Intelligent Agents

Authors: Dr. Gerhard Weiss, SCCH GmbH, Austria
Dr. Lars Braubach, University of Hamburg, Germany
Dr. Paolo Giorgini, University of Trento, Italy

Outline

Abstract

The concept of an intelligent agent, having its origin in artificial intelligence agent technology, denotes a computational unit which is able to carry out tasks flexibly, autonomously, and interactively in complex environments. Today this concept is well established in computer science and information technology and agent technology is an integral part of an increasing number of industrial and commercial applications.

This article overviews key aspects of intelligent agents, including engineering issues, architectural issues, development methodologies, implementation frameworks (tools, platforms and programming languages), and agent-specific industrial standards and application domains. The article starts with an introduction to the elementary foundations of intelligent agents and concludes with references to useful related literature and online resources.

Key Words

intelligent agents; multiagent systems; agent architecture; agent-oriented engineering; agent platform; agent-oriented programming; computational autonomy; cooperation; coordination

1 Foundations of Intelligent Agents

The agent concept originated in artificial intelligence and especially in the field of agent and multi-agent 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 continues to some extent today. Here we introduce two notions of agent concepts that have become established in more recent literature.

Weak notion. Recent years have seen broad acceptance of the following notion of agent concepts, which the literature frequently terms *weak agents (weak notion)*:

> *An 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 interaction 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, i.e., 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 non-trivial, i.e., that they might 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. At the bottom line, autonomy implies the ability of an

agent to independently handle its own complexity and that of its application and thus especially to relieve its users while protecting their interests.

Note that each of these three key attributes can exist in various forms and intensities and so the transition from agent to non-agent is a grey zone.

Strong notion. A widely disseminated alternative to the above agent concept is that of strong agents (strong notion), by which an agent is a (hardware/software) unit that, analogous to people, possesses mental attitudes or states. Three types of mental states play a particular role for intelligent agents:

- *information-related states* such as knowledge, presumptions and assumptions
- *connotative states* such as intentions, plans and duties (with respect to others or themselves)
- *affective states* such as goals, preferences and desires

Another type of mental states that we can distinguish is emotional states. Over the past several years, agents that can show emotion (joy, surprise, fear, etc., e.g., as mimic and gesture) have been increasingly addressed, especially in the context of multimedia human/computer interfaces.

Relationship of the two notions. While the weak notion of agents primarily involves generic functional attributes, the strong notion of agents primarily involves the architecture and the internal (control) structure and thus generic structural attributes of an agent. For example, from the statement that an agent has knowledge, it can be derived that it has a structural component (a knowledge base) in which it stores such knowledge, and from the statement that an agent pursues plans, it can be derived that it contains a planning module as well as a plan-conforming control system. Weak and strong notions of agents overlap at least partially and are seen as complementary perspectives of the agent concept; in fact, the vast majority of work in research and application builds on both notions; i.e., both approaches are normally applied in combination.

Further agent attributes. Very often the above notions are 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. Thus it acts and interacts directly in a concrete and socio-technical environment and not only in an abstract model of this environment.
- *Learning capability/adaptivity.* An agent independently optimizes its functionality with respect to the tasks that are assigned to it, which might change over time.

Other attributes that the literature often designates as elementary for agents and that are noteworthy from a software engineering viewpoint include *persistency* (an agent does not simply implement a one-time computation, but acts over a longer period of time); *rationality* (an agent acts in the realm of its capability and knowledge as well as possible with respect to fulfilling its task and goals; i.e., it maximizes its chances of success); *benevolence* (an agent does not deliberately act contrary to the interests of a human user); and *self-containment* (an agent is a functionally complete and executable entity).

Usage notes. In agent-oriented software engineering as well as in the field of agent technology, the term *agent* is often used in modified form. Examples of such modifications, intended to underscore the most important attribute of the respective agent, include *autonomous* agent, *cooperating* agent, *reactive* agent, *adaptive* agent and *rational* agent. Often the terms *agent* and *intelligent agent* are used as synonyms.

In addition to such emphasis on certain attributes, we also find common formulations to precisely formulate the term *(software) agent* by complementing it with its domain or purpose. Familiar examples include *information* agent, *interface* agent, *wrapper* agent, *transaction* agent, *sales* agent, *assistance* agent, *virtual* agent and *mobile* agent.

The remainder of this article ist structured as follows. Secction 2 describes the agent-oriented perspective on software and systems engineering. Section 3 describes available approaches to agent architectures. Section 4 overviews the state of the art in systematically developing systems from the perspective of agent orientation. Section 5 presents important frameworks for building agent applications, including tools, platforms and programming languages. Section 6 presents current standardization efforts for agent-oriented systems. Section 7 characterizes applications and application areas that are particularly suited for the agent-oriented approach.

2 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. These attributes also provide the reason for the rapidly growing interest that intelligent agents have been enjoying in recent years.

2.1 Key Attributes of Agent-Oriented 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:

- Decomposition: the reduction into smaller and thus comprehensible parts that can then be developed largely independently.
- Abstraction: the creation of a model that encompasses significant aspects while hiding unimportant aspects.
- Structuring: the specification of (ordered) relationships and (desired) mutual effects among the components of the overall system.
- 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. First, it enables the targeted decomposition of a software system into atomic parts (agents) and constructs (agent groups) composed of these agents. Due to the semantics of the agent concept, a random or completely unsuitable (with respect to the application) system decomposition is unlikely. That is, the agent concept is semantically rich enough to provide concrete help and directions for system decomposition (compare the concepts *object* and *component*). Second, the paradigm enables the modeling of systems and applications on the knowledge level and social level and thus affords multifarious options for systematic abstraction of implementation and requirement details. Third, relationships and dependencies between individual parts (agents) can be derived directly from the interactions that are required or permitted in the realm of their task processing. The spectrum of possible relationships extends from classical client/server structures to market-based structures to peer-to-peer structures. Fourth, there are a number of agent-specific artifacts that are normally generated in agent-oriented software development and that are superbly suited for reuse. Examples of such artifacts include an individual software agent (i.e., program code that implements an agent), a team of software agents that jointly handle a task; agent-internal components (e.g., the knowledge base or planning component of an agent); architectures of individual agents and of agent teams; interaction structures and protocols; and complete agent platforms.

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 thus evolved – 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 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 literature often compares this extended encapsulation of the agent concept with the object concept in the sense of the market-dominant object oriented paradigm; a similar comparison applies for the relationship between agent and component concepts: While objects, in addition to their identity (who?) and their state (what?), encapsulate passive behavior (what, if activated?), agents encapsulate additional degrees of freedom in their (inter)activity and thus active behavior (how, when and with whom, if at all?). These familiar slogans express the difference: "Objects do it for free; agents do it because they want to." "Objects do it for free; agents do it for money."

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 quasi per 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). Autonomy as a desired or required attribute of IT systems – in various nuances and variants (self-governing, self-structuring, self-healing, self-repairing, etc.) – has also served as the focus of various initiatives by leading representatives of the IT branch in recent years. Examples include IBM's Autonomic Computing Initiative, Sun's N1 Initiative, HP's Adaptive Enterprise Initiative and Microsoft's Dynamic Systems Initiative (whereby the last three focus almost exclusively on servers and infrastructure). These initiatives share the vision of autonomous IT systems that hide their own complexity and that of their environment from their human users.

