coordination protocol between two agents. A conversation consists of two communication class diagrams one for initiator, one for responder. A

communication class diagram is a pair of finite state automaton that defines the conversation states of two participant agent classes. The syntax of a transition inside the automaton follows the conventional UML notation. Conversations must support and be consistent with all sequence diagrams derived in an early analysis phase. Conversations are built by first adding all states and transitions that can be derived from the sequence diagram and from tasks. For the rest of the conversation diagram, the designed adds states and transitions necessary to convey the required messages and provide robust operation.
Assembling agent classes phase consists of building internal of agent classes. A designer may define internal components of an agent from scratch or using pre-existing components. Furthermore, components may have sub-architectures containing other components. Components are joined with inner and outer agent connectors. Inner-agent connectors define visibility between components. Outer-agent connectors define connection with external resources such as other agents, sensors, databases.
The final step of MaSE – system design, takes agent classes and creates actual agents out of them. The Deployment diagram shows the number, type and location of agents within the system. Instantiating agents from agent classes are similar with instantiating objects from classes, in object-oriented programming.
MaSE is concerned about code generation, after deploying agents on the diagram. The authors are content of the importance of this last step, finishing the design methodology, toward a running agent-based system. Further research of the MaSE authors [DeLoach 2001], conducted toward a tool that supports and helps system design with MaSE, and contains a module that allows some code generation.

3.2. MaSE evaluation

In this subsection we will evaluate MaSE methodology, according with some criteria well-accepted for agent systems. More precisely, we will check how MaSE fulfills the properties, concepts and pragmatics of agent theory.
In MaSE autonomy is expressed by the fact that the role encapsulates its functionality. Reactiveness is not expressed explicitly. There is no explicit connection between the event and the action taken. Yet, reactiveness can be expressed using the conversation state machines. Proactiveness is expressed by the role's tasks. These tasks are modeled using finite state automaton. MaSE does not mention about the social aspect of the system, except for communication.
Besides agent properties, other software-oriented features are considered. Therefore, MaSE provides a very simple set of models that enhance accessibility. MaSE supports internal verification and consistency checking of the models. However, there are still some cases where inconsistencies may occur. Regarding complexity management, there are several layers of abstraction within MaSE: agents, roles and tasks. There is no support of managing the complexity of complex tasks and roles. Modularity is supported within the agent template diagram.
MaSE is adequate for creating new software, reengineering and designing systems with reuse components and prototyping. It covers the entire lifecycle except for testing. The deliverables of MaSE are well-defined. Regarding practical implementation issues, we notice the presence of a case tool – agentTool, the fact that MaSE is not coupled with any architecture or programming language, being a generally-purpose methodology for designing multi-agent systems.

4. AUML

We should start by mentioning the fact that the authors of AUML are scientists working for Siemens research; then we could assume that AUML efforts were raised out of some commercial interest.
The Unified Modeling Language gained wide acceptance for the representation of engineering artifacts in the object-oriented software design. [Bauer 1999] sees agents as the next step beyond objects and propose extensions to UML in order to accommodate UML with the distinctive requirements of agents. Agent

UML (AUML) is the proposed language in this direction, being accepted as part of FIPA[2]-99 standard.

4.1. AUML methodology map

AUML extends UML with the following issues: a special organized agent class, the new concept of role, the new Agent Interaction Protocol Diagram. The classical diagrams of UML still need to be considered during the phases of the software product design. Table 1 describes the methodology map for AUML, considering a "waterfall model" for system implementation, with the following essential stages: requirements gathering, system analysis, system design and implementation. Pluses mean that at a stage, a kind of diagram needs to be created and consulted.

AUML introduces Agent Interaction Protocol (AIP) diagrams. AIPs are a specific class of software design patterns in that they describe problems that occur frequently in multi-agent systems and they describe the core of a reusable solution to that problem [Bauer 1999].

A definition of an AIP describes: (i) a communication pattern with an allowed sequence of messages between agents having different roles and constraints on the content of the messages, and (ii) a semantics that is consistent with the communicative acts within a communication pattern. Messages must satisfy standardized communicative (speech) acts that define the type and the content of the messages (e.g. FIPA-ACL, KQML[3]). Interaction protocols are described by the new introduced "(Agent Interaction) Protocol Diagrams".

