Jadex: Engineering goal-oriented agents

Alexander Pokahr, Lars Braubach, Andrzej Walczak, and Winfried Lamersdorf

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

In previous sections of the book agents have been considered as software artifacts that differ from objects mainly in the capability to autonomously execute tasks and to decide for themselves if and how something will be done. Following this view it is possible to design autonomous entities that communicate to solve application problems by cooperation or negotiation. An even more abstract view can be obtained if, besides autonomy, one explicitly supports the proactiveness of agents. Proactiveness basically means that agents can have their own goals and have behavioral freedom in which ways the goals are pursued, i.e. which means are used to fulfill the goals.

Most software systems and therefore also software agents are goal-oriented in the way that they pursue the design goals laid down by the developer of the system. This is not what is meant by proactiveness as the software artifacts only have implicit goals and are not aware of them. In contrast to this, proactiveness is closely related to the explicit representation of goals and other mental attitudes. By using explicit representations, agents can be made aware of their goals and get a chance to reason about their goals, e.g. for finding alternative ways to achieve goals, giving up unreachable goals or adopting new ones given the right opportunities.

Explicit representations of goals and other mental attitudes have been advocated by many well-known researchers such as Dennett [6], McCarthy [9] and Newell [10] for a long time as they drastically increase the human abilities to understand and predict behavior of systems and also establish a natural abstraction layer on which software agents can be built.

Several approaches exist proposing different kinds of mental attitudes and their relationships. One prominent approach is the BDI model that was originally conceived by Bratman [1] as a philosophical theory of practical reasoning explaining human behavior with the attitudes: beliefs, desires and intentions. The basic assumption of the BDI model is that actions are derived in a two-step process called *practical reasoning*. In the first step – (goal) *deliberation* – it is decided which set of desires should be pursued in the current situation represented in the agent's beliefs. The second step – *means-end reasoning* – is responsible for determining how such concrete desires produced as result of the former step can be achieved by employing the means available to the agent [16].

Rao and Georgeff adopted the BDI model for software agents by introducing a formal theory and an abstract BDI interpreter [14] that is the basis of nearly all historical and current BDI systems. The BDI systems in the spirit of Rao and Georgeff's interpreter are also called Procedural Reasoning Systems (PRS), termed after the first successfully implemented system. The interpreter operates on the agent's beliefs, goals and plans which represent slightly modified concepts of the original mentalistic notions. The most

significant difference is that goals are considered as consistent set of concrete desires that can be achieved altogether thereby avoiding the need for a complex goal deliberation phase. Main task of the interpreter therefore is the realization of the means-end process by selecting and executing plans for a given goal or event.

This chapter describes one of those interpreters – the Jadex BDI reasoning engine. The organization of this chapter is as follows. In the following Section 1.1 an overview about the Jadex reasoning engine will be given by explaining its underlying concepts as well as its abstract architecture. In addition a short introduction to the available tool suite will be given. In Section 1.2 it will be shown how the book-trading scenario can be realized using Jadex BDI agents. For this purpose in a first step the book-trading scenario will be modeled with the Tropos methodology, which encourages descriptions in terms of mentalistic notions. In a second step it will be shown how a natural mapping can be applied to the Tropos models yielding Jadex agent code. A summary of this chapter is given in Section 1.3.

1.1 Jadex reasoning engine overview

The Jadex reasoning engine enables the construction of rational agents following the BDI model. In contrast to all other available BDI engines Jadex fully supports the two-step practical reasoning process instead of operationalizing only the means-end process. This means that Jadex allows for building agents with beliefs, goals and plans that automatically deliberate about their goals and subsequently pursue them by applying appropriate plans.

1.1.1 Decoupling of middleware and reasoning

Agent technology has to address many fundamentally different aspects such as infrastructure needs, an agent programming model and a social layer allowing for agent teamwork. Jadex explicitly concentrates on the agent programming model and does not impose constraints on the other aspects. For this reason we call Jadex a BDI reasoning engine and not per se an agent platform. The reasoning engine is clearly separated from its underlying infrastructure that provides basic platform services such as life cycle management and communication means. Therefore, different kinds of (agent) middleware can be used for operating the Jadex engine. So far, adapters for the agent platforms JADE, Diet[1] as well as a small-footprint Standalone version have been developed. Further adapters allowing Jadex also being used in J2EE application servers are currently under development.

1.1.2 Jadex architecture

In Figure 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. The interpreter handles *practical reasoning* via two components responsible for *goal deliberation* and *means-end reasoning*. The goal-deliberation is mainly state-based and has the purpose to select the current non-conflicting set of goals.

[1] http://diet-agents.sourceforge.net/

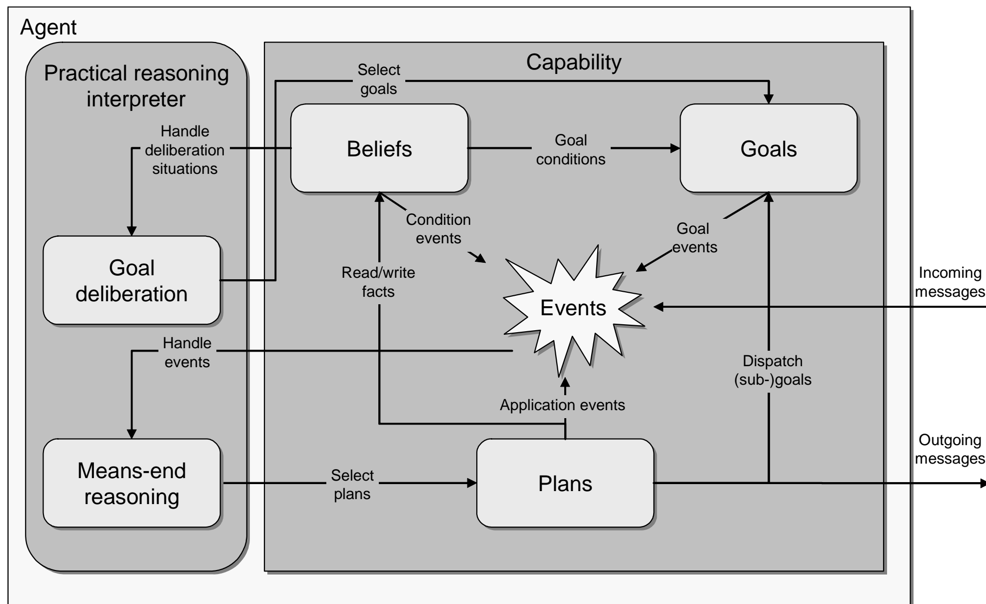

Figure 1 Jadex abstract architecture

Incoming messages, as well as internal events and new goals serve as input to the means-end reasoning component that dispatches these events to plans selected from the plan library for further processing. 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 interpreter represents the only global component within Jadex. All other components are contained in reusable modules called capabilities. In the following a short introduction to the programming concepts introduced above is given. For a more detailed explanation of the concepts the reader can refer to [2, 11]. The internal mode of operation, which is quite different to traditional BDI systems, has been formally explained in [12].

1.1.3 Agent programming concepts

Contrary to most of other BDI systems Jadex intentionally does not promote a new agent programming language but instead relies completely on established software engineering techniques such as Java and XML. As a BDI agent consists of structural as well as behavioral parts Jadex chooses a hybrid approach for defining and programming agents. The structural part comprises the agent's static design composed of elements such as belief-, goal- and plan types and additionally the agent's initial state consisting of e.g. goals and plans that will be pursued once the agent is born. These structural aspects are specified in the Jadex XML language following an XML-Schema defining the BDI metamodel. On the other hand, the dynamic agent behavior needs to be encoded, too. All such procedural knowledge is contained in the Jadex plans and is described using the plain Java programming language. Figure 2 illustrates the two conceptually different parts a Jadex agent is composed of: on the left hand side the agent definition file (ADF) for the agent type definition is shown and on the right hand side the procedural plans are depicted. The connection between both parts is established by an API enabling the Java plan classes accessing BDI aspects such as reading the beliefs or issuing new subgoals.

