Agent-Oriented Software Engineering "Research Area Examination"*

March 31, 2005

1 Introduction

Agent-oriented software engineering is considered to be a new paradigm. It represents an exciting new means for analyzing, designing, and building complex software systems. Because it is a new paradigm, researchers have directed their efforts towards accomplishing three primary tasks.

The first task is to qualitatively prove that this technology is an appropriate way to effectively handle the complexity of current software systems. The common argument is based on showing that agent-oriented software engineering is a natural evolution of object techniques and is often better in dealing with the complexity of today's software systems. Signs of such an evolution in the current software are: 1) most software systems are now de-facto concurrent and distributed and are expected to interact with components and exploit services that are dynamically found on the web and 2) software systems tend to be open in that they exist in dynamic operating environments where new components join these systems and existing components leave them. Current technologies (e.g. object technologies) fall short or at best require extremely hard solutions for today's complex software.

The second task is to build appropriate tools and abstractions to represent the new concepts introduced by this new paradigm. Researchers have been trying to extend UML (Unified Modeling Language) with new notation so that it becomes appropriate for agents. Other researchers take agent theory as their inspiration and produce notation peculiar to agents.

The third task is to take agent technology from research to industry. This requires the creation of methodologies for guiding developers to efficiently and inexpensively analyze and design agent-based systems.

*This work is supported by the National Science Foundation under grant #IIS-0083127.

This survey is outlined as follows. Section 2 introduces the fundamental concepts of software engineering. Section 3 provides an overview of agent theory. Section 4 introduces agent-oriented software engineering. Finally, in section 5, we give some directions for future work in agent research.

2 Software Engineering

The purpose of this section is to highlight the main principles of software engineering. The relevancy of this section emerges from the fact that software engineering principles provide the basis for agent-oriented software engineering.

2.1 Fundamentals of Software Engineering [GJM91]

This book covers the fundamental principles of software engineering. The book discusses in detail many principles, including rigor and formality, separation of concerns, modularity, abstraction, anticipation of change, generality, and incrementality, and their effects on software engineering. Software design is discussed in detail. Specifically, the design activity and its objectives such as design for change, modularization techniques, and design notations. The book also discusses top-down and bottom-up strategies to design a system. Software specification is also discussed. Specifically, the book focuses on uses of specification (e.g. statement of user needs), specification qualities (e.g. clearness, consistency, etc.), classification of specification styles and how the specification highly affects the whole software development process. The types of system verification such as testing and analysis are also covered. The book also discusses the software production process in much detail. Specifically the book sheds light on many software production process models such as the water fall model and the evolutionary model. Finally, the book discusses in detail the tools and the environments that can be used to automate or semi-automate some of the software engineering activities and how these tools can simplify the job of engineers

2.2 Software Architecture: A Roadmap [Gar00]

The important trends of software architecture in research and practice are addressed in this paper. First, the author discusses the key roles of architectures in software systems development (e.g. as a bridge between requirement and implementation). Second, the author summarizes the past and current state of software architectures. In the past decade, architectures were

ad-hoc. Descriptions depended on informal box-and-line diagrams and architectural choices were made by adapting some previous design. Today, the technological basis for architectural design has improved. Specifically three advancements have had a major role in improving architectural design: *Architectural Description Languages* and *Tools* such as the Adage language to formally represent and analyze architectural design, *Product Lines* and *Standards* to exploit commonalities across multiple products, and *Codification* and *Dissemination* to spread knowledge about architectures and techniques. Third, the author discusses the future changes in the software world and their impacts on software architecture. Three major trends and their impacts on software architecture are discussed. The first trend is *Changing Build-Versus-Buy Balance* where software is not totally developed in the same company; rather it is built from components bought from other companies. This definitely has consequences for software architecture, examples of which are the need for industry-wide standards and the need for standardization of notations and tools across vendors. The second trend is *Network-Centric Computing*, which changes the nature of software from having central control to being distributed. Distribution creates new challenges for software architecture, examples of which are the need to scale up with the size and variability of the Internet and to support computing with dynamically formed coalitions of distributed resources. The third trend is *Pervasive Computing*, which has many consequences for software architecture, examples of which are the need for architecture suited for systems in which the resource usage is the critical issue and architectures that need to be more flexible.

