Using UML in the context of agent-oriented software engineering: state of the art

Bernhard Bauer, Jörg Müller

Angaben zur Veröffentlichung / Publication details:

Bauer, Bernhard, and Jörg Müller. 2004. "Using UML in the context of agent-oriented software engineering: state of the art." In *Agent-oriented software engineering IV : 4th International Workshop, AOSE 2003, Melbourne, Australia, July 15, 2003, Revised Papers*, edited by Paolo Giorgini, Jörg Müller, and James Odell, 1–24. Berlin u.a.: Springer. https://doi.org/10.1007/978-3-540-24620-6_1.

Using UML in the Context of Agent-Oriented Software Engineering: State of the Art

Bernhard Bauer[1] and Jörg P. Müller[2]

[1] Institute of Computer Science, University of Augsburg, D-86135 Augsburg, Germany
Bernhard.Bauer@informatik.uni-augsburg.de

[2] Siemens AG, Corporate Technology, CT IC 6, D-81730 Munich, Germany
joerg.p.mueller@siemens.com

Abstract. Most of the methodologies and notations for agent-oriented software engineering developed over the past few years are based on the Unified Modeling Language (UML) or proposed extensions of UML. However, at the moment an overview on the different approaches is missing. In this paper. we present a state-of-the-art survey of the different methodologies and notations that, in one way or the other, rely on the usage of UML for the specification of agent-based systems. We focus on two aspects, i.e., design methodologies for agent-oriented software engineering, and different types of notations (e.g., for interaction protocols, social structures, or ontologies) that rely on UML.

1 Introduction

The complexity of commercial software development processes increasingly requires the usage of software engineering techniques, including methodologies and tools for building, deploying, and maintaining software systems and solutions. In this context, software methodologies play a key role. 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 interrelationships 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. 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. Over the past year, agents have been very successful from the scientific point of view; also, the beginning commercial success of agent technology at the application level (in the sense of: intelligent components supporting intelligent applications, see e.g., [44]) is evident today. However, the potential role of agent technology as a new paradigm for software engineering has not yet met with broad acceptance in industrial and commercial settings. We claim that the main reason for this is the lack of accepted methods for software development depending on widely standardized representations of artifacts supporting all phases of the software lifecycle. In particular, these standardized representations are needed by tool developers to provide commercial quality tools that mainstream software engineering departments need for industrial agent systems development.

Currently, most industrial methodologies are based on the Object Management Group's (OMG) Unified Modeling Language (UML) accompanied by process frameworks such as the Rational Unified Process (RUP), see [28] for details. The Model-Driven Architecture (MDA [40]) from the OMG allows a cascade if code generations from high-level models (platform independent model) via platform dependent models to directly executable code (e.g., see the tool offered by Kennedy Carter [39]).

Thus, one possibility to provide an answer regarding the state-of-the-art in agent-oriented software engineering is to look at the level of support currently provided for UML technologies by recent agent-based engineering approaches. In this paper we will provide a detailed survey of methodologies and notations for agent-based engineering of software systems based on UML.

In Section 2 we will have a closer look at different methodologies for designing agent-based systems. In Section 3 focuses on notations based on UML. In particular, we shall look at notations for interaction protocols, social structures, agent classes, ontologies, and goals and plans. The paper concludes with a summary and an outlook for further research in Section 4.

2 Methodologies

In this we will take a closer look at agent methodologies that directly extend object-oriented – UML approaches. In the next section we will also give an overview of UML notations and extensions available for the specification of agent-based systems. Since most of the notations use graphical representations of software artifacts we will use examples taken from the original research papers.

2.1 Agent Modeling Techniques for Systems of BDI Agents

One of the first methodologies for the development of BDI agents based on OO technologies was presented in [2][3][4][5]. 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. For each of these views different models are described (based on [2] and [5]):

The *external view* is characterized by two models which are largely independent of the underlying BDI architecture:

Agent Model: This model describes the hierarchical relationship among different abstract and concrete agent classes (*Agent Class Model*) similar to a UML class diagram denoting both abstract and concrete (instantiable) agent classes, inheritance and aggregation as well as predefined reserved attributes, e.g., each class may have associated belief, goal, and plan models; and identifies the agent instances which may exist within the system, their multiplicity, and when they come into existence (*Agent Instance Model*) with the possibility to define *initial-belief-state* and *initial-goal-state* attributes.

Interaction Model: describes the responsibilities of an agent class, the services it provides, associated interactions, and control relationships between agent classes. This includes the syntax and semantics of messages used for inter-agent communication and communication between agents and other system components, such as user interfaces.

BDI agents are *internally viewed* as having certain mental attitudes, *Beliefs*, *Desires* and *Intentions*, which represent, respectively, their informational, motivational and deliberative states. These aspects are captured, for each agent class, by the following models.

Belief Model describes the information about the environment and internal state that an agent of that class may hold, and the actions it may perform. The possible beliefs of an agent and their properties, such as whether or not they may change over time, are described by a *belief set*. In addition, one or more *belief states* - particular instances of the belief set - may be defined and used to specify an agent's initial mental state. The *belief set* is specified by a set of object diagrams which define the domain of the beliefs of an agent class. A belief state is a set of instance diagrams which define a particular instance of the belief set. Formally, defined by a set of typed predicates whose arguments are terms over a universe of predefined and user-defined function symbols.

Goal Model describes the goals that an agent may possibly adopt, and the events to which it can respond. It consists of a *goal set* which specifies the goal and event domain and one or more *goal states* - sets of ground goals - used to specify an agent's initial mental state. A goal set is, formally, a set of goal formula signatures. Each such formula consists of a modal goal operator applied to a predicate from the belief set.

Plan Model describes the plans that an agent may possibly employ to achieve its goals. It consists of a *plan set* which describes the properties and control structure of *individual plans*. Plans are modeled similar to simple UML State Chart Diagrams, which can be directly executed showing how an agent should behave to achieve a goal or respond to an event. In contrast to UML activities may be sub-goals, denoted by formulae from the agent's goal set; conditions are predicates from the agent's belief set; actions include those defined in the belief set, and built-in actions. The latter include assert and retract, which update the belief state of the agent.

2.2 Message

MESSAGE (Methodology for Engineering Systems of Software Agents) [6][7] 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

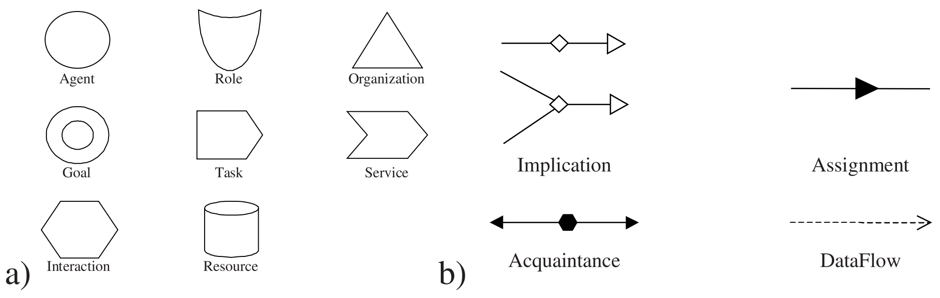


Fig. 1. a) concept symbols in MESSAGE; b) relations in MESSAGE

extends UML by agent-related concepts (inspired e.g. by Gaia); and (iii) a process for analysis and design of agent systems based on Rational unified Process. The MESSAGE modeling notation extends UML notation by key agent-related concepts. We describe the notation used in MESSAGE based on the example presented in [7]. For details on the example we refer to this paper. The used concept and relation symbols are shown in Fig. 1.

The main focus of MESSAGE is on the phase of analysis of agent-based systems. For this purpose, MESSAGE presents five analysis models, which analysts can use to capture different aspects of an agent-based system. The models are described in terms of sets of interrelated concepts. The five models are (following [7][6]):

Organization Model: The Organization Model captures the overall structure and the behavior of a group of agents and the external organization working together to reach common goals. In particular, it represents the responsibilities and authorities with respect to entities such as processes, information, and resources and the structure of the organization in terms of sub-organization such as departments, divisions, sections, etc. expressed through power relationships (e.g. superior-subordinate relationships). Moreover it provides the *social view* characterizing the overall behavior of the group, whereas the agent model covers the *individual view* dealing with the behavior of agents to achieve common/social goals. It offers software designers a useful abstraction for understanding the overall structure of the multi-agent system, what the agents are, what resources are involved, what the role of each agent is, what their responsibilities are, which tasks are achieved individually and which achieved through co-operation. Different types of organization diagrams are available in MESSAGE to support the graphical representation of social concepts (see Fig. 2).

