# Jadex Active Components: A Unified Execution Infrastructure for Agents and Workflows

Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf

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

**Summary.** Agent technology has proven to contribute rich abstractions that facilitate the construction of complex systems. Especially, in case of challenges regarding distribution and concurrency aspects, agents provide high-level solution concepts that are intuitive to understand and directly transferable from design to implementation. Despite these advantages agent technology currently has not found widespread application in industry settings. One important reason for the slow adoption is that the conceptual integration of agents with other prevalent software engineering approaches like software components or service oriented architecture is still low and does not permit the easy usage of agents in concert with other technologies. In this chapter, with the notion of active components, a new conceptual abstraction is presented that combines agent and component characteristics in order to foster the integration of both strands. The active component concept has been implemented within the Jadex Active Components middleware, which is an infrastructure that permits the execution of different types of active components like agents and workflows. The underlying architecture of this middleware is presented and illustrated by example scenarios.

**Key words:** agents, workflows, active components

## 1 Introduction

Since its beginnings in the nineties, agent technology has evolved into an active research and engineering field that provides concepts and solutions for building complex distributed systems [12, 14]. Software agents - as a design metaphor for open, distributed and concurrent systems - are commonly characterized as being *autonomous* (independent of other agents), *reactive* (advertent to changes in the environment), *proactive* (pursue their own goals), and *social* (interact with other agents) and may be realized using *mentalistic notions* (e.g. beliefs and desires)[27]. Using this design metaphor, an agent-based software application can be realized as a multi-agent system (MAS),

which is a set of agents that interact using explicit message passing, possibly following sophisticated negotiation protocols.

Current technology trends, such as increasing hardware concurrency and delegation of tasks to computer programs [14, 27], reinforce the need of conceptually rich abstractions for building complex software. While agent technology offers this kind of abstractions, the integration with existing software technology like object-orientation, service-oriented computing, workflow management systems, application servers, etc. is essential for being able to quickly build industry-quality solutions in adequate cost and time frames.

This chapter presents one approach of such an integration of agent concepts with existing software technology: The Jadex Active Components (AC) infrastructure. The basic idea of Jadex is providing a unified execution infrastructure for different kinds of entities (e.g. agents or workflows). Besides a seamless integration of these components, the unification further facilitates cross-fertilization between the different concepts. Foundation of this approach is the newly conceived notion of *active components*, which unify base concepts of software components with that of a minimal agent.

The remainder of this chapter is structured as follows. Section 2 introduces the main objectives and the overall approach of the Jadex project. In Section 3 the concepts and components for building applications with Jadex are presented. The current state of the realization is described in Section 4. Section 5 presents sample projects in which Jadex is used and Section 6 concludes the chapter with a summary and an outlook.

## 2 Design Rationale

The main goal of the Jadex project is simplifying the development of complex distributed applications. The approach towards achieving this aim is providing a middleware that aids in addressing common challenges of such distributed applications. This middleware on the one hand delivers sound conceptual metaphors for the design of distributed systems. On the other hand the implementation of these concepts is supported by a software infrastructure including reusable components and frameworks as well as development and runtime tools. Among others, the following challenges are prominently addressed by Jadex:

- Dealing with concurrency and distribution
- Realizing applications composed of heterogeneous components
- Offering versatile interaction styles
- Being confronted with a multitude of execution scenarios
- Monitoring and debugging distributed applications

The first three challenges are addressed by the abstract notion of an active component and its concrete incarnations. The active component concept is inspired by aspects of the agent metaphor for addressing concurrency and distribution on the conceptual level. As each active component is an autonomous

(i.e. independently executing) entity, the risk of concurrency-related problems, such as race-conditions or deadlocks, is reduced already during application design. Different component types are suitable for different types of applications (e.g. agents vs. workflows). Jadex addresses this heterogeneity of components by providing a unified execution infrastructure, which allows executing different components in the same application. Also, component types differ in their modes of interaction (e.g. message-based vs. method-call-based). Jadex allows all component types to make use of different interaction styles as needed. Besides synchronous and asynchronous method calls, message-based asynchronous interaction is supported by the infrastructure. Moreover, several complex interaction scenarios, such as well-known negotiation protocols, are provided in a reusable fashion for different component types.

The last two challenges are captured by the way, active components relate to their execution platform. Clearly defined interfaces between the component and the execution platform allow both being developed independently. Considering the multitude of potential execution scenarios, each imposes different challenges, e.g. a backend server solution requires scalability and transaction while deployment on a mobile device has to deal with limited resources. Due to the independence of components and platform, different execution environments can be built which specifically tackle the respective challenges, yet allow executing all available component types. Besides different execution environments, also runtime tools can make use of the common interface for all component types. This facilitates building sophisticated tools, which not only operate on any component type, but also allow observing heterogeneous applications composed of different component types in a unified way.

## 3 Design Concepts

In the following sections the central design concepts of Jadex Active Components (AC) are presented. On the infrastructure layer the notions and the distinctive characteristics of platforms and kernels is important and will be introduced first. Thereafter, a definition for an active component as base element of the architecture is given followed by an overview of currently available active component kernels.

### 3.1 Platform, Kernel and Active Component Notions

The Jadex infrastructure basically distinguishes between *platforms*, *kernels* and *active components* that are executed on this infrastructure (cf. Fig. 1).

**Definition 1.** *A platform is the management infrastructure for components, which is responsible for their execution as well as for providing administration capabilities like a messaging system or a component service registry.*

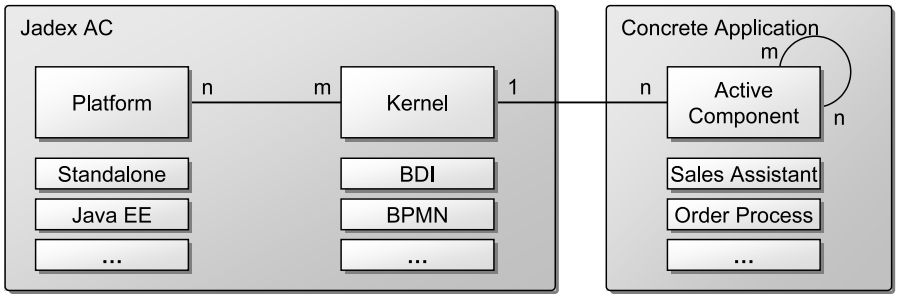


**Fig. 1.** Platforms, kernels and components

A platform itself provides all its functionalities in terms of *platform services*, i.e. it can be easily customized by changing the offered services in a configuration file. This allows including exactly those services that deem appropriate in a given setting. On mobile devices one would e.g. use resource minimal versions of platform services and reduce the number of available services whereas on a backend server with high storage and performance capacities e.g. services with persistency and transaction support may be a good fit. Public services can be accessed from active components via the platform. Active components may use services by invoking methods on object oriented service interfaces. Different platform implementations are already available that allow executing components in a *Standalone* Java application as well as on top of the well-known *JADE* agent framework [2]. A platform for executing active components in *Java EE* application servers is currently under development.