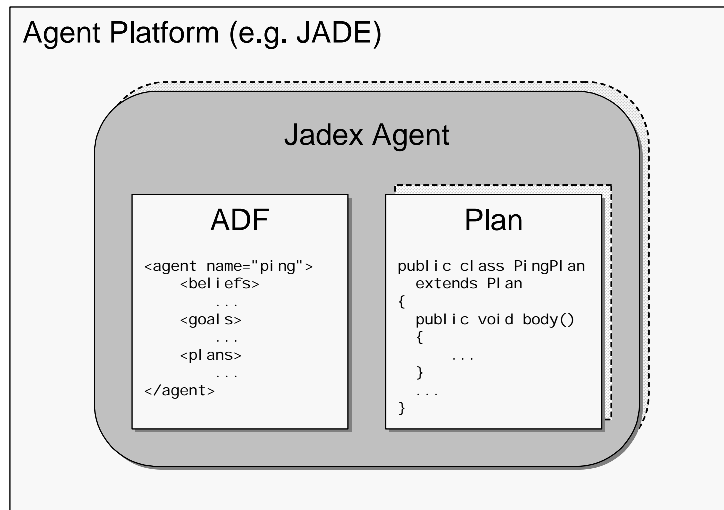

Figure 2 Jadex agent specification

Beliefs

Beliefs represent the agent's knowledge about the world, itself and other agents. The belief representation in Jadex is software engineering centric in allowing arbitrary Java objects being stored instead of relying on a logic-based representation. This facilitates the integration with existing software, e.g. classes generated by ontology modeling tools or database mapping layers can directly be reused. Objects are stored as named facts (called beliefs) or named sets of facts (called belief sets). Using the belief names, the beliefbase can be manipulated by setting, adding, or removing facts. In addition a more declarative way of accessing beliefs and beliefsets is provided by OQL-like queries. Besides being a passive data store for the agent the beliefbase also plays a vital role in the reasoning processes as changes in the beliefs are automatically monitored by the engine and conditions can be defined referring e.g. to domain relevant belief values. Hence belief changes may trigger a goal's creation or drop condition, or render the context of a plan invalid leading to a plan abort.

Goals

Goals are the motivational force driving the agent's actions. They come in different flavors allowing various attitudes of an agent being expressed towards its goals. Jadex currently supports four application relevant goal types: perform, achieve, query and maintain goals. **Perform goals** express the agent's wish to directly engage into actions being already satisfied if something (regardless what exactly) has been done. Such behavior is suitable for abstractly modeling activities such as driving around with the car just for fun. In contrast to such activity centric goals, **achieve goals** are associated with a desired world state. An example for such kind of a goal is e.g. having the car parked near the driving destination. In this case the state and position of the car is relevant for the goal's fulfillment state and hence expresses the goal's target world state. Therefore, achieve goals clearly emphasize the decoupling of what is to be achieved and how it will be brought about. **Query goals** instead can be used for information retrieval, i.e. their outcome is not defined by some target condition but by a query that needs to be answered. Depending on the agent's current knowledge plans may or may not be

executed, meaning that if the information is readily available no additional work will be necessary. An example for a query goal is finding out the way to drive by car to the intended destination. Finally, **maintain goals** are used to describe situations that should be preserved by the agent under all circumstances. Strictly speaking, it means that whenever the situation specified gets violated the agent will activate any applicable means to re-establish the desired world state. Having the car functioning properly is an example for such a kind of goal. This involves e.g. getting it repaired after an accident or renewing the car's technical admittance whenever necessary. More information about the goal representation in Jadex can be found in [4].

When describing real world scenarios with goals often the problem arises that not all of the individual agent goals can be pursued at the same time. Such goal interferences are an important design aspect that is directly addressed by the goal deliberation facilities of Jadex. With the built-in Easy Deliberation strategy [13] goal cardinalities and inhibition links between goals can be modeled and the system will take care at runtime that at any time only valid goal subsets are active. If a goal set contains conflicting goals, the system will exploit the defined inhibition links to delay less important goals while executing the more important ones. Whenever goals are finished, the system considers the re-activation of currently inhibited goals.

Plans

Means-end reasoning is performed with the objective of determining suitable plans for pursuing goals or handling other kinds of events such as messages or belief changes. Instead of performing planning from first principles PRS systems like Jadex use the plan-library approach to represent the plans of an agent. A plan consists of two distinct parts: the plan head and body. The plan head contains information about the situations in which the plan will be used. Most importantly it includes the events and goals the plan can handle and conditions that are used to restrict the applicability of a plan. Using conditions it is also possible to abort a running plan immediately if the current situation demands this. On the other hand the plan body represents the recipe of actions that will be performed if the plan is chosen for execution. Depending on the plan's purpose its degree of abstractness varies continuously between very concrete and fully abstract. Concrete plans are fully specified at design time and consist of directly executable actions only whereas fully abstract plans are specified in terms of subgoals only.

Plan programming in Jadex requires the definition of the plan head in the ADF and the programming of the plan body in a pure Java class. The body is implemented by extending an existing Jadex framework class. This enables plan classes to access agent and BDI specific functionality such as sending messages, accessing beliefs or dispatching subgoals. Errors that may occur in BDI processing such as a failed subgoal or a timeout when waiting for a reply message are mapped to BDI exceptions causing the plan to fail if not intentionally catched by the programmer. Besides the main body, a plan may include special methods for clean up operations needed for a proper completion of a plan in case of failure or success.

Capabilities

Creating BDI agents for complex problems requires software engineering mechanisms supporting modularity and reuse of code. This is where capabilities come into play [3, 5]. They allow for packaging a clear-cut piece of functionality into a module with precisely

defined interfaces. An agent can be composed of an arbitrary number of capabilities that themselves may include any number of subcapabilities making capabilities a hierarchical decomposition instrument. A capability description is defined in a separate XML document similar to the ADF and also consists of beliefs, plans and goals needed to generate the intended behavior. Per default all capability elements have local scope and thus cannot be seen or used in other capabilities or in the agent. This ensures a maximum degree of encapsulation and avoids any kind of unmeant interference between different capabilities. The parts of the capability that should make up the capability's interface need to be explicitly exported.

To ease the user's life, the Jadex distribution already contains several generic plans and predefined capabilities in the package jadex.planlib ready for direct usage. Basic platform features can be accessed by using the AMS and DF capabilities. The AMS capability offers goals for agent management such as creating new agents or destroying existing ones whereas the DF capability can be used for accessing yellow pages services such as registering agents or searching specific services via goals. Furthermore, from the protocols capability several well-known FIPA interaction protocols such as request or contract-net are available also via several goals. The rationale behind the goal oriented view provided also for protocols is that it allows adopting a more abstract viewpoint that concentrates not on the message flows but on the domain activities within a protocol.

1.1.4 Starting Jadex

Jadex can be downloaded from the project homepage at http://jadex.sourceforge.net/. For the purpose of this chapter it is recommended that you download the latest[2] jadex-full.zip, which contains the standalone version of Jadex, and optionally also download the latest jadex-jadeadapter.zip, which contains the necessary files to run Jadex on top of JADE. Unzip both zip-files to a directory of your choice. When using Jadex 0.95 unzipping will create two new directories jadex-0.95 and jadex-jadeadapter-0.95.

