Intelligent Agents

Gerhard Weiss, *SCCH GmbH, Austria*
Lars Braubach, *University of Hamburg, Germany*
Paolo Giorgini, *University of Trento, Italy*

FOUNDATIONS OF INTELLIGENT AGENTS

The agent concept originated in artificial intelligence and especially in the field of agent and multiagent technology (Weiss 1999; Wooldridge 2002). Thus the roots of the concept extend back to the 1950s. In the past decade the agent concept has been successfully established in various fields of computer science, especially in the fields of software and software engineering. The term *agent* was long the subject of intensive discussion and efforts toward precise specification, which continue to some extent today. Recent years have seen broad acceptance of the following notion of agency:

An intelligent *agent is a self-contained software/hardware unit that can handle its tasks in a knowledge-based, flexible, interactive, and autonomous way.*

The following ideas underlie these key attributes of an agent as formulated above:

- *Flexibility*. An agent can act reactively as well as proactively. *Reactive* means that the agent reacts in reasonable time and in an appropriate way to changes in its environment and to changes in the requirements placed on it. *Proactive* means that the agent acts with prediction, planning, and goal orientation. Flexibility, consisting of reactivity and proactivity, is thus the capability to handle possibly unexpected events and simultaneously to act with planning and goal orientation.
- *Interactivity*. An agent can interact with its environment—especially with human actors and with other agents. Such interactions can be on a very high level (i.e., they can be markedly communication- and knowledge-intensive) and they serve the purpose of coordination with third parties, that is, the coordination of activities and the handling of mutual dependencies. Here we mean coordination in the sense of cooperation (joint pursuit of possibly shared plans and goals) as well as in the sense of competition (pursuit of partially or even wholly exclusive goals). Examples of forms of interaction that are considered typical of agents include negotiation and conflict resolution in the realm of cooperative planning activities and competitive sales processes. Interactivity requires a precise interface that normally overshadows all the internals of the agent. Thus in general *interactivity* designates all the (higher) social—communicative, cooperative, and competitive—capabilities of an agent.
- *Autonomy*. In the realm of its task processing, an agent can decide largely autonomously and without consultation or coordination with third parties (human users or other agents) which activities to execute. This frequently requires or implicitly assumes that the decisions to be made by the agents are nontrivial, that is, that they may require extensive knowledge processing or that the effects are significant. An agent has a certain scope of decision-making authorization and freedom of action and so is subject to control by third parties only to a restricted degree.

Often the above notion of an agent is extended and concretized by associating further attributes with agents. The most prominent of these attributes include:

- *Situatedness/embeddedness*. An agent is connected to its environment via close sensory and/or actuatory coupling.
- *Learning capability/adaptivity*. An agent independently optimizes its functionality with respect to the tasks that are assigned to it, which may change over time.

Other attributes that the literature often designates as elementary for agents and that are noteworthy from a software engineering viewpoint include mobility, *persistency*, *rationality*, and *self-containment*.

The remainder of this chapter is structured as follows. The section on The Intelligent Agents Perspective on Engineering describes the agent-oriented perspective on software and systems engineering. The section on Architecture for Intelligent Agents describes available approaches to agent architectures. The section on Development Methodologies overviews the state of the art in systematically developing systems from the perspective of agent orientation. The section on Tools, Platforms, and Programming Languages (Frameworks) presents important frameworks for building agent applications, including tools, platforms, and programming languages. The section on Application Areas characterizes applications and application areas that are particularly suited for the agent-oriented approach.

THE INTELLIGENT AGENTS PERSPECTIVE ON ENGINEERING

One of the great steps forward in software and systems engineering was the evolution of fundamental system views—paradigms—that support successful, systematic, and efficient development of software systems. Examples of such paradigms include structure orientation, object orientation, component orientation, aspect orientation, model orientation, architecture orientation, pattern orientation, task orientation, and (usually in the context of business information systems) process orientation. Agent orientation, based on the notion of intelligent agents, is a new member of this list of paradigms (Jennings 2000). In the following, we describe fundamental qualitative attributes of agent orientation that confirm a very high potential for utilization and acceptance in software and systems engineering.

System view and abstraction level. Agent orientation suggests the metaphor of a software system as a human organization and thereby opens an innovative, high-quality, and at the same time intuitively comprehensible view of software. This paradigm is innovative and high-quality because it enables viewing software design as organizational design; for a software developer, this opens a gold mine of organization theory concepts and techniques that can be applied in software engineering. The paradigm is intuitively comprehensible because organizational terminology is part of our everyday life; therefore we have no problem in viewing a software system as an organization (or as a combination of multiple organizations) in which software units (agents) handle tasks under consideration of prescribed computation and behavioral guidelines (rules, standards, laws, etc.) and for this purpose negotiate autonomously, resolve (resource) conflicts, dynamically form and dissolve superordinate organizational units (e.g., teams), play certain roles within these superordinate units (e.g., resource manager, service provider), and assume certain obligations with their roles.

Especially characteristic of the agent-oriented system paradigm is that it affords a new abstraction level that is distinct from other paradigms. The step to this abstraction level conforms to a development that is reflected in higher programming languages and that is a necessary prerequisite for programming in the large: the rise in the degree of abstraction, away from the machine level and to the problem level.

Complexity management. Software is inherently complex and its complexity will continue to rise dramatically as it has done in the past. A decisive criterion for the evaluation of a software development approach is thus its suitability for managing complexity. Four elementary techniques for managing complexity are very important in software engineering:

1. **Decomposition.** The reduction into smaller and thus comprehensible parts that can then be developed largely independently.
2. **Abstraction.** The creation of a model that encompasses significant aspects while hiding unimportant aspects.
3. **Structuring.** The specification of (ordered) relationships and (desired) mutual effects among the components of the overall system.
4. **Reuse.** The systematic use of past results (documents and processes) from software projects in future projects.

The intelligent agents perspective supports all four of these techniques in a very natural way.

Autonomy as a system attribute. From the software and systems engineering view, autonomy is the most striking and, in terms of effect, the most far-reaching attribute of intelligent agents. This attribute, even if it seems radical and revolutionary at first glance, can be seen as the next natural step in the evolution of generic engineering principles. This is best seen in the software field. Elementary software units that have evolved up to now—monolithic programs, modules, procedures, objects, components, and services—demonstrate a rising degree of locality and increasing encapsulation of data and state control. All these software units have in common that their activation can be forced via external events (e.g., the start command by the user or the receipt of a message from another software unit); the unit itself does not decide whether to activate upon such a message. Agent orientation overcomes this restriction by providing the autonomy attribute to additionally encapsulate the control over activation of a software unit (self-activation over outside activation, self-determination instead of foreign determination, and self-responsibility rather than foreign responsibility).

The step to software autonomy not only is historically motivated but also reflects practical requirements. On the one hand, a number of applications indirectly imply the necessity to equip software with autonomy. On the other hand, autonomy is increasingly being required directly as a system attribute almost by definition. For example, it has become common to view a peer-to-peer system as a self-organizing system of equal autonomous units, and in the context of Web services, autonomy is usually seen as an important attribute (in addition to the attributes specified in the W3C definition of Web services).
Compatibility. A decisive factor for the potential of a new paradigm—or a view, a technique, a method, etc.—is

its compatibility with existing and established approaches. Agent orientation is to a high degree compatible with other approaches. In particular, the agent-oriented view does not claim to displace or exclude other views.

- The abstraction levels of agent-orientation and object-orientation complement one another in a meaningful way.
- Agents and components share the attribute of *self-containment* and their focus on their interfaces, and the agent concept can be seen as a specialization or generalization (depending on viewpoint) of the component concept.
- Due to its focus on organizational structures (at the level of individual agents), agent orientation has a close relationship to architecture orientation.
- With its focus on interactivity and thus on consequences of coordinated actions, agent orientation has a fundamental commonality with process orientation.
- Similar to task orientation, agent orientation emphasizes the importance of defining overall tasks (at the actor level instead of the object level) and their dependencies.

