Chapter 6

JADEX: A BDI REASONING ENGINE

Alexander Pokahr,[1] Lars Braubach,[1] and Winfried Lamersdorf[1]

[1] *University of Hamburg*
Distributed Systems and Information Systems
22527 Hamburg, Germany
{*pokahr|braubach|lamersd*}*@informatik.uni-hamburg.de*

Abstract This chapter presents Jadex, a software framework for the creation of goal-oriented agents following the belief-desire-intention (BDI) model. The Jadex project aims to make the development of agent based systems as easy as possible without sacrificing the expressive power of the agent paradigm. The objective is to build up a rational agent layer that sits on top of a middleware agent infrastructure and allows for intelligent agent construction using sound software engineering foundations. Fostering a smooth transition from traditional distributed systems to the development of multi-agent systems, well established object-oriented concepts and technologies such as Java and XML are employed wherever applicable. Moreover, the Jadex reasoning engine tries to overcome traditional limitations of BDI systems by introducing explicit goals. This allows goal deliberation mechanisms being realized and additionally facilitates application development by making results from goal-oriented analysis and design easily transferable to the implementation layer. The system is freely available under LGPL license and provides extensive documentation as well as illustrative example applications.

Keywords: BDI agents, FIPA standard, object-oriented software engineering, explicit goals.

6.1 Motivation

Today, a numerousness of different agent platforms is available for developing multi-agent applications [144]. Nevertheless, most of these platforms are developed with a specific technological focus such as the cognitive or infrastructural architecture. Hence, not all aspects of agent technology are

covered equally well. General applicability of an agent platform for a great variety of domains demands that at least three categories of requirements are considered: openness, middleware, and reasoning. Openness is closely related to the vision of interconnected networks of originally unrelated applications whereas middleware aspects emphasize traditional software engineering concerns such as service management, security and persistency aspects. Reasoning, in turn, focuses on the agent's internal decision-making process and mostly tries to map this process from a natural archetype such as insects or humans.

According to these aspects, the existing platforms can be classified into two almost distinct groups. On the one hand, FIPA-compliant platforms mainly address openness and middleware issues by realizing the FIPA communication respectively platform standards [172]. On the other hand, reasoning-centered platforms exist, that focus on the behaviour model of a single agent, e.g. trying to achieve rationality and goal-directedness. This gap between middleware and reasoning-centered systems is one main motivation for the realization of the Jadex BDI (Belief-Desire-Intention) reasoning engine [30, 171], which aims to bring together both research strands.

Besides this overall objective to support both classical virtues from middleware and BDI reasoning, the design of the system is driven by two main factors. On the one hand, the development of the reasoning engine is accompanied by an ongoing effort of enhancing the BDI architecture in general. The system addresses shortcomings of earlier BDI agent systems, e.g. by providing an explicit representation of goals and a systematic way for the integration of goal deliberation mechanisms. On the other hand, the system respects the current state of the art regarding mainstream object-oriented software engineering, and is designed to be used not only by AI experts, but also by the normally skilled software developer. Therefore, agent development is based on established techniques such as Java and XML, and is further supported by software engineering aspects, such as reusable modules and development tools.

6.2 Architecture

This section presents the architectural underpinnings of the Jadex system. It starts with a short review of the BDI model and related systems. Subsequently, an overview of the architecture of Jadex is presented. The basic concepts – beliefs, goals, and plans – of the system are introduced by highlighting their main characteristics and differences to other BDI agent systems. Finally, the execution model is shortly sketched, showing how the components of the system interoperate.

6.2.1 BDI Models and Systems

The BDI model was initially conceived by Bratman as a theory of human practical reasoning [28]. Its success is based on its simplicity reducing the explanation framework for complex human behavior to the *motivational stance* [58]. In this model, causes for actions are only related to desires ignoring other facets of 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 communicate about human behavior [157].

The BDI theory of Rao and Georgeff [182] defines beliefs, desires, and intentions as mental attitudes represented as possible world states. The intentions of an agent are subsets of the beliefs and desires, i.e., an agent acts towards some of the world states it desires to be true and believes to be possible. To be computationally tractable Rao and Georgeff also proposed several simplifications to the theory, the most important one being that only beliefs are represented explicitly. Desires are reduced to events that are handled by predefined plan templates, and intentions are represented implicitly by the runtime stack of plans to be executed.

According to Martha Pollack [96], work on BDI can be further subdivided into three categories: 1. General models for practical reasoning, based on BDI concepts. 2. Computational models based on the "Intelligent Resource-Bounded Machine Architecture" (IRMA) [27], exhibiting close correspondence to Bratman's philosophy. 3. The computational model employed in the PRS family of systems [98, 118], which found many uses in practice. Nowadays, current descendants of the PRS family, in particular commercial products and solutions such as Agent Oriented Software's JACK 7 and Agentis' AdaptivEnterprise Suite [127] have the most practical relevance concerning development of agent-based software systems.

In the next sections, the architecture of the Jadex reasoning engine, which basically follows the PRS computational model, will be described. Important differences to other representatives of the PRS family will be highlighted in the corresponding subsections.

6.2.2 Concepts within Jadex

In Fig. 6.1 an overview of the abstract Jadex architecture is presented. Viewed from the outside, an agent is a black box, which receives and sends messages. As common in PRS-like systems, all kinds of events, such as incoming messages or goal events serve as input to the internal reaction and deliberation mechanism, which dispatches the events to plans selected from the plan library. In Jadex, the reaction and deliberation mechanism is the only global component of an agent. All other components are grouped into reusable modules called capabilities.

