Jadex: A BDI-Agent System Combining Middleware and Reasoning

Lars Braubach, Alexander Pokahr and Winfried Lamersdorf

Abstract. Nowadays a multitude of different agent platforms exist that aim to support the software engineer in developing multi-agent systems. Nevertheless, most of these platforms concentrate on specific objectives and therefore cannot address all important aspects of agent technology equally well. A broad distinction in this field can be made between middleware- and reasoning-oriented systems. The first category is mostly concerned with FIPA-related issues like interoperability, security and maintainability whereas the latter one emphasizes rationality and goal-directedness. In this paper the Jadex reasoning engine is presented, which supports cognitive agents by exploiting the BDI model and is realized as adaptable extension for agent middleware such as the widely used JADE platform.

1. Introduction

Today various different agent platforms are available providing support for the development of agent applications [21]. As agent orientation is a very broad field covering topics concerning inter alia agent organizations, agent behaviour as well as messaging it becomes obvious that most of these platforms focus on specific objectives and therefore cannot address all important aspects of agent technology equally well. Two important categories of platforms are middleware- and reasoning-oriented systems.

The first category is mostly concerned with FIPA-related issues that address interoperability and various infrastructure topics such as white and yellow page services. Hence agent middleware is an important building block that forms a solid foundation for exploiting agent technology. Most middleware platforms intentionally leave open the issue of internal agent architecture and employ a simple task oriented approach [2, 15, 32]. In contrast, reasoning-centered platforms focus on the behaviour model of a single agent trying to achieve rationality and

goal-directedness. Most successful behaviour models are based on adapted theories coming from disciplines such as philosophy, psychology or biology. Depending on the level of detail of the theory the behaviour models tend to become complicated and can result in architectures and implementations that are difficult to use. Especially when advanced artificial intelligence and theoretical techniques such as deduction logics are necessary for programming agents, mainstream software engineers cannot easily take advantage of agent technology.

In this paper the Jadex agent framework is presented, which builds upon an existing middleware agent platform and supports easy to use reasoning capabilities. It adopts the BDI model [5] and combines it with state-of-the-art software engineering technologies like XML and Java. In the following, section 2 motivates the need for agent-oriented middleware. In section 3 reasoning approaches for agents are sketched and the BDI fundamentals regarding the individual concepts and their interrelationships are described. Section 4 explains the design and implementation of the Jadex system by detailing the abstract architecture and several implementation aspects. In section 5 the approach taken by Jadex is classified and compared to other approaches - in particular to the JACK agent framework. A summary and an outlook describing ongoing work and planned extensions conclude the paper.

2. Agent Middleware

Typically *middleware* in the field of distributed systems is seen as"[...] network-aware system software, layered between an application, the operating system, and the network transport layers, whose purpose is to facilitate some aspect of cooperative processing. Examples of middleware include directory services, message-passing mechanisms, distributed transaction processing (TP) monitors, object request brokers, remote procedure call (RPC) services, and database gateways." [1]

As agent orientation builds on concepts and technology of distributed systems middleware for agents is equally important for the realization of agent-based applications. Thereby, the term *agent middleware* is used to denote common services such as message passing or persistency management usable for agents. The paradigm shift towards autonomous software components in open, distributed environments requires on the one hand new standards to ensure interoperability between applications. On the other hand new middleware products implementing these standards are needed to facilitate fast development of robust and scalable applications. Agents can be seen as application layer software components using middleware to gain access to standardized services and infrastructure.

The Foundation for Intelligent Physical Agents (FIPA) [28] is an international non-profit organization providing standards for heterogeneous interacting agents and multi-agent systems. Since 1997 a number of specifications have been released which are replaced or updated regularly. The work on specifications focuses on

[1] http://iishelp.web.cern.ch/IISHelp/iis/htm/core/iigloss.htm

application as well as on middleware aspects. Specifications related to applications provide systematically studied example domains with service and ontology descriptions. The middleware-related specifications address in detail all building blocks required for an abstract agent platform architecture. This includes mechanisms for agent management, as well as infrastructure elements such as directory services and message delivery. Besides, there are extensive specifications on the syntactic and semantic layer of agent communication. This concerns inter alia the format and meaning of individual messages as well as the standard interaction protocols providing a unified basis for agent communication and interaction.

The FIPA specifications have been implemented in a number of agent platforms [2, 15, 32] and interoperability among those platforms has been shown, for example in the agentcities network.[2] In addition to the FIPA specifications, several platforms also address further middleware issues and provide specialized solutions, e.g. for security, persistency, or mobility. Although the available middleware platforms therefore provide a solid basis for developing open, interoperable agent systems, not all important aspects of agent development are supported equally well. The middleware platforms provide generic abstractions for application independent distribution and communication issues, but most of them realize a simple task-based agent model. This approach allows decomposing the overall agent behaviour into smaller pieces and attaching them to the agent as needed. Additionally, the tasks themselves can be implemented in an object-oriented language such as Java allowing the software developer to easily start using the agent paradigm. Once agent applications become more complex, another abstraction layer is needed to support the implementation of high-level decision processes inside the agents. Such abstractions are provided by cognitive agent architectures as described in the next section.

3. Reasoning for Agents

To build agents with cognitive capabilities, several theories from different disciplines like psychology, philosophy and biology can be utilized. Most cognitive architectures are based on theories for describing behaviour of individuals. The most influential theories with respect to agent technology are the Belief-Desire-Intention (BDI) model [5], the theory of Agent Oriented Programming (AOP) [31], the Unified Theories of Cognition (UTC leading to SOAR) [20, 23] and the subsumption theory [9]. Each of these theories has its own strengths and weaknesses and supports certain kinds of application domains especially well. The Jadex reasoning engine is based on the BDI model due to its simplicity and folk psychological background as explained further in the following section.

[2] http://www.agentcities.net

3.1. BDI Foundations

The BDI model was conceived by Bratman as a theory of human practical reasoning [5]. Its success is based on its simplicity reducing the explanation framework for complex human behaviour to the *motivational stance* [13]. This means that the causes for actions are always related to human desires ignoring other facets of human cognition such as emotions. Another strength of the BDI model is the consistent usage of folk psychological notions that closely correspond to the way people talk about human behaviour.

Beliefs are informational attitudes of an agent, i.e. beliefs represent the information, an agent has about the world it inhabits, and about its own internal state. But beliefs do not just represent entities in a kind of one-to-one mapping; they provide a domain-dependent abstraction of entities by highlighting important properties while omitting irrelevant details. This introduces a personal world view inside the agent: the way in which the agent perceives and thinks about the world.

The motivational attitudes of agents are captured in *desires*. They represent the agent's wishes and drive the course of its actions. Desires need not necessarily be consistent and therefore may not be achieved simultaneously. A "goal deliberation" process has the task to select a subset of consistent desires (often referred to as *goals*). Actual systems and formal theory mostly ignore this step (with the exception of 3APL [11, 12]) and assume that an agent only possesses non-conflicting desires. In a goal-oriented design, different goal types such as achieve or maintain goals can be used to explicitly represent the states to be achieved or maintained, and therefore the reasons, why actions are executed [8]. When actions fail it can be checked if the goal is achieved, or if not, if it would be useful to retry the failed action, or try out another set of actions to achieve the goal. Moreover, the goal concept allows to model agents which are not purely reactive i.e., only act after the occurrence of some event. Agents that pursue their own goals exhibit pro-active behaviour.