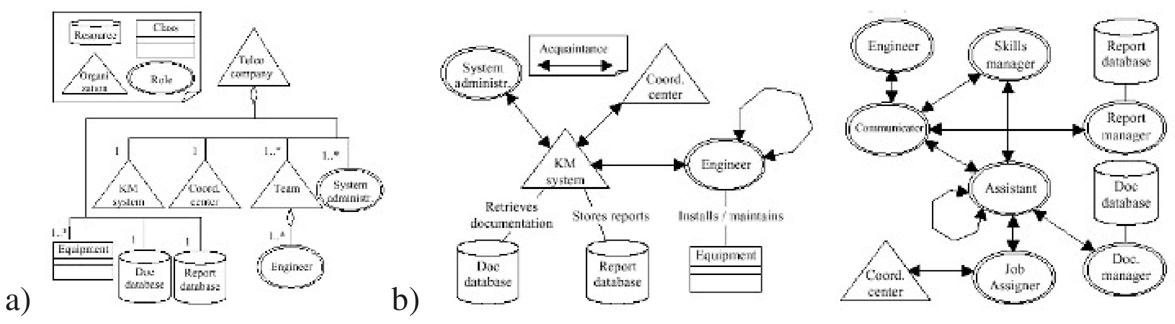
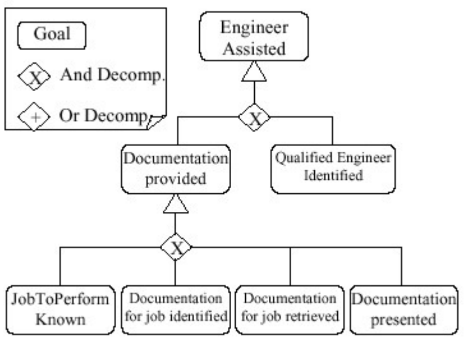


Fig. 2. Examples of organization diagrams: a) structural relationships, b) acquaintance relationships (analysis phase 0 and 1)

Goal/Task Model: The Goal/Task Model defines the goals of the composite system, i.e. the agent system and its environment, and their decomposition into sub-goals; the responsibility of agents for their commitments; the performance of tasks and actions by agents, the goals the tasks satisfy and the decomposition of tasks into sub-tasks as well as to describe tasks involved in an organizational workflow. It captures what the agent system and constituent agents do in terms of the goals that they work to attain and the tasks they must accomplish. The model also captures the way that goals and tasks of the system as a whole are related to goals and tasks assigned to specific agents and the dependencies among them. Goals and tasks both have attributes of type Situation, such that they can be linked by logical dependencies to form graphs that show e.g. decomposition of high-level goals into sub-goals, and how tasks can be performed to achieve goals. UML Activity Diagrams are applied for presentation purposes. Goals describe the desired states of the system and its environment, whereas tasks describe state transitions that that are needed to satisfy agent goal commitments. The state transition is specified as a pre-and post-condition attribute pair. Actions are atomic tasks that can be performed by the agents to satisfy their goal commitments. Task inputs are Model Elements (adapted from UML defining elements composing models) that are processed in task. Task outputs are updates of the input Model Elements plus any new Model Element produced by the task. The desired states of a Model Element are specified by attributes called invariants, which are conditions that should always be true. An example is shown in Fig. 3.

a) 

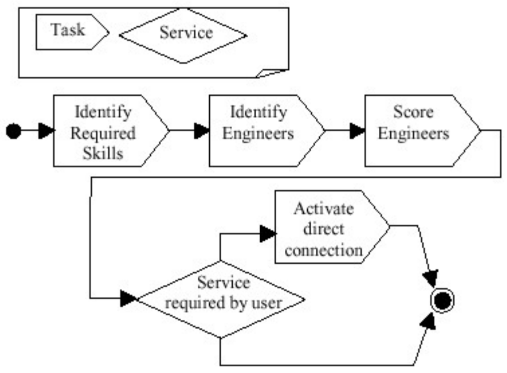
b) 

Fig. 3. Example of a) goal implication diagram, b) workflow diagram

*Agent/Role Model***:** The Agent Model consists of a set of individual agents and roles. The relationship between role and agent is defined analogous to that between an interface and an object class: a role describes the external characteristics of an agent in a particular context. An agent may be capable of playing several roles, and multiple agents may be able to play the same role. roles can also be used as indirect references to agents. An element of the Agent Model gathers together information specific to an individual agent or role, including its relationships to other entities. In particular, it contains a detailed and comprehensive description of each individual agent providing an internal view including the agent's goals and the services, i.e. the functional capability, they provide. In contrasts to the external perspective provided by the Organization Model. For each agent/role it uses schemata supported by diagrams to define its characteristics such as what goals it is responsible for, what events it needs to sense, what resources it controls, what tasks it knows how to perform, 'behavior rules', etc. An example for an agent model is given in Fig. 4.

The Domain (Information) Model: The Domain Model functions as a repository of relevant information about the problem domain. The conceptualization of the specific domain is assumed to be a mixture of *object-oriented*, i.e. all entities in the domain are classified in classes and each class groups all entities with a common structure, and *relational*, i.e. a number of relations describe the mutual relationships between the entities belonging to the different classes. Thus the Domain Model defines the *domain-specific classes* agents deal with and describes the structure of each class in terms of a number (possibly null) of attributes having values that can belong to primitive types or can be instances of other domain specific classes.

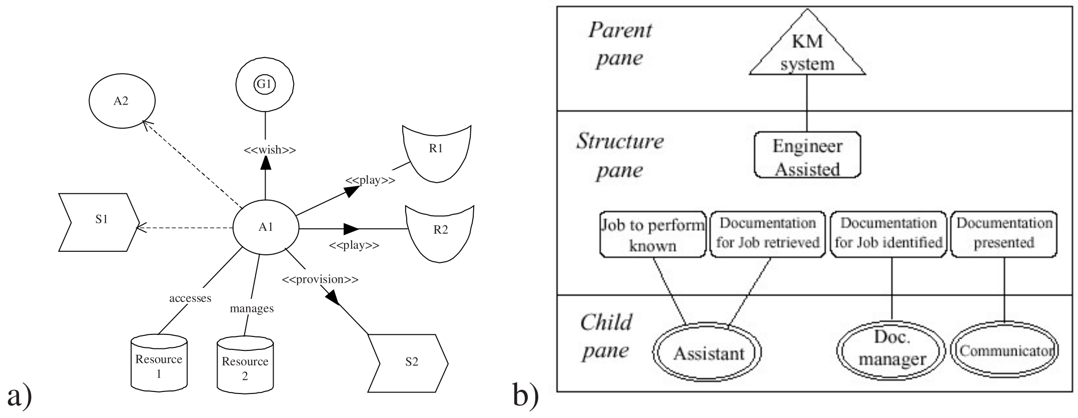


Fig. 4. Example of a) agent diagram[2], b) delegation structure diagram

In addition, *domain specific relations* holding among the instances of the domain specific classes are captured. Class diagrams are used for this model, as illustrated:

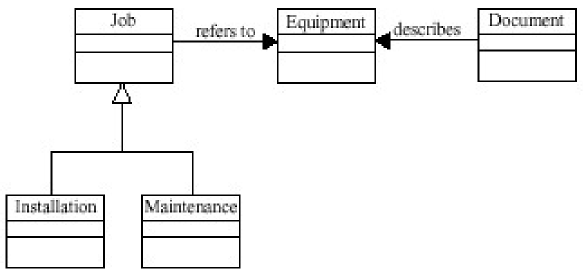


Fig. 5. Example of Domain Model as UML class diagrams

The Interaction Model: The Interaction Model is concerned with capturing the way in which agents (or roles) exchange information with one another (as well as with their environment captures). The content of the messages within an interaction may be described in the Domain Model. Interactions are specified from both a high-level and low-level perspective (interaction protocols based on the UML interaction protocols).

[2] Taken from [6].

For each interaction among agents/roles, shows the initiator, the collaborators, the motivator (generally a goal the initiator is responsible for), the relevant information supplied/achieved by each participant, the events that trigger the interaction, other relevant effects of the interaction (e.g. an agent becomes responsible for a new goal). Larger chains of interaction across the system (e.g. corresponding to uses cases) can also be considered such as delegation or workflows. An example for interaction is shown in Fig. 6 and on agent interaction diagrams.

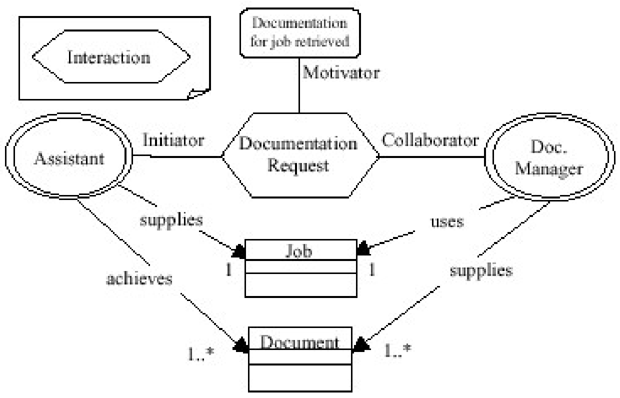


Fig. 6. Example of an Interaction

2.3 Tropos

Tropos [27][30][29] is another good example of a 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 [34]. 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* [32] 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 [33]. Thus, Tropos was developed around two key features: Firstly, the notions of agent, goal, plan and various other knowledge-level concepts are provided as fundamental primitives used uniformly throughout the software development process; secondly, a crucial role is assigned to requirements analysis and specification when the system-to-be is analyzed with respect to its intended environment using a phase model: *Early Requirements:* identify relevant stakeholders (represented as actors), along with their respective objectives (represented as goals); *Late Requirements:* introduce system to be developed as an actor describing the dependencies to other actors indicating the obligations of the system towards its environment; *Architectural Design:* introduce more system actors assigned sub-goals or subtasks of the goals and tasks assigned to the system;