3 Agent Theory

The purpose of this section is to introduce *software agents*, *agent-based systems*, and *multi-agent systems*. The relevancy of this section immediately emerges from the fact that agent-oriented software engineering takes the agent as its fundamental abstraction.

3.1 Introduction to Software Agents [Bra97]

In the first chapter of his book (software agents) Bradshaw discusses two major concepts, namely the definition of software agents and the advantages of software agents. According to Bradshaw an agent is "a software entity, which functions continuously and autonomously in a particular environment, often inhabited by other agents and processes." The requirement for continuity and

autonomy comes from the desire that agents are able to carry out activities in a flexible way without direct guidance from a human. Also, agents that inhabit the same environment should ideally be able to cooperate. Agents usually enjoy some properties such reactivity—able to sense the environment and act on it and socialability—able to cooperate. In addition, Bradshaw also discusses how the agent could be classified using different perspectives. For example, the AI (Artificial Intelligence)community distinguishes two types of agents, weak and strong agency, and the DAI (Distributed AI) community classifies agents by their degree of problem solving (e.g. reactive agents and intentional agents).

Software agents provide two major advantages. First they can be used to simplify distributed computing. The major barrier to intelligent interoperability is that the low level of interoperability among systems. Software agents can help mitigate this problem by embedding one or more peer agents within cooperating systems. Applications can request services through these agents at a higher level corresponding more to user intentions than to specific implementations. Second, software agents can overcome the limitation of the current user interfaces approaches. Examples of these limitations include large-scale spaces, actions in response to immediate user interaction only, and no improvement of behavior. Software agents can help overcome these limitations by allowing an indirect management style of interaction. In such an approach, users are not required to spell out each action for the computer explicitly; instead, the flexibility of software agents will allow them to only give general guidelines

3.2 Is it an Agent or just a Program?: A Taxonomy for Agents [FG96]

The authors of this paper try to make a clear distinction between agents and normal programs by formally defining the notion of an agent. After listing many definitions given by others and finding neither consensus on what constitutes an agent nor how an agent differs from a normal program, the authors give their own: "An autonomous agent is a system situated within and part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future."

This definition seems to rule out many ordinary programs, for instance, a payroll is a program, but not an agent, because it fails to satisfy the definition above, namely it fails to affect "what it senses in the future" and also fails "over time." However, they observe that this definition is still very broad

and even a thermostat could qualify as an agent, therefore they discussed various properties of agents and offered a taxonomy that covers most of the agent examples found in literature. Below this initial classification, they suggest that agents could be further categorized by their control structure, their environment (e.g. planning, database, file system, network, Internet), by languages in which they are written, or by applications on which they operate.

3.3 A Roadmap of Agent Research and Development [JSW98]

The authors summarize many issues in agent research. First, the notion of agent is defined as "a software system that is situated in some world and is capable of flexible autonomous actions in order to meet its design objectives." Here, *situated* means that an agent receives input from its world and acts on that world, *autonomy* means an agent acts without direct intervention of a human and has control over its state and behavior, and *flexible* means *reactive* (responds to world changes in a timely fashion), *proactive* (goal-driven), and *social* (interaction ability). Second, an *agent-based system* is defined as a system whose key abstraction is an agent and could have one or more agents and a *multi-agent system* is defined as a collection of interacting agents, which is ideally suited for representing problems that have multiple problem solving methods/entities. Third, the authors briefly show that the current interest in autonomous agents did not emerge from a vacuum because researchers from different disciplines have been talking about closely related issues such as *artificial intelligence*, which is concerned with building artificial artifacts that if they sense the environment, they can be considered as agents, and *objects* and *concurrent object systems*, where objects are similar to agents in that both support, for instance, encapsulation, but largely differ in other aspects such as the degree to which objects and agents are autonomous. Concurrent objects, however, are close to the notion of agents although not capable of flexible behavior. Fourth, the authors point out some incentives for the increasing interest in multi-agent systems such as the ability to provide robust behavior and to allow inter-operation among legacy systems.