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 *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.

2.2 Summary and Challenges

Software development is much too multifaceted and complex for any silver bullet that always (or even usually) delivers an optimum software system while upholding available time and cost constraints. Obviously object orientation is no such panacea, and it would be unrealistic to assume that agent orientation or any other development approach would be such. The agent-oriented paradigm affords a number of innovative concepts and potential benefits. The "real" utility of any engineering perspective or paradigm can only be assessed in practice based on years of experience. Like component and architecture orientation, agent orientation is still too young to provide solid, empirical answers to the question of utility. In the following, we address three fundamental challenges that must be mastered in order to harvest these benefits.

One challenge that is easy to underestimate is the professional and well-founded handling of the agent concept. Due to its intuitive comprehensibility, this concept can quickly mislead developers (especially with a lack of knowledge and experience in agent-oriented development and agent technology) to lose all correlation to software-engineering relevance and feasibility and last but not least to the actual requirements on the target system. Truly there are many examples that show that the term *agent* is used too loosely and that some system units known as agents simply do not live up to this designation. Such superficial handling of the agent concept can render a suitable and sensible agent-oriented system implementation encumbered or even impossible.

A second core challenge is the correct and precise formulation and specification of autonomy as a software attribute. This challenge, which has become very important even beyond the agent-oriented paradigm due to the growing demand for autonomous information systems, also raises important questions of system security and privacy. This results because a software agent as an autonomous unit in the realm of its assigned responsibilities can make decisions that could have significant financial or legal consequences for its human users. Therefore a developer is confronted with the dilemma of making autonomy neither too restrictive nor too permissive, because otherwise desired effects (e.g., relieving users) are not achieved or undesired effects (e.g., emerging instability of the overall system) cannot be precluded.

The third and most prominent challenge is to link the agent-oriented software paradigm to quality and development standards that are relevant in practice. Although enormous progress has been made here in recent years, it does not suffice to enable widespread and comprehensive industrial and commercial application of agent-oriented software systems.

3 Architectures for Intelligent Agents

In the context of multi-agent 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, e.g. 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.

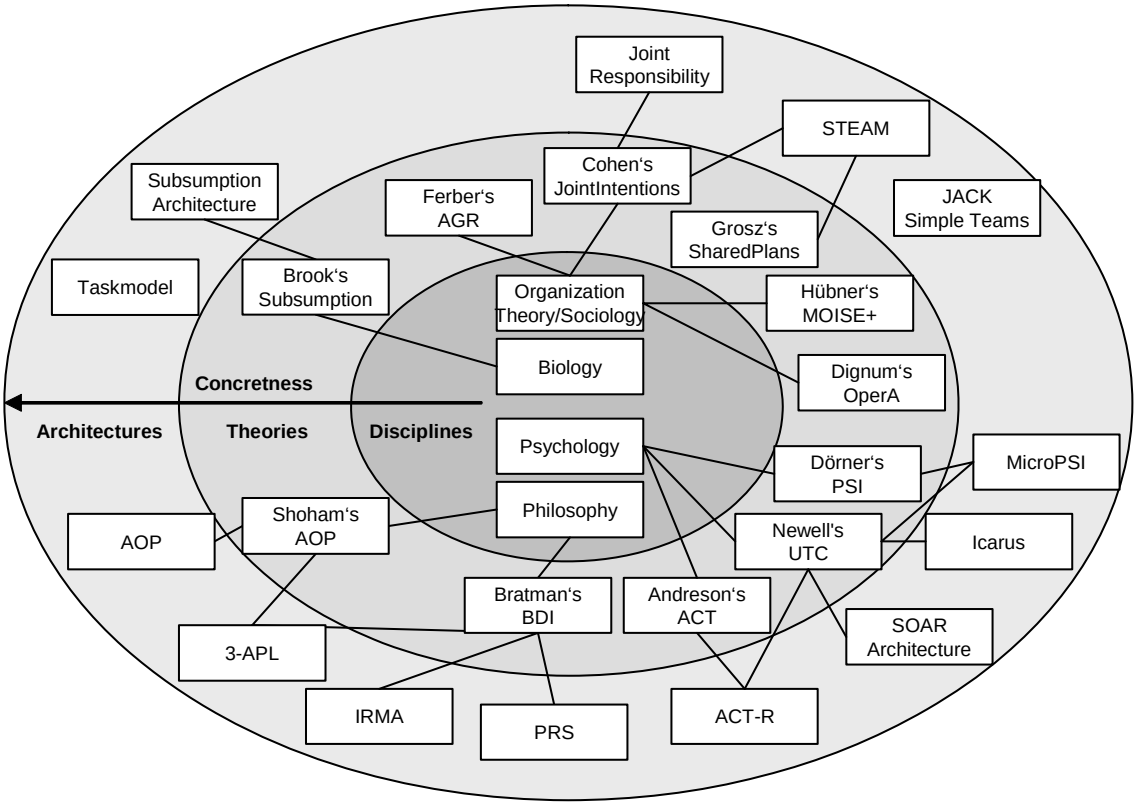

Figure 1: Overview of agent architectures (from Braubach et al. 2008)

Figure 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 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.

3.1 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 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 might be no longer applicable in the current situation. The subsumption architecture (Brooks 1989) is a typical example for a reactive control mechanism, which has been utilized very successfully in the robot domain for coordinating the real-time perception and movement tasks.

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 infers new knowledge and actions from the existing knowledge. Of course, this also means that deliberative agents are dependent on the efficiency and speed of the inference mechanism and possibly cannot react to urgent events as fast as needed. A well-known deliberative agent architecture is IRMA (Intelligent Resource-bounded Machine Architecture) (Bratman et al. 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.

As 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 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. Direct communication between tasks is generally avoided in order to keep tasks reusable and mostly independent of other behavior modules.

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, i.e. 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. In case that no plan could achieve the goal it is considered as failed. This flexible event-driven processing allows PRS-agents to perform *reactive planning*, which is efficient by virtue of the predefined plans from the plan library and also very robust thanks to the built-in plan retry mechanism.

3.2 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 multi-agent 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 being able to support 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 multi-agent systems (Ferber et al. 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 arbitrary many roles in possibly different groups. Groups can freely overlap, which allows an agent being part of different groups at the same time. Group membership is based on a mechanism, which requires an agent sending a membership request to the responsible group leader and when accepted the agent takes over the responsibilities and duties of the granted role. One important property of the AGR-model is that it intentionally does not enforce a particular type of agents being used for group activities and also allows heterogeneous types of agents working together within the same group. This makes AGR independent of any particular agent architecture and allows simple agents as well as very complex agents, possibly employing the intentional stance (Dennett 1971), being part of the same organizational structure.

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, i.e. it will inform the others if it beliefs 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. Despite the neatness of the formal framework, it leaves some activities unresolved, which are needed for using it in practice. For example, no method for establishing a joint intention is proposed and also the defection of a single agent leads to the breakdown of the whole group task. For these reasons the joint intentions theory was subject to several extensions, which expanded and enhanced the basic model. Examples for such extensions are Jennings' joint responsibility theory (Jennings and Mamdani 1992) and Tambe's STEAM model (Tambe 1997).