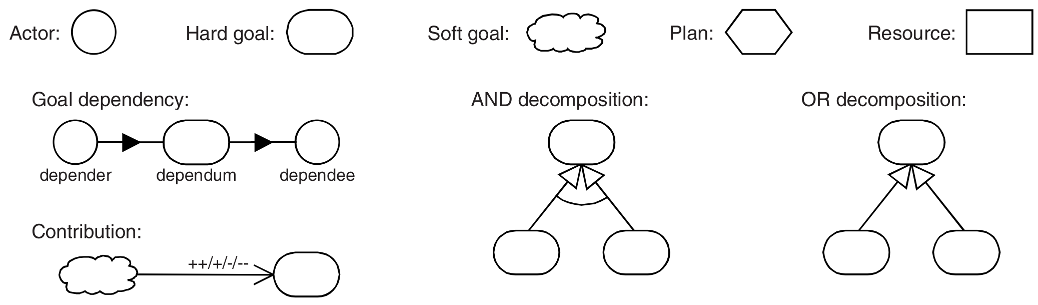


Fig. 7. Examples of Tropos notation

Detailed Design: define system actors in detail, including communication and coordination protocols; *Implementation:* transform specifications into a skeleton for the implementation mapping from the Tropos constructs to those of an agent programming platform. The specification covers the following notation illustrated in Fig. 7.

The Tropos specification makes use of the following types of models (following [27]):

Actor and Dependency Model: Actor and dependency models graphically represented through actor diagrams result from the analysis of social and system actors, as well as of their goals and dependencies for goal achievement as shown in Fig. 8. An actor has strategic goals and intentionality and represents a physical agent (e.g., a person), or a software agent as well as a role (abstract characterization of the behavior of an actor within some specialized context) or a position (a set of roles, typically played by one agent). An agent can occupy a position, while a position is said to cover a role. Actor models are extended during the late requirements phase by adding the system as another actor, along with its inter-dependencies with social actors. Actor models at the architectural design level provide a more detailed account of the system-to-be actor and its internal structure. This structure is specified in terms of subsystem actors, interconnected through data and control flows that are modeled as dependencies. A dependency between two actors indicates that one actor depends on another in order to attain some goal, execute some plan, or deliver a resource. By depending on other actors, an actor is able to achieve goals that it would otherwise be unable to achieve on its own, or not as easily, or not as well.

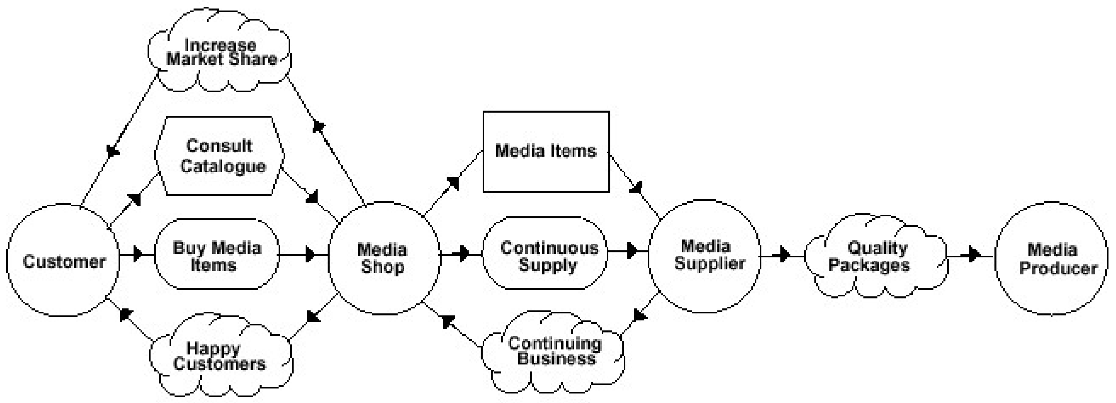


Fig. 8. Actor Diagram in Tropos (taken from [33])

Goal and Plan models: Goal and plan models allow the designer to analyze goals representing the strategic interests of actors and plans representing a way of a goal is satisfied from the perspective of a specific actor by using three basic reasoning techniques: *means-end analysi*s refining a goal into subgoals in order to identify plans, resources and soft goals that provide means for achieving the goal (the end); *contribution analysi*s pointing out goals that can contribute positively or negatively in reaching the goal being analyzed, and *AND/OR decompositio*n allowing to combination of AND and OR decompositions of a root goal into sub-goals, thereby refining a goal structure. Between two kinds of goals is distinguished, namely hard goals and soft goals, the latter having no clear-cut definition and/or criteria as to whether they are satisfied. Goal models are first developed during early requirements using initially-identified actors and their goals.

Capability diagram: A capability, modeled either textually (e.g. as a list of capabilities for each actor) or as capability diagrams using UML activity from an agent's point of view, represents the ability of an actor to define, choose and execute a plan to fulfill a goal, given a particular operating environment. Starting states of a capability diagram are external events, whereas activity nodes model plans, transitions model events, and beliefs are modeled as objects. Each plan node of a capability diagram can be refined by UML activity diagrams.

Agent interaction diagrams: Protocols are modeled using the Agent UML sequence diagrams [1]

2.4 Prometheus

Similar to Tropos, Prometheus [37][36][35] 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 [34]. The Prometheus methodology covers three phases, namely those of System specification, architectural design, and detailed design. Fig. 9 illustrates the Prometheus process [35].

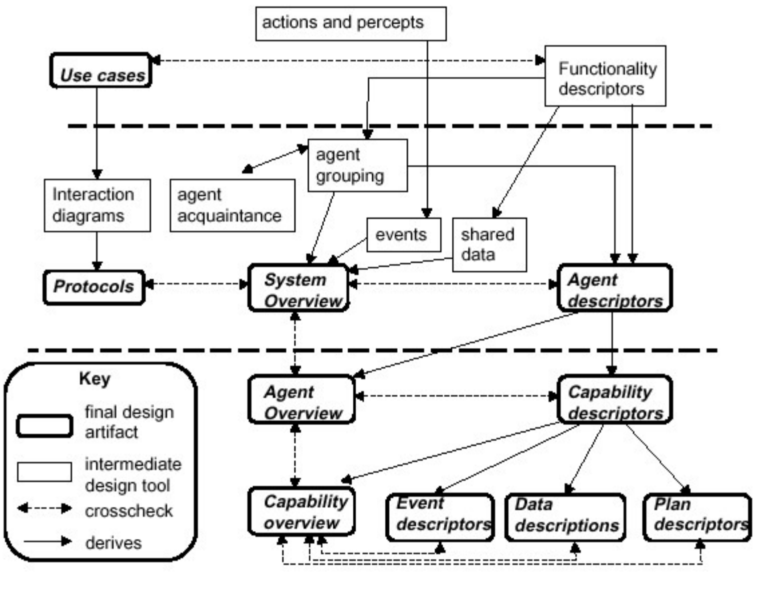


Fig. 9. Prometheus process overview

In the following, we describe the three phases of the Prometheus methodology according to [36], [35] .
System Specification: The System Specifications focuses on identifying the basic functions of the system, along with inputs (percepts), outputs (actions) as well as their processing (e.g. how are percepts to be handled and any important shared data sources to model the system's interaction with respect to its changing and dynamic environment. To understand the purpose of a system, use case scenarios borrowed from object-orientation with a slightly enhanced structure give a more holistic view than the mere analysis of the system functions in isolation.
Architectural Design: The architectural design phase subsequent to system specification determines which agents the system will contain and how they will interact. The major decision to be made during the architectural design is which agents should exist within the system. The key design artifacts used in this phase are the *system overview diagram* tying together agents, events and shared data objects, *agent descriptions* and the *interaction protocols* (based on Agent UML sequence diagrams [1]) specifying fully the interaction between agents. Agent messages are also identified, forming the interface between agents. Data objects are specified using traditional object oriented techniques. Taken the examples from [35] the diagrams look as illustrated in Fig. 10:

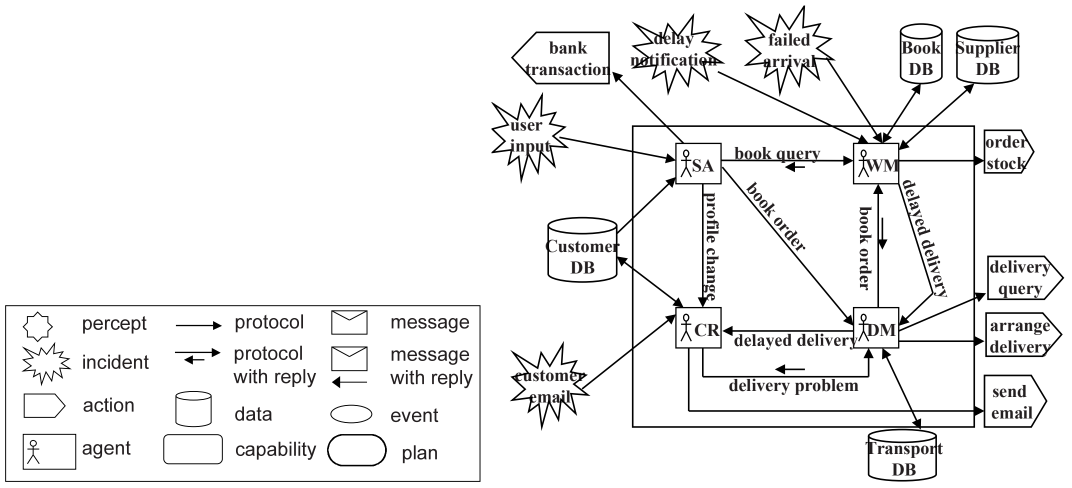


Fig. 10. Example of system overview diagram

Detailed design: The detailed design phase describes the internals of each agent and how it will achieve its tasks within the overall system. The focus is on defining capabilities (modules within the agent), internal events, plans and detailed data structures. Outcomes from this phase are *agent overview diagrams* (see Fig. 11a) providing the agent's top-level capabilities, *capability diagrams* (see Fig. 11b), detailed *plan descriptors* and *data descriptions*. Capabilities can be nested within other capabilities; thus this model supports arbitrarily many layers in the detailed design, in order to achieve an understandable complexity at each level. They are refined until all capabilities are defined in terms of other capabilities, or (eventually) in terms of events, data, and plans.