To start the Jadex platform, double-click[3] on the jadex_standalone.jar in the jadex-0.95/lib directory. After a few seconds the Jadex control center (JCC) will show up (see Figure 3). When you first start Jadex, the JCC will open the default project containing the examples supplied with the distribution. Later, when you create your own projects, the JCC will always open the last used project on startup. To start an example, select an ADF file in the tree on the left and click the start button. The multi-agent examples such as BlackJack, Cleanerworld, and Hunterprey, can be started by executing the manager agent available for each of these examples. To execute the JADE version of Jadex, first copy all jar files (except the jadex_standalone.jar) from jadex-0.95/lib to jadex-jadeadapter-0.95/lib. Then you can start JADE by double-clicking the jadex_jadeadapter.jar. The JCC is realized as a portable Jadex agent, and therefore is also used as the main control center when running Jadex on JADE.

[2] The following explanations are based on Jadex version 0.95. If you use another version, please refer to the user guide for possible differences between the versions.

[3] When you have a proper installation of a Java runtime environment, double-clicking the jar file will execute the Java application. If you experience problems, open a console window change to the jadex-0.95/lib directory and enter *java -jar jadex_standalone.jar*. Refer to the user guide for more details on starting the standalone platform.

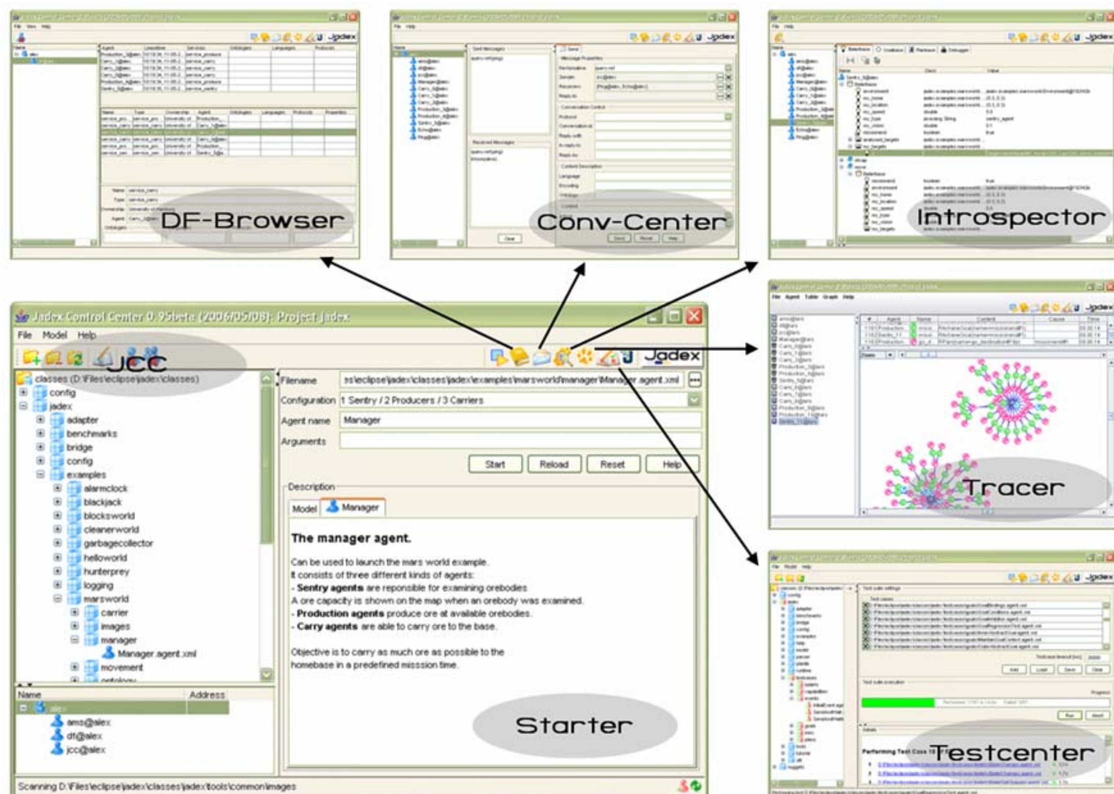

Figure 3 Jadex control center (JCC) and plugin tools

1.1.5 Jadex runtime tools

The JCC is the central access point to all runtime tools provided by Jadex. The tools are included as plugins, which can be accessed by the toolbar buttons at the top right (see Figure 3). The first plugin is the Starter component, which you already have used to start the example applications. The other plugins allow inspecting currently registered agent services (DF Browser), manually sending messages to agents (Conversation Center), introspecting and debugging the internal state of running agents (Introspector), tracing one or more agents to obtain a global view of the system (Tracer), and executing test cases (Test Center). When running on top of JADE an additional RMA plugin provides access to all existing JADE tools using the familiar JADE remote monitoring agent (RMA) interface.

In the following, the individual plugins will be shortly introduced. For more details on the usage and functionality of the tools you may have a look at the tool guide included in the distribution or consult the context sensitive help available from the help menu of the JCC. The **Starter** plugin is divided in four sections. The top-left tree shows the agents included in the directories and jar-files, which are part of the current project. In the Starter you can add directories and jar-files to the project. When you select an agent from the tree, a description as specified in the ADF will be shown in the bottom right panel. If the agent file contains errors, this is indicated by a red bolt on the agent symbol, and detailed descriptions of the errors will be shown in the bottom right panel. The error indication in the tree view is propagated to the containing packages of erroneous files

facilitating the localization of incorrect program parts. The top right section provides settings to start a new instance of a loaded agent file. The list at the bottom-left shows the agent instances currently running on the platform, allowing to stop and kill any agent instance. The **DF Browser** shows the registrations held by a DF agent selected in the list on the left. The right side shows the agent (top) and service registrations (middle) as well as details for a selected service (bottom). The **Conversation Center** allows manually sending messages to agents and viewing their replies. The panel on the right allows editing a new message to be sent, and to view the properties of received messages. Sent messages are stored in the upper list to the right of the message panel. Received messages will appear in the lower list. Messages from both lists can be opened for viewing.

The **Introspector** contains a list of the currently running agents at the left side. The introspector view shown on the right for a selected agent is divided in four tabs (Beliefbase, Goalbase, Planbase, Debugger), which can be activated separately. The belief-, goal-, and planbase tabs show the current belief, goals, and plans of the agent. The debugger allows executing the agent step by step. The **Tracer** collects internal reasoning events of agents, such as adoptions of new goals, changes to beliefs, or sending and receiving of messages. The agents to be traced can be selected from the list at the left. At the top right, a table view of the sequence of observed events is shown. The lower right presents a graphical 2D view of the events and their causal dependencies. The **Test Center** allows creating and executing test suites (top right panel) composed of test case agents selected from directories or jar files shown in the tree on the left. The detailed results of test cases are shown at the bottom right.

The **RMA Plugin** is only available when running on JADE. It contains the interface of the remote monitoring agent of JADE. From this plugin you can access all JADE-specific functionality, e.g. for migration of agents. You can also use the plugin to start the runtime tools provided by JADE, such as the Sniffer. The RMA plugin is useful even when developing pure Jadex applications, because all JADE tools like the Sniffer can also be used for Jadex agents.

1.2 Developing the book-trading scenario with Jadex

After having presented an overview of the concepts behind the Jadex reasoning engine, we now delve into the details of developing goal-oriented agent-based applications. For this purpose, we use the book-trading scenario already presented in earlier chapters as a concrete example.

When developing a multi-agent application using Jadex it is advantageous that the design of the application to be built is specified in mentalistic terms. Thanks to the abstractness of Jadex agent specifications, the mentalistic design is preserved in code-centric development processes like eXtreme Programming (XP). A more traditional systematic approach involves the adoption of a software methodology, which guides the development process from requirements to code. In this case, from the multitude of available agent methodologies [8, 15] especially those are useful that describe the problem and solution in terms of BDI notions. In this section we will use the Tropos methodology for a goal-driven modeling of the domain and solution space.

