A survey of Agent-Oriented Software Engineering

Amund Tveit*
amund.tveit@idi.ntnu.no
Norwegian University of Science and Technology

May 8, 2001

Abstract

Agent-Oriented Software Engineering is the one of the most recent contributions to the field of Software Engineering. It has several benefits compared to existing development approaches, in particular the ability to let agents represent high-level abstractions of active entities in a software system. This paper gives an overview of recent research and industrial applications of both general high-level methodologies and on more specific design methodologies for industry-strength software engineering.

Keywords: Intelligent Agents, Software Engineering, UML, Design Patterns and Components

1 Introduction

Agent-Oriented Software Engineering is being described as a new paradigm [22] for the research field of *Software Engineering*. But in order to become a new paradigm for the software industry, robust and easy-to-use methodologies and tools have to be developed.

But first, let us explain what an agent is. An agent, also called a software agent or an intelligent agent, is a piece of autonomous software, the words intelligent and agent describe some of its characteristic features. Intelligent is used because the software can have certain types of behavior (*"Intelligent behavior is the selection of actions based on knowledge"*), and the term *agent* tells something about the purpose of the software. An agent is *"one who is authorized to act for or in the place of another"* (Merriam Webster's Dictionary).

Examples of software agents

1. The animated paperclip agent in Microsoft Office
2. Computer viruses (destructive agents)
3. Artificial players or actors in computer games and simulations (e.g. Quake)
4. Trading and negotiation agents (e.g. the auction agent at EBay)
5. Web spiders (collecting data to build indexes to used by a search engine, i.e. Google)

A common classification scheme of agents is the weak and strong notion of agency [32]. In the *weak notion of agency*, agents have their own will (*autonomy*), they are able to interact with each other (*social ability*), they respond to stimulus (*reactivity*), and they take initiative (*pro-activity*). In the *strong notion of agency* the weak notions of agency are preserved, in addition agents can move around (*mobility*), they are truthful (*veracity*), they do what they're told to do (*benevolence*), and they will perform in an optimal manner to achieve goals (*rationality*).

Due to the fact that existing agents have more in common with software than with intelligence, they will be referred to as software agents or agents in this context.

1.1 Terminology

Being a relatively new research field, agent-based software engineering currently has a set of closely related terms used in research papers, I will thus try to clarify and explain the terms and their relations below.

Agent-Oriented Programming (AOP)[29, 30] is seen as an improvement and extension of Object-Oriented Programming (OOP). Since the word "Programming" is attached, it means that both concepts are close to the programming language and implementation level. The term "Agent-Oriented Programming" was introduced by Shoham in 1993 [28].

*http://www.elcomag.com/amund/

Agent-Oriented Development (AOD) [8] is an extension of Object-Oriented Development (OOD). The word "Development" is sometimes interpreted as "Programming", on the other hand it is frequently interpreted to include the full development process that covers the requirement specification and design, in addition to the programming itself.

Software Engineering with Agents [33], *Agent-Based Software Engineering* [12], *Multi-agent Systems Engineering* (MaSE) [3, 31] and *Agent-Oriented Software Engineering* (AOSE) [22, 20, 35, 15] are semantically equivalent terms, but *MaSE* refers to a particular methodology and *AOSE* seems to be the most widely used term. The difference between AOSE and AOD, is that AOSE also covers issues such as re-use and maintenance of the agent-system in addition to the development of the system itself. However, to be on the safe side, one should omit the use of the term AOD since it can easily be misinterpreted as pointed out earlier (due to the different interpretations).

The term *Agent-Based Computing* [16] can be applied to describe all issues related to agent-oriented software engineering, but it also covers issues regarding *how* and *what* agents compute.

1.2 Scope and limitations

In this paper we will present a topical overview of recent advances of methodologies for development of agent-based systems. The focus is both on general high-level methodologies and on more specific design methodologies related to software engineering. This means that specialized agent methodologies, e.g. to improve coordination, cooperation, communication and artificial intelligence in agents and agent systems, are outside the scope of this paper. Suggested readings that give good overviews of other aspects of the agent research field are presented in the work by Jennings et al. [11] and by Nwana et al. [27].

