Chapter 4

Methodologies and Modeling Languages

Bernhard Bauer, University of Augsburg, Germany
Jörg Müller, Siemens AG, Germany

4.1 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, and for defining the elements of the model together with a specific syntax or notation, and associated semantics) and a *software process* (defining the development activities, the interrelationships among the activities, and the ways in which 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, and so on, serving as input for subsequent activities.

The three key phases that one is likely to find in any software engineering process are those 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, and 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, and 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 nomadic users with proactive and intelligent assistance based on principles of adaptation and self-organization. Hence, agent technology can open the way to new application 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.

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. There is 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 [1] 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 [2]. 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 [3], Prometheus,[1] and MESSAGE.[2]

Various researchers have developed methodologies for agent design, touching on representational mechanisms, such as the Gaia methodology [4], or the extensive program under way at the Free University of Amsterdam on compositional methodologies for requirements [5], design [6], and verification [7]. In [8, 9], Kinny et al. propose a modeling technique for BDI agents. The close relationship between design mechanisms employed for agent-based systems and those used for object-oriented systems is identified by numerous authors (see, for example, [10]).

Having been the subject of intensive research activity for more than a decade, agent technology has still not met with broad acceptance in industrial settings (despite some encouraging success stories). We believe that three characteristics of industrial development have so far prevented wider adoption of agent technology:

1. The scope of industrial projects is much larger than typical research efforts.
2. The skills of developers are focused on established technologies, as opposed to leading-edge methods and programming languages.
3. The use of advanced technologies is not part of the success criteria of a project.

In order to establish a solid grounding for leveraging the usage of agent technologies in industrial applications, we recognize that accepted methods for industrial development must depend on widely standardized representations of artifacts supporting all phases of the software life cycle. 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 [11] for details). The Model-Driven Architecture (MDA[3]) from the OMG allows a cascade of code generation from high-level (platform-independent) models via platform-dependent

1 http://www.cs.rmit.edu.au/agents/SAC/methodology.shtml
2 http://www.eurescom.de/public/projects/P900-series/p907
3 http://www.omg.org/mda

models to directly executable code (see, for example, the tool offered by Kennedy Carter[4]). Another approach for agile software engineering that has also been receiving active coverage is Extreme Programming [12].

In this chapter, we provide a detailed survey of methodologies for agent-based engineering of software systems. We start with a classification of the different approaches with respect to their formal basis, their goals, and their completeness. Subsequently, we review two popular classes of approaches to the software engineering of agent-based systems: knowledge engineering approaches and agent-oriented approaches. This is followed by overviews of approaches adapting object-oriented techniques to agent-based development, including both methodologies and modeling notations. The chapter closes with a discussion of open issues and future research opportunities. (Note that in this chapter, we assume that the reader already has some familiarity with standard object-oriented concepts and techniques, particularly UML.)

4.2 A CLASSIFICATION OF EXISTING METHODOLOGIES AND NOTATIONS

Most early approaches supporting the software engineering of agent-based systems were inspired by the knowledge engineering community. We are now witnessing a renaissance of these approaches, notably CommonKADS, when faced with the problem of managing large ontologies in the context of Semantic Web applications (discussed in more detail in Chapter 5). Agent-oriented approaches, prominently represented by Gaia (described in Section 4.4.1), focus directly on the properties of agent-based systems and try to define a methodology to cope with all aspects of agents. A relatively new tendency is to base methodologies and modeling languages on object-oriented techniques, like UML, and to build the agent-specific extensions on top of these object-oriented approaches. These three approaches, knowledge engineering, agent-oriented, and object-oriented, are reflected in the structure of this chapter. However, before going into further detail of the different methodologies and notations, we introduce a classification schema, with the aim of structuring the field. We propose the following dimensions for our schema:

- *Scope* (methodology, process, or modeling language): Is the approach a methodology consisting of a process and a modeling language; is only the process part of a software engineering approach defined such as the RUP; or is only a modeling language available like UML?
- *Basis*: Which software technology has the strongest influence on the approach (for example, a pure agent-oriented approach, formal specification techniques, object-orientation, or knowledge engineering)?
- *Phases*: In which phases of the software process is the methodology, process, or modeling language applicable (for example, early requirements, analysis, design, or implementation)?

4 http://www.kc.com/MDA/xuml.html

- *Syntax and semantics*: What is the approach regarding syntax and semantics; is there an underlying formal model, like in model-theoretic descriptions; or do we have an informal description, such as for the dynamic aspects of UML?
- *Application areas*: Is the approach domain-specific (for example, Internet agents), or is it a general-purpose approach?
- *Agency support*: What aspects specific to agent-based systems are supported (such as goals, intentions, interactions)?

A summary of existing methodologies compared against these criteria is shown in Table 4.1.

4.3 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 [13], CoMoMAS [14], and MAS-CommonKADS [15].

Knowledge engineers need tools and methods to design good knowledge-based systems (KBS), but these rely on the knowledge engineer's abilities. The CommonKADS methodology was developed to support knowledge engineers in modeling expert knowledge and developing design specifications in textual or diagrammatic form. Thus, CommonKADS is a knowledge engineering methodology as well as a knowledge management framework. Two extensions to CommonKADS have been developed that take agent-specific aspects into account: CoMoMAS and MAS-CommonKADS. In the following, we only discuss Common-KADS and MAS-CommonKADS; for CoMoMAS, we refer to [14].

MAS-CommonKADS [15] adds various extensions to CommonKADS: protocol engineering techniques (namely software development life cycle and MSC96 [16]); object-oriented techniques (OMT [17, 18] and OOSE); and enhanced support for additional phases within the software life cycle.

As shown in Figure 4.1, the CommonKADS methodology is described by components arranged at three layers: the context layer, the concept layer, and the artifact layer.

- *Organization model* describes the organizational context in which the knowledge-based system will work. In particular, it identifies the various stakeholders of the organization, namely *knowledge providers*, being experts in a specific domain; *knowledge users*, needing the knowledge to do their work successfully; and *knowledge decision makers*, influencing the work of the others (the knowledge providers and users) by making decisions. The organization model is described by organizational aspects about the function, involved departments (structure), business processes, power, quality, accessibility in time or space, authority, needed knowledge, and used resources of an organization. In addition, business processes have to be broken down into tasks (see the task model).
- *Task model* describes the tasks representing a goal-oriented activity, adding value to the organization, and executed in the organizational environment. The task model is represented as a set of tasks with a structure imposed on it. A task is described by well-defined input and output, defining

Table 4.1
Classification of Existing Methodologies

Approach	**Scope**	**Basis**	**Phases**	**Syntax-Semantics**	**Application Area**	**Agency Support**
Common KADS CoMoKADS MAS-CommonKADS	Methodology	KM	Analysis and design; CommonKADS: impl.	Syntax and semantics to some extent	Knowledge-centered applications	Organization tasks, agents, knowledge, interaction
Gaia ROADMAP	Methodology	AO	Analysis and high-level design (Gaia)	Syntax and semantics to some extent	Coarse-grained computational systems	Roles, agents, knowledge, interaction, services, acquaintance
SODA	Mainly process	AO society centered	Analysis and design	Some syntax	Open systems	Roles, agents, resources, societies, interaction
Kinny et al.	Methodology	OO and BDI agents	Analysis and design	Syntax and semantics to some extent	BDI agents	Agents, interaction, belief, goals, plans
MESSAGE	Methodology	OO and RUP	Mainly analysis	Syntax and semantics to some extent	Coarse-grained computational systems	Organizations, goals, tasks, agents, roles, knowledge, interaction
Tropos	Methodology	OO and BDI	Analysis, design, and implementation	Syntax and semantics to some extent	BDI agents	Actor, goal, plan, resource, capability, interaction
Prometheus	Methodology	OO and BDI	Analysis, design, and implementation	Syntax and semantics to some extent	BDI agents	Goals, beliefs, plans, events, agents, interaction, capabilities
MaSE	Methodology	OO and RUP	Analysis, design, and implementation	Syntax and semantics to some extent	Heterogeneous MAS	Goals, roles, interaction, agents
PASSI	Methodology	OO and RUP	Analysis, design, and implementation	Syntax and semantics to some extent	Mainly robotics	Societies, agents, roles, knowledge

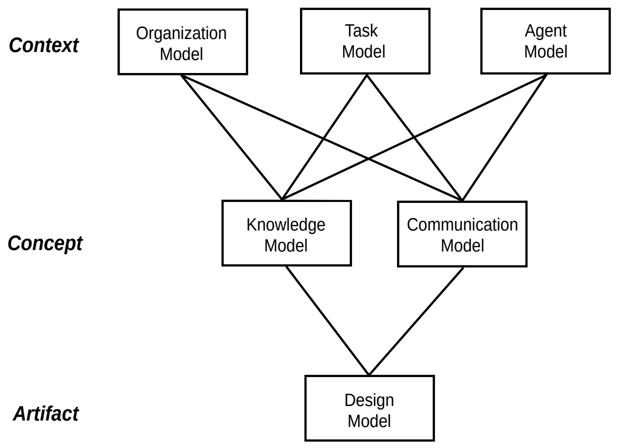

Figure 4.1 CommonKADS layers and components.

the goal of the task, control, features, environmental constraints such as quality or performance criteria, and its required capabilities, like knowledge or specific competencies. In particular, tasks are performed by agents. An information modeling view consists of the functional view (for example, the information flow), the static information structure view (for example, the structure of objects and their associations), and the control view (defining the ordering of the subtasks).

- *Agent model* describes all relevant properties like various roles, competencies, and reasoning capabilities of agents able to achieve tasks of the task model. It mainly provides a different viewpoint on information already available in other models. An agent is any actor performing some task, such as a database, an expert system, a user, or any software system. The agent model is described with the attributes: *name*, *type* (human, agent, and software system), *subclass-of* (to define inheritance), *position*, and *groups* (the groups to which an agent belongs). In addition, MAS-CommonKADS adds to this model the following components: *services*, describing specific facilities offered to other agents; *goals*, being the objectives of an agent; *reasoning capabilities* stating the requirements on the agent imposed by the task assignment; *general capabilities* such as its skills (including sensors and effectors) and agent (communication) languages; and *constraints* such as norms, preferences, and permissions of the agent.
- *Knowledge model* or *Expertise model* describes the capabilities of an agent with a bias towards knowledge-intensive problem-solving capabilities. The expertise model is divided into *domain knowledge* relevant for the application, *inference knowledge* being the basic reasoning knowledge, *task knowledge* being the goal and its decomposition into subgoals and subtasks, and *strategic knowledge*. The focus of CommonKADS lies in the definition of these models.

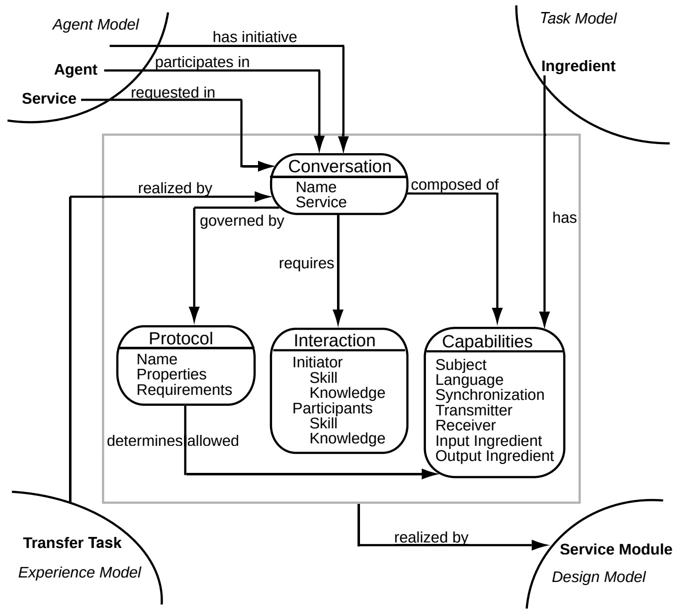

Figure 4.2 The CommonKADS coordination model. (*After:* [15].)

- *Communication model* describes—in an implementation-independent way—all the communication between agents in terms of transactions, transaction plans, initiatives, and capabilities needed in order to take part in a transaction. The main drawback of CommonKADS is the restricted communication model: due to its original aim, its focus is on the definition of human-computer interaction, and no support is provided for complex interaction between agents. As a consequence of this, MAS-CommonKADS adds an additional model, the *coordination model* (an example of which is shown in Figure 4.2), which applies different protocol techniques, as above, to extend the expressiveness of the specification method. The coordination model consists of conversations used to request a service or to request or update information; interactions defining simple interchange of messages; capabilities being the skills and knowledge of the agents involved in conversations; and protocols defining the set of rules for a conversation. The graphical notations used are, for example, message sequence charts [16] and communicating extended finite-state machines that support three types of events: message, external, and internal events, as shown in Figure 4.3.

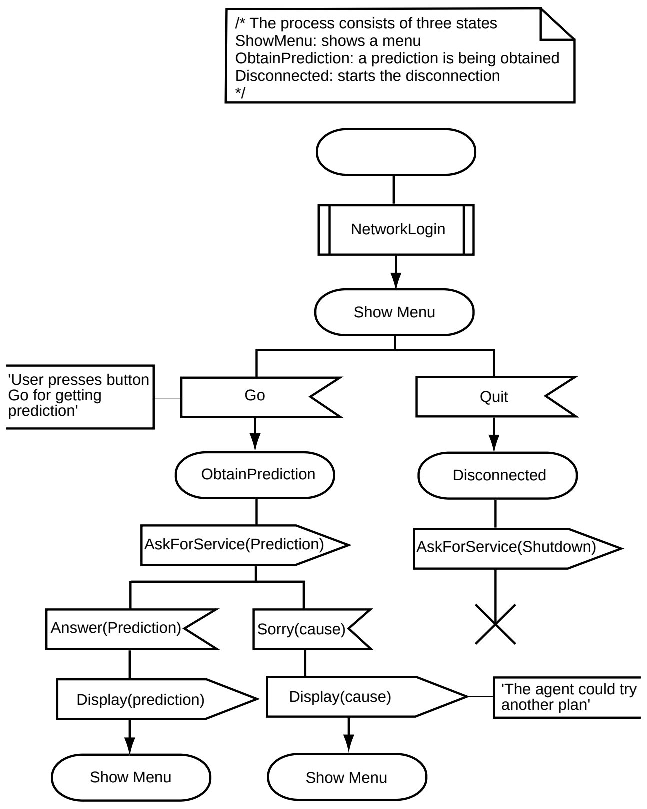
/* The process consists of three states
ShowMenu: shows a menu
ObtainPrediction: a prediction is being obtained
Disconnected: starts the disconnection
*/
NetworkLogin
Show Menu
'User presses button Go for getting prediction'
Go
Quit
ObtainPrediction
Disconnected
AskForService(Prediction)
AskForService(Shutdown)
Answer(Prediction)
Sorry(cause)
Display(prediction)
Display(cause)
'The agent could try another plan'
Show Menu
Show Menu

Figure 4.3 Extended finite-state machine in CommonKADS. (*After:* [15].)

