Jadex
Tutorial

Release 0.96
15. June 2007
http://vsis-www.informatik.uni-hamburg.de/projects/jadex/

Lars Braubach
Alexander Pokahr

Distributed Systems Group
University of Hamburg, Germany
http://vsis-www.informatik.uni-hamburg.de

If you have support questions about Jadex please use the sourceforge help forum and mailing list for that purpose (available at http://sourceforge.net/projects/jadex/).

Table of Contents

Chapter 1. Introduction

Jadex is a Belief-Desire-Intention (BDI) reasoning engine for intelligent agents. The term reasoning engine means that it can be used together with different kinds of (agent) middleware providing basic agent services such as a communication infrastructure and management facilities. Currently, two mature adapters are available. The first adapter is available for the well-known open-source JADE multi-agent platform [Bellifemine et al. 2007] and the second one is the Jadex Standalone adapter which is a small but fast environment with a minimal memory footprint. In this tutorial the Jadex Standalone adapter is used, but in principle the used adapter is not of great importance as it does not change the way Jadex agents are programmed or way the Jadex tools are used.

The concepts of the BDI-model initially proposed by Bratman [Bratman 1987] were adapted by Rao and Georgeff [Rao and Georgeff 1995] to a more fomal model that is better suitable for multi-agent systems in the software architectural sense. Systems that are built on these foundations are called Procedural Reasoning Systems (PRS) with respect to their first representative. Jadex builds on experiences gained from leading existing BDI systems such as JACK [Winikoff 2005] and consequently improves previously not-addressed BDI weaknesses like the concurrent handling of inconsistent goals with built-in goal deliberation [Pokahr et al. 2005a].

This tutorial is a good starting point for agent developers, that want to learn programming Jadex BDI agents in small hands-on exercises. Each lesson of this tutorial covers one important concept and tries to illustrate why and especially how the concept can be used in Jadex. In the following Chapter 2, *Starting an Agent* it is described how to setup the Jadex environment properly and how to start a simple agent. It is explained step by step how to handle plans (Chapter 3, *Using Plans*), beliefs (Chapter 4, *Using Beliefs*) and goals (Chapter 6, *Using Goals*) and how these elements can be composed (Chapter 5, *Using Capabilities*) into reusable agent modules. Another lesson covers some aspects about information exchange on the intra and inter-agent level and builds up a multi-agent scenario Chapter 7, *Using Events*. Thereafter, in Chapter 8, *External Processes* the integration of Jadex agents with external processes is exemplarily explained. Finally a conclusion and an outlook is given in Chapter 9, *Conclusion and Outlook*. After having worked through this tutorial the reader should be familiar with all basic agent concepts provided by Jadex. Whenever the reader encounters facts that are not explained in detail here but may need some elaboration for a thorough understanding further reading in the Jadex user guide [Jadex User Guide] is recommended. If you are interested in less technical documentation you may also consider reading about Jadex in one of these book chapters [Pokahr et al. 2005c][Braubach et al. 2005a].

1.1. Application Context

In this tutorial a simple translation agent for single words will be implemented. This agent has the basic task to handle translation requests and produce for a given term in some language the translated term in the desired target language. This base functionality will be extended in the different exercises, but it is not our goal to build up a translation agent, that combines all the extensions, because this would lead to difficulties concerning the complexity of the agent. Instead this tutorial will concentrate on setting up simple agents that explain the Jadex concepts step by step.

1.2. How to Use This Tutorial

- Work through the exercises in order, because later exercises require knowledge from the earlier ones.
- Don't destroy your solutions of an exercise by modifying the old files. The different exercises often use the plans and agent description files (ADF) of a preceeding exercise. Copy all files and apply a simple naming scheme which contains the name of the exercise in the plan and ADF file names, e.g. the ADF in the exercise A1 is called TranslationA1.agent.xml and in exercise B1 TranslationB1.agent.xml.

- Help us to make this tutorial better with your feedback. When you find errors or have problems that are directly concerned with the exercise descriptions feel free to let us know.
- Whenever you encounter problems with Jadex we would be happy to help you. Please use therefore primarily the sourceforge help forum[1] available on the Jadex sourceforge.net page. There is also a Jadex mailing list[2] that can be used for asking questions about Jadex.

[1] http://sourceforge.net/forum/forum.php?forum_id=274112
[2] http://sourceforge.net/mail/?group_id=80240

Chapter 2. Starting an Agent

2.1. Exercise A1 - Jadex Platform

Setting up the Jadex environment properly is pretty easy and can be done in a few simple steps. Generally, Jadex is realized as reasoning engine meaning that it can be used on top of different (agent) middlewares. In this tutorial we will use the Standalone version of Jadex. The Jadex distribution should be extracted to some local directory, called JADEX_HOME here.

Set the Java CLASSPATH variable properly by adding the following jars:

The actual filenames of the jar files may differ slightly due to versioning conventions.

Necessary libraries for the Jadex kernel:

- JADEX_HOME/lib/jadex_rt.jar
- JADEX_HOME/lib/jibx-run.jar
- JADEX_HOME/lib/xpp3.jar
- JADEX_HOME/lib/nuggets.jar
- JADEX_HOME/lib/janino.jar

Necessary libraries for the Jadex Standalone platform:

- JADEX_HOME/lib/jadex_standalone.jar

Necessary libraries for the Jadex tools:

- JADEX_HOME/lib/jadex_tools.jar
- JADEX_HOME/lib/GraphLayout.jar
- JADEX_HOME/lib/jhall.jar

The command for launching the Jadex Standalone platform is:

java [conf=<platform.properties>] jadex.adapter.standalone.Platform
If the configuration file is not specified, the default configuration (`jadex.properties`) will be read from the root directory of the jadex_standalone.jar.

Create a simple Jadex agent. Open a source code editor or an IDE of your choice and create a new agent definition file (ADF) called `TranslationA1.agent.xml` (cf. Figure 2.1, “A1 XML ADF”). We recommend using eclipse with web-tools plug-in for editing ADFs or some other advanced XML-Editor such as the commercial application Altova XMSpy. In this file all important agent startup properties are defined in a way that complies to the Jadex schema specification. First property of the agent is its type name which must be the same as the file name (similar to Java class files), in this case it is set to TranslationA1. Additionally you can specify a package attribute, which has a similar meaning as in Java programs and serves for grouping purposes only (you will need to alter the package name with respect to your actually used directory structure). All plans and other Java classes from the agent's package are automatically known and need not to be imported via an import tag.

```
 1 <!--
 2     TranslationAgent
 3 -->
 4 <agent xmlns="http://jadex.sourceforge.net/jadex"
 5     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 6     xsi:schemaLocation="http://jadex.sourceforge.net/jadex
 7                         http://jadex.sourceforge.net/jadex-0.96.xsd"
 8     name="TranslationA1"
 9     package="jadex.tutorial">
10 </agent>
```

Figure 2.1. A1 XML ADF

Start your first Jadex agent. Start the Jadex platform with the command explained above. After some short time the Jadex Control Center should show up with its user interface. Within the "Start Agents" panel browse the directories and select your TranslationA1.agent.xml. The effect of choosing the input file is, that the agent model is loaded. When it contains no errors, the description of the model, taken from the XML comment above the agent tag, is shown in the description view. In case there are errors in the model, correct the errors shown in the description view and press “reload”. Below the file name, the agent name and its default configuration are shown. After pressing the start button the new agent should appear in the agent tree. It is also possible to start an agent simply by double-clicking it in the model tree or by activating the popup menu in the tree on an agent model and the select the start option. Curiously, you can start a second JCC by choosing it from: **jadex/tools/jcc/JCC.agent.xml** and giving it a name like JCC2.

2.2. Exercise A2 - Creating a Project

The JCC provides a project management facility that simplifies working on specific applications. Basically, a project contains settings about the used project folders as well as miscellaneous tool and user interface settings. All your settings - as window settings, added paths and so on - will be stored in a .prj-file and additional properties-files will be created automatically to store the individual settings for the plugins you are using. In order to avoid making all the settings in the JCC on every startup in this exercise it will be shown in this exercise how a project is set-up.

To create a project you have to choose “New Project” in the “File”-menu of the JCC. For saving the project on disk select “Save Project As...” from the “File”-menu. In the appearing file requester, please choose an arbitrary name for your project and choose the folder in which the project files shall be saved.

Whenever you choose “Save Project” from the “File”-menu the current settings will be saved to disk. Additionally, the settings are also saved automatically when shutting down the JCC via its window close button or “Exit” from the “File”-menu.

You can also use multiple projects with different settings. To switch projects simply click on “Open Project” in the “File”-menu and choose the project you want to work with.

Verify project behaviour. In order to verify that you project has been set up correctly you should perform some visible changes such as changing the JCC's window size or switching to another plugin than the starter. After that you should shut down the JCC and platform and restart it. If the project has been created correctly, the JCC will show up in exactly the same state you left it, i.e. the last active plugin will be activated and the gui settings are remembered, too.

2.3. Exercise A3 - Execute Examples

The Jadex distribution already contains a couple of examples. The objective of this exercise consists in trying out the examples and also in roughly understanding their application purpose. Open the JCC and select the starter view as described in exercise A1 and click on the button  to add a new path or jar to the project. A file chooser window will open and you should choose the jadex_examples.jar from your “JADEX_HOME/lib”-folder. Please note that you should not add arbitrary paths to your project as these paths are added to the project specific classpath. Instead, only the root directories of Java packages (such as a "classes" or "build") should be added to the project.