3.3 Summary & Challenges

In multi-agent systems it can be distinguished between internal and social agent architectures. The former can be seen as a blueprint for building agents, whereas the latter is used for structuring and coordinating whole agent groups. In the area of internal agent architectures a lot of different approaches exist, ranging from simple reactive to highly complex cognitive proposals. Despite the many options, in practice especially the task model and PRS-based BDI

architectures have been widely used and proved their usefulness in many application domains. With respect to social architectures the complexity is even higher and current approaches try to reduce this complexity by addressing exclusively the structure or the behavior dimension of teamwork. Hence, future challenges include the development of holistic team architectures incorporating both dimensions adequately and additionally some form of integration between internal and social architectures.

4 Development Methodologies

Developing an agent-based software requires, as for 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. A software engineering methodology aims to describe all the elements necessary for the development of a software system. So, a methodology uses a modeling language to capture and describe requirements of the system-to-be, a modeling language to describe architectural components and details of their interaction, various forms of analysis to reason about models, a structured process to guide analysts and developers activities, and tools to support and semi-automate the developing process.

Similar to the growing availability of Object-Oriented (OO) systems development methodologies in the nineties, we are now seeing the burgeoning of a number of innovative Agent-Oriented (AO) methodologies. However, in contrast to OO methodologies, nearly all AO methodologies are not industry-driven and have been developed by the academic research community. Most AO methodologies are (at the time of writing) in an early stage, albeit many of them have been tested and evaluated at least in small, industrial applications.

Many AO methodologies use the metaphor of an human organization (possibly divided into sub-organizations) 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.

4.1 Overall Characterization

Giving an exact definition of software methodology arose a long debate in various subfield of information systems and software engineering. Traditionally (Rolland et al. 1999), work products and their documentation can be considered important components of a methodology. They result the most visible part the usage of a methodology, which is why the object-oriented modeling language UML (OMG 2007) is so frequently (totally incorrectly) equated with "all things OO" or even described as a methodology. Instead, in the AO methodologies world are adopted several notations: someone uses UML or its agent-focused counterpart AUML (Odell et al. 2000), while

others eschew this as being inadequate to support the concepts of agents introducing instead their own individualistic notation and underpinning concepts.

An AO methodology should offer sufficient abstractions to fully model and analyze society of agents – arguably, simple extensions of OO methodologies are too highly constrained by the sole focus on objects. An AO methodology needs to focus on organizations of agents, which can play different roles and interact one another accordingly to protocols determined by their roles. But, we should also ask what does it mean for a methodology to be "agent-oriented"? In the OO field, we use OO concepts to describe the methodology, which in turn can be used to build object-oriented software. Generally, when we speak of an AO methodology, we do not mean a methodology that itself is founded on the agent paradigm, but rather on a methodology that is oriented towards the creation of agent-based software. AS we will discuss next, the Tropos methodology is the only exception that uses agent-oriented principles as basic concept of the methodology but it does not put any constraint on the implementation language for the actual system.

The scenario of AO methodologies is quite complex and it results very difficult to give a complete characterization of all its 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 on ideas from artificial intelligence (AI), others as direct extensions of existing OO methodologies, whilst 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 et al. 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 et al. 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 et al. 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). Secondly, 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 et al 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).

Further comparisons of these methodologies are undertaken in (Tran and Low 2005), which complements and extends earlier framework-based evaluative studies. 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) (Weiss 2002).

4.2 Selected AO Methodologies

We briefly describe in the following the most popular and used AO methodologies: *GAIA*, *Promethous* and *Tropos*.

GAIA (Zambonelli et al. 2003) is one of the first proposed agent-oriented software engineering methodology. In GAIA, it is assumed that for the development of medium and large multi-agent 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 which allow the role to perform its responsibilities. Activities are computations that can be executed by the role along, 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 defines 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 agents specific role, as well as by its current status. Identifying and modelling the environment involves 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, i.e. 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 organizations activities. That is, it implicitly defines the overall architecture of the MAS organization, i.e. 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 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 variant of the scenarios introduced by UML's 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 interaction 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 inter-action 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 relationship. Dynamic behaviour is described with UML and AUML diagrams. In the system specification phase, Promethous gives a strong emphasis to the determination of system's goals and functionalities. The determination of goal results in an iterative process: identifying and refining system goals, grouping goals into functionalities, describing 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 system's functionalities. A role deals with a single aspect or subgoal of the system and it has to be very specific avoiding thus to have too general functionalities that can drives to potential misunderstanding. The definition of functionality provides also 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 diagram 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 during the architectural design phase interaction diagrams and in turn 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 behaviour of the system.

UML sequence diagrams are adapted to represent interaction diagrams and are used as initial representation of agent interactions. Interaction protocols are final design artefacts.

Prometheus is tool-supported (Padgham et al 2008): Prometheus Design Tool (PDT) and JACK Development Environment (JDE). PDT allows users to create and elaborate Prometheus design. Particularly, PDT helps in avoiding the introduction of inconsistencies and it provides cross checking that detects other forms of inconsistency. It is also used to export individual diagrams and generate documentation of the overall design. 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 requirements-driven in the sense that it is based on concepts used during early requirements analysis. Tropos adopts the concepts offered by *i** (Yu E. 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: (i) early requirements, (ii) late requirements, (iii) architectural design, (iv) detailed design. Although, there are many proposals to integrate Tropos with agent-oriented programming frameworks, originally Tropos does 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 non-functional 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 which describes all functional and non-functional requirements for the system-to-be. In Tropos, the system is represented as one or more actors which 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 fulfilments 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) 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 multi-agent 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 multi-agent system architecture to the organizational context within which the system will operate.

Detailed design introduces additional detail for each architectural component of a system. In particular, this phase determines how the goals assigned to each actor are fulfilled by agents in terms of design patterns. Design patterns have attracted much attention, but unfortunately, the literature focuses on object-oriented patterns, rather than the intentional and social ones that are relevant here. Within Tropos, social patterns are used to find a solution to a specific goal defined at the architectural level through the identification of organizational styles and relevant quality attributes. Detailed design in Tropos also includes the specification of agent communication and agent behavior. To support this task, Tropos proposes to adopt existing agent communication languages, such as FIPA-ACL, and extensions to UML, such as AUML.

Tropos supports the application of various forms of formal analysis techniques for the verification of requirements and system specifications. Goal analysis techniques are used to reason about goal models identifying possible alternative ways to satisfy actors' goals. Risk analysis is used to identify possible risks and treatments to be adopted during the execution of an agent. Security analysis techniques are used to analysis security concerns along the whole software engineering process. Tropos has also a formal specification language (Formal Tropos -- hereafter FT) that offers all the standard mentalistic notions of Tropos and supplements them with a rich temporal specification language. FT allows for the description of the dynamic aspects of Tropos models. More precisely, in FT the analyst focuses not only on the intentional elements themselves, but also on the circumstances in which they arise, and on the conditions that lead to their fulfilment. In this way, the dynamic aspects of a requirements specification are introduced at the strategic level, without requiring an operationalization of the specification. With an FT specification, one can ask questions such as: Can we construct valid operational scenarios based on the model? Is it possible to fulfil the goals of the actors? Do the dependencies represent a valid synchronization between actors?