Thus agent orientation merges various core aspects of other paradigms and also can be used in combination with other approaches.

ARCHITECTURES FOR INTELLIGENT AGENTS

In the context of multiagent systems two different kinds of architectures can be distinguished. *Internal agent architectures* determine the kinds of components an agent consists of and additionally define how these components interact. An internal agent architecture therefore has the main purpose to implement a reasoning process that ultimately leads to agent actions. Many different agent architectures have been developed until today. Among them are simple architectures, such as those inspired by lower animals like ants as well as also very sophisticated architectures, which inter alia build on explanations of the human behavior determination process.

On the other hand, *social agent architectures* have been devised for describing group structures and behavior. In many cases social agent architectures provide concepts on an organizational level, which allow the description of structures and behavior similar to how work is organized within human organizations.

Figure 22.1 gives an overview of existing internal and social agent architectures. Besides the architectures themselves, it is also sketched what their origin is. In general, most agent architectures build on agent theories, which describe the basic building blocks of agents including the behavior determination mechanism in a more abstract way than architectures. In many cases, agent

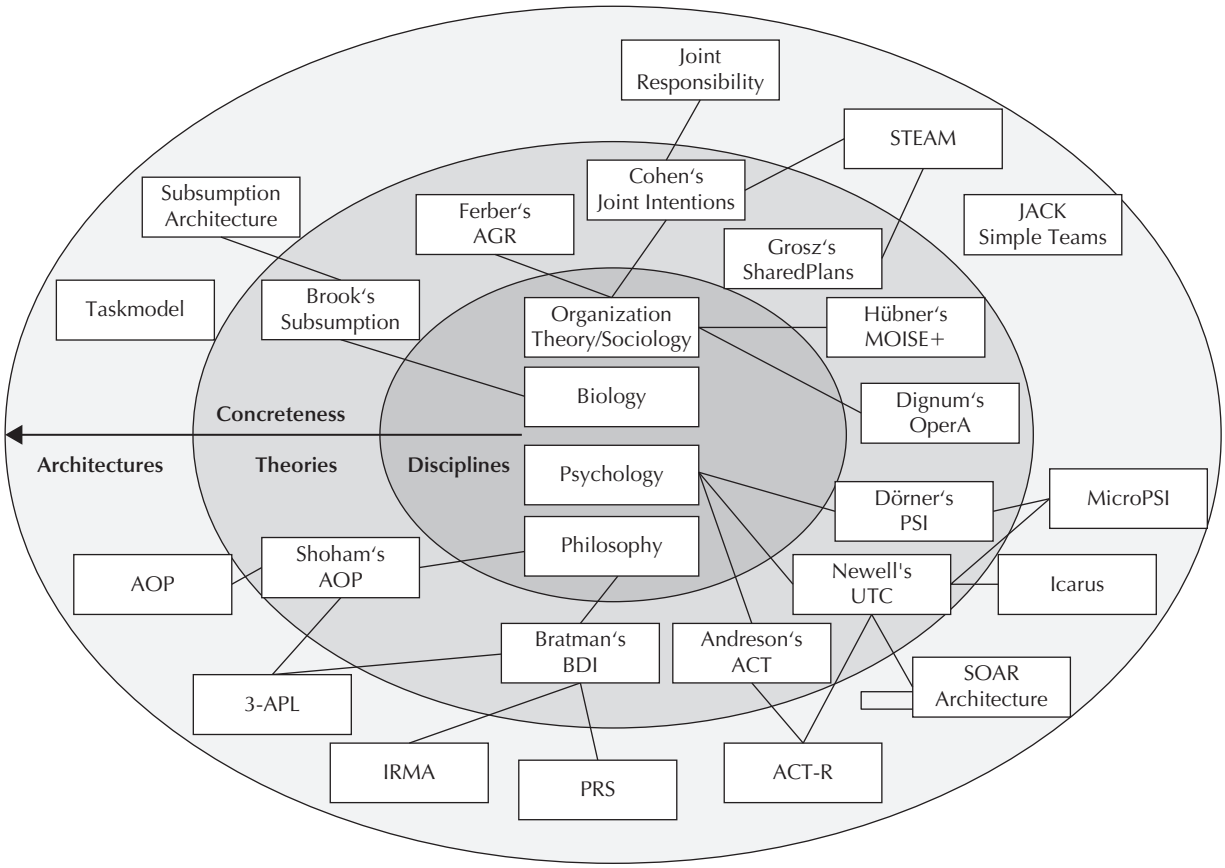

Figure 22.1: Overview of Agent Architectures (from Braubach et al. 2008)

theories have been deeply inspired by existing research of non–computer science-related disciplines such as organization theory, biology, psychology, and philosophy. Hence architectures can be seen as technical interpretations of theories, which concretize and operationalize the underlying ideas and conceptual framework in a way that makes them implementable in software.

Internal Agent Architectures

Due to the variety of different internal agent architectures that have been developed, several classification schemes have been proposed (Braubach et al. 2008). Most influential, the scheme of Wooldridge and Jennings (1994) assumes a distinction between *reactive*, *deliberative*, and *hybrid* agent architectures. A reactive agent architecture underpins the importance of fast reactions to changes within highly dynamic environments. In its purist form, reactive agents do not possess a symbolic representation of the world, and they build their decisions on the received percepts from the environment and other agents only. This also means that an agent has no memory, where it can save experiences from the past and thus cannot learn from failures made earlier. Nonetheless, in specific application domains fast reactions outweigh correct behavior, which is generated too slowly and may no longer be applicable in the current situation. The subsumption architecture (Brooks 1989) is a typical example of a reactive control mechanism, which has been utilized very successfully in the robot domain.

In contrast to reactive agent architectures, deliberative architectures take up a different position and emphasize a symbol-based reasoning process, which requires an agent to posses a local worldview. In line with the physical symbol system hypothesis (Newell and Simon 1976) symbol manipulation is necessary for producing general intelligent action. In consequence, it is often assumed that deliberative agents store their beliefs as logical formulae and have some inference mechanism at their disposal, which implies new knowledge and actions from the existing knowledge. A well-known deliberative agent architecture is IRMA (Intelligent Resource-bounded Machine Architecture) (Bratman, Israel, and Pollack 1988), which exploits traditional planning techniques for goal achievement. IRMA has been successfully used to explore agent reasoning in a relatively simple artificial environment called tile world, in which agents have to transport tiles to holes.

Because both architecture styles exhibit weaknesses when implemented in their strict form, many hybrid architectures try to unify aspects from both approaches and therefore combine timely reactions with well-planned behavior. Hybrid architectures have gained high attention in practice, and nearly all internal architectures, which are supported by agent frameworks, build on the balanced reactive as well as deliberative actions. Due to the high significance of hybrid architectures, in the following paragraphs two typical representatives are presented in more detail.

Task model. The task model is an agent architecture, which has been extracted and consolidated from practical experiences building agent platforms (cf., e.g., JADE, ZEUS, LS/TS). It is based on the observation that agent behavior should be hierarchically decomposable into smaller pieces of work similar to different components in object-oriented settings. Hence, an agent comprises an interpreter, which executes tasks that have been specified in task templates at design time. In general, the architecture permits a complex task to be composed of an arbitrary number of subtasks, which themselves can be complex or simple. Concurrent agent behavior can be established by using more than one active top-level task at the same time. On the other hand, a sequential execution of behaviors can be achieved by scheduling the following task at the end of the current behavior. If coordination between different tasks is necessary, this is normally done by using specific data stores, which can be accessed from multiple tasks and can be employed for exchanging processing results.

