Chapter 1

The Jadex Project: Programming Model

Alexander Pokahr, Lars Braubach, and Kai Jander

Abstract

This chapter describes the priciples of the Jadex programming model. The programming model can be considered on two levels. The intra-agent level deals with programming concepts for single agents and the inter-agent level deals with interactions between agents. Regarding the first, the Jadex belief-desire-intention (BDI) model will be presented, which has been developed for agents based on XML and Java encompassing the full BDI reasoning cycle with goal deliberation and means-end reasoning. The success of the BDI model in general also led to the development goal based workflow descriptions, which are converted to traditional BDI agents and can thus be executed in the same infrastructure. Regarding the latter, the Jadex active components approach will be introduced. This programming model facilitates the interactions between agents with services and also provides a common back box view for agents that allows different agent types, being it BDI or simple reactive architectures, being used in the same application.

1.1 Introduction

This chapter is one of two chapters describing practical applications built with the Jadex agent framework. The applications are structured according to the main features of Jadex that were required for building these applications. In this chapter, the focus is on features regarding the programming model of Jadex. Therefore, this chapter is subdivided into three thematic sections that cover different programming model aspects and applications. Each section starts with a short background about why a certain topic was considered

Distributed Systems and Information Systems
Computer Science Department, University of Hamburg
{pokahr | braubach | jander}@informatik.uni-hamburg.de

important for programming in Jadex, followed by a more general motivation about the relevance of the concept itself. A related work section is presented for each concept, trying to give an overview of the field with pointers to other relevant works in the area. Afterwards the approach as implemented in Jadex is covered in detail and further illustrated by example applications that have been built. Each section closes with a short summary.

In particular, the following topics are described in this chapter. Section 1.2 discusses the behavior model of agents which, in Jadex, was initially realized according to the belief-desire-intention (BDI) model that was extended for Jadex in several substantial ways. With workflows, Section 1.3 addresses an interesting application area for agents regarding the support of e.g. complex and dynamic business processes. The last topic in Section 1.4 is called active components and introduces a unification of agent concepts with concepts from service- and component-based software engineering. Finally, Section 1.5 summarizes the chapter and identifies important challenges with respect to the programming model that remain to be tackled for promoting industrial take-up of agent technology.

1.2 Agent Programming: BDI Architecture

The ever increasing computational power causes an ever increasing complexity of software systems. The tasks performed by computer systems become more and more advanced including e.g. automating complex processes or providing intelligent support for humans during their execution of activities. Engineering science strives to develop new concepts, methods and tools for dealing with the increasing complexity of systems. All systems are ultimately built by humans for humans. Therefore, ideas from disciplines like philosophy or psychology have been applied to engineering for better supporting the process of comprehension of typical human system engineers and human system users. One well known example is the so called *Intentional Stance* coined by Daniel Dennett [23]. When applied to software systems, it allows considering system components as intentional entities that have certain responsibilities with respect to local and overall system goals and that act rationally and independently of each other towards achieving these goals. This approach fits well to the way how humans conceive their own thinking processes (a.k.a. folk psychology) and thus simplifies reasoning and discussing about system designs.

Intentional approaches haven proven useful early on, for example with respect to goal-driven requirements engineering [21]. When considering more and more complex systems, where typically autonomous and/or adaptive behavior is required from the system's components, it becomes apparent that intentional notions such as goals and rational action are useful also for improving system design and implementation. An intentional approach simplifies tracing requirements to design and implementation artifacts, as each are based on the same mental model of responsibilities, system goals and rational action. As an additional advantage, systems start to "behave like humans would do", i.e. they behave understandably according to the mental models of system designers and system users. This further simplifies, e.g. debugging of the system and leads to an intuitive usage.

1.2.1 Related Work

The term *agent architecture* is used to describe the concepts and constructs for specifying behavior. In this respect between internal and social agent architectures is distinguished. The first refers to architectures that deal with concepts for programming a single agent while the latter are concerned with how group behavior and teamwork can be described and programmed. With regard to different application contexts, simple or complex agent architectures may be better suited. Figure 1.1 shows an overview of well-known agent architectures. The figure highlights how the architectures are influenced by theories from different disciplines, such as philosophy and psychology. E.g. the agent architectures AOP [47], 3-APL [22], IRMA [5] and PRS [44] incorporate the Intentional Stance and are therefore related to philosophical theories like the belief-desire-intention (BDI) model. Theories from the field of psychology focus on lower-level cognitive processes such as learning and have led to architectures like SOAR [30] that largely differ from those that originate from philosophical theories. For social architectures that focus on coordination in multi-agent systems, organization theory and sociology have been sources of inspiration, e.g. the Joint Intentions theory [19] as incorporated in the Joint Responsibility model [27]. Finally, the Subsumption architecture [13] is a biologically inspired architecture for building simple reactive insect-like agents.

The BDI model [4] is a good trade-off between complexity and expressiveness as it is based on a simple set of intuitive concepts with a natural meaning (e.g. beliefs representing the knowledge of an agent about the world). The first implemented system based on a BDI-like model was the procedural reasoning system PRS [24]. The mapping to BDI was later made explicit and formalized in [44]. A number of successor systems have transported original PRS ideas to newer runtime infrastructures, e.g. the Java-based JAM [25] and the commercial JACK [17]. In addition, with AgentSpeak(L) a BDI-style programming language has been proposed in [43], which is supported by interpreters such as Jason [3].

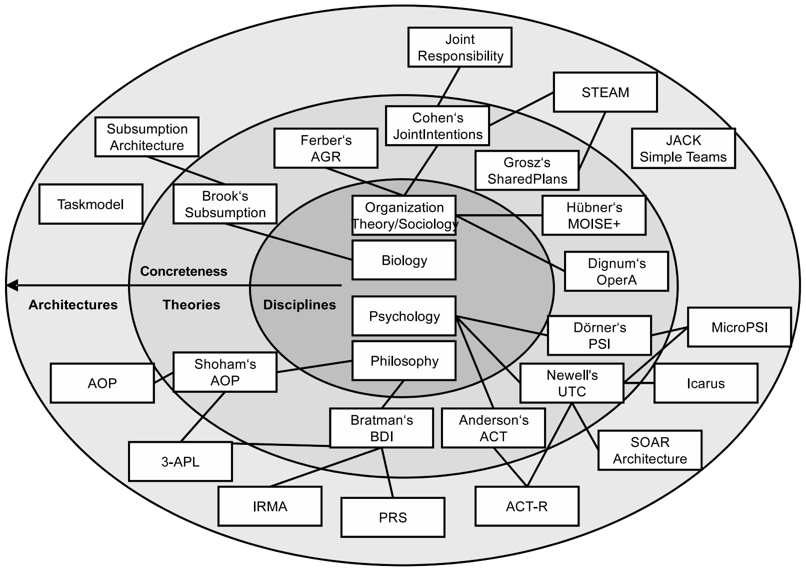

Fig. 1.1 Agent architectures (from [12])

1.2.2 Approach

The Jadex BDI architecture has been conceived and realized with conceptual as well as technical goals in mind. Conceptual aim was developing an agent behavior model that intuitively resembles human decision making. This model should act as a blueprint (pattern) for commonly found problems in agent systems. The Jadex BDI agent architecture thus provides ready to use functionality and reduces the need for manually coding aspects of the agent behavior.

On a technical level the idea is making agents more close to mainstream programming. Therefore, the realization makes use of established technologies like Java and XML. This facilitates the integration with existing technologies, 3rd-party libraries and legacy systems and further allows developing agent applications using existing development environments.

1.2.2.1 Goal Representation and Processing

The Jadex BDI architecture comprises several aspects of agent behavior and development support. In the following, the basic goal-based behavior model will be described. Put simply, it allows defining agent behavior in terms of goals to be achieved and plans to be executed towards achieving the intended goals. The behavior model is based on the means-end reasoning process found in earlier PRS systems. These realize a reactive planning approach as follows:

Given a goal or event, the agent will choose a plan from a library of procedural plans and execute the plan in a step-by-step fashion. Each plan specification incorporates one or more triggering events, i.e. goals for which the plan may be applicable. If the plan succeeds (i.e. completes without error), the goal is considered achieved. Otherwise the agent may choose another plan from the plan library and start over. The PRS reasoning cycle is well-suited for realizing adaptive behavior as plans are selected based on their applicability to the current situation. An agent can react to changing environments by simply retrying with a different plan. Furthermore, the PRS approach facilitates an extensible system design, as new plans can be added to the plan library without the need of touching other parts of the agent code.

Goal Lifecycle

In PRS, goals are only considered as ephemeral events. Jadex extends the PRS model by introducing a lifecycle for goals that allows treating goals as first class programming concepts [11]. The goal lifecycle is depicted in Figure 1.2 in an extended state-chart notation. The rounded rectangles represent the possible lifecycle states of a goal and the arrows indicate the possible transitions between the states. A goal can be created (state *New*) as a programming construct to configure its contents before making it accessible to an agent. Once the goal is *adopted*, the agent is aware of the goal such that it may influence the agents behavior. To simplify dealing with many goals at a time, three substates of the adopted state are introduced. Only *active* goals are currently pursued following the PRS reasoning approach described above. Goals may be *suspended*, when they cannot be pursued, e.g. due to external conditions. Furthermore, goals can be *options*, when their processing is delayed, e.g. in favor of other more important goals. To stop the agent from working on a goal, a goal may be dropped, putting the goal in the *finished* state.