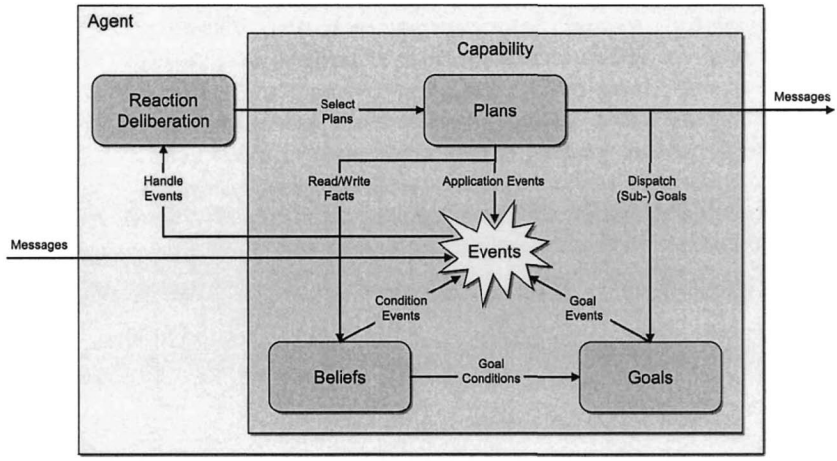

Figure 6.1. Jadex abstract architecture

Beliefs

One objective of the Jadex project is the adoption of a software engineering perspective for describing agents. In other BDI systems, beliefs are represented in some kind of first-order predicate logic (e.g. Jason, described in chapter 1) or using relational models (e.g. JACK and JAM [114]). In Jadex, an object-oriented representation of beliefs is employed, where arbitrary objects can be stored as named facts (called beliefs) or named sets of facts (called belief sets). Operations against the beliefbase can be issued in a descriptive set-oriented query language. Moreover, the beliefbase is not only a passive data store, but takes an active part in the agent's execution, by monitoring belief state conditions. Changes of beliefs may therefore directly lead to actions such as events being generated or goals being created or dropped.

Goals

Goals are a central concept in Jadex, following 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 wanted any more. In other PRS-like systems, goals are represented by a special kind of event. Therefore, in these systems the current goals of an agent are only implicitly available as the causes of currently executing plans. In Jadex, goals are represented as explicit objects contained in a goalbase, which is accessible to the reasoning

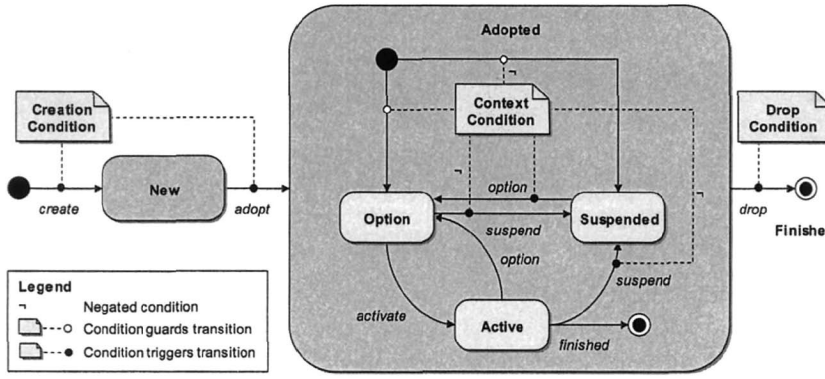

Figure 6.2. Goal lifecycle (from [32])

component as well as to plans if they need to know or want to change the current goals of the agent. Because goals are represented separately from plans, the system can retain goals that are not currently associated to any plan. As a result, unlike other BDI systems, Jadex does not require that all adopted goals are consistent to each other, as long as only consistent subsets of those goals are pursued at any time. To distinguish between just adopted and actively pursued goals, a goal lifecycle is introduced which consists of the goal states *option*, *active*, and *suspended* (see Fig. 6.2). When a goal is adopted, it becomes an option that is added to the agent's goalbase, either as top-level goal, or when created from a plan as subgoal of a plan's root goal. Application specific goal deliberation settings specify dependencies between goals, and are used for managing the state transitions of all adopted goals (i.e. deciding which goals are active and which are just options). In addition, 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.

Jadex supports four types of goals, which extend the general lifecycle and 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 world state 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, but the desired state is not a state of the (outside) world, but an internal state of the agent, regarding the availability of some information the agent wants to know about.

For goals of type *maintain*, an agent keeps track of a desired state, and will continuously execute appropriate plans to re-establish this maintained state whenever needed. More details about goal representation and processing in Jadex can be found in [32].

Plans

Plans represent the behavioural elements of an agent and are composed of a head and a body part. The plan head specification is similar to other BDI systems and mainly specifies the circumstances under which a plan may be selected, e.g. by stating events or goals handled by the plan and preconditions for the execution of the plan. Additionally, in the plan head a context condition can be stated that must be true for the plan to continue executing. The plan body provides a predefined course of action, given in a procedural language. This course of action is to be executed by the agent, when the plan is selected for execution, and may contain actions provided by the system API, such as sending messages, manipulating beliefs, or creating subgoals.

Capabilities

Capabilities, introduced in [39], represent a grouping mechanism for the elements of a BDI agent, such as beliefs, goals, plans, and events. In this way, closely related elements can be put together into a reusable module, which encapsulates a certain functionality (e.g. for interaction with a FIPA directory facilitator). The enclosing capability of an element represents its scope, and an element only has access to elements of the same scope (e.g. a plan may only access beliefs or handle goals or events of the same capability). To connect different capabilities, flexible import / export mechanisms can be used that define the external interface of the capability (e.g. beliefs or goals visible to the outside).

6.2.3 Execution Model

This section shows the operation of the reaction and deliberation component, given the Jadex BDI concepts as described earlier. All of the required functionality is assigned to cleanly separated components, which will be explained in turn. Incoming messages are placed in the agent's global message queue by the underlying agent platform such as JADE (see chapter 5). Before the message can be forwarded to the system, it has to be assigned to a capability, which is able to handle the message. If the message belongs to an ongoing conversation, an event for the incoming message is created in the capability executing the conversation. Otherwise, a suitable capability has to be found, which is done by matching the message against event templates defined in

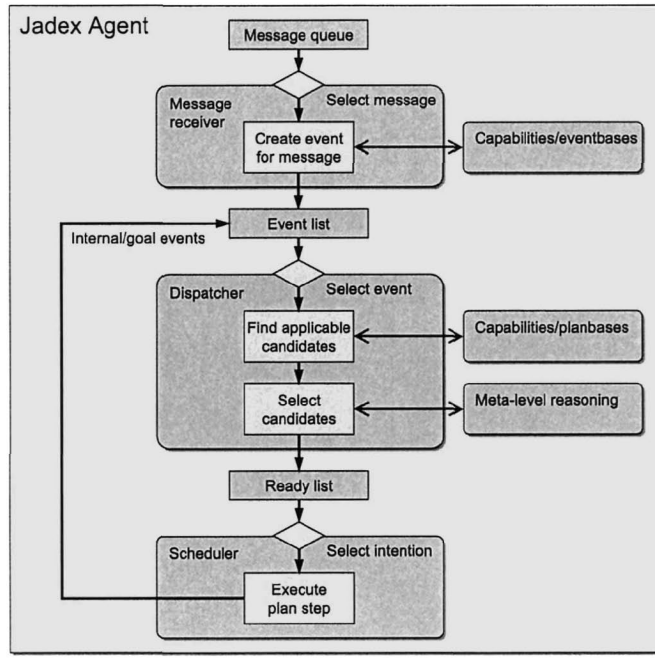

Figure 6.3. Jadex execution model

the eventbase of each capability. The best matching template is then used to create an appropriate event in the scope of the capability. In either case, the created event is subsequently added to the agent's global event list.