Procedural Reasoning System (PRS). The PRS architecture is loosely based on the BDI (belief-desire-intention) model, which has been proposed by Bratman (1987) as a theory for explaining rational human behavior using a framework of folk-psychological mentalistic notions and their interplay. The model assumes that human *practical reasoning* is a two-staged process, which consists of a *goal deliberation* and a *means-end reasoning* phase. While the objective of goal deliberation is to find a consistent goal set without conflicting goals, means-end reasoning is concerned with fulfilling a concrete goal via plans, which describe predefined procedural knowledge of an agent. In PRS, an agent is therefore specified using the mentalistic concepts—*beliefs, goal events,* and *plans*—whereby beliefs are used to store the agent's knowledge about the word, goal events indicate the currently active desires and plans represent the procedural means for achieving goals. The PRS agent interpreter operates on these notions and realizes the means-end reasoning while assuming that an agent only possesses consistent goal sets, that is, goal deliberation is not considered at the architecture level. The interpreter has a relatively simple deliberation cycle, which works on an event queue. In this queue all events that need to be processed, including incoming messages as well as new goal events, are contained. In each deliberation cycle, the agent interpreter selects the next event from the queue and searches for plans that can handle the current event. These plans are subsequently checked for their applicability and one of the applicable plans is then selected for execution. Given that a plan failure occurs, alternative plans can be executed until either the underlying goal has been achieved or no further plans are available.

Social Agent Architectures

In the context of social agent architectures, different approaches have been proposed that either focus on the structure or on the behavior dimension of organizations. Structure-based approaches exploit organizational concepts, which allow multiagent systems to be hierarchically broken down in group-based units, which can themselves be assembled by subgroups or individual agents. This facilitates the construction of highly complex applications by using natural abstractions and applying the well-known divide and conquer principle. On the other hand, approaches, which emphasize the behavioral dimension

primarily, aim at supporting teamwork in cooperative scenarios. For to be able to support the teamwork of agents the approaches have to provide solutions for different kinds of activities including at least team formation, operation, and termination. To be usable in practice, team mechanisms should also consider the degradation of a team (e.g., when a member leaves the group) and provide adequate compensation strategies. In the following, a structure-centered as well as a behavior-centered approach will be presented.

Agent-Group-Role Model. An influential and simple structuring mechanism for agent teams is the agent-group-role (AGR) model, which relies on an organizational viewpoint for multiagent systems (Ferber, Gutknecht, and Michel 2003). This schema assumes that an agent is an active, communicating entity playing roles within certain groups. In this respect, a group is seen as a set of agents sharing some common property, and groups are used as basic structuring means for an application. Groups are defined in terms of their associated roles, which represent placeholders for the different kinds of members forming a group. Therefore, a role is an abstract representation of a functional position or just an identification of a member within a group. At runtime an agent must play at least one role within some group, but is allowed to play arbitrarily many roles in possibly different groups. Groups can freely overlap, which allows an agent to be part of different groups at the same time.

Joint Intentions. A well-known cognitive framework for describing the behavioral aspects of teamwork is the joint intentions theory (Cohen and Levesque 1991). It has been devised in order to set up the formal principles for describing how agents can pursue a common goal. Therefore, the joint intentions theory assumes a mentalistic view of agents and extends the individual notions of belief, goals, and intentions to their group-related counterparts. The key concept of a joint intention is considered as a joint commitment of an agent team to perform a collective action while being in a shared mental state. This joint commitment is expressed with a joint persistent goal held by every agent of the team. In contrast to an individual goal, a joint persistent goal involves further responsibilities for the involved agents. This basically means that each agent not only pursues the goal individually, but also that it will inform the others about important goal changes, that is, it will inform the others if it believes the goal being achieved or unattainable. Based on this general commitment the group can act in a coherent manner and single teammates will not unilaterally drop the joint intention.

DEVELOPMENT METHODOLOGIES

Developing an agent-based software requires, like any other type of software, a systematic engineering approach that supports and drives a development team along all the phases of the software production process. Previous years have shown a burgeoning of a number of innovative agent-oriented (AO) methodologies. Many of these methodologies use the metaphor of an human organization (possibly divided into suborganizations) in which agents play one or several roles and interact with each other. Human organization models and structures are employed for the design of MAS. Concepts like role, social dependency, and organizational rules are used not just to model the environment in which the system will work, but the system itself. Given the organizational nature of a MAS, one of the most important activities in an AO methodology results in the definition of the interaction and cooperation models that capture the social relationships and dependencies between agents and the roles they play within the system. Interaction and cooperation models are generally very abstract, and they are concretized implementing interaction protocols in later phases of the design.

Overall Characterization

The range of AO methodologies is quite complex, and it is therefore very difficult to give a complete characterization of all their facets and dimensions. A tentative analysis proposed in Henderson-Sellers and Giogini (2005) showed a genealogy where lineages and influences among a number of methodologies have been characterized starting from their roots. Particularly, some of them are clearly based on ideas from artificial intelligence (AI), others as direct extensions of existing OO methodologies, while yet others try and merge the two approaches by taking a more purist approach yet allowing OO ideas when these seem to be sufficient.

Several methodologies acknowledge a direct descent from full OO methods. In particular, MaSE (or the more recent O-MaSE) (Garcia-Ojeda et al. 2007) acknowledges influences from (Kendall, Malkoun, and Jiang 1996) as well as an inheritance from AAII (Kinny et al. 1996), which in turn was strongly influenced by the OO methodology of Rumbaugh and colleagues called OMT (Rumbaugh et al. 1991). Similarly, the OO methodology of Fusion (Coleman et al. 1994) was said to be highly influential in the design of Gaia (Zambonelli, Jennings, and Wooldridge 2003). Two other OO approaches have also been used as the basis for AO extensions. RUP (Kruchten 1999) has formed the basis for Adelfe (Bernon et al. 2002) and also for MESSAGE (Caire et al. 2001), which, in turn, is the basis for INGENIAS (Pavon, Gomez-Sanz, and Fuentes 2005; Gómez-Sanz et al. 2008). More recently, RUP has also been used as one of the inputs, together with AOR (Wagner 2003), for RAP (Taveter and Wagner 2005). Second, the OPEN approach to OO software development has been extended significantly to support agents, sometimes called Agent OPEN (Debenham and Henderson-Sellers 2003). Finally, two other methodologies exhibit influences from object-oriented methodological approaches. Prometheus (Padgham and Winikoff 2004; Padgham, Thangarajah, and Winikoff 2008), although not an OO descendant, does suggest using OO diagrams and concepts whenever they exist and are compatible with the agent-oriented paradigm. Similarly, PASSI (Cossentino 2005) merges OO and MAS ideas, using UML as its main notation. Somewhat different is the MAS-CommonKADS methodology (Iglesias et al. 1998). This is a solidly AI-based methodology that claims to have been strongly influenced by OO methodologies,

notably OMT. Then there are the methodologies that do not acknowledge any direct genealogical link to other approaches, OO or AO, such as Tropos (Bresciani et al. 2004), Nemo (Huget 2002), MASSIVE (Lind 1999), and Cassiopeia (Collinot and Drogoul 1998).

Additional useful literature on agent-oriented software and systems development is Bergenti, Gleizes, and Zambonelli (2004), Henderson-Sellers and Giorgini (2005), Luck, Ashri, and D'Inverno (2004), and Weiss (2002).

Selected AO Methodologies

We will briefly describe three of the most popular AO methodologies: GAIA, Prometheus, and Tropos.

GAIA (Zambonelli et al. 2003) is one of the first proposed agent-oriented software engineering methodologies. In GAIA, it is assumed that for the development of medium and large multiagent systems (MAS), possibly situated in open and dynamic environments that have to guarantee predictable and reliable behaviors, the most appropriate metaphor is that of an organization. Organizations are viewed in GAIA as collections of roles, which are defined in terms of responsibilities, permissions, activities, and protocols. Responsibilities define the functionality of the role, while permissions are the rights that allow the role to perform its responsibilities. Activities are computations that can be executed by the role, and protocols define the interaction between roles. As soon as the complexity of systems increases, modularity and encapsulation principles suggest dividing the system into different suborganizations, with a subset of the agents being possibly involved in multiple organizations.