3.4 Agent Communication Languages: Knowledge Querying and Manipulating Language—KQML [FLM97]

KQML is a high-level, message-oriented communication language and protocol for information exchange among agents. The KQML language is divided into three layers. The content layer carries the actual message that the sending agent wants to deliver to a receiving agent. The communication layer encodes a set of parameters that describe some lower level communication parameters such as a sender, a receiver, and a unique identifier that uniquely identifies a message. The message layer specifies the speech act (or performative) of the message, the ontology that defines the concepts in the content, and the language in which the content is encoded.

4 Agent-Oriented Software Engineering

The four subsections in this section provide an overview of the main topics in agent-oriented software engineering (AOSE). Section 4.1 justifies this new paradigm in software engineering. Section 4.2 provides an overview of the modeling tools that are used to model agent systems. Section 4.3 gives some of the methodologies. Section 4.4 discusses the reuse in agent paradigm.

4.1 On Agent-Based Software Engineering [Jen00]

In this paper, the author advocates the agent-oriented approach to software engineering. He makes a qualitative argument to show that the agent approach is very effective in dealing with the complexity exhibited in today's software systems. The author argues that analyzing, designing, and implementing complex software systems as a collection of interacting, autonomous, flexible components (i.e. agents) provide software engineers several advantages over contemporary methods. The author uses qualitative justification (although a quantitative one would be more compelling, but no supporting data is available) to support his argument. The qualitative justification is built on three principles, which are exactly the principles used by software engineers to deal with the complexity of systems: 1) showing that agent-oriented decompositions are an effective way of partitioning the problem space, 2) showing that the key abstractions of agent-oriented mindset (agents, interactions, and organizations) are a natural means for modeling complex systems, and 3) showing that the agent-oriented philosophy for modeling

and managing organizational relationships is appropriate for dealing with dependencies and interactions that exist in complex systems.

In addition, the author investigates the possibility of wide adoption for agent-oriented techniques. He considerers two pragmatic issues that determine whether agent-oriented will catch on as a software engineering paradigm, namely the degree to which agents represent a radical departure from current software engineering thinking and the degree to which existing software can be integrated with agents.

4.2 Agent Modeling Tools

This section provides an overview of the tools (languages) to model agent systems. Two major papers are presented here.

4.2.1 AML: Agent Modeling Language Toward Industry-Graded Agent-Based Modeling [CTCG04]

The authors describe a modeling language, called AML (Agent Modeling Language). AML is a semi-formal visual language for specifying, modeling, and documenting systems that incorporate concepts from Multi-Agent Systems(MAS) theory. The motivation behind AML is the need for a highly expressive modeling language suitable for developing software systems based on multi-agent technologies. AML takes UML 2.0 as its starting point and augments it with modeling concepts that suitably capture typical features of MAS. In addition, AML can be extended with new concepts to cover any new concepts in MAS. AML provides constructs to model many aspects of MAS. Examples include:

1. Constructs to model architectural aspects of MAS (e.g. ontologies and social aspects).

2. Constructs to model the aspects of MAS behavior (e.g. communicative interactions)

3. Constructs to model mental aspects of agents (e.g. beliefs, goals, plans, and mental relations).

4.2.2 Representing Agent Interaction Protocols with Agent UML [PO04]

The authors suggest extensions to the Unified Modeling Language, UML 2.0, to efficiently support agent protocol modeling. Specifically, the authors suggest some extensions that increase the expressivity of sequence diagrams to

take into account the requirement imposed by the richness of agent interactions. The extensions affect the following UML 2.0 elements.

1. *Lifeline*. Lifeline can represent a set of agents rather than a unique agent as in UML 2.0 through adding roles and grouping agents that have the same behavior in this interaction. Also, the dynamics of roles are captured by adding two stereotypes $<< addrole >>$ and $<< changerole >>$.