The dispatcher is responsible for selecting applicable plans for the events from the event list. This is done in two steps: First, a list of applicable plans is generated by matching the event against the plan heads as defined in the planbases of each capability, whereby only those capabilities have to be considered, where the event is visible. The second step is to select a subset of the applicable plans for execution. Regarding this step several important questions arise, such as if all of the applicable plans should be executed concurrently, or if the event is posted to another plan if the first plan fails [39]. The decision of which plan to execute is called meta-level reasoning and may be as simple as selecting the first plan from the list, or as complicated as finding and executing meta-plans for the decision. Jadex provides flexible settings to influence this event processing individually for event types and instances. As a default, messages are posted to only one single plan, while for goals, many plans are executed sequentially until the goal is reached or fi-

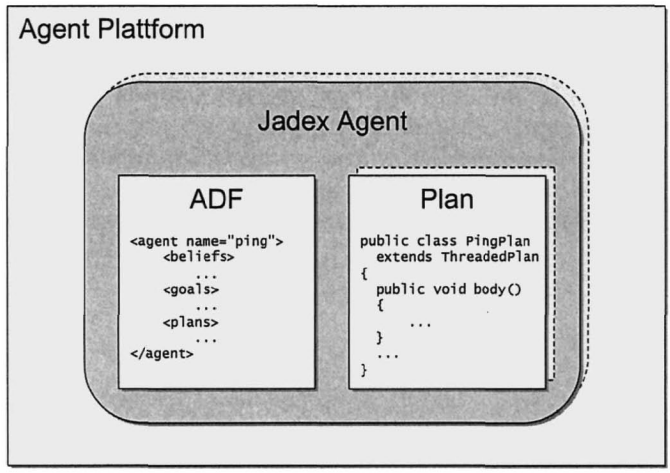

Figure 6.4. Jadex agent

nally failed, when no more plans are applicable. Internal events are posted to all plans at once, as they are considered only as a change notification and no return value is expected from executed plans. After plans have been selected, they are placed in the ready list, waiting for execution.

The execution of plans is performed by a scheduler, which selects the plans from the ready list. Plans are executed step-by-step, whereby (in contrast to other PRS-like systems) the length of plan step depends on the context, and not only on the plan itself. A plan is executed only until it waits explicitly or significantly affects the internal state of the agent (e.g. by creating or dropping a goal). Internal state changes can be caused directly or through side effects, e.g. when a belief change triggers the creation condition of a goal. After the plan waits or is interrupted, the state of the agent can be properly updated, e.g. a newly created goal might lead to other plans being scheduled.

6.3 Language

Jadex is neither based on a new agent programming language nor does it employ or revise an existing one. Instead, a hybrid approach is chosen, distinguishing explicitly between the language used for static agent type specification and the language for defining the dynamic agent behaviour. According to this distinction, a Jadex agent consists of two components: An agent definition file (ADF) for the specification of inter alia beliefs, goals and plans as

well as their initial values and on the other hand procedural plan code (see Fig. 6.4). For defining ADFs, an XML language is used that follows the Jadex BDI metamodel specified in XML Schema. The XML structure specification is augmented by a declarative expression language, e.g. for specifying goal-conditions. The procedural part of plans (the plan bodies) are realized in an ordinary programming language (Java) and have access to the BDI facilities of an agent through an application program interface (API).

6.3.1 Specifications and Syntactical Aspects

The Jadex BDI metamodel defined in XML Schema is very extensive and hence cannot be presented completely in this paper (for a complete introduction see [170]). Generally, the corresponding language was specified with two design principles in mind. The first design objective is the support for strong typing and explicit representation of all kinds of elements, be it beliefs, goals or events. In consequence, this requires users to write detailed ADFs, but in return allows for more rigorous consistency checking of agent models. Additionally, at runtime certain kinds of failures can be discovered more easily, e.g. the attempt of storing a fact value in an undefined belief can be immediately reported.

The second design objective regards increasing the expressive power of the ADF for the following purposes: The arbitrary complex creation of objects (e.g. values within beliefs or parameters), the description of boolean conditions (e.g. when a certain goal should be dropped) and the construction of queries (e.g. for retrieving values from the beliefbase). To achieve this, an embedded expression language is used for specifying parts of the agent model, not easily represented in XML. Expressions are used throughout the XML ADF, whenever values have to be obtained for certain elements at runtime, e.g. values of beliefs, conditions of goals, etc. Expressions should be side effect free, because they are often evaluated internally by the system. The expression language has been designed to fully comply with the syntax of Java expressions (right hand side of assignments) extended with a subset of OQL (object query language) instructions [15]. The syntax of the OQL extension is depicted in Fig. 6.5 in EBNF notation. It allows for query statements being created in the well-known *select-from-where* form, whereby it can be additionally specified if exactly one (iota), the first satisfying (any) or all satisfying results are expected (line 1). In the *from* clause (lines 3–4) it is specified from which object set (line 4) or joined sets (line 3) results are generated. The identifiers define variables, which iterate over the object sets specified as arbitrary expressions. These iterated values are checked against the boolean *where* condition (line 6) and can possibly be ordered (line 7). The example query, corresponding to the example presented in section

```
01: select_expression ::= "SELECT" ("ALL" | "ANY" | "IOTA")?
02: (
03:   expression "FROM" ("$" identifier "IN" expression) ("," "$" identifier "IN" expression)*
04:   | "$" identifier "FROM" expression
05: )
06: ("WHERE" expression)?
07: ("ORDER" "BY" expression ("ASC" | "DESC")? )?


Example: SELECT $block FROM $beliefbase.blocks WHERE $block.isClear()
```

Figure 6.5. OQL syntax in EBNF and query example

6.3.3, shows that it is possible to use Java method calls like isClear() in the expression language. While queries can be used in any expression, they are most useful for predefined views on subsets of the agent's beliefs, which can be evaluated at runtime (e.g. from within plans).

