UML 2.0 and agents: how to build agent-based systems with the new UML standard

Bernhard Bauer[a], James Odell[b,*]

[a]*Programming of Distributed Systems, Institute of Computer Science, University of Augsburg, D-86135 Augsburg, Germany*
[b]*Agentis Software, Ann Arbor, Michigan, USA*

1. Introduction

Software engineering techniques are a key prerequisite of running successful software projects. Without a sufficient approach and adequate tools to support the development of software systems, it is virtually impossible to cope with the complexity of commercial software development processes. This tendency will increase over the years to come and appropriate software engineering methods will continually be in high demand. A *software methodology* is typically characterized by a *modeling language*—used for the description of models, defining the elements of the model together with a specific syntax (notation) and associated semantics—and a *software process*—defining the development activities, the inter-relationships among the activities, and how the different activities are performed. In particular, the software process defines phases for process and project management as well as quality assurance. Each activity results in one or more deliverables—such as specification documents, analysis models, designs, code, testing specifications, testing reports, performance evaluation reports, etc.—serving as input for subsequent activities.

The three key phases that one is likely to find in any software engineering process are that of analysis, design and implementation. In a strict waterfall model these are the only phases; more recent software development process models employ a "round-trip engineering" approach, i.e. provide an iteration of smaller granularity cycles, in which models developed in earlier phases can be refined and adapted in later phases.

Agent technology enables the realization of complex software systems characterized by situation awareness and intelligent behavior, a high degree of distribution, as well as mobility support. Agent technology has the potential to play a key role in enabling intelligent applications and services by improving automation of routine processes, and supporting the nomadic users with pro-active and intelligent assistance based on principles of adaptation and self-organization. Hence, agent technology can open the way to new application

*Corresponding author. Programming of Distributed Systems, Institute of Computer Science, University of Augsburg, D-86135 Augsburg, Germany. Tel.: +1 734 994 0833.

E-mail addresses: bauer@informatik.uni-augsburg.de (B. Bauer), email@jamesodell.com (J. Odell).

URL: http://www.jamesodell.com.

domains while supporting the integration of existing and new software, and make the development process for such applications easier and more flexible. However, deploying agent technology successfully in industrial applications requires industrial-quality software methods and explicit engineering tools, such as Unified Modeling Language (UML 2.0). Therefore, we will take a closer look at the new UML standard and its application to agent-based systems.

This paper is structured as follows. In Section 2 we give a short overview on agent methodologies and notations to have a reference what has to be specified in agent-based systems. The main chapter is Section 3 showing the different UML diagrams and their application for agent-based systems. Moreover, we apply the model-driven architecture (MDA) of the Object Management Group (OMG) for the specification of agent-based systems. The paper concludes with some open issues using UML 2.0 for the specification of such systems and concludes the paper.

2. Short overview on agent methodologies and notations

A considerable number of agent-oriented methodologies and tools are available today, and the agent community is facing the problem of identifying a common vocabulary to support them (for details see our work in Bauer and Müller, 2004, this section is based on it). There is a considerable interest in the agent R&D community in methods and tools for analyzing and designing complex agent-based software systems, including various approaches to formal specification (see Iglesias et al., 1998 for a survey). Since 1996, agent-based software engineering has been in the focus of the ATAL workshop series; it also was the main topic of the 1999 MAAMAW workshop (Garijo and Boman, 1999). Various researchers have developed methodologies for agent design, touching on representational mechanisms, like the GAIA methodology (Wooldridge et al., 2000a) or the extensive program underway at the Free University of Amsterdam on compositional methodologies for requirements (Herlea et al., 1999), design (Brazier et al., 1998), and verification (Jonker and Treur, 1997). Kinny et al. (1996a) and Kinny and Georgeff (1996a) propose a modeling technique for BDI agents. The close affinity between design mechanisms employed for agent-based system and those used for object-oriented systems is shared by a number of authors, e.g. Birgit Burmeister (1996). In particular, since 2000, the Agent-Oriented Software Engineering Workshop (AOSE) has become the major forum for research carried out on these topics, including new methodologies such as Tropos (Schreiber et al., 1994), Prometheus home page: http://www.cs.rmit.edu.au/agents/SAC/methodology.shtml, and MESSAGE web site: http://www.eurescom.de/public/projects/P900-series/p907/. Currently, most industrial methodologies are based on the Object Management Group's (OMG) UML accompanied by process frameworks such as the Rational Unified Process (RUP), see Jacobson et al. (1998) for details. The MDA (http://www.omg.org/mda/) from the OMG allows a cascade if code generations from high-level models (platform-independent model—PIM) via platform-dependent models to directly executable code (see Section 3.2 for details). Another approach for agile software engineering that has been receiving active coverage is Extreme Programming (Beck, 1999).

2.1. Knowledge engineering approaches

Most early approaches supporting the software engineering of agent-based systems were inspired by the knowledge engineering community. The three most influential methodologies inspired by this strand of research are CommonKADS (Schreiber et al., 1994), CoMoMAS (Norbert Glaser, 1996) and MAS-CommonKADS (Iglesias et al., 1996). Knowledge engineers need tools and methods to design good knowledge-based systems; however, this relies on knowledge engineer abilities. The CommonKADS methodology was developed to support knowledge engineers in modeling expert knowledge and developing design specifications in textual or diagrammatic form. CommonKADS is a knowledge engineering methodology as well as a knowledge management framework. Two extensions to CommonKADS were developed that take agent-specific aspects into account: CoMoMAS and MAS-CommonKADS. MAS-CommonKADS (Iglesias et al., 1996) adds various extensions to CommonKADS: protocol engineering techniques (namely software development lifecycle and MSC96, Rudolph et al., 1996); object-oriented techniques (OMT Rumbaugh, 1995a, b) and OOSE); and enhanced support to additional phases within the software life cycle.

2.2. Agent-oriented approaches

A perceived lack of the knowledge engineering software development methodologies was that they were not designed before the background of supporting the development of agent systems, and that hence they had limited capability to support agent-specific functions, which could only partly be overcome by extensions such as those seen for MAS-CommonKADS. Gaia (Wooldridge et al., 2000b) is a methodology for agent-oriented analysis and design supporting macro (societal) level as well as micro (agent)-level aspects. Following Wooldridge et al. (2000b), Gaia was designed to deal with coarse-grained computational systems, to maximize some global quality measure, to handle heterogeneous agents independent of programming languages

and agent architectures, having static organization structures and agents having static abilities and services, with less than 100 different agent types. ROADMAP extends Gaia by adding elements to deal with the requirements analysis in more detail by using use cases and to handle open systems environments. Moreover, it focuses more on the specification of interactions based on AUML (Bauer et al., 2001a). Another AOSE methodology mainly focusing on societies similar to Gaia's organizations is Societies in Open and Distributed Agent (SODA) spaces (Omicini, 2000). As ROADMAP it addresses some of the shortcomings of Gaia such as the insufficiencies in dealing with open systems or self-interested agents. Moreover, SODA takes the agent environment into account and provides mechanisms for specific abstractions and procedures for the design of agent infrastructures. Based on the analysis and design of *agent societies* (exhibiting the global behaviors not deducible from the behavior of the individual agents) and *agent environments* (the space where agents live and interact, like open, distributed, decentralized, heterogeneous, dynamic, and unpredictable environment), SODA provides support for modeling the *inter-agent* aspects. However, *intra*-agent aspects are *not* covered. Therefore, SODA is not a complete methodology; rather, its goal is to define a coherent conceptual framework and a comprehensive software engineering procedure that accounts for the analysis and design of individual agents from a behavioral point of view, agent societies, and agent environments. As most of the considered methodologies do, SODA supports the analysis and design phase.

2.3. Object-oriented approaches

A good procedure (if not the only viable one) for successful industrial deployment of agent technology is to present the new technology as an incremental extension of known and trusted methods, and to provide powerful engineering tools that support industry-accepted methods of technology deployment. Accepted methods of industrial software development depend on standard representations for artifacts to support the analysis, specification, and design of agent software. At the moment AOSE is still lacking the availability of suitable software processes and tools. The UML is gaining wide acceptance for the representation of engineering artifacts using the object-oriented paradigm. Viewing agents as the next step beyond objects leads several authors (see on a discussion of this topic, e.g. (Bauer et al., 2001b) to explore extensions to UML and idioms within UML to accommodate the distinctive requirements of agents as well as defining software methodologies based on object-oriented approaches. In general, building methods and tools for agent-oriented software development on their object-oriented counterparts seems suitable as it lends itself to smoother migration between these different technology generations and at the same time improves accessibility of agent-based methods and tools to the object-oriented developers' community, which, as per today, prevails in industry. One of the first methodologies for the development of BDI agents based on OO technologies was presented in Kinny et al. (1996b) and Kinny and Georgeff (1995a, b, 1996b). The agent methodology distinguishes between the *external viewpoint*—the system is decomposed into agents, modeled as complex objects characterized by their purpose, their responsibilities, the services they perform, the information they require and maintain, and their external interactions—and the *internal viewpoint*—the elements required by a particular agent architecture must be modeled for each agent, i.e. an agent's beliefs, goals, and plans. MESSAGE (Methodology for Engineering Systems of Software Agents) (MESSAGE web site: http://www.eurescom.de/public/projects/P900-series/p907/; Caire et al., 2001) is a methodology which builds upon best practice methods in current software engineering such as for instance UML for the analysis and design of agent-based systems. It consists of (i) applicability guidelines; (ii) a modeling notation that extends UML by agent-related concepts (inspired, e.g. by Gaia); and (iii) a process for analysis and design of agent systems based on RUP. The MESSAGE modeling notation extends UML notation by key agent-related concepts. *Tropos* (Fausto Giunchiglia et al.; Tropos web site http://www.cs.toronto.edu/km/tropos/; Mylopoulos et al., 2001) is another good example of an agent-oriented software development methodology that is based on object-oriented techniques. In particular, Tropos relies on UML and offers processes for the application of UML mainly for the development of BDI agents and the agent platform JACK (Busetta et al., 1999). Some elements of UML (like class, sequence, activity and interaction diagrams) are adopted as well for modeling object and process perspectives. The concepts of *i** (*i** web site: http://www.cs.toronto.edu/km/istar/) such as actor (actors can be agents, positions or roles), as well as social dependencies among actors (including goal, soft goal, task and resource dependencies) are embedded in a modeling framework which also supports generalization, aggregation, classification, and the notion of contexts (Castro et al., 2002). Similar to Tropos, Prometheus (Padgham and Winikoff, 2002b; Prometheus home page: http://www.cs.rmit.edu.au/agents/SAC/methodology.shtml; Padgham and Winikoff, 2002a) is an iterative methodology covering the complete software engineering process and aiming at the development of intelligent agents using goals, beliefs, plans, and events, i.e. in particular BDI agents, resulting in a specification which can be implemented with JACK (Busetta et al., 1999). The Prometheus methodology