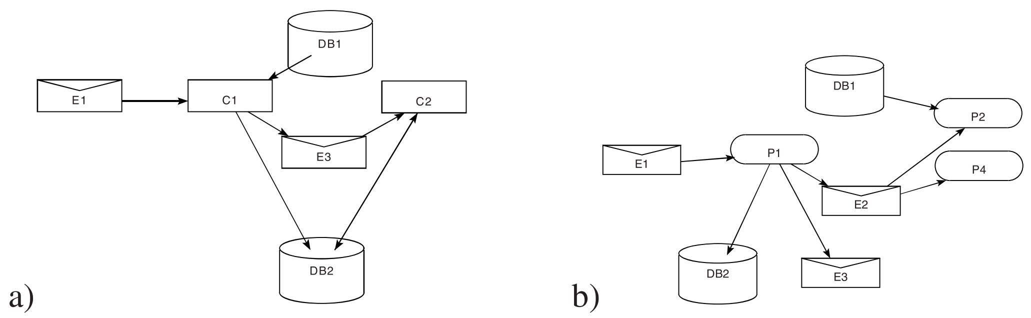


Fig. 11. Example of a) agent overview diagram, and b) capability overview diagram taken from [35]

2.5 MaSE

Multiagent Systems Engineering (MaSE) (we base our presentation on [24], for details we refer to [25][26]) 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. The MaSE methodology is heavily based on UML and the RUP. The software development process is detailed in analysis and design. The different models to be covered are:

Capturing Goals: In this phase, the initial requirements are transformed into a structured set of system goals. A goal is always defined as a system-level objective. Goals are identified by distilling the essence of the set of requirements and are then analyzed and structured into a form that can be passed on and used in the design phases. Therefore the goals are organized by importance in a goal hierarchy diagram. Each level of the hierarchy contains goals that are roughly equal in scope and all sub-goals relate functionally to their parent.

Applying Use Cases: Use cases are drawn from the system requirements as in any UML analysis. Subsequently, sequence diagrams are applied to determine the minimum set of messages that must be passed between roles. Typically, at least one sequence diagram is derived from a use case.

Refining Roles: The roles and concurrent tasks are assigned from the goal hierarchy diagram and the sequence diagrams. A role in MaSE is an abstract description of an entity's expected function and encapsulates the system goals the entity is responsible for. MaSE allows a traditional role model and a methodology-specific role model including information on interactions between role tasks shown in Fig. 12.

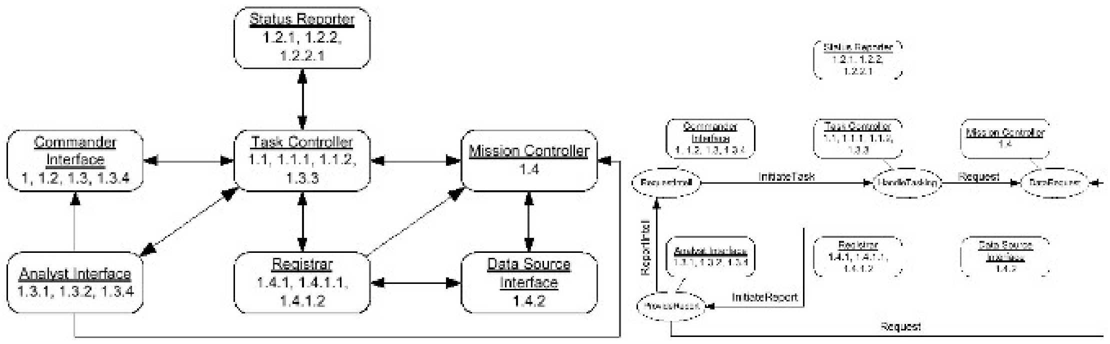


Fig. 12. MaSE a) traditional role model and b) MaSE role model.

Creating Agent Classes: The agent classes are identified from component roles. The result of this phase is an agent class diagram depicting agent classes and the conversations between them. .

Constructing Conversations: A MaSE conversation defines a coordination protocol between two agents. Specifically, a conversation consists of two communication class diagrams, one each for the initiator and responder. A communication class diagram is a pair of finite state machines that define the conversation states of the two participant agent classes.

Assembling Agent Classes: the internals of agent classes are created based on the underlying architecture of the agents, like BDI, re-active agents, etc.

System Design: This model takes the agent classes and instantiates them as actual agents. It uses a Deployment Diagram to show the numbers, types, and locations of agents within a system.

2.6 PASSI

PASSI (Process for Agent Societies Specification and Implementation) [16][17] 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. We base our survey on [17].

The PASSI methodology consists of five models (System Requirements, Agent Society, Agent Implementation, Code Model and Deployment Model) which include several distinct phases as described in the following.

System Requirements Model: The System Requirements model is obtained in different phases: The *Domain Description* Phase results in a set of use case diagrams where scenarios are detailed using sequence diagrams. The next phase, namely the *Agent Identification*, defines, based on use cases, packages where the functionality of each agent is grouped and activity diagrams for the task specification of this agent. I.e., in contrast to most of the agent-oriented methodologies agents are identified based on their functionality and not on their roles. The *Role Identification* Phase is a functional/behavior description of the agents as well as a representation of its relationships to other agents described by a set of sequence diagrams. Roles are viewed as in traditional object-oriented approaches. One activity diagram is drawn for each agent in the *Task Specification* Phase where each diagram is divided into two segments, one dealing with the tasks of an agents and one with the tasks for the interacting agent.

Agent Society Model: The agent society model is derived in the phases: *Ontology Description* describes the agent society or organization from a ontological point of view. Therefore two diagrams are introduced, the *Domain Ontology Description* and *Communication Ontology Description* usually presented using Class Diagrams and XML Schema for textual representation. The *Role Description* Phase models the life of the agents looking at its roles, therefore social or organizational roles and behavioral roles, represented by class diagrams where roles are classes grouped in packages representing the agents. In particular role changes can be defined. Roles are obtained by composing several tasks (roles are based on the functionality of an agent!). A part of such a diagram is shown in Fig. 13.

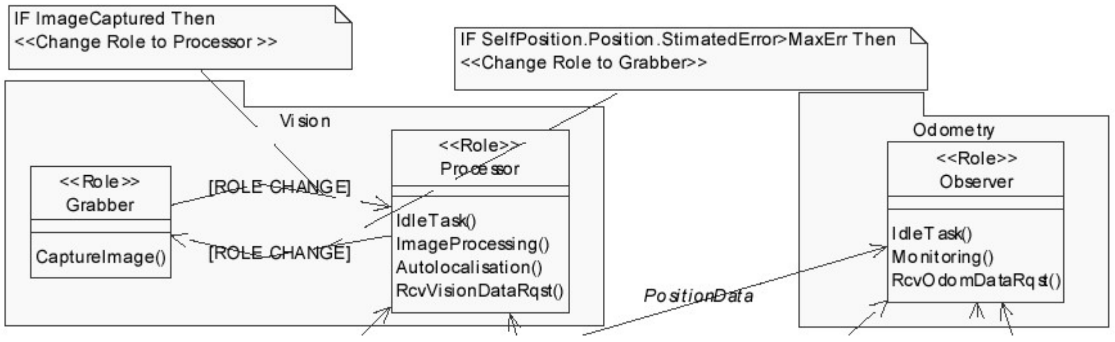


Fig. 13. Excerpt from PASSI role diagram

Agent Implementation Model: This model covers the *Agents Structure Definition* and the *Agents Behavior Description* Phases, describing respectively the *multi-agent level* represented by classes where attributes are the knowledge of the agent, methods are the tasks of an agent and relationships between agents define the communication between them; and *the single-agent level* defines one single class diagram for each agent, describing the complete structure of an agent with its attributes and methods. In particular, the methods needed to register the agent and for each task of the agent is represented as a class.

Code Model: Based on the FIPA standard architecture standard code pieces are available for re-use and therefore automatic code generation from the models is partly supported.

Deployment Model: UML deployment diagrams are extended to define the deployment of the agents and in particular to specify the behavior of mobile agents.