In the following the essential BDI concepts as presented in Section 6.2.2 will be taken on and their realization on language level will be detailed. These concepts are specified as part of an agent or capability description in the same manner. In Fig. 6.6 (left hand side) the allowed attributes and subtags of the agent tag are shown. Each agent type is identified by a name and package declaration and can be provided with a description text. In addition, the corresponding agent class and runtime properties can be set. For most cases, the default values are sufficient and need not be modified. It can be seen that besides the subtags for the core BDI concepts (beliefs, goals, plans and events which are explained below) several other elements can be declared. Most of these elements (languages, ontologies, servicedescriptions and agentdescriptions) are FIPA related and facilitate agent communication respectively the interaction with yellow page services. The remaining elements (imports, expressions, properties) are implementation details, serving for convenience (e.g. to avoid duplicate declarations) and agent configuration purposes, such as logging or debugging settings.

Beliefs

In Jadex, beliefs are represented in an object-oriented way allowing arbitrary Java objects being stored as facts. Like all elements of a capability, beliefs and belief sets can be supplied with a name, a description text and an exported flag. Exporting an element makes it accessible from the outer scope (respectively a capability or an agent) and is turned off by default. For beliefs and belief sets, the Java class for facts must be defined. Besides the type-relevant information, initial fact data can also be supplied for configur-

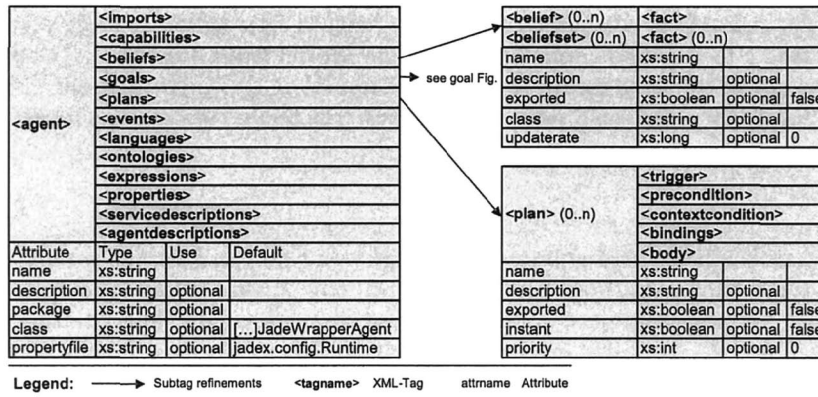

Figure 6.6. Agent metamodel specification fragment (XML-schema)

ing an agent's mental state at creation time. The value of a fact has to be stated in the expression language and can be declared as static or dynamic, whereby dynamic facts are useful e.g. for representing values continuously sensed from an environment or time-relevant aspects. Re-calculation of such dynamic facts occurs on access and additionally in fixed time intervals (using the update rate). At runtime, beliefs and belief sets are accessible from within plans via operations on the beliefbase and additionally by issuing OQL-like queries.

Goals

As described earlier in Jadex four different goal types are distinguished (perform, achieve, maintain and query). All these goal types are based on the generic life cycle and hence exhibit many common properties that are summarized in an abstract base goal type (see Fig. 6.7). According to the lifecycle, creation, drop and context conditions can be specified as boolean expressions. Customization of goal types can be further achieved by defining named in-, out- and inout-parameters that are used to transfer information between a goal's originator and its processing plans. Additionally, binding parameters can be used for generating one goal instance for every possible binding. The runtime processing of goals can be refined using the various BDI-flags, which inter alia control if a goal is retried when a plan fails (retry), if meta-level reasoning is used (mlreasoning) and if applicable plans are tried sequentially or in parallel (posttoall). A complete explanation can be found in [170].

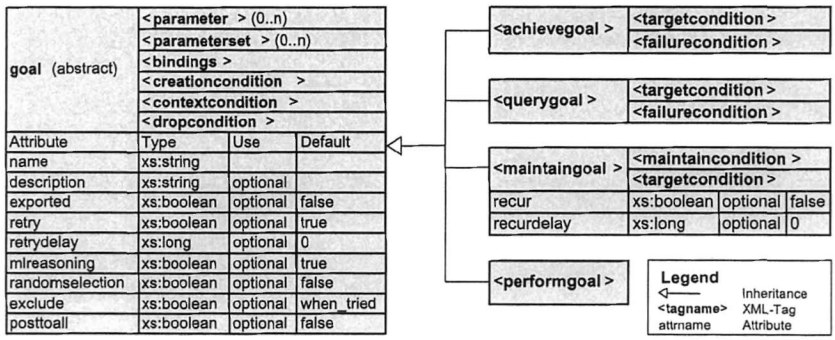

Figure 6.7. Goal metamodel specification (XML-schema)

From the abstract goal type, all concrete types are derived. The simplest one being the perform goal used for executing (possibly repeatedly) certain actions, which does not require extra specification data. An achieve goal extends this abstract goal type and adds support for the specification of a target and a failure condition. The target condition is used for describing the world state this goal seeks to bring about as a boolean expression. Similarly, a boolean failure condition has the purpose to abort goal processing in case its achievement has become impossible. The query goal provides the same kind of conditions, but exhibits a slightly different behaviour in that it is used for information retrieval purposes.

Most complex behaviour is exposed by the maintain goal type, which is used to monitor a specific world state (maintain condition) and automatically tries to reestablish this state whenever it becomes invalid. A boolean target condition can be used to refine the state that is tried to be restored. Maintain goals are not dropped when they are achieved once, but remain inactive until the monitored state is violated again. Moreover, a maintain goal can be configured to retry re-establishment in certain time intervals (recur and recurdelay), when it has failed for some reason. In addition to the specification of the four types of goals, possibly parametrized initial goals can be declared that will be created when the agent is born. At runtime, goal instances can be created from within plans by referring to their type name. Typically, some parameter values need to be supplied before a goal can be dispatched as top-level goal or as subgoal of the current plan.

Plans

The declaration of plans in Jadex is very similar to other PRS-like systems and requires the specification of the plan heads describing the circumstances under which a plan is applicable in the ADF. As plan trigger, internal events, messages, and goals, as well as a belief state condition (for data driven plans) can be provided. The pre- and context condition of a plan can be specified as boolean expressions. To facilitate goal achievement with plans, it is sometimes advantageous to create several different parametrized plan instances of a plan type and try them one after another until a plan succeeds. For this purpose, binding parameters can be specified and used for plan configuration. Furthermore, the selection of which plan is executed in response to an occurring trigger can be adjusted by setting a priority value. As part of the initial mental state of an agent, it can be further declared whether a plan is instantiated when the agent is created (using the instant flag).