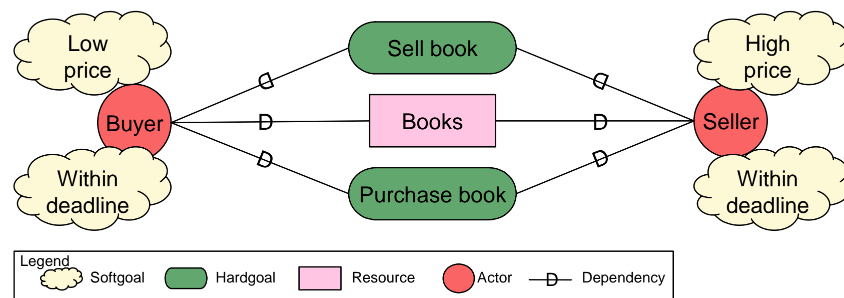

Figure 4 Initial actor/goal book-trading diagram

1.2.1 A short introduction to Tropos

In Tropos the problem domain is described with individual actors and their relationships. Thereby, actors represent a very generic concept that unifies concrete agent types as well as roles and human actors in one notion. For each actor it is elaborated which individual goals she pursues and additionally on which other actors she depends in fulfilling her goals, executing her tasks or receiving necessary resources. In this way strategic dependencies between different actors are made explicit leading to a problem and solution description from a bird's eye perspective concentrating on the important social aspects instead of internal workings. In subsequent steps the individual actors are refined with respect to their internal goal hierarchies and the plans and resources that are needed to achieve their goals. Basically, two techniques are employed for refinement: and/or decomposition of goals and plans and means-end analysis of goals for the determination of plans that fulfill these goals. For a deeper understanding of the Tropos methodology the reader may refer to further literature [7].

1.2.2 Modeling the book-trading domain with Tropos

Using the Tropos methodology one normally begins with an analysis of the problem domain initially ignoring the system-to-be. As the objective of the book-trading scenario is clearly defined we will ignore this step and directly start with the analysis of the system requirements (see Figure 4). The system basically consists of buyers and sellers that try to buy and sell books respectively. Hence a buyer actor has a purchase book goal that depends on the seller actor leading to a goal fulfillment dependency. The same is true the other way round for the seller having a sell goal and the corresponding strategic goal fulfillment dependency with a buyer. In addition it is assumed that the seller owns a set of books represented as the seller's resource. As a buyer desires to have some of the seller's books a resource dependency between buyer and seller is introduced. Besides the wish to obtain a book the buyer has clear ideas about the conditions that should be met when a transaction is carried out. Hence, the buyer wants to buy its books as cheap as possible within a given deadline. The same applies for the seller who wants to achieve a good price and sell the book within its given deadline. All these qualities are modeled as softgoals, i.e. as goals with qualitative properties.

In the next step of the modeling process the actor diagrams are refined with respect to the above mentioned decomposition and means-end analysis techniques (see Figure 5). First, contribution links indicate that the buy and sell goals of actors should contribute to the fulfillment of the corresponding softgoals of each actor in a positive way.

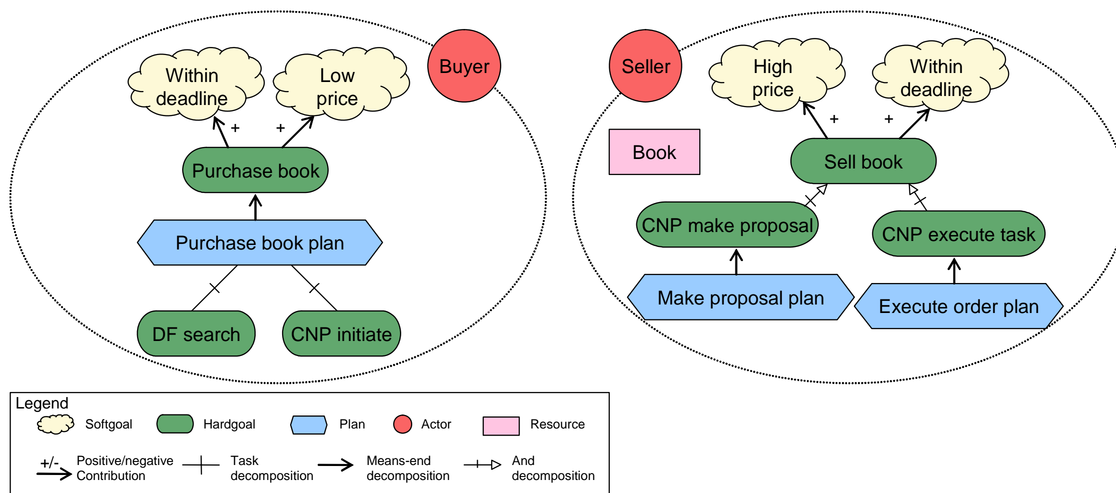

Figure 5 Refined actor/goal book-trading diagram

Further on the buyer's side, the purchase book goal will be handled by a purchase book plan which coordinates two tasks. In a first step available sellers have to be searched – this is represented by the *DF search* subgoal – and in a second step a parallel negotiation with all sellers should be performed to buy the book at best conditions – represented by the *CNP initiate* (contract-net protocol) goal. If a purchase book plan fails because one of the subgoals does not succeed, i.e. because no sellers could be found or no seller offers the book for an acceptable price, the goal should not be given up but should be retried until the book could be bought or the goal's deadline has expired.

Unlike the buyer, which actively pursues its purchase goals, the seller waits for requests from buyers. Therefore, the sell book goal is decomposed into two subgoals *CNP make proposal* and *CNP execute task*. The *CNP make proposal* goal is created, when the seller receives a call for proposal from a buyer. It is handled by a *make proposal plan*, which has to check, if the seller is currently willing to sell the requested book and for what price. If the buyer accepts a proposal made by the seller, the *execute task* goal is created to finish the transaction between buyer and seller, which is done by the *execute order plan*

1.2.3 Implementing the book-trading domain with Jadex

Development of Jadex agents is best supported by common integrated development environments (IDE) for Java such as eclipse or Intellij IDEA[4]. Using such standard IDEs is possible because Jadex does not introduce a new agent programming language but relies on established techniques only. Eclipse webtools[5] for example provides enhanced support for the creation of ADFs including XML auto completion and error indication features based on XML-Schema as well as for the implementation of plans in Java.

The implementation of the book-trading example is divided in four packages. Separate packages contain the plans specially designed for the buyer and seller agents, as well as their ADFs. The buyer agent buys books on behalf of its user by actively

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

[5] http://www.eclipse.org/webtools/

searching for sellers and negotiating with them about the prices. If the negotiation does not succeed the buyer will periodically try again to buy the book until the deadline has elapsed or the book could be bought. The seller agent holds different offers and responds to proposal requests with an offer that is timely adjusted to changing conditions. The manager package contains a simple manager agent that is responsible for starting and stopping the sellers and buyers.

Classes that are shared by the buyer and seller are stored in the package common. It contains the Gui of both agent types and the order object. The order concept is introduced at the implementation level to aggregate the detailed information about book offers and requests such as the start price, price limit, deadline, execution price, execution date, fulfillment state etc. An order is directly associated with a corresponding purchase or sell book goal, as it contains the detailed conditions under which the goal needs to be pursued.

1.2.4 Implementing the buyer agent

The buyer agent is completely described in the Buyer.agent.xml ADF (see Figure 6). The basic functionality of the buyer is initiated through its user who can issue new purchase book orders from the Gui. For each such order a new purchase_book goal is created. All the agent's open orders are also represented in the orders belief (lines 18-21), which is a virtual belief set meaning that values are obtained by a query against the goal base.