The content of the new jar will be added as new node to the model tree. At the bottom you can now see this jar-file beeing scanned (is the scanning option has not been disabled). Next, you can expand the model tree by double clicking on the corresponding node  , which represents the jar-file. After openening the folders “jadex” and “examples” you will see the different example folders containing several different kinds of single- and multi-agent applications. Concretely the following applications are available:

- **Blackjack.**  A distributed Blackjack implementation that allows agents as well as human players playing the famous Blackjack card game. In each round the participants bet virtual money and are rewarded if they win against the dealer.
- **Blocksworld.**  In the well-known blocksworld scenario the problem consists of stacking differently colored blocks on a table. The objective for the agent is here to restack the blocks (and block towers) on the table in a way that the target configuration is achieved. This target configuration can be entered via the gui by the human user.
- **Booktrading.**  In the booktrading scenario human book buyers and sellers delegate buy resp. sell orders to their corresponding agents. The agents negotiate using a contract-net protocol at a distributed market place in order to fulfill the strategic goals of their principals (low/high prices, deadline etc.).
- **Cleanerworld.**  The cleanerworld example deals with the exploring and cleaning of an unknown environment. Basically, a cleaner robot has the tasks to search for waste, pick it up and bring it to a nearby wastebin. Additionally, it has to patrol in the environment at night and must always ensure that its battery does not run low.
- **Garbagecollector.**  This example is a simpler version of the cleaner world. Cleaner robots have a predefined route which the use for picking up waste. When they found a piece of waste they carry it to a burner agent nearby which is responsible for burning it.
- **Helloworld.** The helloworld example contains a simple agent that when started prints out some welcome message and terminates itself after that.
- **Hunterprey.**  This is a quite complex multi-agent simulation which realizes a variant of the hunter prey scenario. In general there are two different kinds of animals sheeps (the preys) and wulfs (the hunters). Both animals can get points by comsuming food. A sheep looks for salad whereas the wolfs hunt sheeps.
- **Marsworld.**  In this scenario three different kind of robots operate on mars and search for ore. The main strategic objective is to carry as much ore as possible to the homebase of the robots. For this purpose sentry robots scan promising locations and commission production robots. These production robots convey the ore and order carry robots which have the task to bring the ore to the homebase.

- **Ping.** The ping example shows how an agent can send a simple "ping"-message to another agent which reacts on this message with an "alive"-reply.

- **Puzzle.**  The puzzle agent has the task to solve a complex board game. The main goal is to swap the positions of the red and white pegs whereby the pieces can only move forward one field and can jump over one opposite peg. The puzzle agent solves this game by using an goal-oriented form of backtracking.

Starting an example can in most cases be done by opening the corresponding folder and searching for a so called "manager" agent. Such managers have the purpose to start-up all relevant application agents. In case there is no manager agent the example is so simply that it can be started by executing the contained application agent(s) directly (e.g. in the puzzle example you can directly start Sokrates). To start an agent you have to activate the starter perspective and double click on the agent in the model tree or click once and use the start button at the right hand side.

If you double click e.g. on the “Blocksworld.agent.xml” the blocksworld example will be started and you should see the following gui:

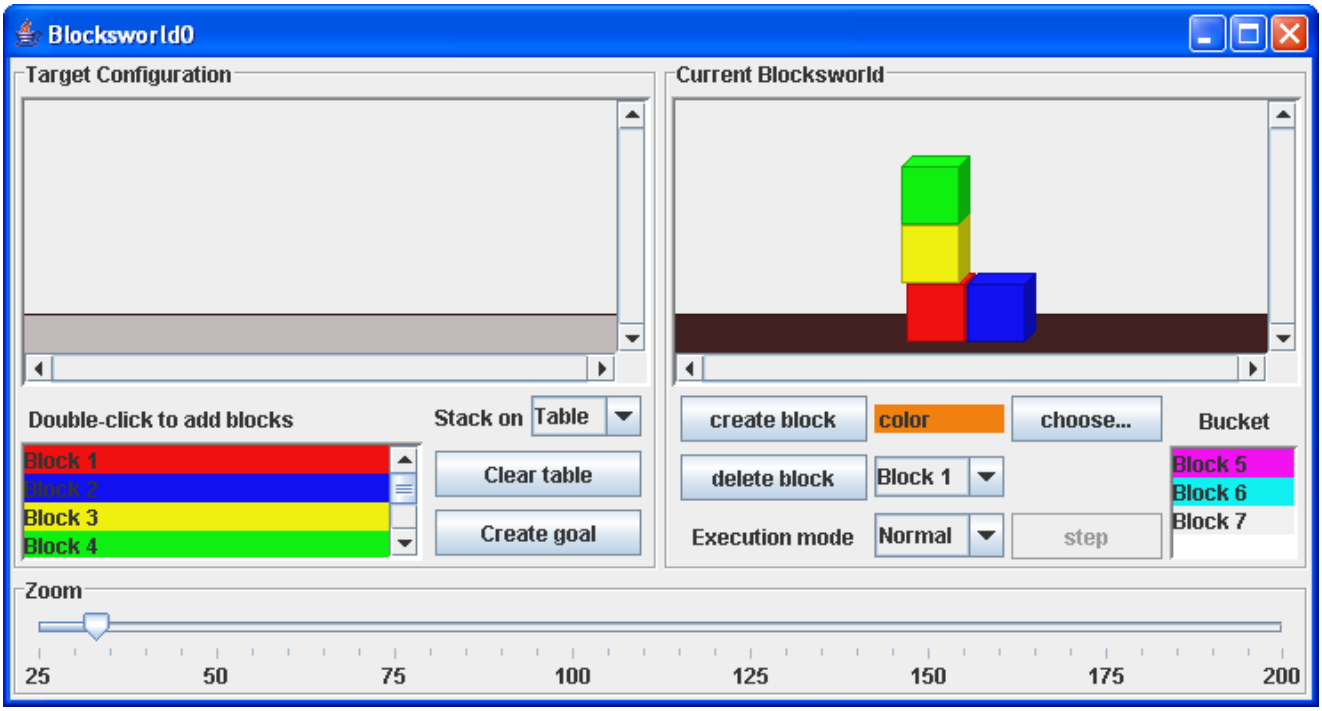

Figure 2.2. The blocksworld example

In the same way you can add other paths and execute your own agents later. In this case you will not add jar-files, but the root directory of your packages.

Explain example behaviour. Choose one of the more complex examples for a more detailed analysis. Select the manager agent of the example and read through its documentation shown in the starter panel. Then look into the selected example directory and read also the documentation of the other application agents which belong to that manager. Finally, open the source code of these agents (*.agent.xml) in your source code editor and try to grasp roughly of what they are comprised. Write down a (simple!) explanation how the multi-agent system and the involved agents work.

Chapter 3. Using Plans

Plans play a central role in Jadex, because they encapsulate the recipe for achieving some state of affair. Generally, a plan consists of two parts in Jadex. The plan body is a standard Java class that extends a predefined Jadex framework class (`jadex.runtime.Plan` or `jadex.runtime.MobilePlan`) and has at least to implement the abstract `body()` resp. `action()` method which is invoked after plan instantiation. The plan body is associated to a plan head in the ADF. This means that in the plan head several properties of the plan can be specified, e.g. the circumstances under which it is activated and its importance in relation to other plans.

In contrast to other well-known PRS-like systems, Jadex supports two styles of plans. A so called *service plan* is a plan that has service character in the sense that a plan instance of the plan is usually running and waits for service requests. It represents an easy way to react on service requests sequentially without the need to synchronize different plan instances for the same plan. Therefore a service plan can setup its private event waitqueue and receive events for later processing, even when it is working at the moment.

A so called *PRS-style or passive plan* is the normal version of a plan, as can be found in all other PRS-systems. This means that usually such a plan is only running, when it has a task to achieve. For this kind of plan the triggering events and goals must be specified in the agent definition file to let the agent know what kinds of events this plan can handle. When an agent receives an event, the BDI reasoning engine builds up the so called applicable plan list (that are all plans which can handle the current event or goal) and candidate(s) are selected and instantiated for execution. PRS-style plans are a good choice, when the parallel execution of one kind of task is needed or is at least not disturbing. For more detailed information about plans have a look in the [Jadex User Guide].

Often a plan does some action and then wants to wait until the action has been done before continuing (e.g. dispatching a subgoal, sending a message and waiting for the reply). Therefore a plan can use one of the various `waitFor()` methods, that come in quite different flavors. Coming back to the examples mentioned, e.g. the `dispatchSubgoalAndWait(IGoal subgoal [, long timeout])` can be used to dispatch a subgoal and wait for its completion (optionally with some timeout). Similar, for sending a message and waiting for a reply the `sendMessageAndWait(IMessageEvent me [, long timeout])` method can be used. For an extensive overview of all available methods, please refer to the [Jadex User Guide] or the API documentation[1].

3.1. Exercise B1 - Service Plans

In this exercise we will use a service plan for translating words from English to German. Create a new TranslationB1.agent.xml file by copying the TranslationA1.agent.xml file and modify all occurrences of "A1" to "B1".

Create a new file called EnglishGermanTranslationPlanB1.java responsible for a basic word translation with the following properties:

- Create the plan as extension to the `jadex.runtime.Plan` class:

```
public class EnglishGermanTranslationPlanB1 extends Plan {
    // Plan attributes.

    public EnglishGermanTranslationPlanB1() {
        // Initialization code.
    }
```

[1] http://vsis-www.informatik.uni-hamburg.de/projects/jadex/jadex-0.96x/kernel/index.html

```
    public void body() {
        // Plan code.
    }
}
```

- Import the needed classes:

```
import java.util.*;
import jadex.runtime.*;
```

- Let the no argument constructor print out the text "Created:"+this.