4.3 Summary & Challenges

Current literature on agent-oriented methodologies offers a wide range of methods, modelling languages, analysis techniques and tools to support different phases of the agent-based software development process. Some of these methodologies characterized by the use of extended object-oriented techniques and methods to analyze and design an agent-based software. Other methodologies have agent-specific concepts and techniques as their key ingredients. AO

methodologies are mainly the result of academic research activities (in contrast to OO methodologies which are industry-driven). While in many cases the available agent-oriented methodologies are not yet suited for broad real-world usage, it can be observed that the research focus currently is moving from basic foundational issues to more practical software engineering issues such as method integration, testing, verification and empirical studies (Gomez-Sanz and Luck 2008). One of the most important issues for agent-oriented methodologies (as for any new technology) to be accepted by industry is that it must to be fully supported by tools and standards. Many research groups already work along this direction by developing CASE tools and promoting standardization activities (e.g., IEEE FIPA initiative).

5 Tools, Platforms and Programming Languages (Frameworks)

For the development of multi-agent 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..

Basically, it is desirable having tool support in each phase of a development process, which typically comprises analysis, design, implementation, testing and deployment activities. These development phases of multi-agent systems are often guided by agent-specific methodologies, which introduce new kinds of models and hence should be supported by agent-specific tools. In addition, also tools are necessary that support crosscutting activities such as repository and project management. The crosscutting activities are quite independent of the handled artifact types. This allows existing tools to be reused also in agent projects. In the current landscape of existing mainstream object-oriented development tools two broad categories of tools can be identified: modeling tools such as well-known UML-CASE (computer aided software engineering) tools and integrated development environments (IDEs) such as eclipse. While modeling tools primarily address the design phase and aim at bridging towards the implementation phase e.g. via code generation facilities, IDEs are code-centric tools and focus mainly on the implementation phase. In many cases, IDEs offer additional support also for subsequent phases such as testing and deployment by integrating testing frameworks and automating tedious deployment tasks. In both tool categories several agent-specific solutions exist. Nonetheless, as the field of multi-agent systems exhibits a high degree of heterogeneity, tool developments are often tailored towards a specific methodology or agent platform. For this reason in the following mainly agent platforms will be described and development tools will only be considered in the context of these concrete platforms.

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. section 5). 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 can be 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 in (Braubach et al. 2006), is depicted in Figure 2. It distinguishes platforms by means of their primary focus and proposes three main categories: middleware, reasoning and social oriented platforms. The meaning of these categories will be explained in the following subsections.

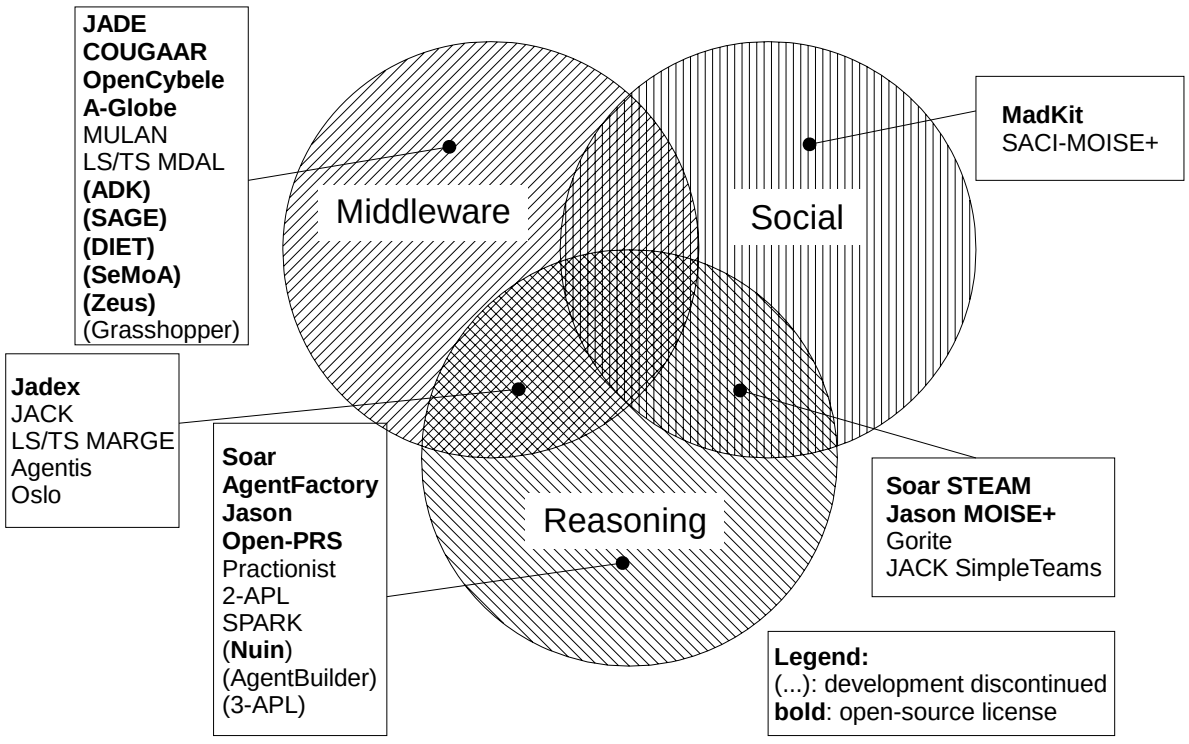

Figure 2: Classification and overview of commercial and open-source agent platforms

5.1 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 et al. 2005). Examples of such functionalities include directory services and message passing mechanisms.

In the field of multi-agent 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, also mobile agent toolkits such as Grasshopper (Bäumer et al. 1999), which allow agents to migrate between different hosts, can be seen as part of the middleware category. Most important characteristic supported by nearly all middleware platforms is the 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 rather on the *task model*, which assembles the overall behavior from simpler behavior modules. For this reason, most middleware platform 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 one 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 corresponds to a task and serves for the encapsulation of a specific functionality. An agent can be supplied with arbitrary 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 non-preemptive scheduling, i.e. 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). As 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 en- 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 multi-agent 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. 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), the stepwise execution and monitoring of agent behavior (introspector agent). For the debugging of multi-agent system the sniffer tool is quite helpful, as it visualizes the messages between agents in a style similar to UML sequence diagrams.

5.2 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 in 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 e.g. 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 is developed as open-source project at the University of Hamburg since 2003 (Pokahr et al. 2005). It follows the BDI model (Bratman 1987) and allows goal-oriented agents being 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 multi-agent 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 which 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 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, i.e. its plan bodies, can directly be programmed in plain Java. Agent-related behavior is made accessible through a framework API, which permits e.g. 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 e.g. 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 also debugging tools 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. This means that the same application can be executed as time-stepped or event-driven simulation as well as in realtime.