Figure 2 depicts the protocol diagram for the FIPA English-Auction Protocol [Bauer 1999]. In English auction, the auctioneer initially proposes a price lower than the expected market price and then, gradually, raises the price. The communication starts from the auctioneer side, informing the participants that the auction has started (inform-start-of-auction). Each time a new price is announced (cfp-n message), the auctioneer waits until a given deadline to see if any participant signal its willingness to pay the proposed price (propose message). If a participant does not understand the ontology or the syntax of the cfp message, it replies with a not-understood communicative act.

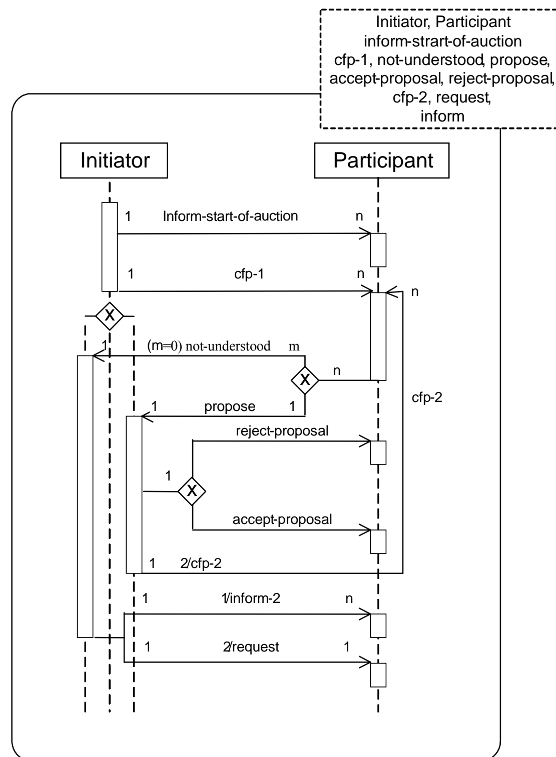

Figure 2. FIPA – English-auction protocol

Besides AIPs, AUML considers other extensions, for representing agent concepts in UML.

In UML, a role is an instance-focused term, referring to a sole realization. In AUML an agent role means a set of agents satisfying distinguished properties, interfaces, service descriptions or having a distinguished behavior. Agents can perform various roles within one interaction protocol; therefore, the implementation of an agent can satisfy different roles. A protocol can be defined at the level of concrete agent instances or for a set of agents satisfying a distinguished role or class. Such an agent is called *agent of a given role and class.*

---

[2] FIPA is a non-profit organization aimed at producing standards for the interoperation of heterogeneous software agents; http://www.fipa.org

[3] these notations represents agent-used languages for encoding agent communication

Diagram \ stage	Requirements	Analysis	Design	Implementation
Class		+	+	
Object		+	+	
Component			+	+
Deployment			+	
Sequence	+	+	+	
Collaboration	+	+	+	
Use case	+			
State chart		+	+	
Activity		+	+	
Package		+	+	
Model			+	+
Subsystem			+	+
Extension mechanism			+	+
Agent Interaction Protocol		+	+	+

Table 1. AUML methodology map

The agent lifeline in the protocol diagram defines the time period during which an agent exists. The lifeline may split up into two or more lifelines to show AND and OR parallelism decisions, corresponding to branches in the message flow. Figure 2 presented only the XOR connector type, when only one message type could be derived at a moment.
The sending of messages can be done either in parallel or as a decision between different communicative acts. Receiving different communicative acts usually results in different behavior and different answers. That means the behavior of an agent role depends on the received message. Therefore, the *thread of interaction*, i.e. the processing of the incoming messages, has to be split up into different threads. It results that the lifeline of an agent role is split and the threads of interaction define the reaction to received messages. The thread of interaction shows the period during which an agent role is performing some tasks and a reaction to an incoming message.
Sending a communicative act conveys information and entails the sender's expectation that the receiver will react according with the semantics of the communicative act. This semantic meaning of a message represents another extension for the UML concept of message.
Agent Interaction protocol diagram represent the most important extension, being defined only in AUML. Class diagrams in AUML look similar with the ones in UML, with the difference that they describe agent roles or agent classes. Class diagrams represent the knowledge structure of the agent system, with the composition relationship as a defining element.