The transitions between goal states can be performed manually by the agent programmer (e.g. writing code in a plan to create or suspend some goals). Additionally, the goal specification can be equipped with declarative conditions to indicate situations, when state transitions should happen automatically. These are shown in the figure as note boxes. The *creation condition* leads to the creation of new goals, which are initialized with contents according to the condition (e.g. the creation condition might state to create a new goal for each new item observed by the agent) and directly adopted by the agent. The *context condition* controls in which of the substates of the adopted state a goal is in. When the context is valid, the goal becomes an option and may be activated. Otherwise, the goal is automatically suspended. In some situations it is useful to stop processing of a goal, even when it is not achieved (e.g. when a goal has become obsolete). Such situations can be declaratively specified using the *drop condition*.

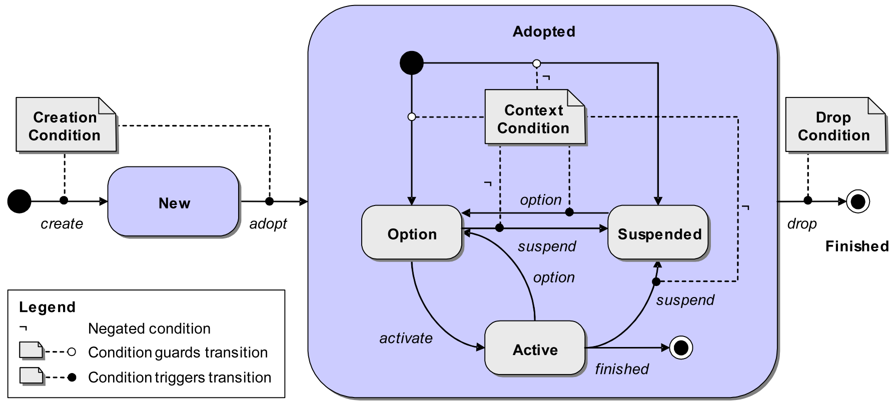

Fig. 1.2 Goal lifecycle (from [11])

Goal Kinds

The goal lifecycle as introduced above facilities the management of goals as a first class programming construct. Yet, it does not further clarify the semantics of the goal itself, i.e. how an agent should behave according to its currently active goals. Therefore, the active state is further refined in different goal kinds. In the literature, many kinds of goals can be found [11] and a common classification considers goals as a specification of a world state and an intention towards this world state (e.g. achieve, maintain, avoid, ...). Jadex supports four goal kinds, which cover a wide variety of usage patterns.

The *perform* goal is the simplest goal type and comes close to the original PRS semantics. The goal tries to execute all applicable plans, succeeding if at least one plan could be found. The *achieve* goal specifies a desired world state as a so called target condition. The goal succeeds, when the target condition is fulfilled, regardless if plans have been executed or not. Thus, the success of a perform goal only depends on the availability of plans while the success of an achieve goal is only related to the world state. Therefore, the former is often called a procedural goal, while the latter represents a declarative goal. Another common kind of declarative goal is the *maintain* goal. Unlike the achieve goal, which describes a state to be achieved only once, a maintain goal intends to keep a state after it has been achieved. Therefore, every time the state is violated plans are executed for re-achieving the state. A maintain goal is never considered succeeded and is thus only dropped, when explicitly requested by the agent programmer or the optional drop condition. The final goal kind is the *query* goal. It is similar to an achieve goal with the difference that the target condition does not represent a potentially external world state, but instead demands some information from the agent's beliefs. If the information is readily available, no plans need to be executed. Otherwise, the executed plans are expected to lead to the adoption of the required information as beliefs.

1.2.2.2 Goal Deliberation

The goal representation described in the previous section allows for dealing with multiple goals at once. Following the goal lifecycle one can influence the order in which goals are processed by moving goals between the option and active state. The mechanism of selecting goals to actively pursue is called goal deliberation strategy. While such a strategy can also be implemented manually, Jadex provides a default deliberation strategy that allows an intuitive specification and covers many recurrent application cases [41]. The so called "easy deliberation" strategy is based on two concepts: a *cardinality* to restrict the number of active goals of a given type and *inhibition arcs* to define a partial order of importance between goals.

Both concepts allow a developer to take a local perspective when writing goal specifications. The cardinality is concerned only with a single type of goal. The inhibition arc expresses a local conflict or precedence between two types of goals. It specifies that the first goal "inhibits" the second, meaning that if both are options the first may become active. Inhibition arcs can be specified on the type level or on the instance level. A type level inhibition arc means that as long as one goal of the first type is active no goal of the second type may be pursued. An instance level inhibition arc contains an expression restricting to which specific goal instances the arc applies. This allows also drawing arcs between two goals of the same type and establishing an order for goal processing based on goal properties.

1.2.2.3 Capabilities

An important concept in software engineering is modularization as it allows reducing system complexity by decomposition in software modules, which can be to some extent treated (e.g. designed, implemented, tested, . . .) in isolation. The BDI architecture as such does not support modularization with regard to a single agent. Although plans can be developed independently of each other they typically require access to global data structures like the agent's beliefs. The capability concept, initially proposed by Busetta et al. in [16], allows grouping BDI elements (e.g. beliefs, goals and plans) pertaining to a specific functionality into a separate module. The agent implementation can then be composed of existing modules. The concept has been adopted and extended for Jadex [10].

The extensions concern important software engineering aspects like parameterization, which allows external configuration of existing capabilities for making them applicable to different usage contexts, and dynamic composition, i.e. the addition and removal of capabilities during the life time of an agent. Another important extension is a generic import/export mechanism that allows establishing relationships between elements from different capabilities without violating module independence. Therefore one may specify plans that are triggered in response to goals from other capabilities and also establish inhibition arcs for goal deliberation across capabilities.

1.2.2.4 Goal-oriented Interaction protocols

The concepts that have been described until now have only considered the (intelligent) behavior of a single agent. In multi-agent systems the interaction between agents, typically based on asynchronous message exchange, also plays an important role. Therefore the question arises how the internal behavior can be linked to the external communication. As a manual approach one can send messages directly in plans. The disadvantage is that the complete code for a potentially complex negotiation needs to be placed in a single plan leading to poorly maintainable code. The concept of goal-oriented interaction protocols, proposed in [6], allows capturing agent intentions pertaining to interactions. The concept allows making use of deliberation and goal/plan decompositions for interactions as well.

The general approach defines a process for analyzing an interaction protocol, which describes the allowed sequences of messages, and attaching goals to each role in the interaction. Based on such an interaction specification, the developer can simply define separate plans for the activities and decisions required during an interaction. Besides the general approach, several ready-to-use goal oriented interaction specifications are included in Jadex that implement standardized interaction patterns like Dutch or English auction and contract-net negotiations.

Figure 1.2.2.3 shows the result of the protocol analysis for the contract-net protocol. The left hand side represents the *initiator* role of the negotiation while the right hand side illustrates the behavior of each of the potentially many *participants*. The relationship between the *domain layer* (i.e. business logic) and *protocol layer* (i.e. exchanged messages) is captured in a number of goals, which may be posted or handled at each role. The domain layer of the initiator role starts the interaction by creating the *achieve cnp_initiate* goal. During the negotiation, the *query cnp_evaluate_proposals* goal is created by the initiator's protocol layer and needs to be handled in the domain layer. When the negotiation ends, the result is made available as success or failure of the *cnp_initiate* goal, such that the initiator domain layer can proceed appropriately. At the participant side all goals are created automatically in the protocol layer. The participant's domain layer handles the *query cnp_make_proposal* goal to generate an offer to be sent to the initiator. In case a participant's offer is accepted, the *achieve cnp_execute_request* goal causes the execution of the requested task in the domain layer.

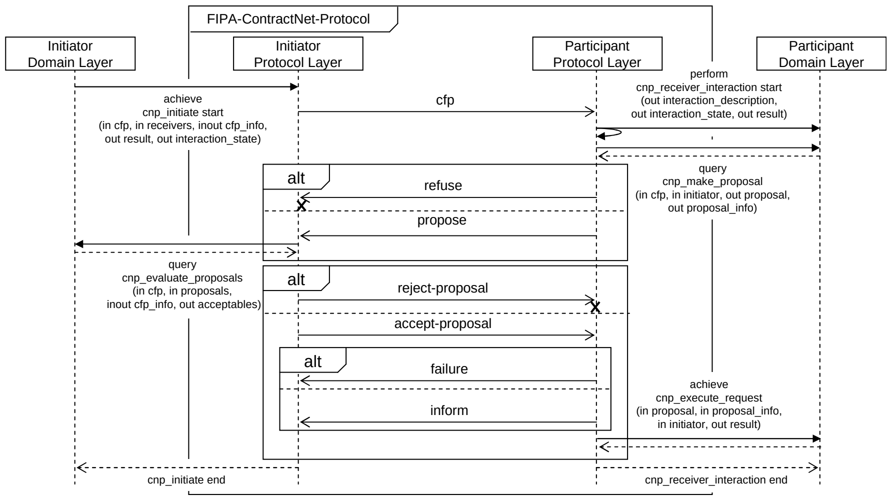
FIPA-ContractNet-Protocol
Initiator Domain Layer
Initiator Protocol Layer
Participant Protocol Layer
Participant Domain Layer
achieve cnp_initiate start (in cfp, in receivers, inout cfp_info, out result, out interaction_state)
cfp
perform cnp_receiver_interaction start (out interaction_description, out interaction_state, out result)
query cnp_make_proposal (in cfp, in initiator, out proposal, out proposal_info)
alt
refuse
propose
query cnp_evaluate_proposals (in cfp, in proposals, inout cfp_info, out acceptables)
alt
reject-proposal
accept-proposal
alt
failure
inform
achieve cnp_execute_request (in proposal, in proposal_info, in initiator, out result)
cnp_initiate end
cnp_receiver_interaction end

Fig. 1.3 Goal-oriented contract-net protocol (from [6])

1.2.3 Application: MedPAge

The described features of the Jadex BDI architecture will be illustrated with an example application called *MedPAge*, which is a real world multi-agent application that additionally makes use of capabilities for modularization and reusability as well as goals, goal-oriented interaction protocols for complex negotiations. The aim of the MedPAge ("*Medical Path Agents*") project [38, 37, 52] was improving patient scheduling in hospitals. Approach of the project was representing the different goals of the involved stakeholders by intelligent agents. E.g. patient agents would try to minimize the waiting times for their patients, whereas resource agents would try to maximize the utilization of hospital resources such as radiology units. As these goals are usually in conflict, the agents perform autonomous negotiations for producing schedules that balance the individual goals.