The definition of the purchase_book goal type (lines 28-32) consists of a parameter storing the actual order (line 29) and a target resp. a failure condition. The target condition states in what case the goal has succeeded. In the example, a purchase_book goal succeeds when the state of order (stored in its parameter) has changed from *open* to *done* (line 30). The goal will be considered to have failed when its failure condition triggers. Here, the failure condition (line 31) states that the goal has failed when the order's deadline has expired. The deadline is compared against the actual system time contained in the time belief (lines 22-24). A failure condition using directly System.currentTimeMillis() instead of $beliefbase.time would not work, as it has to be ensured that the condition will be evaluated whenever the system time changes. This has been accomplished by using a dynamic time belief, which is evaluated every 1000 ms (as specified with the updaterate attribute, line 22). The purchase_book goal leads to the execution of plans which try to fulfill the goal by negotiating with sellers. As such negotiations might fail the goal should be able to try its achievement again. Therefore the recur attribute is used (line 28). If this attribute is set to false and all available plans have failed once, the goal will not be reattempted. To assure that the agent does not reattempt the goals immediately also a recurdelay has been specified (line 28). It determines that a gap of 10000 ms exists between a failed goal execution and the next try.

Besides the purchase_book goal two other goal types are defined in the buyer ADF (lines 33, 34). The df_search goal is used to search seller agents at the DF and the cnp_initiate goal is employed to perform a contract-net negotiation with multiple sellers. Both goals are declared as goal references, meaning that they refer to goals declared and exported in other capabilities. The df_search goal references the corresponding goal within the jadex.planlib.DF capability (with local name dfcap) and the cnp_initiate goal refers to the cnp_initate goal from the jadex.planlib.Protocols capability (with local name procap).

```
01: <agent xmlns="http://jadex.sourceforge.net/jadex"
02:            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03:            xsi:schemaLocation="http://jadex.sourceforge.net/jadex
04:                                   http://jadex.sourceforge.net/jadex-0.95.xsd"
05:            name="Buyer" package="jadex.examples.booktrading.buyer">
06:
07:   <imports>
08:     <import>jadex.examples.booktrading.common.*</import>
09:     <import>jadex.adapter.fipa.SFipa</import>
10:   </imports>
11:
12:   <capabilities>
13:     <capability name="procap" file="jadex.planlib.Protocols" />
14:     <capability name="dfcap" file="jadex.planlib.DF" />
15:   </capabilities>
16:
17:   <beliefs>
18:     <beliefset name="orders" class="Order">
19:       <facts evaluationmode="dynamic">select $g.getParameter("order").getValue()
20:         from $g in $goalbase.getGoals("purchase_book")</facts>
21:     </beliefset>
22:     <belief name="time" class="long" updaterate="1000">
23:       <fact>System.currentTimeMillis()</fact>
24:     </belief>
25:   </beliefs>
26:
27:   <goals>
28:     <achievegoal name="purchase_book" recur="true" recurdelay="10000">
29:       <parameter name="order" class="Order" />
30:       <targetcondition>Order.DONE.equals($goal.order.getState())</targetcondition>
31:       <failurecondition>$beliefbase.time > $goal.order.getDeadline().getTime()</failurecondition>
32:     </achievegoal>
33:     <achievegoalref name="df_search"><concrete ref="dfcap.df_search" /></achievegoalref>
34:     <achievegoalref name="cnp_initiate"><concrete ref="procap.cnp_initiate" /></achievegoalref>
35:   </goals>
36:
37:   <plans>
38:     <plan name="gui_plan">
39:       <body>new GuiPlan(new Gui($agent.getExternalAccess(), true))</body>
40:     </plan>
41:     <plan name="purchase_book_plan">
42:       <parameter name="order" class="Order">
43:         <goalmapping ref="purchase_book.order" /></parameter>
44:       <body>new PurchaseBookPlan()</body>
45:       <trigger><goal ref="purchase_book" /></trigger>
46:     </plan>
47:   </plans>
48:
49:   <properties>
50:     <property name="service_seller"> SFipa.createAgentDescription(null,
51:       SFipa.createServiceDescription(null, "service_seller", null)) </property>
52:   </properties>
53:
54:   <initialstates>
55:     <initialstate name="default">
56:       <plans><initialplan ref="gui_plan" /></plans>
57:     </initialstate>
58:   </initialstates>
59:
60: </agent>
```

Figure 6 XML code (ADF) of the buyer agent

Both capabilities are included and assigned a local name within the capabilities section. Using a local name for capabilities allows for including the same type of capability under different identifiers.

The purchase_book goal leads to the execution of the purchase_book_plan (lines 41-46), as specified by the trigger of the plan (line 45). In addition to the trigger the plan head defines a parameter for the order the plan should handle (line 42). This parameter correlates to the triggering goal's order parameter and is hence directly mapped to it (line 43), i.e. the plan uses its parameter as usual but access will be redirected to the goal.

The advantage of using a goal mapping is that the plan will be decoupled from the goal and other goals can be handled by the same plan if corresponding mappings are defined. Furthermore, in the plan definition, a reference to the plan body is required, i.e. the plan class written in Java that will be executed at runtime whenever a triggering event occurs (line 44).

As there is no direct coupling between goals and plans, the developer can easily provide alternative ways for achieving goals, just by adding more plans capable of handling a goal to the plan library. E.g., in addition to the purchase_book_plan another plan might be introduced to handle the purchase_book goal. This plan could be based on a different negotiation strategy or allow buying from a different source, such as an online book store.

The PurchaseBookPlan Java class represents the plan body of the purchase_book_plan head defined in the ADF (lines 41-46). The order parameter is directly mapped from the order parameter of the goal. The implementation core of the plan is given by the steps of calculating the acceptable price, finding available sellers, initiating the contract net protocol, and finalizing the order. In this example, the acceptable price is a linear function given by the span of the price in the beginning of the request (start price) and the maximum price (limit) the agent is willing to pay for a book. It is adjusted by the time elapsed since the order is active and the ultimate deadline for the process of buying the book (lines 11-16).

To find available sellers, one needs to create an appropriate agent description that is sent to the directory facilitator (lines 18-27). The buyer searches for the service called service_seller. To achieve this, a df_search goal from the DF capability is created, provided with search conditions and dispatched for fulfillment. The result of this operation contains the description of all sellers on the platform that is known to the directory facilitator. From these descriptions, agent names and addresses are extracted and used in the next step of creating a contract net.

A call for proposals is prepared by the buyer and sent to the sellers using a contract net protocol (CNP, lines 29-41). Jadex provides this protocol in the protocols capability. It is accessed by the parameterized goal cnp_initiate. Before dispatching it, the agent supplements it with the title of the book sought after and the list of sellers that should be asked for it. A selection criterion is provided to choose proposals on the basis of the above calculated price (lines 33-36). A comparator allows differentiating among proposals and to choose the best suitable one (lines 37-40). In the case of the buyer, a proposal with a lower price outrivals any foregoing proposal with a higher price.

```
01: package jadex.examples.booktrading.buyer;
02:
03: import jadex.examples.booktrading.common.Order;
04: import jadex.adapter.fipa.*;
05: import jadex.planlib.ISelector;
06: import jadex.runtime.*;
07: import java.util.*;
08:
09: public class PurchaseBookPlan extends Plan {
10:   public void body() {
11:     // Get order properties and calculate acceptable price.
12:     Order order = (Order)getParameter("order").getValue();
13:     double time_span = order.getDeadline().getTime() - order.getStartTime();
14:     double elapsed_time = System.currentTimeMillis() - order.getStartTime();
15:     double price_span = order.getLimit() - order.getStartPrice();
16:     final int acceptable_price = (int)(price_span * elapsed_time / time_span) + order.getStartPrice();
17:
18:     // Find available seller agents.
19:     IGoal df_search = createGoal("df_search");
20:     df_search.getParameter("description").setValue(getPropertybase().getProperty("service_seller"));
21:     dispatchSubgoalAndWait(df_search);
22:     AgentDescription[] result = (AgentDescription[])df_search.getParameterSet("result").getValues();
23:     if (result.length == 0)
24:       fail();
25:     AgentIdentifier[] sellers = new AgentIdentifier[result.length];
26:     for(int i = 0; i < result.length; i++)
27:       sellers[i] = result[i].getName();
28:
29:     // Initiate a call-for-proposal.
30:     IGoal cnp = createGoal("cnp_initiate");
31:     cnp.getParameter("content").setValue(order.getTitle());
32:     cnp.getParameterSet("receivers").addValues(sellers);
33:     cnp.getParameter("selector").setValue(new ISelector() {
34:       public boolean isAcceptable(Object proposal) {
35:         return (proposal instanceof Integer)          && ((Integer)proposal).intValue() < acceptable_price;
36:       }});
37:     cnp.getParameter("comparator").setValue(new Comparator() {
38:       public int compare(Object o1, Object o2) {
39:         return ((Comparable)o2).compareTo(o1);
40:       }});
41:     dispatchSubgoalAndWait(cnp);
42:
43:     // If contract-net succeeds, store result in order object.
44:     order.setExecutionPrice((Integer)cnp.getParameter("result").getValue());
45:     order.setExecutionDate(new Date());
46:   }
47: }
```