- Implement the plan's `body()` method as infinite loop. At the beginning of this loop the plan should wait for translation requests:

```
IMessageEvent me = waitForMessageEvent("request_translation");
```

  Instead of performing a database query let us use a simple hashmap for the word lookup. The creation and initialization of this word table with a few word pairs can already be done in the constructor. As result the plan should print `"Translating from English to German: "+eword+" - "+gword` or `"Sorry word is not in database: "+eword`. To get the content from the request-event use `me.getContent()`.

Add the plan to the agent by putting it into the agent definition file:

- Therefore, a new plans section is introduced (lines 13-20), in which all plans for the agent have to be declared. In this simple example only one plan named here "egtrans" is added (lines 14-19). In line 15 the Java expression for creating the plan body is stated. Note that it is allowed to use any Java mechanism to create the plan (e.g. one could use a static method instead of a constructor call). In the next lines the plan's waitqueue is declared to handle all message events of type "request_translation". This means that the plan has its own event waitqueue in which all matching events are dispatched, even when the plan is busy and currently waits for other events. These events are collected in its queue till it calls a suitable `waitFor()` matching one of the collected events. In this case this collected event is directly dispatched to the plan.

- The plan should be started when the agent is born. For this purpose a configuration has to be declared within the ADF (lines 30-36). It is sufficient in this case to define one configuration (named "default") with an initial plan (lines 31-35). The initial plan simply references the plan for which an instance should be created (line 33).

- Besides the introduction of the new plan we also need to make explicit what exactly a request_translation event means. For this purpose a new events section is introduced (lines 22-28). In this section the request_translation event is declared being a message event with one parameter (lines 23-27). This parameter specifies that its performative has the fixed value request. Whenever the agent receives a message it will search its declared events for the best matching event type. In this case all messages with performative request it will be treated as request_translation events.

```
 1 <!--
 2     Creating an initial plan.
 3     The agent has one initial plan (created when the agent is born)
 4     for translating words from English to German.
 5 -->
 6 <agent xmlns="http://jadex.sourceforge.net/jadex"
 7     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 8     xsi:schemaLocation="http://jadex.sourceforge.net/jadex
 9                         http://jadex.sourceforge.net/jadex-0.96.xsd"
10     name="TranslationB1"
```

```
11     package="jadex.tutorial">
12
13     <plans>
14         <plan name="egtrans">
15             <body>new EnglishGermanTranslationPlanB1()</body>
16             <waitqueue>
17                 <messageevent ref="request_translation"/>
18             </waitqueue>
19         </plan>
20     </plans>
21
22     <events>
23         <messageevent name="request_translation" direction="receive" type="fipa">
24             <parameter name="performative" class="String" direction="fixed">
25                 <value>jadex.adapter.fipa.SFipa.REQUEST</value>
26             </parameter>
27         </messageevent>
28     </events>
29
30     <configurations>
31         <configuration name="default">
32             <plans>
33                 <initialplan ref="egtrans"/>
34             </plans>
35         </configuration>
36     </configurations>
37
38 </agent>
```

Figure 3.1. B1 XML ADF

Start and test the agent. Create a translation agent via the Control Center and observe the standard output, if the initial plan is created at startup. Use the Conversation Center to send a translation request to the TranslationAgent by setting the performative to *request* and the content to some word to translate. Observe the TranslationAgent's output on the console when it receives the request.

3.2. Exercise B2 - Passive Plans

In constrast to the last exercise we will now use a passive plan to react on translation requests. To show the difference between the two forms of plans we now modify the service plan slightly to become a passive plan. Create the files EnglishGermanTranslationPlanB2.java and TranslationB2.agent.xml by copying the files from exercise B1.

Modify the copied file TranslationPlanB2.java.

- Replace all occurrences of "B1" in the Plan with "B2"
- In contrast to the initial plan, the passive plan's body method is only invoked, when an event matches the plan's trigger. So use the method `getInitialEvent()` to retrieve the event that caused the execution. Because we know that only certain messages activate the plan the event can directly be cast to type `jadex.runtime.IMessageEvent` and the content can be retrieved. The infinite loop in the body should be discarded, because for each event a new plan instance is created, which only handles a single message.

Modify the copied file TranslationB2.agent.xml.

- Replace all occurrences of "B1" in the ADF file with "B2"

- Modify the plan declaration in the ADF by removing the configurations section. Additionally a passive plan needs a trigger, that specifies under what circumstances a new plan instance is created. Therefore remove the waitqueue statement and add a new statement for the plan trigger:

```
<trigger>
        <messageevent ref="request_translation"/>
</trigger>
```

Start and test the agent. Start the agent as explained in the preceding exercise. Observe that a new instance of the translation plan is created everytime an appropriate event arrives. The passive plan is instantiated and each instance processes a different message event. Many different plan instances may remain active while processing their triggers.

3.3. Exercise B3 - Plan Parameters

In this exercise we will use plan parameters to supply the plan with arguments. Plan parameters can directly be accessed from within the plan body via the `getParameter("paramname")` and `getParameterSet("paramsetname")` methods. Generally parameters can have the directions *in* (default), *out* and *inout* describing parameters that are used for supplying values or resp. gathering return values from the plan. Plan parameters can be supplied with fixed values via the `<value>` or `<values>` tags. More interestingly parameter values can be mapped from and to the triggers by using parameter mappings. If a plan could be activated by more than one trigger (e.g. two different messages, or a message and a goal, etc.) multiple goal mappings (one for each trigger type) have to be used to unify the plans view on its arguments.

Create the files EnglishGermanTranslationPlanB3.java and TranslationB3.agent.xml by copying the files from exercise B2. Apply the same replacements B2->B3 as in the previous exercise.

Modify the EnglishGermanTranslationPlanB3.java.

- Instead of using the `getInitialEvent()` method to retrieve the English word, we use the the statement:

```
String eword = (String)getParameter("eword").getValue();
```

Modify the copied file TranslationB3.agent.xml to include the new plan parameter.

- Add the new plan parameter with a message event mapping to the ADF:

```
<plan name="egtrans">
    <parameter name="eword" class="String">
        <messageeventmapping ref="request_translation.content"/>
    </parameter>
    <body>new EnglishGermanTranslationPlanB3()</body>
    <trigger>
        <messageevent ref="request_translation"/>
    </trigger>
</plan>
```

Start and test the agent. Test and verify that the agent behavior is the same as in the last exercise.

3.4. Exercise B4 - Plan Selection

In this exercise we will use plan priorities to establish a plan selection order. Create the files EnglishGermanTranslationPlanB4.java and TranslationB4.agent.xml by copying the files from exercise B2. Apply the same replacements B2->B4 as in the previous exercise.

Create a new plan file named SearchTranslationOnlineB4.java.

- This plan should be used when the agent cannot find the word in its (currently very small) dictionary. In this case the online search plan will try to connect to a web dictionary and report the found translations. The address of a simple English-German dictionary is http://wolfram.schneider.org/dict/dict.cgi (you may use any other dictionary for this purpose if you are not afraid of parsing the result HTML page). To issue a query against this online database you need to create a URL and read the data from there as outlined below:

```
URL dict = new URL("http://wolfram.schneider.org/dict/dict.cgi?query="+eword);
BufferedReader in = new BufferedReader(new InputStreamReader(dict.openStream()));
String inline;
while((inline = in.readLine())!=null) {
    if(inline.indexOf("<td>")!=-1 && inline.indexOf(eword)!=-1) {
        try {
            int start = inline.indexOf("<td>")+4;
            int end = inline.indexOf("</td", start);
            String worda = inline.substring(start, end);
            start = inline.indexOf("<td", start);
            start = inline.indexOf(">", start);
            end = inline.indexOf("</td", start);
            String wordb = inline.substring(start, end==-1? inline.length()-1: end);
            wordb = wordb.replaceAll("<b>", "");
            wordb = wordb.replaceAll("</b>", "");
            System.out.println(worda+" - "+wordb);
        }
        catch(Exception e) {
            System.out.println(inline);
        }
    }
}
in.close();
```

Modify the EnglishGermanTranslationPlanB4 having a static dictionary.

- Make the variable for the dictionary static and initialize it in a static block instead of in the constructor:

```
static {
    wordtable = new HashMap();
    wordtable.put("coffee", "Kaffee");
    wordtable.put("milk", "Milch");
    wordtable.put("cow", "Kuh");
    wordtable.put("cat", "Katze");
    wordtable.put("dog", "Hund");
}
```

- Provide a public static method for testing if a word is contained in the dictionary:

```
public static boolean containsWord(String name) {
        return wordtable.containsKey(name);
}
```

Modify the copied file TranslationB4.agent.xml to include the new plan.

- Add the new online search plan to the plan declarations using a low priority:

```
<plan name="searchonline" priority="-1">
    <body>new SearchTranslationOnlineB4()</body>
    <trigger>
        <messageevent ref="request_translation"/>
    </trigger>
</plan>
```

- Modify the applicability of the translation plan by introducing a precondition

```
<plan name="egtrans">
    <body>new EnglishGermanTranslationPlanB4()</body>
    <trigger>
        <messageevent ref="request_translation"/>
    </trigger>
    <precondition>
        EnglishGermanTranslationPlanB4.containsWord((String)$event.getContent())
    </precondition>
</plan>
```

Start and test the agent. When the agent receives translation request it searches applicable plans to handle this request. If the word is contained in the dictionary both plans are applicable and the one with the higher priority is chosen (in this case it is the egtrans plan because the standard priority is 0). When the word is not contained in the dictionary only the searchonline plan is applicable and will be used.