**Definition 2.** *A kernel encapsulates the internal behavior definition of a specific active component type.*

Thus a kernel realizes a specific internal architecture determining the component type. The separation of platforms and kernels allows the independent development of kernels that can then be used in conjunction with arbitrary Jadex platform implementations. A kernel thereby has complete control about the way its active components are specified and is thus responsible for loading a component and creating component instances out of this model information. Each kernel implements a distinct behavior model so that the full range from purely reactive to deliberative components can be realized. The execution of all components is kernel independent and performed by the underlying platform. It is assumed that each component is executed sequentially, i.e. true concurrency exists only between active components. Within an active component a kernel may offer *quasi-parallel* execution by interleaving the execution of active behaviors. On the one hand this has the advantage of simple active component programming without the need for concurrency language elements like locks and on the other hand it is also a common requirement of existing management infrastructures such as Java EE. The execution of active components without consideration of their concrete type is possible due to a common base concept for all variants of active components.

Basically, an active component is defined as a mixture of minimal agent and software component properties. Adapted agent characteristics are *autonomous entity behavior* (i.e. self-acting) as well as *message-based communication* means. In addition they share with software components that they are seen as at the same time as *service provider* and *consumer* and may be composed to composite components using service dependencies. Active components can be accessed via *method-calls* using provided service interfaces and are *managed* within a container infrastructure. This yields to the following definition of an active component:

**Definition 3.** *An active component is an autonomous and managed software entity that may expose publicly accessible service interfaces and is capable of interacting with other active components in different modes including message passing and method calls.*

Further details about the rationale for choosing these characteristics of active components and further explanations can be found in [19].

In Fig. 1 also the cardinalities of Jadex entities are shown. It is highlighted that on a platform any number of kernels can be executed. This allows heterogeneous applications being developed, which are composed of entities of different kernel types, e.g. a workflow based application that also employs agents for specific tasks. One kernel can also be used with arbitrary platforms thanks to the loose coupling between both concepts. It is also shown on the right hand side that an active component instance always belongs to one dedicated kernel, which takes over the aforementioned tasks regarding this component. It has to be noted that no specific application element is shown in the figure, as applications are themselves components that may include other components. This means that components are a hierarchical concept similar to holons [10], facilitating the recursive construction and decomposition of systems.

### 3.2 Available Kernel Types

In Jadex currently three kinds of kernels exist: *agent kernels*, *workflow kernels* and *other kernels*. Agent kernels are used to realize internal agent architectures, whereby kernels for belief-desire-intention (BDI) and simple reflex agents, called micro agents, exist. Workflow kernels implement process execution logic and provide a business level perspective on task execution. In this category a BPMN (business process modeling notation) kernel as well as a goal-oriented (GPMN) process kernel are available. In the third group of kernels, especially the application kernel is of relevance because it facilitates the definition of active component systems.

#### BDI Agent Kernel

In former versions of Jadex, BDI was the only component architecture available. As the way agents are described using BDI has not changed much with

regard to earlier versions, here only a short description is given (for more details refer to [5, 22]). BDI agents consist of beliefs (subjective knowledge), goals (desired outcomes) and plans (procedural code for achieving goals). Jadex BDI agents are based on the PRS (procedural reasoning system) architecture [23], which has been substantially modified and extended in previous works to support the full practical reasoning process [21, 20]. Practical reasoning has two main tasks, namely *goal deliberation* and *means-end reasoning* [26], whereby only the latter is considered in original PRS. Goal deliberation is used by the agent to determine a consistent, i.e. conflict-free goal set it can pursue at the considered moment. In Jadex the Easy Deliberation strategy is used, which introduces goal cardinalities and inhibition arcs between goals [21]. For each selected goal means-end reasoning is employed to achieve that goal by executing as many plans as necessary. More specifically, means-end reasoning first collects applicable plans and then selects a candidate among these that is subsequently executed. Given that this plan is not able to fulfill the goal, e.g. because it fails, means-end reasoning tries to activate other plans.

To support a wide spectrum of use cases different goal kinds have been introduced, from which *achieve, maintain, query* and *perform* are the most important ones. Achieve goals are used to bring about a specific world state, which can be described as declarative target condition. The goal is considered as fulfilled when this target condition becomes true. In contrast, maintain goals are utilized to preserve a specific world state and reestablish this state whenever it gets violated. Query goals can be used to retrieve information. If the requested piece of knowledge is already known to the agent the goal is immediately finished, whereas otherwise plan execution is started to fetch the needed data. The perform goal kind is a purely procedural goal that is directly connected to actions, i.e. a perform is considered as fulfilled when at least one plan could be executed. A detailed description of these goal kinds can be found in [6, 3].

Jadex BDI agents are specified using XML and Java, allowing to separate the descriptive knowledge of the agent structure from the procedural knowledge of plans. An agent type is defined in an agent definition file (ADF), which follows a BDI metamodel described as XML schema. The agent plans are normal Java files that have to extend a given framework class and override at least one method that contains the plan domain logic. From within Java plans agent functionality can be accessed via API (application programming interface) calls, which e.g. allow accessing beliefs or dispatching goals.