This paper is organized as follows: section 2 describes aspect of Agent-Oriented Software Engineering, section 3 gives a description of high-level methodologies, section 4 describes design methods inspired by well-known software engineering methods and standards (e.g. UML, components and design patterns), section 5 describes problems, methodologies and tools for agents in industrial context.

2 Agent-Oriented Software Engineering

The main purposes of Agent-Oriented Software Engineering are to create methodologies and tools that enables inexpensive development and maintenance of agent-based software. In addition, the software should be flexible, easy-to-use, scalable [5] and of high quality. In other words quite similar to the research issues of other branches of software engineering, e.g. object-oriented software engineering.

How are agents distinguished from objects?

Agent-oriented programming (AOP) can be seen as an extension of object-oriented programming (OOP), OOP on the other hand can be seen as a successor of structured programming [29, 30]. In OOP the main entity is the object. An object is a logical combination of data structures and their corresponding methods (functions). Objects are successfully being used as abstractions for *passive* entities (e.g. a house) in the real-world, and agents are regarded as a possible successor of objects since they can improve the abstractions of *active* entities. Agents are similar to objects, but they also support structures for representing mental components, i.e. beliefs and commitments. In addition, agents support high-level interaction (using agent-communication languages) between agents based on the "speech act" theory as opposed to ad-hoc messages frequently used between objects [22], examples of such languages are FIPA ACL and KQML [21].

Another important difference between AOP and OOP is that objects are controlled from the outside (whitebox control), as opposed to agents that have autonomous behavior which can't be directly controllable from the outside (blackbox control). In other words, agents have the right to say "no" [9]

Can agents solve all software problems?

Since this is a new and rapidly growing filed, there is a danger that researchers become *overly optimistic* regarding the abilities of agent-oriented software engineering.

Wooldridge and Jennings [7, 33] discuss the potential pitfalls of agent-oriented software engineering. They have classified pitfalls in five groups: political, conceptual, analysis and design, agent-level, and society-level pitfalls. *Political pitfalls* can occur if the concept of agents is oversold or sought applied as a *the* universal solution. *Conceptual pitfalls* may

occur if the developer forgets that agents are software, in fact multithreaded software. *Analysis and design pitfalls* may occur if the developer ignores related technology, e.g. other software engineering methodologies. *Agent-level pitfalls* may occur if the developer tries to use too much or too little artificial intelligence in the agent-system. And finally, *society-level pitfalls* can occur if the developer sees agents everywhere or applies too few agents in the agent-system

The problem with hype

Being aware of the failing promises of the closely related field of Artificial Intelligence in the 1980s, Jennings, a prominent researcher of the agent field, points out that the failure of keeping promises and becoming an offer of media hype and then "slaughter", could perfectly well happen to the field of agent research [16].

3 High-level Methodologies

This section describes methodologies that provide a top-down and iterative approach towards modeling and developing agent-based systems.

3.1 The Gaia Methodology

Wooldridge, Jennings and Kinny [10, 8] present the Gaia methodology for agent-oriented analysis and design. Gaia is a general methodology that supports both the micro-level (agent structure) and macro-level (agent society and organization structure) of agent development, it is however no "silver bullet" approach since it requires that inter-agent relationships (organization) and agent abilities are static at run-time. The motivation behind Gaia is that existing methodologies fail to represent the autonomous and problem-solving nature of agents; they also fail to model agents' ways of performing interactions and creating organizations. Using Gaia, software designers can systematically develop an implementation-ready design based on system requirements.

The first step in the Gaia *analysis* process is to find the *roles* in the system, and the second is to model *interactions* between the roles found. Roles consist of four attributes: responsibilites, permissions, activities and protocols. *Responsibilites* are of two types: *liveness properties* - the role has to add something good to the system, and *safety properties* - prevent and disallow that something bad happens to the system. *Permissions* represents what the role is allowed to do, in particular, which information it is allowed to access. *Activities* are tasks that a role performs without interacting with other roles. *Protocols* are the specific patterns of interaction, e.g. a seller role can support different auction protocols, e.g. "English auction". Gaia has formal operators and templates for representing roles and their belonging attributes, it also has schemas that can be used for the representation of interactions.