3.5. Exercise B5 - BDI Debugger

Using the Jadex introspector tool agent to control the execution of an agent.

- Prepare the agent debugging by setting the debugging flag in a new properties section (at the end of the file) of the ADF to true, e.g. in the B4 ADF. Therefore the agent will be started in step mode and will only process events when the execution is manually requested in the introspector tool.

```
<properties>
    <property name="debugging">true</property>
</properties>
```

  Note that you can also freeze the execution of the translation agent by setting execution mode to "step" in the tool. Using the debug flag is preferable when the agent directly starts with executing some actions and you want to observe it right from the start.

- Start the translation agent of the last exercise from the Control Center.
- Switch to the introspector perspective in the Control Center open an introspector for the translation agent and activate debugging by choosing the debugger tab and clicking the start button.
- Use the Conversation Center to send some translation requests to the translation agent (as in B4).
- Press the "step" button several times in the dispatcher and observe how an action from the agenda is executed. If the mode is "cycle" instead of step process event actions are executed in one step. Otherwise actions for all intermediate steps - searching applicable plans, selecting candidates from this list and scheduling the candidates for execution - are generated.

The Jadex debugging perspective is conceived to support you in the debugging of agents and helps you to understand what happens inside an agent, e.g. you could use it for the agent from excercise B1 too to grasp the differences between B1 and B2.

3.6. Exercise B6 - Log-Outputs

In this exercise we will use log-outputs instead of printing console outputs. Create the files EnglishGermanTranslationPlanB6.java and TranslationB6.agent.xml by copying the files from exercise B2.

Modify the copied file TranslationPlanB6.java.

- Replace all occurrences of System.out.println(..) to getLogger().info(..).

Modify the copied file TranslationB3.agent.xml.

- Add an imports section and the import statement for the java.logging classes to the imports section.

```
<imports>
    <import>java.util.logging.*</import>
</imports>
```

- Introduce a properties section at the bottom of the ADF to specify the logging behavior. Insert the following code:

```
<properties>
    <property name="logging.level">Level.INFO</property>
    <property name="logging.useParentHandlers">true</property>
</properties>
```

  These properties can be used to control the agent logging. The log-level decides what kind of log-outputs shall be considered for logging, according to the java.util.logging level hierarchy. Increasing the level value, e.g. to warning means, that only log-outputs at this or a higher level are conisdered by the logger. The useParentHandlers property can be used to turn on or off the standard console logging handler (per default it is set to true).

Start and test the agent. Start the translation agent. Send a translation request to the translation agent and watch the console and logger output. To turn off the console output simply set the property useParentHandlers in the ADF to false.

Chapter 4. Using Beliefs

An agent's beliefbase represents its knowledge about the world. The beliefbase is in some way similar to a simple data-storage, that allows the clean communication between different plans by the means of shared beliefs. Contrary to most PRS-style BDI systems, Jadex allows to store arbitrary Java objects as beliefs in its beliefbase. In Jadex between two kinds of beliefs is distinguished. On the one hand there are beliefs that allow the user to store exactly one fact and on the other hand belief sets are supported that allow to store a set of facts. The use of beliefs and belief sets as primary storage capacities for plans is strongly encouraged, because from its usage the user benefits in several ways. If it is necessary to retrieve a cut out of the stored data this is supported by a declarative OQL-like query language. Furthermore, it is possible to monitor single beliefs with respect to their state and cause an event when a corresponding condition is satisfied. This allows to trigger some action when e.g. a fact of a belief set is added or a belief is modified. It is also possible to wait for some complex expression that relates to several beliefs to become fulfilled.

4.1. Exercise C1 - Beliefs

From this point the copying and renaming of files is not explicitly stated anymore. Furthermore, from now on we use a syntax in the request format that looks like this:

```
<action> <language(s)> <content>
```

To translate a word we have to send a request in the form:

`translate english_german <word>`
To add a new word pair to the database we have to send a request in the format:

`add english_german <eword> <gword>`
In this first exercise we will use the beliefbase for letting more than one plan having access to the word table by using a belief for storing the word table.

Modify the existing plan to support the request format and introduce a new plan for adding word pairs.

- Create a new AddGermanWordPlanC1 as passive plan, that handles add-new-wordpair requests. In its body method, the plan should check whether the format is correct (using a `java.util.StringTokenizer`). If it is ok, it should retrieve the hashtable containing the word pairs via:

```
Map words = (Map)getBeliefbase().getBelief("egwords").getFact();
```

  Assuming that the belief for storing the wordpairs is named "egwords". Now the plan has to check if the English word is already contained in the map (using `words.containsKey(eword)`) and if it is not contained, it should be added (using `words.put(eword, gword)`).

- Modify the EnglishGermanTranslationPlanC1 so, that it uses the word table stored as single belief in the beliefbase. Additionally the plan has to check the newly introduced request format by using a `java.util.StringTokenizer`.

- Add a static getDictionary() method to the EnglishGermanTranslationPlanC1. This method should return a hashmap with some wordpairs contained in it. Besides the static method you also need to declare a static variable for storing the dictionary:

```
protected static Map dictionary;
public static Map getDictionary(){
```

```
    if(dictionary==null)
    {
        dictionary = new HashMap();
        dictionary.put("milk", "Milch");
        dictionary.put("cow", "Kuh");
        dictionary.put("cat", "Katze");
        dictionary.put("dog", "Hund");
    }
    return dictionary;
}
```

Update the ADF to incorporate the new plan and the new belief.

- The updated version of the translation agent ADF is outlined in Figure 4.1, “C1 XML ADF”. Note that the agent now has two plans named "addword" for adding a word pair to the database and "egtrans" for translating from English to German. The belief declaration is enclosed by a beliefs tag that denotes that an arbitrary number of belief declarations may follow. Of course, plans can create and delete beliefs (and belief sets) at runtime (see [Jadex User Guide] for more details). The ADF only defines the initially created beliefs, optionally with default fact(s). The belief for storing the wordtable is named "egwords" and typed through the class attribute to `java.util.Map`. The tag of this element is set to `belief` (in contrast to `beliefset`) denoting that only one fact can be stored. Further it is necessary to clarify which kinds of events trigger the plans. Therefore, the events section (lines 41-58) is extended to include a new request_addword event type which also matches request messages. To be able to distinguish between both kinds of events they are refined to match only messages that start with a specific content string (cf. lines 46-48 / 54-56).

```
 1 <!--
 2     Using the beliefbase with a belief.
 3     The agent stores its dictionary in a single-valued
 4     belief that can be accessed from a translation as well
 5     as from an add new word plan.
 6 -->
 7 <agent xmlns="http://jadex.sourceforge.net/jadex"
 8     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 9     xsi:schemaLocation="http://jadex.sourceforge.net/jadex
10                         http://jadex.sourceforge.net/jadex-0.96.xsd"
11     name="TranslationC1"
12     package="jadex.tutorial">
13
14     <imports>
15         <import>java.util.logging.*</import>
16         <import>java.util.*</import>
17         <import>jadex.adapter.fipa.*</import>
18     </imports>
19
20     <beliefs>
21         <belief name="egwords" class="Map">
22             <fact>EnglishGermanTranslationPlanC1.getDictionary()</fact>
23         </belief>
24     </beliefs>
25
26     <plans>
27         <plan name="addword">
28             <body>new EnglishGermanAddWordPlanC1()</body>
29             <trigger>
30                 <messageevent ref="request_addword"/>
31             </trigger>
32         </plan>
33         <plan name="egtrans">
34             <body>new EnglishGermanTranslationPlanC1()</body>
35             <trigger>
36                 <messageevent ref="request_translation"/>
37             </trigger>
38         </plan>
```

```
39     </plans>
40
41     <events>
42         <messageevent name="request_addword" direction="receive" type="fipa">
43             <parameter name="performative" class="String" direction="fixed">
44                 <value>SFipa.REQUEST</value>
45             </parameter>
46             <parameter name="content-start" class="String" direction="fixed">
47                 <value>"add english_german"</value>
48             </parameter>
49         </messageevent>
50         <messageevent name="request_translation" direction="receive" type="fipa">
51             <parameter name="performative" class="String" direction="fixed">
52                 <value>SFipa.REQUEST</value>
53             </parameter>
54             <parameter name="content-start" class="String" direction="fixed">
55                 <value>"translate english_german"</value>
56             </parameter>
57         </messageevent>
58     </events>
59 </agent>
```

Figure 4.1. C1 XML ADF

Start and test the agent. Send several add-word and translation requests to the agent and observe, if it behaves well. In this example the belief is already created when the agent is initialized.

4.2. Exercise C2 - Beliefsets

Using a belief set for storing the word-pairs and employing beliefbase queries to look-up a word in the word table belief set. In this example each word pair is saved in a data structure called jadex.util.Tuple which is a list of entities similar to an object array. In contrast to an object array two tuples are considered to be equal when they contain the same objects. *Of course, in belief sets arbitrary Java objects can be stored, not just Tuples.*

Modify the plans.