*Example*

As an illustrating example of a BDI agent the cleanerworld application (first described in [6]) is shortly presented. The basic scenario idea is that cleaning robots look for waste in a given terrain and bring it to waste bins nearby. Additionally, the robots have to monitor and recharge their battery given that its value is below a specified threshold. At night, the robots do not search for waste but patrol in defined routes to guard the area. The robot objectives

```
1  <agent name="Cleaner" package="...">
2    <beliefs >
3      <beliefset  name="wastes" class="Waste" />
4      ...
5    </beliefs>
6
7    <goals>
8      <achievegoal name="achievecleanup" retry="true" exclude="never">
9        <parameter name="waste" class="Waste">
10         <value>$waste</value>
11       </parameter>
12       <creationcondition language="jcl">
13         Waste $waste && $waste.position!=null
14       </creationcondition>
15     </achievegoal>
16     ...
17   </goals>
18
19   <plans>
20     <plan name="cleanup">
21       <parameter name="waste" class="Waste">
22         <goalmapping ref="achievecleanup.waste"/>
23       </parameter>
24       <body class="CleanUpWastePlan"/>
25       <trigger>
26         <goal ref="achievecleanup"/>
27       </trigger>
28     </plan>
29     ...
30   </plans>
31   ...
32 </agent>
```

**Fig. 2.** Cleaner agent ADF cutout

can intuitively be modeled using a goal-oriented approach and lead to the four corresponding top-level goals: *maintainbatteryloaded*, *achievecleanup*, *performlookforwaste*, and *performpatrol*. The goal names already denote the different goal kinds (e.g. achieve and perform) used for goal modeling and implementation. The relationships between these goals have been further constrained using inhibition arcs. E.g. the maintainbatteryloaded goal is considered as most important and inhibits goals of the other types in order to guarantee that the robot does not break down. In Figure 2 a small cutout of the cleaner agent ADF is shown. It can be seen that the agent mainly has beliefs, goals and plans sections. As part of its beliefs the agent e.g. remembers already spotted wastes (line 3) in a beliefset called *wastes*. The *achievecleanup* goal is defined as achievement goal (lines 8-15) with creation condition (line 14). This condition is triggered whenever the agent senses a new piece of waste. The goal remembers the triggering piece of waste within a parameter also called *waste* (lines 9-11). In the plans section (lines 19-30), the *cleanup* plan has been defined to react on *achievecleanup* goals via a corresponding trigger declaration (lines 25-27). It also defines a parameter for the waste that the plan has to collect (lines 21-23). The value of this parameter is automatically mapped to the waste parameter of the goal using a goalmapping description

(line 22). Finally, the plan head includes a reference to the plan body that realizes the plan logic (line 24). In this case the Java class *CleanUpWastePlan* (not shown) is utilized. For a more complete description of cleanerworld the reader may consider reading [6], whereby deliberation aspects are tackled in [21].

### Micro Agent Kernel

Micro agents represent a very simple internal agent architecture that basically supports an object-oriented behavior specification. A micro agent is very similar to an object with lifecycle and message handling methods. Thus, it has much in common with the notion of an active object [13], which could be considered as a conceptual predecessor of agents. One main difference with respect to active objects is that a micro agent can be accessed not only in an object-oriented way via method invocation, but also by sending agent-oriented messages to it. Micro agents do not offer much functionality, but they have advantages with respect to minimal resource consumption and performance characteristics. Hence, using micro agents can be beneficial whenever the required agent functionality is simple and resource restrictions may apply or a large number of agents is required.

Micro agents are specified as an extension of a predefined agent framework class. It is mandatory that at least one method (*executeBody()*) is overridden, which will contain the domain logic of the agent. In addition, further methods can be supplied with code that are called once at startup (*agentCreated()*) and when termination of the agent is triggered (*agentKilled()*). Whenever an agent receives a message a specific agent method is called (*messageArrived()*) that can also be customized in order to react to incoming requests.

*Example*

Micro agents play out their strengths in scenarios that fit to their characteristics, i.e. scenarios that e.g. only require simple tasks being executed and exhibit device or environmental resource constraints. Examples include wireless sensor networks (WSNs) and RFID (radio-frequency identification) systems. As these technologies are subject to frequent technological changes in [1] a common event-based middleware for WSNs and RFID systems has been proposed, which aims at hiding low level aspects like hardware and basic event processing details. The middleware follows a layered architecture that offers on the application layer an event based processing model purely based on application level, i.e. domain relevant, events. Lower layers are in charge of preprocessing basic sensor and RFID data and employ complex event processing [15] to generate higher level domain events. One element of this middleware is a *duplicate filtering agent* shown in Figure 3. It has the purpose to collect low level events from event sources and forward them to other event processing agents. As event sources like sensors frequently produce events with the same

```
1  package ...;
2  import ...
3
4  public class SensorAgent extends MicroAgent
5  {
6    protected long interval ;
7    protected List events;
8
9    public void messageArrived(Map msg, MessageType mt)
10   {
11     removeOutdatedEvents();
12     if (!( events. contains (msg)))
13     {
14       events. add(new Tuple(new Long(getTime()), msg));
15       msg.put(SFipa.RECEIVERS, getArgument("receivers"));
16       msg.put(SFipa.SENDER, getComponentIdentifier());
17       sendMessage(msg, mt);
18     }
19   }
20
21   public void removeOutdatedEvents()
22   {
23     // Iterate over list starting from oldest entries
24     // and remove due entries until first non-due is found.
25   }
26   ...
27 }
```

**Fig. 3.** Duplicate filter agent cutout

content the filter agent stores events for a specified time interval and only forwards those with new information. The agent is derived from the framework class *MicroAgent* and only overrides the *messageArrived()* method that is automatically called whenever the agent receives a new message. It has two member variables storing the time *interval* and a list for already consumed *events* (lines 6-7). On message arrival the agent first removes outdated events (line 11) and then checks if the event is contained in the events list (line 12). If this is not the case the agent stores the event in the list (line 14) and modifies the receivers and senders of the event to forward it to its predefined receivers (lines 15-17). Theses receivers are fetched as value of an agent argument and are thus passed to the agent at startup.

### BPMN Workflow Kernel

The BPMN workflow kernel allows the execution of business processes described in BPMN [16]. A BPMN process mainly consists of activities that are connected with different kinds of gateways in order to steer the control flow. Furthermore, events play an important role, as they signal important occurrences within a process, e.g. starting, terminating a process instance or signaling message sending and receival. Elements can be allocated to pools and lanes, which allow a process to be aligned according to underlying organizational structures. BPMN was initially conceived as a modeling language for business process that primarily serves documentation and communications

means, but can also be made directly executable, if elements are annotated with execution information and are equipped with a strict semantics.

The BPMN workflow kernel supplies its active components with a BPMN interpreter, which is able to read BPMN models stored in an XML format. The modeling of BPMN diagrams is currently supported by an extended version of the graphical BPMN editor available in eclipse (stp)[1]. The extended editor mainly adds the capability of property views for all kinds of elements. In these properties execution relevant details can be specified so that the diagram remains simple and readable also for non IT experts.