The project was part of a larger initiative investigating the applicability of agent technology to real world business applications. The DFG-funded[1] priority research programme SPP 1083 was conducted from 2000-2006 and involved projects from the areas of hospital logistics as well as manufacturing logistics.[2]

The hospital setting considered for the MedPAge project was derived from a real German hospital with hundreds of patients as well as several functional units with different resources. The resulting agent-based application thus exhibits much more complexity compared to the rather toy-like cleaner world

[1] Deutsche Forschungsgemeinschaft (German Research Council): `http://www.dfg.de`

[2] More details can still be found on the programme web site: `http://www.realagents.org/`

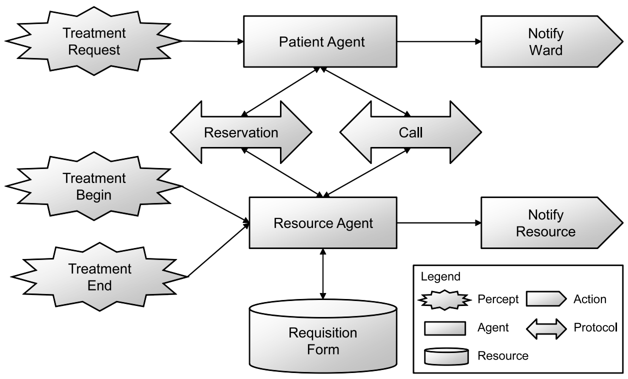

Fig. 1.4 MedPAge system overview diagram

application. Therefore, besides using goal representation and goal deliberation for defining the behavior of the individual agents, also capabilities and goal-oriented interaction protocols have been employed in the implementation.

Architecture and Design

The main goal of the MedPAge system consists in generating an efficient treatment scheduling plan. Thus the main goal of performing treatments can be refined towards two subgoals for each side. With respect to the hospital side, the main objective is to achieve a high resource utilization while the patient side is interested in seeing patient needs being satisfied, e.g. having short waiting times or giving priority to patients with severe diseases. Of course, the pursuit of these system goals has to respect the fundamental medical conditions in place.

The MedPAge system has been developed following the Prometheus methodology [36]. The core of the architecture is the system overview diagram, which is depicted in Fig. 1.4. This design contains two agent types that represent patients and hospital resources respectively. This allows a natural modeling and assignment of goals to the different coordination objects (wards and patients) and also adequately reflects the decentralized structure of hospitals. The patient agent is responsible for announcing these requested treatments at a corresponding functional unit (e.g. at the x-ray unit). Furthermore, it ensures that patients visit treatment rooms and are afterwards brought back to their ward. A resource agent accepts appointment requests from patient agents and is in charge to create treatment schedule. The resource agent is notified whenever a new treatment can begin. In this case it calls the patient from the ward and also informs the resource about the

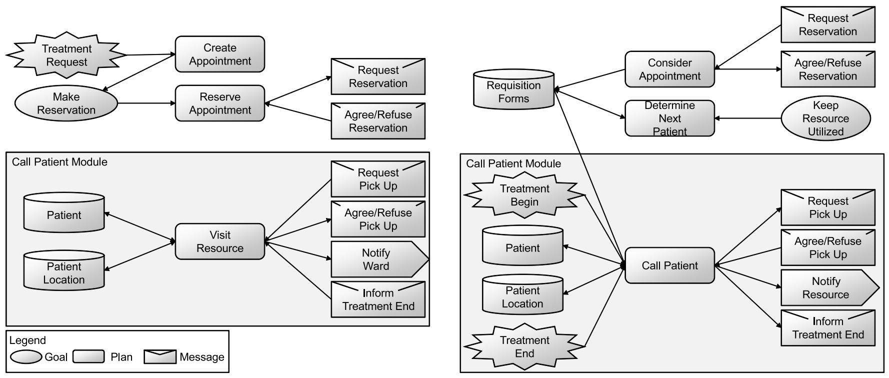

Fig. 1.5 a) patient agent b) resource agent

planned treatment and patient. After treatment end the patient is sent back to its ward.

In order to implement the MedPAge system the high-level system design has been further concretized to the patient and resource agent design shown in Fig. 1.5. These diagrams visualize the goals, plans, events, knowledge bases as well as the incoming percepts and outgoing environmental actions of the agents. The agent functionalities have been modeled as goals and plans, which can express the proactive as well as reactive agent behavior. The patient agent reacts on treatment percepts by creating a new make reservation goal for a specific appointment. The goal is handled by the reserve appointment plan, which uses a registry to find resource agents representing the functional unit it needs for the planned treatment. The set of resource agents is subsequently used to find a suitable appointment for the patient by performing negotiations that aim at respecting patient (e.g. health state) as well as resource (e.g. other appointments and utilization) needs. At resource side the new requisition form has to be taken into account and is thus added to the agent's knowledge base. The knowledge base is monitored by a keep resource utilized goal, which is used to assure a beneficial appointment ordering from the resource's point of view. Similar to the appointment reservation the patient pick up mechanism has been modeled.

The functional unit signals the readiness for a new treatment to the resource agent, which activates the call patient plan that contacts the patient agent with a pick up request. The receiving patient agent starts the visit resource plan and decides if the visit is possible (e.g. the patient could not be at the ward). The resource agent is informed about the decision. Furthermore, if the decision is positive, the ward is notified to send to patient to the functional unit and the internal beliefs of the patient location is updated. The treatment end is again announced to the resource agent. It reacts by using the call patient plan to update its beliefs and forward the information to the patient agent.

The design diagrams from Fig. 1.5 have been used to implement the application with Jadex BDI agents. The high correspondence between the Prometheus design concepts and the Jadex BDI concepts led to a straight forward implementation process that directly mimics the design.

Capabilities

Capabilities allow decomposition and reusability of agent functionality. In MedPAge, different scheduling mechanisms have been tested under realistic conditions. To keep implementation efforts low it was critical to modularize the agent designs and factor out common functionality. The primary components of the application were the patient and resource agents, which were accompanied by some support agents [39] of limited complexity. Common functionality of the patient as well as resources agents that was independent of the scheduling algorithm concerns the *call patient* module, introduced above. Regardless of how the agents negotiate the time slots for treatments and examinations, the actual calling of a patient from the ward to the corresponding resource has to be performed as a separate step allowing manual intervention of hospital personnel in case of, e.g., emergencies. Additionally, the implementation of the call patient module might differ with respect to the existing IT systems already available in the hospital.

For each tested scheduling algorithm, two capabilities have been implemented: one for the patient side and one for the resource side. Using the import/export interfaces of the capability concept, these modules can be seamlessly integrated into the agents and coupled with the remaining functionality, such as the call patient module. Each implemented scheduling approach defines a different pattern of message exchange according to an interaction protocol. The capabilities for the patient and resource agent complementarily implement either the initiator or participant role of this protocol. Details of the protocol implementations are given in the next section.

Goal-oriented Interaction Protocols

In MedPAge, scheduling mechanisms of varying complexity were implemented. The MedPaCo ("$\underline{\text{Med}}$ical $\underline{\text{Pa}}$th $\underline{\text{Co}}$ordination") algorithm incorporates stochastic knowledge about the probability of future treatments based on predefined clinical pathways as well as statistical data on previous patients with the same diagnosis. Based on this knowledge, a patient agent can estimate the value of a time slot offered by some required hospital resource. E.g. a slot would be assigned a higher value, when waiting for the next slot would significantly increase the overall staying time of the patient at the hospital. The resource agents collect estimations from multiple patients and adapt their local schedule accordingly.