3 Modeling Notations Based on UML

The UML modeling notation is applied in various papers for the modeling of different aspects of agent-based software systems. While some approaches (e.g., [9], [10]) use plain UML 1.4 as a base notation for agent-based software development, there is a shared understanding, that UML as presented in version 1.4 is not sufficient for modeling agent-based systems [11]. The upcoming UML 2 standard will address some current limitations and some parts of UML extension for agent-based systems will be taken into consideration in UML 2. Therefore in the following we will only present those extensions of UML 1.4, for which to our knowledge no updated version based on UML 2 is available.

3.1 Interaction Protocols

One of the first extensions to UML, in particular sequence diagrams were proposed in [12][1]. This notation was also applied as a basis for the specification of FIPA interaction protocols. In the meantime, this description was adapted to UML 2. An agent interaction protocol [13] is then represented as a sequence diagram as shown in Fig. 14 (taken from that source):

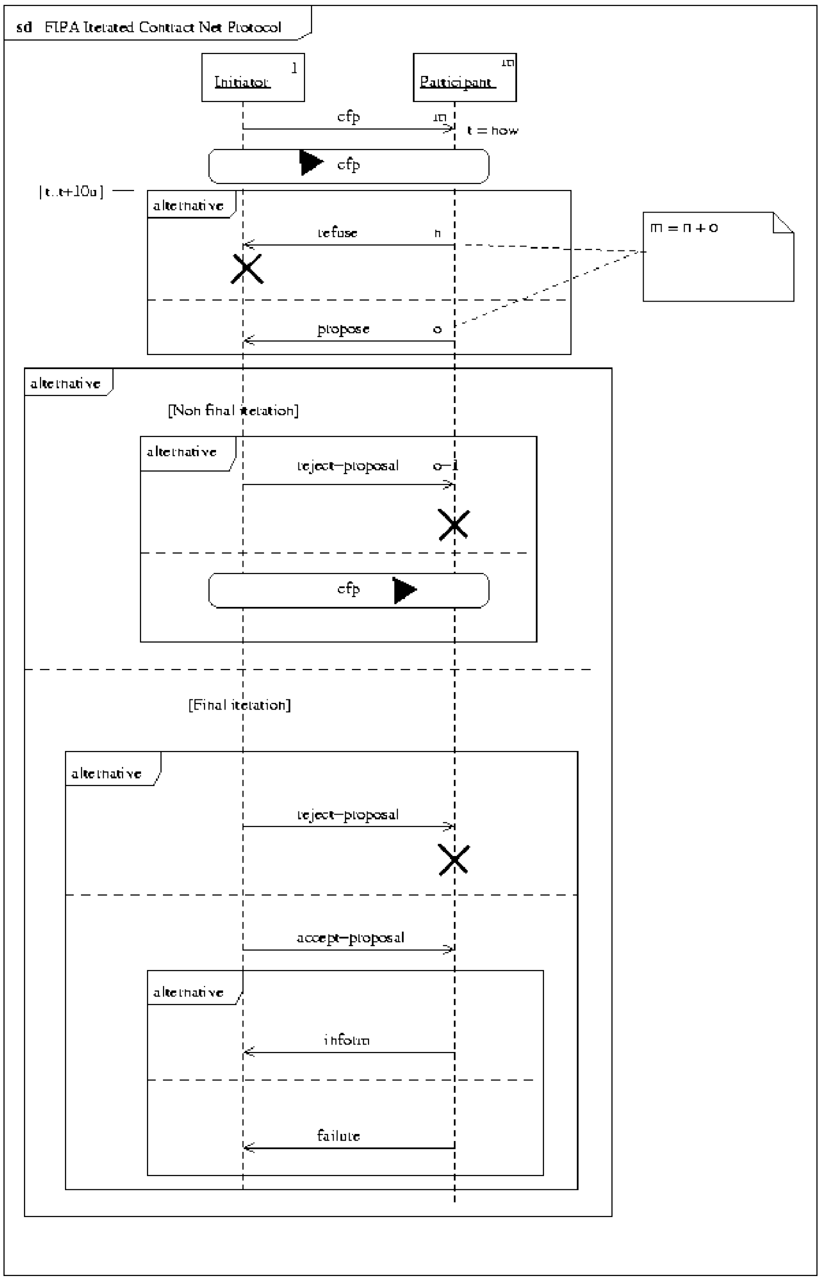


Fig. 14. FIPA Iterated Contract Net Protocol

In the contract net protocol, one agent takes the role of manager, e.g. a customer. The manager wishes to have some task performed by one or more other agents e.g. order some items, and further wishes to optimize a function that characterizes the task e.g. price and time of good. The Customer solicits proposals from the order acquisition by issuing a call for proposals (cfp), which specifies the task and any conditions the manager (Customer) is placing upon the execution of the order. Agents receiving the call for proposals are viewed as potential contractors, and are able to generate proposals to perform the task, e.g. the ordering as propose acts. The contractor's proposal e.g. Order Acquisition includes the preconditions that the contractor is setting out for the task, being the price and time when the order will be done. Alternatively, the contractor may refuse to propose or may iterate the process by issuing a revised cfp. The intent is that the Customer seeks to get better bids from the Order Acquistion by modifying the call and requesting new (equivalently, revised) bids. Once the Customer receives back replies from all of the Order Acquisition, it evaluates the proposals and makes its choice of which agents will perform the task. The process terminates when the Customer refuses all proposals and does not issue a new call, accepts one or more of the bids, or the Order Acquisitions all refuse to bid. The agents of the selected proposal(s) will be sent an acceptance message, the others will receive a notice of rejection. The proposals are assumed to be binding on the Order Acquisition, so that once the Customer accepts the proposal the Order Acquisition acquires a commitment to perform the task. Once the Order Acquisition has completed the task, it sends a completion message to the Customer.

3.2 Social Structures

Based on the emphasis on the correspondence between multi-agent systems and social systems, Parunak and Odell [38] combine several organizational models for agents, including AALAADIN, dependency theory, interaction protocols, and holonic modeling, in a general theoretical framework, and show how UML can be applied and extended to capture constructions in that framework. Parunak and Odell's model is based on the following artifacts: *roles*: They assume, that the same role can appear in multiple groups, if they embody the same pattern of dependencies and interactions. If an agent in a group holds multiple roles concurrently, it may sometimes be useful to define a higher-level role that is composed of some of those more elementary roles; *environments* environment are not only passive communications framework and everything of interest is relegated to it, but actively provides three information processing functions; It *fuses* information from different agents passing over the same location at different times; it *distributes* information from one location to nearby locations; it provides *truth maintenance* by forgetting information that is not continually refreshed, thereby getting rid of obsolete information; *Groups*: groups represent social units that are sets of agents associated by a common interest, purpose, or task. Groups can be created for three different reasons, i.e.: (i) for achieving more efficient or secure interaction between a set of agents (*intra-group associations*); (ii) for taking advantage between the synergies between a set of agents, resulting in an entity (the group) that is able to realize products, services, or processes that no individual by itself would be capable of (*group synergies*); and (iii) establishing a group of agents that interacts with other agents or groups in a coherent way, e.g., to represent a shared position on a subject (*inter-group associations*).

The conceptual model of Parunak and Odell's approach is illustrated in Fig. 15. In [38], the authors provide some examples for modeling social agent environments, namely a terrorist organization and its relationship to a weapons cartel. Groups are modeled by class diagrams and swimlanes as shown in Fig. 16, denoting that the Terrorist Organization involves two roles, Operative and Ringleader, where the Ringleader agent coordinates Operative agents.

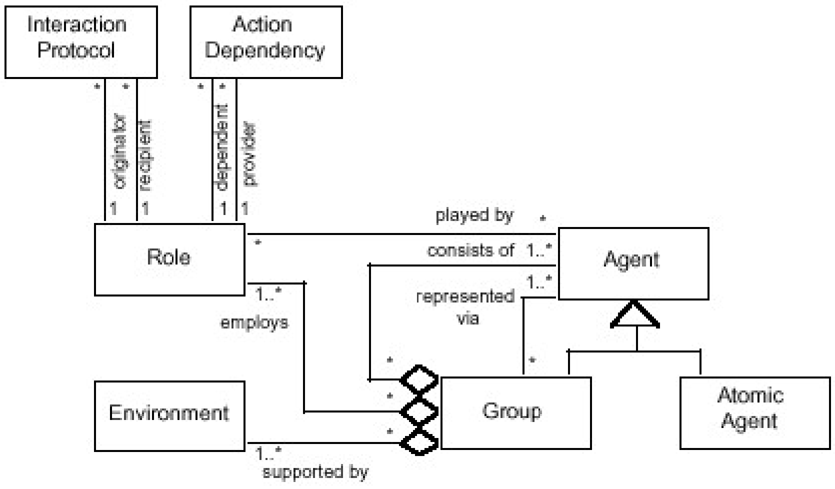


Fig. 15. Conceptual model of Parunak and Odell's approach

The second swimlane is based on agent instances, e.g. agent A plays the roles of Operative, Customer, and Student (expressed in b).