In each organization, an agent can play one or more roles, which define what it is expected to do in the organization, both in concert with other agents and in respect to the organization itself. The notion of a role in GAIA gives an agent a well-defined position in the organization, with an associated set of expected behaviors. To accomplish their roles, agents typically need to interact with each other to exchange knowledge and coordinate their activities. These interactions occur according to patterns and protocols dictated by the nature of the role itself. In addition, an MAS is typically immersed in an environment with which the agents may need to interact in order to accomplish their roles. That portion of the environment that agents can sense and effect is determined by the agent's specific role, as well as by its current status. Identifying and modeling the environment involve determining all the entities and resources that the MAS can exploit, control, or consume when it is working towards the achievement of the organizational goal.

However, although role and interaction models can be useful to fully describe an existing organization, they are of limited value in building an organization. This motivates the introduction of the notions of organizational rules and organizational structures. Indeed, before being able to fully characterize the organization, the analysis of an MAS should identify the constraints that the actual organization, once defined, will have to respect, that is organizational rules. It is possible to distinguish between safety and liveness organizational rules. The former refer to the invariants that must be respected by the organization for it to work coherently; the latter express the dynamics of the organization. A role model implicitly defines the topology of the interaction patterns and the control regime of the organization's activities. That is, it implicitly defines the overall architecture of the MAS organization, that is, its organizational structure. It is more natural for the choice of the organizational structure to follow from the identification of the organizational rules.

The GAIA design process starts with the analysis phase, whose aim is to collect and organize the specification, which is the basis for the design of the computational organization. This means defining an environmental model, preliminary roles and interaction models, and a set of organizational rules. Then, the process continues with the architectural phase, aimed at defining the system's organizational structure in terms of its topology and control regime, which, in turn, helps to identify complete roles and interaction models. During the detailed design phase a detailed, but technology-neutral, specification of an MAS is produced.

Prometheus (Padgham and Winikoff 2004) is agent-based software engineering methodology supposed to cover the overall development process. Three main phases are supported: (1) *system specification*, where the operating environment is identified along all goals and functionalities of the system; (2) *architectural design*, where the overall structure of the system is given and needed type of agents and their interactions are specified; (3) *detailed design*, which focuses on defining capabilities, internal events, plans, and detailed data structures for each agent.

Prometheus uses scenarios as variants of the scenarios introduced by UML use cases, and interaction diagrams are essentially UML sequence diagrams. Use cases scenarios are used in Prometheus to specify aspects of the system and describe examples of the system in operation. In the architectural design phase, the interactions between agents are defined using interaction diagrams and interaction protocols. The notation for this is a simplified variant of UML sequence diagrams for interaction diagrams, and AUML for the interaction protocol.

The overall structure of the system is specified in a single diagram type at different levels of detail: system, agent, and capability. Further diagrams are used to show data coupling and agent acquaintance relationships. Dynamic behavior is described with UML and AUML diagrams. In the system specification phase, Prometheus gives a strong emphasis to the determination of the system's goals and functionalities. The determination of goals results in an iterative process: identifying and refining system goals, grouping goals into functionalities, describing a functionality descriptor, defining use case scenarios (useful to identify missing goals), and checking whether all goals are covered by scenarios. Given an initial set of goals elicited from the initial requirements, the analyst refines and elaborates them using a hierarchical structure answering questions such as why goals are needed and how they can be achieved.

During the system specification phase, roles are defined and mapped into the system's functionalities. A role deals with a single aspect or subgoal of the system, and it has to

be very specific to avoid thus having functionalities that are too general, which can lead to potential misunderstanding. The definition of functionality also provides the specification of the information needed and produced, and it is linked to one or more system goals. Roles are also used in the architectural design phase to build data coupling diagrams that describe functionalities and identified data. From data coupling diagrams, it is possible to extract and elaborate constraints that can be used to build actual agents.

From scenarios, analysts develop interaction diagrams during the architectural design phase and then interaction protocols. Information about agent interactions are extracted from the functionality descriptors, and each agent type is linked to other agent types it interacts with. The specification of agents' interaction focuses mainly on the dynamic behavior of the system. UML sequence diagrams are adapted to represent interaction diagrams and are used as initial representations of agent interactions. Interaction protocols are final design artifacts.

Prometheus is tool-supported (Padgham, Thangarajah, and Winikoff 2008) by the Prometheus Design Tool (PDT) and the JACK Development Environment (JDE). PDT allow users to create and elaborate Prometheus design. In particular, PDT helps in avoiding the introduction of inconsistencies, and it provides cross-checking that detects other forms of inconsistency. Differently, JDE is used for the skeleton code generation from design diagrams. It guarantees also that changes made to the code are reflected in the design diagrams and vice versa.

Tropos (Bresciani et al. 2004) is requirement-driven in the sense that it is based on concepts used during early requirements analysis. Tropos adopts the concepts offered by *i** (Yu 1995), a modeling framework proposing concepts such as actor (actors can be agents, positions, or roles), as well as social dependencies among actors, including goal, softgoal, task, and resource dependencies. These concepts are used in all software development phases of Tropos, from the early requirements analysis down to the actual implementation. Tropos is a full tool-supported methodology (Morandini et al. 2008) that spans four phases that can be used either following the waterfall or the spiral model respectively for sequential and iterative development: (1) early requirements, (2) late requirements, (3) architectural design, (4) detailed design. Although, there are many proposals to integrate Tropos with agent-oriented programming frameworks, originally Tropos did not support the implementation phase.

Early requirements analysis focuses on the intentions of stakeholders. Intentions are modeled as goals. Through some form of goal-oriented analysis, these initial goals eventually lead to the functional and nonfunctional requirements of the system-to-be. In Tropos, stakeholders are represented as (social) actors who depend on each other for goals to be achieved, tasks to be performed, and resources to be furnished. The Tropos framework includes the strategic dependency model for describing the network of relationships among actors, as well as the strategic rationale model for describing and supporting the reasoning that each actor goes through concerning its relationships with other actors. A strategic dependency model is a graph involving actors who have strategic dependencies among each other. A dependency describes an "agreement" (called dependum) between a depending actor (depender) and an actor who is depended upon (dependee). The type of the dependency describes the nature of the agreement. Goal dependencies are used to represent delegation of responsibility for fulfilling a goal; softgoal dependencies are similar to goal dependencies, but their fulfilment cannot be defined precisely (for instance, the degree of fulfilment is subjective); task dependencies are used in situations where the dependee is required to perform a given activity; and resource dependencies require the dependee to provide a resource to the depender.

Late requirements analysis results in a requirements specification that describes all functional and nonfunctional requirements for the system-to-be. In Tropos, the system is represented as one or more actors that participate in a strategic dependency model, along with other actors from the system's operational environment. In other words, the system comes into the picture as one or more actors who contribute to the fulfillment of stakeholder goals. As late requirements analysis proceeds, the system is given additional responsibilities, and ends up as the dependee of several dependencies. A strategic rationale model determines through a means-ends analysis how the system goals (including softgoals) that were identified during early requirements can actually be fulfilled exploiting the contributions of other actors. A strategic rationale model is a graph with four types of nodes—goal, task, resource, and softgoal—and two types of links—means-ends links and decomposition links. A strategic rationale graph captures the relationship between the goals of each actor and the dependencies through which the actor expects these dependencies to be fulfilled.

