Jadex: A Short Overview

L. Braubach,[1] A. Pokahr,[1] W. Lamersdorf[1]

Distributed and Information Systems Group, University of Hamburg, Germany
`{braubach, pokahr, lamersd}@informatik.uni-hamburg.de`

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 realised 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 [10]. 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. 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 adapt to the agent paradigm.

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

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

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 supports easy to use reasoning capabilities by exploiting the BDI model combining it with state-of-the-art software engineering techniques like XML and Java. In section 2 reasoning approaches for agents are sketched and the BDI fundamentals regarding the individual concepts and their interrelationships are described. Section 3 explains the design and implementation of the Jadex system by detailing the abstract architecture and its integration into the JADE platform. A summary and an outlook describing ongoing work and planned extensions conclude the paper.

2 Reasoning for Agents

To build agents with cognitive capabilities several architectures from different disciplines like psychology, philosophy and biology can be utilised. 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) [16], the Unified Theories of Cognition (UTC leading to SOAR) [11,9] and the subsumption theory [4]. 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.

2.1 BDI Foundations

The BDI model was conceived by Bratman as a theory of human practical reasoning [2]. Its success is based on its simplicity reducing the explanation framework for complex human behaviour to the *motivational stance* [8]. 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 psychologcial notions that closely correspond to they way humans talk about behavioural aspects.

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

Algorithm 1 BDI-interpreter, taken from [15]

```
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
```

referred to as *goals*). Actual systems and formal theory mostly ignore this step (with the exception of 3APL [7,6]) 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 maintaind, and therefore the reasons, why actions are executed [3]. 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.

2.2 BDI Realisation

Foundation for most implemented BDI systems is the abstract interpreter proposed by Rao and Georgeff (see Fig. 1) [15]. 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, the authors feel that it should nowadays 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 appraoch.

3 Jadex Architecture and Implementation

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 [1] 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 realise 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.

The following sections present the architecture and execution model of the newly developed reasoning engine (see also [14]). 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.

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

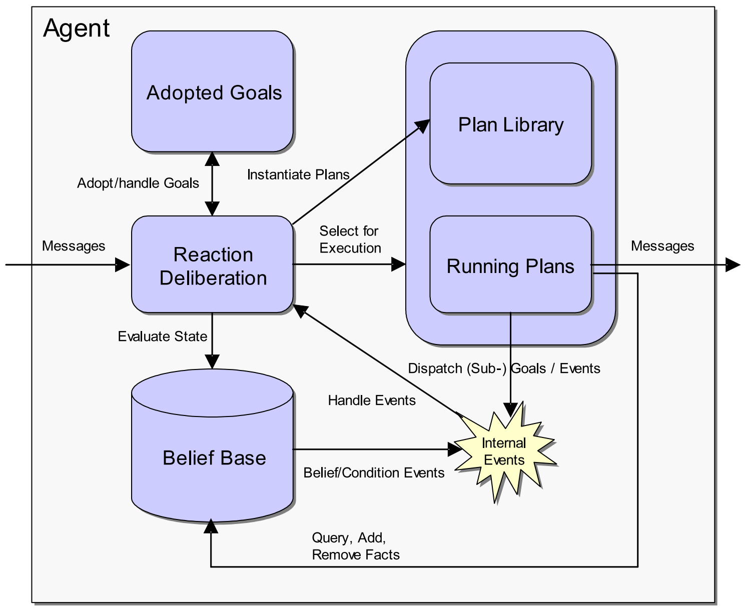


Fig. 1. Jadex abstract architecture

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[2]-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.

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 pursued goals, a goal lifecycle is introduced which consists of the goal states *option*, *active*, and *suspended* [3].

[2] www.odmg.org

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 options. Additionally, 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 [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, Jadex uses 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. 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.

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. 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 using an XML lan-

```
<!--
 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>
```

Fig. 2. Example agent definition file

guage that follows the Jadex meta-model specified in XML schema[3] and allows for creating Jadex objects in a declarative way. For the purpose of reusability Jadex supports a flexible module-concept called capability [5], which enables the packaging of functionally related entities (beliefs, goals and plans) into a cluster. Capabilities exhibit a clearly defined interface and therefore can be nested allowing an agent being composed of predefined functionalities. Within the XML capability or 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.,