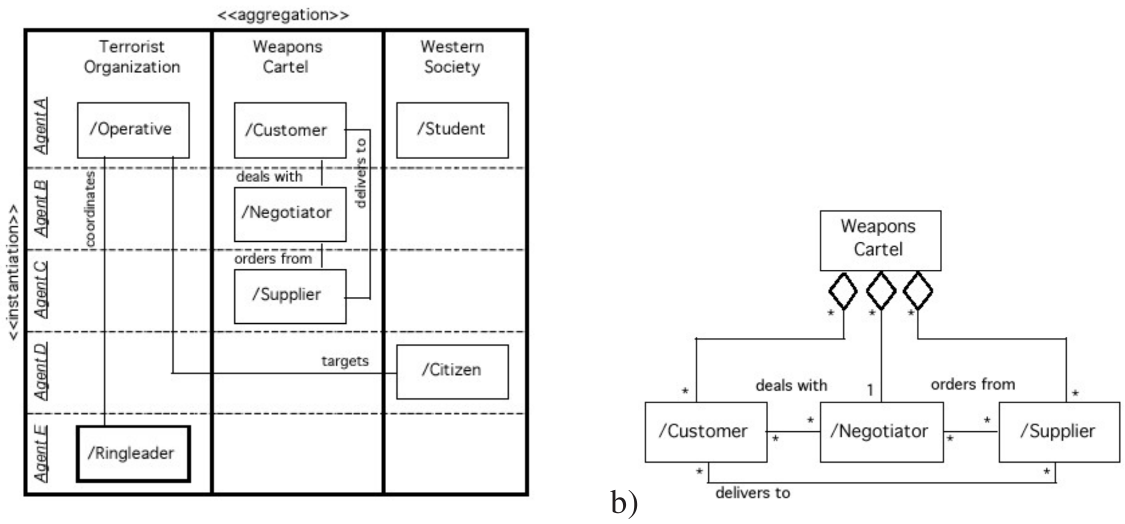
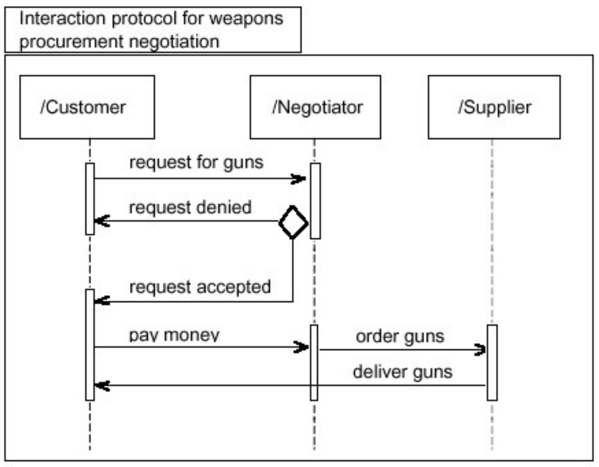


Fig. 16. a) Swimlanes as groups; b) class diagrams defines roles

Sequence diagrams are used to show roles as patterns of interactions; class diagrams model the kinds of entities that exist in a system along with their relationships, whereas sequence diagrams model the interactions that may occur among these entities. Fig. 17a) depicts the permitted interactions that may occur among Customer, Negotiator, and Supplier agents for a weapons procurement negotiation. Fig. 17b) shows an activity graph modeling groups of agents as agents. In this way, the kinds of dependencies are expressed that are best represented at a group level.

a)



b)

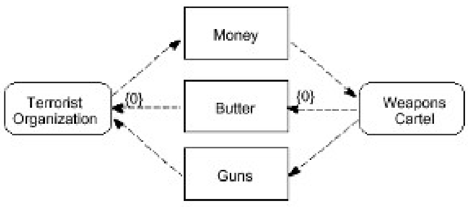


Fig. 17. a) Sequence diagram depicting an interaction protocol b) object-flow activity graph specifies roles as patterns of activities

3.3 Agent Classes

To our knowledge, basing agent classes on UML class diagrams was so far only considered by [15] and [14], with the notable exception of [43], where Wagner presents an UML profile for an agent-oriented modeling approach called an Agent-Object-Relationship modeling language (AORML). AORML can be viewed as an extension of UML covering (among others) *Interaction Frame Diagrams* describing the action event classes and commitment/claim classes determining the possible interactions between two agent types (or instances), *Interaction Sequence Diagrams* depicting prototypical instances of interaction processes, and *Interaction Pattern Diagrams* for representing general interaction patterns. The latter [14] is currently revisited within FIPA; an adapted version will be available by the end of 2003. Following [14] a distinction is made between an *agent class*, defining a blueprint for and the type of an individual agent, and between *individual agents* (being instances of an agent class). An agent class diagram shown in Fig. 18 specifies agent classes.

[14] states that usual UML notation with stereotypes can be used to define such an agent class, but for readability reasons the above notation was introduced:

Agent Class Descriptions and Roles: As we have seen agents can satisfy distinguished roles in most of the methodologies. The general form of describing agent roles in Agent UML [12] is

instance-1 ... instance-n / role-1 ... role-m : class

denoting a distinguished set of agent instances instance-1,..., instance-n satisfying the agent roles role-1,..., role-m with n, m ≥ 0 and class it belongs to. Instances, roles or class can be omitted, for classes the role description is not underlined.

State description: A *state description* is similar to a field description in class diagrams with the difference that a distinguished class *wff* for *well-formed formula* for all kinds of logical descriptions of the state is introduced, independent of the underlying logic. This extension allows the definition of e.g. BDI agents. Beyond the extension of the type for the fields, visibility and a persistency attributes can be added (denoted by the stereotype <<persistent>>) to allow the user agent to be stopped and re-started later in a new session. Optionally the fields can be initialized with some values. In the case of BDI semantics three instance variables can be defined, named *beliefs*, *desires*, and *intentions* of type *wff*. Describing the beliefs, desires, and intentions of a BDI agent. These fields can be initialized with the initial state of a BDI agent. The semantics state that the *wff* holds for the beliefs, desires, and intentions of the agent. In a pure goal-oriented semantics two instance variables of type *wff* can be defined, named *permanent-goals* and *actual-goals*, holding the formula for the permanent and actual goals. Usual UML fields can be defined for the specification of a plain object oriented agent, i.e. an agent implemented on top of e.g. a Java-based agent platform. However in different design stages different kinds of agents can be appropriate, on the conceptual level BDI agents can be specified implemented by a Java-based agent platform, i.e. refinement steps from BDI agents to Java agents are performed during the agent development.

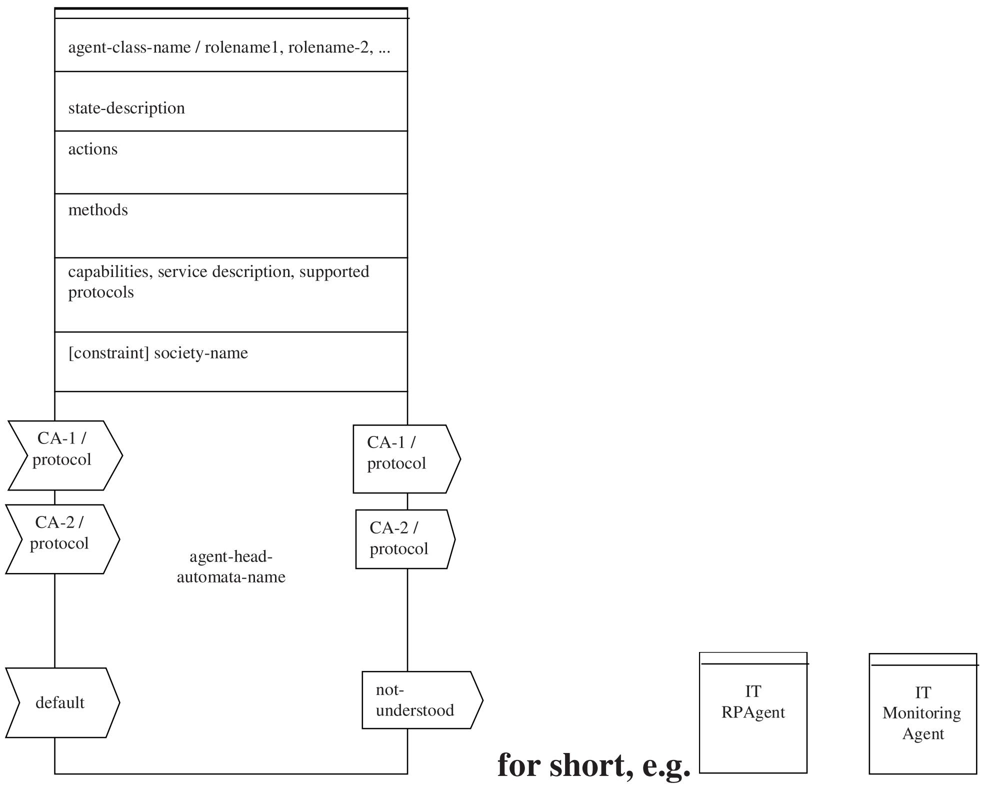
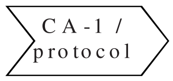


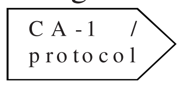
Fig. 18. Agent class diagram and its abbreviations

Actions: Pro-active behavior is defined in two ways, using *pro-active actions* and pro-active agent state charts. The latter one will be considered later. Thus two kinds of actions can be specified for an agent: pro-active actions (denoted by the stereotype <<pro-active>>) are triggered by the agent itself, if the pre-condition of the action evaluates to true. *re-active actions* (denoted by the stereotype <<re-active>>) are triggered by another agent, i.e. receiving a message from another agent. The description of an agent's actions consists of the action signature with visibility attribute, action-name and a list of parameters with associated types. Pre-conditions, post-conditions, effects, and invariants as in UML define the semantics of an action.