In the Gaia *design* process, the first step is to map roles into *agent types*, and then to create the right number of *agent instances* of each type. The second step is to determine the *services model* needed to fulfill a role in one or several agents, and the final step to is create the *acquaintance model* for the representation of communication between the agents.

Due to the mentioned restrictions of Gaia, it is of less value in the open and unpredictable domain of Internet applications, on the other hand it has been proven as a good approach for developing closed domain agent-systems. As a result of the domain restrictions of the Gaia method, Zambonelli, Jennings et al. [35] proposes some extensions and improvements of it with the purpose of supporting development of Internet applications.

Other sources for the discussion of micro and macro aspects of agent modeling include work by Chaib-draa [2]

3.2 The Multiagent Systems Engineering Methodology

Wood and DeLoach [3, 31] suggest the Multiagent Systems Engineering Methodology (MaSE). MaSE is similar to Gaia with respect to generality and the application domain supported, but in addition MaSE goes further regarding support for automatic code creation through the MaSE tool. The motivation behind MaSE is the current lack of proven methodology and industrial-strength toolkits for creating agent-based systems. The goal of MaSE is to lead the designer from the initial system specification to the implemented agent system. Domain restrictions of MaSE is similar to those of Gaia's, but in addition it requires that agent-interactions are one-to-one and not multicast.

The MaSE methodology are divided into seven sections (phases) in a logical pipeline. *Capturing goals*, the first phase, transforms the initial system specification into a structured hierarchy of system goals. This is done by first identifying goals based on the initial system specification's requirements, and then ordering the goals according to im-

portance in a structured and topically ordered hierarchy. *Applying Use Cases*, the second phase, creates use cases and sequence diagrams based on the initial system specification. Use cases presents the logical interaction paths between various roles in and the system itself. Sequence diagrams are used to determine the minimum number of messages that have to be passed between roles in the system. The third phase is *refining roles*, it creates roles that are responsible for the goals defined in phase one. In general each goal is represented by one role, but a set of related goals may map to one role. Together with the roles a set of tasks are created, the tasks defines how to solve goals related to the role. Tasks are defined as state diagrams. The fourth phase, *creating agent classes*, maps roles to agent classes in an agent class diagram. This diagram resemble object class diagrams, but the semantic of relationships is high-level conversation as opposed to the object class diagrams' inheritance of structure. The fifth phase, *constructing conversations*, defines a coordination protocol in the form of state diagrams that define the conversation state for interacting agents. In the sixth phase, *assembling agent classes*, the internal functionality of agent classes are created. Selected functionality is based on five different types of agent architectures: Belief-Desire-Intention (BDI), reactive, planning, knowledge based and user-defined architecture. The final phase, *system design*, create actual agent instances based on the agent classes, the final result is presented in a deployment diagram.
Visions of the future for MaSE is to provide completely automatic code generation based on the deployment diagram.

3.3 Modeling database information systems

Wagner [29, 30] suggests the Agent-Object Relationship (AOR) modeling approach in the design of information systems. AOR is inspired by the two widely applied models of databases, i.e. the Entity-Relationship (ER) meta-model and the Relational Database (RDB) model.

The purpose of the ER meta-model is to ease the transformation of relations between different types of data (entities) into an implementation-ready (database) information system design. This transformation is well-supported for *static* entities or objects, but falls short in modelling *active* entities or agents in an information system; the purpose of the AOR-model is to extend the ER-model by providing the ability to model relations between agents in addition to static entities.

In AOR, entities can be of six types: agents, events, actions, commitments, claims and objects. Commitments and claims are dualistic, commitments of one agent are seen as claims against other agents. Organizations are modeled as a group of sub-agents. Each of the sub-agents has the *right* to perform certain actions, but they are also commited to *duties* such as monitoring claims and events relevant for the agent-organization. The interpretation of duties and permissions seems to correspond with services and permissions found in the Gaia methodology [10]. An example of an agent-based database information system can be found in Magnanelli et. al [23].