5.3 Social platforms

Social agent platforms underline the importance of coordination and cooperation aspects within multi-agent 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 behaviour 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 is developed as open-source by Ferber and colleagues (Gutknecht et al. 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 micro-kernel, 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 multi-agent systems and hence does not implement a specific agent architecture to be used by an agent programmer. On the one hand, this gives an agent developer the complete freedom 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 be e.g. 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 goups 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 which 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.

5.4 Summary & Challenges

Agent platforms make up a central part of tool support for the development of multi-agent applications and many different kinds of open-source as well as commercial platforms have developed until now. Thes platforms can be broadly categorized according to their main focus leading to middleware-, reasoning- and social-centric platforms (cf Figure 2). Their main characteristics have been described and with JADE, Jadex and MadKit one typical representatives for each category has been further illustrated. Besides these main categories the Figure 2 also highlights that intersections between these categories may occur, when platforms conceptually and technically address more than one aspect. This also means that having platforms offering support for all three aspects would be favorable but has not been achieved so far. Thus, the integration of the conceptual models in the three primary categories remains one of the practically important research objectives in the area of multi-agent systems. To date, the specialization of agent platforms towards one of the three main categories induces pragmatic difficulties for agent software projects, as a suitable platform for the problem at hand has to be

found. The selection of an agent platform is nowadays a non-trivial task, because the characteristics of the problem domain should fit as good as possible to the concepts offered by the agent platform (Braubach et al. 2006). This fact also makes the scientific and commercial adoption of agent platforms an important challenge.

6 Standards

In the context of multi-agent systems two standard bodies have worked on specifications for different aspects of interoperability: the Foundation for Intelligent Physical Agents (FIPA) and the Object Management Group (OMG). The OMG initiative aimed exclusively at a standard for mobile agents and produced the mobile agent system interoperability facility (MASIF) specification (OMG 2000) as result. The main objective of MASIF consists in establishing a common ground that allows MASIF compliant agent frameworks to perform agent migration even in heterogeneous environments (assuming the same platform implementation language). The way MASIF addresses this aspect is by standardizing the agent information format used for transferring data between platforms allowing a platform to understand the demands of the migrating agent. The transmitted data makes explicit the agent profile describing the language, serialization, and further agent requirements on the platform.

In contrast, the FIPA has worked on a broad range of standards with the objective of enabling interoperability between different agent platforms. The FIPA was set-up as an independent organization and joined IEEE in June 2005. The FIPA standards are middleware centered and cover all building blocks required for an abstract agent platform architecture (see Figure 3).

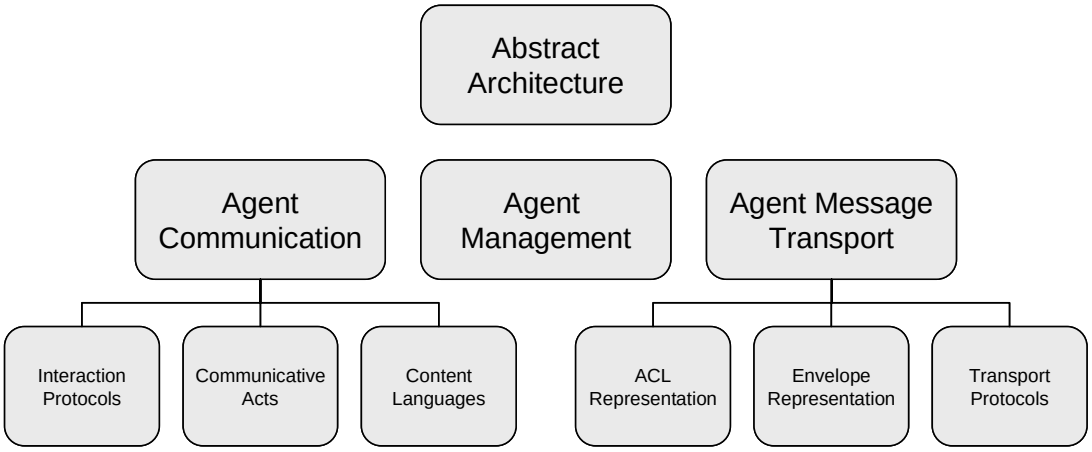

Figure 3: Overview of FIPA standards (from FIPA website)

On basis of the abstract architecture specification (FIPA0001) that describes at a high abstraction level how agents can find and communicate with each other, the agent management specification has been derived (FIPA00023). This specification is essential for understanding the platform operation and elaborates on the necessary platform components and their required interplay. As can be seen in Figure 4 an agent platform consists of three management components. The agent management system (AMS) has the task to exert supervisory control over access to and the usage of the agent platform. Furthermore, the AMS keeps a list of all agents currently hosted on the platform and can be inquired e.g. about agent addresses and lifecycle states. In addition to the

AMS, the directory facilitator (DF) is a management component responsible for administering service registrations of agents. It allows agents to register, modify and cancel service registrations. Moreover, agents can request the DF to search for suitable service providers. Besides these management components, a platform is expected to have a message transport service (MTS), which can be used to send local as well as remote messages transparently for an agent.

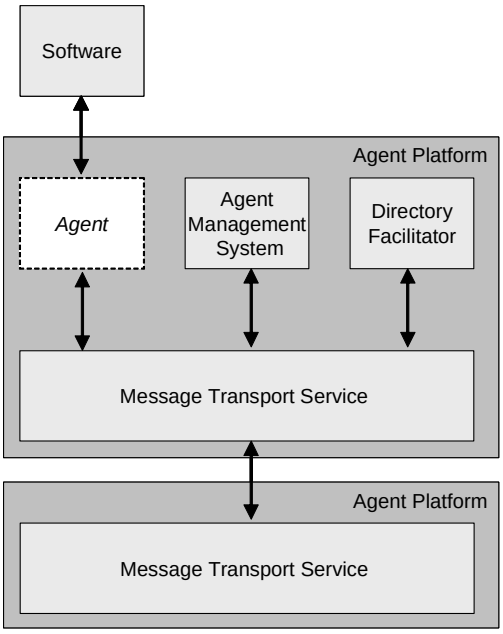

Figure 4: FIPA agent management reference model (from FIPA00023)

Interoperability between agent platforms mainly depends on standards on two levels: the agent and the platform level. From an agent's perspective most importantly the interaction protocols and the message format have been defined. Interaction protocols represent generic communication patterns between agents that can be used for recurring tasks such as specific negotiations. The FIPA has proposed interaction protocols for simple request-reply scenarios (FIPA00026) as well as more complex scenarios such as English and Dutch auctions (FIPA00031/32). The representation of messages has been defined in several interrelated standards. In addition to the basic message structure (FIPA00061), also a library of permissible communicative acts (FIPA00037) and several content languages have been developed. The message structure basically dictates the information of a message and includes the participants, the content and the interaction control. Hereby, the content of a message is handled separately from the rest of the message. This allows specific languages to be used for the en- and decoding of the message content, such as the FIPA semantic language (SL), the KQML knowledge interchange format (KIF) and the resource description framework (RDF) as specified in FIPA0008/10/11. On the platform level, the message transport has to deal with the representation of messages and their envelopes as well as with the underlying transport protocols. Besides the basic transport service (FIPA00067), especially encodings for messages and envelopes has been subject of standardization. In this respect, XML (FIPA00071/85) and a bit-efficient representation (FIPA00069/88) have been devised. Transport protocol specifications exist for the

middleware IIOP mechanism (FIPA00075), the internet HTTP protocol (FIPA00084) and additionally for mobile settings via WAP (FIPA00076).