- Modify the EnglishGermanTranslationPlanC2 so, that it uses a query to search the requested word in the belief set. Therefore use an expression defined in the ADF: `this.queryword = getExpression("query_egword");` (Assuming that the `jadex.runtime.IExpression` queryword is declared as instance variable in the plan) To apply the query insert the following code at the corresponding place inside the plan's body method: `String gword = (String)queryword.execute("$eword", eword);`
- Modify the EnglishGermanAddWordPlanC2 so, that it also uses the same query to find out, if a word pair is already contained in the belief set. Apply the query before inserting a new word pair. When the word pair is already contained log some warning message. To add a new fact to an existing belief set you can use the method:

```
getBeliefbase().getBeliefSet("egwords").addFact(new jadex.util.Tuple(eword, gword));
```

Modify the ADF.

- For checking if a word pair is contained in the wordtable and for retrieving a wordpair from the wordtable a query expression will be used. Insert the following code into the ADF below the events section:

```
<expressions>
    <expression name="query_egword">
        select one $wordpair.get(1)
        from Tuple $wordpair in $beliefbase.egwords
        where $wordpair.get(0).equals($eword)
        <parameter name="$eword" class="String"/>
    </expression>
</expressions>
```

We don't cover the details of the query construction in this tutorial. If you are interested in understanding the details of the Jadex OQL query language, please consult the [Jadex User Guide].

- Modify the ADF by defining a belief set for the wordtable. Therefore change the tag type from "belief" to "belief set" and the class from "Map" to "Tuple". Note that Tuple is a helper class that is located in jadex.util and has to be added to the imports section if you don't specify the fully-qualified classname. Remove the old Map fact declaration and put in four new facts each surrounded by the fact tag. Put in the same values as before (using `new Tuple("milk", "Milch"))` etc. for each fact.

Start and test the agent. Send several add-word and translation requests to the agent and observe, if it behaves well. Verify that it behaves exactly like the agent we built in exercise C1. This exercise does not functionally modify our agent.

4.3. Exercise C3 - Belief Conditions

In this exercise we will use a condition for triggering a passive plan that congratulates every 10th user.

Create and modify plans.

- Create a new passive ThankYouPlanC3 that prints out a congratulation message and the actual number of processed requests. The number of processed requests will be stored in a belief called "transcnt" in the ADF. Retrieve the actual request number by getting the fact from the beliefbase with:

```
int cnt = ((Integer)getBeliefbase().getBelief("transcnt").getFact()).intValue();
```

- Modify the EnglishGermanTranslationPlanC3 to count the translation requests:

```
int cnt = ((Integer)getBeliefbase().getBelief("transcnt").getFact()).intValue();
getBeliefbase().getBelief("transcnt").setFact(new Integer(cnt+1));
```

Modify the ADF.

- Modify the ADF by defining the new ThankYouPlanC3 as passive plan (with the name thankyou in the ADF) in the plans section. Instead of defining a triggering event for this passive plan we define a condition that activates the new ThankYouPlanC3. A condition has the purpose the monitor some state of affair of the agent. In this case we want to monitor the belief "transcnt" and get notified whenever 10 translations have been requested. Insert the following in the plan's trigger:

```
<condition>$beliefbase.transcnt>0 &amp;&amp; $beliefbase.transcnt%10==0</condition>
```

This condition consists of two parts: This first transcnt>0 makes sure that at least one translation has been done and the second part checks if transcnt modulo 10 has no rest indicating that 10*x translations have been requested. The two parts are connected via a logical AND (&&), that has to be written a little bit awkwardly with the xml entities &&.

- Define and initialize the new belief in the ADF by introducing the following lines in the beliefs section:

```
<belief name="transcnt" class="int">
    <fact>0</fact>
</belief>
```

Start and test the agent. Send some translation requests and observe if every 10th time the congratulation plan is invoked and prints out its message.

4.4. Exercise C4 - Agent Arguments

In this exercise we will use agent arguments for the custom initialization of an agent instance.

- Use the translation agent C3 as starting point and specify an agent argument in the ADF. Arguments are beliefs for which a value can be supplied from outside during the agent start-up. For declaring a belief being an agent argument simply mark it as exported by using the corresponding belief attribute exported="true". In this case we want the belief "transcnt" being the argument. Note that only beliefs not belief sets can be used as arguments.
- Use the Starter to create instances of the new agent model. The Starter automatically displays textfields for all agent arguments and also shows the default model value (if any) that will be used when the user does not supply a value. Try entering different values into the textfield, what happens if you enter e.g. a string instead of the integer value that is needed here?
- Start the agent with different argument values. Verfify, that the agent immediately invokes the congratulation plan if the initial number of translation requests is e.g. 10.

4.5. Exercise C5 - BDI Viewer

In this exercise we will use the Jadex BDI introspector tool agent to view the beliefs of the agent.

- Start the translation agent from the last exercise. Before sending requests to the translation agent start the Jadex BDI introspector agent by selecting the translation agent in the AgentManager and activating the "Show Jadex Introspector" via the start button.
- Use the Conversation Center to send translation or add-word requests to the translation agent.
- Observe the belief change of the translation count, whenever a translation request is processed.
- Observe the changes of the word pair belief set, whenever an add-word request is processed.
- Use the example from C1 to see the difference in the representation of the word table as belief and belief set.

Chapter 5. Using Capabilities

Different agents often need to use the same or similar functionalities that incorporate more than just plan behavior. Often private or shared beliefs and goals are part of a common functionality of one agent. These units of functionality are comparable to the module concept in the object oriented paradigm, but exhibit very different properties because of the use of mentalistic notions. For this reasons the capability concept was originally introduced [Busetta et al. 2000] and enhanced in [Braubach et al. 2005b] that allows for packaging a subset of beliefs plans and goals into an agent module and reuse this module wherever needed. The capability structure of an agent forms a tree. A superordinated (parent) capability may contain an arbitrary number of subcapabilities. All elements of a capability have per default private visibility and need to be explicitly made available for usage in a connected capability. For this purpose elements can be defined as abstract or exported enabling access from another capability.

5.1. Preparation

We use the functionality of the C2 Agent and build up a capability of its plans and beliefs. Therefore, it is necessary to copy and rename all files from C2 to D1. We slightly modify these plans to make the translation agent answer to a request with a reply message. Hence the following has to be done in both plans:

- Declare two variables at the beginning of the plans:

```
String reply;  // The message event type of the reply.
String content; // The content of the reply message event.
```

- Set both variables with respect to the success of the translation. In the success case set (assuming that gword and eword are variables for the English and German word respectively):

```
reply = "inform";
content = gword;
```

  And in the failure case:

```
reply = "failure";
content = "Sorry, word could not be translated: "+eword;
```

- Send an answer to the caller at the end of the event processing:

```
sendMessage(((IMessageEvent)getInitialEvent()).createReply(reply, cont));
```

- Add the new message event types "inform" and "failure" to the ADF:

```
<events>
    ...
    <messageevent name="inform" direction="send" type="fipa">
        <parameter name="performative" class="String" direction="fixed">
            <value>SFipa.INFORM</value>
        </parameter>
    </messageevent>

    <messageevent name="failure" direction="send" type="fipa">
        <parameter name="performative" class="String" direction="fixed">
            <value>SFipa.FAILURE</value>
        </parameter>
    </messageevent>
</events>
```

- Test the agent and verify that it answers to the request messages by sending an answer message (for correct as well as for incorrect requests).

5.2. Exercise D1 - Creating a Capability

In this exercise we will create a translation capability.

Create a new Capability ADF.

- Create a new file TranslationD1.capability.xml with the skeleton code from Figure 5.1, “D1 XML ADF”. Now copy the definition of imports, plans, beliefs, events (including the newly defined ones from above) and expressions (in this case there are no goals) from TranslationC2.agent.xml into this file.

```
 1 <!--
 2     Translation capability.
 3 -->
 4 <capability xmlns="http://jadex.sourceforge.net/jadex"
 5     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 6     xsi:schemaLocation="http://jadex.sourceforge.net/jadex
 7                         http://jadex.sourceforge.net/jadex-0.96.xsd"
 8     name="TranslationD1"
 9     package="jadex.tutorial">
10     ...
11 </capability>
```

Figure 5.1. D1 XML ADF

- Modify the agent ADF (TranslationD1.agent.xml) by removing all plan and belief definitions. Instead insert a new section for using the new capability.

```
<capabilities>
    <capability name="transcap" file="TranslationD1"/>
</capabilities>
```

  Note that here the type name is emloyed, but absolute and relative paths to (the model name of) the XML file can also be used.

Start and test the agent. Load the agent model in the RMA and start the agent. Test the agent with add word and translate requests. It should behave exactly like the Agent from C2. Use the introspector agent (using BDI-view) to view the new internal structure of the agent.

5.3. Exercise D2 - Exported Beliefs

In this exercise we will extend the translation agent by making it capable to find synonyms for English words. Therefore we extend the agent from D1 with a new find synonyms plan which will directly be contained in the agent description. Because the plan needs to access the dictionary from the translation capability, the egwords belief will be made usable from external.

Create a new plan.

- Create the file FindEnglishSynonymsPlanD2.java as a passive plan which reacts on messages with performative type request and starts with "find_synonyms english". Therefore, you need to introduce the new mes-

sage event type "find_synonyms" that fits for request messages that start with "find_synonyms english".