- *Design model* describes the design of the system, its architecture, implementation platform, and software modules. Therefore a three-stage transformation process is suggested, consisting of *application design* describing the functional decomposition, object-oriented decomposition, and AI paradigms like constraint-based programming; *architectural design* defining a computational infrastructure capable of implementing the architecture; and *platform design* considering how the ideal knowledge representation and inference techniques should be implemented in the chosen software.

In addition, while the scope of CommonKADS covers the *analysis phase of software engineering*, MAS-CommonKADS extends support for the software engineering phases to the complete software life cycle, including the following (see [15]):

- *Conceptualization*: As in object-oriented analysis and design, MAS-CommonKADS starts with a use case centered approach to formalize the first description of the problem. The use cases also provide the basis for further testing.
- *Analysis*: The output of this phase is a detailed requirements specification obtained by *delimitation* to distinguish the agent-based system from the external nonagent system (resulting in a first version of the agent model and the coordination model); *decomposition* of the system based on the geographical, logical, and knowledge distribution; and *validation*, or correctness with respect to previous definitions and other models. The obtained models are the organization, task, agent, communication, cooperation, and expertise models.
- *Design*: This phase covers aspects such as *application design* through decomposition into submodules; *architecture design* through selection of a multiagent architecture and determining the infrastructure based on the applied network, used knowledge and the coordination; and *platform design*, addressing the needed software and hardware. The basis for this phase is mainly the expertise model and the task model.
- *Coding and testing*: This is performed on an individual agent basis.
- *Integration*: This relates to integration of the different individual agents and testing of the multiagent system.
- *Operation and maintenance*: This is as for any software system.

4.4 AGENT-ORIENTED APPROACHES

A perceived lack in the knowledge engineering software development methodologies was that they were not designed in the context of supporting the development of agent systems, and that as a result they had limited capability to support agent-specific functions, which could only partly be overcome by extensions such as those seen for MAS-CommonKADS. In this section, we review two main agent-oriented approaches, namely Gaia [4] with its extension, ROADMAP [19], and the SODA methodology [20].

4.4.1 Gaia and Its Extension ROADMAP

Gaia is a methodology for agent-oriented analysis and design supporting macro (societal) level as well as micro (agent) level aspects [4]. Gaia was designed to deal with coarse-grained computational systems, to maximize some global quality measure, and to handle heterogeneous agents independent of programming languages and agent architectures. It assumes static organization structures and agents that have static abilities and services, with fewer than 100 different agent types. ROADMAP extends Gaia by adding elements to deal with 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 (Agent UML, covered in Section 4.6) [21]. Here, we present a unified view of both methodologies.

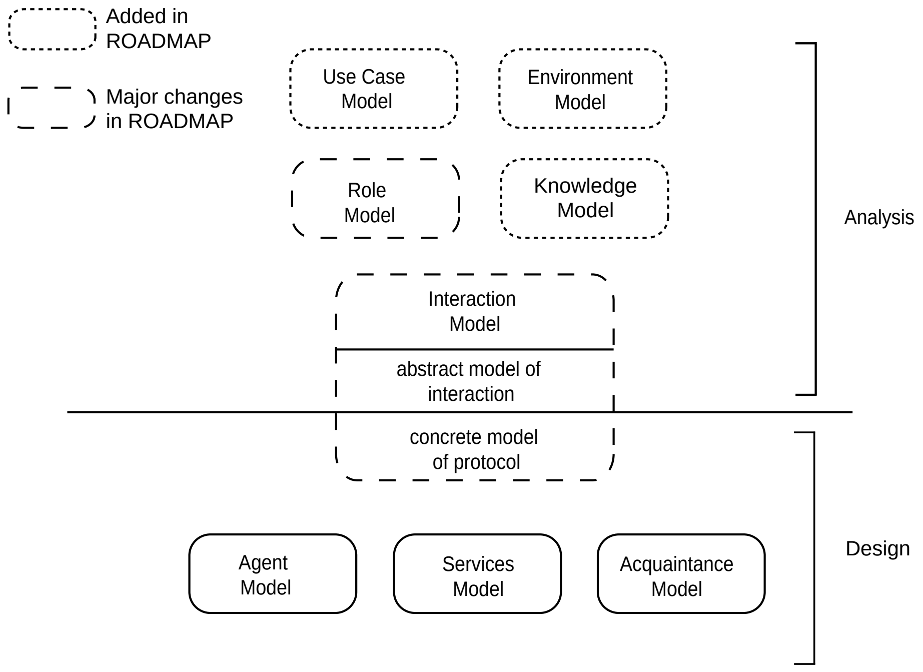

Figure 4.4 Models employed by Gaia and ROADMAP.

In Gaia, a system is developed from an organizational point of view, starting with an organizational model that is refined in subsequent development steps. It is intended to allow an analyst to progress systematically from a statement of requirements not formally described in the methodology (which in ROADMAP is mainly derived by applying use case models) to a design that is sufficiently detailed to be implemented directly, with the aim of obtaining detailed models from which the system can be constructed. The relevant models of Gaia and ROADMAP are shown in Figure 4.4.

In the analysis phase, Gaia defines the role and interaction models, whereas ROADMAP adds the use case, environment, and knowledge models. Gaia views an organization as a collection of roles standing in a certain relationship to one another, and taking part in systematic institutionalized patterns of interactions with other roles. These roles can subsequently be instantiated with actual individuals. ROADMAP starts—comparable to several object-oriented approaches—with a use case model from which it derives the environment, role, and knowledge models. The knowledge model is strongly related to the role model and the environment model. Both approaches deal with the definition of an interaction model, defining the conversation at a high level and at a low level. (ROADMAP allows designers, in addition, to apply the UML extension to sequence diagrams [21].)

In the design phase the agent model derived from roles is specified, as well as the services model and acquaintance model.

4.4.1.1 Analysis

In the analysis phase the following models are created, based on [4, 19].

- *Use case model* (only ROADMAP): Just as in the object-oriented (OO) world, ROADMAP suggests applying use cases to discover requirements in an effective and sufficient way. Similar to UML, the model includes a graphical representation of diagrams and specific text scenarios. However, the semantics of use cases is different from traditional OO approaches. In contrast to a traditional model that would assume a user to be interacting with the software system, the view taken is *imagined*, that a user interacts with a team of abstract ideal agents possessing the knowledge, ability, or mental states required to provide the best service.
- *Environment model* (only ROADMAP): The environment model, derived from the use case model, provides a holistic description of the system environment, since the ROADMAP approach targets complex open systems, usually embedded in highly dynamic and heterogeneous environments. The model is the knowledge foundation for any environmental changes during the system execution. It consists of a *tree hierarchy of zones* in the environment (for example, the Internet, a local computer, or the physical environment of a house) based on OO inheritance and aggregation and *zone schemas*, characterized by a textual description of the zones derived from the use cases and the *attributes*: *static objects* (any entity in the environment known to some agent, but with no interaction), *objects* (any entity an agent interacts with), *constraints*, *sources of uncertainty* (which have to be analyzed), and *assumptions* made about the zones.
- *Knowledge model* (only ROADMAP): The knowledge model, derived from the use case and environment model, provides a holistic description of the domain knowledge used in the system. It consists of a hierarchy of knowledge components and a description for each knowledge component, and thus identifies the knowledge required to deliver the agent behaviors in the appropriate zones for each use case. Knowledge components are assigned to roles and therefore connect the role model with the use case model and the environment.
- *Role model*: The role model identifies the key roles of the system. The roles typically correspond to individuals, departments, or organizations as in real life, and are characterized by four attributes, as follows:
  - *Responsibilities* define the functionality of an agent by giving the *liveness* properties (intuitively stating “something good happens”) and *safety* properties such as invariants.
  - *Permissions* are defined in terms of rights, with information and knowledge resources that can be used by the role, and the resource limits within which the role executor must operate.
  - The *activities* of a role are some private actions a role can perform without interaction with other agents.

Table 4.2

Role Schemas and Names in Gaia

Role Schema	**Name of Role**
Description	*Protocols and activities in which the role plays a part*
Permissions	*Rights associated with the role*
Responsibilities	
Liveness	*Liveness properties*
Safety	*Safety properties*

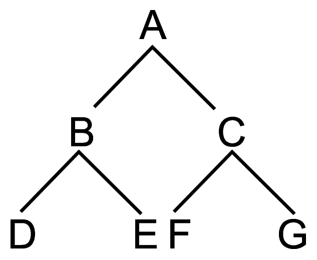

Figure 4.5 Role hierarchy in Gaia.

– Finally, *protocols* define the way that a role can interact with other roles, like the contract net (described in Chapter 2), in an abstract way.

In the design phase, the protocols are defined in more detail. Gaia's role model consists of a set of role schemata, one for each role in a system, as illustrated in Table 4.2. To this, ROADMAP adds a role hierarchy as shown in Figure 4.5. Here, the leaf nodes are atomic roles, representing characteristics of individual agents, whereas the other roles are composite roles, like A, B, and C, using aggregation. In ROADMAP, aggregation assumes a dynamic teamwork semantics; that is, the individual subroles are said to participate in the super-role, and each may own a thread of execution, a set of knowledge, and functionality. In addition, a composite role represents a localized organization or society of roles with attributes modeling social aspects (*subrole attribute*), like the organization structure, social goals, social tasks, or social laws. Moreover, the *knowledge attribute* represents local social knowledge. To extend and maintain open systems, ROADMAP allows changing the application architecture as well as the ability of individual agents during run time. This is obtained by allowing a role to have read, write, or create permissions on definitions of other roles.

- *Interaction model*: The interaction model was originally defined in Gaia for the definition of protocols. It describes the dependencies and relationships between various roles in a multiagent organization (providing a pattern of interaction). ROADMAP names this the *protocol model*, and defines in addition an *interaction model* based on AUML interaction diagrams. In our view, the latter belongs more to the design phase and therefore we discuss it in the next section. Gaia characterizes a protocol by the attribute's purpose (a brief textual description of the nature of the interaction, the *initiator* of the protocol, or the role or roles starting the conversation), the *responder* (or the role participating in the protocol), inputs and outputs (or the information used and supplied by the initiator and responder), as well as a brief textual description of any processing the protocol initiator performs during the course of the interaction (*processing*).

4.4.1.2 Design

In typical design processes such as those used in OO approaches, an abstract model of the analysis phase is transformed into a sufficiently concrete level that can easily be implemented. In contrast to this, the design phase in Gaia has the goal of developing a level of abstraction that is sufficiently specific to allow traditional design techniques to be used to implement the agents.

Both Gaia and ROADMAP define the *agent model*, *services model*, and *acquaintance model*. In addition, ROADMAP allows designers to refine the *interaction model*.

- *Interaction model*: The interaction model provides a detailed definition of the interaction between different roles or individual agents by applying AUML interaction diagrams, an extension of UML sequence diagrams, to model interaction diagrams such as the FIPA Contract Net Protocol. Note that since this work was done, a revised version has been developed in the context of FIPA.[5]
- *Agent model*: The agent model identifies the *agent types* that will make up the system, and can be thought of as a set of agent roles. The *agent instances* that will be instantiated from these types are documented by annotating agent types with, for example, $n, m..n$, or *, describing in the usual way the number of instances of an agent type. Based on efficiency aspects, an agent type implements a set of agent roles.
- *Services model*: The services model identifies the main services, defining the function of an agent as characterized by input, output, preconditions, and postconditions that are required to realize the agent's role. A service does not describe the implementation view of a specific function; instead it is derived from the list of protocols, activities, responsibilities, and liveness properties of a role. ROADMAP also takes the social and knowledge aspects of the previous models into account.
- *Acquaintance model*: The acquaintance model documents the lines of communication between the different agents. The purpose is to identify any potential communication bottlenecks that may cause problems at run time.

Examples of these models are shown in Tables 4.3, 4.4, and 4.5, loosely based on the example from [19]. The scenario assumes that the operation of a research lab is supported by an agent-based system. In Table 4.3, some roles are outlined such as *InformationExchange* for allowing agents to exchange papers and presentations between researchers and *OnlineDiscussion* for participating in

5 http://www.auml.org

Table 4.3

Partial Role Hierarchy

System	
InformationExchange	
OnlineDiscussion	
MeetingSupport	
Admin	
SysConfig	// Software and hardware configuration
MobileDevMgt	// Mobile device management
Profiles	// Students, Researchers, Staff
Install	// Install and update software
Location	// For locating people in the lab
Security	
PhysicalSecurity	
Network_Computers	
Virus_Worm_Protection	
Intrusion_Detection	
Server_Log_Analysis_and_Forensics	
User_and_Program_Authentication	
Content_Scanning	
Reliability	
Performance	

online debates. In Table 4.4, an agent class implementing the *Admin* role is shown. Finally, Table 4.5 shows the Admin role activities and protocols.

4.4.2 SODA

Another agent-oriented software engineering methodology, mainly focusing on societies similar to Gaia's organizations, is SODA (societies in open and distributed agent spaces [20]). Like ROADMAP, it addresses some of the shortcomings of Gaia such as its inadequacies 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 global behaviors not deducible from the behavior of individual agents) and *agent environments* (the space in which agents operate and interact, such as open, distributed, decentralized, heterogeneous, dynamic, and unpredictable environments), SODA provides support for modeling the *inter*agent aspects. However, *intra*agent aspects are *not* covered. Thus, 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. Just like most of the methodologies considered here, SODA supports the analysis and design phase.

Table 4.4

Example of an Agent Class

Agent Class:	AdminAgent
Role Assigned:	Admin
Description:	Taking the role of Admin and providing services for protocols and activities of Admin
Services:	QueryConfigSvc **implements** Admin.SetConfig
	SetConfigSvc **implements** Admin.SetConfig
	QueryProfileSvc **implements** Admin.QueryProfile
	UpdateProfileSvc **implements** Admin.UpdateProfile
	QueryLocationSvc **implements** Admin.QueryLocation
	WaitSvc **implements** Admin.Wait

Table 4.5

Role Description

Role Name:		Admin
Description:		General administration of a researcher's system
Subroles:		SysConfig, MobileDevMgt, Profiles, Install and Location
Knowledge:		Software and hardware types, versions and settings
		User Profile, and Location regulations
Permission:		Changes all software and hardware settings
		Changes all Profiles
		Changes all Location info
		Changes System* // reflection
Responsibilities:		
Liveness:		
Admin	=	(Work | Wait)^W
Work	=	QueryConfig* || SetConfig* || QueryProfile* ||
		UpdateProfile* || QueryLocation*
		involves sysConfig, deviceMgt, profiles, install and location
Safety:		Consistent (software and hardware settings)
		Consistent (profiles)
		Consistent (location info)
Protocols:		QueryConfig **involves** SysConfig.QueryConfig & DevMgt.SetConfig
		QueryProfile **involves** Profiles.QueryProfile
		UpdateProfile **involves** Profiles.UpdateProfile
		QueryLocation **involves** SysConfig.QueryConfig
Activities:		Wait

4.4.2.1 Analysis

During the analysis phase in SODA, the application domain is studied and modeled, the available computational resources and the technological constraints are listed, and the fundamental application goals and targets are identified. The result of the analysis phase is typically expressed in terms of high-level abstractions and relationships, providing designers with a formal or semiformal description of the intended overall application structure and organization. The SODA analysis phase delivers three models [20]:

- *Role model*: The application goals are modeled in terms of the tasks to be achieved, which are associated with *roles* and *groups*. *Tasks* are expressed in terms of the responsibilities they involve, the competencies they require, and the resources they depend upon. Responsibilities are expressed in terms of the states of the world that should result from task accomplishment, while tasks are classified as either individual or social. Typically, social tasks are those that require a number of different competencies and access to several different resources, whereas individual tasks are more likely to require well-delimited competence and limited resources. Each individual task is associated with an individual role, which is defined in terms of its individual task, its permissions to access resources, and the corresponding interaction protocol. Analogously, social tasks are assigned to groups. A group is defined in terms of its social tasks, its permissions to access the resources, the participating social roles, and the corresponding interaction rule. A social role describes the role played by an individual within a group, and may either coincide with an already defined (individual) role, or be defined *ex novo*, in the same form as an individual role, by specifying an individual task as a subtask of one assigned to a group to which the individual belongs.
- *Resource model*: The application environment is modeled in terms of the *services* available, which are associated with abstract *resources*. A resource is defined in terms of the service it provides, its access modes, the permissions granted to roles and groups to exploit its service, and the corresponding interaction protocol. If a task assigned to a role or a group requires a given service, the access modes are determined and expressed in terms of the granted *permission* to access the resource in charge of that service. Such a permission is then associated with that role or group.
- *Interaction model*: The interaction involving roles, groups, and resources is modeled in terms of *interaction protocols*, expressed as *information* required and provided by roles and resources in order to accomplish its individual tasks, and *interaction rules*, governing interaction among social roles and resources so as to make the group accomplish its social task.