[3] www.w3.org/XML/Schema

```
package jadex.examples.tutorial;

import ...

/** Plan for translating an English word to German.
* Requires the following message format: translate english_german <eword>.*/
public class EnglishGermanTranslationPlanB2 extends ThreadedPlan {

 /** The plan body. */
 public void body() {
  StringTokenizer stok = new StringTokenizer(
  ((RMessageEvent)getInitialEvent()).getMessage().getContent(), " ");
  if(stok.countTokens()==3) {
   stok.nextToken();
   stok.nextToken();
   String eword = stok.nextToken();
   String gword = (String)getQuery("query_egword").execute("eword", eword);
   if(gword!=null) {
    System.out.println("Translating from english to german: "+eword+" - "+gword);
   }
   else {
    System.out.println("Sorry, word is not in database: "+eword);
   }
  }
  else {
   System.out.println("Sorry, format not correct.");
  }
 }

 /** Get the event filter. */
 public static IFilter getEventFilter() {
  MessageTemplate mt1 = MessageTemplate.MatchPerformative(ACLMessage.REQUEST);
  MessageTemplate mt2 = new MessageTemplate(
  new MatchStartContentLiteral("translate english_german"));
  return new MessageFilter(MessageTemplate.and(mt1, mt2));
 }
}
```

Fig. 3. Example agent translation plan

default arguments for launching the agent or service descriptions for registering the agent at a directory facilitator.

In Fig. 2 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 belief, and one expression.

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. 3 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).

3.2 Execution Model

For a complete reasoning engine several additional 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 specialised 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. 4 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 2.2. 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 in Jadex those operations are carried out on-the-fly, whenever there are relevant changes in the agent’s beliefs.

3.3 JADE Integration

To easily integrate the Jadex engine into JADE agents, a wrapper agent class is provided, which creates and initialises 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,

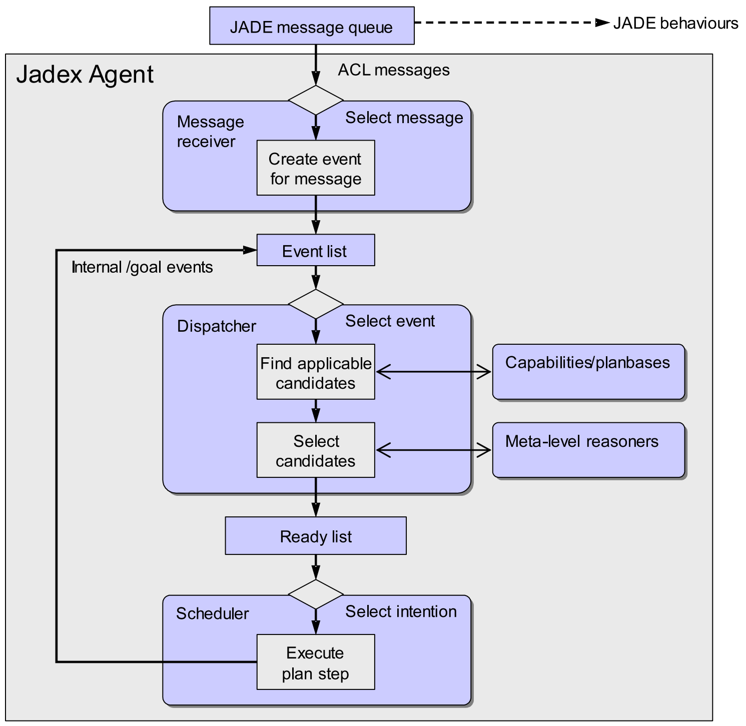


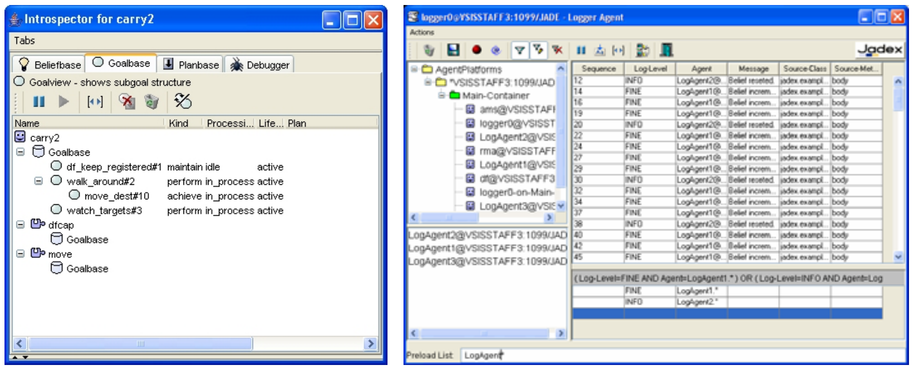
Fig. 4. Jadex execution model

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 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. 4 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.