*Example*

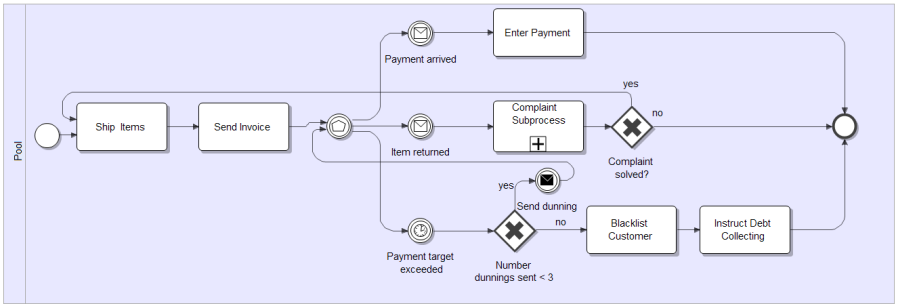


**Fig. 4.** Delivery process example

As an example a small piece of a commercial application scenario is presented. It is assumed that a company exists that sells items to customers. Besides the core processes that are concerned with selling goods and marketing special offers also the delivery of goods and accounting has to be considered. In Figure 4 the delivery process is shown modeled in BPMN using the Jadex eclipse editor. It can be seen that the process first ships the items and sends an invoice to the customer. Thereafter, a multi event is used to disambiguate between different process continuations. In case the payment arrives, it is entered into the books and the process finishes. If instead the customer returns the items e.g. due to quality problems, a specific complaint management subprocess is started to solve the issues. A successful correction of defects leads to reshipping the goods, whereas the process terminates otherwise. It may also happen that the customer does not react at all and the payment target is exceeded. In this case, up to three dunnings are sent and the process then again waits for a customer response. If the customer still does not react, she will be blacklisted and a debt collecting agency will be instructed. In order to make the process executable, element specific Jadex properties are introduced (not shown in Figure 4). Most importantly, activities are connected to Java classes implementing the corresponding domain logic, e.g. the send invoice activity

[1] `http://www.eclipse.org/bpmn/`

prepares an invoice document from a template and sends it per email to the customer. In addition, the dataflow, consisting of local and global parameters, has to be defined. One example is the number of sent dunnings that is saved in a global parameter in order to make it accessible for the checking gateway as well as the send dunning activity, which increments the counter.

### GPMN Workflow Kernel

Basis of the GPMN kernel is the goal-oriented process notation, which is developed in the ongoing Go4Flex project [4] together with Daimler AG. The objective of GPMN consists in providing an additional modeling notation for processes that abstracts away from workflow details and instead focuses on the underlying aims a process shall bring about. For this purpose GPMN introduces different goal types as conceptual elements. These goals are arranged in goal hierarchies for describing how top-level goals can be decomposed into subgoals and plans. A goal hierarchy represents the declarative properties of the process (conditions to be fulfilled), while plans capture procedural aspects (sequences of actions to be executed). The representation and execution semantics for GPMN workflows has been directly adapted from the notion of goals in mentalistic BDI agents as described in Section 3.2. This means that the same goal kinds are available for modeling (achieve, maintain, query, perform) and also deliberation based inhibition arcs can be used. In contrast to conventional BDI, GPMN introduces different modeling patterns capturing recurrent design choices. These patterns e.g. include sequential and parallel subgoal decomposition, i.e. in GPMN a goal may have direct subgoals, which can be declared to be executed one by one or in parallel. It has to be noted that, if the top-level goal has a target condition, subgoal processing will be terminated as soon as the condition becomes true, independent of the processing state of the subgoals.

Goal oriented workflows are executed by a GPMN kernel that converts GPMN to BDI agent models. In this way the GPMN kernel does not have to provide its own execution logic. GPMN diagrams can be graphically modeled by a newly developed eclipse based GPMN editor. The editor allows drawing goal hierarchies and connecting them with BPMN diagrams for concrete subprocesses. The usage of the GPMN editor is very similar to the BPMN version so that an integrated usage of both tools is adequately supported.

*Example*

In the following, an example GPMN manufacturing process will be sketched. Assume a company has specialized in manufacturing cleaner robots. Customers can compose their own cleaners by selecting from a number of configuration options (engine, sensors, garbage claws, etc.). Each cleaner has a control unit, which is a generic component of a 3rd-party supplier, but requires custom software to be installed, depending on the cleaner configuration. Figure 5 shows how this process can be modeled in GPMN. For simplicity, only

the subprocess for building a control unit is shown. The root goal of this subprocess is '*Control Unit Built*'. It is a sequential achieve goal as denoted by the '1..n' at the bottom. This means that the two subgoals '*Control Unit Ready*' and '*Software Installed*' need to be executed in order. The 'Control Unit Ready' goal has two subgoals but does not impose a sequential ordering. Thus the '*Control Software Available*' and '*Control Unit Available*' goals can be executed in parallel.

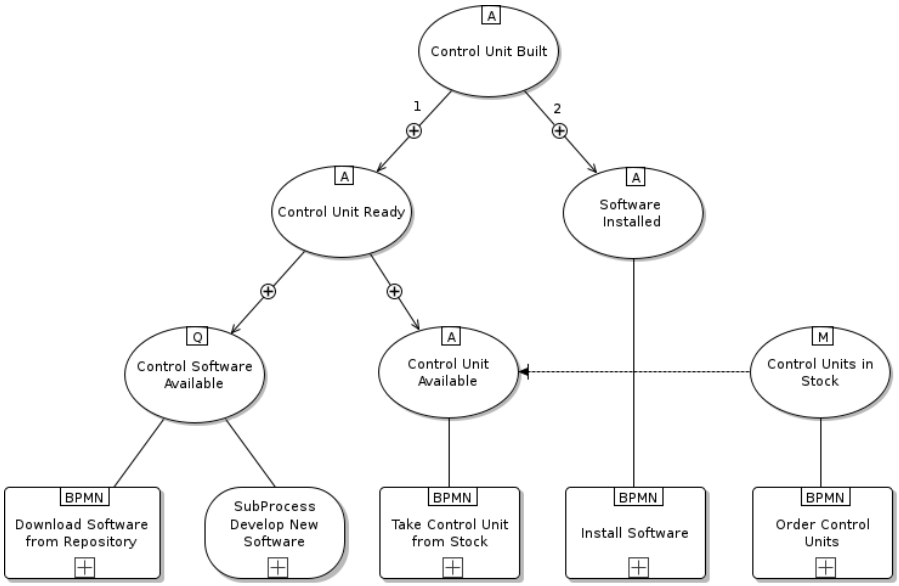


**Fig. 5.** Cutout of a manufacturing process in GPMN

The five goals described above make up the goal hierarchy of the process. The three leaf goals (i.e. goals which are not further decomposed into subgoals) are mapped to concrete plans or subprocesses. The 'Control Software Available' goal is a query goal, which means that when an appropriate software version is readily available, no plan needs to be executed at all. If an appropriate software version is not available locally, yet exists in some repository, the '*Download Software from Repository*' plan is executed. Otherwise no specific software exists for the cleaner configuration selected by the customer and a new software version has to be developed ('*Develop New Software*' subprocess).