The plan body needs to be supplied as expression for the creation of a suitable plan instance. Currently, two different types of plan bodies (standard and mobile) are supported, which both require a Java class to be implemented. Mobile plan bodies have several disadvantages compared to the standard versions, but nonetheless make sense in mobile scenarios as agent migration is provided. In Fig. 6.8 the skeleton of an application plan is depicted. Mandatory is only the extension of a corresponding framework class (Plan) and the implementation of the abstract body() method, in which the domain-specific plan behaviour can be placed. In addition to the body method, three other methods exist that optionally can be implemented. These methods are called when plan processing has finished according to the plans final state. The passed() method is called when the body method completes, whereas the failed() method is invoked when an uncatched exception is thrown 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 or when the plans root goal is dropped.

6.3.2 Software Engineering Issues

The overall goal of the Jadex project is to provide a sophisticated reasoning engine allowing to develop arbitrary complex intelligent agents. Therefore, while trying to be as easily useable as possible, the system does not sacrifice expressiveness for simplicity. Nonetheless, software engineering issues play an important role in the design of the system.

As stated earlier, a primary goal of the project is to facilitate a smooth transition from mainstream object-oriented software development to an agent-oriented approach. This is achieved by resorting to established techniques

```
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 6.8. Plan skeleton

wherever possible. E.g., the system builds on Java and XML, therefore the developer does not have to learn a new language. Another advantage is that the developer can continue to operate in a familiar environment. As the agent developer only has to create Java and XML files, existing development environments such as Eclipse[1] or IntelliJ IDEA[2] can be used to develop Jadex agents. In recent editions of these environments, features such as on-the-fly checking and auto-completion not only apply to Java coding but can also easily be adopted for XML ADF creation,[3] therefore offering extensive support for Jadex agent development.

Moreover, the system provides advanced software engineering features, such as reusability and consistency checking. The capability concept allows encapsulating agent functionality into a reusable module while maintaining the abstraction level of BDI elements. The explicit specification and strong typing of beliefs, goals, etc. facilitates consistency checks of ADFs to detect errors (e.g. spelling mistakes) as early as possible.

6.3.3 Example

To further explain the syntax and semantics of the Jadex agent languages, in this section a simple example is provided. The example does only cover a small subset of the features of Jadex. Another example covering all different types of goals can be found elsewhere [32]. The example presented

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

[2] http://www.jetbrains.com/idea/

[3] In eclipse this can be realized by the XMLBuddy plug-in (see .http://xmlbuddy.com/).

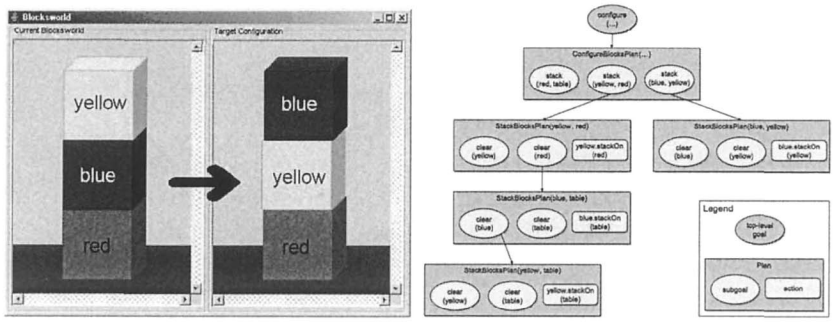

Figure 6.9. Blocksworld scenario (left) and goal/plan tree (right)

here is a fully functional agent, taken directly from the current Jadex distribution. The purpose of the agent is to establish given configurations in a blocksworld environment, where colored blocks are placed in stacks on top of a table. The example provides a graphical user interface, where the user can visually create custom block configurations (see Fig. 6.9, left hand side). The configurations have to be established by the agent by moving the blocks. As only clear blocks (without other blocks on top) can be moved, the agent has to perform some ad-hoc planning. The implemented solution is very simple, creating the stacks bottom-to-top. Fig. 6.9 (right hand side) shows the planning process. To achieve the target configuration, subgoals are created to stack the red block on the table, the yellow block on the red, and the blue block on the yellow (see ConfigureBlocksPlan). To stack two blocks on each other, a StackBlocksPlan clears both blocks and performs the stackOn action. To clear a block, all obstructing blocks are moved to the table.

The ADF of the agent is shown in Fig. 6.10, where tags (elements of the Jadex metamodel) are in boldface, and embedded expressions are in italics. The model starts with the declaration of the agent tag, specifying the name and package of the agent (line 1). The package is used as first place to resolve references to other files such as capabilities and Java classes. More packages and files can be explicitly specified in the imports section (lines 2-4). In this case the class java.awt.Color is imported, because it is used to represent the color of a block.

The beliefs of the agent are given in the beliefs section (lines 6-16). A belief "table" (lines 7-9) is used to represent the environment, which consists of a table on which blocks are located. As initial fact of the belief, an instance of the Table class (located in package jadex.examples.blocksworld) is created (line 8). The known blocks are collected in a belief set "blocks" (lines 10-15).