Methods: Methods are defined as in UML, eventually with pre-conditions, post-conditions, effects and invariants.

Capabilities: The capabilities of an agent can be defined either in an informal way or using class diagrams e.g. defining FIPA-service descriptions.

Sending and Receiving of Communicative Acts: Sending and receiving communicative acts characterize the main interface of an agent to its environment. By communicative act (CA) the type of the message as well as the other information, like sender, receiver or content in FIPA-ACL messages, is covered. It is assumed that classes and objects represent the information about communicative acts. The incoming messages are drawn as 

and the outgoing messages are drawn as 

. The received or sent communicative act can either be a class or a concrete instance. The notation *CA-1 / protocol* is used if the communicative act of class *CA-1* is received in the context of an interaction protocol *protocol*. In the case of an instance of a communicative act the notation *CA-1 / protocol* is applied. As alternative notation *protocol[CA-1]* and *protocol[CA-1]* can be used. The context */ protocol* can be omitted if the communicative act is interpreted independent of some protocol. In order to react to all kinds of received communicative acts, we use a distinguished communicative act *default* matching any incoming communicative act. The *not-understood* CA is sent if an incoming CA cannot be interpreted.

Matching of Communicative Acts: A received communicative act has to be matched against the incoming communicative acts of an agent to trigger the corresponding behavior of the agent. The matching of the communicative acts depends on the ordering of them, namely the ordering from top to bottom, to deal with the case that more than one communicative act of the agent matches an incoming message. The simplest case is the default case, *default* matches everything and *not-understood* is the answer to messages not understood by an agent. Since instances of communicative acts are matched, as well as classes of communicative acts, free variables can occur within an instantiated communicative act. This matching is formally defined in [14].

3.4 Ontologies

As we have already noticed e.g., in PASSI, several research approaches are dealing with the definition of ontologies using UML class diagrams [19][20], not only from the agent-oriented research community but also from the Semantic Web community [17] [22]. Bergenti et al. [19] take a pragmatic view an ontology definition applying UML class diagrams as shown in Fig. 19, defining the entities and on the other relating it to specific agents.

Cranefield et al. use UML to define agent communication languages (ACL) and content languages, like an object-oriented implementation of the FIPA ACL or FIPA SL [21], see also Fig. 20. They also apply UML for ontology definition and rely description logic with UML in [20].

In [22] an extension of UML is defined to cover DAML defining a high-level mapping between UML and DAML, e.g. Ontologies are viewed as packages, classes as classes, properties as attributes, associations and classes.

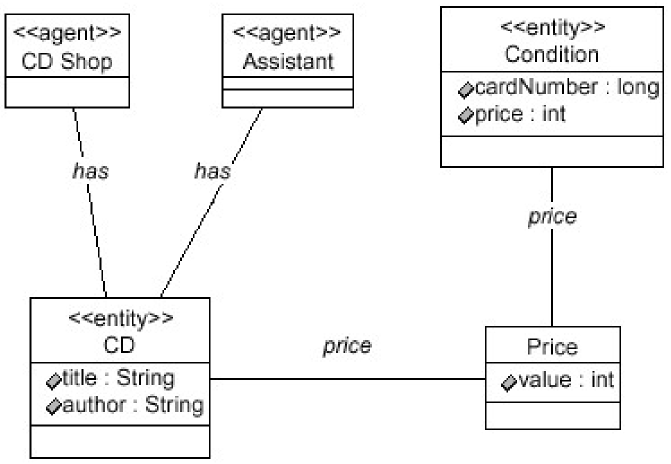


Fig. 19. UML-based ontology definition

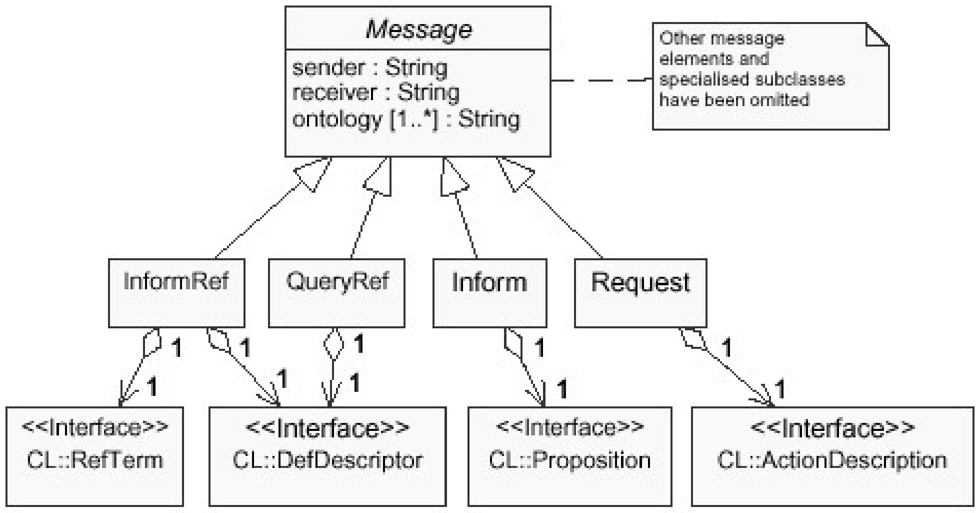

Message
sender : String
receiver : String
ontology [1..*] : String
Other message elements and specialised subclasses have been omitted
InformRef
QueryRef
Inform
Request
1
1
1
1
1
1
1
1
1
1
<<Interface>> CL::RefTerm
<<Interface>> CL::DefDescriptor
<<Interface>> CL::Proposition
<<Interface>> CL::ActionDescription

Fig. 20. Excerpt of object-oriented design of FIPA ACL/SL

3.5 Goals and Plans

Goals and plans are described by state charts or activity diagrams in several methodologies (see above). In [23] Huget uses UML 2.0 activity diagrams for the descriptions of goals and plans. We present here his example of a goal diagram corresponding to the interaction between the customer and the order acquisition (see Fig. 21).

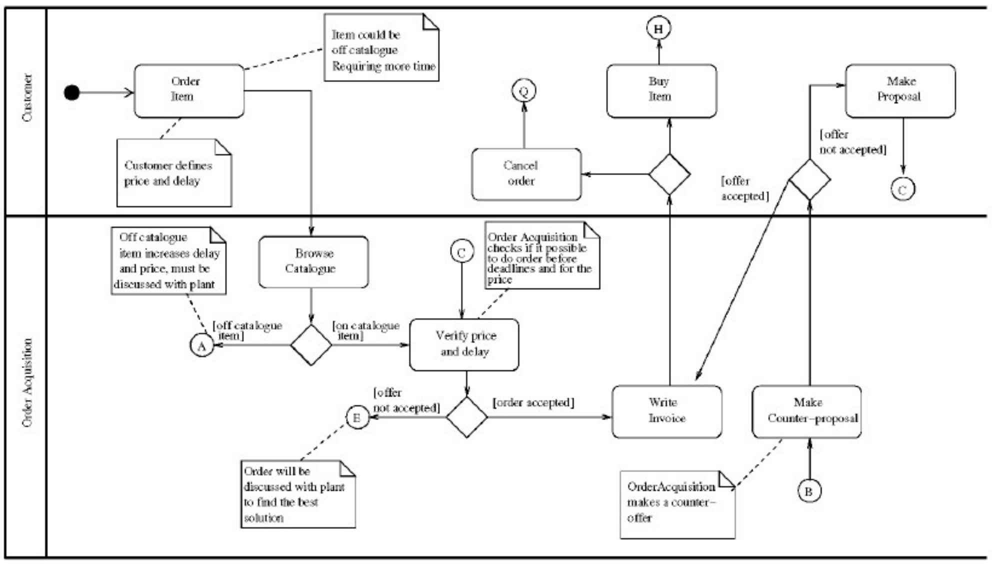

Customer
Order Acquisition
Order Item
Item could be off catalogue Requiring more time
Customer defines price and delay
Q
Cancel order
H
Buy Item
[offer accepted]
[offer not accepted]
Make Proposal
C
Off catalogue item increases delay and price, must be discussed with plant
Browse Catalogue
[off catalogue item]
A
[on catalogue item]
C
Order Acquisition checks if it possible to do order before deadlines and for the price
Verify price and delay
[offer not accepted]
E
[order accepted]
Write Invoice
Make Counter−proposal
Order will be discussed with plant to find the best solution
OrderAcquisition makes a counter−offer
B

Fig. 21. Supply Chain Management scenario as Activity Diagram

The goal diagram expresses the following. Customer first performs the action *Order Item*. This ordered item is received by the Order acquisition. The Order acquisition checks if the ordered item is on catalogue (action *Browse catalogue*). If the ordered item is off catalogue, then the following of the actions is on A[3]. This characteristic only changes how the item is produced and priced. If the ordered item is in the catalogue, the order acquisition checks the price and the delay (action *Verify price and delay*). If the proposal made by the customer cannot be processed for this delay and price then the order acquisition goes to E. After several actions, the order acquisition comes back to B to make a counter-proposal which is accepted or not by the customer. If the customer accepts the counter offer, next action is to write an invoice (action Write invoice). If the customer does not accept, it can make another proposal. The following is as defined above. Finally, after writing the invoice, the customer has two choices: either accepting the order (action Buy item) or canceling the order (action Cancel order).