covers three phases, namely those of System specification, architectural design, and detailed design. Multiagent Systems Engineering (MaSE) (Mark and Wood Scott, 2001; Wood; DeLoach and Wood, 2000) has been developed to support the complete software development lifecycle from problem description to realization. It offers an environment for analyzing, designing, and developing heterogeneous multi-agent systems independent of any particular multi-agent system architecture, agent architecture, programming language, or message-passing system. It takes an initial system specification, and produces a set of formal design documents in a graphical style. In particular, MaSE offers the ability to track changes throughout the different phases of the process. Process for Agent Societies Specification and Implementation (*PASSI*) (PASSI website: www.csai.unipa.it/passi; Cossentino and Potts, 2002) is an agent-oriented iterative requirement-to-code methodology for the design of multi-agent systems mainly driven from experiments in robotics. The methodology integrates design models and concepts from both object-oriented software engineering and artificial intelligence approaches. PASSI is supported by a Rational Rose plug-in to have a dedicated design environment. In particular, automatic code generation for the models is partly supported and a focus lies on patterns and code reuse.

3. UML 2.0 and MDA: their usage for agent-based systems

3.1. UML

UML has a long history and is the result of a standardization effort on different modeling languages (like Entity-Relationship-Diagrams, the Booch-Notation, OMT, OOSE), namely UML. The most popular versions of UML are UML 1.x, but now UML 2.0 is the upcoming new specification for development of systems. The UML is a standard modeling language for visualizing (using the standardized graphic UML notations), specifying the static structure, dynamic behavior and model organization as well as constructing system, by mapping UML to programming environment and generate some code automatically, and documenting every phase of the lifecycle from analysis and design through deployment and maintenance. UML consists of a notation, describing the syntax of the modeling language and a graphical notation, and a meta model, describing the semantics of UML, namely the static semantics of UML, but no operational semantics. However, UML defines no software process, since a software process describes the development activities, dependencies of these activities and how they are applied. Thus UML is not a software methodology/method since a methodology consists of a modeling language *and* software process. But UML can be applied by several methodologies.

The UML 2.0 specification (for details see Upcoming UML 2.0 Standard: http://www.omg.org/uml, note that Upcoming UML 2.0 Standard: http://www.omg.org/uml is also the standard reference for this section) consists of the *Infrastructure Specification*, defines foundational language constructs required for UML 2.0. The achieved results are adjustment between UML, MOF and XMI, restructuring of language definition (meta-model as well as notation) with the goal to increase understandability and extensibility and first class extensibility mechanisms; *Superstructure Specification*, defines user-level constructs (diagrams) required for UML 2.0. The achieved results here are modeling of patterns, e.g. component-based development, specification of run-time architectures; support for scalability and encapsulation as well as unique definition of semantics for relations, like generalization, dependencies and associations; *Object Constraint Language* (*OCL*) a formal language used to describe expressions on UML models. The achieved results are meta-model-based definition of OCL, increased expressability of OCL and a formal semantics of OCL; *Diagram Interchange* enables a smooth and seamless exchange of documents compliant to the UML standard between different software tools. UML 2.0 supports the following diagrams: class, object, component, deployment and composite structure diagrams for modeling the static aspects of the systems and use case, state machine, sequence, activity, interaction overview, timing and communication diagrams for modeling dynamic aspects and packages, models and subsystems for modeling the model management. We focus on the first two groups of diagrams defined in the Superstructure Specification. We will use this distinction to present the diagram types and how they can be applied for modeling agent-based systems.

3.1.1. Structural diagrams—static aspects

3.1.1.1. Class diagram. A *Class Diagram* describes on the one side a data model, i.e. collection of declarative (static) model elements, like classes and types, and on the other side their contents and relationships. Moreover, the static structure of the system to be developed and all relevant structure dependencies and data types can be modeled with class diagrams. They are applied in various phases of the project, e.g. analysis (conceptual modeling of the domain), design (platform-independent description) of the implementation, detailed design (platform-specific model—PSM) and to bridge the gap to the behavior diagrams. Class diagrams describe classes and interfaces with their attributes and operations, as well as associations between them (including aggregation and composition), but also generalization

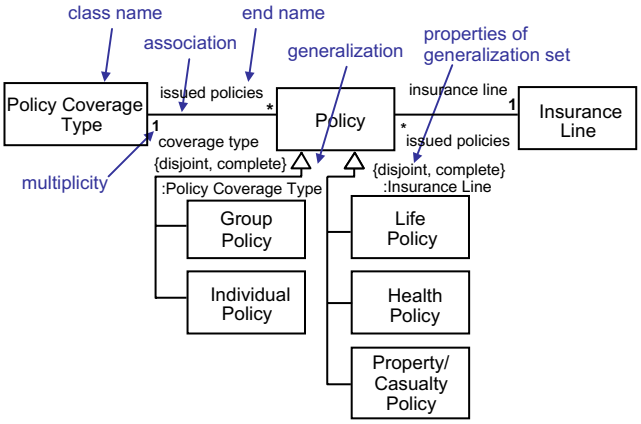

Fig. 1. Class diagram.

(a specific kind of inheritance) and dependencies among them. New to UML 2.0 is that attributes have ordering, graphical notations for associations are defined, graphical interface notation are introduced using lollipops, some unification on the notations, e.g. visibility, names and types has been done. Moreover, attributes have no implicit composition associations and dependencies are completely redefined. An example of a class diagram is illustrated in Fig. 1.

Class diagrams can be used for the definition of *organizational models*; in particular, the static aspects of the organization, its associations and part-of relationships can be modeled. For example departments can be modeled using classes. Their characteristics can be modeled by attributes, whereas their services can be modeled using functions. Associations are used to define relationships between different organizational units. The description of, e.g. subdepartments can be modeled using aggregation and composition, where aggregation and composition are special forms of associations, namely a whole–part relationship (aggregation) and an aggregation that requires that a part instance be included in at most one composite at a time (composition), respectively. Generalization is used for refining existing organizational structures. Dependencies are applied to define a relationship between, e.g. two organizational units, in which a change to one modeling element will affect the other modeling element. Similar s*ocial structures* are, e.g. defined by Odell et al. using class diagrams, see Odell et al. (2002). However, many modelers are now using composite structures to model social structures such as groups, organizations, and roles (see Composite Structure Diagram subsection). *Ontologies* can also be defined with UML. An ontology as a whole is viewed as a package, whereas a class represents, e.g. the class concept of DAML/OWL. The class hierarchy of OWL is defined using generalization, whereas properties are modeled by attributes, associations and classes. Subproperties can be defined by generalization between stereotyped 〈〈property〉〉 classes. sameClassAs, samePropertyAs are modeled by stereotyped dependencies between two classes, and associations, respectively. Cardinalities are applied to define minCardinality and maxCardinalities. For details we refer to Iglesias et al. (1998). Thus *organizational model knowledge* can be modeled by using class diagrams for the definition of ontologies. Moreover, class diagrams can be applied for the definition of *subtasks* and *subgoal* hierarchies (using generalization) as well as to define the structural aspects of tasks (using aggregation and composition). In additional constraints like goals, control features, services (functions as interfaces) can be added via attributes, functions and associations. An *agent model* can be defined using class names, inheritance (generalization) of classes and adding name, type, position/role, capabilities and constrains, either directly or via associations. A *role hierarchy* can be defined using generalization. However, roles cannot be modeled in the necessary detail with any UML 2.0 diagram. *Service models* can also be done by this diagram type, e.g. defining services with input/output parameters and pre/post-conditions as classes with attributes and functions (the service interface).

3.1.1.2. Object diagram. An *Object Diagram* describes a snapshot of the system at a specific time point, where objects and their relationships at a point in time are depicted. Object diagrams are a special case of a class diagram or a communication diagram, since objects are an instance of a class, where a link is an instance of an association and the values of attributes or simple objects. An Object Diagram consists of objects, links and values. Changes in UML 2.0 are that 〈〈copy〉〉 and 〈〈become〉〉 are obsolete and multi-object notation is obsolete. An example of an object diagram is depicted in Fig. 2.

They can be used for the definition of objects, like speech acts viewed as messages send between agents, handled by agents. Moreover, agents as instances of agent classes can be modeled to describe an agent population during the run-time execution of an agent-based system.