```
01: <agent name="Blocksworld" package="jadex.examples.blocksworld">
02:   <imports>
03:     <import>java.awt.Color</import>
04:   </imports>
05:
06:   <beliefs>
07:     <belief name="table" class="Table">
08:       <fact>new Table()</fact>
09:     </belief>
10:     <beliefset name="blocks" class="Block">
11:       <fact>new Block(new Color(240,16,16),$beliefbase.table)</fact>
12:       <fact>new Block(new Color(16,16,240),$beliefbase.table.allBlocks[0])</fact>
13:       <fact>new Block(new Color(240,240,16),$beliefbase.table.allBlocks[1])</fact>
14:       ...
15:     </beliefset>
16:   </beliefs>
17:
18:   <goals>
19:     <achievegoal name="clear">
20:       <parameter name="block" class="Block" />
21:       <targetcondition>$goal.block.isClear()</targetcondition>
22:     </achievegoal>
23:     <achievegoal name="stack">
24:       <parameter name="block" class="Block" />
25:       <parameter name="target" class="Block" />
26:       <targetcondition>$goal.block.lower==$goal.target</targetcondition>
27:     </achievegoal>
28:     <achievegoal name="configure">
29:       <parameter name="configuration" class="Table" />
30:       <targetcondition>
31:         $beliefbase.table.configurationEquals($goal.configuration)
32:       </targetcondition>
33:     </achievegoal>
34:   </goals>
35:
36:   <plans>
37:     <plan name="stack">
38:       <body>new StackBlocksPlan($event.goal.block, $event.goal.target)</body>
39:       <trigger><goal ref="stack"/></trigger>
40:     </plan>
41:     <plan name="configure">
42:       <body>new ConfigureBlocksPlan($event.goal.configuration)</body>
43:       <trigger><goal ref="configure"/></trigger>
44:     </plan>
45:     <plan name="clear">
46:       <bindings>
47:         <binding name="upper">
48:           select $upper from $beliefbase.blocks where $upper.lower==$event.goal.block
49:         </binding>
50:       </bindings>
51:       <body>new StackBlocksPlan($upper, $beliefbase.table)</body>
52:       <trigger><goal ref="clear"/></trigger>
53:     </plan>
54:   </plans>
55: </agent>
```

Figure 6.10. Blocksworld agent model

A number of blocks (class Block) with different colors is initially created given by single fact items (lines 11, 12, 13 ...). The first block is created on the table, while the other blocks are created on top of each other (referenced by table.allBlocks[]).

The agent has three achieve goals, each with a name, parameters and a corresponding target condition (lines 18-34). The "clear" goal (lines 19-22) represents the goal to clear (i.e. remove blocks located on top) a block given in a parameter (line 20). The target condition (line 21) refers directly to the isClear() method of this block. The "stack" goal (lines 23-27) aims at placing a given block (line 24) on a target block (line 25). Achieving this goal means that the block below the first block is now equal to the target block as stated by the target condition (line 26). To establish a complete configuration of blocks on the table, the "configure" goal (lines 28-31) is used. The desired configuration is given as a parameter of type Table (line 29). The target condition (line 30-32) refers to the configurationEquals() method implemented in the Table class. No initial instances of these three goal types are defined in the model. The agent starts idle, waiting for goals to appear, which are created by the user through a GUI.

The goals are handled by the plans of the agent (lines 36-54). In this example, there is one plan for each goal, although this kind of one to one mapping is not required. The plan head declarations of the first two plans "stack" (lines 37-40) and "configure" (lines 41-44) are straightforward. The trigger (lines 39 and 43) defines when the plan is applicable, in this case for goals of type "stack" and "configure", respectively. The body (lines 38, 42) defines how the plan body object is instantiated. In both cases, the creation expression refers to parameters of the triggering goal to supply the arguments for the Java constructor (cf. Figs. 6.11, 6.12). The "clear" plan definition is more complex, as the body of the "stack" plan is reused (see line 51) to move all blocks from the top of the block to be cleared to the table. To resolve the parameters used for body creation, a bindings declaration is used (lines 46-50). The variable $upper is assigned to all blocks located on top of the given block (select statement in line 48). For each of these variable assignments an instance of the plan is created, assuring that all blocks are removed from the given block.

The Java files of the two plan bodies are shown in Figs. 6.11 and 6.12, respectively. References to classes and methods provided by the Jadex engine are shown in boldface. Both plan classes define a constructor which takes the plan arguments and stores them in corresponding fields (lines 6-12 respectively 6-10) such that they are accessible from the body() methods, which will be described in turn.

The body() method of the StackBlocksPlan (Fig. 6.11, lines 14-24) first clears both blocks provided as arguments, and then moves the first block on

```
01: package jadex.examples.blocksworld;
02: import jadex.runtime.*;
03:
04: /** Plan to stack one block on top of another target block. */
05: public class StackBlocksPlan extends Plan {
06:   protected Block block;
07:   protected Block target;
08:
09:   public StackBlocksPlan(Block block, Block target) {
10:     this.block = block;
11:     this.target = target;
12:   }
13:
14:   public void body() {
15:     IGoal clear = createGoal("clear");
16:     clear.getParameter("block").setValue(block);
17:     dispatchSubgoalAndWait(clear);
18:
19:     clear = createGoal("clear");
20:     clear.getParameter("block").setValue(target);
21:     dispatchSubgoalAndWait(clear);
22:
23:     block.stackOn(target);
24:   }
25: }
```

Figure 6.11. Java code for StackBlocksPlan

top of the other. To clear the first block, a goal of type "clear" (cf. Fig. 6.10) is created (line 15) and the parameter is set to the block (line 16). The dispatchSubgoalAndWait() method (line 17) forces the agent to adopt the goal, and halts the execution of the plan until goal processing is finished. If the goal fails, an exception is thrown causing the whole plan to fail. Otherwise, the plan continues to clear the target block in a similar fashion (lines 19-21). Finally, the plan stacks the blocks on each other by calling the stackOn() method of the Block class (line 23).

In the ConfigureBlocksPlan (Fig. 6.12), the body() method (lines 12-25) consists of two loops through all stacks on the table, and all blocks of each stack, as returned by the getStacks() method of the Table class (line 13). This table object represents the desired target configuration. The agent now has to look up the corresponding blocks in its beliefbase, and then operate on these blocks such that they resemble the target configuration. The lookup is simple for the block itself, as the corresponding object can be obtained directly from the belief set (line 16). The lookup of the object below the block (lines 17-19) is somewhat more difficult, because the block could be located directly on the table (line 18) or on top of another block (line 19). To perform the actual changes to the retrieved objects, a "stack" goal is created