Plans are the means by which agents achieve their goals and react to occurring events. Thereby a plan is not just a sequence of basic actions, but may also include more abstract elements such as subgoals. Other plans are executed to achieve the subgoals of a plan, thereby forming a hierarchy of plans. When an agent decides on pursuing a goal with a certain plan, it commits itself (momentarily) to this kind of goal accomplishment and hence has established a so called *intention* towards the sequence of plan actions. Flexibility in BDI plans is achieved by the combination of two facets. The first aspect concerns the dynamic selection of suitable plans for a certain goal which is performed by a process called "meta-level reasoning". This process decides with respect to the actual situation which plan will get a chance to satisfy the goal. If a plan is not successful, the meta-level reasoning can be done again allowing a recovery from plan failures. The second criteria relates to the definition of plans, which can be specified in a continuum from very abstract plans using only subgoals to very concrete plans composed of only basic actions.

Algorithm 1 BDI-interpreter, taken from [29]

```
BDI-interpreter
Initialize-state();
repeat
   options := option-generator(event-queue);
   selected-options := deliberate(options);
   update-intentions(selected-options);
   execute();
   get-new-external-events();
   drop-successful-attitudes();
   drop-impossible-attitudes();
end repeat
```

3.2. BDI Realization

The foundation for most implemented BDI systems is the abstract interpreter proposed by Rao and Georgeff (see algorithm 1) [29]. At the beginning of every interpreter cycle a set of applicable plans is determined for the actual goal or event from the event queue. Thereafter, a subset of these candidate plans will be selected for execution (meta-level-reasoning) and will be added to the intention structure. After execution of an atomic action belonging to some intention any new external events are added to the event queue. In the final step successful and impossible goals and intentions are dropped. Even though this abstract interpreter loop served as direct implementation template for early PRS systems [18], nowadays it should be regarded more as an explanation of the basic building blocks of a BDI system. Several important topics such as goal deliberation and the distinction between goals and events are not considered in this approach.

4. Jadex Realization

The following sections present the motivation, architecture and execution model of the newly developed reasoning engine Jadex (see also [27]). Details about the integration of the reasoning engine into the platform are described in a separate section. Afterwards some tools are introduced which offer extended support for agent debugging.

4.1. Motivation and Project Background

In the context of the MedPAge project the need for an agent platform was identified that would support FIPA-compliant communication with a high-level agent architecture such as BDI. The MedPAge ("Medical Path Agents") project is part of the German priority research programme 1083 *Intelligent Agents in Real-World Business Applications* funded by the Deutsche Forschungsgemeinschaft (DFG). In cooperation between the business management department of the University of Mannheim and the computer science department of the University of Hamburg,

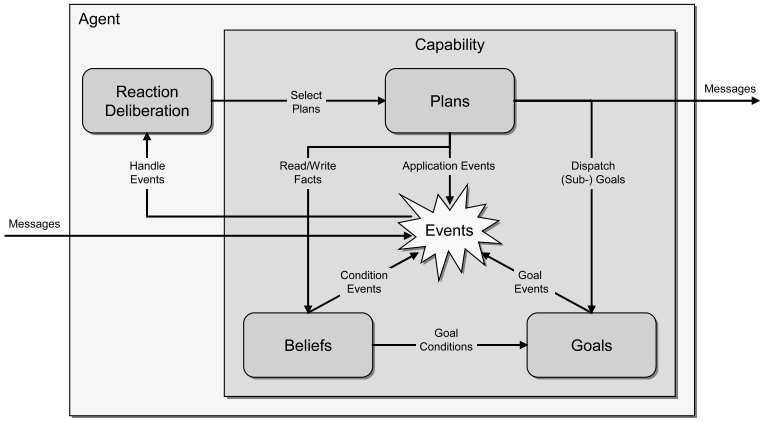

FIGURE 1. Jadex abstract architecture

the project investigates the advantages of using agent technology in the context of hospital logistics [24, 25]. The Jadex project started in December 2002 to provide the technical basis for MedPAge software prototypes developed in Hamburg.

Addressing the need for an agent platform that supports both middleware and reasoning, the approach chosen was to rely on an existing mature middleware platform, which is in widespread use. The JADE platform [3] focuses on implementing the FIPA reference model, providing the required communication infrastructure and platform services such as agent management, and a set of development and debugging tools. It intentionally leaves open much of the issues of internal agent concepts, offering a simple task-based model in which a developer can realize any kind of agent behaviour. This makes it well suited as a foundation for establishing a reasoning engine on top of it. While the agent platform is concerned with external issues such as communication and agent management, the reasoning engine on the other hand covers all agent internals. Therefore the architecture is to a large extent independent from the underlying platform.

4.2. Architecture Overview

In Fig. 1 an overview of the abstract Jadex architecture is presented. Viewed from the outside, an agent is a black box, which receives and sends messages. Incoming messages, as well as internal events and new goals serve as input to the agent's internal reaction and deliberation mechanism. Based on the results of the deliberation process these events are dispatched to plans selected from the plan library. Running plans may access and modify the belief base, send messages to other agents, create new top-level or subgoals, and cause internal events. The reaction and deliberation mechanism represents the only global component within Jadex. All other components are contained in reusable modules called capabilities.

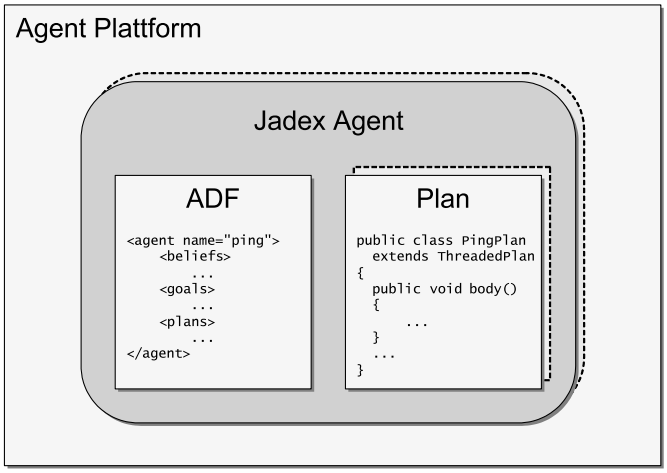

FIGURE 2. Composition of a Jadex agent

4.2.1. Agent Definition. To create and start an agent, the system needs to know the properties of the agent to be instantiated. The initial state of an agent is determined among other things by the beliefs, goals, and the library of known plans. Jadex uses a declarative and a procedural approach to define the components of an agent (see Fig. 2). The plan bodies have to be implemented as ordinary Java classes that extend a certain framework class, thus providing a generic access to the BDI specific facilities. All other concepts are specified in a so called Agent Definition File (ADF) using an XML language that follows the Jadex meta-model (described in [26]) specified in XML schema and allows for creating Jadex objects in a declarative way. Within the XML agent definition files, the developer can use expressions to specify designated properties. The language for these expressions is Java extended with OQL constructs that facilitate e.g. the specification of queries. In addition to the BDI components, some other information is stored in the definition files such as default arguments for launching the agent or service descriptions for registering the agent at a directory facilitator.

The reaction and deliberation mechanism is generally the same for all agents. The behaviour of a specific agent is therefore determined solely by its concrete beliefs, goals, and plans. In the following each of these central concepts of the Jadex BDI architecture will be described in detail.