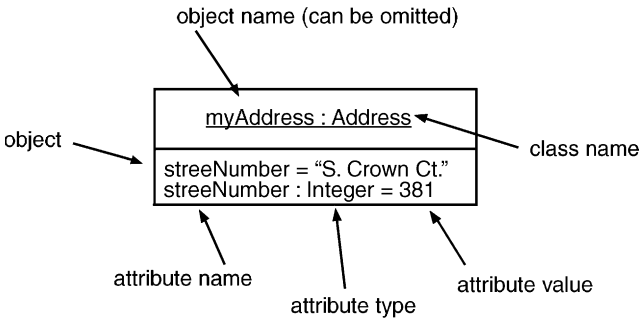

Fig. 2. Object diagram.

3.1.1.3. Composite structure diagram. A *Composite Structure Diagram* describes the internal structure of a classifier, including the interaction points of the classifier to other parts of the system. It shows the configuration of parts that jointly performs the behavior of the containing classifier. Moreover, it defines a set of instances playing parts (roles), as well as their required relationships given in a particular context (architecture). Therefore, the external interfaces are given and an abstraction of operations and signals is performed. It shows how the different architecture components are structured and interworking. They are applied during top-down modeling of the system, to model the relationship between parts of the system through specific interfaces (ports) in a precise manner. They can also be used to describe the architecture of the system (architecture diagram) and for specification and application of patterns. Since this diagram in newly introduced in UML2.0, we will have a closer look at it before showing the application for agent-based systems. An example of a composite structure diagram is contained in Fig. 3.

A *part* is an element representing a set of instances that are owned by a containing classifier instance or role of a classifier. Parts may be joined by attached connectors and specify configurations of linked instances to be created within an instance of the containing classifier. Parts are usable in class, object, component, deployment and package diagrams. A *connector* is a link that enables communication between two or more instances. The link may be realized by something as simple as a pointer or by something as complex as a network connection. A *port* specifies a distinct interaction point between a classifier and its environment or between the (behavior of the) classifier and its internal parts. Ports are connected to other ports through connectors through which requests can be made to invoke the behavioral features of a classifier. External interfaces are specified as indicated in Fig. 4.

Applied to the agent-based modeling, this diagram can be used for modeling an organization, its dependencies and workflows between agents. In addition, the diagram can represent an organization's external interfaces, as well as the internal behavior and interfaces of an agent. The notion of interface for agent-based systems is of course different to usual object-oriented systems. For agent-based systems the interface defines the speech acts understood by the agent as well as the actions performed by an agent. Furthermore, this diagram type allows us to express collaboration collaborations. The specification of how an operation or classifier, such as a use case, is realized by a set of classifiers and associations playing specific roles used in a specific way. The collaboration defines an interaction among roles, as illustrated in Sale and BrokeredSale collaborations in Fig. 5.

This diagram type can be used for defining those agent patterns that can be instantiated in different contexts, such as a typical agent broker architecture, or negotiation pattern. Moreover, it allows to define the architecture of an agent-based system and how a given agent architecture can be instantiated in different contexts. Composite structure diagrams provide a useful way to represent social structures such as groups and roles. A *group* is a set of agents that are related via their roles, where these links must form a connected graph within the group. Another way to look at this is that a group is a composite structure consisting of inter-related roles, where each of the group's roles has any number of agent instances. This definition implies not only that a group is a function of the roles contained within it, but also that roles have no meaning without their group referent. Hence, our ability to understand roles is limited by our ability to understand the groups of which they are a part.

A group can be formed to take advantage of the synergies of its members, resulting in an entity that enables products and processes that are not possible from any single individual. As with roles, groups may be deliberately established (i.e. by a system designer) or they may be emergent. In human organization terms, a

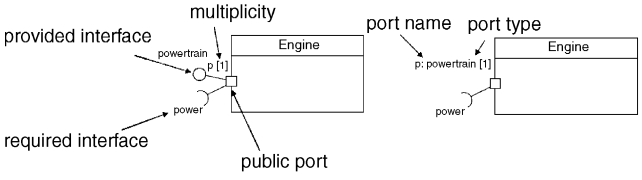

Fig. 4. Composite structure diagram—ports.

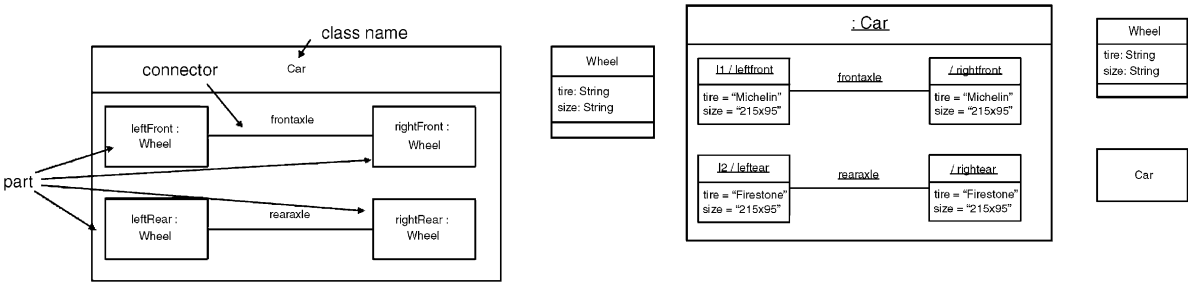

Fig. 3. Composite structure diagram and its instantiation.

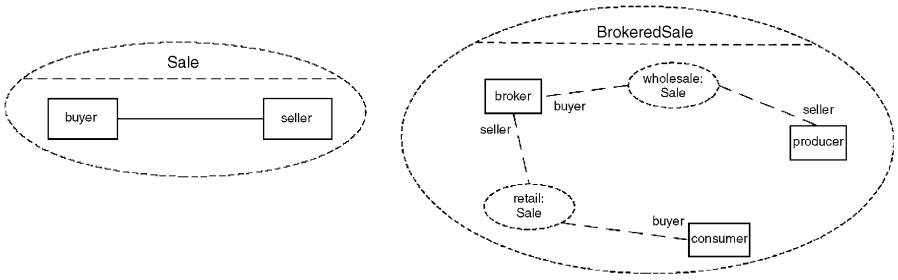

Fig. 5. Collaborations.

deliberately established group could be a department or other workgroup that has been defined by some organizational authority. In contrast, an emergent group might be a social group that forms when several individuals decide to go out for a beer after work. Over time, they define themselves as a group (My Friday Afternoon Drinking Buddies).

Groups are commonly formed to regulate, foster, or support the interaction of those agents *within* the group; so the group provides a place for a limited number of agents to interact among themselves via roles. In this way, intra-group associations encourage resource sharing, promote internal coordination, establish common supervision, and provide a degree of safety in numbers.

Groups can be treated as either agents or objects. An *Agentified Group* possesses all the features that any agent might possess. For example it can send and receive messages directly and take on roles. Such a group is an agent in its own right, and therefore is a subclass not only of Group but also of Agent (such groups can also be referred to as *organizations*). In contrast, Non-Agentified Groups are still first-class entities; however, these entities do not possess agent properties. Thus, they are as objects, rather than agents.

Fig. 6 represents the Group "ABC Ltd." as a composite structure with three associated roles, "Manager", "Broker" and "ABC Buyer". The "Manager" interacts directly with the "ABC Buyer" and the "Broker". In this situation, it is possible to interact with the agent "ABC Ltd." without knowing directly about any specific "Manager", "Broker" or "ABC Buyer" within the department; thus, this group is Agentified. The stereotype $\langle\langle$agent$\rangle\rangle$ indicates that the group is Agentified.

Groups can also be formed simply to establish a set of agents for purposes such as intra-group synergies or conceptual organization. A *Non-Agentified Group* is a Group that is not a subclass of Agent. Fig. 7 shows a Non-Agentified version of "ABC Customer Sales Dept". It has the same associated Roles; however, it does not have the $\langle\langle$agent$\rangle\rangle$ stereotype. In order to interact with this Department, you must interact directly with one of its members: a "Manager", an "ABC Buyer" or a "Broker".

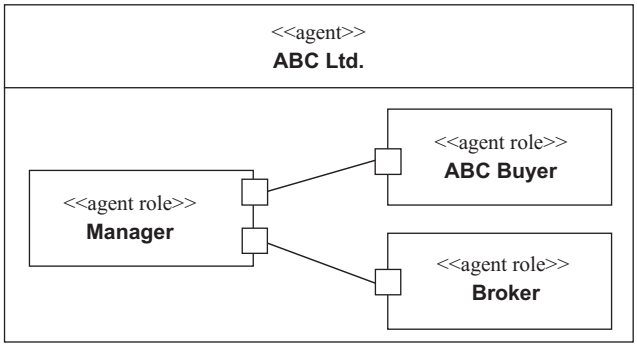

Fig. 6. Example of the ABC Ltd. Agentified group and its associated roles.

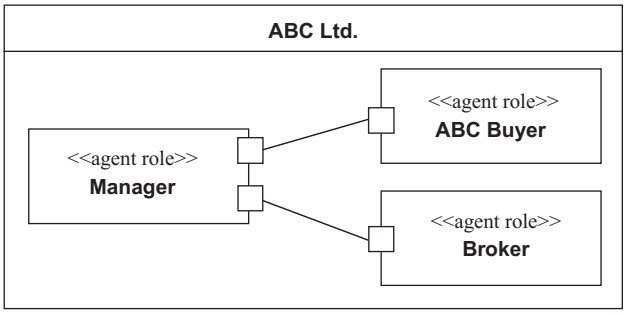

Fig. 7. Example of the Non-Agentified ABC Ltd.