Challenges. Even though many vendors have agreed upon the FIPA standards and interoperability between different FIPA-compliant platforms has been demonstrated also in practice e.g. in the context of the AgentCities project (Willmott et al. 2002), industry has until to date primary adopted non agent-based standards. In particular, web-service standards have gained attention for realizing loosely coupled message-oriented interoperability. In addition, in the area of backend functionalities component-based standards such as the Java Enterprise Edition (Java EE) specifications have been widely employed. Hence, a main future challenge consists in bringing together those established standards with the agent perspective and foster the evolvement of integrated and consolidated agent standards, which can be adopted by industry without giving up existing infrastructures.

7 Application Areas

7.1 Scope of Application

Intelligent agents prove 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 is 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, e.g., 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, e.g., 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, e.g., 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), whether for the purpose of simple data exchange or for knowledge-based negotiation concerning the costs of utilization of a certain resource.

7.2 Application Domains

The application areas for multi-agent 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.

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 5: Overview of multi-agent application areas

Figure 5 presents a matrix according to the two categorization dimensions sketched before. The choice of application sectors used here follows the proposal of Jennings et al. (1998) and adds the military domain. A more fine-grained breakdown of sectors can e.g. be found in Luck et al. (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 to give an impression of the possibilities of multi-agent 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).

Industrial Applications. Industrial applications of multi-agent systems can be found in the areas of production, telecommunication and transport.

An important task in the production area is the efficient control of the production process. In many cases this is a very complex job due to several continuously changing parameters such as the properties of the produced goods or the available resources. One major objective is to be able to anticipate and react in a flexible and timely manner to these changes. One example for multi-agent applications in this area include is the YAMS system (Parunak 1995).

Regarding the telecommunication area, one important task consists in the network management, which has to ensure an effective and efficient operation of the network. In order to achieve this

aim among other things resource allocations, error detections and repair actions have to be carried out. A major challenge hereby is to make the network management independent from the network's size and its distribution. Hence, using centralized solutions may hinder the continuous growth of the network. The MAGENTA (Mobile AGENT for Administration) is an example of an agent-based control system (Sahai et al. 1998).

Finally, with respect to the transport domain, the logistics represents a pressing challenge in the context of globalization. In this area, agent technology can contribute by decentralized and negotiation-based approaches, which are better suited when many unexpected events are likely to happen (e.g. traffic jams or machine breakdowns). A commercial agent-based planning system is ATN (Adaptive Transportation Networks) from Whitestein Technologies.

Commercial Applications. Industrial applications are often highly complex and specialized solutions applicable to niche markets only. In contrast, the target of commercial applications is the mass market including the end customer in many cases (Jennings et al. 1998). The commercial sector includes electronic assistants and e-business applications. Here, from the e-business area only e-commerce and business process management will be picked up.

Electronic assistants in the commercial context have the main objective to support humans in fulfilling their tasks and disburden them from tedious routine activities, e.g. the IDIoMS (Intelligent Distributed Information Management System) has been developed for helping users to search and present information based on their profiles (Soltysiak et al. 2000). In the e-commerce area, agents e.g. have been used on the client side for realizing shopping assistants, i.e. agents that pursue product finding and price comparison for a user given product in the internet. On the merchant's side the technology has e.g. been used to install so called chatter bots, which represent agents living on the merchant's website capable of interacting with users at a (written) language level.

Looking at the business process management area, challenges exist in reaching a high degree of automation of processes using workflow management systems. In this respect, agent technology has especially been used for making workflow systems more flexible in many ways, e.g. for the dynamic work item assignments and also for specification of agile business processes. Agile business processes extend the activity-driven description of workflows using BDI concepts. They emphasize a goal-based view on processes helping to separate the process reasons from its actual implementation. An agile workflow management system is currently developed by Daimler in cooperation with Whitestein Technologies (Burmeister et al. 2008).

Entertainment Applications. Within the entertainment industry agent technology is used iter alia for producing computer games and films and more or less intelligent toys.

An important criterion for the creation of computer games is the generated degree of realism and hence the plausibility of the virtual game world. This plausibility depends on different factors such as the audio-visual receptions, the physical behavior of things and also on the intelligence so called NPCs (non-player characters) exhibit. Especially, for improving the last aspect, artificial intelligence and agent techniques are considered as helpful. E.g. in the famous Black and White computer game, characters are based on a BDI based architecture making their behavior goal-based and reactive.

The production of movies can also benefit from advancements in agent technology. One application area here is the replacement of real with virtual actors. Especially, in case of cartoons and mass scenes with possibly hundreds or thousands of participants such a replacement is promising. To yield realistic scenes, not only the appearance but also the behavior of the virtual actors is of crucial importance. This behavior should reflect the goals and emotions of the character and should not be robotic. One commercial agent software developed in this area is the software Massive from Massive software (www.massivesoftware.com) which e.g. has successfully been used for creating mass scenes such as battles with hundreds of thousands of virtual actors.

Besides virtual characters inside games or movies also intelligent toys are produced. The major aim of these toys is the entertainment of their owner. They represent an artificial friend and therefore take over a similar role as pets do, but have the indisputable advantage to be disengageable, if the situation requires this. On the one hand, these toys can employ agent technology and on the other hand agent technology can use these toys as situated embodied agents e.g. for testing agent architectures. The capabilities of situated embodied agents are tested e.g. within the Robocup tournament, in which robot teams play football against each other.

Medical Applications. The health care sector is characterized by an open environment consisting of a multitude of different distributed stakeholders like hospitals, doctors, pharmacies, health insurance funds and many others. This distribution requires facilities for efficient information exchange and coordination between the involved stakeholders. Agent-based approaches have been developed for resolving medical as well as the administration problems.

Medical support can be given in various computer-supported ways by the different stakeholders of the health care sectors. Examples are computer supervised training for medics, diagnosis support systems using expert knowledge, support of telemedicine and simulations of biological processes. This diversity is also reflected by many different agent approaches tackling the aforementioned problems. Here, only the TeleCARE project will be shortly presented (Camarinha-Matos et al. 2004). The project has developed a service platform on basis of multi-agent technology for the interconnection of elderly people with medical suppliers such as hospitals. The platform integrates different services such as a health monitor, entertainment and social offerings and spans a network of devices such as TV, computer, cell phone etc. The project contributes to the quality of life of elderly people and may allow them to stay living at home, even if they have health problems.

Military Applications. The military domain is characterized by an environment in which unexpected destructive actions can happen at any point in time and therefore a high degree of uncertainty is present (called fog of war). In addition, military environments are often inherently heterogeneous and distributed, which may blur the border between friend and foe. For these kinds of environments it is of central importance to be able to react fast and flexible to changes, exploit short-term opportunities and expose a robust structure that can cope with partial breakdowns while preserving the capacity to act (Beautement et al. 2005). Two interesting application domains for agent technology are decision support systems and military simulation systems.