A Tropos system architecture constitutes a relatively small, intellectually manageable model of system structure, which describes how system components work together. Tropos offers a catalogue of organizational architectural styles for cooperative, dynamic, and distributed applications—such as multiagent systems—to guide the design of the system architecture. These organizational architectural styles are based on concepts and design alternatives coming from research in organization management. As such, they help match a multiagent system architecture to the organizational context within which the system will operate.

TOOLS, PLATFORMS, AND PROGRAMMING LANGUAGES (FRAMEWORKS)

For the development of multiagent systems it is necessary to cast the agent concepts and architectures to concrete implementation means. In order to avoid the burden of constructing agent systems from scratch for each new application, several kinds of ancillary tools can be employed. In general, the tools can be categorized into *development tools* needed for building an application and the runtime infrastructure (called *agent platform*) needed to execute agent applications. An agent platform offers the basic management services for hosting agents

on a uniform infrastructure and additionally exposes ready-to-use communication mechanisms for the agents. Conceptually, a blueprint for agent platforms has been proposed in the FIPA abstract architecture (cf. the section on Tools, Platforms, and Programming Languages (Frameworks)). Besides management functionalities, an agent platform is characterized by the kind of agents that can be executed. Therefore, the development of applications using an agent platform heavily depends on the supported internal and social agent architectures. In this respect, the internal architecture determines the concepts and mechanisms that can be used for agent behavior programming, whereas the social architecture specifies which notions can be used for realizing coordination between agents and team management. Technically, a platform is characterized by the programming language it provides for realizing agents and the available tools for development, administration, and debugging.

Today, there is a multitude of commercial and open-source agent platforms available in the market. Hence, in the following, only a broad overview can be given and a small cutout of these presented in more detail. In order to present a meaningful selection of platforms, agent platforms are categorized based on a coarse classification, and one typical representative of each primary category is exemplified. This classification scheme, which was initially proposed by Braubach and colleagues (2006), is depicted in Figure 22.2. It distinguishes platforms by means of their primary focus and proposes three main categories: middleware, reasoning, and social-oriented platforms.

Middleware Platforms

In the context of distributed systems middleware is seen as a software layer between an application and the operating system providing generic services that are beyond the functionalities of the operating system and can be reused within different kinds of applications (Coulouris, Dollimore, and Kindberg 2005). Examples of such functionalities include directory services and message passing mechanisms.

In the field of multiagent systems middleware platforms play a similar role and have in common that they focus on a sound technological base for the execution of agents. Therefore they emphasize aspects such as interoperability, robustness, scalability, and mobility. Under this point of view, mobile agent toolkits such as Grasshopper (Bäumer et al. 1999), which allow agents to migrate between different hosts, can also be seen as part of the middleware category. The most important characteristic supported by nearly all middleware platforms is interoperability, which has been realized by the adherence to the FIPA standards. In many cases, representatives of this category do not use sophisticated agent architectures but rather rely on the *task model*, which assembles the overall behavior from simpler behavior modules. For this reason, most middleware platforms do not need specific agent programming languages, and typical mainstream object-oriented languages such as Java can be used. In the following the JADE platform will be presented as a typical representative of the agent middleware category.

JADE overview. The JADE platform (Java Agent Development Environment) is developed as open-source software by the Telecom Italia Lab (TILAB) since 1998 (Bellifemine et al. 2005). JADE has a big user community and has been adopted for applications from many different areas. As one example Whitestein has used JADE to construct an agent-based system for decision-making support in organ transplant centers (Calisti et al. 2004).

JADE agent architecture. In JADE, agents are specified in terms of a behavior-based architecture. A behavior

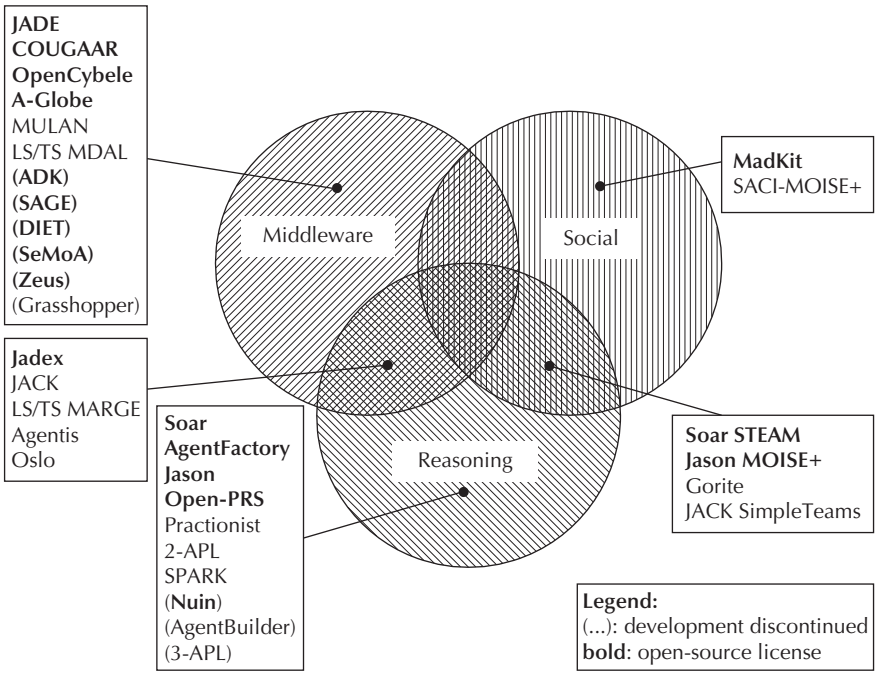

Figure 22.2: Classification and Overview of Commercial and Open-Source Agent Platforms

corresponds to a task and serves for the encapsulation of a specific functionality. An agent can be supplied with arbitrarily with many behaviors in order to work on different tasks concurrently. The communication among behaviors is realized by shared data stores, which can be used to make visible processing results for one another. For managing complexity, behaviors can be hierarchically assembled. The execution of subbehaviors is determined by the containing behavior and can be sequential, parallel or based on a finite state machine. Each JADE agent is executed in a separate thread, which performs a cooperative nonpreemptive scheduling, that is, the agent maintains a list of all active top-level behaviors and executes one step of each behavior in a round-robin fashion.

JADE language. JADE does not utilize an agent-oriented programming language but instead employs Java and offers agent-based functionalities such as message sending through an application programming interface (API). For a communication language the standardized FIPA-ACL (Agent Communication Language) is used, which ensures that JADE agents can communicate with agents living on other FIPA-compliant agent platforms. In addition, JADE supports most of the FIPA content languages such as SL (Semantic Language) and RDF (Resource Description Framework) for describing the message content separately from the rest of the message. To facilitate the communication in open systems JADE also allows using ontologies for a shared understanding of the used domain concepts. If such ontology objects need to be transmitted between agents specific content encoders and decoders are provided that are able to transform the content to a specified content language.

JADE tools. There is a broad range of tools available for developing agent applications with JADE. Nonetheless, most tools target the administration and debugging of multiagent systems, whereas earlier development phases are barely supported. As Java is employed for programming agents, common object-oriented integrated development platforms (IDEs) such as Eclipse can be used without restrictions. The central access point for the standard runtime tool suite of JADE is the remote monitoring agent (RMA), which offers a graphical user interface and can be used for starting the other tools. The RMA mainly exposes basic management functionalities for starting and killing agents. Other runtime tools allow the sending of messages to agents (dummy agent) and the stepwise execution and monitoring of agent behavior (introspector agent). For the debugging of multiagent systems the sniffer tool is quite helpful, as it visualizes the messages between agents in a style similar to UML sequence diagrams.