3.1.1.4. Component diagram. A *Component Diagram* describes the organizations and dependencies among components. A component is a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. A Component Diagram is applied to support self-containment of components, the exchangeability of components and the distributed development and assembling of components. In particular information hiding of internal structures of components is supported. Therefore it describes components, interfaces, ports as well as the realization, implementation and usage relationships with its classes and artifacts. In UML 2.0 implementation is called manifesting with the stereotype $\langle\langle$manifest$\rangle\rangle$. Components are represented

completely different and artifacts can be associated with packaged elements. Moreover, a deployment specification can be given and new stereotypes ⟨⟨device⟩⟩, ⟨⟨execution environment⟩⟩ and ⟨⟨subsystem⟩⟩ were introduced. A component is viewed as a specific class, as depicted in Fig. 8.

In particular, interfaces of a component can either be a black-box and white-box representation (see example in Fig. 9).

An artifact is a physical piece of information that is used or produced by a development process. Examples of artifacts include models, source files, scripts, and binary executable files. An artifact may constitute the implementation of a deployable component (Fig. 10).

Component diagrams can be used to define the input/output behavior of tasks and for the decomposition of the system architecture. Moreover, the black- and white-box notation allows defining private and public interfaces of agents. The manifest-stereotype can be used to show how an agent component is deployed in distinguished systems.

3.1.1.5. Deployment diagram. A *Deployment Diagram* describes the execution architecture of systems and the system architecture. System artifacts are represented as nodes, which are connected through communication paths to create network systems of arbitrary complexity. Nodes represent run-time computational resources, which generally have at least memory and often processing capability. Run-time objects and components may reside on nodes. Nodes are typically defined in a nested manner, and represent either hardware devices or software execution environments. Deployment diagrams are applied to show the run-time environment of a system and to represent "software server" as well as to describe the distribution of components. UML 2.0 adds new elements: device, execution environment, and deployment specification. Moreover, nodes can be defined in more detail and the implement relationship is substituted by ⟨⟨manifest⟩⟩ relationship. Artifacts are implementations of any packageableElement (see Fig. 11).

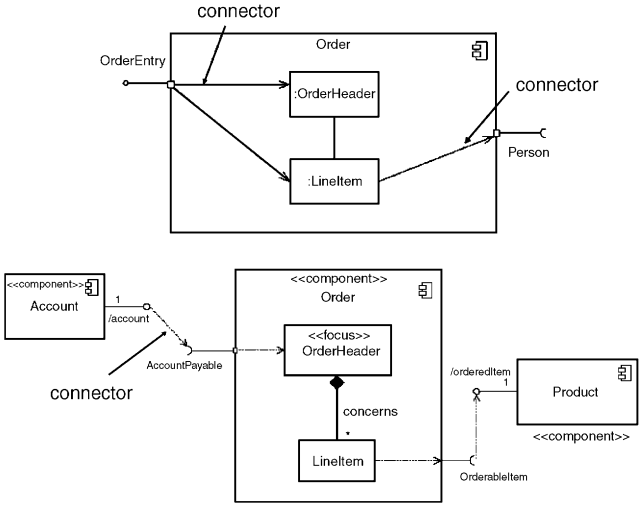

Fig. 8. Component diagram.

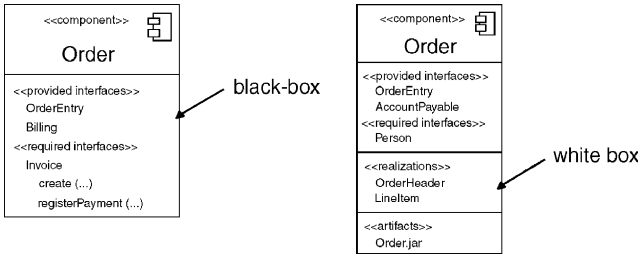

Fig. 9. Component diagram—black and white box.

A deployment specification specifies a set of properties that specify the execution parameters of a component artifact that is deployed on a node and can be aimed at a specific type of container. An artifact that reifies or implements deployment specification properties is a deployment descriptor (Fig. 12).

Deployment diagrams can be used to describe the physical distribution of agent instances; in particular they can be applied for defining the migration of agents. Moreover, this specification defines the platform design of an agent-based system. Having, e.g. generic agents, the deployment specification can be applied to define the customization of agents in a specific context.

3.1.2. Behavioral diagrams—dynamic aspects

3.1.2.1. Use case diagram. Use cases are a means for specifying required usages of a system. Typically, they are used to capture the requirements of a system, i.e. what a system is supposed to do. The key concepts associated with use cases are actors, use cases, and the subject. The subject is the system under consideration to which the use cases apply. The users and any other systems that may interact with the subject are represented as actors. The required behavior of the subject is specified by one or more use cases, which are defined according to the needs of actors. Use case diagrams are applied to define the external viewpoint on the system and to support encapsulation. In particular, they define the "what" instead of "how" a system is realized from the perspective of an external communication partner. Thus it describes the system, the use cases of a system, external actors and their relationships between actors and use-cases, between actors and between use-cases.

Agent-based systems can also use these same notions of actor, use case, and subject, as illustrated in Fig. 13. Some embellishments, however, have been made to this diagram. First, these include the events from the requesting actor to which the subject must respond (sometimes referred to as percepts) and the events that

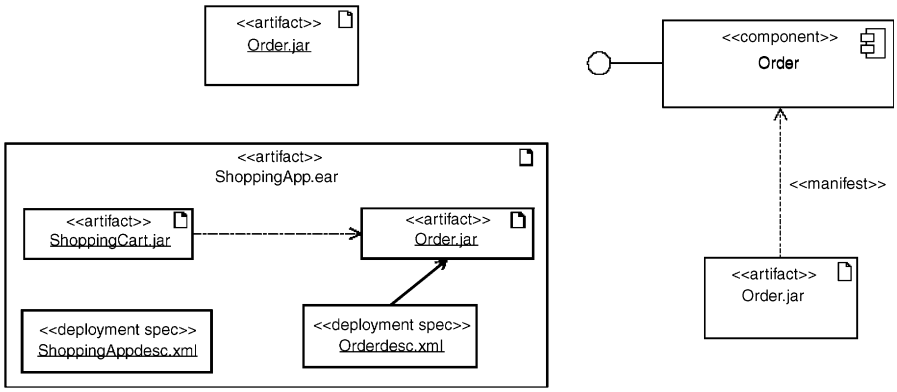

Fig. 10. Component diagram examples.

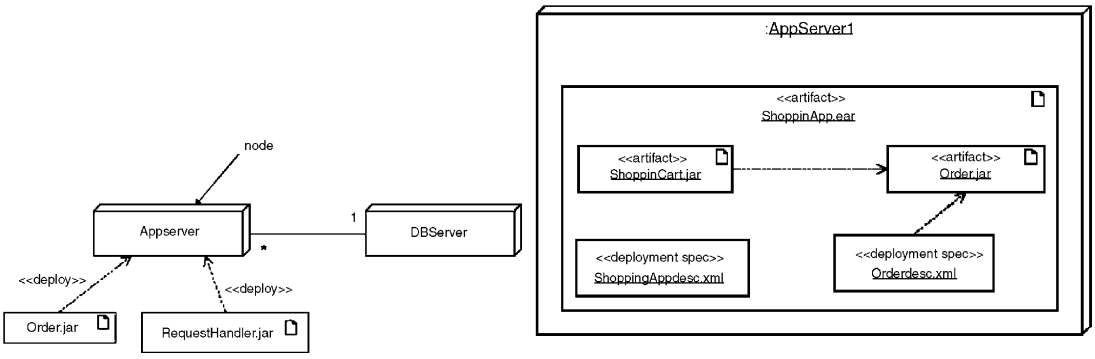

Fig. 11. Deployment diagram and complex node.

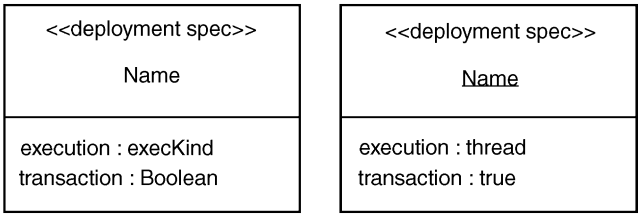

Fig. 12. Composite structure diagram—ports.

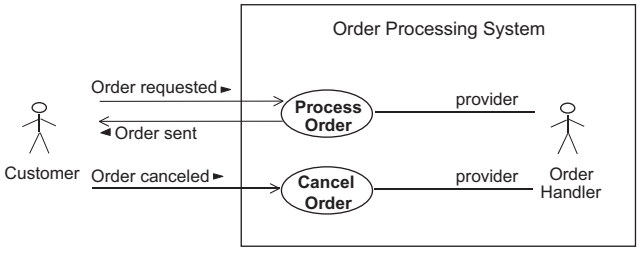

Fig. 13. A use case diagram for an order processing application.

affect an actor in some way (sometimes referred to as actions). These are indicated as names on the associations between actors and use cases. The associations also indicate directionality for these events. The second

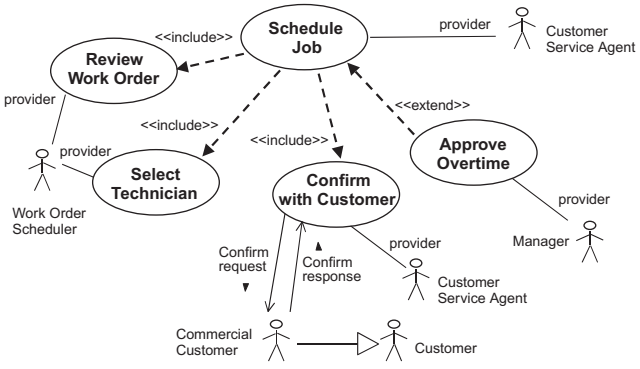

Fig. 14. A job scheduling use case diagram with include and extend relationships.

graphical addition indicates the providing actor for the service defined by the use case. Both the requesting actor and providing actor are vital to the service-oriented approach. In particular, the metamodel for the W3C's Web Services Architecture [W3C, 2004] defines both of these notions in terms of agents. For example the Customer actor is the requesting agent for the Process Order Service and the Order Handler actor is the providing agent. Fig. 14 illustrates some of these same ideas for Use Case Diagrams that represent directed relationships.