4.2.2. Beliefs. One objective of the Jadex project is ease of usage. Therefore Jadex does not enforce a logic-based representation of beliefs. Instead, ordinary Java objects of any kind can be contained in the beliefbase, allowing reuse of classes generated by ontology modeling tools or database mapping layers. Objects are stored as named facts (called beliefs) or named sets of facts (called belief sets). Using the belief names, the beliefbase can be directly manipulated by setting, adding, or

```
01  <belief name="alarm_time" class="long">
02    <fact>System.currentTimeMillis()+360000</fact>
03  </belief>
04
05  <belief name="system_time" class="long" updaterate="1000">
06    <fact>System.currentTimeMillis()</fact>
07  </belief>
08
09  <beliefset name="alarm_times" class="long">
10    <fact>$beliefbase.system_time+360000*2</fact>
11    <fact>$beliefbase.system_time+360000*3</fact>
12  </beliefset>
13
14  <beliefset name="alarm_times_from_db" class="long">
15    <facts>Database.queryAlarmTimes()</facts>
16  </beliefset>
```

FIGURE 3. Belief and belief set examples

removing facts. A more declarative way of accessing beliefs and beliefsets is provided by queries, which can be specified in an OQL-like language [4]. The beliefs are used as input for the reasoning engine by specifying certain belief states e.g. as preconditions for plans or creation conditions for goals. The engine monitors the beliefs for relevant changes, and automatically adjusts goals and plans accordingly. E.g. a belief change can trigger a goal's creation or drop condition, or render the context of a plan invalid leading to a plan abort.

In Fig. 3 some example belief and belief set declarations are depicted that e.g. could be usable for realizing some kind of alarm clock agent. The belief "alarm_time" (lines 1-3) represents time in milliseconds and therefore requires its value being of the Java class "long". It provides already an initial fact (line 2) that will be initialized at the agent start-up. Hence the alarm time is set to one hour in the future. For being able to check whether the alarm time has been reached a dynamic "system_time" belief is declared. Using an updaterate for this belief leads to continuous reevaluations of the belief value (here every second). For representing more than a single alarm time a belief set can be employed. The "alarm_times" belief set (lines 9-12) declares two alarm times as initial facts. Note that these facts are only evaluated once and access the belief "system_time" from the beliefbase by using the reserved variable "$beliefbase". If the number of initial facts is unknown, it is also possible to retrieve those values dynamically. E.g. the belief set "alarm_time_from_db" queries a database to retrieve its initial facts.

4.2.3. Goals. Jadex follows the general idea that goals are concrete instantiations of an agent's desires. For any goal it has, an agent will more or less directly engage into suitable actions, until it considers the goal as being reached, unreachable, or

not desired any more. Unlike most other systems, Jadex does not assume that all adopted goals need to be consistent to each other. To distinguish between just adopted (i.e. desired) but not yet active goals and actively pursued goals, a goal lifecycle is introduced which consists of the goal states *option*, *active*, and *suspended* [8]. When a goal is adopted, it becomes an option that is added to the agent's desire structure. A deliberation mechanism is responsible for managing the state transitions of all adopted goals (i.e. deciding which goals are active and which are just options). Some goals may only be valid in specific contexts determined by the agent's beliefs. When the context of a goal is invalid it will be suspended until the context is valid again.

Based on the general lifecycle described above, Jadex supports four types of goals, which exhibit different behaviour with regard to their processing as explained below. A *perform* goal is directly related to the execution of actions. Therefore the goal is considered to be reached when some actions have been executed, regardless of the outcome of these actions. An *achieve* goal is a goal in the traditional sense, which defines a desired outcome without specifying how to reach it. Agents may try several different alternative plans, to achieve a goal of this type. A *query* goal is similar to an achieve goal. Its outcome is not defined as a state of the world, but as some information the agent wants to know about. For goals of type *maintain*, an agent keeps track of the desired state, and will continuously execute appropriate plans to re-establish the maintained state whenever needed. More details about goal representation and processing in Jadex can be found in [8].

Fig. 4 shows some goal declarations picking up the alarm clock agent example again. For realizing the alarm functionality of the agent a "notify_user" achievement goal (lines 5-10) is declared. It has the purpose to notify the user when the alarm time has been reached. Hence, it has a creation condition that leads to a goal instantiation when the alarm time is due (lines 6-8). The goal will be satisfied, when the user is aware of the alarm (e.g. represented by a belief "user_notified"), which may be signaled to the agent by pressing some button of the alarm clock. This is intuitively expressed with the goal's target condition (line 9). If the user does not respond to the alarm, the goal will be retried every ten minutes (cf. retrydelay and exlude settings).

In response to this goal some plan has to be executed. Such a plan could e.g. notify the user by playing one of her favorite songs. To achieve this the plan has to ensure that the favorite song is available (e.g. as mp3 file) as well as it will be played. For retrieving the favorite song a "retrieve_song" query goal can be used (lines 12-15), which requires as input the name of the song to retrieve (line 13) and gives back the song location (line 14). Note, that the direction attribute is used to declare the "song" parameter as return value. Subsequent plans could handle a "retrieve_song" goal e.g. by simply fetching it from a local directory or by downloading it from the internet. For playing the song a "play_song" goal (lines 1-3) is provided. This goal is very simply as it just has one input parameter for

```
01  <performgoal name="play_song">
02    <parameter name="song" class="MediaLocator"/>
03  </performgoal>
04
05  <achievegoal name="notify_user" retrydelay="600000" exclude="never">
06    <creationcondition>
07      $beliefbase.system_time==$beliefbase.alarm_time
08    </creationcondition>
09    <targetcondition>$beliefbase.user_noti  ed</targetcondition>
10  </achievegoal>
11
12  <querygoal name="retrieve_song">
13    <parameter name="song_name" class="String"/>
14    <parameter name="song" class="MediaLocator" direction="out"/>
15  </querygoal>
16
17  <maintaingoal name="keep_clock_adjusted">
18    <maintaincondition>
19      Math.abs($beliefbase.system_time-$beliefbase.reference_time)<500
20    </maintaincondition>
21  </maintaingoal>
```

FIGURE 4. Goal examples

the song file. Suitable plans supporting different sound formats could be provided to actually play the music.

In addition to the alarm functionality, a "keep_clock_adjusted" maintenance goal (lines 17-21) could be used to ensure that the clock is in line with a reference time. Therefore, a maintain condition is defined that is valid as long as system and reference time do not differ more than 0.5 secs (line 19). Whenever this condition is violated the goal will become active and trigger plan executions for synchronizing the system clock.

4.2.4. Plans. The reasoning engine handles all events such as the reception of a message or the activation of a goal by selecting and executing appropriate plans. Instead of performing planning from first principles for each event, BDI systems like Jadex use the plan-library approach to represent the plans of an agent. For each plan a plan head defines the circumstances under which the plan may be selected and a plan body specifies the actions to be executed. In Jadex, the most important parts of the head are the goals and/or events which the plan may handle and a reference to the plan body. Additionally, a context condition as well as variable bindings can be specified in the plan head.

The agent programmer decomposes concrete agent functionality into separate plan bodies, which are predefined courses of action implemented as Java classes.

```
01  /** Plan skeleton for an application plan. */
02  public class SomePlan extends jadex.runtime.Plan {
03
04      public void body() {
05          // Plan code.
06      }
07
08      public void passed() {
09          // Optional cleanup code in case of a plan success.
10      }
11      public void failed() {
12          // Optional cleanup code in case of a plan failure.
13      }
14      public void aborted() {
15          // Optional cleanup code in case the plan is aborted.
16      }
17  }
```

FIGURE 5. Plan body skeleton

Object-oriented techniques and existing Java IDEs can be exploited in the development of plans. Plans can be reused in different agents, and can incorporate functionality implemented in other Java classes e.g., to access a legacy system. To access functionality of the Jadex system, a Java API is provided for basic actions such as sending messages, manipulating beliefs, or creating subgoals.