- Create one query (called query_translate here) in the constructor for translating an English word (the query expression can be copied from the EnglishGermanTranslationPlanD2). Create another query (called query_find here) with the purpose to find all English words that match exactly a German word and are unequal to the given English word.

```
String find = "select $wordpair.get(0) " +
    "from Tuple $wordpair in $beliefbase.egwords " +
    "where $wordpair.get(1).equals($gword) && !$wordpair.get(0).equals($eword)";
this.queryfind = createExpression(find, new String[]{"$gword", "$eword"},
    new Class[]{String.class, String.class});
```

- In the body method, search for synonyms when the message format is correct, what means that the request has exactly three tokens. Use a `StringTokenizer` to parse the request and apply the translation query on the the given English word. When a translation was found, use the result to apply the query find for searching for synonyms. Create a reply and send back the found synonyms as an inform message in the success case and a failure message with a failure reason in the error case. The following code snippet outlines how the second query can be realized (eword is the English word for which synonyms are searched, gword is the German translation of the given English word):

```
query_find.setParameter("$gword", gword);
query_find.setParameter("$eword", eword);
List syns = (List)query_find.execute();
```

Create a new Capability ADF.

- Create a new file TranslationD2.capability.xml by copying the capability from exercise D1.

- Modify the belief set declaration of "egwords" by setting the belief set `type="exported"`. Add some facts to the belief "egtrans" to have some synonyms present.

```
<beliefset name="egwords" class="Tuple" exported="true">
    <fact>new Tuple("milk", "Milch")</fact>
    <fact>new Tuple("cow", "Kuh")</fact>
    <fact>new Tuple("cat", "Katze")</fact>
    <fact>new Tuple("dog", "Hund")</fact>
    <fact>new Tuple("puppy", "Hund")</fact>
    <fact>new Tuple("hound", "Hund")</fact>
    <fact>new Tuple("jack", "Katze")</fact>
    <fact>new Tuple("crummie", "Kuh")</fact>
</beliefset>
```

- Also use the exported attribute to make the "inform" and "failure" messages accessible, as we want to use these in the new synomyms plan.

Create a new TranslationD2 Agent ADF.

- Create a new file TranslationD2.agent.xml by copying the file from D1. Extend this definition by adding the new plan to the plans section and adding a referenced belief, that relates to the egwords belief from the capability. Note that the name of the referenced belief can be chosen arbitrarily (in this case we name it egwords, too). Additionally change the capability reference to the newly created TranslationCapabilityD2 and add the new message event type request_findsynonyms.

```
<beliefs>
    <beliefsetref name="egwords">
        <concrete ref="transcap.egwords" />
```

```
    </beliefsetref>
</beliefs>

<plans>
    <plan name="find_synonyms">
        <body>new FindEnglishSynonymsPlanD2()</body>
        <trigger>
            <messageevent ref="request_findsynonyms"/>
        </trigger>
    </plan>
</plans>

<events>
    <messageevent name="request_findsynonyms" direction="receive" type="fipa">
        <parameter name="performative" class="String" direction="fixed">
            <value>SFipa.REQUEST</value>
        </parameter>
        <parameter name="content-start" class="String" direction="fixed">
            <value>"find_synonyms english"</value>
        </parameter>
    </messageevent>

    <messageeventref name="inform">
        <concrete ref="transcap.inform"/>
    </messageeventref>

    <messageeventref name="failure">
        <concrete ref="transcap.failure"/>
    </messageeventref>
</events>
```

Start and test the agent. Start the agent and send it some find synonyms requests, e.g. "find_synonyms english dog". When your agent works ok, you should be notified that the synonyms for dog are hound and puppy. Use the bdi viewer (from the introspector) to understand what the belief set reference means.

Chapter 6. Using Goals

Goal-oriented programming is one of the key concepts in the agent-oriented paradigm. It denotes the fact that an agent commits itself to a certain objective and maybe tries all the possibilities to achieve its goal. A good example for a goal that ultimately has to be achieved is the safe landing of an aircraft. The agent will try all its plans until this goal has succeeded, otherwise it will not have the opportunity to reach any other goal when the aircraft crashes. When talking about goals one can consider different kind of goals. What we discussed above is called an *achieve goal* , because the agent wants to achieve a certain state of affairs. Similar to an achieve goal is the *query goal* which aims at information retrieval. To find the requested information plans are only executed when necessary. E.g. a cleaner agent could use a query goal to find out where the nearest wastebin is. Another kind is represented through a maintain goal, that has to keep the properties (its maintain condition) satisfied all the time. When the condition is not satisfied any longer, plans are invoked to re-establish a normal state. An example for a maintain goal is to keep the temperature of a nuclear reactor below some specified limit. When this limit is exceeded, the agent has to act and normalize the state. The fourth kind of goal is the perform goal, which is directly related to some kind of action one wants the agent to perform. An example for a perform goal is an agent that has to patrol at some kind of frontier.

6.1. Exercise E1 - Subgoals

In this exercise we will use a subgoal for translating words. Extend the translation agent C2 to have a second translation plan for translations from English to French. Introduce a ProcessTranslationRequestPlanE1 that receives all incoming translation requests and uses a subtask triggered by an achieve goal to perform the translation.

Remove, create and modify plans.

- Remove the EnglishGermanAddWordPlan to keep the agent simple.
- Create a new initial ProcessTranslationRequestPlanE1 that reacts on all incoming messages with performative type request and creates subgoals for all (correctly formatted) requests. Because we are using a service plan implement the body method with an infinite loop and start waiting for a message to process. Assuming that the plan has extracted the action (translate), the language direction (english_german or english_french) and the word(s) from an incoming message the following code can be used to create, dispatch and wait for a subgoal:

```
IGoal goal = createGoal("translate");
goal.getParameter("direction").setValue(dir);
goal.getParameter("word").setValue(word);
try {
    dispatchSubgoalAndWait(goal);
    getLogger().info("Translated from "+goal.getName()+" "+
        word+" - "+goal.getParameter("result").getValue());
}
catch(GoalFailureException e) {
    getLogger().info("Word is not in database: "+word);
};
```

  After the goal returns successfully, read the result from the goal and log some translation message.

- Modify the EnglishGermanTranslationPlanE1 so that it can handle a translation goal with direction="english_german". Therefore, the body method has to be adapted so that it extracts the word from the plan parameter mapping (using `getParameter("word").getValue()`). After having performed the query on the wordtable, set the result using `getParameter("result").setValue(gword)`. When no translation could be retrieved, the plan has failed and this should be indicated calling the `fail()` method (which throws a

plan failure exception).

- Create a new EnglishFrenchTranslationPlanE1 as a copy of the EnglishGermanTranslationPlanE1 and make sure to work on a new wordtable belief efwords. Modify the query_word and the body method accordingly.

Modify the ADF.

- Add the ProcessTranslationRequestPlanE1 to the ADF as initial plan with a waitqueue for translation requests. Add an configurations section and declare a configuration with an initial plan for the ProcessTranslationRequestPlanE1.

- Adapt the plan head declarations of both plans to include plan parameters and the new triggers. The plan parameters are directly mapped to the corresponding goal parameters so that the input as well as the result are automatically transferred from resp. to the goal. In addition, both translation plans should handle exactly suitable translation goals. In the following the modified plan head for the EnglishGermanTranslationPlanE1 is depicted:

```
<plan name="egtrans">
    <parameter name="word" class="String">
        <goalmapping ref="translate.word"/>
    </parameter>
    <parameter name="result" class="String">
        <goalmapping ref="translate.result"/>
    </parameter>
    <body>new EnglishGermanTranslationPlanE1()</body>
    <trigger>
        <goal ref="translate">
            <match>"english_german".equals(
                    $goal.getParameter("direction").getValue())</match>
        </goal>
    </trigger>
</plan>
```

- Introduce a new goals section and declare the achieve goal for translations:

```
<goals>
    <achievegoal name="translate">
        <parameter name="direction" class="String"/>
        <parameter name="word" class="String"/>
        <parameter name="result" class="String" direction="out"/>
    </achievegoal>
</goals>
```

- Modify the ADF by adjusting the plan declarations to include the new EnglishFrenchTranslationPlanE1 and exclude the add word plan. Additionally a new belief efwords has to be declared in the beliefs section:

```
<beliefset name="efwords" class="Tuple">
    <fact>new Tuple("milk", "lait")</fact>
    <fact>new Tuple("cow", "vache")</fact>
    <fact>new Tuple("cat", "chat")</fact>
    <fact>new Tuple("dog", "chien")</fact>
</beliefset>
```

- Introduce a second query for the new belief efword in the expressions section of the ADF:

```
<expression name="query_efword">
    select one $wordpair.get(1)
    from Tuple $wordpair in $beliefbase.efwords
    where $wordpair.get(0).equals($eword)
    <parameter name="$eword" class="String" />
```

```
</expression>
```

Start and test the agent. Start the agent and supply it with some translation requests. Observe which plans are activated in what sequence and how the goal processing is done. Change the translation direction in the requests and check if the right plan is invoked.

6.2. Exercise E2 - Retrying a Goal

Using the BDI-retry mechanism for trying out different plans for one goal. This can be usefull, for example if there are several plans for one specific goal, but all plans work under different circumstances. With the retry mechanism, all plans will be tried until one plan let the goal succeed or any plan has been tried.

Modify the following.

- Modify the trigger of both translating plans so, that they react on every translation goal by removing the lines:

```
<match>"english_german/english_french".equals(
$goal.getParameter("direction").getValue())</match>
```

  Having done this causes the translation plans to react on every translation goal, even when they can't handle the translation direction or language.