2. *Messages*. Important modifications to UML 2.0 message elements include the name of an instance agent, which must be written on the message close to the receiver lifeline (since the lifeline can represent more than one instance agent) and cardinality, which is added on the message to specify how many agents receive the message (since each lifeline may represent more than one instance agent).

3. *Protocol Template*. The purpose of protocol template is to create reusable patterns for useful protocol instances. To define a protocol template, Agent UML provides a stereotype $<< unbound >>$. The unbound protocol can be instantiated by binding its parameters.

4. *Action*. Sending and receiving messages imply performing actions with agents. An action is depicted as a round-cornered rectangle linked to the message that triggers it. The action is written inside the round-cornered rectangle.

4.3 Methodologies

Agent researchers have produced methodologies to assist engineers to create agent-based systems. Some researchers have taken agent theory as their starting point and have produced methodologies that are rooted in that theory (Section 4.3.1). Other researchers have taken object techniques as their point of departure and have enriched them to be suitable for agents (Section 4.3.2). Others have taken knowledge engineering concepts and extended them (Section 4.3.3). Researchers also have tried to assemble methodologies by combining features from different methodologies (Section 4.3.4). Yet other researchers have produced methodologies based on both agent and object technologies (Section 4.3.5).

4.3.1 Approaches Based on Agent Technology

The rationale behind these approaches is that, despite the fact that objects are superficially similar to agents, there are significant differences. Hence,

methodologies should logically be rooted in agent theory. In what follows, we briefly outline the important features of the most cited methodologies.

Methodologies that are based on agent theory conceive a multi-agent system as a society rather than collections of interacting entities [ZJW03, Omi01, BNR+02, JPS02]. These methodologies cover analysis and design; however, [JPS02] goes a step further by supporting requirement gathering. Generally, the analysis phase of these methodologies produces three models. The *role model* describes the basic skills (roles) needed by a system to accomplish its goals. A *role* is described by the role schema, which contains information about the role such as its responsibilities. The role model in [Omi01] describes not only individual roles, but also groups of roles that are in charge of doing some social tasks. The *environment model*, called *resource model* in [Omi01], describes the place in which a system is situated. This description includes, for instance, the services provided by the environment and the data a role can access or manipulate. The *interaction model* describes a set of protocols associated with roles. The design phase elaborates on the analysis models and produces more detailed models. The *agent model* [ZJW03, JPS02, Omi01] describes the agent classes that populate the system and how many instances of each agent must be used. The *service model* [ZJW03, JPS02] describes the services that an agent must provide. It is worth mentioning though that [Omi01] deals only with inter-agent issues and the intra-agent issues are completely ignored. As a result, it has nothing to say about the service model; instead it produces two models, namely a *society model*, which describes how a group of roles is mapped to a society of agents, and an *environment model*, which describes how resources are mapped to infrastructure classes, which are characterized by the services, the access models, and the permissions granted to roles and groups. The Tropos Software Development Methodology [GMP02] gives strong emphasis on early requirement analysis where the stakeholders and their objectives are identified and analyzed. Tropos uses many primitive knowledge level concepts such as *Actor* and *Goal* to build different types of models.

4.3.2 Approaches Based on Object Technology

Some researchers take object-oriented technologies as their starting point and enriched it with extensions to be suitable for agent concepts. They give many reasons for such an approach. First, there are similarities between the agent paradigm and the object paradigm in that agents can be thought of as active objects. Second, both paradigms use message passing as a means for communication and use inheritance and aggregations for defining architectures. Finally, extending techniques that have been in use for a long time and well

understood by engineers may result in accelerating agent use in industry. In what follows, we outline some of the most cited methodologies.

Multi-agent Systems Engineering [WD01] leads designer from a system specification to an implemented agent system. The development process passes through seven steps such as capturing goals of a system and creating roles that are responsible for achieving these goals.

In [KKB+03] an extension to UML along with a framework to Model and Design Multi-Agent Systems is proposed. Some of the extensions are new modeling constructs such as *Belief* and *Goal* and diagrams such as *Agent Goal Diagram* and *Use Case Goal Diagram*. Using these constructs and diagrams, the paper suggests a many-step process to model a multi-agent system.