4.4.2.2 Design

Design in SODA is concerned with the representation of the abstract models resulting from the analysis phase in terms of the design abstractions provided by the methodology. The result of the design phase is typically expressed in terms of abstractions that can be mapped one-to-one onto the actual components of the deployed system.

- *Agent model*: An *agent class* is defined as a set of (one or more) individual and social roles. As a result, an agent class is characterized by the tasks, the set of permissions, and the interaction protocols associated with its roles. Agent classes can be further characterized in terms of other features: their *cardinality* (the number of agents of that class), their *location* (with respect to the topological model defined in this phase—either fixed for static agents, or variable for mobile agents), and their *source* (from inside or outside the system, given the assumption of openness). Since SODA only deals with *inter*agent aspects (the observable behavior of an agent), the *intra*-agent aspects are not covered in SODA, and can be based on results of other methodologies.
- *Society model*: Each group is mapped onto a *society of agents*. An agent society is first characterized by the social tasks, the set of the permissions, the participating social roles, and the interaction rules associated with its groups. The main issue in the society model is how to design interaction rules so as to make societies accomplish their social tasks. Since this deals with managing agent interaction, the problem of achieving the desired social behavior by means of suitable social rules is basically a *coordination* issue. As a result, societies in SODA are designed around *coordination media*: the abstractions provided by coordination models for the coordination of multicomponent *systems*. Thus, the first task in the design of agent societies is the choice of the most fit coordination model, the one providing the abstractions that are expressive enough to model the society interaction rules. In this way, a society is designed around coordination media embodying the interaction rules of its groups in terms of *coordination rules*. The behavior of suitably designed coordination media, along with the behavior of the agents playing social roles and interacting through such media, causes an agent society to pursue its social tasks as a whole. This allows societies of agents to be designed as first-class entities.
- *Environment model*: Resources are mapped onto *infrastructure classes*. An infrastructure class is first characterized by the services, the access modes, the permissions granted to roles and groups, and the interaction protocols associated with its resources. Infrastructure classes can be further characterized in terms of other features: their cardinality (the number of infrastructure components belonging to that class), their location (with respect to topological abstractions), their *owner* (which may or may not be the same as the owner of the agent system, given the assumption of decentralized control). The design of the components belonging to an infrastructure class may follow the most appropriate methodology for that class; since SODA does not specifically address these issues, components like databases, expert systems, or security facilities can all be developed according to the most suitable specific methodology. What is determined by SODA is the outcome of this phase, the services to be provided by each infrastructure component, and the interfaces resulting from its associated interaction protocols. Finally, SODA assumes that a topological model of the agent environment is provided by the designer but does not provide for topological abstractions on its own, since any system or any application domain may call for different approaches to the problem.

4.4.3 Comparison

Both agent-oriented methodologies presented in this section deal with analysis as well as the design phase, resulting in a design specification that can either be directly mapped onto an underlying system (ROADMAP, SODA) or to which state-of-the-art technologies can be applied to refine the models to obtain a well-designed model of the system (Gaia).

The role model of SODA is similar to the role model of Gaia and ROADMAP, since both associate different functionality with an agent satisfying a role, but with different ways of deriving them. Interactions are covered in all three approaches, with the difference that SODA adds interaction rules among groups within a society, whereas Gaia and ROADMAP deal with interaction between agents of specific roles. The motivation behind the resource model and the environment model of ROADMAP is similar, but its expression is based on different views on the environments. The use case model and knowledge model of ROADMAP seem to be necessary models to ensure that the requirements are well-defined, and to cope with the information (knowledge) available and used in the system. The SODA agent model deals with similar aspects to the Gaia and ROADMAP role and agent model with some smaller differences; in particular, the different aspects are totally shifted to the design phase in SODA.

4.5 METHODOLOGICAL EXTENSIONS TO OBJECT-ORIENTED APPROACHES

A good procedure (if not the only viable one) for the 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 present, agent-oriented software engineering still lacks the availability of suitable software processes and tools, yet the Unified Modeling Language (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, for example, [21] for a discussion of this topic) 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 top of their object-oriented counterparts seems appropriate, 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 developer community which, as of today, prevails in industry.

In this section, we more closely examine agent methodologies that directly extend object-oriented approaches, while in the following section we consider UML notations and extensions available for the specification of agent-based systems. Since most of the notations use graphical representations of software artifacts, we use examples taken from the original research papers, but provide only little explanation for reasons of brevity.

4.5.1 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 [8, 9, 22]. 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 (an agent's beliefs, goals, and plans) must be modeled for each agent.

The methodology for the *external aspects* consists of four major steps (as described in [9]):

1. Identify the roles of the application domain, and identify the lifetime of each role; elaborate an agent class hierarchy. The initial definition of agent classes should be quite abstract, not assuming any particular granularity of agency.
2. For each role, identify its associated responsibilities and the services provided and used to fulfill those responsibilities. As well as services provided to and by other agents upon request, services may include interaction with the external environment or human users. Conversely, a responsibility may induce a requirement that an agent be notified of conditions detected by other agents or users. Decompose agent classes to the service level.
3. For each service, identify the interactions associated with the provision of the service, the performatives (speech acts) required for those interactions, and their information content. Identify events and conditions to be noticed, actions to be performed, and other information requirements. Determine the control relationships between agents. At this point, the internal modeling of each agent class can be performed.
4. Refine the agent hierarchy. Where there is commonality of information or services between agent classes, consider introducing a new agent class that can be specialized by existing agent classes to encapsulate common aspects. Compose agent classes, via inheritance or aggregation, guided by commonality of lifetime, information and interfaces, and similarity of services. Introduce concrete agent classes, taking into account implementation-dependent considerations of performance, communication costs and latencies, fault-tolerance requirements, and so on. Refine the control relationships. Finally, based on considerations of lifetime and multiplicity, introduce agent instances.

The methodology for *internal modeling* can be expressed as two steps, as shown in [9]:

1. Analyze the means of achieving the goals. For each goal, analyze the different contexts in which the goal must be achieved. For each of these contexts, decompose each goal into activities, represented by subgoals, and actions. Analyze the order and the conditions under which these activities and actions need to be performed, the way in which failure should be addressed, and generate a plan to achieve the goal. Repeat the analysis for subgoals.
2. Build the beliefs of the system. Analyze the various contexts, and the conditions that control the execution of activities and actions, and decompose them into component beliefs. Analyze the

input and output data requirements for each subgoal in a plan, and make sure that this information is available either as beliefs or as outputs from prior subgoals in the plan.

These steps are iterated as the models that capture the results of the analysis are progressively elaborated, revised, and refined. Refinement of the internal models feeds back to the external models; building the plans and beliefs of an agent class clarifies the information requirements of services, particularly with respect to monitoring and notification. Analyzing interaction scenarios, which can be derived from the plans, may also lead to the redefinition of services. For each of these views different models are described, based on [8, 9].

External Viewpoint

The external view is characterized by two models that are largely independent of the underlying BDI architecture.

- The *agent model* describes the hierarchical relationship among different abstract and concrete agent classes (*agent class model*), and identifies the agent instances that may exist within the system, their multiplicity, and the time at which they come into existence (*agent instance model*).
- An *agent class model* is similar to a UML class diagram denoting both abstract and concrete (instantiable) agent classes. Abstract classes are distinguished by some kind of stereotype. Inheritance and aggregation are defined in a similar way to UML. Other associations between agent classes are not allowed. Agent classes may have attributes but not operations. Attributes may not be arbitrary user-named data items, but are restricted to a set of predefined reserved attributes. For example, each class may have associated belief, goal, and plan models, specified by the attributes, beliefs, goals, and plans. Other attributes of an agent class include its belief-state-set and goal-state-set, which determine possible initial mental states. Particular elements of these sets may then be specified as the default initial mental state for the agent class, via the initial-belief-state and initial-goal-state attributes. Multiple inheritance is permitted. As usual, inheritance denotes an *is-a* relationship, and aggregation denotes a *has-a* relationship, but in the context of an agent model these relationships have a special semantics. Agents inherit and may refine the belief, goal, and plan models of their superclasses. Note that it is, for example, the set of plans that is refined, rather than the individual plans. Aggregation denotes the incorporation within an agent of subagents that do not have access to each other’s beliefs, goals, and plans.
- An *agent instance model* is an instance diagram that defines both the static agent set instantiated at compile-time (marked by some kind of stereotype) and the dynamic agent set instantiated at run-time. In contrast to UML class and object diagrams, an agent instance is linked to a concrete agent class by an instantiation edge, whereas in UML instances are underlined. Static instances must be named, but the naming of dynamic instances may be deferred until their instantiation. A multiplicity instantiation is supported. The initial mental state of an agent instance may be specified by the *initial-belief-state* and *initial-goal-state* attributes whose values are particular elements of the belief and goal state sets of the agent class. If not specified, the defaults are

the values associated with the agent class. For dynamic agent instances, these attributes may be overridden at the time the agent is created.

- The *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 interagent communication, and for communication between agents and other system components, such as user interfaces.

Internal Viewpoint

BDI agents are 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:

- The *belief model* describes the information about the environment and the 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 that define the domain of the beliefs of an agent class. A belief state is a set of instance diagrams that define a particular instance of the belief set. Formally, it is defined by set of typed predicates whose arguments are terms over a universe of predefined and user-defined function symbols. The classes and instances defined therein correspond, in many cases, to real entities in the application domain but, unlike an object model, the definitions do not define the behaviors of these entities, since they represent an agent's *beliefs* about those entities. A class in a belief set diagram serves to define the type signatures of attributes of an object, functions that may be applied to the object, and other predicates that apply to the object, including actions, which have a special role in plans. Attributes, which define binary predicates, are specified in the usual way.
- The *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* that 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. Formally, a goal set is a set of goal formula signatures. Each such formula consists of a modal goal operator applied to a predicate from the belief set. The modal goal operators are: *achieve* (denoted by !) representing a goal of achievement, *verify* (denoted by ?) representing a goal of verification, and *test* (denoted by $) representing a goal of determination. Goal formulas occur within activities in the bodies of plans and within their activation events.
- The *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 similarly 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, there are several differences: activities may be subgoals, denoted by formulas from the agent's goal set; conditions are predicates from the agent's belief set; actions include those defined in the

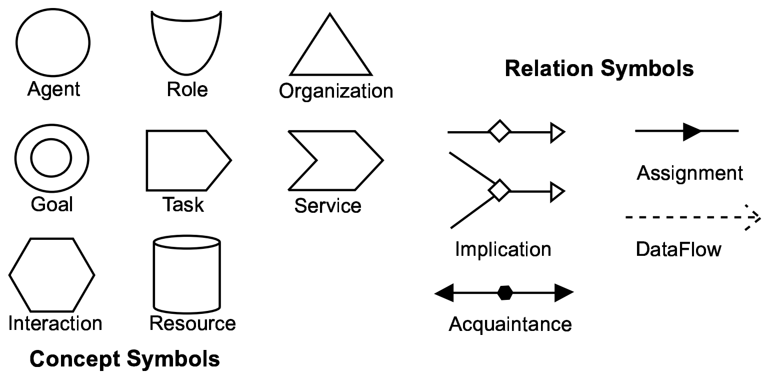

Figure 4.6 Concept symbols and relation symbols in MESSAGE.

belief set, and built-in actions. The latter include *assert* and *retract*, which update the belief state of the agent.

4.5.2 MESSAGE

MESSAGE (methodology for engineering systems of software agents) [23] is a methodology that builds upon best practice methods in current software engineering such as UML for the analysis and design of agent-based systems. It consists of (1) applicability guidelines; (2) a modeling notation that extends UML by agent-related concepts (inspired, for example, by Gaia); and (3) a process for analysis and design of agent systems based on RUP. 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 [23]. For details of the example, refer to this paper. The concept and relation symbols used are shown in Figure 4.6.

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 as follows:

- *Organization model*: The organization model captures the overall structure and 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 suborganizations such as departments, divisions, and sections, expressed through power relationships (for example, superior-subordinate relationships). Moreover, it provides the *social view* characterizing the overall behavior of the group, while the agent model covers the *individual view* dealing with

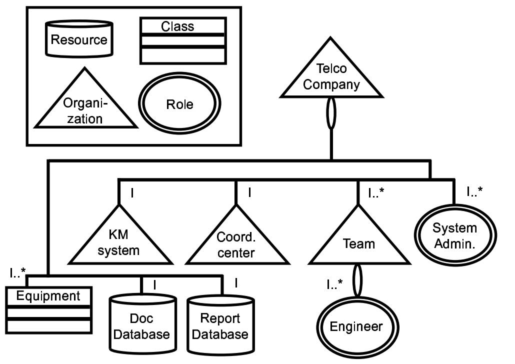

Figure 4.7 Examples of organization diagrams: structural relationships.

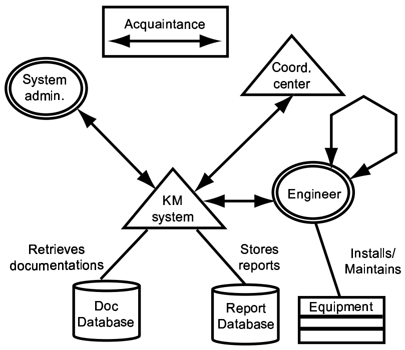

Figure 4.8 Examples of organization diagrams: acquaintance relationships (analysis phase 0 and 1).

the behavior of agents to achieve common or social goals. It offers software designers a useful abstraction for understanding the overall structure of the multiagent system, the agents themselves, the resources involved, the role of each agent, their responsibilities, and those tasks that are achieved individually and those achieved through cooperation. Different types of organization diagrams are available in MESSAGE to support the graphical representation of social concepts (see Figures 4.7 and 4.8).