The basic structure of a plan body is shown in Fig. 5. The plan body is a Java class that extends the Jadex framework class "Plan" and hence has access to the BDI specific methods provided by the Plan API. The domain specific behaviour of the plan will be placed inside the mandatory body method (lines 4-6). Additionally, the three methods passed() , failed() and aborted() are provided allowing a plan to perform clean up operations (lines 8-16). These methods are invoked automatically with respect to the plan's final success state. If the body() method runs through the passed() method is called, whereas the failed() method is called when an uncatched exception occurs within the body() method. Finally, the aborted() method is called, when plan processing was interrupted from outside. Two different abort cases can be distinguished, either when the corresponding goal succeeds before the plan is finished (i.e. its target condition is fulfilled) or when the plans root goal is dropped.

As an example for a plan declaration in Fig. 6 the plan head (top, lines 1-6) and plan body (bottom, lines 1-10) of a "notification" plan suitable for the above described alarm clock agent are presented. The plan head is very simple in this case and consists only of the obligatory body expression (line 2) that describes how a plan body is created at runtime and how it is triggered (line 4). As the

```
01  <plan name="notify">
02    <body>new Noti  cationPlan()<body/>
03    <trigger>
04      <goal ref="notify_user"/>
05    </trigger>
06  </plan>
```

```
01  public class Noti  cationPlan extends Plan{
02    public void body(){
03      IGoal retrieve = createGoal("retrieve_song");
04      retrieve.getParameter("song_name").setValue("Jingle Bells");
05      dispatchSubgoalAndWait(retrieve);
06      IGoal play = createGoal("play_song");
07      play.getParameter("song").setValue(retrieve.getParameter("song").getValue());
08      dispatchSubgoalAndWait(play);
09    }
10  }
```

FIGURE 6. Plan head and body example

trigger refers to the "notify_user" goal type it is applicable for each goal instance of that type. The plan body is a Java class named "NotificationPlan" that extends the Jadex framework class "Plan". Inside the mandatory body() method the plan creates and dispatches a "retrieve_song" (lines 3-5) and a "play_song" goal (lines 6-8). The plan will fail when either of the subgoals fail, because subgoal failures raise BDI exceptions that need explicitly to be catched if the plan wants to proceed execution in an error case.

4.2.5. Capabilities. For the purpose of reusability, Jadex supports a flexible module-concept called capabilities [10], which enables the packaging of functionally related entities (beliefs, goals and plans) into a cluster. A capability definition, written as a separate XML document, is therefore very similar to an agent definition, and usually represents a certain application functionality required by several different agents (e.g., a generic negotiation mechanism). A capability provides a separate namespace for the elements contained within, and therefore avoids name-clashes with other capabilities. Agents can be composed of any number of capabilities, that in turn may contain subcapabilities. For advanced settings it is even possible to add or remove single capabilities at runtime.

Each capability exhibits to the superordinated capability a clearly defined interface by distinguishing e.g. between goals or beliefs that can be used from the outside, and those that are only visible to the capability itself. A fundamental difference to the original capability concept of Busetta et al. is that to be used, an element of an inner capability must be explicitly referenced in the scope of the outside capability or agent (see Fig. 7). Any concrete element is internal per

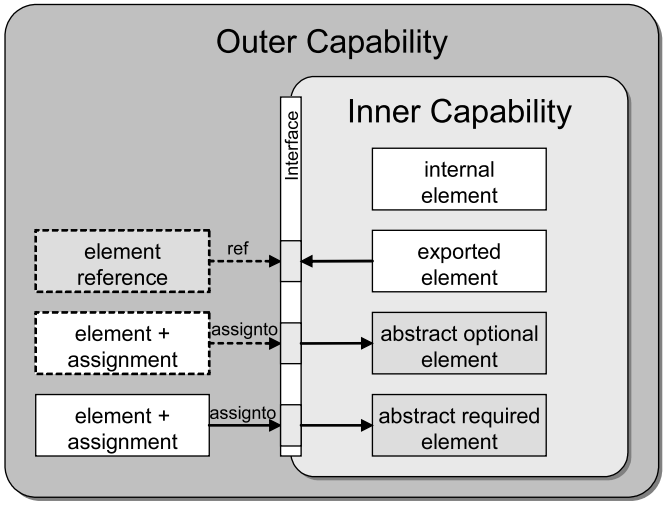

FIGURE 7. Capability concept

default, meaning that it is visible only in the capability it is defined in. For example internal beliefs are only accessible from plans that share the same scope (capability) as the beliefs. To make an element accessible from the outer capability it needs to be exported. Note that this only expresses the possibility to be used from the outer capability. If the outer capability wants to use the exported element it has to explicitly declare this by defining a place-holder (called reference) for the original element. When for example a plan of the outer capability wants to access an exported belief of the inner capability, the outer capability needs to define a belief reference. This reference, acting as a proxy of the original element at runtime, has to be supplied with its own symbolic name. To support usability, for the user (e.g. a plan) it is transparent whether the element is a reference or a concrete element because a unified view for both is provided.

Besides concrete elements a capability may also include abstract elements that either require an element assignment from the outer capability or not (required vs. optional). If an element is abstract and optional the functionality of the enclosing capability does not depend on that element and can be used without an assignment for the element. Otherwise it is mandatory to provide an assignment. E.g. abstract beliefs are a possibility to add knowledge into a capability from the outside. A detailed description about the adapted capability concept can be found elsewhere [7].

4.2.6. Complete Example Agent. In Fig. 8 the complete type declaration of a simple alarm clock agent (as introduced in the last section) is depicted. It has the ability to notify a user at a specified alarm time by playing a song. In the agent tag

```
<agent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://jadex.sourceforge.net/jadex.xsd"
 name="Alarmclock" package="jadex.examples.alarmclock">

 <imports>
  <import>javax.media.MediaLocator</import>
 </imports>

 <beliefs>
  <belief name="alarm_time" class="long">
   <fact>System.currentTimeMillis()+360000</fact>
  </belief>
  <belief name="system_time" class="long" updaterate="1000">
   <fact>System.currentTimeMillis()</fact>
  </belief>
  <belief name="user_notified" class="boolean">
   <fact>false</fact>
  </belief>
 </beliefs>

 <goals>
  <achievegoal name="notify_user" retrydelay="600000" exclude="never">
   <creationcondition>
    $beliefbase.system_time==$beliefbase.alarm_time
   </creationcondition>
   <targetcondition>$beliefbase.user_notified</targetcondition>
  </achievegoal>
  <querygoal name="retrieve_song">
   <parameter name="song_name" class="String"/>
   <parameter name="song" class="MediaLocator" direction="out"/>
  </querygoal>
  <performgoal name="play_song">
   <parameter name="song" class="MediaLocator"/>
  </performgoal>
 </goals>

 <plans>
  <plan name="notify">
   <body>new NotificationPlan()</body>
   <trigger><goal ref="notify_user"/></trigger>
  </plan>
  <plan name="hd_retrieve">
   <body>new HardDiskRetrievePlan()</body>
   <trigger><goal ref="retrieve_song"/></trigger>
  </plan>
  <plan name="web_retrieve">
   <body>new WebRetrievePlan()</body>
   <trigger><goal ref="retrieve_song"/></trigger>
  </plan>
  <plan name="play">
   <body>new PlaySongPlan()</body>
   <trigger><goal ref="play_song"/></trigger>
  </plan>
 </plans>
</agent>
```

FIGURE 8. Example agent definition

(lines 1-3) the type name "Alarmclock" and package name "jadex.examples.alarmclock" are defined. Additionally, the URL to the Jadex schema is declared for validation purposes.

