Jadex: A BDI-Agent System Combining Middleware and Reasoning

Lars Braubach, Alexander Pokahr and Winfried Lamersdorf

Abstract. Nowadays a whole bunch of different agent platforms exists 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 agent framework is presented, which supports reasoning by exploiting the BDI model and is realized as an extension to the widely used JADE middleware platform.

1. Introduction

Nowadays a whole bunch of different agent platforms exists that aim to support the software engineer in developing multi-agent systems [16]. 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[1] issues that address interoperability and various infrastructure topics such as white and yellow page services. Hence agent middleware is an important buiding 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. 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.

[1] http://www.fipa.org

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 and combines it with state-of-the-art software engineering techniques 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

Agent orientation builds on concepts and technology of distributed systems. 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) [22] ist an international nonprofit organization providing standards for heterogeneous interacting agents and multiagent systems. Since 1997 a number of specifications have been released which are replaced or updated frequently. The work on specifications focuses on application as well as 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 sematic layer, to provide a unified basis for agent communication and interaction.

The FIPA specifications have been implemented in a number of agent platforms 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

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

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 to decompose the overall agent behaviour into smaller pieces and attach 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 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 architectures 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, the theory of Agent Oriented Programming (AOP) [25], the Unified Theories of Cognition (UTC leading to SOAR) [17, 15] and the subsumption theory [6]. 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.

3.1. BDI Foundations

The BDI model was conceived by Bratman as a theory of human practical reasoning [3]. Its success is based on its simplicity reducing the explanation framework for complex human behaviour to the *motivational stance* [10]. This means that the causes for actions are always related to the human desires ignoring other facets of human recognition 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 maybe cannot 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 [9, 8]) 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

Algorithm 1 BDI-interpreter, taken from [23]

```
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
```

are executed [5]. 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.

3.2. BDI Realization

Foundation for most implemented BDI systems is the abstract interpreter proposed by Rao and Georgeff (see algorithm 1) [23]. 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 [14], 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 [21]). 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, the project investigates the advantages of using agent technology in the context of hospital logistics [18, 19]. 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 [2] 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 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 already running plans, or to new plans instantiated 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 is generally the same for all agents. The behaviour of a specific agent is therefore determined solely by its concrete beliefs, goals,

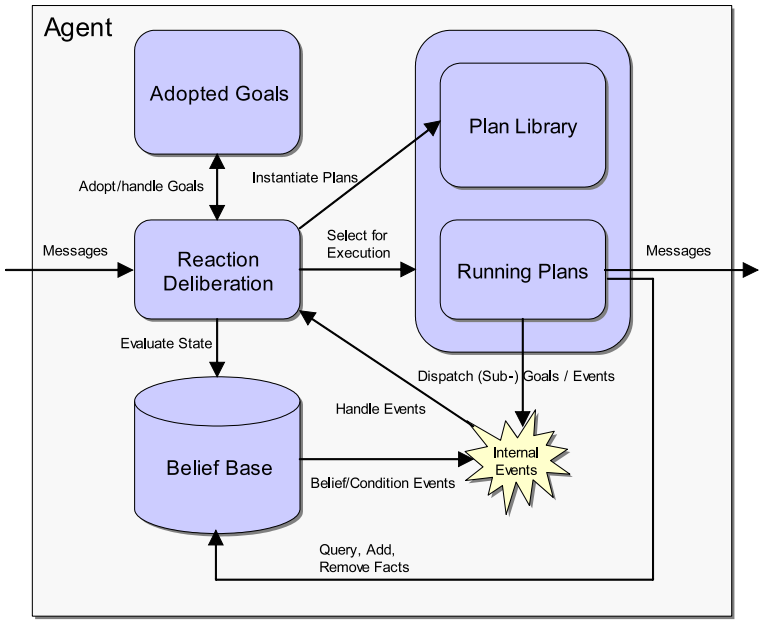

FIGURE 1. Jadex abstract architecture

and plans. In the following each of these central concepts of the Jadex BDI architecture will be described in detail.