The control units from the 3rd party supplier are kept in a stock. Thus the 'Control Unit Available' goal can be achieved by a simple '*Take Control Unit from Stock*' plan. Besides the main goal hierarchy, the separate '*Control Units in Stock*' maintain goal has the responsibility to assure that there are always enough control units in stock. Whenever the number of units drops below a threshold, the '*Order Control Units*' plan is executed. When there are no units in stock, the maintain goal will inhibit the 'Control Unit Available' goal. Thus

the construction process will not fail, but wait until there are new control units in stock.

### Application Kernel

The application kernel belongs to the "other kernels" category. Its main purpose is to provide required functionality for defining applications, e.g. specifying the required components and their interrelations. An application specification thus mainly contains structural information about the application type. A key concept of an application type definition is that of necessary *component types*. Additionally, so called *space types* are introduced, which have been inspired by the context and projection concepts of the Repast simulation toolkit [8]. A space is a very general concept for the representation of non-active elements. It is a structure that contains application specific data and functionality independently from a single component. Therefore a space provides a convenient way of sharing resources among components without using message-based communication. The space concept can be seen as an additional structuring element. It does not impose constraints on components, i.e. components from the same or different applications can communicate via other means such as messages. Spaces also can be seen as an extension point of the component platform as spaces offer application functionality, independent of component behavior. Please note that the concrete functionalities of a space depend on its concrete type and are not directly part of the application concepts. In order to define in what way an application instance should be created from an underlying application type, the concept of configurations is introduced. A configuration describes which runtime entities comprise a specific application instance, i.e. which components and spaces should be created at startup time. At runtime an application represents a component in its own right, which mainly acts as a container for components and spaces. Components that are part of an application can access the spaces via the containing application instance. In this way the access to spaces is restricted to components from the same application context. Representing applications as components also allows for handling them at the tool level, i.e. instead of starting or stopping many single components, whole applications can be managed.

The space concept is very general and can be interpreted e.g. in structural or behavioral ways. Several space types are provided as part of Jadex that capture different recurring functional requirements. A simplified version of Ferber's agent-group-role model [9] allows defining group structures for components and assigning roles to component instances. Another space type is currently under development for weaving de-centralized coordination mechanisms in the application without changing the component's behavior descriptions [25]. The most elaborated space type is the so called EnvSupport [11]. This space is a virtual 2d environment for situated agents, in which they can perceive and act via an avatar object connected to them. The space facili-

tates the construction of simulation examples, as it takes over most parts of visualization and environment/component interaction.

In Jadex, applications and their spaces are described using an XML descriptor file following a metamodel defined as XML schema. An application is mainly composed of space and component types as well as initial instances of both. Component types represent references to other component specification files, which will be included with a logical name in the application context. At startup of an application the kernel will create the declared component instances and spaces of the given configuration.

*Example*

An example application is shown in Figure 6. It represents a virtual environment for testing cleaner robots. The example makes use of the EnvSupport space as explained in the following. The environment is defined as a continuous 2D area (lines 3-27), in which space objects such as cleaners, charging stations and waste items are located (lines 4-10). The cleaner objects, called avatars, are connected to the cleaner agents and allow them to act and perceive in the environment via user defined actions (lines 11-14) and percepts (omitted for brevity), e.g. for spotting and picking up waste. Furthermore, tasks (lines 15-18) can be directly attached to space objects, such as moving the cleaner robot or charging its battery. Besides object behavior, also global behavior can be specified in terms of environment processes (omitted for space reasons), which may operate on all objects of the environment. Such processes can e.g. be used to model environmental activities, like random appearance of waste. Using the aforementioned concepts the application domain can be described. In addition, also the visualization can be specified in terms of possibly different perspectives (lines 19-26). A perspective basically consists of drawables, which specify a graphical representation of a space object type, e.g. of the chargingstation object (lines 21-23).

The application further defines the component types to be used, such as the cleaner robot and a truck, which periodically empties waste bins (lines 30-33). Finally, application configurations are specified (lines 35-51) that denote how an application should be started. Possible settings are initial objects and their locations (e.g. placement of a chargingstation is done in lines 40-42) as well as the initially started components (two cleaner agents are created in line 48).

## 4 Realization

In this section details of the Jadex active components infrastructure implementation will be given. Concretely, it will be shown how the platform architecture has been conceived and the Standalone Platform has been implemented. Thereafter, the generic kernel architecture will be presented and further explained exemplarily by the BPMN kernel.

```
<applicationtype name="CleanerWorldSpace" package="jadex.bdi.examples.cleanerworld">
  <spacetypes>
    <env:envspacetype name="2dspace" class="ContinuousSpace2D"width="1"height="1">
      <env:objecttypes>
        <env:objecttype name="cleaner">
          <env:property name="vision_range">0.1</env:property>
          ...
        </env:objecttype>
        ...
      </env:objecttypes>
      <env:actiontypes>
        <env:actiontype name="pickup_waste" class="PickupWasteAction"/>
        ...
      </env:actiontypes>
      <env:tasktypes>
        <env:tasktype name="move" class="MoveTask" />
        <env:tasktype name="load" class="LoadBatteryTask" />
      </env:tasktypes>
      <env:perspectives>
        <env:perspective name="icons" class="Perspective2D" opengl="true">
          <env:drawable objecttype="chargingstation" width="0.06" height="0.06">
            <env:texturedrectangle imagepath="cleanerworld/images/chargingstation.png"/>
          </env:drawable>
          ...
        </env:perspective>
      </env:perspectives>
    </env:envspacetype>
  </spacetypes>

  <componenttypes>
    <componenttype name="Cleaner" filename="cleanerworld/cleaner/Cleaner.agent.xml"/>
    <componenttype name="Truck" filename="cleanerworld/truck/Truck.agent.xml"/>
  </componenttypes>

  <applications>
    <application name="Two cleaners">
      <spaces>
        <env:envspace name="my2dspace" type="2dspace" width="1.0" height="1.0">
          <env:objects>
            <env:object type="chargingstation">
              <env:property name="position">new Vector2Double(0.8, 0.8)</env:property>
            </env:object>
            ...
          </env:objects>
        </env:envspace>
      </spaces>
      <components>
        <component type="Cleaner" number="2"/>
      </components>
    </application>
  </applications>
</applicationtype>
```