4 Design Methods

This section describes methodologies that are mainly inspired by the methodologies and standards of the object-oriented software engineering field.

4.1 UML

The Universal Modeling Language (UML) is a graphical representation language originally developed to standardize the design of object classes. It has later been greatly extended with support for designing sequences, components etc., in fact all parts of an object-oriented information system design.

Yim et al. [34] suggest an architecture-centric design method for multi-agent systems. The method is based on standard extensions of UML using on the Object Constraints Language (OCL), and it supports the transformation of agent-oriented modeling problems into object-oriented modeling problems. In the transformation process, relations between agents are transformed to design patterns, these patterns are then used as relations between object classes, in contrast to the more commonly applied relation types between object classes such as inheritance. The result of this method is that designers and developers are able to use existing UML-based tools in addition to knowledge and experience from developing object-oriented systems.

Odell, Parunak and Bauer [14] suggested a three-layer representation of Agent-Interaction Protocols (AIP). AIP are defined as patterns representing both the message communication between agents, and to the corresponding constraints on the content of such messages. In contrast to Yim et al.'s UML-based architecture [34], Odell et al.'s approach requires changes of the UML visual language and not only the expressed semantics. The representation requires changes of the following UML represen-

tations: packages, templates, sequence diagrams, collaboration diagrams, activity diagrams and statecharts. In the *first layer*, the communication protocol (i.e. type of interaction) is represented in a reusable manner applying UML packages and templates. The *second layer* represents interactions (i.e. which type of agents can communicate with whom) between agents using sequence, collaboration and activity diagrams as well as statecharts. In the *third layer*, the internal agent processing (i.e. why and how the agent acts) is represented using activity diagrams and statecharts.

In "Extending UML for Agents" [13], Odell et al. suggests further extensions to UML called Agent UML (AUML) to be able to represent all aspects of agents using AUML. AUML has been submitted to the UML standardization committee as a proposal for inclusion in the forthcoming UML 2.0 [17]. According to the suggestion, UML has to include *richer role specification* that requires modification of the UML sequence diagram format. To be able to represent agents instead of operations as interface points, the UML package definition has to modified. Agents have the ability to be mobile in the sense that they can move between different agent systems autonomously. In order to represent this in UML, the deployment diagram definition has to be changed.

Bergenti and Poggi [15] suggest the application of four agent-oriented UML diagrams at the highest abstraction level of Agent-Oriented Software Engineering, namely the agent level. It is similar to Yim's approach in the sense that there are no required changes of the UML standard itself. The first is the *ontology diagram*, it is used to model the world as relations between entities using the UML static class diagram format. The second is the *architecture diagram* that is used in modeling the configuration of a multi-agent system by applying the UML deployment format. Diagram three is the *protocol diagram*, it is used to represent the language of interaction, and is based on the UML collaboration diagram format. This *protocol diagram* corresponds to Odell et al.'s [14] first layer model of the communication protocol. The fourth is the *role diagram* based on the UML class diagram, it is used to represent the functionalities each agent role has.

Parunak and Odell [9] combine existing organizational models for agents in a UML-based framework in order to model and represent social structures in UML. This work is an improvement oo the Agent UML extensions to UML.

4.2 Design Patterns

Design patterns are reoccuring patterns of programming code or components software architecture.

Aridor and Lange [1] suggest a classification scheme for design patterns in a mobile agent context. In addition they suggest patterns belonging to each the classes. The purpose is to increase re-use and quality of code and at the same time reduce the effort of development of mobile agent systems. The classification scheme has three classes: traveling, task and interaction. Patterns in the *traveling class* specify features for agents that move between various environments, e.g. the forwarding pattern that specifies how newly arrived agents can be forwarded to another host. Patterns of the *task class* specify how agents can perform tasks, e.g. the plan pattern specifies how multiple tasks can be performed on multiple hosts. Patterns of the *interaction class*, specify how agents can communicate and cooperate. An example of an interaction class pattern is the facilitator, it defines an agent that provides services for identifying and finding agents with specific capabilities.