Reasoning Platforms

These kinds of platforms center on the internal reasoning processes of agents and aim at providing possibilities for the efficient specification and execution of intelligent agent behavior. The common characteristic of reasoning platforms is that they rely on psychological or philosophical theories for explaining rational human behavior. Thus, the primary aim of platforms from this category consists of making those rather abstract theories usable for the concrete task of application development. For this purpose agent architectures and agent programming languages have been conceived which refine, extend, and interpret the basic theories. In many cases these theories adopt the intentional stance (Dennett 1971), which uses human-centered mentalistic notions such as beliefs and goals for behavior explanations. It has been argued that it is useful to preserve the intentional stance also for the implementation of agents, because the notions can be used for such matters as simplifying debugging of complex systems (McCarthy 1979). Thus, in many cases reasoning platforms encompass newly conceived agent programming languages including mentalistic notions. As an example reasoning platform Jadex will be further illustrated.

Jadex overview. The Jadex framework has been developed as an open-source project at the University of Hamburg since 2003 (Pokahr, Braubach, and Lamersdorf 2005). It follows the BDI model (Bratman 1987) and allows goal-oriented agents to be built with standard software-engineering technologies such as Java and XML. Jadex separates the reasoning engine for managing agent behavior from the underlying agent execution infrastructure. Given this separation, Jadex can be used in conjunction with different kinds of middleware such as other agent platforms (like JADE) or component-based approaches (like J2EE application servers). Jadex has been used to realize applications in different domains such as simulation, scheduling, and business process management. For example, Jadex was used to realize a multiagent application for negotiation of treatment schedules in hospitals (Paulussen et al. 2006).

Jadex agent architecture. The behavior of an agent is defined in terms of beliefs, goals, and plans in Jadex. Goals represent the motivations of an agent and finally determine the procedural behavior pursued that is encoded within plans. Beliefs represent the knowledge of an agent and typically reflect its perception of the environment, itself, and other agents. In Jadex, goals are decoupled from any concrete behavior specification and just express what an agent wants to achieve, avoid, or maintain from a high-level perspective. The notion of goals is very similar to its general usage and supports many important characteristics such as the possibility for handling strategic long-lived as well as more tactical short-term goals. Given that an agent can posses an arbitrary number of goals, it is of vital importance to decide which of its goals may conflict and what to do if such situations arise. For this purpose Jadex offers a generic *goal deliberation* strategy, which enables an agent to reason about its current goals and is driven by the overall objective of pursuing only conflict-free goal sets at any point in time. The relationships among goals are specified by the agent developer at design time and will be enforced by the reasoning engine at runtime. A further important step is that an agent has to determine how it can achieve these goals. For this purpose PRS means-end reasoning is used, meaning that appropriate plans are dynamically selected and executed for a goal until the goal has been achieved or no more plans are available.

Jadex language. Even though Jadex allows for programming with mentalistic notions, it does not introduce a new agent programming language but relies on the standard languages XML and Java. XML is used for the specification of the agent structure according to a BDI metamodel, which defines the permissive tags and

attributes of an agent. In addition, the procedural knowledge of an agent, that is, its plan bodies, can be directly programmed in plain Java. Agent-related behavior is made accessible through a framework API, which permits, for example, the dispatching of subgoals and the reading and writing of belief values. The communication language of Jadex depends on the middleware it is used with and can, for instance, be made FIPA-compatible by using JADE as infrastructure layer.

Jadex tools. Jadex offers various tools for developing agent systems and focuses on activities for administration and debugging. The implementation of agents can be done using standard object-oriented IDEs that already offer sophisticated programming support for Java as well as schema-based XML documents. The tool suite mainly consists of the Jadex Control Center, which represents the plugin-based entry point for tool components. Besides administration tools for starting and stopping agents and monitoring the state of directory services debugging tools also allow the inspection of an agent's state as well as its stepwise execution. Using the simulation tool it is possible to control the advancement of time within an execution

Social Platforms

Social agent platforms underline the importance of coordination and cooperation aspects within multiagent systems. Thus, the focus of social platforms is not so much concerned with providing concepts for specifying individual behavior. Instead, concepts and mechanisms are targeted that allow for setting-up group behavior of teams of agents. These systems build upon the already discussed group behavior theories and architectures. Due to the lack of integrated approaches, the support of agent platforms for the organizational metaphor is rather limited and restricted to either the structure or behavior dimension. In the following, the MadKit framework will be presented as an example for a platform using structural behavior concepts.

MadKit overview. The MadKit (Multi-Agent Development kit) platform was developed as open-source by Ferber and colleagues (Gutknecht, Ferber, and Michel 2001). It represents an agent framework adhering to the AGR model and therefore takes a structural perspective on organization modeling. The platform is based on a microkernel, which only includes indispensable services for agent lifecycle management, group management, and local message transport. All further services have been agentified and can be added to the kernel on demand. The framework has already been used for the realization of applications covering a wide range of domains including simulations of submarine robots and production line logistics.

MadKit agent architecture. MadKit focuses strongly on the organizational view of multiagent systems and hence does not implement a specific agent architecture to be used by an agent programmer. On the one hand, this gives agent developers complete freedom about how to build their agents manually without further support from the framework, but on the other hand this also *requires* him to do so. An agent in MadKit is regarded as an autonomous object that can communicate via messages and play roles in groups. The framework specifies from an outside view how an agent can be executed, and the adherence to this interface is the only restriction MadKit agents need to follow. Basically, the platform expects an agent to have methods for the initialization, execution, and shutdown that will automatically be called by the platform when an agent will be executed. MadKit exploits this freedom by already providing different simple agent types that can, for instance, be rule-based or state-oriented.

MadKit language. In addition to the agent architecture independence of MadKit, the platform also supports different (standard) languages for programming agents. Besides Java, which is the main language, the platform also has built-in support for Scheme, Python, and Jess. This allows developers to implement agents with a programming language of their choice. The communication language of MadKit is also configurable. In its basic form, agents communicate via simple message objects that can contain arbitrary content objects. Using specialized message objects it is also possible to transmit FIPA-ACL messages. Interestingly, the communication in MadKit is also connected to the underlying AGR concepts. Hence, it is possible to send or broadcast messages to specific roles or groups instead of concrete agents.

MadKit tools. The Madkit distribution contains, besides the platform, various development and runtime tools. The platform offers a MadKit desktop, which contains shortcuts to the available tools as well as many example applications. For the implementation of agents a developer can make use of source code editors that support the different built-in programming languages. In addition, a designer tool can be used to set up MadKit projects and associate agents and other resources to a project context. At runtime, MadKit provides the group observer tool, which makes the organizational structures visible and shows which groups and agents exist. In addition, the tool allows conversations to be visualized as UML sequence diagrams.

APPLICATION AREAS

Scope of Application

Intelligent agents have proven particularly suitable for the implementation of applications with the following characteristics:

- *Distribution:* Data, information, and knowledge are geographically and/or logically distributed and are processed as such.
- *Parallelism/concurrency:* The data are processed in parallel/concurrently.
- *Openness:* The number and the type of hardware and software components involved in the application are variable and possibly not precisely known a priori (on design).
- Embedded in complex (dynamic, unpredictable, limited transparency, heterogeneous, etc.) socio-technical environments (situated applications).

With advancing technological progress, such as computer networking and platform interoperability, such applications are gaining importance in a broad range of commercial, industrial, and scientific domains. In general, these three characteristics represent a multitude of applications that

are based on new models of and approaches to computer-supported information processing, such as grid computing, peer-to-peer computing, Web computing, pervasive and ubiquitous computing, autonomic computing, and mobile computing. Their suitability for such applications ensues from their attributes corresponding with the three key attributes of an agent—flexibility, interactivity, and autonomy. First, the characteristics *distribution* and *openness* imply a distributed and open control structure (which enables parallel and concurrent processing) and thus the necessity to use software units for the implementation that can act autonomously (without central control). Second, the characteristics *openness* and *embeddedness* imply the necessity to employ software units that are as flexible as possible, such as software units that are capable of acting suitably despite unexpected changes in the technological infrastructure or in the user requirements. Third, the characteristics *distribution*, *openness*, and *embeddedness* imply the necessity to employ software units that are capable of interacting as flexibly and autonomously as possible.