- *Goal/task model*: The goal/task model defines the goals of the composite system (the agent system and its environment) and their decomposition into subgoals. It also covers 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 subtasks, as well as describing 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. Finally, 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, for example, decomposition of high-level goals into subgoals, 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 are needed to satisfy agent goal commitments. Such state transitions are specified as precondition and postcondition attribute pairs. Actions are atomic tasks that can be performed by agents to satisfy their goal commitments. Task inputs are model elements (adapted from UML defining elements composing models) that are processed in the task, while 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. Examples of goals and tasks are shown in Figures 4.9 and 4.10.

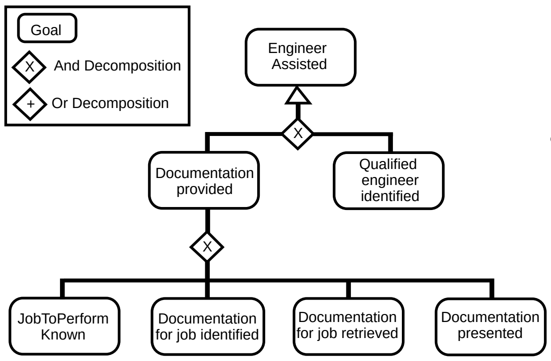

Figure 4.9 Example of a goal implication diagram.

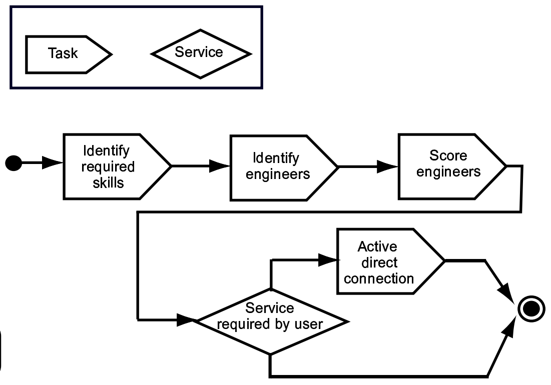

Figure 4.10 Example of a workflow diagram.

- *Agent/role model*: The agent model consists of a set of individual agents and roles. The relationship between a role and an agent is defined analogously 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. One aspect of the agent model is that it 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 that includes the agent's goals and the services (or the functional capability) they provide. This contrasts with the external perspective provided by the organization model. For each agent or role, the agent model uses schemata supported by diagrams to define its characteristics, such as the goals it is responsible for, the events it needs to sense, the resources it controls, the tasks it knows how to perform, *behavior rules*, and so on. An example of an agent model is illustrated in Figure 4.11.

- *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* (by which all entities in the domain are classified in classes, and each class groups all entities with a common structure) and *relational* (by which 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 of (possibly zero) attributes with values that can belong to primitive types or that can be instances of other domain-specific classes. 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 in Figure 4.12.

- *Interaction model*: The interaction model is concerned with capturing the way in which agents (or roles) exchange information with each another (as well as with their environment). The

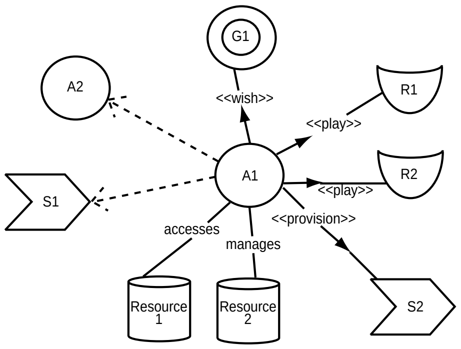

Figure 4.11 Example of agent diagram.

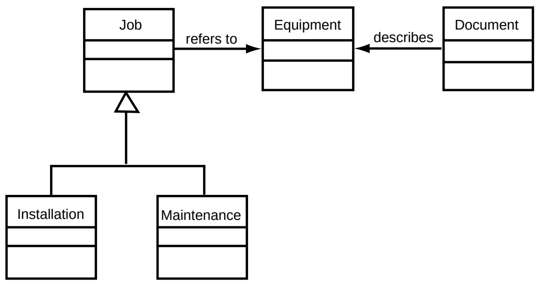

Figure 4.12 Example of domain model as UML class diagrams.

content of the messages within an interaction may be described in the domain model. Interactions are specified from both a high-level perspective (based on the interaction protocols from the Gaia methodology, described in Section 4.4.1) and a low-level perspective (based on the UML interaction protocols, described in Section 4.6.1). For each interaction among agents and roles, the model shows the initiator, the collaborators, the motivator (which, generally, is a goal for which the initiator is responsible), the relevant information supplied or achieved by each participant, the events that trigger the interaction, other relevant effects of the interaction (such as an agent becoming responsible for a new goal). Larger chains of interaction across the system (corresponding, for example, to use cases) can also be considered, such as delegation or workflows. An example of interaction is shown in Figure 4.13 and later in Section 4.6.1 on agent interaction diagrams.

4.5.3 Tropos

Tropos[6] [3, 24] 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 [25]. Some elements of UML (like class, sequence, activity, and interaction diagrams) are also adopted for modeling object and process perspectives. Tropos also uses the concepts of i*,[7] such as actors (where actors can be agents, positions, or roles), as well as social dependencies among actors (including goals, soft goals, tasks, and resource dependencies), which are embedded in a modeling framework that also supports generalization, aggregation, classification, and the notion of contexts [26]. Thus, Tropos was developed around two key features. First, the notions of agents, goals, plans, and various other knowledge-level concepts are provided as fundamental primitives used uniformly throughout the software development process. Second, a crucial role is assigned to requirements analysis and

6 http://www.cs.toronto.edu/km/tropos
7 http://www.cs.toronto.edu/km/istar

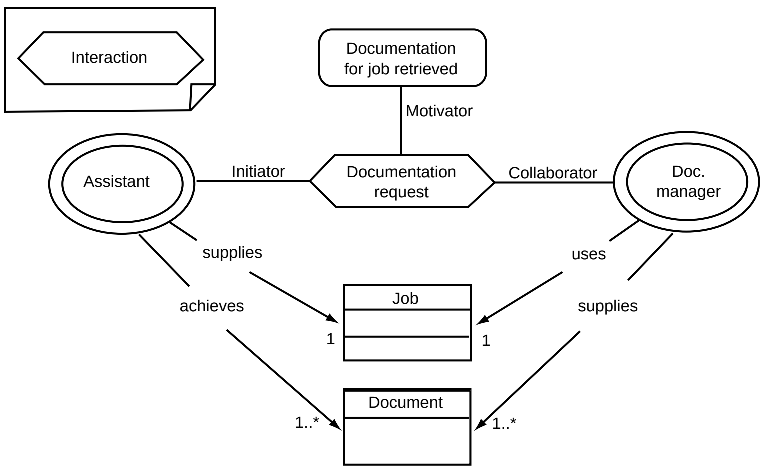

Figure 4.13 Example of a MESSAGE interaction.

specification when the system-to-be is analyzed with respect to its intended environment using a phase model, as follows:

- *Early requirements*: Identify relevant stakeholders (represented as actors), along with their respective objectives (represented as goals).
- *Late requirements*: Introduce the system to be developed as an actor, describing the dependencies to other actors and indicating the obligations of the system towards its environment.
- *Architectural design*: Introduce more system actors with assigned subgoals or subtasks of the goals and tasks assigned to the system.
- *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 Tropos specification uses the notation illustrated in Figure 4.14, and also makes use of the types of models described here [3]:

- *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 from their goals and dependencies for goal achievement, as shown in Figure 4.15. An actor has strategic goals and intentionality, and represents a physical agent (such as a person) or a software agent, as well as

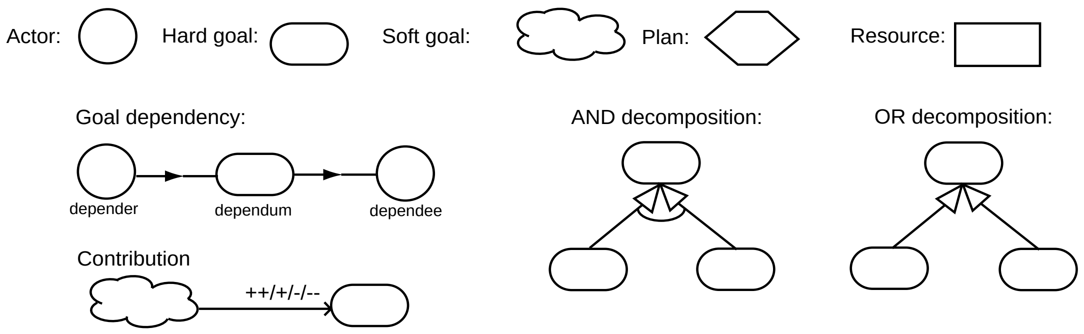

Figure 4.14 Examples of Tropos notation.

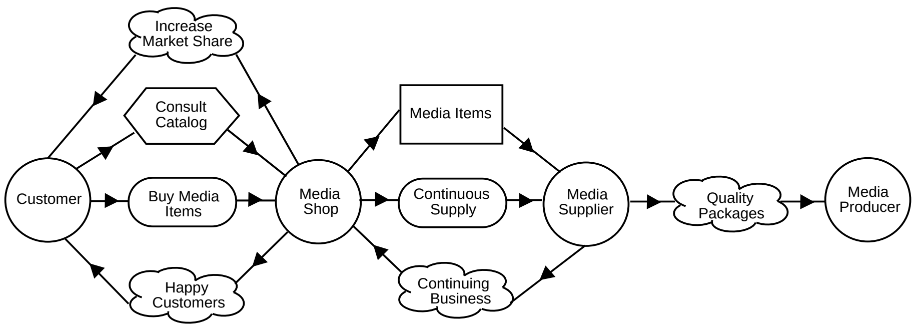

Figure 4.15 Actor diagram in Tropos. (*After:* [26].)

a role (which is an abstract characterization of the behavior of an actor within some specialized context) or a *position* (or 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 interdependencies 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.

- *Goal and plan models*: Goal and plan models allow the designer to analyze goals representing the strategic interests of actors and plans. A goal is satisfied from the perspective of a specific actor by using three basic reasoning techniques:
  - *Means-ends analysis*, refining a goal into subgoals in order to identify plans, resources, and soft goals that provide a means for achieving the goal (the *end*);
  - *Contribution analysis*, pointing out goals that can contribute positively or negatively in reaching the goal being analyzed;
  - *AND/OR decomposition*, allowing the combination of AND and OR decompositions of a root goal into subgoals, thereby refining a goal structure.

  Two kinds of goals are distinguished, namely hard goals and soft goals, the latter having no clear-cut definition or criteria as to whether they are satisfied. Goal models are first developed during early requirements using the initially identified actors and their goals.
- *Capability diagram*: A capability, modeled either textually (for example, as a list of capabilities for each actor) or as capability diagrams using UML activity diagrams 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 objects are used to model beliefs. 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 [21].

4.5.4 Prometheus

Like Tropos, Prometheus [27] is an iterative methodology covering the complete software engineering process and aiming at the development of intelligent agents (in particular BDI agents) using goals, beliefs, plans, and events, resulting in a specification that can be implemented with JACK [25]. The Prometheus methodology covers three phases, namely those of system specification, architectural design, and detailed design. Figure 4.16 illustrates the Prometheus process [27].

In the following, we describe the three phases of the Prometheus methodology [27]. More details can be found at the Prometheus Web site.[8]

- *System specification*: The system specification focuses on identifying the basic functions of the system, along with inputs (percepts), outputs (actions), and their processing (for example, how percepts are 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 mere analysis of the system functions in isolation.

8 http://www.cs.rmit.edu.au/agents/SAC/methodology.shtml

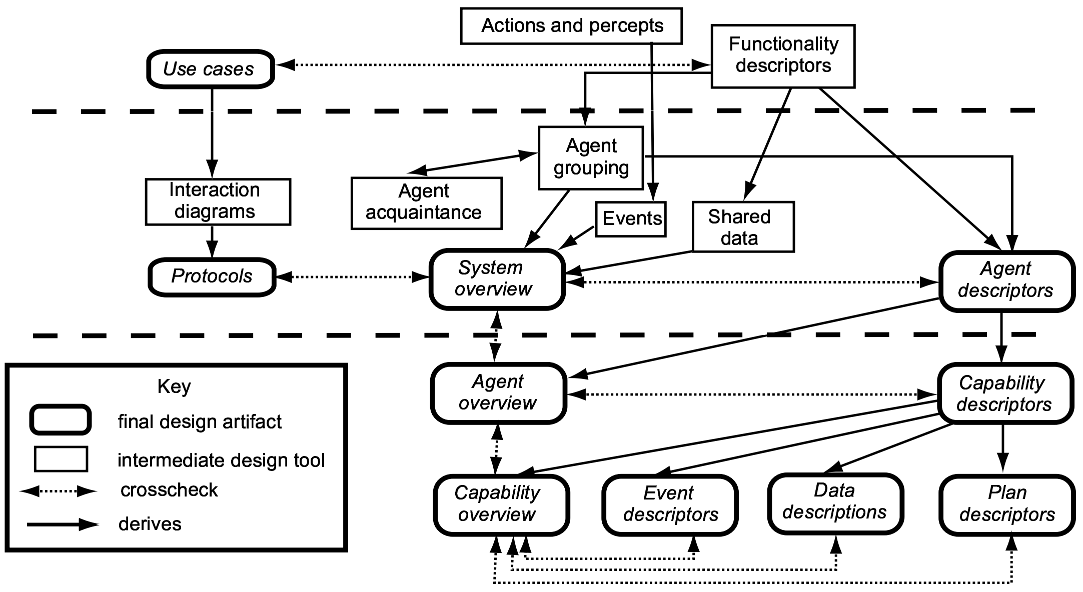

Figure 4.16 Prometheus process overview.

- *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 [21]) fully specifying the interaction between agents. Agent messages are also identified, forming the interface between agents. Data objects are specified using traditional object-oriented techniques. Using the examples from [28], the diagrams are as illustrated in Figure 4.17.

- *Detailed design*: The detailed design phase describes the internals of each agent and the way in which 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. The outcomes from this phase are *agent overview diagrams* (see Figure 4.18) providing the agent's top-level capabilities, *capability diagrams* (see Figure 4.19), detailed *plan descriptors*, and *data descriptions*. Capabilities can be nested within other capabilities so that the model supports an arbitrary number of 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.