**Fig. 6.** Cutout of an application.xml

### 4.1 Platform Architecture

The overall Jadex AC platform architecture is shown in Figure 7. Its setup directly contributes to two of the initial design challenges. First, it facilitates the execution of applications composed of heterogeneous components, because the kernel is realized as a separate layer on top of the platform layer. Both lay-

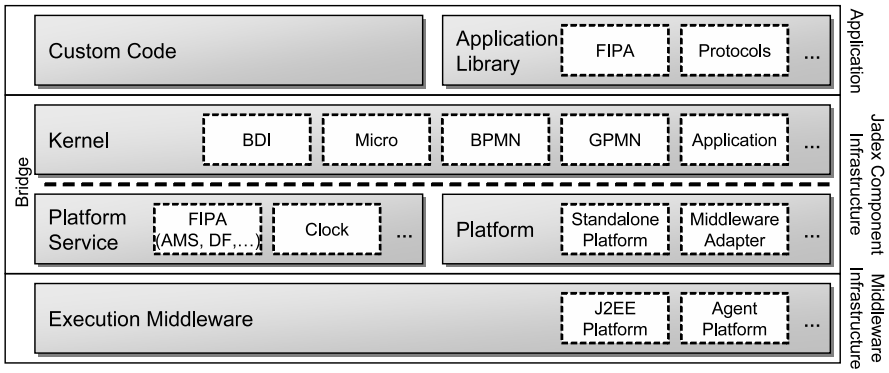


**Fig. 7.** Platform architecture

ers form the core of the Jadex component infrastructure but are only loosely coupled. This coupling is based on a set of common interfaces defined as underlying bridge between both layers, i.e. both layers have access to the *bridge* interfaces. Second, the usage of services for providing platform functionalities allows Jadex to be used in a multitude of execution scenarios due to the high adaptability of the platform layer. Especially, it allows existing middleware to be reused by wrapping offered functionalities in terms of Jadex platform services. In order to execute different component architectures within a single platform it is necessary to specify the responsibilities of the kernel and the platform. A platform is mainly in charge of executing a component, delivering messages to the component and notifying the component at certain time points. From kernel side it is necessary to have access to the platform services.

### 4.2 Standalone Platform Implementation

The Jadex Standalone Platform is a lightweight pure Java SE based execution environment that is not based on an underlying middleware. It realizes a service container concept, which means that it only exposes basic functionalities for managing platform services. The service container allows for adding, removing services and fetching services by their type. A service client thus only has to know the interface of the needed service in order to retrieve it.

Platform services can be customized freely and hence the platform can be individualized with regard to the concrete application scenario by simply changing its declarative configuration. On startup the Standalone Platform reads configuration files and evaluates them with respect to the initial services and components to start.

The Jadex platform services include internal services as well as public services, whereby internal services are only used from other platform services. Public services are available to active components as well. Internal services include a thread pool service and an execution service that are responsible for running active components. Public services encompass infrastructure func-

tionalities similar to FIPA agent specifications[2], i.e. a component management service for component creation and termination, a directory facilitator that represents a service registry as well as a message transport service. Furthermore, a clock and a simulation service exist to enable application execution in real-time as well as in event-driven or time-stepped simulation modes.

### 4.3 Kernel Architecture

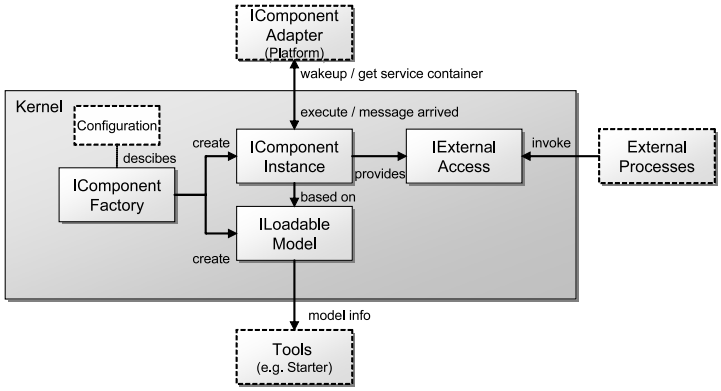


**Fig. 8.** Kernel architecture

In Figure 8 the basic Jadex kernel architecture is depicted. It mainly comprises implementations of interfaces for the *component factory* (*IComponentFactory*), the *component model* (*ILoadableModel*), the *component instance* (*IComponentInstance*) as well as the *external access* point (*IExternalAccess*).

The component factory provides functionalities for loading component models as well as creating instances of those models. In this regard a component model represents the type information about a specific user defined domain component, e.g. which belief, plan and goal types belong to a custom BDI agent. The associated *ILoadableModel* interface allows model consumers such as the Starter tool to handle all models in the same way without knowledge about the underlying kernel type used. Each active component consists of two parts that interact closely: the component instance and the component adapter. The component instance contains the execution logic of the specific component type and uses its associated component adapter as platform mediator e.g. for locating platform services. The component adapter holds platform related information like the component identifier and delegates execution requests from the platform to the component instance. A detailed description of the interaction relationship between component instance and adapter can be found in [18].

[2] `http://www.fipa.org/`

For accessing a component from external threads, e.g. from a user interface, a dedicated external access view on the component exists. It can be retrieved by the component instance and offers general as well as kernel specific methods for component interaction, e.g. in case of a BDI agent an external process can access beliefs or create and dispatch a goal. The integration of a kernel is done via a platform configuration file in which the component factory is announced as new platform service, i.e. kernels are pluggable by configuration.