Figure 7 Java code of the purchase book plan

The contract net succeeds, whenever there is at least one proposal that fulfills the selection criterion. On success, the result of the CNP protocol, representing the lowest bid price is saved the corresponding order and the state of the order is automatically marked as finalized (lines 43-45). In the other case, a subgoal failure exception is thrown by Jadex and the plan subsequently fails, which allows the agent to retry the goal of purchasing the book.

1.2.5 Implementing the seller agent

The seller agent is defined in the Seller.agent.xml ADF (see Figure 8). The user of a seller agent can issue new sell book orders from the Gui. In contrast to the active buyer agent, the seller agent is passive in the sense, that it registers itself as a service and waits for requests of buyer agents to achieve its sell book goals. The seller agent therefore implements the responder side of the contract-net protocol. For the contract-net responder side, there is also a ready to use implementation in the Jadex protocols capability.

As you can see in Figure 8, up to line 17, the seller agent has some commonalities with the buyer agent (see Figure 6). E.g., the time and orders beliefs and the sell_book goal are just like the time and orders beliefs and the purchase goal of the buyer. The seller has two additional goal declarations, cnp_make_proposal and cnp_execute_task (line 18, 19), which are related to the contract-net handling and one df_keep_registered goal (line 20) for publishing the seller service. The cnp_make_proposal goal is automatically created by the protocols capability, whenever the agent receives a call-for-proposal (CFP) message. The goal is handled by the make_proposal plan (lines 25-38), which checks if there is a suitable order in the agent's beliefs that matches the CFP. The plan will generate a proposal that is stored in the corresponding plan parameter (lines 27, 28). In addition to the proposal, which is sent to the buyer, the plan may store additional information in the proposal_info parameter, which will not be exposed to the buyer. When the plan finishes, the protocols capability automatically generates a proposal message and returns it to the buyer agent.

If the buyer accepts the proposal, the protocol capability creates the cnp_execute_task goal, which is handled by the execute_order plan. To process the order, the plan can reuse the information produced by the make_proposal plan, which will be present in the proposal and proposal_info parameters (lines 40, 41). After processing the order, the plan stores result of the transaction (line 42), which is sent automatically to the buyer. Finally, in its initial state, the agent defines a df_keep_registered goal (lines 51-55). The description parameter contains the service description that is used to publish the seller service. The initial goal causes the seller service to stay registered at the DF once the seller agent is started.

The most complex part of the make_proposal plan is already contained in declaration in the ADF (Figure 8 lines 25-38). The parameter set suitableorders is initialized with an OQL query, which returns open orders matching the requested book's title. The orders are sorted by the remaining time, such that urgent orders are handled first (without considering the orders prices). The plan's Java code, shown in Figure 9, selects the first order (line 11) and returns a price, calculated according to the negotiation strategy (lines 12-14). The order and the calculated price are stored in the result parameters of the plan (lines 15, 16) and will be used by the protocols capability to generate a propose message. If the result parameters are not set, because no suitable order is found, the capability will answer with a refuse message automatically.

```
01: <agent … name="Seller" package="jadex.examples.booktrading.seller">
02:   <imports>…</imports>
03:   <capabilities>…</capabilities>
04:
05:   <beliefs>
06:     <belief name="time" class="long" updaterate="1000"><fact>System.currentTimeMillis()</fact></belief>
07:     <beliefset name="orders" class="Order">
08:       <facts evaluationmode="dynamic"> select $g.getParameter("order").getValue()
09:         from $g in $goalbase.getGoals("sell_book") </facts></beliefset>
10:   </beliefs>
11:
12:   <goals>
13:     <achievegoal name="sell_book" recur="true" recurdelay="10000">
14:       <parameter name="order" class="Order" />
15:       <targetcondition>Order.DONE.equals($goal.order.getState())</targetcondition>
16:       <failurecondition>$beliefbase.time > $goal.order.getDeadline().getTime()</failurecondition>
17:     </achievegoal>
18:     <achievegoalref name="cnp_make_proposal"><concrete ref="procap.cnp_make_proposal"/></achievegoalref>
19:     <achievegoalref name="cnp_execute_task"><concrete ref="procap.cnp_execute_task" /></achievegoalref>
20:     <maintaingoalref name="df_keep_registered"><concrete ref="dfcap.df_keep_registered" /></maintaingoalref>
21:   </goals>
22:
23:   <plans>
24:     <plan name="gui_plan"><body>new GuiPlan(new Gui($agent.getExternalAccess()))</body></plan>
25:     <plan name="make_proposal_plan">
26:       <parameter name="task" class="Object"><goalmapping ref="cnp_make_proposal.task" /></parameter>
27:       <parameter name="proposal" class="Object" direction="out">
28:         <goalmapping ref="cnp_make_proposal.proposal" /></parameter>
29:       <parameter name="proposal_info" class="Order" direction="out">
30:         <goalmapping ref="cnp_make_proposal.proposal_info" /></parameter>
31:       <parameterset name="suitableorders" class="Order">
32:         <values> select Order $order from $beliefbase.orders where $order.getTitle().equals($plan.task)
33:           && $order.getState().equals(Order.OPEN) order by ($beliefbase.time -$order.getStarttime())
34:           / ($order.getDeadline().getTime()-$order.getStarttime()) </values>
35:       </parameterset>
36:       <body>new MakeProposalPlan()</body>
37:       <trigger><goal ref="cnp_make_proposal" /></trigger>
38:     </plan>
39:     <plan name="execute_order_plan">
40:       <parameter name="proposal" class="Object"><goalmapping ref="cnp_execute_task.proposal" /></parameter>
41:       <parameter name="proposal_info" class="Object"><goalmapping ref="cnp_execute_task.proposal_info"/></parameter>
42:       <parameter name="result" class="Object"><goalmapping ref="cnp_execute_task.result" /></parameter>
43:       <body>new ExecuteOrderPlan()</body>
44:       <trigger><goal ref="cnp_execute_task" /></trigger>
45:     </plan>
46:   </plans>
47:
48:   <initialstates>
49:     <initialstate name="default">
50:       <goals>
51:         <initialgoal ref="df_keep_registered">
52:           <parameter ref="description"><value> SFipa.createAgentDescription(null,
53:             SFipa.createServiceDescription("sell", "service_seller", "UniHH")) </value></parameter>
54:           <parameter ref="leasetime"><value>200000</value></parameter>
55:         </initialgoal>
56:       </goals>
57:       <plans><initialplan ref="gui_plan" /></plans>
58:     </initialstate>
59:   </initialstates>
60: </agent>
```