4 Conclusions and Further Research

In this paper we surveyed a number of important research contributions in the area of methodologies and notations for the development of agent-based systems based on UML. The general approach of building agent-based features on top of an established object-oriented model introduces a number of trade-offs, in particular regarding the natural design of agent-based systems. Yet, the large advantage of these approaches is that they fit easier into the object-oriented conception, and that it is relatively easy to present higher-quality tools by extending existing object-oriented tools. It appears that while objects and agents are certainly different notions (see e.g., the discussion in [41]), agent-oriented software engineering can greatly benefit from OO technologies and approaches. In particular, agent-oriented approaches are also suitable for areas

[3] There exists only one matching for a letter: one encircled letter A with incoming arrow on this figure and one encircled letter A with outgoing arrow defined elsewhere

where object-oriented modeling has shortcomings. Here the abstractions inherent to agent-oriented software engineering can help us to overcome the limitations of the object-oriented approach.

Acknowledgement. We would like to thank the AOSE chairs for the invitation to contribute to this workshop proceedings. Moreover we want to thank all the people involved within FIPA TCs dealing with methodologies and notation for preparing an excellent collection of state-of-the-art paper on the AUML web-site.

References

[1] Bauer, B., Müller, J.P., Odell, J.: Agent UML: A Formalism for Specifying Multiagent Software Systems, International Journal on Software Engineering and Knowledge Engineering (IJSEKE), Vol. 11, No. 3, pp.1–24, 2001 Engineering, 2001.

[2] Kinny, D., Georgeff, M., Rao, A.: A Methodology and Modeling Technique for Systems of BDI Agents, in Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW 96), LNAI 1038, Springer, 1996.

[3] Kinny, D. and Georgeff, M.: A design methodology for BDI agent systems. Technical Report 55, Australian Artificial Intelligence Institute, Melbourne, Australia, 1995.

[4] Kinny, D. and Georgeff, M: Modelling techniques for BDI agent systems. Technical Report 54, Australian Artificial Intelligence Institute, Melbourne, Australia, 1995.

[5] Kinny, D. and Georgeff, M: Modelling and Design of Multi-Agent Systems, Proc. ATAL 96, 1996.

[6] MESSAGE web site: http://www.eurescom.de/public/projects/P900-series/p907/

[7] Giovanni Caire , Wim Coulier , Francisco Garijo, Jorge Gomez, Juan Pavon , Philippe Massonet, Francisco Leal, Paulo Chainho , Paul Kearney, Jamie Stark, Richard Evans , Agent Oriented Analysis using MESSAGE/UML, Proceedings AOSE 2001, Springer 2001.

[8] Elisabeth A. Kendall, Margaret T. Malkoun, and Chong Jiang. A methodology for developing agent based systems for enterprise integration. In D. Luckose and Zhang C., editors, Proceedings of the First Australian Workshop on DAI, Lecture Notes on Artificial Intelligence. Springer-Verlag: Heidelberg, Germany, 1996.

[9] Jürgen Lind: Iterative Software Engineering for Multiagent Systems: The MASSIVE Method. Springer, 2001

[10] Ralf Kühnel, Agentenbasierte Software - Methode und Anwendungen, Addison-Wesley, 2000.

[11] Bauer, B.; Bergenti, F., Massonet, Ph., Odell, J.: Agents and the UML: A Unified Notation for Agents and Multi-Agent Systems, Proceeding AOSE 2001, Montreal, Springer, 2001.

[12] Bauer, B.; Müller, J. P.; Odell, J.: An Extension of UML by Protocols for Multiagent Interaction, Proceeding, Fourth International Conference on MultiAgent Systems, ICMAS 2000, Boston, IEEE Computer Society, 2000.

[13] Marc-Philippe Huget (editor): FIPA-Modelling – Interaction Diagrams, first draft, online available at www.fipa.org

[14] Bauer, B.: UML Class Diagrams Revisited in the Context of Agent-Based Systems, in Proceedings AOSE 2001, Montreal, Springer, 2001.

[15] Wagner, G., *The Agent-Object-Relationship Metamodel: Towards a Unified View of State and Behavior.*, Technical Report, Eindhoven Univ. of Technology, Fac. of Technology Management, May 2002.

[16] PASSI website: www.csai.unipa.it/passi
[17] M. Cossentino, C. Potts: A CASE tool supported methodology for the design of multi-agent systems, in Proc. The 2002 International Conference on Software Engineering Research and Practice (SERP'02) Las Vegas (NV), USA, 2002.
[18] W3C Note NOTE-rdf-uml-19980804, available online at http://www.w3.org/TR/1998/NOTE-rdf-uml-19980804/
[19] Federico Bergenti, Agostino Poggi: A Development Toolkit to Realize Autonomous and Inter-operable Agents, in Proc. Autonomous Agents 2001, 2001.
[20] Stephen Cranefield, Stefan Haustein, Martin Purvis: UML-Based Ontology Modelling for Software Agents.
[21] Stephen Cranefield, Martin Purvis: Generating ontology-specific content languages
[22] Baclawski, K., Kokar, M., Kogut, P., Hart, L., Smith, J., Holmes, W., Letkowski, J., and Aronson M., "Extending UML to Support Ontology Engineering for the Semantic Web." *Proc. of the Fourth International Conference on UML (UML2001)*, Toronto, October 2001
[23] Marc-Philippe Huget: Representing Goals in Multi-Agent Systems, unpublished paper, 2003
[24] Mark F. Wood Scott A. DeLoach An Overview of the Multiagent Systems Engineering Methodology, In: Proceedings of the First International Workshop on Agent-Oriented Software Engineering, P. Ciancarini, M. Wooldridge, (Eds.) LNCS. Vol. 1957, Springer, 2001.
[25] Wood, M. F.: Multiagent Systems Engineering: A Methodology for Analysis and Design of Multiagent Systems. MS thesis, AFIT/GCS/ENG/00M-26. School of Engineering,
[26] DeLoach, S. A., Wood M. F.: Multiagent Systems Engineering: the Analysis Phase. Technical Report, Air Force Institute of Technology, AFIT/EN-TR-00-02, June 2000.
[27] Paolo Bresciani, Paolo Giorgini, Fausto Giunchiglia, John Mylopoulos , and Anna Perini. TROPOS: An Agent-Oriented Software Development Methodology. Journal of Autonomous Agents and Multi-Agent Systems. 2003. Kluwer Academic Publishers (to appear).
[28] Ivar Jacobson, Grady Booch, James Rumbaugh: The Unified Software Development Process, Addison Wesley, 1998.
[29] J. Mylopoulos, M. Kolp, and J. Castro. UML for agent-oriented software development: The Tropos proposal. In Proc. of the 4th Int. Conf. on the Unified Modeling Language UML'01, Toronto, Canada, Oct. 2001
[30] Tropos web site http://www.cs.toronto.edu/km/tropos/
[31] GRL web site: http://www.cs.toronto.edu/km/GRL/
[32] i* web site: http://www.cs.toronto.edu/km/istar/
[33] J. Castro, M. Kolp and J. Mylopoulos. Towards Requirements-Driven Information Systems Engineering: The Tropos Project. Information Systems, Elsevier, Amsterdam, The Netherlands, 2002.
[34] P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents – Components for Intelligent Agents in Java. Technical Report TR9901, AOS, January 1999. http://www.jackagents.com/pdf/tr9901.pdf.
[35] Lin Padgham and Michael Winikoff: Prometheus: A Methodology for Developing Intelligent Agents, In: Proceedings of AOSE 2002, Springer, 2002.
[36] Prometheus home page: http://www.cs.rmit.edu.au/agents/SAC/methodology.shtml
[37] Lin Padgham and Michael Winikoff, Prometheus: A Pragmatic Methodology for Engineering Intelligent Agents. In Proceedings of the workshop on Agent-oriented Methodologies at OOPSLA 2002. November 4, 2002
[38] H. Van Dyke Parunak and James Odell. Representing Social Structures in UML, In: Proceedings of AOSE 2001, Springer, 2001.
[39] Kennedy Carter eXecutable UML: http://www.kc.com/MDA/xuml.html
[40] Model-driven Architecture: http://www.omg.org/mda/

[41] Wooldridge, M.J. An introduction to multiagent systems. John Wiley & Sons, 2002.
[42] Bauer, B. and Müller, J.P.: Agent-Oriented Software Engineering: Methodologies and Modeling Languages - A State of the Art Survey -, to be published as book chapter, 2003.
[43] Wagner, G., A UML Profile for External AOR Models in Proceedings AOSE 2002, Springer, 2002
[44] Luck, M., McBurney P., and Preist, C., eds. Agent Technology: Enabling Enxt Generation Computing. AgentLink, http://www.agentlink.org, 2003.