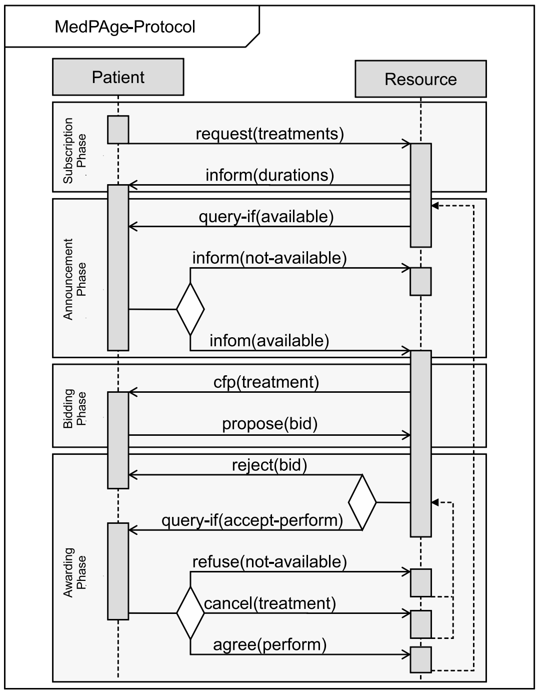

Fig. 1.6 MedPaCo3 negotiation protocol

The MedPaCo protocol is shown in Figure 1.6. The protocol is split into four phases. The first two phases involve communication the need for a time slot from the patient to the resource (*subscription phase*) and announcing the start of an auction for an upcoming time slot (*announcement phase*). The last two phases correspond to the contract-net protocol as already introduced in Section 1.2.2.4. In the *bidding phase*, the resource agent collects the bids from the patient and selects the winning patient in the *awarding phase*. At any time multiple negotiations between overlapping sets of patient and resource agents may take place. Therefore a patient might simultaneously win two negotiations at different resources. As a result, the awarding phase needs to be cyclic, because a winning patient might have accepted another time slot for the same treatment already (*cancel(treatment)*) or for a different treatment (*refuse(not-available)*).

Based on the goal-oriented interaction protocols approach, the business logic of the negotiation can be cleanly separated from the protocol specification. Important domain interaction points of the protocol are the evaluation of the time slot by the patient agent after receiving the *cfp(treatment)* message and the evaluation of the patient proposals by the resource agent to reject or accept bids.

1.2.4 Summary

The Jadex BDI architecture simplifies agent programming as it allows for intuitively decomposing agent behavior into responsibilities and abilities, which can be treated separately. Responsibilities of an agent can be obtained from a requirements analysis or an abstract system design and are described explicitly as goals (e.g. world states to be achieved or maintained). The abilities are defined as plans, i.e. procedural recipes how some goals might be pursued. The built-in goal deliberation strategy further allows intuitively controlling the order of goal processing by taking a local perspective of conflicts and precedence relations between goals. Capabilities are a modularization concept that respects all aspects of the BDI architecture and deliberation and can be used for decomposing an agent design into parts that can be independently developed. The goal-oriented interaction protocols approach connects the internal BDI concepts to message-based interaction multi-agent systems and thus allows a seamless integration of both. Ready-to-use predefined interaction protocols, such as the contract-net, further simplify the development of common interaction patterns.

One design focus of the Jadex BDI architecture was providing a means of agent programming that can be easily learned by programmers with a traditional (e.g. object-oriented) background. On the other hand, the programming model should fit well with a high-level intuitive understanding of an intelligent agent. Experiences with the Jadex framework in numerous software projects as well as teaching courses have shown that the BDI model can be easily understood and represents a natural way of thinking. Following the provided Jadex programming tutorials, students with only Java-knowledge are usually capable of developing their own agents in a short time frame.

In the MedPAge project using agent technology helped with several difficult problems. First, it perfectly mimics the decentralized nature of hospitals with wards and different functional units. The approach respects the existing autonomy of these entities and uses the agent metaphor to represent them explicitly. This allowed modeling the scheduling problem as decentralized coordination approach, in which self-interested patient and resource agents negotiate with each other to reach their goals. Using Jadex facilitated the implementation of the MedPAge system in several ways. Most noteworthy, it allowed a high level system design using Prometheus with a direct mapping to a Jadex implementation, it enabled reuse of functionalities using agent modules and it helped hiding negotiation complexities using interaction goals.

1.3 BDI in Workflows: GPMN

While a number of challenges in business process management, especially in the area of production workflows, have been addressed in various ways

[31], there remains a set of business processes with particular challenges. For example, processes like car model development cover a considerable time span, often multiple years, yet the processes themselves are dynamic. Specific practices may change while the process is in progress and unforeseen events outside the process may have an impact selecting the next set of actions in the process. Furthermore, collaborative processes like product development tend to be unstructured in terms of control flow. The control flow of such a process depends on the actions and discussions of the process participants and is difficult to predict in advance.

Faced with these challenges, it can be seen that a new approach is necessary to address a changing process environment and dynamic business processes if those processes are to be modeled as executable workflows. Since most aspects of the processes are subject to change, the question becomes which parts of the processes are actually stable and can be modeled in an executable workflow. It became clear that the only stable aspects these processes were strategic aspects like business goals. For example, during car development, the business goal of developing a new car model remains the same, even if the actual means of achieving the goal, the order in which they are achieved or the process environment like new parts or schedules may change over the years.

Thus, a goal-driven workflow modeling language would allow for the required flexibility and agility of the processes. Goals would have to be evaluated during execution and appropriate actions should be selected to further the currently active goals. Since the BDI agent model already offers a goal-centric approach, it is a good candidate for the execution of such workflows. The integration of workflow concepts in Jadex began with the DFG project "Go4Flex" [9] in cooperation with Daimler AG based on previous research conducted at Daimler Group Research regarding goal-oriented workflow concepts [15].

1.3.1 Related Work

A diverse collection of workflow languages are available both in literature and practice. Often, each language has a particular focus on either business domain-oriented modeling of business processes or the automated execution of processes as workflows. Examples for business domain-oriented approaches include languages such as Yet Another Workflow Language (YAWL [50]), Event-driven Process Chains (EPCs [45]) and BPMN. Execution-centered approaches include ECA (Event Condition Action [29]), Petri nets and the Business Process Execution Language (BPEL [34]).

This distinction is primarily one of degree and not of fundamental limitations. For example, it is certainly possible, provided the semantics are sufficiently defined, to directly execute BPMN using an interpreter and it is also possible, albeit inconvenient, to directly implement a business process in BPEL. In addition, conversion of, for example, BPMN models to BPEL workflow has become a common practice [35].

The languages can be evaluated based on how they address the five perspectives of the holistic business process view proposed by List and Korherr [32] based on earlier work of Curtis et al. [20]. The *functional view* focuses on the actions of a process, i.e. the execution of tasks. This view introduces modeling concepts such as atomic tasks and subprocesses. The *behavior view* centers around the control flow by defining the sequence of the elements of the functional view. This view is often represented using sequence edges and branching elements like XOR- or AND-splits and joins. The necessary data for tasks and data produced by tasks are represented in the *informational view*. This can include both simple information as well as complex business data structures, products and services. Organizational structures such as roles, actors and organizational units are represented in the *organizational view*. This includes the representation of work distribution and responsibilities. Finally, process meta issues and important process characteristics like strategic and operational business goals and their performance metrics in the form of key performance indicators are included in the *context perspective*.

The first four perspectives are relatively well-established and represented in workflow and business process modeling language to a varying degree. The most comprehensive approach in this regard is the ARIS house of business engineering [45]. The context perspective is a more recent addition and, as a result, tends to be less represented and connected to the other four perspectives. Most modeling languages like YAWL, BPEL and BPMN are strongly focused on the behavior and functional perspectives, featuring a limited support for the organizational and informational perspectives, often relying on external models and means to provide more comprehensive support. The context perspective generally receives little support or is completely ignored.

This situation is based both on practical consideration as well as difficulties integrating the various perspectives in a comprehensive model. The ARIS approach, which tends to be the most comprehensive, solves the problem of multiple perspectives by introducing a myriad of models to represent them. The disadvantage of this approach is the lack of integration and the risk of diverging models during both the initial development of a workflow model and later workflow reengineering.

The business goals of a process could potentially be used to integrate both the context perspective and the behavior perspective. They not only represent the reasons and motivation for the process but they would also influence the execution of a workflow model in a workflow engine depending on their specification. Before our approach, attempts have been made to integrate the context perspective using the user requirements notation (URN) in conjunction with use case maps (UCM) and the goal-oriented requirements language (GRL) [42]. However, unlike the approach presented here, this does not use goals as both functional and non-functional features and therefore does not integrate the context and behavior perspective.

Our approach is based on earlier work on the goal-context method developed at Daimler AG [15], which has also been spun off as a commercial tool [18]. However, this commercial tool uses are more straightforward processing of the goals and does not include the BDI reasoning process central to our approach.

1.3.2 Approach

Most business process modeling languages are centered on the ordering and execution of tasks. For example, BPMN uses sequence edges and gates to direct the control flow towards the appropriate task elements. In contrast, the approach presented here attempts to focus on the business reasons for the process instead of the individual actions that are required to satisfy the process. This shifts the perspective away from the question of how to solve a problem and emphasizes why action is needed and what target state is desired.

This is accomplished by introducing business goals as process modeling element. In order to model a new workflow, the workflow engineers first determine the central business goal that the process aims to accomplish. For example, in case of a car development process, this central goal can be to develop a new car model. This first goal tends to be very abstract and cannot be easily reflected with concrete tasks and actions. Therefore, the next step involves decomposing the goal into multiple *subgoals*, which, when accomplished, implicitly achieve the original goal. These subgoals then can be further broken down into more subgoals until the goals are sufficiently concrete and simple enough to accomplish them using a relatively basic and straightforward set of actions, which are then expressed as a simple BPMN workflow fragment.

This section will elaborate on the goal modeling language used to specify such goal-oriented processes and describe the technical infrastructure used to support such processes in a productive environment.