PASSI (a Process for Agent Societies Specification and Implementation) [CP02] is a methodology for designing and developing multi-agent societies, integrating design models and concepts from both object-oriented software engineering and artificial intelligence approaches using the UML notation. The methodology produces many models that cover all the development process from the system requirement (System Requirements Model) to code (Code and Deployment Model).

The Prometheus[1] methodology [PW02] is a detailed and complete process for specifying, analyzing, designing, and implementing intelligent agent systems. The methodology consists of three phases: *system specification*—determining actions, perceptions, and functionalities, *architectural design*—determining types of agents in a system and their functionalities, and *detailed design*—developing internal structure of each agent and how it accomplishes its task(s).

4.3.3 Approaches Based on Knowledge Engineering Technology

Knowledge engineers argued that methodologies from knowledge engineering are useful to model agent-based systems because agents have cognitive characteristics and these methodologies best model this knowledge. In addition, the expertise gained in the field of knowledge engineering can expedite adopting agent technology in industry.

The most cited methodology is MAS-CommonKADS [IMGV98]. The methodology starts with a conceptualization phase that is an informal phase for collecting the user requirements and obtaining an initial description of the system from the user's point of view. The methodology then produces many

[1]Prometheus was the wisest Titan. His name means "forethought" and he was able to foretell the future. Prometheus was known as a protector and benefactor of man. He also gave mankind a number of gifts including fire (CF. http://www.greekmythology.com)

models such as the *Agent Model* and the *Task Model* for the analysis and design of the system. For each model, the methodology defines the constituents (entities to be modeled) and the relationships between the constituents. The methodology defines a textual template for describing every constituent and a set of activities for building every model.

4.3.4 Assembling Methodologies

Some researchers believe that creating a general-purpose methodology that satisfies all the issues in agent software engineering is unlikely to be feasible. They, therefore, suggest merging the strong features from some of the current methodologies to obtain a new methodology.

In [JSW02] a skeleton methodology is assembled from the Prometheus and ROADMAP methodologies. The skeleton methodology is independent of the implementation architecture and supports analysis and architectural design. It has six models (common to both the Prometheus and ROADMAP) that represent the core of the methodology and are created during the analysis phase and elaborated during the design phase. There are optional models obtained from the Prometheus and ROADMAP (peculiar to each) that can be used as needed. It is worth noting that the core six models are sufficient to model systems with low agency needs. However, when a stronger agency is required, the required optional models can replace some of the core models.

In [JSMM03] a conceptual framework for creating and reusing modular methodologies is proposed. The foundation of the framework is the concept of an *AOSE feature*. An AOSE feature is an encapsulation of software engineering techniques, models, supporting CASE tools, and development knowledge design such as patterns. An AOSE feature can be used by following the procedure of the encapsulated techniques to create models. The features can be created by a three-step process.

4.3.5 Other Apporaches

Other methodologies rely on both agent theory and object techniques. In [HGR03] a three-level technique for modeling agent-based systems is proposed, the role model, the agent model, and the object model. Each level uses some methodology such as GAIA [ZJW03]. The main idea behind this approach is to produce a meta-model for each level and a translator that transforms the models in a previous level to models accepted by the next level.

In [KR02] a methodology that enables collaboration between domain experts and engineers is proposed. The main idea of this approach is to start a

multi-agent system development with one model, called the process model, and refine this model to produce other models, which evolve to the multi-agent system.

4.4 Reuse in Agent Software

Agent-oriented software engineering, like other software engineering approaches, addresses reuse issues.

4.4.1 Reuse in Agent-Based Application Development [Gir02]

The author discusses software reuse in the agent world and proposes a model to exploit reuse. In this model, the reusable agent-based software abstractions are described in terms of their abstraction level and their domain dependency. In the model, the domain model and the user model abstractions are produced by the requirements analysis. The requirements analysis produces the requirements specification of a group of similar systems in the domain. The domain model (domain dependent, but specified at a high-level of abstraction) represents the formulation of a problem. The user model specifies the features, needs, preferences, and goals of the users.