Such association, therefore, may include multiplicities, end names, association name, and so on. While most OO modelers might not take advantage of these features, they are useful for agent-based development. One minor extension to the Use Case diagram is the change in the definition of actor. UML 2.0 defines an actor as being "played by an entity that interacts with the subject…but which is *external* to the subject." Since agent-based actors can request and provide services inside or outside the subject area, the definition needs to be changed to indicate that an actor is "played by an entity that interacts with the subject's use case…but which is external to the use case." In other words, an actor may interact with the subject, or within the subject—and therefore can be internal or external to the subject." This definitional extension enables an agent-based approach to application development. Fig. 15 depicts two internal actors for a Bus Transportation System: Bus Driver and Bus Payment Machine. The Bus Driver is the actor that provides the general Bus Service. However, an Obtain Payment use case is included in the Bus Service which is provided by a different actor.

In UML 2.0, an actor "specifies a role played by a user or any other system that interacts with the subject. (The term "role" is used informally here and does not necessarily imply the technical definition of that term….)." An actor, then, can represent a single role, such as the Customer or Manager actors in the figures, above.

Internal actors might also become requestors for services from external actors. For example in Fig. 16, the Shipment Inquiry Interface actor requests a Shipment Search service from the Tracking Inquiry agent. While no association has been drawn directly from the Shipment Inquiry Interface actor and the Tracking Inquiry agent, it is implied. First, the Shipment Inquiry Interface actor is the provider for the determine Shipment Interface service; therefore, this actor is in charge of requesting services to support it. Second, the request is for the Shipment Search service; therefore, the request is drawn to the service directly. Since the service is provided by the Tracking Inquiry actor, the link to the actor is implied. While this approach may seem non-standard *in practice*, it does not violate the UML 2.0 metamodel. Furthermore, to support the service-oriented approach, clearly specifying the requested service would seem to provide better clarity than just drawing an association to the external agent. Groups of actors and functionality can be expressed using Composite Structure Diagrams.

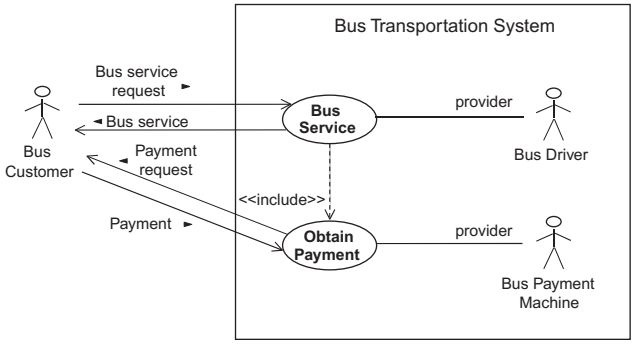

Fig. 15. An example when internal actors can be providing and/or requesting actors.

3.1.2.2. Activity diagram. Activity modeling emphasizes the sequence and conditions for coordinating lower-level behaviors. These are commonly called control flow and object flow models. The actions coordinated by activity models can be initiated because other actions finish executing, because objects and data become available, or because events occur external to the flow. For agents, all these conditions are useful. However, from the agent standpoint, additional UML 2.0 features are also practical. For example in the following figure (Fig. 17), the Activity Diagram represents a business process that an agent system might support. By its title, it suggests that this is a plan for the Process Order service.

Each plan can be expressed as an Activity diagram. However, UML 2.0 needs to be extended to define plan

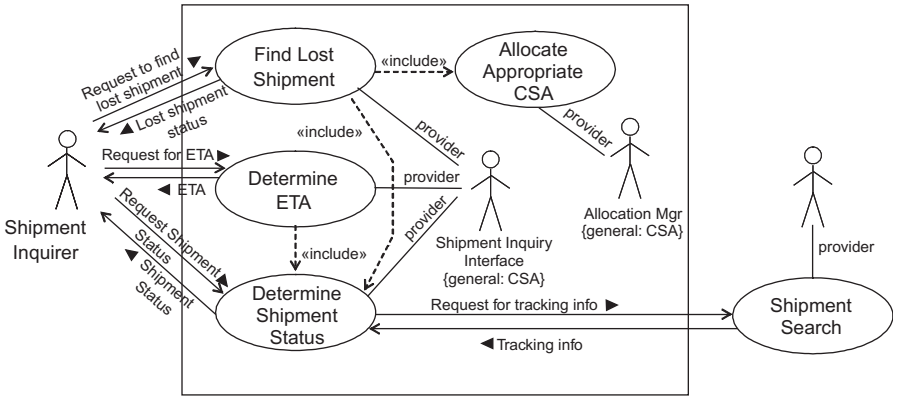

Fig. 16. Indicating actor roles and resources.

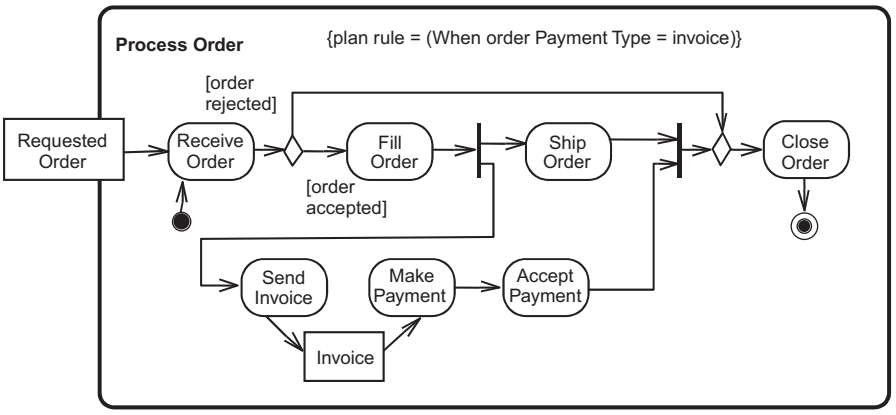

Fig. 17. An activity diagram for a process order service.

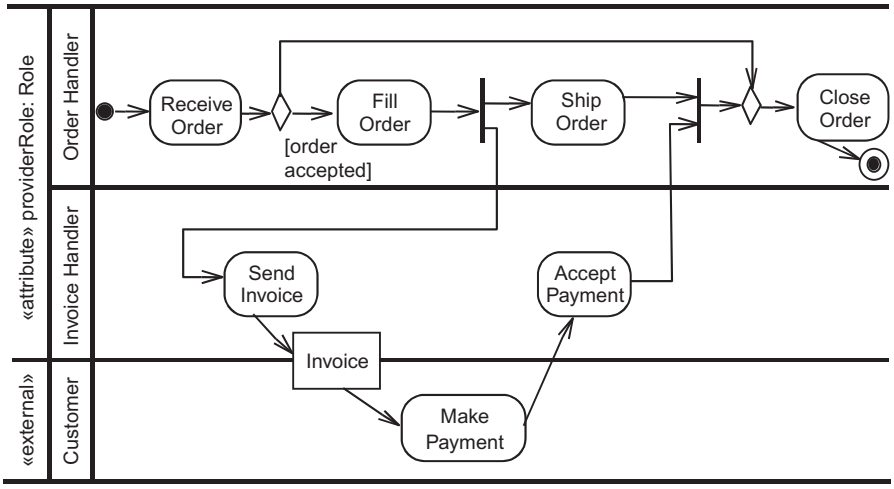

Fig. 18. An activity diagram with role-based swim lanes.

rule conditions. As mentioned earlier, the *plan rule* specifies those conditions under which the associated activity may be invoked. In the example below, the plan rule condition indicates that when Process Order service is requested for an order is to be invoiced, this particular Activity Diagram plan is executed. Different Activity Diagrams may specify alternate plans for Process Order based on, say, credit card or cash payment instead. It should be noted that to support the ability of a process (i.e. service or goal) to choose from multiple plans requires an extension to UML 2.0. Currently, UML 2.0 can only invoke a single Activity Diagram for a given process. For implementations of BDI planning systems, such as Agentis [Agentis Software, http://www.agentis-software.com], activity-based plans are typically simplified by removing branching conditions and replacing them with finer-grain plans. This normalization process can be accomplished during design time and does not require an extension to UML 2.0.

Indicating the role for the processes *within* an Activity Diagram is also useful. In UML 2.0, Activity Diagrams can represent this in two ways: partitions and annotated processes. In the diagram below, partitions for Order Handler and Invoice Handler roles are represented as swimlanes (Fig. 18).

However, graphical swimlanes are not always that clearest to express partitioning. In UML 2.0, each process on an Activity Diagram can be notated individually with the appropriate designation. For example in the diagram below, the Order Handler and Invoice Handler roles are placed within each processes' round-cornered rectangle (Fig. 19).