4.2. AUML evaluation

In this subsection we will evaluate AUML according with the same principles as for MaSE. More precisely, we will check how AUML fulfills the properties, concepts and pragmatics of agent theory.
In AUML autonomy is expressed within the agent class. We may observe that class description of UML could be seen as a sufficient formalism for describing the autonomy of agents. Reactiveness and proactiveness are expressed by the set of behavioral diagrams. In AUML there is no special treatment of sociality. Regarding reactiveness and proactiveness, AUML extends UML with the introduction of the agent interaction protocol diagrams, which constitutes as templates for communicative acts.
Regarding software-oriented principles, AUML is not a language yet; there are no formal definitions. AUML states only some extensions to UML and assumes that all UML forms are adopted. Modularity is supported by UML (and by object-oriented paradigm), while complexity management is supported via packages, models, and subsystems defined in UML. AUML as a descendant of UML can use the techniques of UML for rapid prototyping. It can provide code skeleton or working applications through state charts. AUML can make advantage of existing UML tools capabilities.

AUML is adequate for creating new software, reengineering, reverse engineering, prototyping, designing systems with reuse components. All these advantages come from the UML. Regarding lifecycle coverage, RUP is probably the methodology to be used when adopting AUML because it provides a rich set of guidelines for performing the development states' activities. The deliverables of AUML and RUP are well defined.
The required knowledge of the designer is minimal. A person with object-oriented knowledge can easily move to agents. AUML is not targeted at a specific language or architecture. AUML is mainly recommended for computational-oriented applications; however it can handle knowledge-based applications as well.

5. Conclusions

This paper presents MaSE and AUML as representative methodologies for software-engineering based approaches for design of agent systems. Section 2 introduced agent-oriented software engineering as required when dealing with the new concepts of agent theory when building software systems. Section 3 and 4 entered the details of the analyzed methodologies, presenting their methodology map and a short evaluation with respect to some well-accepted criteria.
MaSE succeeds in creating a useful context and framework for building agent systems. Taking MaSE guidelines, a small team of developers can bring a system to a functional, running state. The analysis and the design can pass from the outer conceptual level to a micro inner level of the components. Specification deliverables for the internal agent representations are provided (a designer can specify sequence and state diagrams, agent interaction and composition). In order to achieve these performances, MaSE left the agent theory and agent concepts un-attained, and proposed its own substitutes. We think that this approach is worth to be considered, as MaSE succeeded in fulfilling the most important features of agent theory. However, from the theoretical point of view, MaSE does not provide functionality validation and verification.
As MaSE was build for a specific project inside US Air Force, and at the moment of publishing, other software engineering efforts were focused toward standardization of object-oriented modeling, MaSE did not become popular. It is difficult to propose in a software house a MaSE approach for a project, even if it is an agent-based one. MaSE will remain a good choice for the moment of time when agent systems will overpasses object-oriented approaches.
AUML intends to anchor agent-based system development into an object-oriented framework. Their authors observed the need of the industry to be able to reuse old development patterns even when adopting new and challenging technologies. Therefore, instead of proposing a new approach to deal with agent technology, they chose to extend an existing and wide-adopted methodology. But they provided only with some extensions to UML, in order to be able to represent the new concepts of agency, and they left unachieved the definition of the software engineering process, integrating AUML in RUP or other object-oriented software engineering methodology.
Therefore, AUML is a different kind of approach, as it focuses on extending an existing and wide-accepted approach for agents. But, it can become useless, as when starting the analysis and design for a system with object-oriented tools, a designed can ignore the agent extensions and use only object-oriented building blocks. Therefore, we think that AUML is an attempt to move closely with the practical issues regarding system design and development and it put under question the theoretical aspects of the agency, the logical and knowledge-based foundation.
Software-engineering based approaches come closer with the need of the industry for a useful formalism in order to approach large-scale agent systems development. As the most popular agent development tools (JADE, FIPA-OS, IBM-Aglets) are object-oriented, AUML constitutes the description language used to represent agent-specific concepts. Therefore, one who intends to approach building multi-agent systems for spe-

cific problems should consider software-oriented design methodologies.
We recommend a dual approach: when prototyping and in the first iterations of the agent-based system a knowledge-based approach [Silaghi 2004] or MaSE is worth for consideration. When moving the system into in-house development, considering the up-to-date development technologies, AUML is the required design language.