It consists of beliefs (lines 9-19) for the "alarm time" (lines 10-12), the dynamic "system time" (lines 13-15) and a flag indicating if the user has responded to the notification (lines 16-18). The goals section (lines 21-35) contains the top-level goal "notify user" (lines 22-27) which is created when the alarm time has been reached. Additionally, the two subgoal types for retrieving (lines 28-31) and playing a song (lines 32-34) are provided. The plans section (lines 37-54) contains the corresponding plans that are capable of handling the goals. Note, that two different plans are specified to handle a "retrieve song" goal. As no priorities are specified for these plans, the order of declaration determines the execution order. This means, only if the song could not be located on the hard disk (hd_retrieve plan, lines 42-45) it will be tried to load the song from the internet (web_retrieve plan, lines 46-49).

Together with the plan bodies (not shown here), the example provided in this section can directly be executed. Therefore it is only necessary to compile the plan classes with a normal Java compiler. Having started the JADE platform, the Jadex remote monitoring agent (rma) allows for starting Jadex agents simply by selecting agent models (ADFs) from a file-chooser.

4.3. Execution Model

For a complete reasoning engine several different components are necessary. The core of a BDI architecture is obviously the mechanism for plan selection. Plans not only have to be selected for goals, but for internal events and incoming messages as well. To collect the incoming messages and forward them to the plan selection mechanism a specialized component is needed. Another mechanism is required to execute selected plans, and to keep track of plan steps to notice failures. In Jadex, all of the required functionality is implemented in cleanly separated components. The relevant information about beliefs, goals, and plans is stored in data structures accessible to all these components.

Fig. 9 shows the interrelations between those components. The functional elements of the execution model can also be found in the abstract BDI interpreter presented in section 3.2. The difference between Jadex and the abstract interpreter is, that in Jadex these functionalities are carried out independently by three distinct components (message receiver, dispatcher, and scheduler).

The message receiver has the purpose to take messages from the platform's message queue and create Jadex message events which are placed in the event list (similar to the get-new-external-events() operation of the abstract interpreter). The dispatcher continuously consumes the events from the event list and builds the applicable plan list for each event (corresponds to the option-generator() function). This is done by checking for all plans if the considered event or goal triggers the plan and additionally if the plan's pre- and context conditions are valid. Corresponding plans are added to the list of applicable plans. In a subsequent step, the

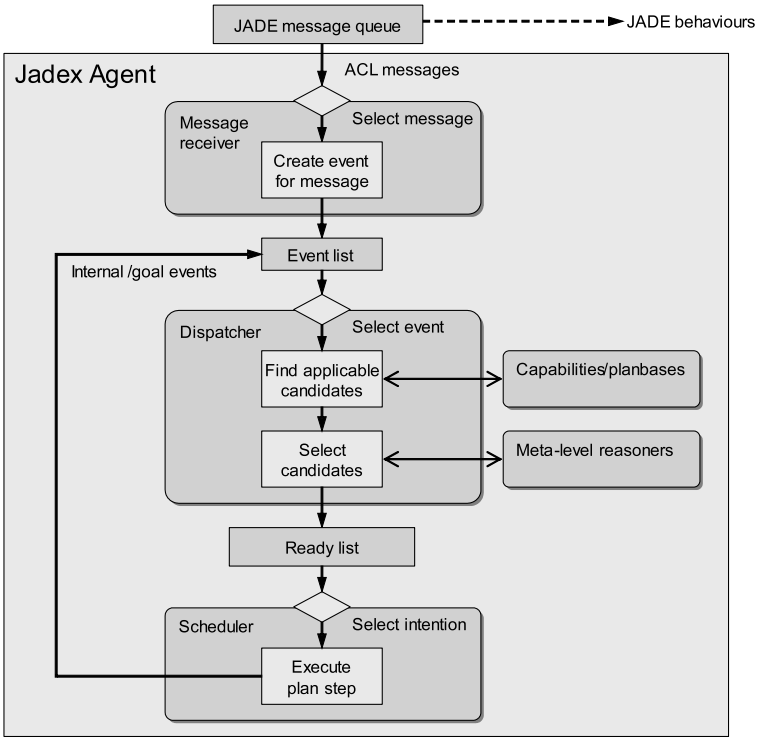

FIGURE 9. Jadex execution model

dispatcher selects plans to be executed (similar to the deliberate(options) operation) by possibly utilizing meta-level reasoning facilities. This means that if more than one plan is principally applicable for the given event or goal the decision process is delegated to a user-defined meta-level reasoning plans. The meta-level reasoner has the task to rank the plan candidates with respect to domain-dependent characteristics.

The selected plans are placed in the ready list after associating the selected plans to the corresponding events or goals, like it is done in update-intentions(selected-options). This makes the plan aware of the goal or event to handle and allows for reading goal or event details from within the plan body.

Finally, the scheduler takes the plans from the ready list and executes them (corresponds to the execute() operation). Thereby, execution of plans is done stepwise, which means that only one plan step of a single plan is executed uninterruptedly. In contrast to the original approach, in which pre-defined actions are used as plan steps, Jadex introduces the notion of *dynamic plan steps,* which will interrupt a plan whenever relevant internal changes occur. Internal changes are considered

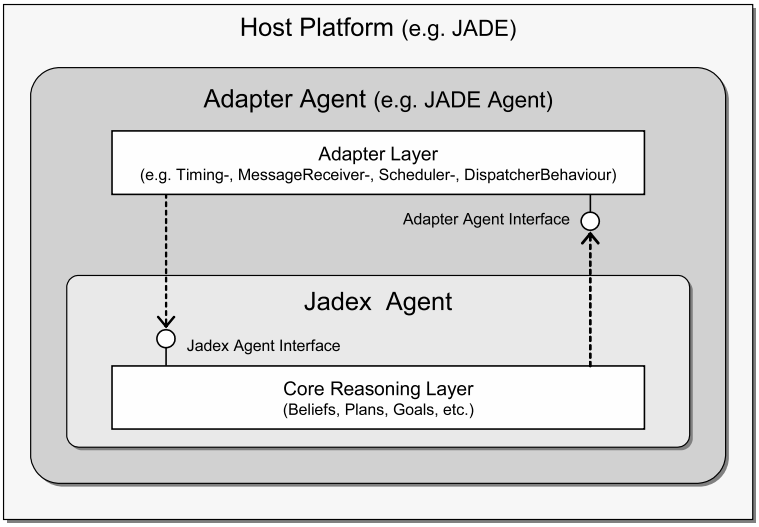

FIGURE 10. Integration mechanism

as relevant when such changes have side-effects, e.g. if a belief change triggers the creation of a new goal.

Note, that the drop-impossible/successful-attitudes() operations of the abstract interpreter are not part of the execution model, because in Jadex those operations are carried out on-the-fly, whenever there are relevant changes in the agent's beliefs.