Group can be depicted as partitions and annotated processes in UML 2.0, as well. However, both role and group associations need to be added to the UML 2.0 metamodel to use the partitions and annotated processes. Within UML 2.0, no notion of goal, per se, exists. However, two ways of think-about goals for Activity Diagrams are supported. First, the *activity final* node (the bulls eye) is considered the goal for the activity, because it is the end point for the process. Second, at a more macro level, the service can be

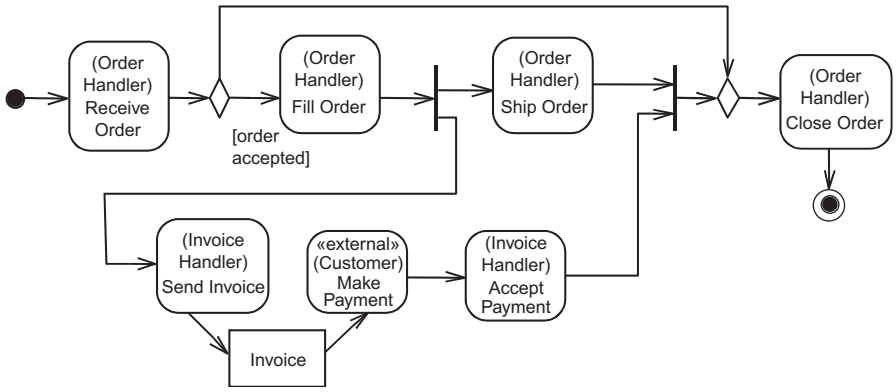

Fig. 19. An activity diagram with role-based annotations.

thought of as supporting a goal. For example the Process Order activity can be thought of as providing a service that supports the goal of processing orders. For example in Agentis, a Process Order goal is synonymous with being able to provide a Process Order service. The only other issue that has been identified involves the control nodes in the flow (decision/merge nodes and fork/join nodes). While the semantics of the control node is understood, the responsibility for its underlying processing is not. If the Activity Diagram has a "control" agent that coordinates the process flow, each Activity Diagram can be associated with a providing role, as mentioned earlier. Another option is that each of the control nodes could be associated with a role that provides the control node functionality within the activity context (as indicated in the example, above). Both options may also be chosen, where a control agent is responsible for coordinating the overall process flow by delegating to more specialized roles to handle each control node—just as it could for the individual processes in the diagram. Both options are also supported by UML 2.0.

3.1.2.3. State machine diagram. A State Machine Diagram describes the discrete behavior modeled through finite state transition systems. The sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions can be modeled. State Machine Diagrams are applied for the state descriptions of, e.g. classifiers, for detailing of use cases, for behavior description of interfaces and ports, for detailed descriptions of event and signal handling. They describe states (simple, composite, submachine states), transitions, state machine, regions, initial and final state and pseudostates. In UML 2.0 interfaces can posses protocol state machines, state entry and exit and termination can be formulated and rules for transitions in inherited state machines are added and updated.

UML 2.0 distinguishes between *behavioral state machines*, i.e. state machines can be used to specify behavior of various model elements. For example they can be used to model the behavior of individual entities (e.g. class instances). The state machine formalism described is an object-based variant of Harel statecharts; and Protocol State machines, i.e. *Protocol state machines* are used to express usage protocols. Protocol state machines express the legal transitions that a classifier can trigger. The state machine notation is a convenient way to define a lifecycle for objects, or an order of the invocation of its operation. Because protocol state machines do not preclude any specific behavioral implementation, and enforces legal usage scenarios of classifiers, interfaces and ports can be associated to this kind of state machines.

The State Machine Diagram provides a graphical way representing discrete behavior through finite state transition systems (Fig. 20).

State machine can be applied for modeling interaction protocols, similar to sequence diagrams and model plans.

3.1.2.4. Sequence diagram. The most common kind of Interaction Diagram is the Sequence Diagram, which focuses on the message interchange between a number of lifelines. In particular, a Sequence Diagram describes an Interaction by focusing on the sequence of messages that are exchanged, along with their corresponding event occurrences on the lifelines. In particular, time sequences but does not include object relationships. This can be done either in a generic form (describes all possible scenarios) or in an instance form (describes one actual scenario). Sequence diagrams are applied to model interactions and in various phases of the software development process (e.g. use case refinement, modeling of test scenarios, communication model, detailed modeling of message exchanges or specification of interfaces).

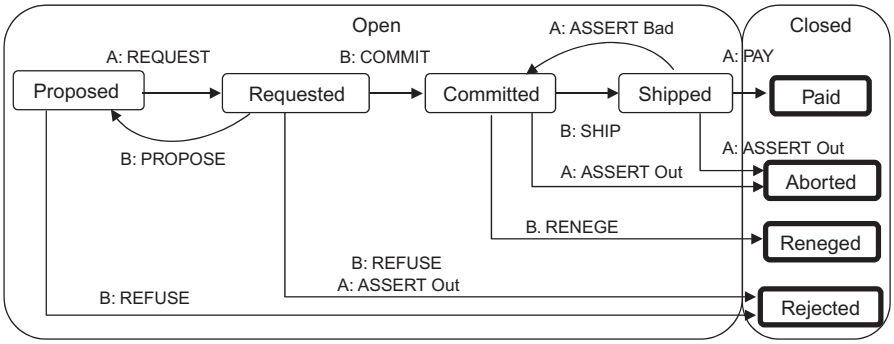

Fig. 20. A state machine diagram for a process order service.

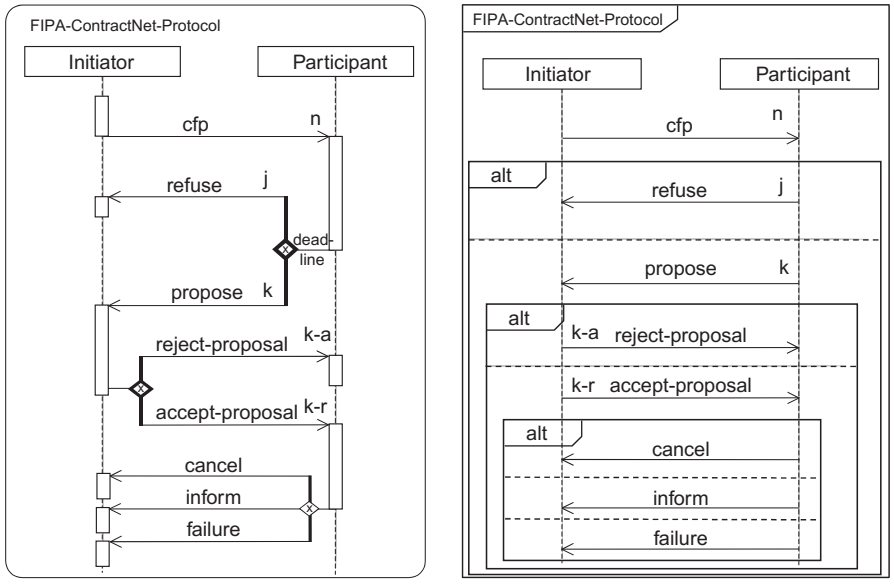

Fig. 21. (a) UML 1.x agent extensions and (b) UML 2.0 sequence diagrams.

Prior to UML 2.0, FIPA defined an agent-based extended UML 1.x to include roles, decision points, concurrency, modularity, and multicasting (example in Fig. 21(a)). UML 2.0 includes representation of all these notions except role and multicasting support. UML 2.0 adds, e.g. loops, alternatives, parallelism, sequences and critical fragments.

Role notation can initially include making each lifeline define a role. The current UML 2.0 metamodel is not far from this general concept, but the agent-based notion of role is not defined by UML 2.0. Therefore, the extension to the Sequence Diagram metamodel needs to include equate the lifeline with a metamodel notion of role. Furthermore, some agent-based applications involve dynamic and multiple classification of agents in their roles. In these kinds of applications, representing role change and multiple roles for an agent requires more research and extended notation. Representing groups is not part of UML 2.0, and would therefore need to be added.

To support the notation of multicast and multi-response, a cardinality-based notation was added to the message lines (Fig. 21(b)). For example the *cfp* message is annotated to indicate a message that would be multicast from an Initiator to *n* Participants. The response then involves a refusal from *j* Participants and proposal from *k* other Participants; and so on. This notation represents a first try at representing multicasting; however, it still requires more consideration. For instance, policies may need to be stated with multiple responses (e.g. the number of refusal and proposal responses may not exceed the number of multicast *cfp* invitations to participants).

3.1.2.5. Communication diagram. Communication Diagrams (formerly known as Collaboration Diagrams in UML 1.x) focus on the interaction between lifelines message passing is central. They correspond to simple Sequence Diagrams that use none of the structuring mechanisms such as interaction occurrences and

combined fragments. It also assumes a strict ordering of messages. A Communication Diagram describes the interaction between lifelines where the architecture of the internal structure and how this corresponds with the message passing is central. In particular, sequencing of messages is characterized through a sequence numbering scheme. Are known as collaboration diagrams of UML 1.x. They are applied similar to simple sequence diagrams. However, due to their limited expressiveness, Communication Diagrams can only be used to represent simple and straightforward interactions. While they remain as part of the UML 2.0 set of diagrams, their usefulness is limited (Fig. 22).

Communication Diagrams require the same role-based extensions as the Sequence Diagrams. Role notation can initially include making each lifeline define a role. The current UML 2.0 metamodel is not far from this general concept, but the agent-based notion of role is not defined by UML 2.0. Therefore, the extension to the Communication Diagram metamodel needs to include equate the lifeline with a metamodel notion of role. Furthermore, some agent-based applications involve dynamic and multiple classification of agents in their roles. In these kinds of applications, representing role change and multiple roles for an agent requires more research and extended notation. Representing groups is not part of UML 2.0, and would therefore need to be added. To support the notation of multicast and multi-response, a cardinality-based notation was added to the message lines in the same manner as sequence diagrams in Fig. 21(b). For example the *cfp* message is annotated to indicate a message that would be multicast from an Initiator to *n* Participants. The response then involves a refusal from *j* Participants and proposal from *k* other Participants; and so on. This notation represents a first try at representing multi-casting; however, it still requires more consideration. For instance, policies may need to be stated that is multiple responses (e.g. the number of refusal and proposal responses may not exceed the number of multicast *cfp* invitations to participants).