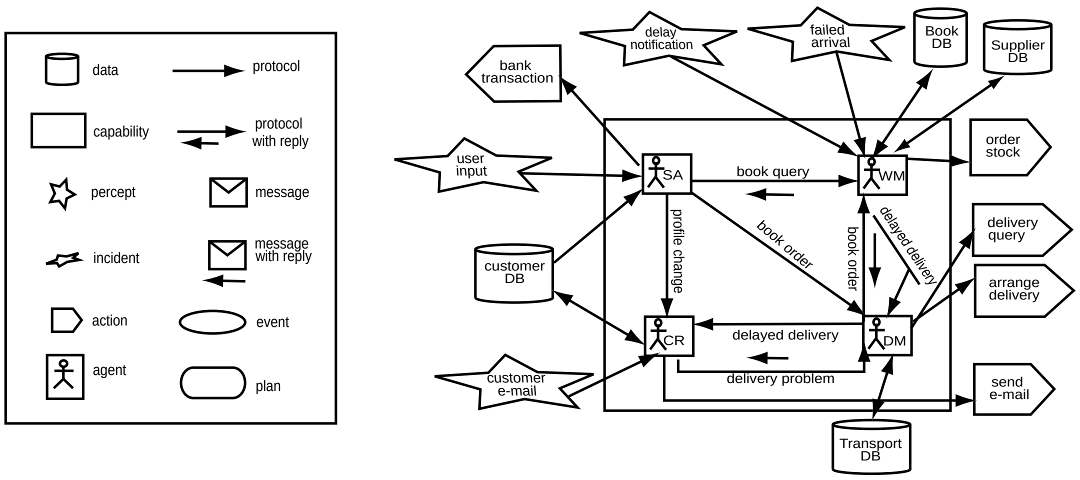

Figure 4.17 Example of system overview diagram.

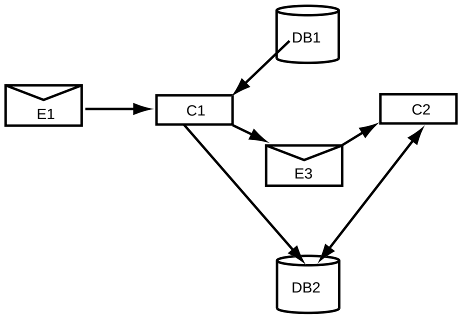

Figure 4.18 Example of agent overview diagram.

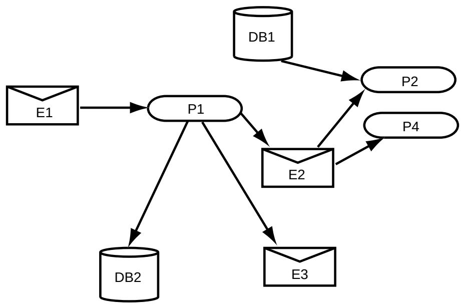

Figure 4.19 Example of capability overview diagram. (*After:* [28].)

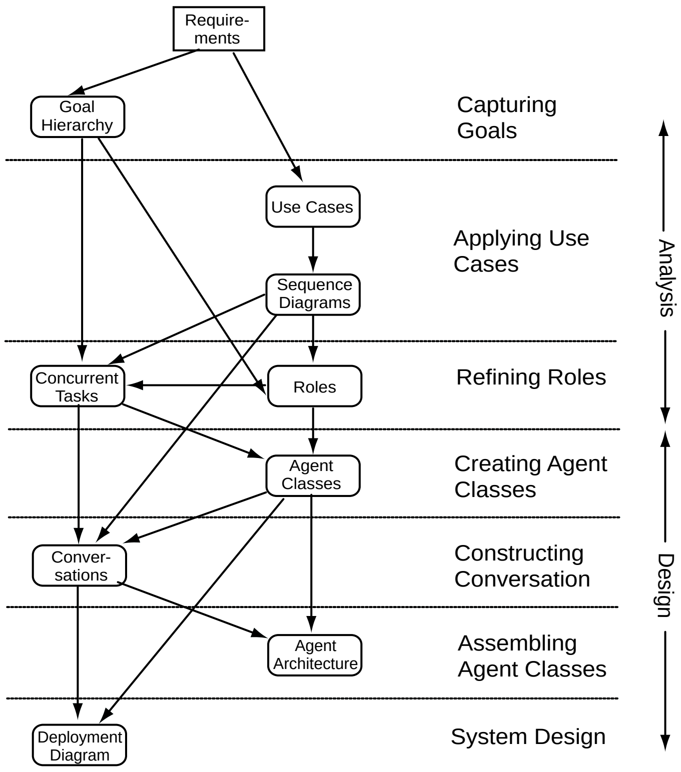

Figure 4.20 MaSE software engineering process. (*After:* [30].)

4.5.5 MaSE

Multiagent systems engineering (MaSE) has been developed to support the complete software development life cycle from problem description to realization. It offers an environment for analyzing, designing, and developing heterogeneous multiagent systems independent of any particular multiagent 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. We base the presentation in this section on [29], but for details we refer the reader to [30, 31]. The complete software engineering process is depicted in Figure 4.20.

The MaSE methodology is heavily based on UML and RUP. The software development process is focused on analysis and design, with the different models to be covered as follows:

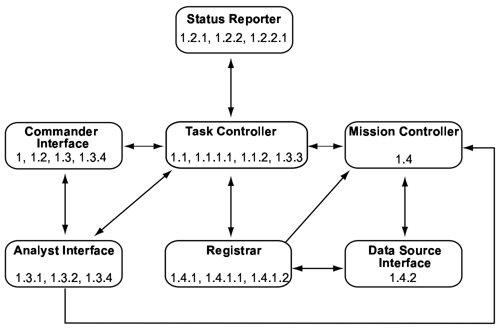

Figure 4.21 Traditional role model.

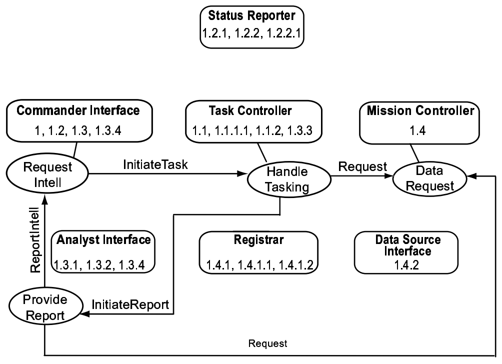

Figure 4.22 MaSE role model.

- *Capturing goals*: In this phase, the initial requirements are transformed into a structured set of system goals, which are 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. The goals are thus organized by importance in a goal hierarchy diagram. Each level of the hierarchy contains goals that are roughly equal in scope, and all subgoals 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 for which the entity is responsible. MaSE allows a traditional role model and a methodology-specific role model including information on interactions between role tasks, as shown in Figures 4.21 and 4.22. Interactions are represented by the ellipses, while the numbering below role names indicates which goals are associated with each role.
- *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, such as BDI agents, reactive agents, and so on.

- *System design*: System design 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.

4.5.6 PASSI

PASSI (process for agent societies specification and implementation)[9] [32] is an agent-oriented iterative requirement-to-code methodology for the design of multiagent systems mainly driven by 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 provide a dedicated design environment. In particular, automatic code generation for the models is partly supported, and one focus of the work lies in patterns and code reuse. The review in this section is based on [32]. The models and phases of the PASSI methodology are shown in Figure 4.23.

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 below.

- *System requirements model*: The system requirements model is obtained in different phases: The *domain description* phase results in a set of use case diagrams in which scenarios are detailed using sequence diagrams. Based on use cases, the next phase of *agent identification* defines packages in which the functionality of each agent is grouped, and activity diagrams for the task specification of the agent concerned. That is, in contrast to most agent-oriented methodologies, agents are identified based on their functionality and not on their roles. The *role identification* phase is a functional or behavioral 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 agent and one with the tasks of the interacting agent.
- *Agent society model*: The agent society model is also derived in several phases. The *ontology description* describes the agent society or organization from an ontological point of view. Thus, two diagrams are introduced, the *domain ontology description* and the *communication ontology description*, usually presented using class diagrams and XML Schemas for textual representation. The *role description* phase models the life of the agents in terms of their roles, so that social or organizational roles and behavioral roles are represented by class diagrams in which roles are classes grouped in packages representing the agents. In particular, role changes can be defined. Roles are obtained by composing several tasks (based on the functionality of an agent).
- *Agent implementation model*: Agent implementation covers the *agent structure definition* and the *agent behavior description* phases. The former describes the *multiagent 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. The latter phase

9 http://mozart.csai.unipa.it/passi

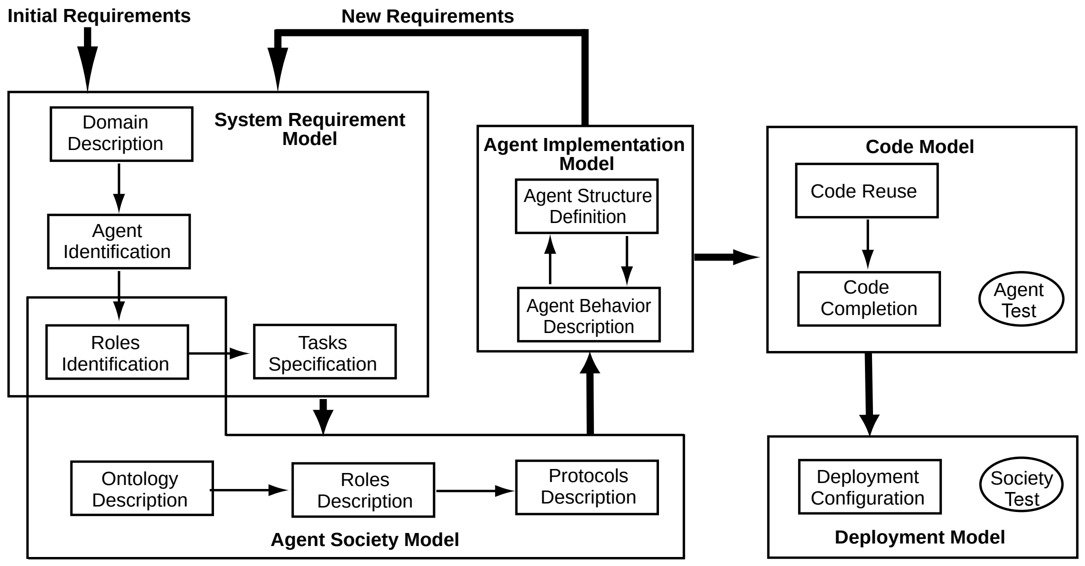

Figure 4.23 Models and phases of the PASSI methodology.

describes the *single-agent level*, which defines a 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, are represented as a class.

- *Code model*: Based on the FIPA standard architecture (see Chapter 5), standard code pieces are available for reuse 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.

4.5.7 Comparison

The six methodologies reviewed in this section all reflect the view of an agent as an autonomous entity that attempts to reach its goals in a dynamic environment. All these approaches assume an explicit (symbolic) mental model of the agent including notions such as knowledge or beliefs, goals, roles, and some sort of means to achieve goals (such as intentions or plans). In this respect, there are minor variations in the concepts that are supported by the different approaches and in the terminology. For example, the concept of a plan in the BDI model corresponds to that of a task in MESSAGE. Also, there is a common distinction between some sort of micromodel describing the agent, and a macromodel describing the multiagent system (or, more accurately, the agent's model of the multiagent system). In different methodologies, this is covered by the concepts of an external model in BDI, organizational and interaction models in MESSAGE, capability and agent interaction diagrams in Tropos, or acquaintance models in Prometheus. Differences in this respect are fairly minor.

The main criteria that allow us to compare (and differentiate between) the different approaches discussed in this section are:

- Their degree of coverage of the software development process;
- The quality of the tools provided and compatibility with software development standards.

In this respect, the BDI methodology described in Section 4.5.1 is the first and oldest attempt to provide development support for BDI agents. It is based on the proprietary dMARS model and does not provide any standard-compatible tools. What makes this approach unique is the fact that the authors were the first to recognize the importance of agent technology as a software engineering paradigm and the need to provide developers with support in designing agent-based software. MESSAGE focuses on the analysis and early design phase, while extending UML and providing a design and analysis process based on RUP, which makes the methodology more easily accessible for software engineers with an object-oriented mindset. The differentiating factor of the Tropos methodology is that it provides extensive support for the requirements analysis and, in particular, the early requirements analysis, which is beyond the scope of most other approaches. Like MESSAGE, Tropos relies on UML and provides some extensions to UML. Prometheus provides a wide process coverage ranging from analysis to detailed design. In particular, it allows designers to specify, for example, interaction diagrams using Agent UML, which may turn out to be advantageous if Agent UML becomes the predominant UML extension for implementing agent-based systems. The MaSE methodology focuses on the phases of analysis and design. Being based on UML and RUP, MaSE provides interesting practical features such as the ability to track changes made throughout the different phases of the process. Finally, the interesting aspect of the PASSI methodology is that it provides a dedicated design environment via a Rational Rose plug-in, as well as support for automated code generation, patterns and code reuse. As a possible limitation, PASSI has been driven by work on robotics applications, which may impose some constraints on its suitability for other domains.

To conclude, none of the more recent methodologies is definitely better than the other. Also none of them has yet reached commercial status, so using them to develop agent-based software may require some patience and goodwill from the software designer. On the other hand, they support the design of agent-based, proactive, open systems at a level of abstraction that greatly extends that of state-of-the-art object-oriented methodologies. In the end, the quality of tool support may be the main factor for choosing between them.

4.6 MODELING NOTATIONS BASED ON UML: AGENT UML

The UML modeling notation has been applied by many for the modeling of different aspects of agent-based software systems. While some approaches (such as [33, 34]) use plain UML 1.4 as a

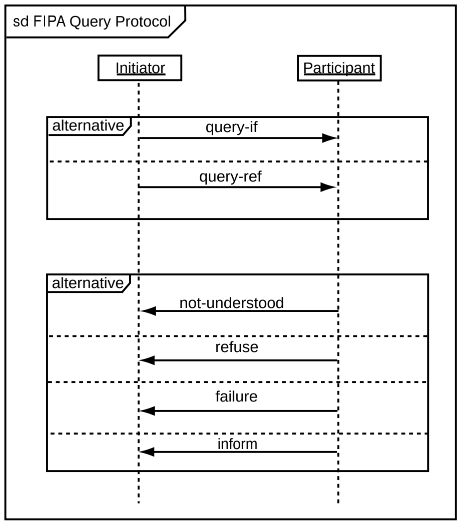

Figure 4.24 FIPA Query Protocol.

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 [21]. The forthcoming UML 2 standard will address some current limitations, and some parts of a UML extension for agent-based systems will be taken into consideration in UML 2. Consequently, in what follows, we only present those extensions of UML 1.4 for which, to our knowledge, no updated version based on UML 2 is available. The notion of Agent UML will be used, summarizing any extensions of UML to agents. UML (extensions) can be used to define interaction protocols, roles, societies, agent classes, ontologies, and plans. In the following, we show how these specific aspects of a multiagent system are modeled using UML.

4.6.1 Interaction Protocols

One of the first extensions to UML, in particular sequence diagrams, was proposed in [21, 35]. This notation was also applied as a basis for the specification of FIPA interaction protocols. In the meantime, this description has been adapted to UML 2. An agent interaction protocol [36] can thus be represented as a sequence diagram, as shown in Figure 4.24.