### 4.4 BPMN Kernel implementation

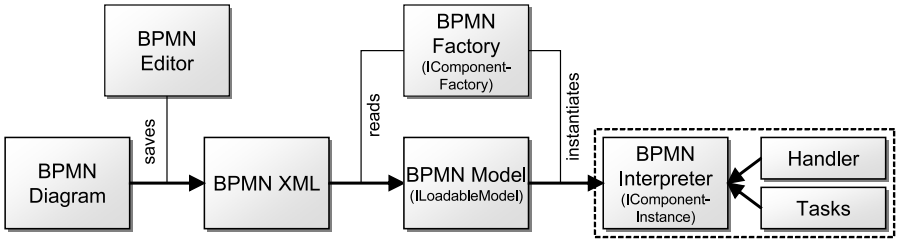


**Fig. 9.** Loading a BPMN model

The BPMN kernel implementation basically provides BPMN specific implementations of the kernel interfaces described in the last section. In Figure 9 it can be seen how a BPMN diagram is processed from the kernel in order to execute a process instance on basis of this diagram. Starting point is the eclipse BPMN editor, which saves the diagram in a proprietary XML file format, which consists of two files: one representing the model and one describing the layout information.[3] The first one serves as input for the BPMN component factory, which reads the file and generates a BPMN Java model. On basis of this model the component factory can create BPMN process instances (interpreter components), which may be executed on the Jadex active components infrastructure.

The interpreter follows a lightweight and extensible approach for realizing the BPMN functionality. Internally, it uses so called process threads for managing concurrent flows within a process. Such a process thread represents a virtual thread in contrast to a real one managed by the operating system. A virtual process thread records the execution and memory state of a process flow and is used to steer the execution of process activities. Due to their virtual character, a process instance is executed in a quasi-parallel way like all active components. The interpreter functionality is realized using *handlers* and *tasks*. Handlers exist for each predefined BPMN element, like gateway types and different kinds of events, and implement their internal behavior. A parallel split gateway e.g. creates virtual process threads for each of the outgoing parallel branches. The corresponding join gateway then waits for all incoming threads

[3] In future versions BPMN 2.0 will allow using a standard XML format.

and unifies them to one that follows the outgoing edge. Changing or adding handlers allows flexibly controlling the execution behavior of the interpreter. Tasks are used for implementing activity behavior. Several predefined tasks are available for standard activities like printing on the console or requesting user input. Besides these ready-to-use tasks, user-defined tasks can be created by extending a specific framework class. E.g. a workflow management system (WfMS) is currently being developed based on the BPMN kernel, which provides custom tasks for connecting the modeled processes to other aspects of the WfMS infrastructure, such as a worklist client (for manual tasks) and external applications (for automated tasks).

## 5 Example Projects

The Jadex active component infrastructure has been developed and is continued to be used in various real-world application projects. Three of these projects will be described exemplarily in the following.

### 5.1 MedPAge: Agent-based Hospital Scheduling

The MedPAge (medical path agents) project was part of the German priority research programme "intelligent agents and realistic commercial application scenarios" (SPP 1083), which was funded by the Deutsche Forschungsgemeinschaft (DFG) from 1999-2006. The aim of the research programme was to show the applicability and advantages of intelligent agent technology in real world applications from the hospital and manufacturing logistics domains. As a joint project between the University of Mannheim and the University of Hamburg, in the MedPAge project a demonstrator was developed that improved treatment scheduling for patients in hospitals. The main approach was representing patients and hospital resources as Jadex BDI agents that negotiate treatment slots. The approach assures that patient goals (low waiting times) and resource goals (high utilization) are equally respected. Details about the scheduling algorithm and the application can be found e.g. in [17].

The main focus of the Jadex framework at that time was BDI-based agents. During the course of the MedPAge project, the Jadex framework has been evaluated in depth (see e.g. [7]). As a result of this evaluation, two important factors where identified for further improvement, which required tedious and error-prone manual work in MedPAge: 1) the support for non-functional requirements such as persistence and scalability and 2) the integration of agent technology with other mainstream technology like software components and workflows. Both factors led to the development of the Jadex active components infrastructure, which broadens the scope of the original Jadex framework and is an essential foundation for our current application projects described next.

### 5.2 Go4Flex: Agile Process Management

The DFG technology transfer project Go4Flex (goal-orientation for flexible and agile processes) [4] is conducted in cooperation between Daimler AG and the University of Hamburg and aims at providing advanced conceptual and software technical means for modeling and executing complex business processes. In practice, experience has shown that modeling means offered by traditional workflow languages such as event process chains (EPCs) or BPMN are insufficient for many processes at large companies like Daimler AG. While processes can be documented with EPCs or BPMN, they are not directly adopted by the workflow participants. One major issue is the strong focus on activities and their ordering. As processes are typically prone to frequently change, the abstractness of the process descriptions is essential for their long-term usefulness.

The Go4Flex framework is based on concepts, which have been developed in the area of agents and multi-agent systems. The main research question of the DFG-funded Go4Flex project is to isolate interesting multi-agent ideas and make them usable also for workflows. Most importantly, Go4Flex focuses on the behavior and context perspectives, which suffer among other things from their low conceptual connectivity. Thus Go4Flex introduces the goal process modeling notation (cf. Section 3.2) for abstract modeling of flexible workflows. The Jadex active component infrastructure forms the basis of the Go4Flex workflow management system and allows using seemingly disparate concepts like agents and workflows seamlessly.

### 5.3 SodekoVS: Systematically Engineering Self–Organizing Systems

The DFG-funded research project SodekoVS (self-organisation based on decentralized co-ordination) [24] is a cooperation of the University of Applied Sciences Hamburg and the University of Hamburg and aims at tackling coordination problems by utilizing nature-inspired design paradigms. These provide coordination strategies to equip software architectures with adaptability and robustness, based on decentralized self-organization principles. Basis of the approach are a newly conceived generic reference architecture as well as an adapted development methodology for the systematic construction of such systems. Coordination mechanisms are made available as middleware services and a minimally intrusive programming model allows developers to configure and integrate representations of nature-inspired coordination strategies in their applications. The systematic utilization of these development tools requires support to design, i.e. model, select, combine and refine self-organizing dynamics, and to simulate the resulting application prototypes.

The project heavily relies on many features of the Jadex active component infrastructure. The coordination algorithms require large numbers of components being executed in parallel. This kind of scalability is provided by the

micro agent kernel. The external access facility (cf. Section 4.3) is used for the programming model to integrate the coordination layer with the application functionality, which can further be realized using any component type. Finally, the support for simulated execution is used as a basis of a validation tool to automatically check if the employed coordination strategies converge and adapt as desired.