Figure 8 XML code (ADF) of the seller agent

```
01: package jadex.examples.booktrading.seller;
02:
03: import jadex.examples.booktrading.common.Order;
04: import jadex.runtime.Plan;
05:
06: public class MakeProposalPlan extends Plan {
07:   public void body() {
08:     // Check if there is a suitable order and return it.
09:     Order[] orders = (Order[])getParameterSet("suitableorders").getValues();
10:     if (orders.length > 0) {
11:       Order order = orders[0];
12:       double discount = (double)(order.getDeadline().getTime() - System.currentTimeMillis())
13:         / (double)(order.getDeadline().getTime() - order.getStarttime());
14:       int price = (int)(order.getLimit() - (order.getLimit() - order.getStartPrice()) * discount);
15:       getParameter("proposal").setValue(new Integer(price));
16:       getParameter("proposal_info").setValue(order);
17:     }
18:   }
19: }
```

Figure 9 Java code of the make proposal plan

When the proposal is accepted by the buyer, it will answer with an accept-proposal. In this case, the protocols capability will create a cnp_execute_task goal at the seller side to finish the transaction between buyer and seller. To achieve this goal, it is handled by the execute_order plan. In the plan body (Figure 10), the seller uses the calculated price of the proposal (line 10) and updates the order object accordingly (lines 12, 13). In addition, the price is used as the result of the transaction, which is then sent automatically as an inform message to the buyer.

```
01: package jadex.examples.booktrading.seller;
02:
03: import jadex.runtime.Plan;
04: import jadex.examples.booktrading.common.Order;
05: import java.util.Date;
06:
07: public class ExecuteOrderPlan extends Plan {
08:   public void body() {
09:     // Buyer agreed to our proposal: store execution price and date.
10:     Integer price = (Integer)getParameter("proposal").getValue();
11:     Order order = (Order)getParameter("proposal_info").getValue();
12:     order.setExecutionPrice(price);
13:     order.setExecutionDate(new Date());
14:     getParameter("result").setValue(price);
15:   }
16: }
```

Figure 10 Java code of the execute order plan

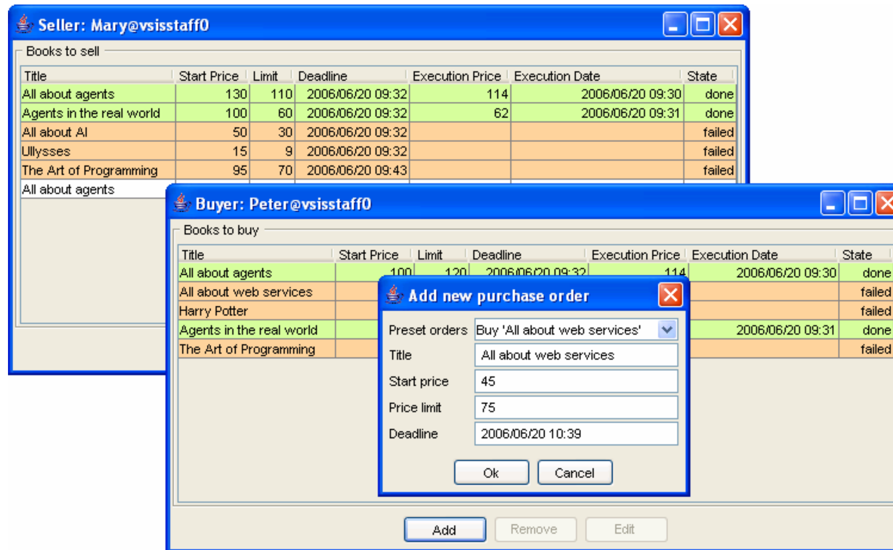

Figure 11 Screenshot of the buyer (front) and seller Gui

1.2.6 Adding the user interface

Until now, we did not consider in detail how a human user can interact with the buyer and seller implementations. One advantage of the explicit representation of the agent's mental state in terms of beliefs, goals, and plans is that these features can be used to underpin the user interface of an agent. For the buyer and seller this means that we can use the orders belief set to create a list of orders, and we can create purchase or sell goals when the user enters a new order. Because the mental state is managed by the Jadex reasoning engine automatically, there will be no dependencies between the Gui code and the code for buying or selling books.

The Gui code is realized in two parts, which are used by both the buyer and the seller agent. The Gui plan is responsible for opening and closing the Gui window and for updating the Gui whenever the orders belief set changes. The Gui window itself contains a graphical representation of the orders (see Figure 11) and allows the user to add, edit, and remove orders, thereby modifying the goals of the agent.

As you can see in Figure 6 and Figure 8 (lines 39 resp. 24), the gui_plan is initialized with an instance of the Gui class representing the window to be shown. The plan is included in the initial state of the agents, such that the plan is executed when the agent is born. The Java code of the GuiPlan class is shown in Figure 12. When the plan is executed, the body() method is called. First the window is opened (lines 16, 17) and a listener is added, which kills the agent, when the user closes the window (lines 18-22). The loop at the end of the body() method (lines 25-28) continuously refreshes the Gui Window, which will then update its presentation based on the current orders. To avoid refreshing the Gui too often, the waitForBeliefSetChange() method is used (line 27), which puts the plan to sleep until the next change of the orders belief set. Finally, the plan overrides the aborted() method (lines 31-38), which is called when the plan is aborted, e.g. when the agent dies. In the method, the window is closed. SwingUtilities.invokeLater() is used to execute the window closing on the Swing thread as demanded by Swing.

```
01: package jadex.examples.booktrading.common;
02:
03: import …
06:
07: public class GuiPlan extends Plan {
08:   private Gui gui;
09:
10:   public GuiPlan(Gui gui) {
11:     this.gui = gui;
12:   }
13:
14:   public void body() {
15:     // Show the gui.
16:     gui.pack();
17:     gui.setVisible(true);
18:     gui.addWindowListener(new WindowAdapter() {
19:       public void windowClosing(WindowEvent e) {
20:         getExternalAccess().killAgent();
21:       }
22:     });
23:
24:     // Update the gui whenever the orders change.
25:     while (true) {
26:       gui.refresh();
27:       waitForBeliefSetChange("orders");
28:     }
29:   }
30:
31:   public void aborted() {
32:     // Gui must be closed from the Swing thread.
33:     SwingUtilities.invokeLater(new Runnable() {
34:       public void run() {
35:         gui.setVisible(false);
36:       }
37:     });
38:   }
39: }
```

Figure 12 Java code of the gui plan

The Java code of the Gui is mostly plain Java and Swing code. We will show here only the parts of the code, which are related to agent programming. You can find the complete source code in the Jadex distribution. The relevant excerpt of the Gui class is shown in Figure 13. The Gui extends JFrame and contains the user interface components shown earlier in the screenshot. When the user clicks the Add button, an input dialog is created allowing the user to enter title, start price, limit, and deadline for a new buy or sell order or just select one of several predefined orders from a choice box. The values are used to create a new order to fill in the order parameter of a new purchase_book or sell_book goal (lines 21-25). To create a new goal (lines 26-27), the Gui class uses the external access interface of the agent, supplied in the constructor (Figure 6 and Figure 8, lines 39 resp. 24). The external access represents a technical means allowing external threads to invoke the agent's BDI functionality such as accessing beliefs or dispatching events and goals, thereby guaranteeing the agent's consistency. In this case the external access is also used from the Gui thread to dispatch the goal as a new top-level goal of the agent (line 28).

```
01: package jadex.examples.booktrading.common;
02:
03: import …
04:
05: public class Gui extends JFrame {
06:   private List orders;
07:   private AbstractTableModel items;
08:   private DateFormat dformat;
09:   private IExternalAccess agent;
10:   ...
11:
12:   public Gui(IExternalAccess agent, final boolean buy) {
13:     this.agent = agent;
14:     ...
15:
16:     JButton add = new JButton("Add");
17:     final InputDialog dia = new InputDialog(buy);
18:     add.addActionListener(new ActionListener() {
19:       public void actionPerformed(ActionEvent e) {
20:         try {
21:           String title = dia.title.getText();
22:           int limit = Integer.parseInt(dia.limit.getText());
23:           int start = Integer.parseInt(dia.start.getText());
24:           Date deadline = dformat.parse(dia.deadline.getText());
25:           Order order = new Order(title, deadline, start, limit, buy);
26:           IGoal purchase = Gui.this.agent.createGoal(buy?"purchase_book":"sell_book");
27:           purchase.getParameter("order").setValue(order);
28:           Gui.this.agent.dispatchTopLevelGoal(purchase);
29:           orders.add(order);
30:           items.fireTableDataChanged();
31:         } catch (Exception e) {
32:           JOptionPane.showMessageDialog(Gui.this, "Input error", "Input error", 0);
33:         }
34:       }
35:     });
36:   }
37:
38:   public void   refresh() {
39:     // Use invoke later as refresh() is called from plan thread.
40:     SwingUtilities.invokeLater(new Runnable() {
41:       public void run() {
42:         Order[] aorders = (Order[])agent.getBeliefbase().getBeliefSet("orders").getFacts();
43:         for(int i=0; i<aorders.length; i++)
44:           if(!orders.contains(aorders[i]))
45:             orders.add(aorders[i]);
46:         items.fireTableDataChanged();
47:       }
48:     });
49:   }
50: }
```