4.2.1. 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 to reuse classes generated by ontology modelling 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 removing facts. A more declarative way of accessing beliefs and beliefsets is provided by queries, which can be specifed in an OQL[3]-like language. 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.

4.2.2. Goals. Jadex follows the general idea that goals are concrete, momentary desires of an agent. 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) goals and actively

[3]Object Query Language, see http://www.odmg.org

pursued goals, a goal lifecycle is introduced which consists of the goal states *option*, *active*, and *suspended* [5]. 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). A sophisticated goal deliberation mechnism is not yet available, therefore currently the Jadex engine automatically activates all valid 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 [5].

4.2.3. 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 ad-hoc planning 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.

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

4.3. 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

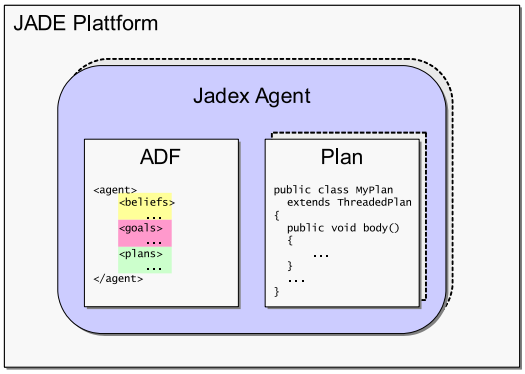

FIGURE 2. Composition of a Jadex agent

meta-model (described in [20]) specified in XML schema [4] 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 e.g., default arguments for launching the agent or service descriptions for registering the agent at a directory facilitator.

4.3.1. Capabilities. For the purpose of reusability Jadex supports a flexible module-concept called capabilities [7], 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. 3). This reference, that acts as a proxy of the original element at runtime, is conceptually treated as a first level element with its own name.

The explicit declaration of references induces some specification overhead compared to the original capability concept, but offers several advantages. The concept of a capability as a *scope* for elements is cleaner on the conceptual and implementation

[4] http://www.w3.org/XML/Schema

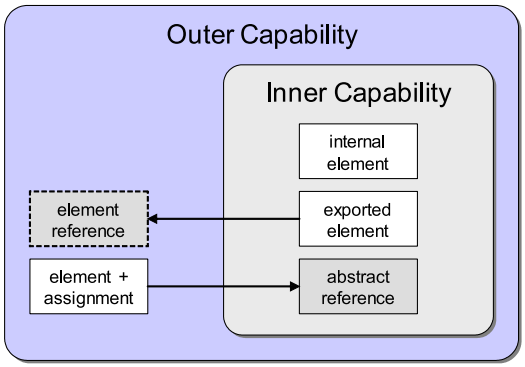

FIGURE 3. Capability concept

level, as there are only relationships between elements of the same capability. For example, a plan will only handle goals and events of its own scope. Expressions, used e.g. in queries or goal parameters only access beliefs defined locally in the capability. Because the references to elements of inner capabilities are specified in the declarative XML documents, and not inside Java plan code, consistency can be checked at design time (e.g., if referenced beliefs exist and have the correct type). The usage of referenced elements is transparent to the plan programmer. References can be accessed in the same way as locally defined elements of the same type. This also means, that one does not need to know, which capability actually implements a referenced element.

4.3.2. Example Agent. In Fig. 4 an example for an agent definition file is depicted. It shows the type declaration of a simple translation agent that can translate words from English to German. In the agent tag (lines 4-7) the type name "ta" and package name "jadex.examples.tutorial" are defined. Additionally the URL to the Jadex schema is declared for validation purposes. For reasons of simplicity this agent only consists of one plan, one beliefset, and one expression (predefined query).