The design stage of agent-based application engineering produces a reusable design specification for a family of similar systems in the application domain. The design specification consists of agent-based architectural styles, software patterns, and frameworks.

4.4.2 Agent Patterns

In [Lin02] a pattern catalog for agent-oriented patterns and a pattern description schema is suggested. The suggested catalog categorizes patterns under classes. These classes are: Interaction Pattern, Role Pattern, Architectural Pattern, Society Pattern, System Pattern, Task Pattern, and Environment Pattern. Pattern description schemata bring together a set of features that capture a software pattern.

In [GL04] a catalog of organizational patterns is added to the GAIA methodology. Organizational patterns are exploited in the design step of the methodology. The organizational patterns are described using a comprehensive structured description that facilitates the selection of the most appropriate pattern. The description is divided into parts, *general part*, which is similar to those found in other pattern descriptions and *particular part*, which is specific to organizational patterns.

5 Future Directions

In my opinion, there are two main directions for future work in agent-oriented software engineering.

1. *Openness.* Domain specific AOSE does not yet have good support for open systems for specific domains that permit any agent to participate, provided that this agent conforms to the system's rules. This requires changes to current methodologies. Methodologies need to be more specialized to specific applications (e.g. information gathering) and need to provide more guidance and models to account for, at least, all the foreseen scenarios.

2. *Semantic Web.* The semantic web is likely to significantly change the way in which agents and agent systems will be created. We expect that agents and agent systems will be developed by users (not engineers) within a short time. This imposes many requirements. First, it requires libraries of ready-to-use high-level components that can be used to assemble an agent. Second, it requires a methodology that guides users to configure these components and to combine specific components to create an agent. Third, the methodology must also suggest how to collect together the created agents into a multi-agent system that satisfies a user's goals.

References

[BNR+02] W. Brauer, M. Nickles, M. Rovatsos, G. Weib, and K. Lorentzen. Expand: expectation-oriented analysis and design. In *Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE 2002)*, pages 46–54, New York, 2002.

[Bra97] J. BradShaw. Introduction to Software Agents. In J. BradShaw, editor, *Software Agents*, pages 3–46. AAAI Press, Menlo Park, California, 1997.

[CP02] M. Cossentino and C. Potts. A CASE Tool Supported Methodology for the Design of Multi-Agent Systems. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'02)*, pages 113–120, Las Vegas, Nevada, June 2002.

[CTCG04] R. Cervenka, I. Trencansky, M. Calisti, and D. Greenwood. AML: Agent Modeling Language Toward Industry-Grade Agent-Based Modeling. In *Proceedings of the Fifth International Workshop on Agent-Oriented Software Engineering (AOSE 2004)*, New York City, July 2004.

[FG96] S. Franklin and A. Graesser. Is it an Agent or just a Program?: A Taxonomy for Autonomous Agents. In *Proceedings of the Third International Workshop on Agent Theory, Architectures, and Languages*, pages 21–35, New York, New York, July 1996.

[FLM97] T. Finn, Y. Labrou, and J. Mayfield. KQML as an Agent Communication Language. In J. Bradshaw, editor, *Software Agents*, pages 291–316. AAAI Press, Menlo Park, California, 1997.

[Gar00] D. Garlan. Software Architecture: A Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[Gir02] R. Girardi. Reuse in Agent-Based Application Development. In *Proceedings the First International Workshop on Software Engineering for Large-Scale Multi-Agent Systems*, Orland, Florida, May 2002.

[GJM91] C. Ghezzi, M. Jazayeri, and D. Mandrioli. F*undamentals of* S*oftware* E*ngineering*. Prentice Hall, New Jersy, 1991.

[GL04] J. Gonzales and M. Luck. A Framework for Patterns in Gaia: A Case-Study with Organizations. In *Proceedings of the Fifth International Workshop on Agent-Oriented Software Engineering (AOSE 2004)*, New York, New York, July 2004.