Other approaches for design patterns for mobile agents include the approach of Rana and Biancheri [26] applying Petri Nets to model the meeting pattern of mobile agents.

Kendall et al. [6] ([19, 18]) suggest a seven-layer architecture pattern for agents, and sets of patterns belonging to each of the layers. The seven layers are: mobility, translation, collaboration, actions, reasoning, beliefs and sensory. The three lowest layers have patterns that select the mental model of the agent, e.g. if the agent is to respond to stimulus the reactive agent pattern should be selected, if it is to interact with human users the interface agent pattern should be selected. Selecting patterns as a methodology for agent development is being justified by referring to the previous successes of applying patterns in traditional software technology.

Compared to the previously mentioned pattern classification scheme in the work by Aridor and Lange, the layered architecture has a similar logical grouping of patterns. The mobility layer together with the translation layer corresponds to the class of traveling, the collaboration layer corresponds to the class of interaction, and the actions layer corresponds to the class of task. The main difference between this and the previously mentioned approaches for mobile agents, is that this one aims to cover all main types of agent design patterns.

4.3 Components

Components are logical groups of related objects that can provide certain functionalities. This might sound quite similar to agents, but in fact components are not autonomous as opposed to agents. By grouping related objects, components allow more coarse-grained re-use than the combination of single classes from scratch, this has shown to an effective and popular development approach in the software industry.

Erol, Lang and Levy [5] suggest a three-tier architecture that enables composition of agents by applying reusable components. The first tier is *interactions*, it is built up by agent roles and utterances. The second tier is *local information and expertise*, that enables the storage of information such as execution state, plan and constraints of the agent. *Information-content*, the third tier, is passive and often domain-specific, since it is often used to wrap legacy systems, e.g. a mainframe database application.

4.4 Graph Theory

Depke and Heckel [4] apply formal graph theory on requirement specifications for agent-systems in order to maintain consistency when the requirements are transformed into a design model.

5 Agents in the real-world

The agent-oriented approach is increasingly being applied in industrial applications, but it is far from as widespread as the object-oriented approach. This section describes where and how agents have been applied with success in the manufacturing industry.

Parunak [25] defines agenthood, a taxonomy and a maturity metric in an industrial context. His purpose is to improve the understanding and utilization of agent-oriented software engineering in industry.

Agenthood, i.e. agent-oriented programming, is explained as an iterative improvement of the industry-strength methodology of object-oriented programming.

The *taxonomy* classifies agent systems as belonging to one of the following *environments*: digital (i.e. software and digital hardware), social (involving human users) or electromechanical (non-digital hardware, e.g. a motor). Thereafter the taxonomy classifies agents according to the *interface* they support. Interface types are similar to the environments: digital (e.g. communication protols), social (e.g. user interfaces) and electromechanical (e.g. motor control interfaces).

Few business users, as opposed to researchers, are early-adapters of new and immature technology, as a result of this a *maturity metric* of agent-based systems is developed to be able to measure the level of agent technology and systems. The maturity metric has six degrees ranging from modeled applications to products. *Modeled applications*, the least mature, are theoretical applications in the form of architectural descriptions or analyses. The metric continues with *emulated applications* that are relatively immature due to the fact that they are simulations in a lab environment. *Prototype applications* represent the next maturity degree, they run in a non-commercial environment but on real hardware. *Pilot applications* are relatively mature applications, however they are not expected to be completely bug-free, and after a certain period they usually become more mature and become *production applications*. A production application is being applied in several businesses, but they require support for installation and maintenance. The most mature applications are *products*, they are usually shrink-wrapped and sold over desk, and they can usually be installed and maintained by the non-expert user.

5.1 Agents in the industry - where and how?

Parunak [24] presents a review of industrial agent applications. Application areas considered are: manufacturing scheduling, control, collaboration and agent simulation. Thereafter tools, methodologies, insights and problems for development of agent systems are presented and discussed.