## 6 Summary and Outlook

In this chapter active components have been presented as novel notion for building complex distributed and concurrent systems. The active component concept has emerged from an integration of agent and software component concepts and thus tries to combine the strength of both. Active components are the conceptual basis of the Jadex middleware, which has been conceived to be able to execute arbitrary types of active components. The internal architecture of active components is realized in different kernels that can be used together on the same component platform allowing heterogeneous applications being constructed. Currently, agent, workflow and application kernels exist, which have been presented in detail according to their architecture as well as usage by drawing on typical example scenarios. Furthermore, the underlying implementation concepts of a kernel have been discussed, which mainly rely on a component factory that is able to load active component models and instantiate them. The active component concept has been further illustrated by selected research projects, which make use of different platform features. MedPAge, a decentralized appointment scheduling solution for hospitals, makes heavy use of agent-based negotiation capabilities. In contrast, in Go4Flex, a project aiming at flexible workflow description and execution, the integration of workflow and agent concepts helped solving the agility demands. Finally, for the SodekoVS project, which aims at providing self-organization algorithms as reusable patterns, the scalability of the micro agent kernel and the simulation features of the platform are essential. Future work will especially tackle the conceptual integration challenges of the service oriented architecture and active components.

## References

1. Dirk Bade and Winfried Lamersdorf. An agent-based event processing middleware for sensor networks and rfid systems. *Computer Journal, Special Issue on "Agent Technologies for Sensor Networks"*, 2009.
2. F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent systems with JADE*. John Wiley & Sons, 2007.
3. L. Braubach and A. Pokahr. Representing long-term and interest bdi goals. In Thangarajah Braubach, Briot, editor, *Proceedings of International Workshop on Programming Multi-Agent Systems (ProMAS-7)*, pages 29–43. IFAAMAS Foundation, 5 2009.

4. 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.
5. L. Braubach, A. Pokahr, and W. Lamersdorf. Jadex: A BDI Agent System Combining Middleware and Reasoning. In R. Unland, M. Calisti, and M. Klusch, editors, *Software Agent-Based Applications, Platforms and Development Kits*, pages 143–168. Birkhäuser, 2005.
6. L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal Representation for BDI Agent Systems. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, editors, *Proceedings of the 2nd International Workshop on Programming Multiagent Systems (ProMAS 2004)*, pages 44–65. Springer, 2005.
7. Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. A universal criteria catalog for evaluation of heterogeneous agent development artifacts. *International Journal of Agent-Oriented Software Engineering (IJAOSE)*, 2009. to appear.
8. N. Collier. RePast: An Extensible Framework for Agent Simulation. Working Paper, Social Science Research Computing, University of Chicago, 2001.
9. J. Ferber, O. Gutknecht, and F. Michel. From Agents to Organizations: an Organizational View of Multi-Agent Systems. In P. Giorgini, J. Müller, and J. Odell, editors, *Proceedings of the 4th International Workshop on Agent-Oriented Software Engineering IV (AOSE 2003)*, pages 214–230. Springer, 2003.
10. K. Fischer, M. Schillo, and J.H. Siekmann. Holonic multiagent systems: A foundation for the organisation of multiagent systems. In Vladimír Maník, Duncan C. McFarlane, and Paul Valckenaers, editors, *HoloMAS*, volume 2744 of *Lecture Notes in Computer Science*, pages 71–80. Springer, 2003.
11. K. Jander, L. Braubach, and A. Pokahr. Envsupport: A framework for developing virtual environments. In *Seventh International Workshop From Agent Theory to Agent Implementation (AT2AI-7)*. Austrian Society for Cybernetic Studies, 2010.
12. N. R. Jennings. An agent-based approach for building complex software systems. *Communications of the ACM*, 44(4):35–41, April 2001.
13. G. Lavender and D. Schmidt. Active object - an object behavioral pattern for concurrent programming. In J. Vlissides, J. Coplien, and N. Kerth, editors, *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.
14. M. Luck, P. McBurney, O. Shehory, and S. Willmott. *Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing)*. AgentLink, 2005.
15. David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
16. Object Management Group (OMG). *Business Process Modeling Notation (BPMN) Specification*, version 1.1 edition, February 2008.
17. 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.
18. A. Pokahr and L. Braubach. From a research to an industrial-strength agent platform: Jadex V2. In Hans-Georg Fill Hans Robert Hansen, Dimitris Karagiannis, editor, *Business Services: Konzepte, Technologien, Anwendungen - 9.*

*Internationale Tagung Wirtschaftsinformatik (WI 2009)*, pages 769–778. Österreichische Computer Gesellschaft, 2 2009.
19. A. Pokahr, L. Braubach, and K. Jander. Unifying agent and component concepts - jadex active components. In J. Dix and C. Witteveen, editors, *Proceedings of the 8th German conference on Multi-Agent System TEchnologieS (MATES-2010)*. Springer, 2010.
20. A. Pokahr, L. Braubach, and W. Lamersdorf. A Flexible BDI Architecture Supporting Extensibility. In A. Skowron, J.-P. Barthès, L. Jain, R. Sun, P. Morizet-Mahoudeaux, J. Liu, and N. Zhong, editors, *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2005)*, pages 379–385. IEEE Computer Society, 2005.
21. 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.
22. A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI Reasoning Engine. In R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming: Languages, Platforms and Applications*, pages 149–174. Springer, 2005.
23. 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.
24. J. Sudeikat, L. Braubach, A. Pokahr, W. Renz, and W. Lamersdorf. Systematically engineering self–organizing systems: The sodekovs approach. In M. Wagner, D. Hogrefe, K. Geihs, and K. David, editors, *Proceedings des Workshops über Selbstorganisierende, adaptive, kontextsensitive verteilte Systeme (KIVS 2009)*, page 12. Electronic Communications of the EASST, 3 2009.
25. A. Vilenica, A. Pokahr, L. Braubach, W. Lamersdorf, J. Sudeikat, and W. Renz. Coordination in multi-agent systems: A declarative approach using coordination spaces. In *Seventh International Workshop From Agent Theory to Agent Implementation (AT2AI-7)*. Austrian Society for Cybernetic Studies, 2010.
26. M. Wooldridge. *Reasoning about Rational Agents*. MIT Press, 2000.
27. M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2001.