1.3.2.1 Goal-oriented Process Modeling Notation

Since current workflow languages like BPMN are task-centric, a new language or at least language elements are needed to represent functional business goals in a process. While it is technically simpler to represent goal hierarchies in a purely textual fashion like BPEL represents traditional workflow models, the goal hierarchy is supposed to represent an abstraction from the technical details and center around business functionality, which also help non-technical

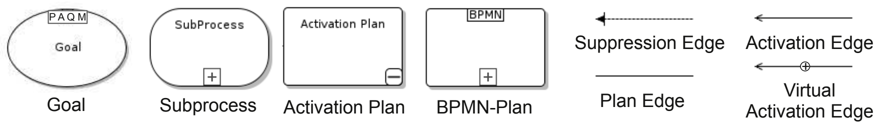

Fig. 1.7 GPMN elements (from [26])

and more business-centric people to understand the workflow models. As a result, a graphical representation for the language is desirable.

This language called the Goal-oriented Process Modeling Notation (GPMN) currently consists of four node elements and four edge elements shown in Fig. 1.7. These elements have been developed and tested within a number of both synthetic test workflows as well as real world workflows found at Daimler AG. The most obvious element is the goal element, which can represent an overall ("top level") business goal of a process or a subgoal of another goal. The language currently offers four kinds of goals that have been derived from the underlying BDI reasoning when found useful in a process. The most common kind of goal is the *achieve goal* which aims to reach a certain process state. On the other hand, if the process simply requires to perform a certain action without regard for the process state, the *perform goal* is used. The third kind of goal, the *query goal,* is used to acquire information relevant to the process. Finally, the most complex goal kind is the *maintain goal* which constantly monitors a condition and if that condition is violated, aims to re-establish a state in which the condition becomes true again.

In order to express available actions to accomplish a goal, one or more *plans* can be attached to the goals using a *plan edge*. Each plan represents an option for achieving the goal and multiple plans may be tried before a goal is achieved. Currently there are two types of plans available. The most direct way of associating tasks with a goal is to attach a *BPMN plan*. This type of plan represents a workflow fragment implemented in BPMN, specifying exactly which tasks are required to attempt to achieve the goal. However, in order to decompose goals into subgoals, the second type called *activation plan*, is needed. This type of plan is used to activate further subgoals which together achieve the plan's goal. The subgoals are defined by connecting the activation plan with the subgoals using the *activation edge*. Since the activation plan is very simple and is often unnecessary to understand the process, it is possible to hide it. The plan edge, the activation plan and the activation edges are then replaced by multiple *virtual activation edges* directly connecting the main goal and its subgoals.

Sometimes goals are in conflict with each other or can possibly interfere with each other if both are active at the same time. One way of resolving this conflict is to consider one goal to be more important and temporarily suppressing the other goal while it is active. This situation can be modeled using *suppression edges*. A goal with a suppression edge pointing to a second goal will suppress that second goal until it becomes inactive either through success or failure.

Finally, a *subprocess* element enables the workflow engineer to modularize the workflow. This is useful when the workflow is very large and the resulting model would consist of an overwhelmingly complex goal hierarchy. The subprocess element lets the workflow engineer split off part of that hierarchy and integrate it in a separate process model.

1.3.2.2 Process Context

The order of execution in the workflow is influenced by conditions based on the *process context*. The context not only contains the complete state of the workflow during execution but also reflects the environment of the workflow. This can include information such as customer information, delivery estimates, machine states and information about unusual events which have impact on the workflow.

Both goals and plans have a number of conditions whose state is influenced by the context. For example, a drop condition will, if it becomes true, cause the goal to be dropped and no longer considered while a creation condition will pick up a new goal once the condition becomes true. A number of conditions are specific to the goal kind. Achieve conditions specify the context state when an achieve goal should be considered successful. Maintain conditions on the other hand define the context state that a maintain goal aims to maintain. The context conditions of plans are used by the workflow to decide whether a particular plan is applicable under the current circumstances. For example, an achieve goal which tries to acquire transportation for an employee between two locations may have two plans, one for booking plane flights and one for train rides. However, if one of the locations lack an airport, the plan for booking flights is inadequate for achieving the goal and thus is excluded based on the context.

The process context emphasizes the context perspective and deemphasizes the behavior view by making task selection and order implicit and context-dependent instead of explicit using sequences and branches in traditional workflow languages. This allows the workflow engineer to trivially include escalation and exception handling in the workflow by adding an appropriate set of maintain goals instead of including a large number of branches and event triggers within the workflow.

1.3.2.3 Technical Implementation

A number of tools have been implemented to support GPMN workflows. Modeling and reengineering GPMN workflows is done using two editors. The GPMN editor is used to model the goal hierarchy and define the process context. The second editor is used to model the workflow fragments used for the BPMN plans. Both editors generate XML files which contain the model of the workflow.

The next step after generating the workflow models using the editors involves their execution using a workflow engine. A workflow engine creates an instance of the workflow based on the workflow model, coordinates the execution of workflow steps and manages the workflow state and context. Since GPMN workflows are inspired by BDI semantics, using a BDI agent platform like Jadex as the basis for a workflow engine was considered to be a good starting point. The models provided by the editors are first loaded and then transformed into a BDI agent model by adding additional parts needed for the agent such as the predefined activation plans.

In order to enable the BDI agent to execute the BPMN workflow fragments used for the BPMN plans, a BPMN interpreter has been developed. This editor uses a loaded BPMN model in conjunction with an internal BPMN state to interpret the BPMN elements in the model. As BPMN tends to contain some ambiguities and inconsistencies in its semantics, only a subset of BPMN elements is currently supported. The BPMN interpreter itself can also be used as an interpreter for standalone BPMN processes, enabling Jadex to execute BPMN workflows as well.

In addition, a workflow management system (WfMS) has been developed around Jadex as the workflow engine roughly based on the reference model of the Workflow Management Coalition (WfMC)[51]. This system provides addition components like user management, security and administrative features like monitoring and model deployment. This workflow management system can be accessed by client software for which an example implementation is also available.

1.3.3 Application

Goal-oriented workflow modeling has been used in a number of applications at Daimler AG. The example presented here is a partial model of a process used for preparing the production of a new car model.[3] During this process, the production of the car as well as the parts of the car are tested in a production-like environment in order to identify issues both with the car parts as well as the production process. This allows the designers of the parts and workflow engineers to address issues in their respective areas before the new car model is put into factory production.

The process shown in Figure 1.8 starts with the main “Production Preparation” goal which has to be achieved in order to reach the business goal of the process. From there, it decomposes into multiple subgoals which address

[3] The original workflow has been made abstract due to business secrecy reasons.

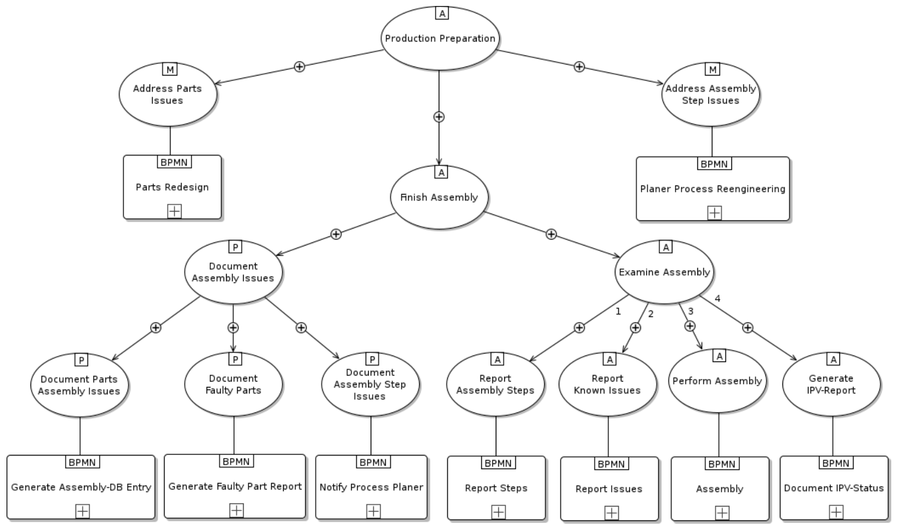

Fig. 1.8 Partial production preparation process (from [26])

three different areas. The first area is the test assembly of the vehicle itself which is a comparably regular and sequential part of the process following a predefined order. This part of the process is consolidated under the "Examine Assembly" goal.

While the assembly is in progress, issues that are identified by examining the assembly have to be documented to be addressed at a later point. This is the second area of the process which is summarized by the "Document Assembly Issues" goal. It involves the documentation of part changes that may easy the assembly, parts that are faulty to the point of not allowing proper assembly and defects in the steps of the assembly process, such as missing assembly steps or improper order of assembly steps.

Both the "Examine Assembly" and the "Document Assembly Issues" goals are part of the test assembly and as such are subgoals of the overarching "Finish Assembly" goal which controls the overall performance of the test assembly. Outside the test assembly, the issues that have been found need to be addressed in the appropriate parts or production process workshops. This is accomplished by the third area of the process which consists of the two maintain goals "Address Parts Issues" and "Address Assembly Step Issues". The maintain condition of these goals aim to keep the list of outstanding issues empty. If new issue are found during assembly, the maintain condition is violated and the associated plan is executed which then schedules a workshop to address this new issue.

Since these maintain goals stay active during the whole product preparation process, they are direct subgoals of the main "Production Preparation" goal along with the actual test assembly subgoal "Finish Assembly". This means the main goal and thus the process is not considered to be successful until the test assembly is over and all issues found during assembly have been resolved.