The figure defines a sequence diagram (the FIPA Query Protocol in the example) for a specific protocol between two individual agents (individuals are indicated by underlining the agent names, Initiator and Participant). Alternatively, such a protocol can start with a *query-if* or a *query-ref* (depicted by the alternative box) and the Participant can answer with either a *not-understood*, *refuse*,

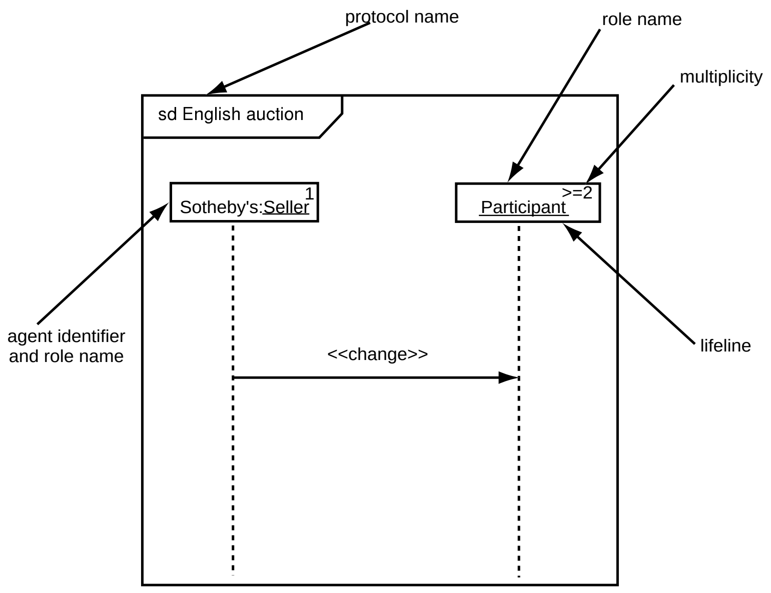

Figure 4.25 Detailed view of interaction protocol definitions.

failure, or *inform*. The vertical lifeline (dotted line) defines the lifetime of the agents, which can explicitly be stopped by an x. Role changes are marked by the stereotype <<change>>. Details are shown in Figure 4.25. Messages can be sent between agents, but also within an agent as shown in Figure 4.26.

This extension distinguishes between asynchronous and synchronous messages, takes blocking, nonblocking, and time constraints into consideration, and allows the definition of templates (interaction diagrams with formal parameters), which can then be instantiated in different contexts, like a generic FIPA contract net protocol. It applies UML 2 concepts such as Alternative, Option, Break, Parallel, Weak Sequencing, Strict Sequencing, Negative, Critical Region, Ignore/Consider, Assertion, and Loop.

In addition, however, UML 2 demands the definition of further diagram types:

- The *interaction overview diagram* is a diagram that depicts interactions through a variant of activity diagrams in a way that promotes overview of the control flow. It focuses on the overview of the flow of control, in which each node can be an interaction diagram.
- The *communication diagram* (formerly called collaboration diagram) focuses on object relationships in which the message passing is central. The sequencing of messages is given through

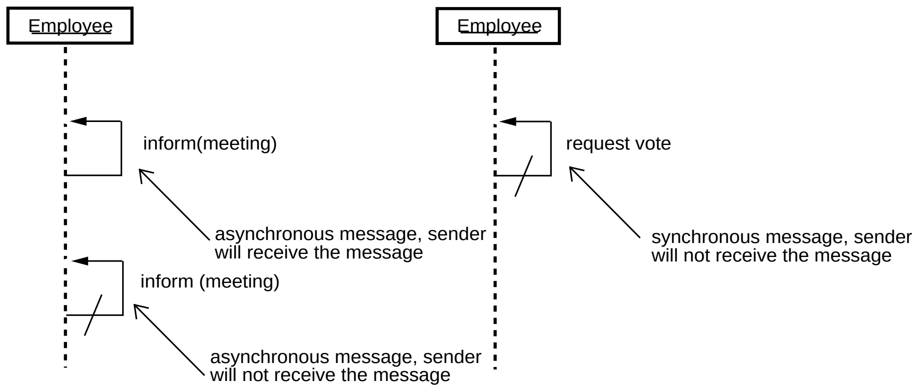
Employee
Employee
inform(meeting)
request vote
asynchronous message, sender will receive the message
synchronous message, sender will not receive the message
inform (meeting)
asynchronous message, sender will not receive the message

Figure 4.26 Messages within an agent.

a sequence numbering scheme. Sequence diagrams and collaboration diagrams express similar information, but show it in different ways.

- The *timing diagram* is an interaction diagram that shows the change in state or condition of a lifeline (representing a classifier instance or classifier role) over linear time. The most common usage is to show the change in state of an object over time in response to accepted events or stimuli.

4.6.2 Social Structures

We have seen in the above methodologies that the modeling of roles, groups, and societies are important aspects that must be taken into consideration during the specification of agent-based systems. We have already noticed how different approaches deal with these aspects in different ways, in particular in the context of using UML.

In this section we take a closer look at an approach suggested by Parunak and Odell [37]. Based on the emphasis on the correspondence between multiagent systems and social systems, they 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:* It is assumed 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.

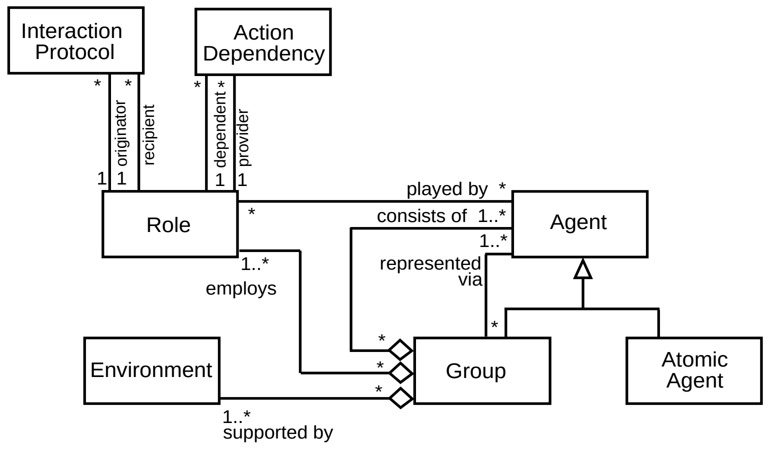

Figure 4.27 Conceptual model of Parunak and Odell's approach.

- *Environments:* Environments are not only passive communications frameworks with everything of interest relegated to them, but actively provide three information processing functions: They *fuse* information from different agents passing over the same location at different times; they *distribute* information from one location to nearby locations; and they provide *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:

1. To achieve more efficient or secure interaction between a set of agents (*intragroup associations*).
2. To take advantage of 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 alone would be capable of (through *group synergies*).
3. To establish a group of agents that interacts with other agents or groups in a coherent way, for example, to represent a shared position on a subject (*intergroup associations*).

The conceptual model of Parunak and Odell's approach is illustrated in Figure 4.27.

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 Figures 4.28 and 4.29, denoting that the Terrorist Organization involves two roles, Operative and Ringleader, where the Ringleader agent coordinates Operative agents. The

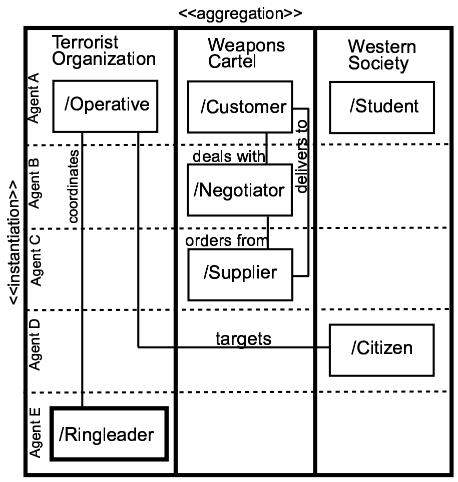

Figure 4.28 Swimlanes as groups.

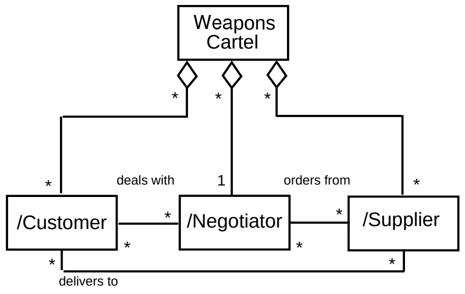

Figure 4.29 Class diagrams define roles.

second swimlane is based on agent instances; for example, agent A plays the roles of Operative, Customer, and Student.

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, and sequence diagrams model the interactions that may occur among these entities. Figure 4.30 depicts the permitted interactions that may occur among Customer, Negotiator, and Supplier agents for a weapons procurement negotiation. Figure 4.31 shows an activity graph modeling groups of agents as individual agents. In this way, the kinds of dependencies are expressed that are best represented at a group level.

4.6.3 Agent Classes

To our knowledge, basing agent classes on UML class diagrams has so far only been considered by [39, 40]. The latter is currently being revisited within FIPA, and an updated version was scheduled to be available by the end of 2003. As described by [40], 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 Figure 4.32 specifies agent classes.

Bauer [40] states that usual UML notation with stereotypes can be used to define such an agent class, but for readability reasons, the notation of Figure 4.32 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 [35] is

$$\underline{instance - 1 ... instance - n / role - 1 ... role - m : class}$$

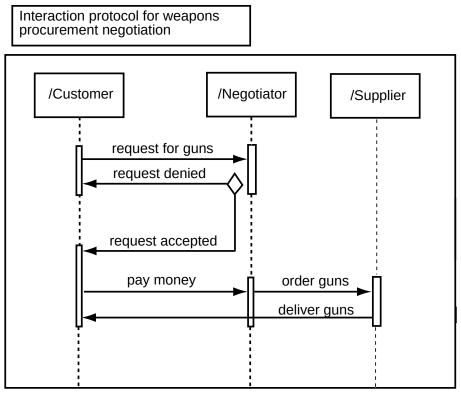

Figure 4.30 Sequence diagram depicting an interaction protocol.

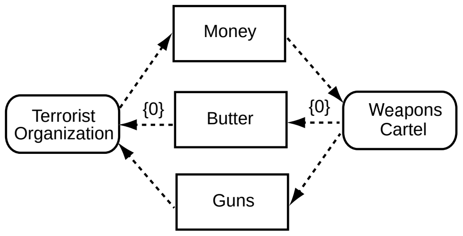

Figure 4.31 An object-flow activity graph specifying roles as patterns of activities.

denoting a distinguished set of agent instances, with $instance - 1, \ldots, instance - n$ satisfying the agent roles, $role - 1, \ldots, role - m$, with $n, m \geq 0$ and the class it belongs to. Instances, roles, or the class can be omitted, and 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, for example, BDI agents. Beyond the extension of the type for the fields, visibility and a persistency attribute can be added (denoted by the stereotype <<persistent>>) to allow the user agent to be stopped and restarted 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*, and 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 (an agent implemented on top of, for example, a Java-based agent platform). However, in different design stages, different kinds of agents can be appropriate; at a conceptual level, BDI agents can be implemented by a Java-based agent platform, so that refinement steps from BDI agents to Java agents are performed during the agent development.

- *Actions*: Proactive behavior is defined in two ways, using proactive actions and proactive agent state charts (the latter of which is considered later). Thus, two kinds of actions can be specified for an agent: proactive actions (denoted by the stereotype <<pro-active>>) are triggered by the agent itself if the precondition of the action evaluates to true. *Reactive* actions (denoted by

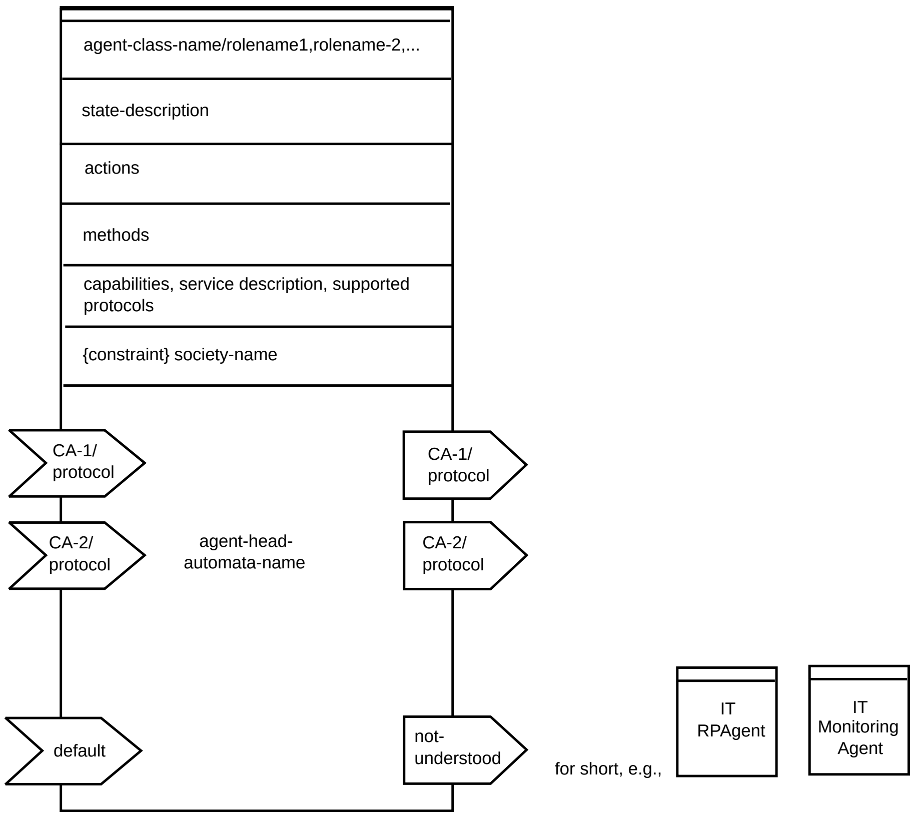

Figure 4.32 Agent class diagram and its abbreviations.

the stereotype <<re-active>>) are triggered by another agent (on receiving a message from that other agent). The description of an agent's actions consists of the action signature with visibility attribute, the action name, and a list of parameters with associated types. Preconditions, postconditions, effects, and invariants, as in UML, define the semantics of an action.

- *Methods*: Methods are defined as in UML, eventually with preconditions, postconditions, effects, and invariants.
- *Capabilities*: The capabilities of an agent can be defined either in an informal way or using class diagrams; for example, defining FIPA-service descriptions.
- *Sending and receiving of communicative acts*: The sending and receiving of communicative acts characterize the main interface of an agent to its environment. By communicative act (CA), the

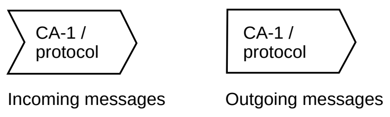

Figure 4.33 Incoming and outgoing messages.

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 and outgoing messages are drawn as shown in Figure 4.33. Then, 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. In the case of an instance of a communicative act, the notation *CA-1 / protocol* is applied. As an 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 their 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, by which *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 [40].

4.6.4 Representing Ontologies by Using UML

As we have already mentioned, for example, with PASSI, several research approaches, such as [41] and [42], are dealing with the definition of ontologies using UML class diagrams, not only from the agent community but also from the Semantic Web community [32, 43].