[GMP02] F. Giunchiglia, J. Mylopoulos, and A. Perini. The Tropos Software Development Methodology: Processes, Models, and Diagrams. In *Proceedings the Third International Workshop on Agent-Oriented Software Engineering*, Bologna, Italy, July 2002.

[HGR03] F. Hernandez, J. Gray, and K. Reilly. A Multi-Level Technique for Modeling Agent-Based Systems. In *Proceedings of the Second International Workshop on Agent-Oriented Methodologies*, Anaheim, California, October 2003.

[IMGV98] A. Iglesias, G. Mercedes, C. Gonzalez, and R. Velasco. Analysis and Design of Multiagent Systems using MAS-CommonKADS. In

Proceedings of the Fourth International Workshop on Intelligent Agents IV, Agent Theories, Architectures, and Languages, pages 313–327. Springer-Verlag, 1998.

[Jen00] N. Jennings. On Agent-Based Software Engieering. *Artificial Intelligence*, 117(1):277–296, June 2000.

[JPS02] T. Juan, A. Pierce, and L. Sterling. ROADMAP: Extending the Gaia Methodology for Complex Open Systems. In *Proceedings of the First ACM Joint Conference on Autonomous Agents and Multi-Agent Systems, ACM Press*, pages 13–10, Bologna, Italy, July 2002.

[JSMM03] T. Juan, L. Sterling, M. Martelli, and V. Mascardi. Customizing AOSE Methodologies by Reusing AOSE Features. In *Proceedings of the Second International Conference on Agents and Multi-Agent Systems (AAMAS 2003)*, pages 113–120, Melbourne, Australia, July 2003.

[JSW98] N. Jennings, K. Sycara, and M. Woodridge. A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems*, 1:7–38, 1998.

[JSW02] T. Juan, L. Sterling, and M. Winikoff. Assembling Agent Oriented Software Engineering Methodology from Features. In *Proceedings of the third International Workshop on Agent-Oriented Software Engineering*, pages 198–209, Bologna, Italy, July 2002.

[KKB+03] K. Kavi, D. Kung, H. Bhambhani, G. Pancholi, and M. anikarla. Extending UML for Modeling and Design of Multi-Agent Systems. In *Proceedings of the Second International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2003)*, Portland, Oregon, May 2003.

[KR02] Knubaluch and T. Rose. Tool-Supported Process Analysis and Design for the Development of Multi-Agent Systems. In *Proceedings of the third International Workshop on Agent-Oriented Software Engineering (AOSE 2002)*, Bologna, Italy, July 2002.

[Lin02] J. Lind. Patterns in Agent-Oriented Software Engineering. In *Proceedings of the Third International Workshop on Agent-Oriented Software Engineering (AOSE 2002)*, Bologna, Italy, July 2002.

[Omi01] A. Omicini. SODA: Societies and Infrastructures in the Analysis and Design of Agent-Based Systems. In *Proceedings of the First International Workshop on Agent-Oriented Software Engineering (AOSE 2001)*, pages 185–194, New York, May 2001.

[PO04] M. Philippe and J. Odell. Representing Agent Interaction Protocols with Agent UML. In *Proceedings of the Fifth International Workshop on Agent-Oriented Software Engineering (AOSE 2004)*, New York, New York, July 2004.

[PW02] L. Padgham and M. Winiko. Prometheus: A Methodology for Developing Intelligent Agents. In *Proceedings the Third International Workshop on Agent-Oriented Software Engineering (AOSE 2002)*, Bologna, Italy, July 2002.

[WD01] M .Wood and S. DeLoach. An Overview of the Multiagent Systems Engineering. In *Proceedings of the First International Workshop Agent-Orinted Software Engieering (AOSE 2000), Lecture Notes in Artificial Intelligence*, volume 1957, pages 207–215, Springer Verlag, Berlin, January 2001.

[ZJW03] F. Zembonelli, N. Jennings, and M. Woodridge. Developing Multiagent Systems: The GAIA Methodology. *ACM Transactions on Software and Methodologies*, 12(3):317–370, July 2003.