Most activities in the described process involve human tasks. The aim of the agent-based workflow management system is supporting human experts ("knowledge workers") in their activities by improving their coordination. The goal-oriented process description allows the agent to determine dynamically, which activities are enabled or required in response to certain events. The agent thus knows to re-enable corresponding activities automatically (e.g. scheduling a "Parts Redesign" when hen a faulty part issue was found). Using techniques such as work item lists, the knowledge workers can quickly asses the state of the process and which activities are required by them. The process state is automatically managed by the agent and updated to reflect the current situation. E.g. if a faulty part issue was found, but a change of the overall car design no longer requires the part, then the issue is automatically removed, because resolving the part issue is no longer a subgoal of the process.

1.3.4 Summary

The GPMN workflows presented demonstrate how BDI reasoning and agent-centric approaches can be used to address challenges in the area of business process management and workflow modeling. The language has been found particularly useful for processes that are either subject to a highly dynamic process context, are particularly long-running or have a low degree of structuring like collaborative and development processes. The goal-based approach lets the workflow engineer focus more on the process objectives than on the order of tasks and puts the context perspective into focus instead of modeling the workflow around the behavior perspective. The result of this additional abstraction allows the workflows to be more accessible by non-technical participants who are more focused on the business side of process management since the concept of business goals are already well known and map well onto GPMN processes.

Overall, GPMN processes offer some unique opportunities to business process management. In addition, they already have a background of being tested against real world challenges at Daimler AG that have so far been hard to address using traditional means and known workflow modeling languages.

1.4 Agents, Components and Services: Active Components

In practice only few agent-based systems have been developed and deployed in an operative setting. In contrast, other programming models such as object, component and service orientation have gained wide industrial acceptance. One could argue that agent orientation is still a very new conceptual approach and its market penetration will is still to come and will steadily increase in future. An argument that debilitates this view is the fact that service orientation is newer than agent orientation and industry interest has been much higher already since the beginnings of the adoption. The reasons for not using agent technology in practice are manifold but several obstacles can be clearly identified.

One such obstacle of particular importance is the set of programming abstractions for agent systems, which is very different from the other programming paradigms. A developer has to deal with ontologies, asynchronous message based communication, speech acts, internal and possibly social agent architectures. So the learning effort required for developers is high and existing knowledge e.g. from object orientation only partially helps to cope with these new concepts. In order to alleviate the low conceptual integration of agents the active component metaphor has been conceived. The objective consists in combining the advantages of agents with those of services and components by bringing together their main characteristics in a new conceptual entity. The resulting active components still have all characteristics of agents but extend and enhance the software technical construction means by fostering explicitly reusability, modularity and service based interactions. This does not only lead to a steeper learning curve as active components are more similar to already known approaches, it also helps using active components, hence agents, in the context of today's predominant service oriented projects.

1.4.1 Related Work

There are many approaches aiming at a combination of different software technical strands, whereby these can be distinguished by the dominating paradigm that was used as starting point for the fusion. Furthermore, the approaches can be classified according to the integration layer targeted, i.e. is a conceptual or a rather technical solution sought.

Considering agents as primary conceptual background most approaches remain oriented towards a technical integration of agents with services. Prototypical examples are the WSIG [2] and WADE projects, which are extensions of the widely used JADE agent platform [2]. WSIG is the web services integration gateway and facilitates the interaction of web services and JADE agents. On the other hand, WADE extends agents with workflows, so that agent behavior can be modeled graphically as processes.

In the area of component models several approaches exist that target distribution and concurrency and for this reason partially adopt agent or actor model ideas. An example is the Fractal framework [14], which has been advanced in the ProActive [1] project towards active objects. Similarly, in the JCoBox project [46] a component model with active object ideas has been devised. This model introduces coboxes as active entities that own passive objects and use tasks inside of coboxes for behavior execution. The model isolates objects of coboxes from other coboxes and thus adopts the typical separated actor memory model. In addition, with AmbientTalk [49] a new programming language for ambient intelligence has been proposed. The ambient communication and memory model is similar to JCoBox but its focus is on providing solutions for mobile ad-hoc networks. Furthermore, the component model of AmbientTalk is rather restricted and does not provide composition means so far. Both approaches, JCoBox and AmbientTalk, share some important conceptual ideas with active components with the main differences that they do not introduce internal component architectures for behavior definition and follow a language based instead of a framework based specification path.

It can be seen that conceptual integration of agent, component and services has been tackled partially by other existing approaches. Most close to active components are two promising strands of research. Firstly, SCA[33] successfully integrates services with components and leverages the way SOA based application can be built. Secondly, some component models like JCoBox and AmbientTalk bring together concurrency and distribution with traditional component concepts. Hence, they foster the usage of component models in dynamic application scenarios. Active component combines these efforts and further leverages the behavior specification means by introducing the internal architecture concept from agents.

1.4.2 Approach

Recently, major IT industry vendors such as IBM, Oracle and TIBCO have proposed a new software engineering approach called service component architecture (SCA) [33], which is meant to be a unification of component and service oriented architecture (SOA) concepts. The general idea of SCA consists in introducing a hierarchical component model for distributed systems. The SCA approach fosters dealing with *complexity* and *reuse*. Complexity is addressed by separating the programming model from concrete communication protocols so that these protocols are largely part of the application configuration and not of the functional program part itself. In this way SCA

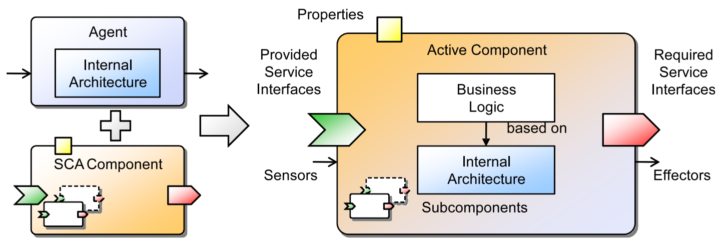

Fig. 1.9 Active component structure

shields developers from protocol details and allows building applications that communicate using a different set of protocols. Reuse is facilitated by SCA by relying on components and services as basic building blocks of software. Per definition components are considered as rather self-contained entities that exactly define what they need and offer via required and provided services. Hence, components make clear in which contexts they can be used and which functionality can be expected from them. Active components aim at combining the SCA model with agent characteristics in order to conceive a programming model that is capable to deal with scenarios that exhibit a *highly dynamic* and *concurrently acting* set of service providers. In the following subsections the structure, behavior and composition of active components are explained in detail.

1.4.2.1 Structure

Fig. 1.9 presents an overview of the synthesis of SCA and agents to active components. On the left hand side schematic views of an agent and an SCA component are depicted. In can be seen that an agent is characterized by its capability of interacting via asynchronous message passing and internally uses an internal agent architecture for encapsulating its behavior control. In contrast, an SCA component interacts with other components by relying on interconnected required and provided services. By including subcomponents higher-level functionalities can be composed of available lower-level component building blocks. Furthermore, an SCA component has clearly defined configuration points called properties, which can be used to equip it with specific startup argument values.

The merger of both approaches is shown at the right hand side of Fig. 1.9. Using a black-box perspective, an active component looks very similar to a traditional SCA component except for the small extension that an active component allows for message based interactions in the same way as agents do. The most significant enhancement of the SCA component concept results from the inclusion of the internal architecture concept as component part. This allows active components to realize autonomous behavior that goes

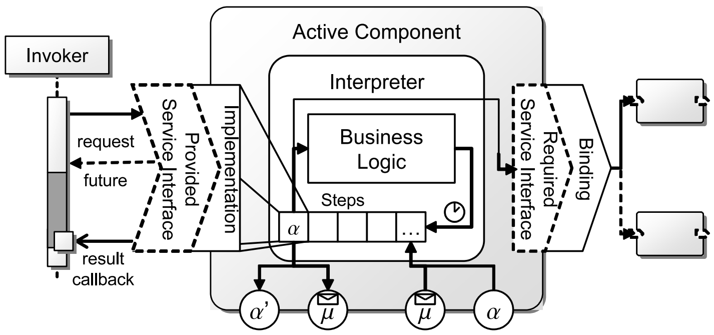

Fig. 1.10 Active component behavior

beyond its passive service functionalities. In contrast, internal architectures enable the development of reactive and also proactive component behavior that can e.g. be used for expressing workflow logic.

1.4.2.2 Behavior

In Figure 1.10 the behavior model of active components is shown. It is considerably different if compared with agents and SCA components because it has to combine a service oriented with an agent oriented perspective on how behavior can be realized. Especially, active components need to respect the most important property of agents, their autonomy, in order to be usable for constructing scenarios of components with possibly cooperative or defective intentions. With respect to active components, autonomy needs to be reflected in the way service calls are processed. No active component should be forced to execute a service call if it cannot or does not want to do it. Hence, service calls have always to be decoupled from the caller to allow the called component to reason about the service request. The only contract that is ensured by service invocations is that the caller is eventually notified about the call result, being it a value or an exception.

Technically, the decoupling is realized using futures [48], which represent place holders for the result of asynchronous processing. For each arriving service call a component immediately returns a future return value and also schedules an action representing the call in an action queue. Each component is equipped with an interpreter operating on the queue and processing the contained actions one by one. The action representing the call may optionally lead to reasoning about the call and eventually to its execution or refusal. After service processing has finished, the interpreter fills the future with the real result or exception triggering the resumption of the caller's processing.