```
01: package jadex.examples.blocksworld;
02: import jadex.runtime.*;
03:
04: /** Plan to to establish a given configuration of blocks. */
05: public class ConfigureBlocksPlan extends Plan {
06:   protected Table table;
07:
08:   public ConfigureBlocksPlan(Table table) {
09:     this.table = table;
10:   }
11:
12:   public void body() {
13:     Block[][] stacks = table.getStacks();
14:     for(int i=0; i<stacks.length; i++) {
15:       for(int j=0; j<stacks[i].length; j++) {
16:         Block block=(Block)getBeliefbase().getBeliefSet("blocks").getFact(stacks[i][j]);
17:         Block target=stacks[i][j].getLower()==table
18:         ?(Table)getBeliefbase().getBelief("table").getFact()
19:         :(Block)getBeliefbase().getBeliefSet("blocks").getFact(stacks[i][j].getLower());
20:
21:         IGoal stack = createGoal("stack");
22:         stack.getParameter("block").setValue(block);
23:         stack.getParameter("target").setValue(target);
24:         dispatchSubgoalAndWait(stack);
25:       }
26:     }
27:   }
28: }
```

Figure 6.12. Java code for ConfigureBlocksPlan

and dispatched (lines 21-24). Because the loop processes the stacks bottom-to-top, the sequential execution of all "stack" goals ensures that the final configuration resembles the desired target configuration.

6.4 Platform

This section describes the realization of the Jadex reasoning engine, and its integration into the JADE platform. Figure 6.13 shows the essential components required for developing and executing a Jadex agent, and highlights the dependencies between those components. The components are distinguished in *core system components* (upper row) which realize the reasoning engine, *system interface components* (middle row) that provide and define the access points to the system, and *custom application components* (lower row) which have to be supplied by the agent developer. The links between the components can be categorized in *runtime* dependencies (i.e. between components in the first two columns from the left), dependencies that only apply during the *agent startup* phase (see third column components), and dependencies resolved at *design time* (right column).

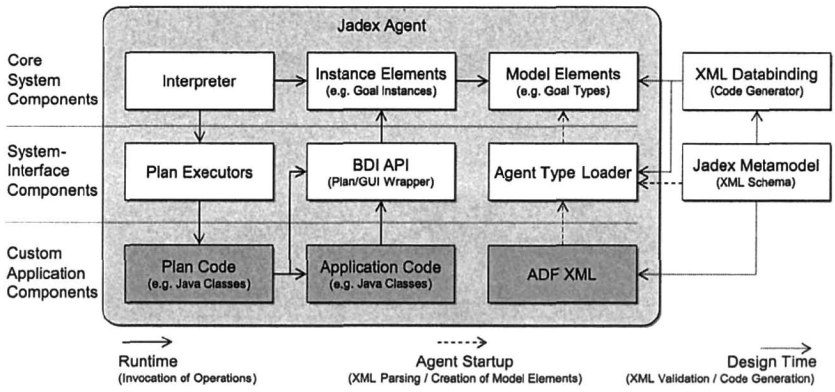

Figure 6.13. System realization

We will describe the components starting from the right. Jadex is based on a BDI *metamodel* defined in XML Schema (cf. Sect. 6.3.1). This schema is on the one hand used to validate the agent models specified in XML agent definition files (*ADF*). On the other hand, an *XML databinding* framework[4] is used to generate Java classes for the elements of the metamodel and for reading model elements from XML. When an agent is instantiated, the generated *agent type loader* reads the user supplied XML agent model and automatically creates the corresponding *model elements*.

From these model elements, instances are continuously created at runtime, represented by *instance elements*. The main *interpreter* operates on the current instance elements and executes plans to handle events and goals. *Plan executors* are used to hide the details of plan implementation types from the system. As a default, there is a plan executor for executing *plan code* written in Java. Plan code may access any other *application code* or third party libraries written in a suitable language. Both plan and application code has access to the reasoning engine through a *BDI API*. It is provided to plan and other application (e.g. GUI) code by wrappers that encapsulate the instance elements, and ensure proper synchronization and deadlock-avoidance when the API is called from the plans, or from external threads respectively.

For integration into JADE, the platform management tool (RMA) has been extended slightly to support launching of Jadex agents, by selecting the corresponding agent model with a file chooser. The Jadex interpreter itself is realized as a special type of JADE agent, which loads an agent model supplied

[4]JBind Java-XML Data Binding Framework, see http://jbind.sourceforge.net/

at startup, and creates its own instance of the reasoning engine according to the settings given in the model (e.g. initial beliefs, goals, and plans). The functionalities corresponding to the execution model components (message receiver, dispatcher, scheduler, cf. section 6.2.3), are implemented as cyclic behaviours (cf. chapter 5), always running inside the agent. These behaviours call the reasoning engine to process incoming messages, and perform internal reasoning. In each JADE agent cycle, the reasoning engine is called to process one event and execute one plan step. Using a reference to the JADE agent object, Jadex plans have direct access to all operations of the JADE API as well (e.g. for handling of FIPA ACL messages).

6.4.1 Available tools and documentation

The system distribution contains complete documentation materials for quick start and reference purposes. An introductory tutorial made up of several exercises shows the usage of basic system features in a step-by-step manner. Moreover, the distribution provides several example applications including their commented source code. A user guide provides a systematic overview of all features and also serves as a reference manual. In addition, Javadocs of the plan programming API and a reference to the metamodel defined in XML Schema are provided. The available tools are covered in a separate guide. Apart from the documentation material included in the distribution, there are publicly available online tools kindly hosted by SourceForge.net, such as web forums for discussion and support requests, a database for bug-reports and feature requests, and a general mailing list with online archives.

As a Jadex agent is still a JADE agent, all runtime tools provided by the JADE platform such as Sniffer and Dummy agent can also be used with Jadex agents. To enable a comfortable testing of the internals of Jadex agents additional tool agents have been developed. In Fig. 6.14 an example application (marsworld) is depicted together with the logger and introspector tools in a typical debugging session. The BDI introspector (Fig.6.14 bottom left and right hand side) serves two purposes. 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 a given event or goal.

With the help of the logger (see Fig.6.14 on the top right) 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

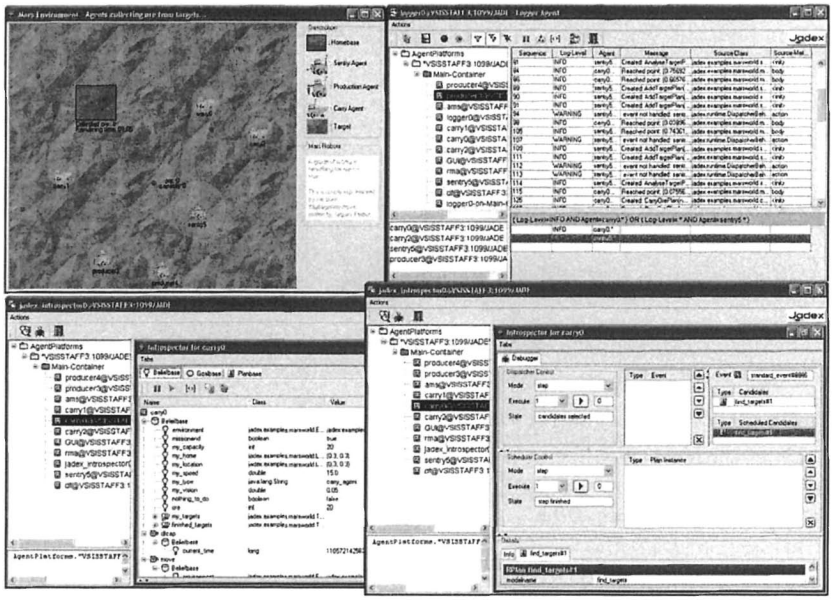

Figure 6.14. BDI introspector and logger screenshots

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.

Moreover, a tracer tool for on-line visualization of agent execution based on ideas from [132] is provided. It generates a unified view of multi-agent and internal agent behaviour, relating message-based communication and internal agent processes. The Jadexdoc tool allows generating documentation of agent applications similar to Javadoc. In addition to these tools already included in the latest release, a tool for multi-agent application deployment is currently in development (see [29]).

6.4.2 Standards compliance, interoperability and portability

One driving factor for the development of Jadex was the need for a FIPA-compliant platform supporting advanced BDI reasoning capabilities. FIPA-compliance is achieved through the JADE platform, which provides sophisticated implementations of all important FIPA specifications. The Jadex reasoning engine, realized on top of the JADE platform, in itself only supports