Bergenti et al. [41] take a pragmatic view of an ontology definition by applying UML class diagrams as shown in Figure 4.34, defining the entities and relating them 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 [42] (as in Figure 4.35). They also apply UML to ontology definition in [44]. In [45], an extension of UML is defined to cover DAML, with Table 4.6 showing the high-level mapping between UML and DAML (described in more detail in Chapter 5). The correspondence relations between UML extensions and DAML are summarized in Table 4.7.

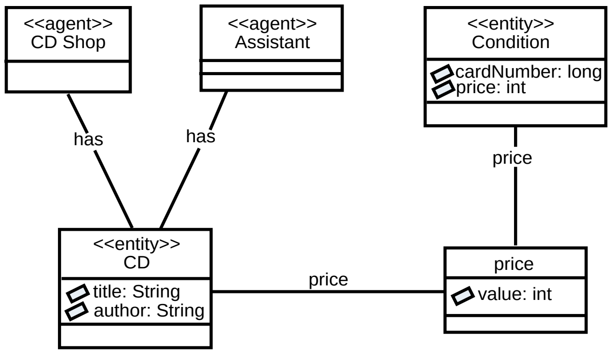

Figure 4.34 UML-based ontology definition.

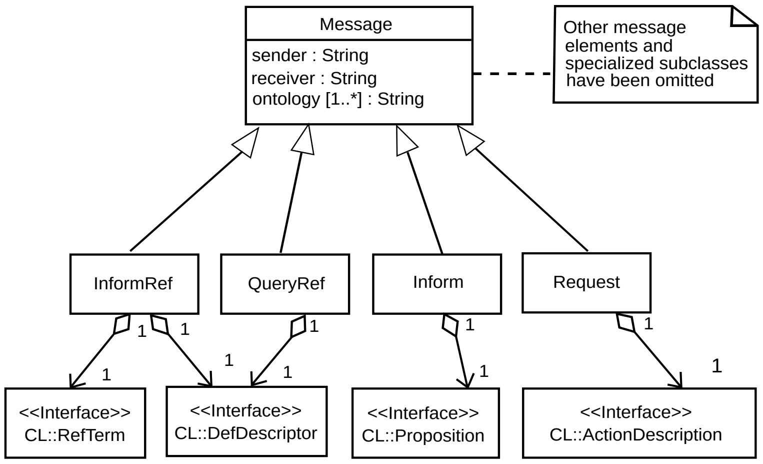

Figure 4.35 Excerpt of object-oriented design of FIPA ACL/SL.

Table 4.6
Comparison Between DAML and UML

DAML Concept	**Similar UML Concepts**
Ontology	Package
Class As sets (disjoint, union)	Class Difficult to represent
Hierarchy	Class generalization aspects
Property	Aspects of attributes, associations, and classes
Hierarchy	None for attributes, limited generalization for associations, class generalization relations
Restriction	Constrains association ends, including multiplicity and roles. Implicitly as class containing the attribute.
Data types Instances and values	Data types Object instances and attribute values

Source: [45].

4.6.5 UML Representation for Goals and Plans

Goals and plans are described by state charts or activity diagrams in several methodologies (see above). Huget [36], uses UML 2 activity diagrams for the description of goals and plans. Here, we present his example of a goal diagram corresponding to the interaction between the customer and the order acquisition (illustrated in Figure 4.36).

The goal diagram expresses the following. The 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 catalog (through the action, *Browse Catalog*). If the ordered item is off catalog, then the next action is on A.[10] This characteristic only changes how the item is produced and priced. If the ordered item is in the catalog, 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 counterproposal that may or may not be accepted by the customer. If the customer accepts the counteroffer, the next action is to write an invoice (action *Write invoice*). If the customer does not accept, it can make another proposal in the same way as 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*).

To conclude, the use of UML as a representation for goals and plans has several advantages, including the possibility of using commercial tools (such as Rational Rose plug-in), easier accessibility

10 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.

Table 4.7
Mapping of DAML and UML Extensions

UML	**DAML**
class instanceOf	Class type
type of ModelElement	type
attribute binary association	ObjectProperty or Datatype Property ObjectProperty
generalization	subClassOf
<<subPropertyOf'>> stereotyped dependency between 2 associations	subPropertyOf
generalization between stereotyped classes	subPropertyOf
note	comment
name	label
"seeAlso" tagged value on a class and association	seeAlso
"isDefinedBy" tagged value on a class and association class containing the attribute	isDefinedBy "subClassOf" a property restriction
source class of an association attribute type	"subClassOf" a property restriction "toClass" on a property restriction
target class of an association	"toClass" on a property restriction
<<equivalentTo>> stereotyped dependency	equivalentTo
<<sameClassAs>> stereotyped dependency between two classes <<samePropertyAs>> stereotyped dependency between two associations <<Ontology>> stereotyped package	sameClassAs samePropertyAs Ontology
"versionInfo" tagged value on a package	versionInfo
import (dependency stereotype)	imports
multiplicity	cardinality
multiplicity range Y..Z	Y = minCardinality, Z = maxCardinality
association target with end multiplicity = 0..1 or 1 association source with end multiplicity = 0..1 or 1	UniqueProperty UnambiguousProperty
<<inverseOf>> stereotyped dependency between two associations	inverseOf
<<TransitiveProperty>> stereotype of an association	TransitiveProperty

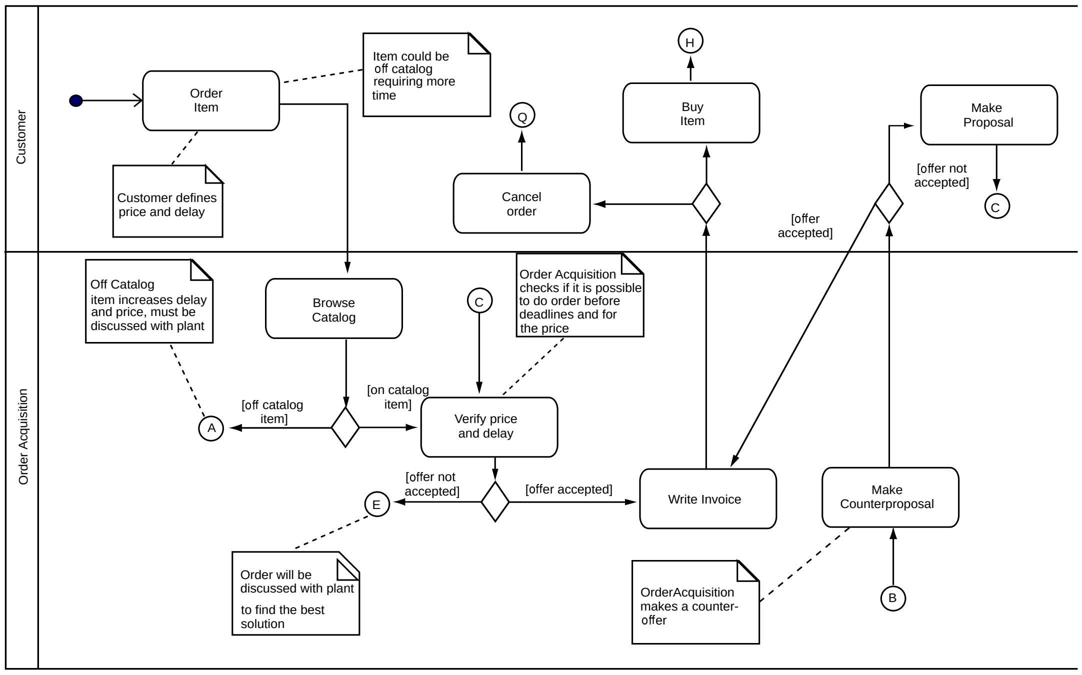
Customer
Order Acquisition
Order Item
Item could be off catalog requiring more time
Customer defines price and delay
Q
Cancel order
H
Buy Item
Make Proposal
[offer not accepted]
C
[offer accepted]
Off Catalog item increases delay and price, must be discussed with plant
Browse Catalog
C
Order Acquisition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

Figure 4.36 Supply chain management scenario as activity diagram. (*After:* [36].)

for software designers with an object-oriented mindset, as well as the possibility of defining mappings to and from other representations.

4.7 MISCELLANEOUS APPROACHES

In the literature, there are many other methodologies based on different software techniques. The purpose of this section is to provide the reader with a brief overview and links to some of these approaches.

ADELFE (Atelier pour le developpement de logiciels à fonctionnaliteé emergente) [46, 47] is a methodology mainly based on the Rational Unified Process and UML extensions for agents. The aim is to allow the specification of systems coping with unpredictable events that occur in the environment, and to select the appropriate action under these unpredictable events. Flake, Geiger, and Küster [48] present a UML-based approach for the specification of agent-based systems, and BDI agents in particular. They extend use case diagrams and class diagrams by the notions of agents and goals, using new actor symbols and stereotypes. High-level and detailed plans are models using restricted UML state charts.

One of the first approaches towards modeling agent-based systems using OO techniques was that described by Burmeister [10]. Here, the approach relies on the use of three models: an *organization model*, an *agent model* based on BDI, and a *cooperation model* (the latter of which roughly corresponds to the interaction model seen in Gaia, as described in Section 4.4.1).

MASB [49, 50], an agent-oriented method, is mainly influenced by research done in the field of cooperative work. It covers both the analysis and design phases. The former is specified by scenario description, role functional description, data and world conceptual modeling, and a system-user interaction model. The design phase covers the MAS architecture and scenario description, object modeling and agent modeling, as well as the conversation modeling and the overall validation of system design.

OPM/MAS (Object-Process Methodology for MAS) [51] is an agent-oriented methodology based on OPM (object-process methodology [52]), combining objects and processes. OPM/MAS deals with static declarative building blocks defined as objects, like organizations and societies, and includes building blocks characterizing behavior, and dynamics defined as processes, like agents, tasks, or messages.

Bush et al. [53] propose Styx, a methodology guiding the development of collaborative agent-based systems. Styx applies use case maps describing potential processes through the system, and domain concepts to describe the world of discourse, the role responsibility model, role relationship model, and deployment model.

Kendall et al. [54] present an agent-based methodology based on object-oriented methodologies and enterprise modeling methodologies, IDEF (integration definition for function modeling), as well as CIMOSA (computer integrated manufacturing open system architecture) for enterprise modeling.

Other types of methodologies include formal approaches, such as Cassiopeia [55], DESIRE [56, 57], the SMART agent framework for representing and reasoning about agents in Z [58, 59] and CAMLE [60] to mention only a few. Knublauch also presented an agent approach based on Extreme Programming in [61]. Further methodologies and approaches are discussed in [1, 62].

In [63], Wagner presents a 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 other things) *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.

4.8 SUMMARY AND CONCLUDING REMARKS

In this chapter, we have surveyed a number of important research contributions in the area of methodologies and notations for the development of agent-based systems. Three main roots can be identified, starting first with approaches based on knowledge engineering principles. While the strength of these approaches lies in their ability to represent the entities and relationships relevant for a domain from a knowledge-level perspective, they lack support for specific agent-related constructs. The recognition of these limitations has led to the emergence of purely agent-oriented approaches that provide rich support for modeling artifacts such as goals, intentions, and organizations. A problem with these approaches was that their underlying conceptual models were proprietary and difficult to understand for industrial software developers without an education in agent technology. At the same time, no industrial-strength tools were available to alleviate some of the complexity for developers. As a consequence, purely agent-oriented software methodologies did not easily migrate to industrial applicability. In response, a number of methodologies were developed that extended state-of-the-art object-oriented approaches, such as UML and RUP, and built agent-oriented features into and on top of these object-oriented models. While this introduces a number of trade-offs, in particular regarding the natural design of agent-based systems, the main advantage of these approaches is that they fit more easily into the object-oriented paradigm and that high-quality tools can be developed by extending existing object-oriented tools. It appears that while objects and agents are clearly different notions (see, for example, the discussion in [64]), agent-oriented software engineering can greatly benefit from OO technologies and approaches. In particular, agent-oriented approaches are also suitable for areas in which object-oriented modeling has shortcomings. Here, the abstractions inherent in agent-oriented software engineering can help to overcome the limitations of the object-oriented approach.

Summarizing the different approaches, we distill the following necessary aspects to be covered by a methodology covering major areas of agent-based systems.

4.8.1 Analysis

The analysis must 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.
- *Environment model*: In [37], Odell et al. consider several aspects of environment modeling ranging from physical environments to agent communication, and how these 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, for example, UML and Semantic Web representation languages.
- *Role model*: This model describes the roles in a domain, on the one hand in the traditional object-oriented sense (actor-role relationship), and on the other 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 that abstracts away 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,[11] and to relate business processes to the organizational model, the interaction model, and the task model.

4.8.2 Design

- *Interaction protocol model*: This model defines the interaction between different agent classes, 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: how different agent collaborate 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 in terms of, for example, BDI, and the agent model defines the behavior of an agent as seen by other agents.
- *Service/capability model*: This 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 other behavior, for example, in a coordination task.
- *Deployment/agent instance model*: This model describes which agent instances exist, migration of agents, and the dynamic creation of agents.

4.8.3 Conclusions

It appears that one single methodological approach does not fit all purposes. This is particularly true given the breadth of domains for agent technology, and the diversity of the underlying agent and multi-agent architectures. How to deal with this situation is an open question. One possible approach that might allow us to cope with the different requirements of different application developers could be the introduction of a meta-methodology that supports the various types of models described above and

11 http://www.agentissoftware.com

provides adequate mappings. An example of such an approach currently being pursued in the object-oriented world is the SPEM[12] initiative currently in discussion at the OMG, allowing the construction of different methodologies on top of an existing meta-methodology.

An important prerequisite to bringing agent technology to market successfully is the availability of expressive and usable development tools, to enable software engineers to construct methodologies, define the various models listed above, and to achieve automatic model transformation as far as possible, for example based on the model-driven architecture (MDA).[13] Finally, it appears that (independent of the methodology used) the question of how agent-based approaches can be embedded and migrated into mainstream IT infrastructures and solutions is another key factor to determine which elements of the current work on agent-oriented software engineering will be successful in real-world software engineering environments. The interested reader should consult [65] for a detailed discussion of some of the issues that are related to interoperability and migration of agent-based software.

Acknowledgments

We would like to thank Michael Luck for the invitation to contribute a chapter on agent methodologies in this book. 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 papers on the AUML Web site, `http://www.auml.org`.

References

[1] Iglesias, C., M. Garrijo, and J. Gonzalez, "A Survey of Agent-Oriented Methodologies," *Proceedings of the 5th International Workshop on Intelligent Agents V: Agent Theories, Architectures, and Languages (ATAL-98),* J. Müller, M. P. Singh, and A. S. Rao, (eds.), Volume 1555 of *LNCS*, New York: Springer, 1999, pp. 317–330.

[2] Garijo, F., and M. Boman, (eds.), *Multi-Agent System Engineering (MAAMAW'99)*, Volume 1647 of *LNCS*, New York: Springer, 1999.

[3] Giunchiglia, F., J. Mylopoulos, and A. Perini, "The TROPOS Software Development Methodology: Processes, Models and Diagrams," *Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems AAMAS'02*, C. Castelfranchi and W. Johnson, (eds.), New York: ACM Press, 2002, pp. 35–36.