Military simulations are used for battlefield simulations as well as for training of military personnel. In the context of battlefield simulations various aspects have been investigated. One

example is the coordination of unmanned surface vehicles (USVs) in order to efficiently perform marine surveillance tasks (Cioppa et al. 2004). With respect to training software, e.g. flight simulators have been enhanced with agent logic to adequately represent the intelligence of other military units such as enemy fighters within the simulations. For example TacAir-Soar (Laird et al. 1994) is a training software for pilots allowing them to participate in complex and realistic missions.

Support systems have been built for tactical (short-term) as well as strategical (long-term) decision making. Tactical support systems are e.g. developed to supply soldiers with additional information during the battle. An example is the Eyekon system (Hicks 2002), which provides soldiers with augmented reality devices visualizing team knowledge for all participants, e.g. where teammates are located. Strategical support systems have e.g. been conceived for handling tasks in the logistics area. For instance, the DARPA founded UltraLog project aimed to control the complete military logistics chain including demand determination, ordering, replenishment and transport. The system has the task to generate very complex logistics plans and monitor their progress. In case of unexpected occurrences the system replans automatically (Adali et al. 2003).

7.3 Summary & Challenges

Intelligent agents have a broad range of possible applications and meanwhile many companies such as Siemens, IBM, Sun, Apple and Microsoft have incorporated software agents and agent technology into their products and/or projects, and there are companies who specialize in one way or another in agent-oriented software and its development. Such companies include Whitestein Technologies (http://www.whitestein.com/), agentscape (http://www.agentscape.de/), Agent Oriented Software Pty. Ltd., Agentis Software (http://www.agent-software.com.au/), Lost Wax (http://www.lostwax.com/) and Savannah Simulations (Savannah Simulations).

Despite this, the penetration of agent technology in mainstream software projects is still rather low. Hence, one important challenge consists in making agent technology accessible by providing commercial-off-the-shelf solutions exhibiting industrial strength characteristics. The burden to use agent software also exists, because agent technology is still too much different from exiting approaches such as object-orientation or component technology. An integration with these established mainstream approaches could further enhance the acceptability of multi-agent systems and foster their usage.

8 Conclusion

Over the past decade considerable progress has been achieved in the field of agent and multi-agent 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 world-wide research on intelligent agents aims at putting this potential into practice.

This article concentrated on several aspects of intelligent agents which are of particular and direct relevance to broad industrial acceptance and dissemination. Other facets which are also essential to computational agency but are not covered in this article 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; Wooldridge 2002).

9 Literature

Adali, S. ; Pigaty, L. 2003. The DARPA Advanced Logistics Project. In: Annals of Mathematics and Artificial Intelligence 37, No. 4, pp. 409–452.

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

Beautement, P. ; Allsopp,D. N. ; Greaves, M. ; Goldsmith, S. ; Spires, S. ; Thompson, S. G. ; Janicke, H.: Autonomous Agents and Multi–agent Systems (AAMAS) for the Military - Issues and Challenges. In: Proceedings of International Workshop on Defence Applications of Multi-Agent Systems (DAMAS), 1–13.

Bergenti, F., Gleizes, M.-P., and Zambonelli, F. (Editors). 2004. Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook. Kluwer Academic Publishers.

Bernon, C., Gleizes, M.-P., Picard, G. and Glize, P. 2002. The ADELFE methodology for an intranet system design, *Proc. of AOIS workshop 2002.*

Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., Perini, A. 2004. Tropos: An Agent-Oriented software development Methodology, *Autononuous Agents and Multi-Agent Systems, vol 18, 203-236.*

Bratman, M. 1987. Intention, Plans, and Practical Reason. Harvard University Press.

Bratman, M., Israel, D., and Pollack, M.. 1988. Plans and Resource-Bounded Practical Reasoning. In Computational Intelligence, vol. 4, no. 4, 349–355.

Brooks, B. 1989. How To Build Complete Creatures Rather Than Isolated Cognitive Simulators. In Architectures for Intelligence, 225–239.

Burmeister, B., Arnold, M., Copaciu, F., and Rimassa, G. 2008. BDI-agents for agile goal-oriented business processes. AAMAS 2008 (Industry Track), 37-44.

Caire, G., Coulier, W., Garijo, F., Gomez, J., Pavon, J., Leal, F., Chainho, P., Kearney, P., Stark, J., Evans, R., Massonet, P. 2001. Agent oriented analysis using MESSAGE/UML, AOSE II, LNCS 2222, Springer-Verlag.

Camarinha-Matos, L. M ; and Afsarmanesh, H. 2004. A multi-agent based infrastructure to support virtual communities in elderly care. Int. Journal of Networking and Virtual Organisations, vol. 2, pp. 246–266.

Calisti, M., Funk, P., Biellman, S., and Bugnon, T. 2004. A Multi-Agent System for Organ Transplant Management. In: Applications of Software Agent Technology in the Health Care Domain, Springer-Verlag, Heidelberg.

Cioppa, T. M. ; Lucas, T. W. ; Sanchez, S. M.. 2004. Military Applications of Agent-Based Simulations. In: Winter Simulation Conference (WSC 2004), pp. 171–182.

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

Coleman, D., Arnold, P., Bodoff, S., Dollin, C. and Gilchrist, H. 1994. Object-Oriented Development. The Fusion Method, Prentice Hall.

Collinot, A., and Drogoul, A. 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, B. Henderson-Sellers and P. Giorgini (Editors). Idea Group Inc*., Hershey, PA, USA.

Coulouris, G.F., Dollimore, J., and Kindberg, T. 2005. Distributed Systems. Addison-Wesley.

Debenham, J., and Henderson-Sellers, B. 2003. Designing agent-based process systems - extending the OPEN Process Framework, Intelligent Agent Software Engineering, Idea Group Publishing.

Dennett, D. 1971. Intentional Systems. Journal of Philosophy, no. 68, 87–106.

Ferber, J. 1999. Multi-Agents Systems - An Introduction to Distributed Artificial Intelligence. Addison-Wesley.

Ferber, J., Gutknecht, O., and Michel, F. 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, 2003, 214–230.

Garcia-Ojeda, J.C., DeLoach, S.A., Oyenan, W. and Valenzuela, J. 2007. O-MaSE: A Customizable Approach to Developing Multiagent Development Processes. Proceedings of the 8th International Workshop on Agent Oriented Software Engineering.

Gómez-Sanz J,J; Fuentes-Fernández R.; Pavón J.; García-Magariño I. 2008. INGENIAS Development Kit: a visual multi-agent system development environment, The Seventh International Conference on Autonomous Agents and Multiagent Systems, pp. 1675--1676.

Gutknecht, O., Ferber, J., and Michel, F. 2001. Integrating Tools and Infrastructures for Generic Multi-Agent Systems. In Proceedings of the Fifth International Conference on Autonomous Agents, 441-448.

Gomez-Sanz, J.; and Luck, M. 2008: Proceeding of the 9th International Workshop on Agent-Oriented Software Engineering (AOSE 08).

Henderson-Sellers, B., and Giorgini, P. (Editors).2005. Agent-Oriented Methodologies, Idea Group Publishing.

Hicks, J. ; Flanagan, R. ; and Petrov, P. 2002. Eyekon: Distributed Augmented Reality for Soldier Teams. In: Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop, pp. 156–163.