Manufacturing scheduling is the ordering and timing of processes in a production facility. The purpose is to optimize the production by maximizing the number of units produced per time slot and keep good quality of the product, and minimize the resource requirements per unit and the risk of failures. Processes and machinery has to be *controlled* in order to operate as scheduled. The control can range from simple regulation of the power level for a piece of machinery to advanced real-time cybernetic control of processes. For many industries, human *collaboration* is needed to solve complex problems, e.g. in a design process engineers and designers have to collaborate in order to guarantee that products are pleasent to look in addition to being safe. In industries such as electronics production, there are tremendous setup costs for production facilities, consequently

there is a need for cost-efficient *simulation* of the manufacturing processes.

Agent methodologies in the industry

Methodologies for creating industrial agent systems presented are Rockwell's Foundation Technology and DaimlerChrysler's Agent Design for agent-based control [24].

In Rockwell's Foundation Technology four issues are considered in the development of agent-based control architectures, the first is *flexibility* related to fault-tolerance in a multi-objective environment, the second is *self-configuration* for the support of new products and rapidly changing old ones, without much manual reconfiguration, the third is *productivity* - how to at least maintain and hopefully improve productivity by applying agents, and the final issue is *equipment life span cost* - how to keep the agent in sync with life-cycle costs of the operating equipment.

Similar to Rockwell's approach, Daimler-Chrysler's Agent Design approach is also divided in four steps. The first step is to analyze and create a model of the manufacturing task, the second is to further investigate the model to identify and classify the roles that are needed, the third is to specify interactions between roles, and the final step is to specify agents that will fill these roles. This approach has much in common with the Gaia [10] and MaSE [31] methodologies with respect to role identification and interaction between roles.

6 Conclusion

This paper has sought to give a topical overview of recent progress of agent-oriented software engineering methodologies. Further work should include a more thorough analysis of the field in addition to practical testing of and experiments with the methods.

References

[1] Aridor Y. and Lange D. B. Agent Design Patterns: Elements of Agent Application Design. In *Proc. of the second international conference on Autonomous agents*, pages 108–115, 1998.

[2] Chaib-draa B. Connection between micro and macro aspects of agent modeling. In *Proc. of the first international conference on Autonomous agents*, pages 262–267, 1997.

[3] DeLoach S. A. Multiagent Systems Engineering A Methodology and Language for Designing Agent Systems. In *Proc. of Agent Oriented Information Systems*, pages 45–57, 1999.

[4] Depke R. and Heckel R. Formalizing the Development of Agent-Based Systems Using Graph Processes. In *Proc. of the ICALP'2000 Satellite Workshops, Workshop on Graph Transformation and Visual Modelling Techniques (GTVMT'00)*, pages 419–426, 2000.

[5] Erol K., Lang J. and Levy R. Designing Agents from Reusable Components. In *Proc. of the fourth international conference on Autonomous agents*, pages 76–77, 2000.

[6] Kendall E. A., Krishna P. V. M., Pathak C. V. and Suresh C. B. Patterns of intelligent and mobile agents. In *Proc. of the second international conference on Autonomous agents*, pages 92–99, 1998.

[7] Wooldridge M. J. and Jennings N. R. Pitfalls of agent-oriented development. In *Proc. of the second international conference on Autonomous agents*, pages 385–391, 1998.

[8] Wooldridge M. J., Jenning N. R. and Kinny D. A methodology for agent-oriented analysis and design. In *Proc. of the third international conference on Autonomous agents*, pages 69–76, 1999.

[9] Parunak H. V. D. and Odell J. Representing Social Structures in UML. In *Proc. of the fifth international conference on Autonomous agents*, Forthcoming, 2001.

[10] Wooldridge M. J., Jennings N. R. and Kinny D. The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, September 2000.

[11] Jennings N. R., Sycara K. and Wooldridge M. J. A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[12] Jennings N. R. On agent-based software engineering. *Artificial Intelligence*, 2000.

[13] Odell J., Parunak H. V. D. and Bauer B. Extending UML for Agents. In *Proc. of the Agent-Oriented Information Systems (AOIS) Workshop at the 17th National conference on Artificial Intelligence (AAAI)*, 2000.