3.1.2.6. Interaction overview diagram. Interaction Overview Diagrams (IODs) define interactions through a variant of Activity Diagrams in a way that promotes overview of the control flow. IODs focus on the overview of the flow of control where the nodes are interactions. The lifelines and the messages do not appear at this overview level. For example the UML 1.x diagram in Fig. 23(a) would be graphically cumbersome to express as a Sequence Diagram in UML 2.0, because of the resulting plethora of boxes within boxes. By using the IOD, the flow can be more clearly delineated as depicted in Fig. 23(b).

The IOD contains Sequence Diagrams and therefore needs to address the extensions indicated for Sequence Diagrams (see above). The only other issue that has been identified involves the control nodes in the flow (decision/merge nodes and fork/join nodes). (This issue is similar to those that exist for the Activity Diagram.) While the semantics of control node is understood, the responsibility for its underlying processing is not. If the Interaction Overview has a "control" agent that coordinates the process flow, each IOD can be associated with a providing role. Another option is that each of the control nodes could be associated with a role that provides the control not functionality within the IOD context. Both options may also be chosen, where a control agent is responsible for coordinating the overall process flow by delegating to more specialized roles to handle each control node.

3.2. MDA

Our approach towards this end is a *model-driven development and architecture* (MDA; for details see Kleppe et al., 2003) as promoted by the OMG for the development of agent-based systems. Key to MDA is

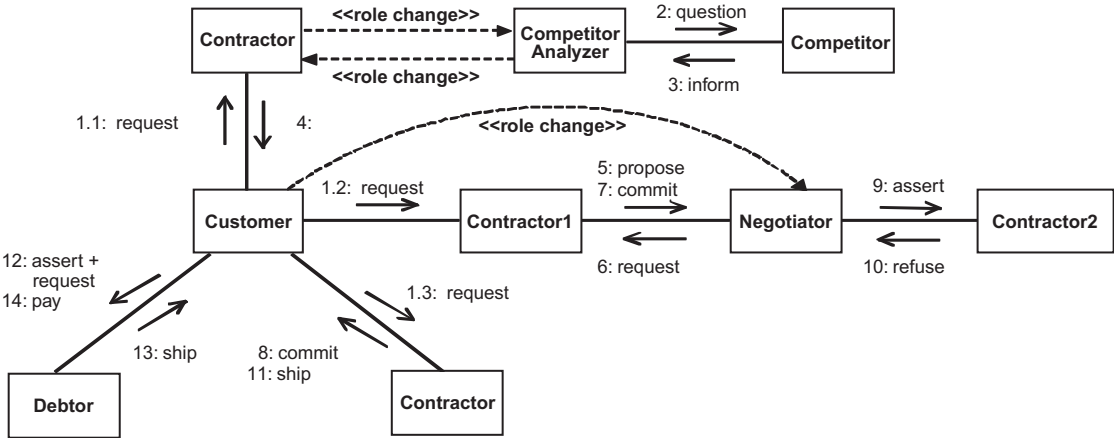

Fig. 22. A communication diagram with role-based annotations.

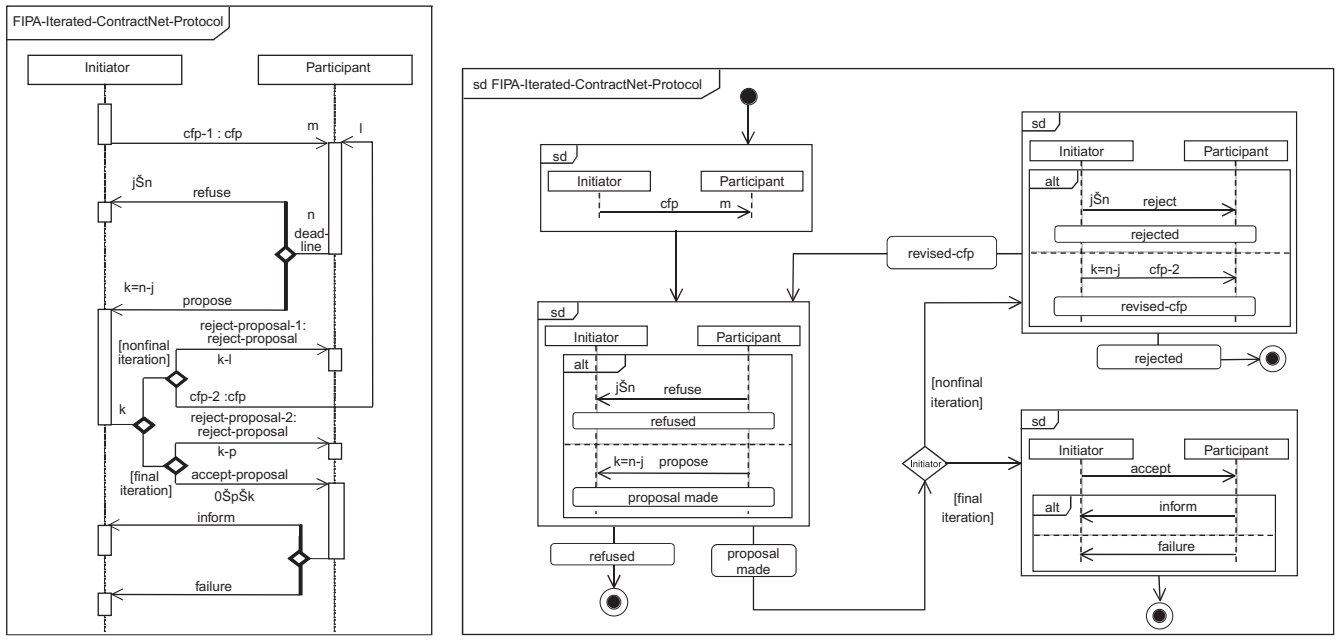

Fig. 23. UML 1.x agent extensions with looping. (a) UML 1.x agent extensions with looping. (b) A UML 2.0 IOD representation.

the importance of models in the software development process. Within MDA the software development process is driven by the activity of modeling the business software system. The MDA development process does not look very different from a traditional lifecycle, containing the same phases (requirements, analysis, low-level design, coding, testing, and deployment). One of the major differences to traditional development processes lies in the nature of the artifacts that are created during the development process. These artifacts are formal models, i.e. models that can be understood by computers and finally be transformed into a representation that lends itself to execution. The following three models are at the core of the MDA: *Computation-Independent Model* (*CIM*): This is the most abstract model within MDA which is independent of computational technology. It describes the business (logic) and therefore defines business processes and workflows in detail. *PIM*: This model is defined at a high level of abstraction; it is independent of any implementation technology. It describes a software system that supports some business. Within a PIM, the system is modeled from the viewpoint of how it best supports the business. Whether a system will be implemented on a mainframe with a relational database, on an EJB application server or on an agent-platform is irrelevant at the PIM level. *PSM:* In the next step, the PIM is transformed into one or more PSMs. It is tailored to specify a system in terms of the implementation constructs available in one specific implementation technology. A PIM is transformed into one or more PSMs. For each specific technology platform a separate PSM is generated. Most systems today span several technologies; therefore, it is common to have many PSMs with one PIM. The final step in the development is the transformation of each PSM to code. Because a PSM fits its technology rather closely, this transformation is relatively straightforward.

Summarizing the different approaches of Section 2 and the usage of UML 2.0 diagrams from Section 3.1, we distill the following necessary aspects to be covered by an MDA covering major areas of agent-based systems (only focusing on CIM and PIM):

3.2.1. *Computational-independent model*

The CIM has to deal with the following aspects: *Use Cases*: Taken from object-oriented software development, use case scenarios are a suitable method to derive the functional requirements of a system need to be derived. That is UML 2.0 use case diagrams are applied. *Environment Model*: In (Odell et al., 2002), Odell et al. consider several aspects of environment modeling ranging from physical environments to agent communication and to how their considerations could be embedded into the FIPA architecture. *Domain/Ontology Model*: This model defines the ontologies of the domain and relates them to other existing ontologies using, e.g. UML class diagrams and Semantic Web representation languages (Semantic Web Inititiative: http://www.semanticweb.org). *Role Model*: This model describes the roles in a domain, on the one hand in the traditional object-oriented sense (actor-role relationship), but also defining roles characterizing social

relationships within an agent-based system. *Goal/Task Model*: This model defines the objectives of an agent in terms of soft and hard goals, and should also support means–ends analysis (as in Tropos). Moreover, the notion of tasks and plans should be provided to support the description of agent behavior at a high level of abstraction. *Interaction Model*: This model defines the regime of interaction and collaboration among entities and groups of entities, at a level which abstracts from specific interaction protocols. *Organization/Society Model*: This model defines to a reasonable extent the real-world society and organization and hence the social context within which agents in an agent-based system act and interact. *Business process models*: The notion of business processes is key for corporate business applications. Business processes describe the means and the ends of business interactions. For agents to support corporate applications, it is important to be able to access executable definitions of business processes, to reason about the semantics of goal-directed business processes (see Agentis. http://www.agentissoftware.com.), and to relate business process to the organizational model, the interaction model, and the task model.