4.4. JADE Integration

The integration of the Jadex BDI reasoning engine into JADE follows a generic mechanism depicted in Fig. 10. It consists mainly of three distinct layers: The *Host Platform* (here JADE), an *Adapter Agent* and the *Jadex Agent* which encapsulates the BDI reasoning engine. The host platform is only capable to execute agents of a certain type (here JADE agents). Therefore, a Jadex agent has to be wrapped into an adapter agent which is generically done by providing necessary services for the Jadex agent as well as for the adapter agent through small interfaces. Both sides are aware of the other side only in terms of these interfaces to minimize dependencies. In general the *Jadex Agent Interface* offers methods for performing reasoning steps, whereas the *Adapter Agent Interface* provides notification and message sending facilities.

The JADE adapter agent has the purpose to create an instance of the Jadex engine with an agent definition file, which will be interpreted by the Jadex agent to initialize its state. The above mentioned components are implemented in three JADE behaviours of the adapter agent using functionalities from the reasoning layer. In addition, there is a simple timing behaviour with the purpose to add

timeout events to the event list (e.g. when awaited messages do not arrive). Implementing the functionalities into separate behaviours provides a clean design and allows for flexible replacement of the behaviours with custom implementations, e.g. alternative scheduling mechanisms could be tried out, using modified versions of the corresponding behaviours.

The Jadex project facilitates a smooth transition from developing conventional JADE agents to employing the mentalistic concepts of Jadex agents. All available JADE functionality can still be used in Jadex plans. Moreover, it is possible to use some of the Jadex functionality e.g., the belief base or the goal base, from conventional JADE behaviours. To use JADE behaviours in conjunction with Jadex plans the message receiver behaviour supports filtering of incoming ACL messages (see Fig. 9 at the top). It is necessary to sort out those messages which are handled by plans and therefore have to be dispatched to the internal Jadex system and keep the other messages available for the JADE behaviours.

Besides JADE, current work also addresses the integration of Jadex into other host platforms. So far a standalone version and an integration for the DIET platform [22] have been successfully realized. Support for other kinds of middleware such as J2EE is also feasible.

4.5. Tool Support

The tool support for the Jadex BDI reasoning engine mainly focuses on the debugging phase. For the development of Jadex agents ordinary Java IDEs such as Eclipse can be used as plans are written in plain Java. For the creation of agent definition files an XML editor is necessary, e.g. the XML-Buddy plug-in for Eclipse. Provided that a sophisticated XML editor supporting strict schema validation is used, editing becomes very comfortable as auto-completion can be utilized and additionally specification errors are already reported at design time.

For the documentation of agent applications the Jadexdoc tool has been developed (see Fig. 11). It is based on the Javadoc tool and extends it with agent specific characteristics. For all application relevant agent and capability definition files, documentation is generated that summarizes the BDI attitudes and provides links to all used capabilities as well as ordinary Java classes (e.g. a belief class) for which normal Javadocs are provided. Hence the tool enables an integrated view for agent applications consisting of agents and objects.

As a Jadex agent is still a JADE agent, all available tools of JADE can also be used to develop Jadex agents. Most of the JADE platform deals with the external view of an agent, which does not differ between conventional JADE agents and Jadex agents. E.g. the JADE sniffer agent allows for observing agent communications by visualizing the message respective protocol-based interactions and the dummy agent can be used to comfortably enter and send messages. Only the JADE introspector agent is of limited use, because it only shows the four Jadex standard behaviours and not the agent's plans. To enable a comfortable testing of Jadex agents three new tool agents have been developed: the BDI introspector, the logger agent and the tracer tool.

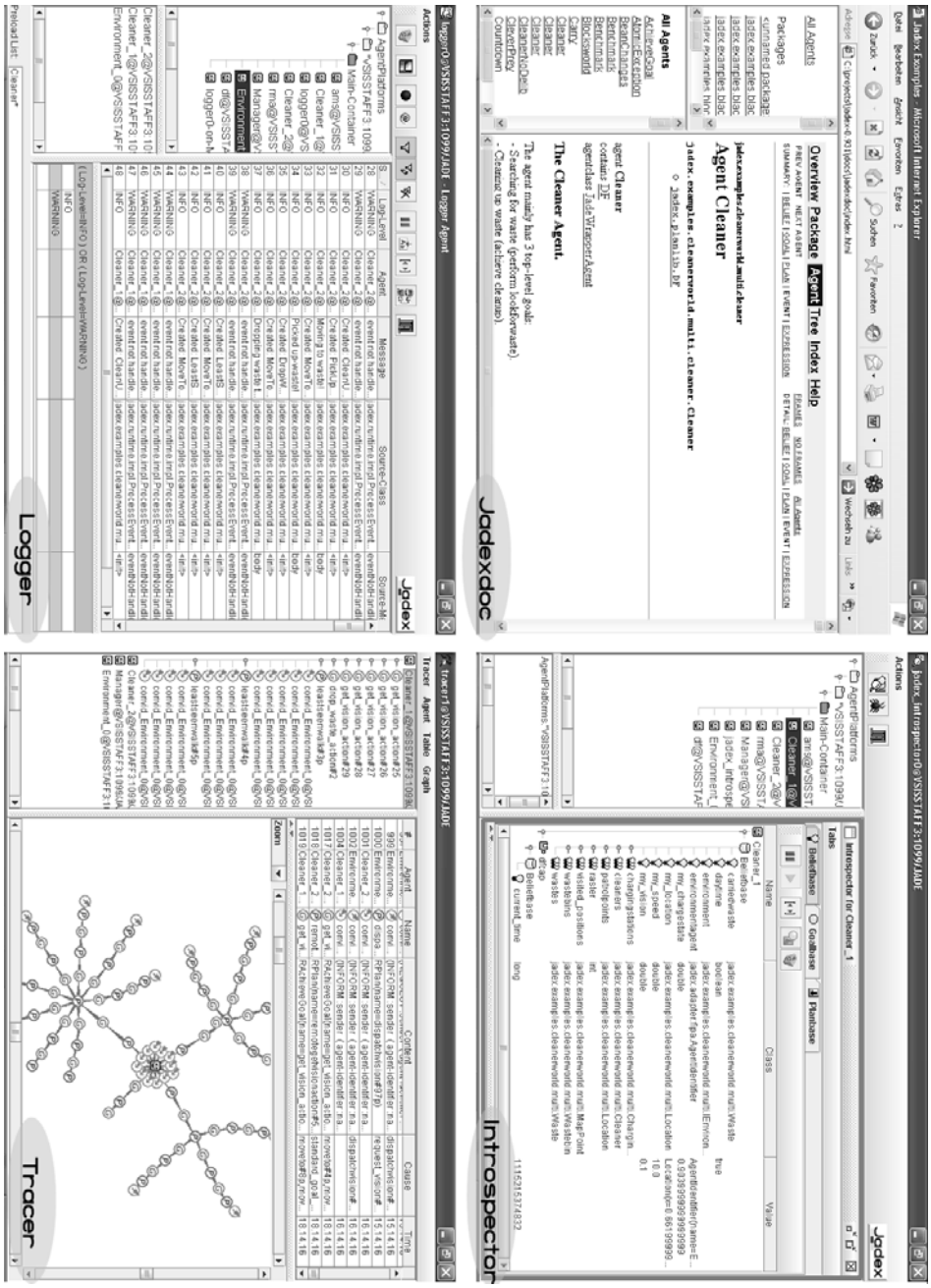

FIGURE 11. Tool screenshots

The introspector's purpose is twofold. First, it supports the visualization and modification of the internal BDI concepts thus allowing inspection and reconfiguration of an agent at runtime. Secondly, it simplifies debugging through a facility for the stepwise agent execution. In the step mode it is possible to observe and control each event processing and plan execution step having detailed control over the dispatcher and scheduler. Hence it can be easily figured out what plans are selected for an event or goal.