The plan declaration (lines 14-17) is used to define under which circumstances (the filter tag) an intention (plan instance) is created for a declared plan body (the constructor tag). In this case the filter object is defined as return value of a static method invocation (line 16) and hence it is necessary to inspect this method to reveal that whenever the agent receives a message containing a translation request, a new plan instance of the Java class "EnglishGermanTranslationPlan" is created. This plan uses the agent's personal dictionary stored as belief set "egwords" (lines 21-26) to figure out the translation of a word. Therefore the translation plan applies the predefined query with the English word as parameter (line 30-34) to find the adequate German word.

In Fig. 5 the corresponding plan body code is depicted (lines 9-28). Most of this code is used for testing the message format and extracting the content. The extracted English word is supplied as parameter for the query that fetches the translated word (line 17).

```
<!--
  A simple translation agent for translating words from English to German.
-->
<agent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://jadex.sourceforge.net/jadex.xsd"
  name="ta"
  package="jadex.examples.tutorial">

<imports>
  <import>jadex.util.*</import>
</imports>

<plans>
  <plan name="egtrans">
    <constructor>new EnglishGermanTranslationPlan()</constructor>
    <filter>EnglishGermanTranslationPlan.getEventFilter()</filter>
  </plan>
</plans>

<beliefs>
  <beliefset name="egwords" class="Tuple">
    <fact>new Tuple("milk", "Milch")</fact>
    <fact>new Tuple("cow", "Kuh")</fact>
    <fact>new Tuple("cat", "Katze")</fact>
    <fact>new Tuple("dog", "Hund")</fact>
  </beliefset>
</beliefs>

<expressions>
  <expression name="query_egword">
    SELECT ANY $wordpair.get(1)
    FROM $wordpair in $beliefbase.egwords
    WHERE $wordpair.get(0)==$eword
  </expression>
</expressions>

</agent>
```

FIGURE 4. Example agent definition file

4.4. 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. 6 shows the interrelations between those components. The functional elements of the execution model can also be found in the abstract BDI interpreter presented in

```
01 package jadex.examples.tutorial;
02
03 import ...
04
05 /** Plan for translating an English word to German.
06 * Requires the following message format: translate english_german <eword>.*/
07 public class EnglishGermanTranslationPlanB2 extends ThreadedPlan {
08
09  /** The plan body. */
10  public void body() {
11   StringTokenizer stok = new StringTokenizer(
12   ((RMessageEvent)getInitialEvent()).getMessage().getContent(), " ");
13   if(stok.countTokens()==3) {
14    stok.nextToken();
15    stok.nextToken();
16    String eword = stok.nextToken();
17    String gword = (String)getQuery("query_egword").execute("eword", eword);
18    if(gword!=null) {
19     System.out.println("Translating from english to german: "+eword+" - "+gword);
20    }
21    else {
22     System.out.println("Sorry, word is not in database: "+eword);
23    }
24   }
25   else {
26    System.out.println("Sorry, format not correct.");
27   }
28  }
29
30  /** Get the event filter. */
31  public static IFilter getEventFilter() {
32   MessageTemplate mt1 = MessageTemplate.MatchPerformative(ACLMessage.REQUEST);
33   MessageTemplate mt2 = new MessageTemplate(
34   new MatchStartContentLiteral("translate english_german"));
35   return new MessageFilter(MessageTemplate.and(mt1, mt2));
36  }
37 }
```

FIGURE 5. Example agent translation plan

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, scheduler). The message receiver performs the get-new-external-events() operation, by taking ACL messages from the platform's message queue and creating Jadex events which are placed in the event list. The dispatcher continuously consumes the events from the event list and builds the applicable plan list for each event, corresponding to the option-generator() function. The dispatcher also selects plans to be executed - similar to deliberate(options) - and places the selected plans in the ready list after associating the selected plans to the corresponding events or goals, like it is done in update-intentions(selected-options). Finally the scheduler takes the plans from the ready list and executes them, as done by the execute() operation. Note, that the drop-impossible/successful-attitudes() operations are not part of the execution model, because