In addition to incoming service calls a component also has to deal with outgoing service calls. These calls are targeted on another active component

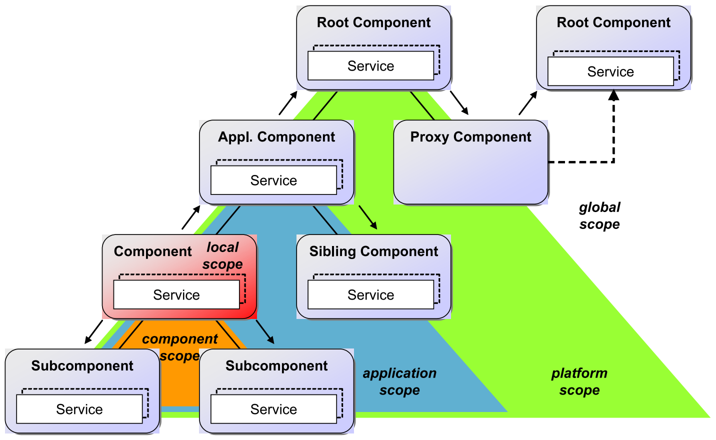

Fig. 1.11 Predefined dynamic binding scopes

and a required service binding defines how this component is found. The mechanisms for specifying and managing such bindings are part of the active component composition as described next.

1.4.2.3 Composition

The composition model of active components allows for *static* as well as *dynamic* component interconnections. Static composition means that developers use a deployment specification in order to directly wire specific component instances with each other. The advantage of this classical composition model, that is adopted by SCA and other component models, consists in the possibility of creating self-contained components that use their own subcomponents to bring about internally needed functionality on their own. Therefore, such components can be made applicable in many usage contexts by minimizing the number of required service interfaces on the component top-level. On the other hand, static wiring is not an acceptable solution for many dynamic real world application scenarios, in which service providers may appear and vanish at runtime [28].

For this reason, the active components approach supports besides a static wiring also dynamic binding based on search specifications (cf. [40]). Dynamic binding specification use search scopes for locating appropriate services in components depending on the proximity with respect to the searching component. Some predefined scopes, that proved to be useful in practice, are depicted in Fig. 1.11. They range from local scope, which only considers services of the searching component itself, over component and application scope, which extend the area towards sub- and all application components respectively towards platform and global scopes that include the whole platform and even all accessible remote sites. Further it is planned to support also the application dependent definition of user search scopes allowing developers to reflect their specific domain needs.

1.4.3 Application: JadexCloud

To illustrate the active component development approach, the JadexCloud infrastructure will be presented. JadexCloud represents a middleware for private enterprise cloud scenarios and especially highlights the advantages of the active component programming model for utility computing. JadexCloud [8] is itself a middleware, currently in development, for running applications based on the active component concept within private enterprise clouds. The general idea consists in supporting cloud application not only in homogeneous high-end data centers, but also in existing heterogeneous company computing networks, which typically consist of a mix of differently powerful and utilized machines. In such a setting cloud applications have to be designed in a very modular fashion, so that dynamic relocations of certain application parts can be performed at runtime, whenever the infrastructure or application needs change. JadexCloud makes use of active components in two ways. First, the infrastructure is built itself based on active component concepts and secondly, it supports the execution of cloud applications developed with active components.

Key concept of the proposed JadexCloud architecture is a layer model that helps separating responsibilities and managing complexity. It is composed of the following three layers: *daemon layer*, *platform layer* and *application layer*.

The daemon layer is the foundation for creating a cloud of interconnected nodes. This is done by small daemon platforms that need to run on each host that should participate in the cloud. The daemon platform includes an awareness service, which is capable of automatically detecting other platforms. The awareness service relies of different discovery mechanisms that can be used to discover new nodes. Currently, several mechanisms for detecting nodes in a local network exist relying IP broadcast and multicast schemes. Furthermore, to build up networks with hosts from different networks a relay discovery mechanism has been developed, which acts as a bridge between platforms. It is planned to further extend the relay mechanism in the direction of a redundant supernode structure known from several peer-to-peer networks. In the network a single node can always construct an actual view of available network resources. Furthermore, the daemon layer allows for basic management functionalities for application handling. Concretely, application components can be started and terminated. In order to enforce a strict separation between applications those components are started on newly started application platforms that are controlled by the daemon. Application management further requires that software bundles of applications can be accessed in specific versions, for normal startup as well as for rolling out updates of existing applications. The daemon layer handles this by utilizing software repositories that can be hierarchically organized, i.e. distinguishing local, companywide and global repositories.

On top of the daemon layer the platform layer offers a global administration view for deployment and management of applications within the cloud. The entry point for the platform layer is the so called JCC (Jadex Control Center), which offers a canon of remotable tools for setting up an application and monitoring its behavior. All nodes build up the cloud from their local perspective so that an arbitrary node with JCC can be used for application management. Based on local configuration options and user privileges, the JCC provides access to a subset of the existing nodes called the *cloud view*. The administrator can choose, which nodes to include in the deployment of an application, by assigning application components to the platforms running on the different nodes. To start the separate components, each platform will obtain the required component implementations from the repository.

The application layer, sitting on top of the platform layer, deals with how a distributed application can be built based on the active components paradigm as well as providing tools for debugging and testing applications during development. Besides the already presented general concepts of active components providing cloud ready applications need to especially consider the specification of non-functional aspects like resource needs of component instances. These aspects will be part of a deployment description for an application, which can be evaluated by the platform layer for creating a deployment plan, i.e. an ideal initial component-resource mapping, and also for component relocations at runtime. One non functional key property that has to be ensured at runtime is fault tolerance of software components. In case fault tolerance is needed for specific components they will be replicated and checkpointing will be employed to ensure that components can be restarted after a crash has occurred. Furthermore, it is planned to wrap already existing cloud services, for example for storage of data in the cloud, and make them in this way accessible for active components in a natural way.

1.4.4 Summary

This section has briefly introduced the active component concept, which unifies central component with agent ideas. The integration has been done by extending the SCA component model with internal architectures. As a result components may own not only service driven passive but also autonomous self governed behavior. Looking at active components from the outside they appear no different to traditional SCA components so that the advantages of managing complexity and reuse by hierarchical composition and abstraction from technology dependent communication means remain established.

Details of active component structure, behavior and composition have been introduced and further explained. A common objection that is put forward against active components concerns the potential loss of autonomy that is caused by using service, i.e. method based interactions. This argument is not valid as a service provider is still free to reason about performing service requests before they are actually executed. Services just introduce sound software technical foundations for typed interactions. An in depth discussion about method calls and agents is out of scope for this chapter but can be found in [7].

In JadexCloud, agent and active component technology is helpful with respect to several aspects. It defines a new programming model for cloud applications that naturally supports a louse coupling of the components and thus allows for dynamic reconfigurations of the applications according to the system demands. Furthermore, the complexity of the JadexCloud architecture became better manageable by explicit service interfaces introduced in active components. In this way the interfaces between the layers and between service requesters and providers could be cleanly software technically described.

1.5 Conclusion and Outlook

Agent technology offers intuitive concepts for describing distributed systems but implementing them is hard, time consuming and very different to other established technologies. In the following, some of the lessons learnt regarding the programming model for agent systems are summarized:

- The concepts for programming agents depend on the internal agent architecture used. In literature many different agent architectures have been proposed, often inspired from other disciplines like biology, psychology or philosophy. At the modeling and implementation layer this leads to a huge heterogeneity of approaches and requires considerable learning efforts. From a developer perspective it is advantageous to use an agent architecture that fits the concrete project requirements, especially the complexity of the agent functionalities is an indicator the choice of the right programming model. Due to this wanted flexibility agent platforms should not prescribe a specific programming model but either allow developers to choose a model among different options or provide a simple model that can be used to build custom extensions on top of it. One architecture of specific importance is the BDI model as it is a hybrid approach that combines reactive with proactive features, i.e. it is able to timely react of environmental events and is also capable cognitive behavior based on the way human rational action is explained. In Jadex both aspects have been taken into account. On the one hand, different agent

architectures are supported by the kernel concept and on the other hand a BDI kernel exists that fits many common use cases.
- Besides the agent itself, the inter agent layer plays an important role for realizing multi-agent system. It has been found that building interaction based purely on speech act based asynchronous messages is error prone and difficult as no compile time checks can be done and profound knowledge regarding the FIPA message format, ontologies and interaction protocols is required. Furthermore, agent systems are typically peer-to-peer and lack a mechanism for hierarchical decomposition, which is essential for handling complexity in large systems. Conceptually, holons fill this gap but existing frameworks do not pick up these ideas. In Jadex these challenges have been addressed by the active components metaphor, which allows service-based asynchronous interactions and allows components to have subcomponents.

The main motivation of Jadex has also been to facilitate the practical usability of agent technology. In this respect all developments described in this chapter have strived to deliver conceptual solutions that are bound to generically usable software. Respecting the problems and challenges from above, Jadex has been developed with the rationale in mind to connect agents tighter to established approaches. Concretely, with a BDI approach on basis of established programming languages like Java and XML programming of goal directed intentional agents was made easily accessible also for inexperienced agent developers. Furthermore, it has been shown how BDI agent concepts can be adopted for goal driven workflow descriptions. As goal are considered more stable than activities these kinds of workflows help to cope with frequently changing business processes. Finally, with active components a unification of agent, services and components has been introduced. Active components strongly contribute to the problem of lacking industry adoption as they are based on the standardized and industry driven SCA model. As part of ongoing work, active components are field tested in commercial applications, concretely tackling the area of business intelligence, as well as scientific mass calculations.