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

Iglesias, C.A., Garijo, M., Gonzalez, J.C., and Velasco, J.R. 1998. Analysis and design of multi-agent systems using MAS-CommonKADS. In: Intelligent Agents IV: Agent Theories, Architectures, and Languages, LNAI Volume 1365, Springer-Verlag.

Jennings, N.R., and Wooldridge, M.J. 1998. Agent Technology - Foundations,Applications and Markets. Springer.

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

Jennings, N.R., and Mamdani, E.H. 1992. Using Joint Responsibility to Coordinate Collaborative Problem Solving in Dynamic Environments. In Proceedings of the 10th National Conference on Artificial Intelligence (AAAI 1992), 269-275.

Kendall, E.A., Malkoun, M.T., and Jiang, C. 1996. A methodology for developing agent based systems for enterprise integration, in Modelling and Methodologies for Enterprise Integration, Chapman and Hall.

Klügel, F. 2004. Applications of Software Agents. Künstliche Intelligenz, Band 2/04 (Schwerpunktheft zu Anwendungen von Softwareagenten, herausgegeben von A. Heinzl und F. Rothlauf, pp. 5–10.

Kruchten, Ph. 1999. The Rational Unified Process. An Introduction, Addison-Wesley, Reading, MA, USA.

Laird, J. E., Jones, O.M., Nielsen, P.E. 1994. Coordinated Behavior of Computer Generated Forces in TacAir-Soar. In: Proceedings of the 4th Conference on Computer Generated Forces and Behavioral Representation.

Luck, M., Ashri, R., and D'Inverno, M. (Editors). 2004. Agent-Based Software Development. Artech House.

Luck, M., McBurney, P., Shehory, O., and Willmott, S. 2005. Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing), AgentLink.

Lind, J.. 1999. Iterative Software Engineering for Multiagent Systems. *The MASSIVE Method, LNAI 1994*, Springer-Verlag.

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

Morandini, M., Nguyen, D.C., Perini, A., and Susi, A. 2008. Tool-Supported Development with Tropos: The Conference Management System Case Study. In the proceedings of 8th International Workshop on AGENT ORIENTED SOFTWARE ENGINEERING (AOSE 07) Revised Selected Papers. LNCS 4951 Springer.

Newell, A., and Simon, H.A. 1976. Computer Science as Empirical Enquiry. In Communications of the ACM, vol. 19, S. 113–126.

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

Odell, J., Van Dyke Parunak, H. and Bauer, B. 2000. Extending UML for agents, in Proc. AOIS Workshop.

OMG. 2007. OMG Unified Modeling Language Specification, Version 2.1.2.

Padgham, L., and Winikoff, M. 2004. Developing Intelligent Agent Systems: A Practical Guide. John Wiley and Sons.

Padgham, L., and Thangarajah, J., and Winikoff, M. 2008. The Prometheus Design Tool - A Conference Management System Case Study. . In: Proceedings of 8th International Workshop on Agent Oriented Software Engineering, LNCS 4951 Springer.

Parunak, V. 1995. Manufacturing Experience with the Contract Net. In: Huhns, M. (Eds.): Proceedings of the 1995 Distributed Artificial Intelligence Workshop, Pitman Publishing, pp. 67–91.

Parunak, V. 2000. A Practitioners' Review of Industrial Agent Applications. Autonomous Agents and Multi-Agent Systems, 3(4), pp. 389–407.

Paulussen, T.O., Zöller, A., Rothlauf, F., Heinzl, A., Braubach, L., Pokahr, A., and Lamersdorf, W. 2006. Agent-Based Patient Scheduling in Hospitals. In: S. Kirn, O. Herzog, O., P. Lockemann, O. Spaniol (Editors): Multiagent Engineering. Theory and Applications in Enterprises, Springer, 255–275.

Pavón, J., Gomez-Sanz, J. and Fuentes, R. 2005. The INGENIAS methodology and tools, Chapter 4, *Agent-Oriented Methodologies*,Idea Group, Hershey

Pokahr, A., Braubach, L., and Lamersdorf, W. 2005. Jadex: A BDI Reasoning Engine. In: R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni: Multi-Agent Programming: Languages, Platforms and Applications, Springer, 149–174.

Rolland, C., Prakash, N. and Benjamen, A. 1999. A multi-model view of process modelling, *Requirements Eng. J.*, 4(4).

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

Sahai, A., and Morin, C. 1998. Enabling a Mobile Network Manager through Mobile Agents. In: Rothermel, K. (Ed.) ; Hohl, Fritz (Ed.): In Proceedings of the 2nd International Workshop on Mobile Agents (MA 1998), Springer, 249–260

Soltysiak, S., Ohtani, T., Thint, M., and Takada, Y. 2000. An Agent-Based Intelligent Distributed InformationManagement System for Internet Resources. In: Proceedings of 10th International Conference INET.

Tambe, M. 1997. Towards Flexible Teamwork. In Journal of Artificial Intelligence Research vol. 7, pp. 83–124.

Taveter, K., and Wagner, G. 2005. Towards radical agent-oriented software engineering processes based on AOR modelling, Chapter 10 in *Agent-Oriented Methodologies*, Idea Group, Hershey, PA, USA.

Tran, Q., and Low, G. 2005. Comparison of ten agent-oriented methodologies. Chapter 12 in *Agent-Oriented Methodologies*, Idea Group, Hershey, PA, USA

Wagner, G. 2003. The Agent-Object Relationship metamodel: towards a unified view of state and behaviour, *Inf. Systems*, vol. 28, no. 5.

Willmott, S., Calisti, M., and Rollon, E. 2002. Challenges in Large-Scale Open Agent Mediated Economies, In: Proceedings of AAMAS '02: Revised Papers from the Workshop on Agent Mediated Electronic Commerce on Agent-Mediated Electronic Commerce IV, Designing Mechanisms and Systems, Springer, Berlin Heidelberg New York.

Weiss, G. (Editor). 1999. Multiagent Systems. MIT Press.

Weiss, G. 2002.. Agent Orientation in Software Engineering. Knowledge Engineering Review, 16(4), pp. 349–373.

Wooldridge, M. 2002. Introduction to Multiagent Systems. Wiley.

Wooldridge, M., and Jennings, N. 1994. Agent Theories, Architectures, and Languages: A Survey. In Proceedings of the International Workshop on Agent Theories, Architectures & Languages (ECAI'94). Springer.

Yu, E. 1995. Modelling Strategic Relationships for Process Reengineering, *PhD, University of Toronto, Department of Computer Science.*

Zambonelli, F., Jennings, N., and Wooldridge M. 2003. Developing Multiagent Systems: the Gaia Methodology. *ACM Transactions on Software Engineering and Methodology*, Vol. 12, No. 3.

Web pointers to further important resources:

AAMAS conference series, http://www.aamas-conference.org/ .

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

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

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

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 which an agent uses in order to decide on his actions

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)

agent communication language: a formal language that allows agents to exchange knowledge and to interact sophistically at the knowledge level

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

multi-agent system: system composed of at least two agents; often used synonymous to agent system

intelligent agent, computational agent, autonomous agent software agent → agent