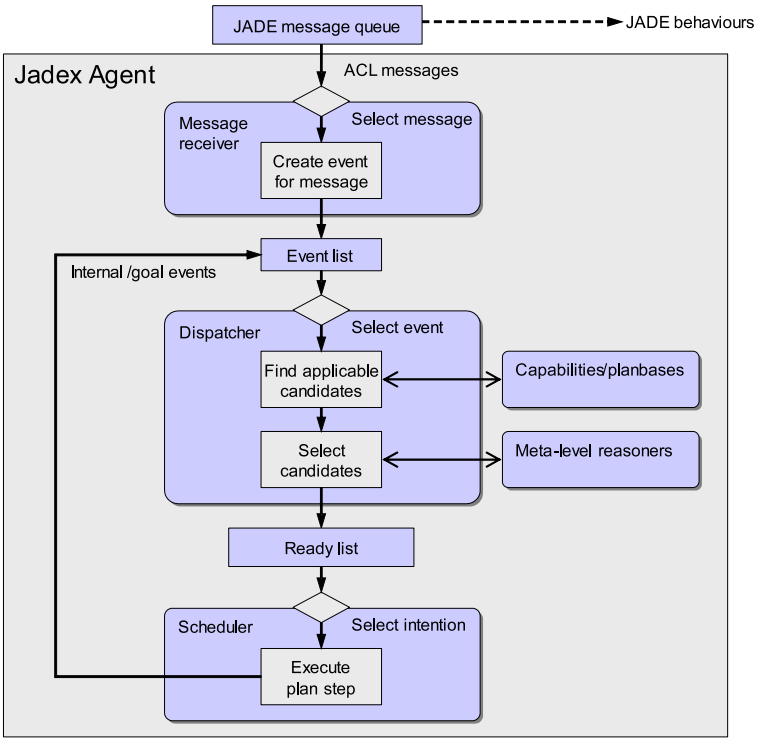

FIGURE 6. Jadex execution model

in Jadex those operations are carried out on-the-fly, whenever there are relevant changes in the agent's beliefs.

4.5. JADE Integration

To easily integrate the Jadex engine into JADE agents, a wrapper agent class is provided, which creates and initializes an instance of the Jadex engine with the beliefs, goals and plans from an agent definition file. The above mentioned components of the reasoning engine are implemented in three JADE behaviours, which are automatically created and added to the wrapper agent. 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

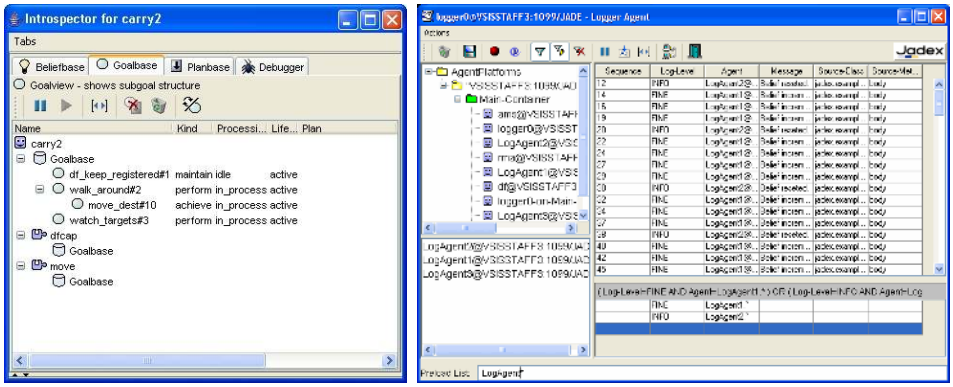

FIGURE 7. BDI introspector and logger screenshots

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. 6 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.

4.6. Tool Support

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. 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 two new tool agents have been developed: the BDI introspector and the logger agent.

The introspector's purpose is twofold. First, it supports the visualization and modification of the internal BDI concepts (see Fig. 7 left hand side) 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 artefacts the logger agent offers facilities for filtering and sorting messages by various criteria allowing a personalized view to be created (see Fig.7 right hand side).