[4] Wooldridge, M., N. Jennings, and D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design," *Autonomous Agents and Multi-Agent Systems*, Vol. 3, No. 3, 2000, pp. 285–312.

[5] Herlea, D. E., et al., "Specification of Behavioural Requirements Within Compositional Multi-Agent System Design," *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World: Multi-Agent System Engineering (MAAMAW'99),* F. J. Garijo, and M. Boman, (eds.), Volume 1647 of *LNCS*, New York: Springer, 1999, pp. 8–27.

[6] Brazier, F. M. T., C. M. Jonker, and J. Treur, "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*, J. Cuena, (ed.), 1998, pp. 347–360.

12 http://www.omg.org/technology/documents/formal/spem.htm
13 http://www.cs.rmit.edu.au/agents/SAC/methodology.shtml

[7] Jonker, C. M., and J. Treur, "Compositional Verification of Multi-Agent Systems: A Formal Analysis of Pro-Activeness and Reactiveness," *Proceedings of the International Workshop on Compositionality, COMPOS'97*, W. P. de Roever, H. Langmaack, and A. Pnueli, (eds.), Volume 1536 of *LNCS*, New York: Springer, 1997, pp. 350–380.

[8] Kinny, D., and M. Georgeff, "Modelling and Design of Multi-Agent Systems," *Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96)*, J. Müller, M. Wooldridge, and N. R. Jennings, (eds.), Volume 1193 of *LNCS*, New York: Springer, 1996, pp. 1–20.

[9] Kinny, D., M. Georgeff, and A. Rao, "A Methodology and Modelling Technique for Systems of BDI Agents," *Agents Breaking Away: Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Y. Demazeau and J.-P. Müller, (eds.), Volume 1038 of *LNCS*, New York: Springer, 1996, pp. 56–71.

[10] Burmeister, B., "Models and Methodology for Agent-Oriented Analysis and Design," *Working Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems*, K. Fischer, (ed.), DFKI, 1996.

[11] Jacobson, I., G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Reading, MA: Addison-Wesley, 1998.

[12] Beck, K., *Extreme Programming Explained*, Reading, MA: Addison-Wesley, 1999.

[13] Schreiber, A. T., et al., "CommonKADS: A Comprehensive Methodology for KBS Development," *IEEE Expert*, Vol. 9, No. 3, 1994, pp. 28–37.

[14] Glaser, N., "Contribution to Knowledge Modelling in a Multi-Agent Framework (the CoMoMAS Approach)," Ph.D. thesis, Université Henri Poincaré, Nancy I, 1996.

[15] Iglesias, C., et al., "A Methodological Proposal for Multiagent Systems Development Extending CommonKADS," *Proceedings of the Tenth Knowledge Acquisition Workshop*, Banff, Canada, 1996.

[16] Rudolph, E., P. Graubmann, and J. Grabowski, "Tutorial on Message Sequence Charts," *Computer Networks and ISDN Systems*, Vol. 28, No. 12, 1996, pp. 1629–1641.

[17] Rumbaugh, J., "OMT: The Development Process," *JOOP Journal of Object Oriented Programming*, Vol. 8, No. 2, 1995, pp. 8–16.

[18] Rumbaugh, J., "OMT: The Dynamic Model," *JOOP Journal of Object Oriented Programming*, Vol. 7, No. 9, 1995, pp. 6–12.

[19] Juan, T., A. R. Pearce, and L. Sterling, "ROADMAP: Extending the Gaia Methodology for Complex Open Systems," *Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems AAMAS'02*, C. Castelfranchi and W. Johnson, (eds.), New York: ACM Press, 2002, pp. 3–10.

[20] Omicini, A., "SODA: Societies and Infrastructures in the Analysis and Design of Agent-Based Systems," *Agent-Oriented Software Engineering*, Volume 1957 of *LNCS*, P. Ciancarini and M. Wooldridge, (eds.), New York: Springer, 2001, pp. 185–193.

[21] Bauer, B., J. Muller, and J. Odell, "Agent UML: A Formalism for Specifying Multiagent Software Systems," *International Journal on Software Engineering and Knowledge Engineering (IJSEKE)*, Vol. 11, No. 3, 2001, pp. 207–230.

[22] Kinny, D., M. P. Georgeff, and A. S. Rao, "A Methodology and Modelling Technique for Systems of BDI Agents," *Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, W. Van de Velde and J. W. Perram, (eds.), Volume 1038 of *LNCS*, New York: Springer, 1996, pp. 56–71.

[23] Caire, G., et al., "Agent Oriented Analysis Using MESSAGE/UML," *Agent-Oriented Software Engineering II*, M. Wooldridge, G. Wei, and P. Ciancarini, (eds.), Volume 2222 of *LNCS*, New York: Springer, 2001, pp. 119–135.

[24] Mylopoulos, J., M. Kolp, and J. Castro, " UML for Agent-Oriented Software Development: The Tropos Proposal," *UML 2001 — The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference*, M. Gogolla and C. Kobryn, (eds.), Volume 2185 of *LNCS*, New York: Springer, 2001, pp. 422–441.

[25] Busetta, P., et al., *JACK Intelligent Agents — Components for Intelligent Agents in Java,* Technical Report tr9901, http://www.agent-software.com/.

[26] Castro, J., M. Kolp, and J. Mylopoulos, "Towards Requirements-Driven Information Systems Engineering: The TROPOS Project," *Information Systems*, Vol. 27, No. 6, 2002, pp. 365–389.

[27] Padgham, L., and M. Winikoff, "Prometheus: A Methodology for Developing Intelligent Agents," *Agent-Oriented Software Engineering III*, F. Giunchglia, J. Odell, and G. Weiß, (eds.), Volume 2585 of *LNCS*, New York: Springer, 2003, pp. 174–185.

[28] Cervenka, R., "Modeling Notation Source: Prometheus," http://www.auml.org/auml/documents/Prometheus030402.pdf, September 2003.

[29] Wood, M. F., and S. A. A. DeLoach, "An Overview of the Multiagent Systems Engineering Methodology," *Proceedings of the First International Workshop on Agent-Oriented Software Engineering*, P. Ciancarini and M. Wooldridge, (eds.), Volume 1957 of *LNCS*, New York: Springer, 2000, pp. 127–141.

[30] Wood, M. F., "Multiagent Systems Engineering: A Methodology for Analysis and Design of Multiagent Systems," M.Sc. thesis, AFIT/GCS/ENG/00M-26, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, 2000.

[31] DeLoach, S. A., and M. F. Wood, *Multiagent Systems Engineering: The Analysis Phase,* Technical Report, Air Force Institute of Technology, AFIT/EN-TR-00-02, 2000.

[32] Cossentino, M., and C. Potts, "A Case Tool Supported Methodology for the Design of Multi-Agent Systems," *The 2002 International Conference on Software Engineering Research and Practice (SERP'02)*, 2002.

[33] Lind, J., *Iterative Software Engineering for Multiagent Systems: The MASSIVE Method*, Volume 1994 of *LNCS*, New York: Springer, 2001.

[34] Kḧnel, R., *Agentenbasierte Software — Methode und Anwendungen*, Reading, MA: Addison-Wesley, 2000.

[35] Bauer, B., J. P. Muller, and J. Odell, "An Extension of UML by Protocols for Multiagent Interaction," *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS-00),* E. H. Durfee, (ed.), IEEE Computer Society, 2000.

[36] Huget, M.-P., "FIPA-Modelling: Interaction Diagrams," http://www.auml.org/auml/documents/ID-03-07-02.pdf, September 2003.

[37] Odell, J. J., et al., "Modeling Agents and their Environment," *Agent-Oriented Software Engineering III*, F. Giunchiglia, J. Odell, and G. Weiß, (eds.), Volume 2585 of *LNCS*, New York: Springer, 2002, pp. 16–31.

[38] Parunak, H. V. D., and J. Odell, "Representing Social Structures in UML," *Agent-Oriented Software Engineering II*, M. Wooldridge, G. Weiß, and P. Ciancarini, (eds.), Volume 2222 of *LNCS*, New York: Springer, 2001, pp. 1–16.

[39] Wagner, G., "The Agent-Object-Relationship Metamodel: Towards a Unified View of State and Behavior," *Information Systems*, Vol. 28, No. 5, 2003, pp. 475–504.

[40] Bauer, B., et al., "Agents and the UML: A Unified Notation for Agents and Multi-Agent Systems?" *Agent-Oriented Software Engineering II*, M. Wooldridge, G. Wei, and P. Ciancarini, (eds.), Volume 2222 of *LNCS*, New York: Springer, 2001, pp. 148–150.

[41] Bergenti, F., and A. Poggi, "A Development Toolkit to Realize Autonomous and Inter-Operable Agents," *Proceedings of the Fifth International Conference on Autonomous Agents*, New York: ACM Press, 2001, pp. 9–16.

[42] Cranefield, S., S. Haustein, and M. Purvis, "UML-Based Ontology Modelling for Software Agents," *Proceedings of the Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence*, 1999.

[43] Chang, W. W., "A Discussion of the Relationship Between RDF-Schema and UML," http://www.w3.org/TR/1998/NOTE-rdf-uml-19980804/, September 2003.

[44] Cranefield, S., and M. Purvis, "Generating Ontology-Specific Content Languages," *Proceedings of the Workshop on Ontologies in Agent Systems, 5th International Conference on Autonomous Agents*, 2001, pp. 29–35.

[45] Baclawski, K., et al., "Extending UML to Support Ontology Engineering for the Semantic Web," *UML 2001 — The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference*, M. Gogolla and C. Kobryn, (eds.), Volume 2185 of *LNCS*, New York: Springer, 2001, pp. 342–360.

[46] Bernon, C., et al., "ADELFE: A Methodology for Adaptive Multi-Agent Systems Engineering," *Engineering Societies in the Agents World III, Third International Workshop, ESAW 2002*, P. Petta, R. Tolksdorf, and F. Zambonelli, (eds.), Volume 2577 of *LNCS*, New York: Springer, 2002, pp. 156–169.

[47] Bernon, C., et al., "The Adelfe Methodology for an Intranet System Design," *AOIS '02, Agent-Oriented Information Systems, Proceedings of the Fourth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2002 at CAiSE*02)*, P. Giorgini et al., (eds.), Volume 57 of *CEUR Workshop Proceedings*, 2002.

[48] Flake, S., C. Geiger, and J. M. Küster, "Towards UML-Based Analysis and Design of Multi-Agent Systems," *International NAISO Symposium on Information Science Innovations in Engineering of Natural and Artificial Intelligent Systems*, 2001.

[49] Moulin, B., and L. Cloutier, "Collaborative Work Based on Multiagent Architectures: A Methodological Perspective," *Soft Computing: Fuzzy Logic, Neural Networks and Distributed Artificial Intelligence*, F. Aminzadeh and M. Jamshidi, (eds.), Upper Saddle River, NJ: Prentice Hall, 1994, pp. 261–296.

[50] Moulin, B., and M. Brassard, "A Scenario-Based Design Method and an Environment for the Development of Multiagent Systems," *Distributed Artificial Intelligence Architecture and Modelling: Proceedings of the First Australian Workshop on Distributed Artificial Intelligence*, C. Zhang and D. Lukose, (eds.), Volume 1087 of *LNCS*, New York: Springer, 1996, pp. 216–231.

[51] Sturm, A., D. Dori, and O. Shehory, "Single-Model Method for Specifying Multi-Agent Systems," *The Second International Joint Conference on Autonomous Agents & Multiagent Systems*, New York: ACM Press, 2003, pp. 121–128.

[52] Dori, D., *Object-Process Methodology: A Holistic System Paradigm*, New York: Springer, 2002.

[53] Bush, G., S. Cranefield, and M. Purvis, "The STYX Agent Methodology," *Information Science Discussion Papers Series*, University of Otago, 2001.

[54] Kendall, E. A., M. T. Malkoun, and C. Jiang, "A Methodology for Developing Agent Based Systems," *Distributed Artificial Intelligence Architecture and Modelling: Proceedings of the First Australian Workshop on Distributed Artificial Intelligence*, C. Zhang and D. Lukose, (eds.), Volume 1087 of *LNCS*, New York: Springer, 1996, pp. 85–99.

[55] Coolinot, A., A. Drougol, and P. Benhamou, "Agent Oriented Design of a Soccer Robot Team," *Proceedings on the Second International Conference on Multi-Agent Systems (ICMAS-96)*, V. Lesser, (ed.), Menlo Park, CA: AAAI Press, 1996, pp. 41–47.

[56] Brazier, F., et al., "Formal Specification of Multi-Agent Systems: A Real-World Case," *Proceedings of the First International Conference on Multi-Agent Systems*, Menlo Park, CA: AAAI/MIT Press, 1995, pp. 25–32.

[57] Dunin-Keplicz, B., and J. Treur, "Compositional Formal Specification of Multi-Agent Systems," *Intelligent Agents: Theories, Architectures, and Languages*, M. Wooldridge and N. R. Jennings, (eds.), Volume 890 of *LNCS*, New York: Springer, 1995, pp. 102–117.

[58] Luck, M., and M. d'Inverno, "A Formal Framework for Agency and Autonomy," *Proceedings of the First International Conference on Multi-Agent Systems*, Menlo Park, CA: AAAI/MIT Press, 1995, pp. 254–260.

[59] Luck, M., N. Griffiths, and M. d'Inverno, "From Agent Theory to Agent Construction: A Case Study," *Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, J. P. Müller, M. Wooldridge, and N. Jennings, (eds.), Volume 1193 of *LNCS*, New York: Springer, 1997, pp. 49–63.

[60] Zhu, H., "A Formal Specification of Evolutionary Software Agents," *ICFEM'2002: Proceedings of the IEEE International Conference on Formal Engineering Methods,* C. George and H. Miao, (eds.), New York: Springer, 2002, pp. 249–261.

[61] Knublauch, H., "Constraining Autonomy Through Norms," *Proceedings of the First International Joint Conference on Autonomous Agents and Multi Agent Systems AAMAS'02,* C. Castelfranchi and W. Johnson, (eds.), New York: ACM Press, 2002, pp. 704–711.

[62] Wooldridge, M., and P. Ciancarini, "Agent-Oriented Software Engineering: The State of the Art," *Agent-Oriented Software Engineering, First International Workshop, AOSE 2000*, M. Wooldridge and P. Ciancarini, (eds.), Volume 1957 of *LNCS*, New York: Springer, 2001, pp. 1–28.

[63] Wagner, G., "A UML Profile for External Agent-Object-Relationship (AOR) Models," *Agent-Oriented Software Engineering III*, F. Giunchiglia, J. Odell, and G. Weiß, (eds.), Volume 2585 of *LNCS*, New York: Springer, 2002, pp. 138–149.

[64] Wooldridge, M. J., *An Introduction to Multiagent Systems*, New York: Wiley, 2002.

[65] Müller, J. P., and B. Bauer, "Agent-Oriented Software Technologies: Flaws and Remedies," *Agent-Oriented Software Engineering III*, F. Giunchiglia, J. Odell, and G. Weiß, (eds.), Volume 2585 of *LNCS*, New York: Springer, 2003, pp. 210–227.