Application Domains

The application areas for multiagent systems can be categorized and described according to different criteria. In the literature two rather orthogonal ways of categorizations can be found: using *application sectors* and *application classes*. Sectors here refer to the type of business such as industry or health care, whereas classes focus on the underlying type of solution such as simulation or robot control.

Figure 22.3 presents a matrix according to the two categorization dimensions sketched before. The choice of application sectors used here follows the proposal of Jennings and Wooldridge (1998) and adds the military domain. A more fine-grained breakdown of sectors can be found in Luck and colleagues (2005). The selection of application classes is loosely based on Ferber (1999), but also incorporates the proposal of Wooldridge (2002). The categorizations of sectors as well as of classes should not be considered as complete, but are open for further refinements and extensions. Despite this issue, the spanned matrix already allows giving an impression of the possibilities of multiagent systems and an overview of the areas in which they have shown to be able to contribute to novel innovative solutions. In the following each of the application sectors will be explained in more detail. For further detailed overviews of agent applications, see, for instance, Klügel (2004) and Parunak (2000).

CONCLUSION

Over the past decade considerable progress has been achieved in the field of agent and multiagent technology, and, as a result, today intelligent agents and agent-oriented systems are gaining increasing attention in industrial contexts. This attention mainly rests on the insight that these systems have a significant application potential in a variety of complex domains, and much of the current worldwide research on intelligent agents aims at putting this potential into practice.

This chapter concentrated on several aspects of intelligent agents that are of particular and direct relevance to broad industrial acceptance and dissemination. Other facets that are also essential to computational agency but are not covered in this chapter due to limited space are, for instance, automated negotiation, cooperative planning, and joint learning; the reader interested in a broader depiction of intelligent agents is referred to Weiss (1999) and Wooldridge (2002).

GLOSSARY

Agent: A self-contained computational (hard/software) entity that handles its tasks in a knowledge-based, flexible, interactive, and autonomous way

Agent architecture: Information and control flow within an agent; more specifically, the arrangement of data, algorithms, and control structures that an agent uses in order to decide on his actions.

Class \ Sector	Industrial Applications	Commercial Applications	Entertainment Applications	Medical Applications	Military Applications	…
Multi-Agent Simulation	Factory simulations	Market/trading simulations	Movie scene Productions/ Games	Hospital simulations	Battlefield Simulations/ Pilot training	…
Problem Solving	Goods transport	E-Business	Strategy games	Hospital logistics	War logistics	…
Robot Control	Production robots	Household robots	"Intelligent" toys	Medical device control	Unmanned aerial vehicles	…
Information Management	Tracking and Tracing	Web search Email filtering	Artificial game reporters	Disaster management/ Medical information management	Decision support/ Smart dust	…
Human Computer Interface Mgmnt.	Augmented reality tools	Shop bots/ Help assistants	Avatars in games	Telemedicine/ Home care management	Augmented reality tools for soldiers	…
…	…	…	…	…	…	…

Figure 22.3: Overview of Multiagent Application Areas

Agent communication language: A formal language that allows agents to exchange knowledge and to interact in a sophisticated manner at the knowledge level.

Agent-oriented programming: The programming of software in terms of agent-specific mentalistic notions (e.g., belief and desire) as well as agent-specific organizational notions (e.g., group and coalition).

Computational autonomy: The ability of a computational entity to act under self-control and to make decisions even in complex and perhaps unforeseen situations.

Multiagent system: A system composed of at least two agents; often used synonymously to agent system intelligent agent, computational agent, autonomous agent software agent → agent.

CROSS REFERENCES

Distributed Intelligent Networks; Expert Systems; Intelligent Manufacturing Systems; Web Intelligence.

REFERENCES AND SUGGESTED READINGS

Bäumer, C., M. Breugst, S. Choy, and T. Magedanz. 1999. Grasshopper—A universal agent platform based on OMG MASIF and FIPA standards. *Proceedings of First International Workshop on Mobile Agent for Telecommunication Applications (MATA'99)*:1–18.

Bergenti, F., M.-P. Gleizes, and F. Zambonelli (Eds.). 2004. *Methodologies and software engineering for agent systems: The agent-oriented software engineering handbook*. Boston, Dordrecht, London: Kluwer Academic Publishers.

Bernon, C., M.-P. Gleizes, G. Picard, and P. Glize. 2002. The ADELFE methodology for an intranet system design. Online-Proceedings of the *AOIS Workshop 2002*. http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-57.

Bratman, M. 1987. *Intention, plans, and practical reason*. Cambridge: Harvard University Press.

Bratman, M., Israel, D., and Pollack, M. 1988. Plans and resource-bounded practical reasoning. *Computational Intelligence* 4 (4):349–355.

Braubach, L., A. Pokahr, and W. Lamersdorf, (2008). A Universal Criteria Catalog for Evaluation of Heterogeneous Agent Development Artifacts. In Jung, B. F., Michel, A., Ricci, and P. Petta, (Eds.), From Agent Theory to Agent Implementation (AT2AI-6), OFAI Technical Report No. 2008-01, Austrian Research Institute for Artificial Intelligence (OFAI), pp. 19–28.

Braubach, L., Pokahr, A., and Lamersdorf, W. (2006). Tools and Standards. In Kirn, S., Herzog, O., Lockemann, P., and Spaniol, O. (Eds.), *Multiagent Engineering—Theory and Applications in Enterprises*. Springer-Verlag, pp. 503–530.

Bresciani, P., P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. 2004. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems* (18):203–236.

Brooks, B. 1989. How to build complete creatures rather than isolated cognitive simulators. In K. VanLehn, (Ed.), *Architectures for Intelligence*. Erlbaum, Hillsdale, NJ, pp. 225–239.

Caire, G., W. Coulier, F. Garijo, J. Gomez, J. Pavon, F. Leal, P. Chainho, P. Kearney, J. Stark, R. Evans, and P. Massonet. 2001. Agent oriented analysis using MESSAGE/UML. In M., Wooldridge, G., Weiß, and P. Ciancarini, (Eds.). Agent-oriented software engineering II, LNCS 2222. Berlin, New York, Heidelberg, Springer-Verlag, pp. 119–135.

Calisti, M., P. Funk, S. Biellman, and T. Bugnon. 2004. A multi-agent system for organ transplant management. *Applications of Software Agent Technology in the Health Care Domain* Heidelberg: Springer-Verlag.

Cohen, P.R., and H.J. Levesque. 1991. Teamwork. Menlo Park, CA., SRI International. Research Report.

Coleman, D., P. Arnold, S. Bodoff, C. Dollin, and H. Gilchrist. 1994. Object-oriented development. *The fusion method*. New Jersey: Prentice Hall.

Collinot, A., and A. Drogoul. 1998. Using the Cassiopeia method to design a soccer robot team *AAI Journal* 12 (2–3):127–147.

Cossentino, M. 2005. From requirements to code with the PASSI Methodology. In *Agent-Oriented Methodologies*, ed. B. Henderson-Sellers and P. Giorgini. Hershey, PA: Idea Group Inc., pp. 79–106.

Coulouris, G.F., J. Dollimore, and T. Kindberg, 2005. *Distributed systems*. New Jersey: Addison-Wesley.

Debenham, J., and B. Henderson-Sellers. 2003. Designing agent-based process systems—extending the OPEN Process Framework. In V. Plekhanova, (Ed.), *Intelligent Agent Software Engineering*, PA: Idea Group Publishing, pp. 160–190.

Dennett, D. 1971. Intentional systems. *Journal of Philosophy* 68:87–106.

Ferber, J. 1999. *Multi-agents systems—An introduction to distributed artificial intelligence*. New Jersey: Addison-Wesley.