3.2.2. Platform-independent model

Interaction Protocol Model: This model defines the interaction between different agent class, agent instances and roles at the level of interaction protocols, such as the Contract Net. *Internal Agent Model*: This model deals in particular with goals, beliefs and plans of agent classes, how they are defined and which underlying architecture is used. *Agent Model*: This model describes the behavior of agents and agent groups, i.e. how different agent are collaborating together independent of their implementation. The interaction model defines the concrete interaction of the agents, whereas the internal agent model defines the internal behavior of an agent, e.g. in terms of BDI, and the agent model defines the behavior of an agent seen by other agents. *Service/Capability Model*: Defines the services and capabilities of agents, mostly using service description languages and mechanisms such as UDDI or DAML-S. *Acquaintance Model*: This model provides agents with models of other agents' beliefs, capabilities, and intentions. It can be used to determine suitable partners for collaboration or to predict others' behavior, e.g. in a coordination task. *Deployment/agent instance model*: This model describes which agent instances exist, the migration is considered as well as the dynamic creation of agents.

4. Open issues and conclusions

As an OMG standard, UML 2.0 is now considered a "final" standard, as of November 2004. In other words, many of the errors and inconsistencies of the original submission have been rectified. More than 3000 issues were files and resolved by the UML 2.0 Finalization Task Force. As such software vendors can begin to build software tools that support the UML 2.0 Superstructure and Infrastructure. In addition, a firmer foundation is now available to adequately support the extensions for agent-based system modeling. The FIPA Modeling Technical Committee and the OMG Agent Special Interest Group are actively working on extending UML for agent-based system modeling. These efforts are primarily supported by the work of more than a dozen software tool vendors. The notation presented in this paper is an interim result of this effort. UML has no "off-the-shelf" constructs to express: goals, agent, groups, multicasting, generative functions, such as cloning, birthing, reproduction, parasitism and symbiosis, emergent phenomena, and many other nature-based constructs that are helpful for representing agent structures. Furthermore, agent researchers are still trying to determine useful ways of representing agents and agent-based systems. As such, we cannot expect to have rich modeling languages for agents for several more years (the first OMG "agent UML" request for proposal is not scheduled to be issued until the Fall of 2005). However, we can begin to provide the agent community with guidelines for notations that provide obvious benefit—such as those presented in this chapter.

Acknowledgements

We would like to thank Gerhard Weiß for the invitation to contribute an article to this journal.

References

Bauer, B., Müller, J.P., 2004. Methodologies and modeling languages. In: Luck, M., Ashri, R., D'Inverno, M. (Eds.), Agent-Based Software Development. Artech House Publishers, Boston, London.

Bauer, B., Müller, J.P., Odell, J., 2001a. Agent UML: a formalism for specifying multiagent software systems. International Journal on Software Engineering and Knowledge Engineering (IJSEKE).

Bauer, B., Bergenti, F., Massonet, Ph., Odell, J., 2001b. Agents and the UML: a unified notation for agents and multi-agent systems, Proceeding of the AOSE 2001. Springer, Montreal.

Beck, K., 1999. Extreme Programming Explained. Addison Wesley, Reading, MA.

Brazier, F.M.T., Jonkers, C.M., Treur, J., 1998. Principles of compositional multi-agent system development. Proceedings of the 15th IFIP World Computer Congress, WCC'98, Conference on Information Technology and Knowledge Systems, IT&KNOWS'98. Chapman & Hall, London, pp. 347–360.

Burmeister, B., 1996. Models and methodology for agent-oriented analysis and design. Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems, DFKI Document D-96-06.

Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A., 1999. JACK intelligent agents—components for intelligent agents in Java. Technical Report TR9901, AOS, January, http://www.jackagents.com/pdf/tr9901.pdf.

Caire, G., Coulier, W., Garijo, F., Gomez, J., Pavon, J., Massonet, P., Leal, F., Chainho, P., Kearney, P., Stark, J., Evans, R., 2001. Agent oriented analysis using MESSAGE/UML, Proceedings of the AOSE 2001. Springer, Berlin.

Castro, J., Kolp, M., Mylopoulos, J., 2002. Towards Requirements-Driven Information Systems Engineering: The Tropos Project. Information Systems, Elsevier, Amsterdam, The Netherlands.

Cossentino, M., Potts, C., 2002. A CASE tool supported methodology for the design of multi-agent systems. Proceedings of the 2002 International Conference on Software Engineering Research and Practice (SERP'02), Las Vegas (NV), USA.

DeLoach, S.A., Wood, M.F., 2000. Multiagent systems engineering: the analysis phase. Technical Report, Air Force Institute of Technology, AFIT/EN-TR-00-02, June 2000.

Garijo, F.J., Boman, M. (Eds.), 1999. Multi-agent system engineering. Proceedings of MAAMAW'99. Springer, Berlin.

Giunchiglia, F., Mylopoulos, J., Perini, A. The Tropos software development methodology: processes, models and diagrams. Proceedings of AAMAS'02, July 15–19, 2002, Bologna, Italy.

Glaser, N., 1996. Contribution to knowledge modelling in a multi-agent framework (the Co-MoMAS approach). Ph.D. Thesis, L'Universtit′ e Henri Poincar′ e, Nancy I, France, November.

Herlea, D.E., Jonker, C.M., Treur, J., Wijngaards, N.J.E., 1999. Specification of behavioural requirements within compositional multi-agent system design. Proceedings of Ninth European Workshop on Modelling Autonomous Agents in a Multi-Agent World. Springer, Berlin, pp. 8–27.

Iglesias, C.A., Garijo, M., Gonzalez, J.C., Velasco, J.R., 1996. A methodological proposal for multiagent systems development extending CommonKADS. Proceedings of the 10th KAW. Banoe, Canada.

Iglesias, C.A., Garijo, M., González, J.C., 1998. A survey of agent-oriented methodologies. Proceedings of the Fifth International Workshop on Agent Theories, Architectures, and Languages, University Pierre et Marie Curie, pp. 185–198.

Jacobson, I., Booch, G., Rumbaugh, J., 1998. The Unified Software Development Process. Addison Wesley, Reading, MA.

Jonker, C.M., Treur, J., 1997. Compositional verification of multi-agent systems: a formal analysis of pro-activeness and reactiveness. Proceedings of the International Workshop on Compositionality (COMPOS'97). Springer, Berlin.

Kinny, D., Georgeff, M., 1995a. A design methodology for BDI agent systems. Technical Report 55, Australian Artificial Intelligence Institute, Melbourne, Australia.

Kinny, D., Georgeff, M., 1995b. Modelling techniques for BDI agent systems. Technical Report 54, Australian Artificial Intelligence Institute, Melbourne, Australia.

Kinny, D., Georgeff, M., 1996a. Modelling and Design of Multi-Agent Systems, Intelligent Agents III. Springer, Berlin.

Kinny, D., Georgeff, M., 1996b. Modelling and design of multi-agent systems. Proceedings of the ATAL 96.

Kinny, D., Georgeff, M., Rao, A., 1996a. A methodology and modelling technique for systems of BDI agents. Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96). Springer, Berlin, pp. 56–71.

Kinny, D., Georgeff, M., Rao, A., 1996b. A methodology and modeling technique for systems of BDI agents. Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW 96), LNAI 1038. Springer, Berlin.

Kleppe, M., Warmer, J., Bast, W., 2003. MDA Explained—The Model Driven Architecture: Practice and Promise. Addison Wesley, Reading MA.

Mark, F., Wood Scott, A., 2001. DeLoach: an overview of the multiagent systems engineering methodology. In: Ciancarini, P., Wooldridge, M. (Eds.), Proceedings of the First International Workshop on Agent-Oriented Software Engineering, Lecture Notes in Computer Science, vol. 1957. Springer, Berlin.

Mylopoulos, J., Kolp, M., Castro, J., 2001. UML for agent-oriented software development: The Tropos proposal. Proceedings of the Fourth International Conference on the Unified Modeling Language UML'01, Toronto, Canada, October.

Odell, J.J., Van Dyke Parunak, H., Fleischer, M., Brueckner, S., 2002. Modeling agents and their environment. Proceedings of the AOSE 2002. Springer, Berlin.

Omicini, A., 2000. SODA: societies and infrastructures in the analysis and design of agent-based systems. Proceedings of the AOSE 2000. Springer, Berlin.

Padgham, L., Winikoff, M., 2002a. Prometheus: a methodology for developing intelligent agents. Proceedings of the AOSE 2002. Springer, Berlin.

Padgham, L., Winikoff, M., 2002b. Prometheus: a pragmatic methodology for engineering intelligent agents. Proceedings of the Workshop on Agent-oriented Methodologies at OOPSLA, November 4, 2002.

Rudolph, E., Grabowski, J., Graubmann, P., 1996. Tutorial on message sequence charts (MSC. Proceedings of the FORTE/PSTV'96 Conference.

Rumbaugh, J., 1995a. OMT: the development model. Journal of Object Oriented Programming 76, 8–16.

Rumbaugh, J., 1995b. OMT: the dynamic model. Journal of Object Oriented Programming, 6–12.

Schreiber, A.Th., Wielinga, B.J., Akkermans, J.M., Van de Velde, W., 1994. CommonKADS: a comprehensive methodology for KBS development. Deliverable DM1.2a KADS-II/M1/RR/UvA/70/1.1, University of Amsterdam, Netherlands Energy Research Foundation ECN and Free University of Brussels.

Wood, M.F., 2000. Multiagent systems engineering: a methodology for analysis and design of multiagent systems. M.S. Thesis, AFIT/GCS/ENG/00M-26, School of Engineering.

Wooldridge, M., Jennings, N.R., Kinny, D., 2000a. The Gaia methodology for agent-oriented analysis and design. International Journal of Autonomous Agents and Multi-Agent Systems 3.

Wooldridge, M., Jennings, N.R., Kinny, D., 2000b. The Gaia methodology for agent-oriented analysis and design. Journal of Autonomous Agents and Multi-Agent Systems 3 (3), 285–312.