[14] Odell J., Parunak H. V. D. and Bauer B. Representing Agent Interaction Protocols in UML. The First International Workshop on Agent-Oriented Software Engineering (AOSE-2000), 2000.

[15] Bergenti F. and Poggi A. Exploiting UML in the Design of Multi-Agent Systems. In *Proc. of the ECOOP - Workshop on Engineering Societies in the Agents' World 2000 (ESAW'00)*, pages 96–103, 2000.

[16] Jennings N. R. Agent-Based Computing: Promise and Perils. In *Proc. 16th Int. Joint Conf. on Artificial Intelligence (IJCAI-99)*, pages 1429–1436, 1999.

[17] Odell J. and Bock C. Suggested UML Extensions for Agents, December 1999.

[18] Kendall E. A., Malkoun M. and Jiang C. Multiagent systems design based on object oriented patterns. *Journal of Object Oriented Programming*, June 1997.

[19] Kendall E. A., Malkoun M. and Jiang C. The application of object-oriented analysis to agent based systems. *Journal of Object Oriented Programming*, February 1997.

[20] Jennings N. R. Building Complex Software Systems: The Case for an Agent-based Approach. *Communications of the ACM*, Forthcoming, 2001.

[21] Labrou Y., Finin T. and Peng Y. Agent Communication Languages: The Current Landscape. *IEEE Intelligent Systems*, 14(2), March/April 1999 1999.

[22] Lind J. Issues in Agent-Oriented Software Engineering. The First International Workshop on Agent-Oriented Software Engineering (AOSE-2000), 2000.

[23] Magnanelli M. and Norrie M. C. Databases for Agents and Agents for Databases. In *Proc. of 2nd International Bi-Conference Workshop on Agent-Oriented Information Systems*, June 2000.

[24] Parunak H. V. D. A Practitioner's Review of Industrial Agent Applications. *Autonomous Agents and Multi-Agent Systems*, 3(4):389–407, December 2000.

[25] Parunak H. V. D. Agents in Overalls: Experiences and Issues in the Development and Deployment of Industrial Agent-Based Systems. *International Journal of Cooperative Information Systems*, 9(3):209–227, 2000.

[26] Rana O. F. and Biancheri C. A Petri Net Model of the Meeting Design Pattern for Mobile-Stationary Agent Interaction. In *Proc. of the 32nd Hawaii International Conference on System Sciences*, 1999.

[27] Nwana H. S. and Ndumu D. A perspective on software agents research. *The Knowledge Engineering Review*, 14(2):1–18, 1999.

[28] Shoham Y. Agent-oriented programming. *Artificial Intelligence*, (60):51–92, 1993.

[29] Wagner G. Agent-Object-Relationship Modeling. In *Proc. of Second International Symposium - from Agent Theory to Agent Implementation together with EMCRS 2000*, April 2000.

[30] Wagner G. Agent-Oriented Analysis and Design of Organizational Information Systems. In *Proc. of Fourth IEEE International Baltic Workshop on Databases and Information Systems, Vilnius (Lithuania)*, May 2000.

[31] Wood M. F. and DeLoach S. A. An Overview of the Multiagent Systems Engineering Methodology. The First International Workshop on Agent-Oriented Software Engineering (AOSE-2000), 2000.

[32] Wooldridge M. J. and Jennings N. R. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 2(10):115–152, 1995.

[33] Wooldridge M. J. and Jennings N. R. Software Engineering with Agents: Pitfalls and Pratfalls. *IEEE Internet Computing*, 3(3):20–27, May/June 1999.

[34] Yim H., Cho K., Jongwoo K. and Park S. Architecture-Centric Object-Oriented Design Method for Multi-Agent Systems. In *Proc. of the Fourth International Conference on MultiAgent Systems (ICMAS-2000)*, 2000.

[35] Zambonelli F., Jennings N. R., Omicini A. and Wooldridge M. J. *Coordination of Internet Agents: Models, Technologies and Applications*, chapter 13. Springer-verlag, 2000. Agent-Oriented Software Engineering for Internet Applications.