3.4 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 debugger and the logger agent.



Name	Kind	Processi...	Life...	Plan
carry2				
Goalbase				
df_keep_registered#1	maintain	idle	active	
walk_around#2	perform	in_process	active	
move_dest#10	achieve	in_process	active	
watch_targets#3	perform	in_process	active	
dfcap				
Goalbase				
move				
Goalbase				

```
AgentPlatforms
  "VSISSTAFF3:1099/JAD
    Main-Container
      ams@VSISSTAFF
      logger0@VSISST
      LogAgent2@VSIS
      rma@VSISSTAFF
      LogAgent1@VSIS
      df@VSISSTAFF3
      logger0-on-Main-
      LogAgent3@VSIS
```

```
LogAgent2@VSISSTAFF3:1099/JAD
LogAgent1@VSISSTAFF3:1099/JAD
LogAgent3@VSISSTAFF3:1099/JAD
```

Sequence	Log-Level	Agent	Message	Source-Class	Source-Met...
12	INFO	LogAgent2@...	Belief reseted.	jadex.exampl...	body
14	FINE	LogAgent1@...	Belief increm...	jadex.exampl...	body
16	FINE	LogAgent1@...	Belief increm...	jadex.exampl...	body
19	FINE	LogAgent1@...	Belief increm...	jadex.exampl...	body
20	INFO	LogAgent2@...	Belief reseted.	jadex.exampl...	body
22	FINE	LogAgent1@...	Belief increm...	jadex.exampl...	body
24	FINE	LogAgent1@...	Belief increm...	jadex.exampl...	body
27	FINE	LogAgent1@...	Belief increm...	jadex.exampl...	body
29	FINE	LogAgent1@...	Belief increm...	jadex.exampl...	body
30	INFO	LogAgent2@...	Belief reseted.	jadex.exampl...	body
32	FINE	LogAgent1@...	Belief increm...	jadex.exampl...	body
34	FINE	LogAgent1@...	Belief increm...	jadex.exampl...	body
37	FINE	LogAgent1@...	Belief increm...	jadex.exampl...	body
38	INFO	LogAgent2@...	Belief reseted.	jadex.exampl...	body
40	FINE	LogAgent1@...	Belief increm...	jadex.exampl...	body
42	FINE	LogAgent1@...	Belief increm...	jadex.exampl...	body
45	FINE	LogAgent1@...	Belief increm...	jadex.exampl...	body

(Log-Level=FINE AND Agent=LogAgent1.*) OR (Log-Level=INFO AND Agent=Log

	FINE	LogAgent1.*			
	INFO	LogAgent2.*			

Fig. 5. Debugger and logger screenshots

The debugger's purpose is twofold. First, it supports the visualization and modification of the internal BDI concepts (see Fig. 5 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 personalised view to be created (see Fig.5 right hand side).

4 Conclusion and Outlook

This article presented an approach to the integration of an agent middleware with a reasoning engine to combine the advantages of both strands. 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. Future improvements will address mainly two directions: architecture and tool-support.

In contrast to other BDI agent systems Jadex supports an explicit and declarative representation of goals. We plan 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 will allow us to investigate task delegation by considering goals at the inter-agent level. 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 is is planned to support the design phase as well with a graphical modeling tool based on the MDA-approach.[4]

Currently the system is used as the basis of the research project MedPAge [12,13], which deals with agent-based management of hospital logistics. Additionally, the system is used in several internal teaching and some third-party projects. The current version 0.921 of the Jadex system is available at the projects home page http://jadex.sourceforge.net.

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

References

1. 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.
2. M. Bratman. *Intention, Plans, and Practical Reason.* Harvard University Press, Cambridge, Massachusetts, 1987.
3. 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.
4. R. Brooks. A Robust Layered Control System For A Mobile Robot. *IEEE Journal of Robotics and Automation*, 2(1):24–30, March 1986.
5. 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.

[4] http://www.omg.org

6. 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.
7. 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.
8. D. Dennett. *The Intentional Stance*. Bradford Books, 1987.
9. 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.
10. E. Mangina. Review of Software Products for Multi-Agent Systems. http://www.agentlink.org/resources/software-report.html, 2002.
11. A. Newell. *Unified Theories of Cognition*. Harvard University Press, 1990.
12. 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.
13. 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.
14. 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.
15. 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.
16. Y. Shoham. Agent-oriented programming. In D. G. Bobrow, editor, *Artificial Intelligence Volume 60*, pages 51–92, Elsevier Amsterdam, The Netherlands, 1993.