A big problem in debugging agent systems consists in the amount and sequence of outputs the agents produce typically on the console. With the help of the logger the agent's outputs can be directed to a single point of responsibility at runtime. In contrast to simple console outputs, the logger agent preserves additional information about the output such as its time stamp and its source (the agent and method). Using these artifacts the logger agent offers facilities for filtering and sorting messages by various criteria allowing a personalized view to be created.

Third tool meant to support the debugging phase is the tracer. Based mainly on ideas from [19] the tool offers the possibility to trace the dynamic behaviour of agents, which means that relevant system changes like reading/writing a belief, sending/receiving a message, pursuing some goal or plan are automatically recorded and displayed. For visualization purposes either a graph structure consisting of interconnected system changes or an agent-centered tree view are available. The tool can inter alia be used to understand why an agent has performed some action (e.g. executed a plan), because the causes for the action are preserved. Additionally, the graph-based visualization can be used to detect unwanted actions (failures) within regular behaviour patterns.

5. Related Work

In Fig. 12, a general overview of several existing agent platforms is given with respect to the dimensions application area (research vs. industrial use) and technical focus (middleware vs. BDI approach).[3] From this classification can be seen that there currently is almost no connection between middleware and BDI systems. Especially for industrial use of agent technology, it is of importance that middleware aspects like interoperability and security as well as aspects for rational decision making are equally well supported. Against this background, a combination of both research strands seems to be a promising approach.

To close the gap between middleware and reasoning two fundamentally different approaches exist. One possibility is to build agent platforms on top of an established industry standard for component oriented software engineering like

[3]References to all depicted agent platforms can be found on the Jadex project page: http://vsis-www.informatik.uni-hamburg.de/projects/jadex/links.php

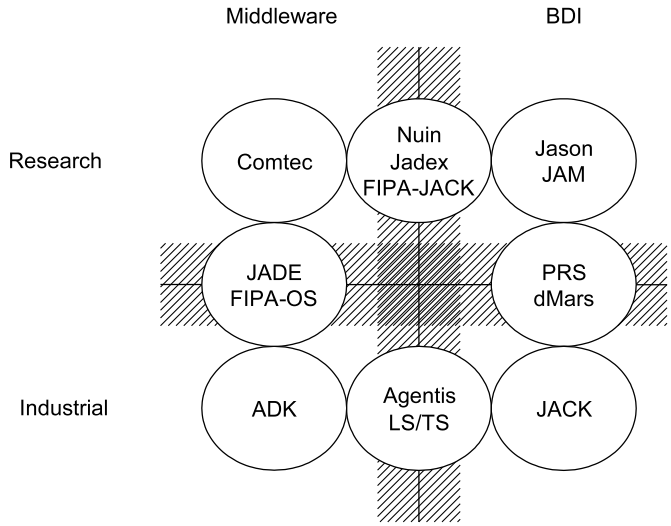

FIGURE 12. Classification of Agent Platforms

Java J2EE and therefore integrate agent technology in application server environments. Typical representatives for this approach are Agentis[4] and Whitestein's LivingSystems technology suite (LS/TS).[5] The other possible approach is based on existing (FIPA-compliant) middleware agent platforms and enhances them with BDI-specific characteristics. Examples for this approach are Nuin [14] and Jadex. In addition, with FIPA-JACK [33] a research approach exists, which enhances the commercial JACK platform with FIPA communication capabilities.

Both integration techniques have different advantages and disadvantages, hence there is not a single predominant solution. General advantages of the application server approach are that industry-grade tools are available and can be utilized to ensure several business critical properties like availability and fault-tolerance. In addition, also development and management tools can be reused to a certain degree. The main drawback of this approach is that it relies on standards for software components that have some similarities with agents, but still need to be adapted to the agent paradigm. On the contrary, using existing agent middleware as the foundation for reasoning has the advantage of being in line with the FIPA-agent standards, but the available tools do not offer the same degree of maturity yet. Due to the primary application domain of Jadex in which FIPA-compliant communication is an essential criterion, Jadex originally took the latter approach and is currently realized as a loosely coupled add-on to a middleware agent platform. Nevertheless, Jadex uses a generic integration mechanism

[4] http://www.agentissoftware.com/
[5] http://www.whitestein.com/

(as described in section 4.4) that allows for flexible adaptation to other kinds of middleware.

Jadex and JACK

Concerning the available BDI-concepts, Jadex has many similarities with the commercial JACK agent platform [17]. Therefore, Jadex will be compared with JACK in the following in more detail.

On the conceptual level the JACK agent platform strictly adopts the BDI interpreter cycle by Rao and Georgeff (see section 3.2) and provides a new agent programming language (JAL) extending Java with BDI-specific file types (agents, capabilities, events, beliefs, plans) and declarative statements. Therefore all of the aforementioned file types including the plans are realized as JACK Framework classes which have to be extended to build an application. JACK programs are compiled to normal Java files with a precompiler and can subsequently be translated to Java classes using the normal Java compiler. In addition to agent-centered BDI concepts, JACK also supports agent teams with the SimpleTeams approach [16]. The runtime infrastructure of JACK consists of an environment for agent execution and proprietary message transport. Management agents for yellow and white pages services are not available. Further on, JACK offers tool support for the development of agents with an integrated development environment (IDE) including a graphical plan editor which allows for visual plan construction. Debugging agent applications is alleviated with runtime tools for stepwise plan execution and observing agent communications.

In contrast to JACK, Jadex does not adhere to the traditional BDI interpreter in a strict manner, but defines separated responsibilities for the important parts of the deliberation cycle. Also different from JACK, Jadex does not define a new agent programming language, but uses a BDI metamodel defined in XML-schema for agent definition and pure Java as implementation language for plans avoiding the need for a precompiler. Jadex supports the same core BDI concepts (except the team concepts) as JACK and additionally introduces several extensions. Most interesting is the extension concerning explicit goal types, which alleviates the disadvantage of treating goals only in the form of simple events [8] and which is the basis for goal deliberation. Because Jadex runs on top of JADE it exhibits all of its middleware features such as FIPA-compliant communication, management agents for yellow and white pages services, security and persistency mechanisms. The same applies for tool support, which means that all of the JADE tools can be used with Jadex agents as well. Furthermore, Jadex provides additional debugging support with the debugger and logger tools, but currently lacks visual tools for agent development.

6. Conclusion and Outlook

This article presents an approach to the integration of an agent middleware with a reasoning engine to combine the advantages of both strands. A motivation for

agent-oriented middleware and an overview of the BDI model was given, and the design and realization of the Jadex BDI engine as an extension to the widely used JADE agent platform was described. The Jadex system allows for the construction of rational agents, which exhibit goal-directed (as opposed to task-oriented) behaviour. The construction of Jadex agents is based on well-established software engineering techniques such as XML, Java and OQL enabling software engineers to quickly exploit the potential of the mentalistic approach. The Jadex project is also seen as a means for researchers to further investigate which mentalistic concepts are appropriate in the design and implementation of agent systems. In addition to its usage in context of the MedPAge project in Hamburg, several other institutes have used Jadex to implement research systems. E.g., the Technical University of Karlsruhe has used Jadex to implement an experimental system for representing norms in multi-agent systems [30] and at the Delft University of Technology, Jadex was used realize a personal travel assistant application [1].