- Introduce a new plan parameter for the translation direction and supply it with a corresponding goal mapping:

```
<parameter name="direction" class="String">
    <goalmapping ref="translate.direction"/>
</parameter>
```

- Modify the translation plans so that they check the translation direction in the body method before translating. When the direction cannot be handled, they should indicate that they failed to achieve the goal by calling the `fail()` method. Addtionally the plans should log or print some warning message, when they fail to process a goal:

```
if("english_french".equals(getParameter("direction").getValue()))
    // print out some message and fail() if this is not the english-french plan
```

- You need not explicitly set the BDI-retry flag of the goal, because in the standard configuration for goals all BDI-mechanisms (retry, exclude and meta-level reasoning) are enabled. This means, that a failed goal will be retried by different plan candidates until it succeeds or all possible candidates have failed to handle the goal and it is finally failed.

Start and test the agent. Provide the agent with some translation work and watch out how the goal processing is done this time. Observe by changing the translation direction of the request how different plans are scheduled to handle a goal.

6.3. Exercise E3 - Maintain Goals

Using a maintain goal to keep the number of wordtable entries below a specified maximum value. For this exercise we will use the TranslationAgentD2 as starting point.

Create a new file RemoveWordPlanE3.java.

- This plan has the purpose to delete an entry from the wordtable.
- Create a constructor with a String parameter. This parameter will tell the plan in which belief set an entry has to be deleted. Save the parameter in an object variable called "beliefsetname".
- In the body method use the following code to delete one entry from the set:

```
Object[] facts = getBeliefbase().getBeliefSet(beliefsetname).getFacts();
getBeliefbase().getBeliefSet(beliefsetname).removeFact(facts[0]);
```

Create the TranslationE3.agent.xml as copy from the TranslationD2.agent.xml.

- Remove the FindSynonymsPlanD2 from the plans section and remove the event section completely.
- Add the RemoveWordPlanE3 to the plans section:

```
<plan name="remegword">
    <body>new RemoveWordPlanE3("egwords")</body>
    <trigger>
        <goal ref="keepstorage"/>
    </trigger>
    <precondition>$beliefbase.egwords.length &gt; 0</precondition>
</plan>
```

- Add the following beliefs to the belief section:

```
<beliefsetref name="egwords">
    <concrete ref="transcap.egwords" />
</beliefsetref>
<belief name="maxstorage" class="int">
    <fact>8</fact>
</belief>
```

- Add a new maitain goal declaration to the goals section:

```
<maintaingoal name="keepstorage" exclude="when_failed">
    <maintaincondition>
        $beliefbase.egwords.length &lt;= $beliefbase.maxstorage
    </maintaincondition>
</maintaingoal>
```

- Create a configurations section with one configuration that creates a maintain goal instance on startup.

```
<configurations>
    <configuration name="default">
        <goals>
            <initialgoal ref="keepstorage"/>
        </goals>
    </configuration>
</configurations>
```

Start and test the agent. Start the translation agent and open the debugging perspective in the Control Center. Now reduce the value of the maxstorage belief, e.g. by setting it to six. The maintain condition is violated and the goal should be activated. This leads to a subsequent removal of entries in the belief set, until the condition holds again. Additionally you can test the agent by sending "add word" requests, observing how a new entry is created. Furthermore the maintain condition could be violated, what should result in an entry removal.

Chapter 7. Using Events

Communication takes place at two different abstraction levels in Jadex. The so called intra-agent communication is necessary when two or more plans inside of an agent want to exchange information. They can utilize several techniques to achieve this. The encouraged possibility is to use beliefs (and conditions). Beliefs in Jadex are containers for normal Java objects, but they are a specially designed concept for agent modelling and therefore using beliefs has several advantages. One advantage is that they allow the usage of conditions to trigger events depending on belief states (e.g. creating a new goal when a new fact is added to a belief set). A further advantage is that using the beliefbase one is able to formulate queries and retrieve only entities that correspond to the query-expression. Another possibility of internal communication is to use explicit internal events. In contrast to goals, events are (per default) dispatched to all interested plans but do not support any BDI-mechanisms (e.g. retry). Therefore the originator of an internal event is usually not interested in the effect the internal event may produce but only wants to inform some interested parties about some (important) occurence.

On the other hand inter-agent communication describes the act of information exchange between two or more different agents. The inter-agent information exchange in Jadex is based on asynchronous message event passing. Each message event in Jadex has a dedicated `jadex.model.MessageType` which constrains the allowed parameters and the parameter types of the message event. Currently, only the FIPA message type is supported. It equips a message type with all possible FIPA parameters such as sender, receivers, performative, content, etc. Besides the underlying message type (which is normally not of very much importance for agent programmers) in the ADF user defined message event types are specified, such as the request_translation message event we already encountered in earlier exercises. Note that the message event types are only locally visible and each agent uses its own message event types for sending and receiving messages. Hence, when an agent receives a message it has to decide which local message event type will be used to represent this message. The details of this process will be outlined in one of the following exercises. In this tutorial we will only show how a basic communication between two agents is implemented, when one agent offers a service that the other one seeks. The supplier therefore has to register it's services by the Directory Facilitator (DF) and is further on available as service provider. Another agent seeks a service by asking the DF and receives the providers address which it subsequently uses for the direct communication with the provider.

7.1. Exercise F1 - Internal Events

In this exercise we will use internal events to broadcast information. We extend the simple translation agent from exercise C2 with a plan that shows the processed requests in a gui triggered by an internal event.

Create a new GUI class named TranslationGuiF1.java as an extension of a JFrame.

- This class has the purpose to show the already performed actions in a table. As member variable the table model is needed to be able to refresh the data in response to update notifications.

```
protected DefaultTableModel tadata;
```

- In the constructor the table and its model should be created and added to the frame:

```
String[] columns = new String[]{"Action", "Language", "Content", "Translation"};
this.tadata = new DefaultTableModel(columns, 0);
JTable tatable = new JTable(tadata);
JScrollPane sp = new JScrollPane(tatable);
this.getContentPane().add("Center", sp);
this.pack();
this.setVisible(true);
```

- Finally, for updating the gui a method is needed:

```
public void addRow(final String[] content){
    SwingUtilities.invokeLater(new Runnable(){
        public void run(){
            tadata.addRow(content);
        }
    });
}
```

Modify the plans.

- Create a new GUIPlanF1 plan that has the purpose to create the gui and update it accordingly. The plan should create the gui in its constructor. In its body method it should wait in an endless lopp for internal events of type gui_update:

```
IInternalEvent event = waitForInternalEvent("gui_update");
```

  Whenever such an event occurs the plan has to invoke the addRow() method of the gui whereby the update information is contained in a parameter named content within the internal event, accessible by:

```
event.getParameter("content").getValue();
```

  In addition the plan's aborted method can be used to close the gui automatically when the agent is terminated:

```
public void aborted(){
    SwingUtilities.invokeLater(new Runnable(){
        public void run(){
            gui.dispose();
        }
    });
}
```

- Modify the EnglishGermanTranslationPlanF1 so that it produces an internal event after translation processing:

```
IInternalEvent event = createInternalEvent("gui_update");
event.getParameter("content").setValue(new String[]{action, dir, eword, gword});
dispatchInternalEvent(event);
```

Modify the ADF .

- The addword plan and event declarations are not used and can be removed for clarity.
- Modify the ADF so, that it contains the declaration for the new gui plan without specifying a trigger:

```
<plan name="gui">
    <body>new GUIPlanF1()</body>
</plan>
```

- Introduce the declaration of the new gui_update event within the events section:

```
<internalevent name="gui_update">
    <parameter name="content" class="String[]"/>
</internalevent>
```

- Add an configurations section with one configuration that creates an initial gui plan:

```
<configurations>
    <configuration name="default">
        <plans>
            <initialplan ref="gui"/>
        </plans>
    </configuration>
</configurations>
```

Start and test the agent.

- Start the agent and send several translation requests to the agent. Observe if the gui displays all the translation requests.

7.2. Exercise F2 - Receiving Messages

This exercise will explain how the mapping of received messages to the agent's message events works. Whenever an agent receives a message it has to decide which message event will internally be used for representing the message. This mapping is very important because any agent behavior such as e.g. plan triggers may only depend on the interpreted message event type. In general the event mapping works automatically and an agent designer does not have to worry about the mappings. Nevertheless, there are situations in which more than one mapping from a received message to different message events are available (normally this is undesirable and should be avoided by using more specific message event declarations). In such situations the agent rates the alternatives by specificity that is simply estimated by the number of parameters used for the declaration and chooses the one with the highest specificity. If more than one alternative has the same specificity the first one is chosen, although this case indicates an implementation flaw and might lead to undesired behavior when the wrong mapping is chosen. In any case, the developer is informed with a logging message whenever more than one mapping was found by the agent.

As starting point for this exercise we take agent B2, which only has one passive translation plan which reacts on request messages. Other kinds of messages are simply ignored by the agent. To improve this situation and let the agent answer on all incoming messages we use the ready to use NotUnderstoodPlan from the Jadex plan library. Additionally, instead of the original B2 translation plan we take the enhanced one from section 5.1 which sends back inform/failure messages to the requesting agent.

Modify the copied file TranslationF2.agent.xml to include the new not-understood plan.