Ferber, J., O. Gutknecht, and F. Michel. 2003. From agents to organizations: An organizational view of multi-agent systems. In *Proceedings of the 4th International Workshop on Agent-Oriented Software Engineering* IV. Springer, pp. 214–230.

Garcia-Ojeda, J.C., S.A. DeLoach, W. Oyenan, and J. Valenzuela. 2007. O-MaSE: A customizable approach to developing multiagent development processes. In M. Luck, and L. Padgham, (Eds.), *Proceedings of the 8th International Workshop on Agent Oriented Software Engineering*. LNCS 4951. Heidelberg, Springer-Verlag, Berlin, New York, pp. 1-15.

Gómez-Sanz J.J R., J. Fuentes-Fernández Pavón, I. García-Magariño. 2008. INGENIAS development kit: A visual multi-agent system development environment. *The Seventh International Conference on Autonomous Agents and Multiagent Systems*:1675–1676.

Gutknecht, O., J. Ferber, and F. Michel. 2001. Integrating tools and infrastructures for generic multi-agent systems. *Proceedings of the Fifth International Conference on Autonomous Agents*:441–448.

Henderson-Sellers, B., and P. Giorgini (Eds.). 2005. *Agent-oriented methodologies*. PA:Idea Group Publishing.

Huget, M.-Ph. 2002. Nemo: An agent-oriented software engineering methodology, *Proc. AOSE Workshop* Sydney.

Iglesias, C.A.M., Garijo, J.C. Gonzalez, and J.R. Velasco. 1998. Analysis and design of multi-agent systems

using MAS-CommonKADS. *Intelligent Agents IV: Agent Theories, Architectures, and Languages, LNAI* 1365, Heidelberg, Berlin, New York: Springer-Verlag,

Jennings, N.R. 2000. On agent-based software engineering. *Artificial Intelligence* 117:277–296.

Kendall, E.A., M.T. Malkoun, and C. Jiang. 1996. A methodology for developing agent based systems for enterprise integration. In P. Bernus, and L. Nemes, (Eds.). *Modelling and Methodologies for Enterprise Integration*. Heidelberg, Berlin, New York: Springer-Verlag.

Kinny, D., Georgeff, M., & Rao, A. (1996). A methodology and modelling techniques for ystems of BDI agents. Technical Note 58, Australian Artificial Intelligence Institute, also published in Proceedings of Agents Breaking Away, the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96) (pp. 56–71). Springer-Verlag.

Klügel, F. 2004. Applications of software agents. *Künstliche Intelligenz* 2 (4):5–10.

Kruchten, P. 1999. *The rational unified process: An introduction.* Reading, MA: Addison-Wesley.

Lind, J. 1999. Iterative software engineering for multiagent systems. *The MASSIVE Method, LNAI* Vol. 1994. Heidelberg, Berlin, New York: Springer-Verlag.

Luck, M., R. Ashri, and M. D'Inverno (Eds.). 2004. *Agent-based software development*. Boston, London: Artech House.

Luck, M., P. McBurney, O. Shehory, and S. Willmott. 2005. *Agent technology: Computing as interaction (A roadmap for agent based computing)*. UK: AgentLink.

McCarthy, J. 1979. Ascribing mental qualities to machines. In M. Ringle (Ed.). *Philosophical Perspectives in Artificial Intelligence*. pp. 161–195. Humanities Press.

Morandini, M., D.C. Nguyen, A. Perini, and A. Susi. 2008. Tool-supported development with Tropos: The conference management system case study. *Proceedings of 8th International Workshop on AGENT ORIENTED SOFTWARE ENGINEERING (AOSE 07)*. Revised Selected Papers. LNCS 4951 Heidelberg, Berlin, New York: Springer.

Newell, A., and H.A. Simon. 1976. Computer science as empirical enquiry. *Communications of the ACM* 19:113–126.

Object Management Group (OMG): Mobile Agent Facility Specification. Available at: www.omg.org/cgi-bin/doc?formal/2000-01-02, 2000.

Padgham, L., and M. Winikoff. 2004. *Developing intelligent agent systems: A practical guide*. Hoboken, NJ: John Wiley & Sons, Inc.

Padgham, L., J. Thangarajah, and M. Winikoff. 2008. The Prometheus design tool—A conference management system case study. In M. Luck, and L. Padgham, (Eds.). *Proceedings of 8th International Workshop on Agent Oriented Software Engineering*, LNCS Vol. 4951. Heidelberg, Berlin, New York: Springer-Verlag, pp. 197–211.

Parunak, V. 2000. A practitioners' review of industrial agent applications. *Autonomous Agents and Multi-Agent Systems* 3 (4):389–407.

Paulussen, T.O., A. Zöller, F. Rothlauf, A. Heinzl, L. Braubach, A. Pokahr, and W. Lamersdorf. 2006. Agent-based patient scheduling in hospitals. In *Multiagent engineering. theory and applications in enterprises*, ed. S. Kirn, O. Herzog, O., P. Lockemann, O. Spaniol: 255–275. Heidelberg, Berlin, New York: Springer-Verlag.

Pavón, J., J. Gomez-Sanz, and R. Fuentes. 2005. The INGENIAS methodology and tools. In B. Henderson-Sellers, and P. Giorgini (Eds.). *Agent-Oriented Methodologies* (pp. 236–276). Hershey, PA: Idea Group.

Pokahr, A., L. Braubach, and W. Lamersdorf. 2005. Jadex: A BDI reasoning engine. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, (Eds.). *Multi-Agent Programming: Languages, Platforms and Applications*. 149–174. Heidelberg, Berlin, New York. Springer-Verlag.

Rumbaugh, J., M. Blaha, W. Premerlani, F.Eddy, and W. Lorensen. 1991. *Object-Oriented Modeling and Design*. New Jersey: Prentice-Hall.

Taveter, K., and G. Wagner. 2005. Towards radical agent-oriented software engineering processes based on AOR modeling. In B. Henderson-Sellers, and P. Giorgini (Eds.). *Agent-oriented methodologies*. Hershey, PA: Idea Group. (pp. 277–316).

Wagner, G. 2003. The agent-object relationship metamodel: Towards a unified view of state and behaviour. *Inf. Systems* 28 (5): 475–504.

Willmott, S., M. Calisti, and E. Rollon. 2002. Challenges in large-scale open agent mediated economies. *Proceedings of AAMAS '02: Revised Papers from the Workshop on Agent Mediated Electronic Commerce on Agent-Mediated Electronic Commerce IV, Designing Mechanisms and Systems*. Berlin Heidelberg New York: Springer.

Weiss, G. 2002. Agent orientation in software engineering. *Knowledge Engineering Review* 16 (4):349–373.

Weiss, G. (Ed.). 1999. Multiagent Systems. Cambridge: MIT Press.

Wooldridge,. M. 2002. *Introduction to multiagent systems.* Hoboken, NJ: John Wiley & Sons, Inc.

Wooldridge, M., and N. Jennings, 1994. Agent theories, architectures, and languages: A survey. *Proceedings of the International Workshop on Agent Theories, Architectures & Languages* (ECAI'94). Heidelberg, Berlin, New York: Springer-Verlag.

Yu, E. 1995. Modelling strategic relationships for process reengineering Ph.D. dissertation. University of Toronto, Department of Computer Science.

Zambonelli, F., N. Jennings, and M. Wooldridge. 2003. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology* 12 (3): 317–370.

WEB POINTERS TO FURTHER IMPORTANT RESOURCES

AAMAS conference series. www.aamas-conference.org.

IEEE FIPA standardization committee. www.fipa.org/about/fipa_and_ieee.html.

International Journal of Agent-Oriented Software Engineering.www.inderscience.com/browse/index.php?journalID=174.

International Journal on Autonomous Agent and Multi-Agent Systems (AAMAS). www.springer.com/computer/artificial/journal/10458.