References

[Bauer 1999] Bernhard Bauer, "Extending UML for the Specifications of Agent Interaction Protocols", submission for the 6th Call for Proposal of FIPA and revised part of FIPA-99, response to OMG Analysis & Design Task Force UML 2.0, December 1999

[Brazier 1997] F. M. T. Brazier, B. Dunin-Keplicz, N. Jennings, J. Treur, "Desire: Modeling Multi-Agent Systems in A Compositional Formal Framework", *International Journal of Cooperative Information Systems*, vol. 6, 1997

[DeLoach 1999] Scott DeLoach, "Multi-agent System Engineering: A Methodology and Language for Designing Agent Systems", in *Proceedings of Agent Oriented Information Systems '99 (AOIS'99)*, Seattle, May 1999

[DeLoach 2001] Scott DeLoach, Mark Wood, "Developing Multi-agent Systems with agentTool", in *Intelligent Agents VII. Agent Theories Architectures and Languages, 7th International Workshop*, Boston, USA, July 2000, published in *LNCS,* Vol. 1986, Springer Verlag, Berlin, 2001

[EURESCOM 2001] EURESCOM project 907, "MESSAGE: Methodology for Engineering Systems of Software Agents – Methodology for Agent-Oriented Software Engineering", Richard Evans Ed. EURESCOM, September 2001

[Iglesias 1996] C. A. Iglesias, M. Garijo, J. C. Gonzàlez, J. R. Velasco, "A Methodological Proposal for Multi-agent Systems Development extending CommonKADS", in *Proceedings of 10th Knowledge Acquisition for Knowledge-Based Systems Workshop*, Alberta, Canada, 1996

[Odell 2000] J. Odell, H.V.D. Parunak, B. Bauer, "Extending UML for Agents" In G. Wagner, Y. Lesperance, and E. Yu editors, *Proceedings of the Agent Oriented Information Systems Workshop (AOIS) at the 17th National Conference on Artificial Intelligence*, Austin, Texas, 2000

[Shoham 1991]Y. Shoham, "AGENT0: A Simple Agent Language and its Interpreter", in *Proceedings of the 9th National Conference on Artificial Intelligence, AAAI-91*, Eindhoven, The Netherlands, 1991, published by Springer-Verlag, LNAI vol. 1038

[Silaghi 2004] Gheorghe Cosmin Silaghi, "Knowledge-engineering approaches for design of multi-agent systems", in *Proceedings of the 2nd International Workshop on Distributed Systems*, University of Suceava, December 2004

[Singh 1994] M. Singh, "Multi-agent systems: A Theoretical Framework for Intentions, Know-How, and Communications", *Lecture Notes in Artificial Intelligence*, no. 799, Springer-Verlag, Heidelberg, Berlin, 1994

[Sturm 2003] A. Sturm, D. Dori, O. Shehory, "Single-Model Method for Specifying Multi-Agent Systems", in *Proceedings of 2nd International Joint Conference on Autonomous Agents and Multi-agent Systems*, *AAMAS*, Melbourne, July 2003,

[Wood 2001] Mark Wood, Scott DeLoach, "An Overview of the Multi-agent Systems Engineering Methodology", in *Agent-Oriented Software Engineering – Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering*, Limerick, Ireland, June 2000, published in LNAI, vol. 1957, Springer Verlag, Berlin, January 2001

[Wooldridge 2000] M. Wooldridge, N.R. Jennings, D. Kinny, "The GAIA methodology for Agent-Oriented Analysis and Design", *Autonomous Agents and Multi-Agent Systems*, vol. 3/3, Kluwer Academic Publishing, 2000

[Wooldridge 2002] Michael Wooldridge, "*An Introduction to Multi Agent Systems*", John Wiley and Sons, Chichester, England, February 2002