References

1. F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In R. Meersman, Z. Tari, and D. C. Schmidt, editors, *CoopIS/DOA/ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 1226–1242. Springer, 2003.
2. F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent systems with JADE*. John Wiley & Sons, 2007.
3. R. Bordini, Jomi F. Hübner, and R. Vieira. Jason and the Golden Fleece of Agent-Oriented Programming. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah

Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 3–37. Springer, 2005.

4. M. Bratman. *Intention, Plans, and Practical Reason.* Harvard University Press, 1987.
5. M. Bratman, D. Israel, and M. Pollack. Plans and Resource-Bounded Practical Reasoning. *Computational Intelligence*, 4(4):349–355, 1988.
6. L. Braubach and A. Pokahr. Goal-oriented interaction protocols. In *5th German conference on Multi-Agent System Technologies (MATES 2007)*. Springer, 2007.
7. L. Braubach and A. Pokahr. Method calls not considered harmful for agent interactions. *International Transactions on Systems Science and Applications (ITSSA)*, 1/2(7):51–69, 11 2011.
8. L. Braubach, A. Pokahr, and K. Jander. Jadexcloud - an infrastructure for enterprise cloud applications. In S. Ossowski F. Klügl, editor, *In Proceedings of Eighth German conference on Multi-Agent System TEchnologieS (MATES-2011)*, pages 3–15. Springer, 2011.
9. L. Braubach, A. Pokahr, K. Jander, W. Lamersdorf, and B. Burmeister. Go4flex: Goal-oriented process modelling. In *Proceedings of the 4th International Symposium on Intelligent Distributed Computing (IDC 2010)*. Springer, 2010.
10. L. Braubach, A. Pokahr, and W. Lamersdorf. Extending the Capability Concept for Flexible BDI Agent Modularization. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Proceedings of the 3rd Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS 2005)*, pages 139–155. Springer, 2006.
11. L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal Representation for BDI Agent Systems. In *Proc. of (ProMAS 2004)*, pages 44–65. Springer, 2005.
12. Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. A universal criteria catalog for evaluation of heterogeneous agent development artifacts. In *Sixth International Workshop From Agent Theory to Agent Implementation (AT2AI-6)*, 2008.
13. R. Brooks. A Robust Layered Control System For A Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):24–30, March 1986.
14. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.
15. B. Burmeister, M. Arnold, F. Copaciu, and G. Rimassa. Bdi-agents for agile goal-oriented business processes. In *AAMAS '08*, pages 37–44. IFAAMAS, 2008.
16. P. Busetta, N. Howden, R. Rönnquist, and A. Hodgson. Structuring BDI Agents in Functional Clusters. In N. R. Jennings and Y. Lespérance, editors, *Proceedings of the 6th International Workshop Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL 1999)*, pages 277–289. Springer, 2000.
17. P. Busetta, R. Rönnquist, A. Hodgson, and A. Lucas. Jack Intelligent Agents - Components for Intelligent Agents in Java. *AgentLink News*, (2):2–5, January 1999.
18. M. Calisti and D. Greenwood. Goal-oriented autonomic process modeling and execution for next generation networks. In *Proc. of MACE '08*. Springer, 2008.
19. P. R. Cohen and H. J. Levesque. Teamwork. Technical Report Technote 504, SRI International, Menlo Park, CA, March 1991.
20. B. Curtis, M. Kellner, and J. Over. Process modeling. *Com. ACM*, 35(9):75–90, 1992.
21. A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1–2):3–50, April 1993.
22. M. Dastani, B. van Riemsdijk, and J. J. Meyer. Programming Multi-Agent Systems in 3APL. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 39–67. Springer, 2005.
23. D. Dennett. Intentional systems. *Journal of Philosophy*, (68):87–106, 1971.

24. M. Georgeff and A. Lansky. A system for reasoning in dynamic domains: Fault diagnosis on the space shuttle. Technical Report Technical Note 375, Artificial Intelligence Center, SRI International, Menlo Park, California, 1986.
25. M. Huber. JAM: A BDI-Theoretic Mobile Agent Architecture. In O. Etzioni, J. Müller, and J. Bradshaw, editors, *Proceedings of the 3rd Annual Conference on Autonomous Agents (AGENTS 1999)*, pages 236–243. ACM Press, 1999.
26. K. Jander, L. Braubach, A. Pokahr, and W. Lamersdorf. Goal-oriented processes with gpmn. *International Journal on Artificial Intelligence Tools (IJAIT)*, 2011.
27. N. Jennings and E. Mamdani. Using Joint Responsibility to Coordinate Collaborative Problem Solving in Dynamic Environments. In *AAAI*, pages 269–275, 1992.
28. P. Jezek, T. Bures, and P. Hnetynka. Supporting real-life applications in hierarchical component systems. In R. Lee and N. Ishii, editors, *7th ACIS Int. Conf. on Software Engineering Research, Management and Applications (SERA 2009)*, volume 253 of *Studies in Computational Intelligence*, pages 107–118. Springer, 2009.
29. G. Knolmayer, R. Endl, and M. Pfahrer. Modeling processes and workflows by business rules. In *Business Process Management, Models, Techniques, and Empirical Studies*, pages 16–29, London, UK, 2000. Springer-Verlag.
30. J. F. Lehman, J. Laird, and P. Rosenbloom. A gentle introduction to Soar, an architecture for human cognition. In S. Sternberg and D. Scarborough, editors, *Invitation to Cognitive Science*, volume 4, pages 212–249. MIT Press, 1996.
31. F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques.* Prentice Hall PTR, 2000.
32. B. List and B. Korherr. An evaluation of conceptual business process modelling languages. In *Proc. of SAC '06*, pages 1532–1539. ACM, 2006.
33. J. Marino and M. Rowley. *Understanding SCA (Service Component Architecture).* Addison-Wesley Professional, 1st edition, 2009.
34. OASIS. *Web Services Business Process Execution Language (WSPBEL) Specification*, version 2.0 edition, 2007.
35. C. Ouyang, M. Dumas, A. ter Hofstede, and W. van der Aalst. From bpmn process models to bpel web services. In *Proc. of ICWS '06*, pages 285–292. IEEE, 2006.
36. L. Padgham and M. Winikoff. Prometheus: a methodology for developing intelligent agents. In Maria Gini, Toru Ishida, Cristiano Castelfranchi, and W. Lewis Johnson, editors, *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, pages 37–38. ACM Press, July 2002.
37. T. Paulussen, A. Zöller, F. Rothlauf, A. Heinzl, L. Braubach, A. Pokahr, and W. Lamersdorf. Agent-based patient scheduling in hospitals. In P. Lockemann O. Spaniol S. Kirn, O. Herzog, editor, *Multiagent Engineering - Theory and Applications in Enterprises*, pages 255–275. Springer, 6 2006.
38. T. O. Paulussen, N. R. Jennings, K. S. Decker, and A. Heinzl. Distributed Patient Scheduling in Hospitals. In G. Gottlob and T. Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003).* Morgan Kaufmann, 2003.
39. T. O. Paulussen, A. Zöller, A. Heinzl, A. Pokahr, L. Braubach, and W. Lamersdorf. Dynamic Patient Scheduling in Hospitals. In M. Bichler, C. Holtmann, S. Kirn, J. Müller, and C. Weinhardt, editors, *Coordination and Agent Technology in Value Networks.* GITO, Berlin, 2004.
40. A. Pokahr and L. Braubach. Active Components: A Software Paradigm for Distributed Systems. In *Proceedings of the 2011 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2011).* IEEE Computer Society, 2011.
41. A. Pokahr, L. Braubach, and W. Lamersdorf. A goal deliberation strategy for bdi agent systems. In T. Eymann, F. Klügl, W. Lamersdorf, M. Klusch, and M. Huhns, editors, *Proceedings of the 3rd German conference on Multi-Agent System TEchnologieS (MATES-2005).* Springer, 2005.

42. A. Pourshahid, D. Amyot, L. Peyton, S. Ghanavati, P. Chen, M. Weiss, and A. Forster. Business process management with the user requirements notation. *Electronic Commerce Research*, 9(4):269–316, 2009.
43. A. Rao. AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In W. Van de Velde and J. Perram, editors, *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW 1996)*, pages 42–55. Springer, 1996.
44. A. Rao and M. Georgeff. BDI Agents: from theory to practice. In V. Lesser, editor, *Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS 1995)*, pages 312–319. MIT Press, 1995.
45. A.-W. Scheer and M. Nüttgens. Aris architecture and reference models for business process management. In *Business Process Management, Models, Techniques, and Empirical Studies*. Springer, 2000.
46. J. Schäfer and A. Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. In *In Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP 2010)*, pages 275–299. Springer, 2010.
47. Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.
48. H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.
49. T. Van Cutsem, S. Mostinckx, E. G. Boix, J. Dedecker, and W. De Meuter. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. *Chilean Computer Science Society, International Conference of the*, 0:3–12, 2007.
50. W. M. P. van der Aalst and A. H. M. ter Hofstede. Yawl: yet another workflow language. *Information Systems*, 30(4):245–275, June 2005.
51. Workflow Management Coalition (WfMC). *Workflow Reference Model*, January 1995.
52. A. Zöller, L. Braubach, A. Pokahr, T. Paulussen F. Rothlauf, W. Lamersdorf, and A. Heinzl. Evaluation of a multi-agent system for hospital patient scheduling. *International Transactions on Systems Science and Applications (ITSSA)*, 1:375–380, 2006.