Figure 13 Java code of the gui window

Whenever the orders belief set changes, e.g., because purchase_book or sell_book goals fail or succeed, the gui_plan calls the refresh() method of the Gui object. In this method (lines 38-49), the Gui uses the external access interface to extract the current orders from the beliefbase (line 42). New orders are stored in the orders list (line 45) and the display is updated accordingly (line 46). Because the method is called from the plan thread but has to update the Swing objects, such as the JTable displaying the orders, the code of the method has to be enclosed in SwingUtilities.invokeLater().

1.3 Summary

In this chapter the Jadex BDI reasoning engine is presented. It allows for developing rational agents using mentalistic notions in the implementation layer. Outstanding features of the engine are the full support for the two phases of the practical reasoning process (goal deliberation and means-end reasoning), and the explicit representation of mental attitudes, such as beliefs, goals, and plans.

Moreover, the engine is specifically designed to build on and extend traditional software engineering principles and practices. Therefore, the engine is independent of the underlying platform (e.g. JADE) and can be flexibly deployed on arbitrary middleware infrastructure. For the programming of agents, the engine relies on established techniques, such as Java and XML, allowing easy agent development in mature state-of-the-art development environments like eclipse and IntelliJ IDEA. To further simplify the development task, the Jadex distribution includes a rich suite of runtime tools for administration and debugging purposes as well as a library of ready-to-use generic functionalities provided by several agent modules (capabilities).

Furthermore, in this chapter the development of an agent application based on Jadex is illustrated by realizing the book-trading scenario with goal-oriented agents. For a deeper understanding of the system-to-be the scenario has been modeled using the Tropos methodology in terms of actors, goals, and strategic dependencies between the corresponding actors. In subsequent refinement steps the initial model has been elaborated showing the goal/plan hierarchies of the participating agent types. This refined Tropos actor/goal model represents an ideal starting point for the implementation within Jadex as an intuitive mapping from the model to the agent definitions can be performed. The implementation of the buyer and seller agents is rather compact as generic functionalities such as searching the DF or initiating a contract-net protocol can be reused via capabilities. Hence the resulting agents only consist of semantically important aspects with respect to the book-trading scenario abstracting away from low-level implementation issues.

The Jadex BDI reasoning engine enables the construction of complex real-world applications by exploiting the ideas of intentional systems going back to Dennett and McCarthy. The high-level intentional view of the system-to-be can be preserved in the Jadex implementation leading to easy understandable and effectively manageable solutions.

1.4 References

[1] Bratman, M.: Intention, Plans, and Practical Reason. Harvard University Press, 1987.

[2] Braubach, L.; Pokahr, A.; Lamersdorf, W.: Jadex: A BDI-Agent System Combining Middleware and Reasoning, in: M. Walliser, S. Brantschen, M. Calisti, T. Hempfling (eds.): Whitestein Series in Software Agent Technologies, Birkhäuser-Verlag, Springer Science+Business Media, Berlin, New York, 2005.

[3] Braubach, L.; Pokahr, A.; Lamersdorf, W.: Extending the Capability Concept for Flexible BDI Agent Modularization, in: R.H. Bordini et al. (eds.): 'Programming Multi-agent Systems (PROMAS 2005)', Springer Verlag, Berlin Heidelberg, New York, 2006, pp. 139-155.

[4] Braubach, L.; Pokahr, A.; Lamersdorf, W.; Moldt, D.: Goal Representation for BDI Agent Systems, in: R.H. Bordini et al. (eds.): Proc. 2nd International Workshop on

Programming Multiagent Systems, Languages and Tools (PROMAS 2004), 3rd International Joint Conference on Autonomous Agents & Multi-Agent Systems (AA-MAS'04), New York, USA, Springer-Verlag, Berlin, New York, Lecture Notes in Computer Science, 2005, pp. 46-67.

[5] Busetta, P.; Howden, N.; Rönnquist, R.; Hodgson, A.: Structuring BDI Agents in Functional Clusters. In: Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL'99), LNCS 1757. Springer, 2000, pp. 277-289.

[6] Dennett, D.: The Intentional Stance. Bradford Books, 1987.

[7] Giorgini, P.; Kolp, M.; Mylopoulos, J.; Pistore, M.: The Tropos Methodology. In: Methodologies and Software Engineering for Agent Systems. Kluwer Academic Publishers, 2004, pp. 89-206.

[8] Henderson-Sellers B.; Giorgini, P.; Agent-Oriented Methodologies. IDEA Group Publishing. 2005.

[9] McCarthy, J.; Ascribing mental qualities to machines. In: Philosophical Perspectives in Artificial Intelligence. Humanities Press, pp. 161-195, 1979.

[10] Newell A.; Unified Theories of Cognition. Harvard University Press. 1990.

[11] Pokahr, A.; Braubach, L.; Lamersdorf, W.: The Jadex BDI Reasoning Engine, in: R. Bordini, M. Dastani, J. Dix and A. El Fallah Seghrouchni (eds.): Programming Multi-Agent Systems, Springer Verlag, Berlin, New York, 2005.

[12] Pokahr, A.; Braubach, L.; Lamersdorf, W.: A Flexible BDI Architecture Supporting Extensibility, in: Skowron, A.; Barthes, J.-P.; Jain, L.; Sun, R.; Morizet-Mahoudeaux, P.; Liu, J.; Zhong, N. (eds.): '2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-2005)', IEEE Computer Society, 2005, pp. 379-385.

[13] Pokahr, A.; Braubach, L.; Lamersdorf, W.: A Goal Deliberation Strategy for BDI Agent Systems, in: Eymann, T.; Klügl, F.; Lamersdorf, W.; Klusch, M.; Huhns, M. (eds.): ‚Procs. Third German Conference on Multi-Agent System TEchnologieS (MATES-2005)', Springer Verlag, Berlin, New York, 2005, pp. 82-94.

[14] Rao, A.; Georgeff, M.: BDI Agents: from theory to practice. In Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95). The MIT Press, pp. 312-319.

[15] Sudeikat, J.; Braubach, L.; Pokahr, A.; Lamersdorf, W.: „Evaluation of Agent–Oriented Software Methodologies – Examination of the Gap Between Modeling and Platform", in: Giorgini, P.; Müller, J.; Odell, J. (eds.): ‚Agent-Oriented Software Engineering V (AOSE-2004)', Springer-Verlag, Berlin, New York , 2005 pp. 126-141.

[16] Wooldridge, M.: Reasoning about Rational Agents. The MIT Press. 2000.