The current version is Jadex 0.931, which can be freely downloaded under LGPL license from the project homepage http://jadex.sourceforge.net/. It is termed a beta stage release, what means that it has reached considerable stability and maturity to be used in practical settings. Ongoing work currently focuses on two aspects of the system: Extensions to internal concepts and additional tool support. On the conceptual level extensions to the basic BDI-mechanisms are developed, such as support for planning, teams, and goal deliberation. In contrast to other BDI agent systems Jadex supports an explicit and declarative representation of goals. It is planned to utilize this explicit representation by improving the BDI architecture with a generic facility for goal deliberation which alleviates the necessity for designing agents with a consistent goal set. Additionally, the explicit representation allows investigating task delegation by considering goals at the inter-agent level.

Work on tools mainly addresses the usability of agent technology as a mainstream software engineering paradigm. The tool support of Jadex currently focuses on the testing phase supplying debugger, logger and tracer agents. To achieve a higher degree of usability it is planned to support the design phase as well with a graphical modeling tool based on the MDA-approach. Additionally, tools for documenting agents and deployment of multi-agent applications are being developed [6].

Acknowledgement

This work is partially funded by the German priority research programme 1083 *Intelligent Agents in Real-World Business Applications*.

References

[1] M. Beelen. Personal Intelligent Travelling Assistant: a distributed approach. Master of science thesis, Knowledge Based Systems group, Delft University of Technology, 2004.

[2] F. Bellifemine, G. Caire, and G. Rimassa. JADE: The JADE platform for mobile MAS applications. In *Net.ObjectDays 2004: AgentExpo*, 2004.

[3] F. Bellifemine, G. Rimassa, and A. Poggi. JADE – A FIPA-compliant agent framework. In *4th International Conference on the Practical Applications of Agents and Multi-Agent Systems (PAAM-99)*, pages 97–108, London, UK, December 1999.

[4] M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers Inc., 2000.

[5] M. Bratman. *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge, Massachusetts, 1987.

[6] L. Braubach, A. Pokahr, K.-H. Krempels, and W. Lamersdorf. Deployment of Distributed Multi-Agent Systems. In *Fifth International Workshop on Engineering Societies in the Agents World (ESAW 2004)*, 2004.

[7] L. Braubach, A. Pokahr, and W. Lamersdorf. Extending the Capability Concept for Flexible BDI Agent Modularization. In *Proceedings of the Third Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS05)*, 2005.

[8] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. Goal Representation for BDI Agent Systems. In *Proceedings of the Second Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS04)*, 2004.

[9] R. Brooks. A Robust Layered Control System For A Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):24–30, March 1986.

[10] P. Busetta, N. Howden, R. Rönnquist, and A. Hodgson. Structuring BDI Agents in Functional Clusters. In N. R. Jennings and Y. Lespérance, editors, *Intelligent Agents VI, Proceedings of the 6th International Workshop, Agent Theories, Architectures, and Languages (ATAL) '99*, pages 277–289. Springer, 2000.

[11] M. Dastani and L. van der Torre. Programming BOID Agents: a deliberation language for conflicts between mental attitudes and plans. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS'04)*, 2004.

[12] M. Dastani, B. van Riemsdijk, F. Dignum, and J.-J. Meyer. A Programming Language for Cognitive Agents: Goal Directed 3APL. In *Proceedings of the First Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS03)*, 2003.

[13] D. Dennett. *The Intentional Stance*. Bradford Books, 1987.

[14] I. Dickinson and M. Wooldridge. Towards practical reasoning agents for the semantic web. Technical Report HPL-2003-99, Hewlett Packard Laboratories, 2003.

[15] Emorphia Limited. *FIPA-OS V2.1.0 Distribution Notes.*, 2001.

[16] A. Hodgson, R. Rönnquist, and P. Busetta. Specification of Coordinated Agent Behavior (The SimpleTeam Approach). In *Proceedings of the Workshop on Team Behaviour and Plan Recognition at IJCAI-99, Stockholm, Sweden*, 1999.

[17] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents - Summary of an Agent Infrastructure. In *Proceedings of the 5th ACM International Conference on Autonomous Agents*, 2001.

[18] F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: A High Level Supervision and Control Language for Autonomous Mobile Robots. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, pages 43–49, Minneapolis, April 1996.

[19] D. Lam and K. Barber. Debugging agent behavior in an implemented agent system. In *Second International Workshop on Programming Multi-Agent Systems at the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems*, pages 45–56, New York, NY, July 20 2004.

[20] J. F. Lehman, J. E. Laird, and P. S. Rosenbloom. A gentle introduction to Soar, an architecture for human cognition. *Invitation to Cognitive Science*, 4, 1996.

[21] E. Mangina. Review of Software Products for Multi-Agent Systems. http://www.agentlink.org/resources/software-report.html, 2002.

[22] P. Marrow. The DIET project: building a lightweight, decentralised and adaptable agent platform. *AgentLink News*, 12:3–6, April 2003.

[23] A. Newell. *Unified Theories of Cognition*. Harvard University Press, 1990.

[24] 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 Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*. Morgan Kaufmann, 2003.

[25] 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.

[26] A. Pokahr and L. Braubach. *Jadex User Guide, Release 0.921*, 2004.

[27] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: Implementing a BDI-Infrastructure for JADE Agents. *EXP – in search of innovation*, 3(3):76–85, 2003.

[28] S. Poslad and P. Charlton. Standardizing Agent Interoperability: The FIPA Approach. In M. Luck et al., editor, *9th ECCAI Advanced Course, ACAI 2001 and Agent Links 3rd European Agent Systems Summer School, EASSS 2001, Prague, Czech Republic, July 2001*, pages 98–117. Springer-Verlag: Heidelberg, Germany, 2001.

[29] A. Rao and M. Georgeff. BDI Agents: from theory to practice. In V. Lesser, editor, *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 312–319, San Francisco, CA, USA, 1995. The MIT Press: Cambridge, MA, USA.

[30] T. Schubert. Normen zur Überwachung und Steuerung autonomer Multi-Agenten Systeme. Diplomarbeit, Institut für Programmstrukturen und Datenorganisation, Fakultät für Informatik, Universität Karlsruhe (TH), 2004. (in German).

[31] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[32] Tryllian Solutions B.V. *The Developer's Guide*, 2004.

[33] K. Yoshimura. FIPA JACK: A plugin for JACK Intelligent Agents. Technical report, RMIT University, 2003.

Information about Software

Software is available on the Internet as:

( ) prototype version
(x) full fledged software (freeware), version no.: 0.931
( ) full fledged software (for money), version no.:
( ) Demo/trial version
( ) not (yet) available

Internet address:

Description of software: http://jadex.sourceforge.net
Download address: http://sourceforge.net/projects/jadex

Contact person for question about the software:

Name: Lars Braubach / Alexander Pokahr
email: {braubach — pokahr}@informatik.uni-hamburg.de

Lars Braubach, Alexander Pokahr and Winfried Lamersdorf
Distributed and Information Systems Group
Computer Science Department, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg
Germany
e-mail: {braubach, pokahr, lamersd}@informatik.uni-hamburg.de