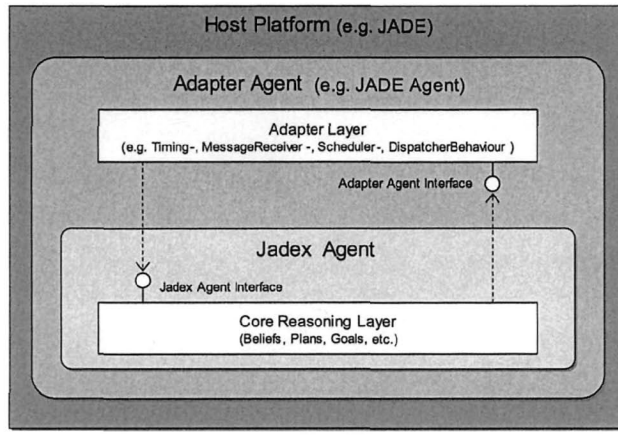

Figure 6.15. Platform integration

homogeneous (i.e. BDI) agents, but provides interoperability with agents based on other models. Agents realized using the conventional JADE programming techniques can be executed directly together with Jadex agents running on the same platform. Interoperability with other kinds of agents is straightforward as long as those agents adhere to the FIPA standard. E.g. in an example application, Jadex agents have been successfully connected to agents running on the CAPA platform [76], which provides a petri-net based computation model for agents.

The reasoning engine has been realized as a separate component, intentionally limiting the dependencies to the underlying platform. To use the reasoning engine on top of other platforms, an adapter has to be realized (see Fig. 6.15). This adapter has to implement a handful of methods used by the Jadex engine (e.g. to send messages) and has to call the engine when it is expected to do the reasoning. Therefore, although the current implementation is designed to be used with JADE, the reasoning engine can be easily integrated with other FIPA-compliant agent platforms such as CAPA [76] or ADK[5], given that they provide a similar interface for message handling. It is also possible to use the system in conjunction with other middleware environments such as J2EE or .NET, when FIPA-compliance is not needed. Currently, in addition to the JADE integration, we have developed experimental adapters for the DIET agent platform [147] and for running a set of Jadex agents as a standalone Java application.

[5]http://www.tryllian.com/

The engine was realized in Java 1.4 and includes the third party packages JBind for XML data binding and Apache Velocity[6] for generating the content of some tool dialogs. To support mobile devices, a port of the engine is also available in a reduced version based on J2ME / CDC. Moreover, all kinds of tools and libraries with a Java API can easily be used to provide additional features. For example, in a larger project the Cayenne database mapping framework[7] was used to connect agents to a relational database.

6.5 Applications supported by the language and/or the platform

Jadex is a general-purpose development environment for creating multi-agent system applications, allowing to build agents with reactive (event-based) and deliberative (goal-driven) behaviour. It is not bound to a specific target domain, but has been used to realize applications in different domains such as simulation, scheduling, and mobile computation. Jadex originated in the MedPAge ("Medical Path Agents") project [166, 167], which is part of the German priority research programme 1083 *Intelligent Agents in Real-World Business Applications* funded by the Deutsche Forschungsgemeinschaft (DFG). In cooperation with the business management department of the University of Mannheim, the project investigates the advantages of using agent technology in the context of hospital logistics. In this project Jadex is used to realize a multi-agent application for market-based negotiation of treatment schedules [167], as well as for the simulation of a hospital model to test the negotiation mechanism [31]. In other contexts, Jadex was used to realize portable PDA-based applications. A personal mobile task planner was developed, to test the Jadex J2ME port and to prove the usefulness of BDI agents on mobile devices [104]. Elsewhere, in the PITA ("Personal Intelligent Travel Assistant") project at the Delft University of Technology, Jadex was used to realize a prototype of a mobile personal travel assistant application [9].

Besides building specific agent applications, Jadex has also been used for teaching and research regarding agent oriented software development in general. Due to its simple language based on well-known technologies such as Java and XML, and the extensive documentation material and illustrative example applications, Jadex is well suited for teaching purposes. It has been successfully applied in several courses at the University of Hamburg, and is also evaluated by other institutes. Regarding research in agent systems, the project is also designed as a means for researchers to further investigate

[6] http://jakarta.apache.org/velocity/
[7] http://objectstyle.org/cayenne/

which mentalistic concepts are appropriate in the design and implementation of agent systems. The combination of XML Schema with Java databinding techniques allows the Jadex metamodel to be flexibly adapted and extended for experimentation purposes. While investigating different representations for beliefs, goals and plans, the system has been applied to several well-known AI problem domains (blocksworld, cleanerworld, mars robots, hunter-prey). These applications are also included in the distribution. Moreover, the Technical University of Karlsruhe has used Jadex to implement an experimental system for representing norms in multi-agent systems [204].

6.6 Final Remarks

In this chapter, the Jadex BDI reasoning engine has been presented. The realization of the system is motivated mainly by three factors. Firstly, the system aims to combine the benefits of agent middleware and internal agent reasoning processes. Secondly, it intends to enhance the state-of-the-art BDI architecture by addressing some shortcomings of current BDI agent platforms such as implicit goal representation and thirdly, the system targets on making agent technology more easily usable by exploiting current software engineering techniques such as XML, Java and OQL.

The architecture of Jadex is in principle similar to traditional PRS systems, when event and goal processing is considered. Nevertheless, conceptual differences exist mainly concerning the representation of BDI core concepts and as well on language level. According to the usability requirement, beliefs are expressed in an object-oriented way instead of using logical formulae or relational models. Moreover, goals are represented as explicit durable entities instead of relying on events. On language level, Jadex differentiates between the description of an agent's behaviour and its static structure. Therefore, for each of these purposes different languages are employed. The static agent structure is declared in an XML-dialect following the Jadex BDI metamodel specified in XML-schema, whereas ordinary Java is used for plan realization. BDI-specific facilities are made accessible from within plan through an application program interface.

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. It is planned to utilize the explicit representation of goals 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 implementation and testing phase supplying tools like the debugger and 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 [8]. Additionally, a tools for deployment of multi-agent applications is being developed [29].

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, and has reached considerable stability and maturity to be used in experimental and practical settings.

Acknowledgments

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