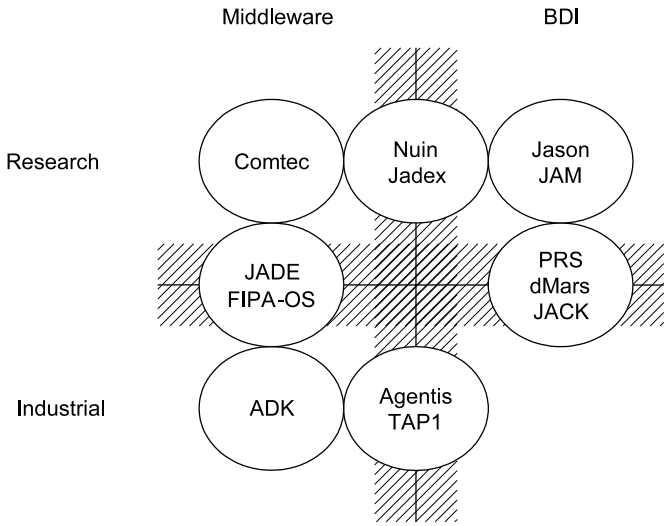

FIGURE 8. Classification of Agent Platforms

5. Related Work

In Fig. 8 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).[5] 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 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 Java J2EE and therefore integrate agent technology in application server environments. Typical representatives for this approach are Agentis[6] and Whitesteins TAP1.[7] 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 [11] and Jadex.

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

[5] References to all depecited agent platforms can be found on the Jadex project page:
http://vsis-www.informatik.uni-hamburg.de/projects/jadex/links.php
[6] http://www.agentissoftware.com/
[7] http://www.whitestein.com/pages/index.html

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 is 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 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 took the latter approach and is currently realized as a loosely coupled add-on to a middleware agent platform.

Jadex and JACK

Concerning the available BDI-concepts, Jadex is most similar to the commercial JACK agent platform [13]. 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 declaratice 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 [12]. The runtime infratructure 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 developement 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 traditinal 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 interestingly is the extension concerning explicit goal types, which alleviates the disadvantage of treating goals only in the form of simple events [5] and which is the basis for goal deliberation. Because Jadex is based on 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 [24] and at the Delft University of Technology, Jadex was used realize a personal travel assistant application [1].

The current version is Jadex 0.921, which can be freely downloaded under LGPL license[8] 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 experimental settings, but compatibility between releases is not guaranteed. 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 to investigate 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 focusses on the testing phase supplying a debugger and a logger agent. 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.[9] Additionally, tools for documenting agents and deployment of multi-agent applications are being developed [4].

Acknowledgement

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

[8] http://www.gnu.org/copyleft/lesser.html
[9] Model-Driven Architecture, see http://www.omg.org

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. 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.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

[11] I. Dickinson and M. Wooldridge. Towards practical reasoning agents for the semantic web. Technical Report HPL-2003-99, Hewlett Packard Laboratories, May 15 2003.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] A. Pokahr and L. Braubach. *Jadex User Guide*, 2003.

[21] 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.

[22] 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.

[23] 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. The MIT Press: Cambridge, MA, USA, 1995.

[24] 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).

[25] Y. Shoham. Agent-oriented programming. In D. G. Bobrow, editor, *Artificial Intelligence Volume 60*, pages 51–92, Elsevier Amsterdam, The Netherlands, 1993.

Lars Braubach
Distributed and Information Systems Group
Computer Science Department, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg
Germany
e-mail: braubach@informatik.uni-hamburg.de

Alexander Pokahr
Distributed and Information Systems Group
Computer Science Department, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg
Germany
e-mail: pokahr@informatik.uni-hamburg.de

Winfried Lamersdorf
Distributed and Information Systems Group
Computer Science Department, University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg
Germany
e-mail: lamersd@informatik.uni-hamburg.de