- Add the following to the imports section:

```
<import>
   jadex.planlib.*
   jadex.adapter.fipa.*
</import>
```

- Add the new not-understood plan to the plan declarations:

```
<plan name="notunderstood">
    <body>new NotUnderstoodPlan()</body>
    <trigger>
        <messageevent ref="any_message"/>
    </trigger>
</plan>
```

- Add the new any_message event which matches all kinds of messages and the not understood message that

will be sent by the NotUnderstoodPlan to the event declarations:

```
<messageevent name="any_message" direction="receive" type="fipa"/>
<messageevent name="not_understood" direction="send" type="fipa">
    <parameter name="performative" class="String" direction="fixed">
        <value>SFipa.NOT_UNDERSTOOD</value>
    </parameter>
</messageevent>
```

- Besides the any_message for receiving arbitrary kinds of messages and the not_understood message which will be send by the not understood plan, we also need message declarations for the other messages to be sent be our agent. Here we need inform and failure messages that will be used by the modified translation plan:

```
<messageevent name="inform" direction="send" type="fipa">
    <parameter name="performative" class="String" direction="fixed">
        <value>SFipa.INFORM</value>
    </parameter>
</messageevent>
<messageevent name="failure" direction="send" type="fipa">
    <parameter name="performative" class="String" direction="fixed">
        <value>SFipa.FAILURE</value>
    </parameter>
</messageevent>
```

Start and test the agent. The added plan provides the agent with the ability to react on arbitrary messages. When the agent receives a message with performative request, both message events match and the request_translation event is chosen due to its higher specificity. Other messages are directly mapped to the any_message event type and hence, the agent will repond with a not understood message. Send the agent different messages and observe if it invokes the right plans.

7.3. Exercise F3 - Service publication

We make the services of our translation agent publicly available by registering its service description at the Directory Facilitator (DF).

Use the translation agent D1 as starting point and extend its copied ADF by performing the following steps.

- Include the DF capability in the ADF to be used:

```
<capabilities>
    <capability name="dfcap" file="jadex.planlib.DF"/>
    <capability name="transcap" file="TranslationD1"/>
</capabilities>
```

- Create a reference for the df_keep_registered goal to make it locally available:

```
<goals>
    <maintaingoalref name="df_keep_registered">
        <concrete ref="dfcap.df_keep_registered"/>
    </maintaingoalref>
</goals>
```

- Create a configurations section with one configuration. In this configuration an initial goal for the df registration should be provided. The agent description that is used for the registration is provided as initial value of the "description" parameter of the df_keep_registered goal:

```
<configurations>
    <configuration name="default">
        <goals>
            <initialgoal ref="df_keep_registered">
                <parameter ref="description">
                    <value>
                        SFipa.createAgentDescription(null,
                        SFipa.createServiceDescription("service_translate",
                        "translate english_german", "University of Hamburg"))
                    </value>
                </parameter>
                <parameter ref="leasetime">
                    <value>20000</value>
                </parameter>
            </initialgoal>
        </goals>
    </configuration>
</configurations>
```

Start and test the agent.

- Start the agent and open the DF GUI. Observe if an entry for the agent exists. Use the DF GUI to deregister your agent (via popup-menu) and observe what happens after a while. Note that when you want to register the agent at a remote df, you only need to slightly modify your initial goal description by adding parameter values for the DF AgentIdentifier and address.

7.4. Exercise F4 - A Multi-Agent Scenario

As a little highlight we now extend our scenario from F3 to become a real multi-agent system. The scenario is depicted in Figure 7.1, “F4 multi-agent scenario”. The user wishes to translate a sentence and sends its requests via the message center to the user agent. The user agent searches for a translation service at the DF and subsequently sends for each word from the sentence, a translation request to the translation agent. The user agent collects the translated words and sends back the translated sentence to the message center, where it is visible for the user.

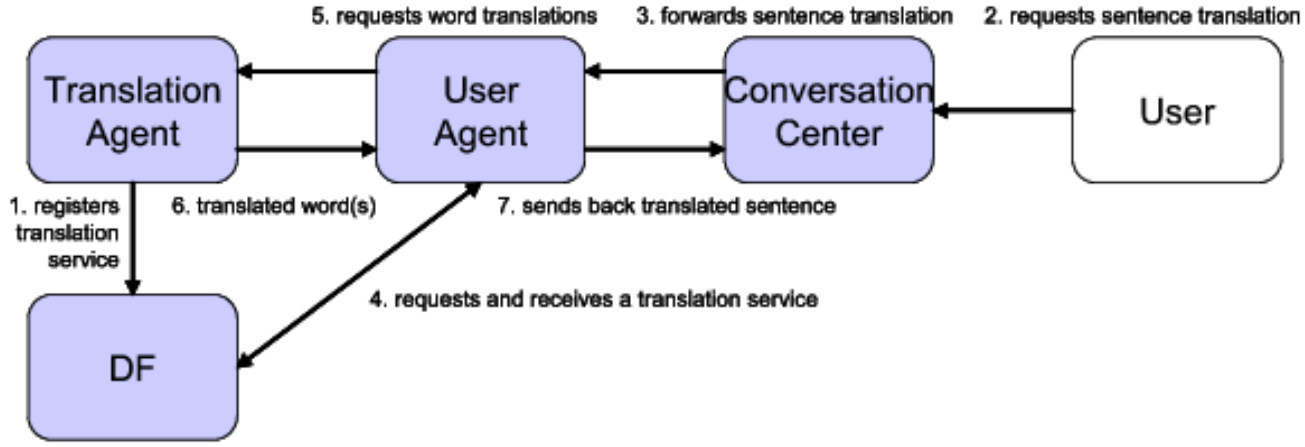

Figure 7.1. F4 multi-agent scenario

Create a new UserAgentF4.agent.xml.

- Create a new agent called UserAgent by creating an ADF and one plan called EnglishGermanTranslateSen-

tencePlanF4. In the ADF define the translate sentence plan with an appropriate waitqueue that handles message events of the new type request_translatesentence. The new request_translatesentence message event should be declared to match request messages that start with "translate_sentence english_german". Additionally incorporate the DF and Protocols capabilities in the capabilities section and create references for the rp_initiate (request protocol initiate) and df_search goals. This agent uses the df search goal to find a translation agent and the request goal to communicate in a similar standard way with the translation agent.

```
<agent xmlns="http://jadex.sourceforge.net/jadex"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jadex.sourceforge.net/jadex
                        http://jadex.sourceforge.net/jadex-0.96.xsd"
    name="UserF4"
    package="jadex.tutorial">

    <imports>
        <import>jadex.planlib.*</import>
        <import>jadex.adapter.fipa.*</import>
        <import>java.util.logging.*</import>
    </imports>

    <capabilities>
        <capability name="procap" file="jadex.planlib.Protocols"/>
        <capability name="dfcap" file="jadex.planlib.DF"/>
    </capabilities>

    <goals>
        <achievegoalref name="rp_initiate">
            <concrete ref="procap.rp_initiate"/>
        </achievegoalref>
        <achievegoalref name="df_search">
            <concrete ref="dfcap.df_search"/>
        </achievegoalref>
    </goals>

    <plans>
        <plan name="egtrans">
            <body>new EnglishGermanTranslateSentencePlanF4()</body>
            <waitqueue>
                 <messageevent ref="request_translatesentence"/>
            </waitqueue>
        </plan>
    </plans>

    <events>
        <messageevent name="request_translatesentence" direction="receive" type="fipa">
            <parameter name="performative" class="String" direction="fixed">
                <value>SFipa.REQUEST</value>
            </parameter>
            <parameter name="content-start" class="String" direction="fixed">
                <value>"translate_sentence english_german"</value>
            </parameter>
        </messageevent>
        <messageevent name="inform" direction="send" type="fipa">
            <parameter name="performative" class="String" direction="fixed">
                <value>SFipa.INFORM</value>
            </parameter>
        </messageevent>
        <messageevent name="failure" direction="send" type="fipa">
            <parameter name="performative" class="String" direction="fixed">
                <value>SFipa.FAILURE</value>
            </parameter>
        </messageevent>
    </events>

    <configurations>
        <configuration name="default">
            <plans>
                <initialplan ref="egtrans"/>
            </plans>
```

```
        </configuration>
    </configurations>
</agent>
```

- The body method of this plan should adhere to the following basic structure:

```
public void body()
{
    protected AgentIdentifier ta;
    ...

    while(true)
    {
        // Read the user request.
        IMessageEvent mevent = (IMessageEvent)
            waitForMessageEvent("request_translatesentence");

        // Save the words of the sentence.
        // Process the message event here.
        ...
        String[] words = ...;
        String[] twords = ...;
        ...

        // Search a translation agent.
        while(ta==null) // ta is the instance variable for the translation agent
        {
            // Create a service description to search for.
            // You can use the ServiceDescription from the ADF of exercise F3.
            // Use a df-search subgoal to search for a translation agent
            // Save the translation agent in the variable ta
            // If no translation agent could be found waitFor() some time and try again
        }

        // Translate the words.
        for(int i=0; i<words.length; i++)
        {
            IGoal tw = createGoal("rp_initiate");
            tw.getParameter("content").setValue("translate english_german "+words[i]);
            tw.getParameter("receiver").setValue(this.ta);
            try
            {
                dispatchSubgoalAndWait(tw);
                twords[i] = (String)tw.getParameter("result").getValue();
            }
            catch(GoalFailureException gfe)
            {
                twords[i] = "n/a";
            }
        }

        // Send the reply with the translation of the whole sentence
        // to the caller (the user agent)
        // ...
    }
}
```

Start and test the agent.

- Start a translation agent and a user agent. Now send a translate sentence request to to the user agent, which will answer with a message in which the translated sentence is contained, e.g."translate_sentence english_german dog cat milk".

Chapter 8. External Processes

One prominent application for agents is wrapping legacy systems and "agentifying" them. Hence, it is an important point how separate processes can interact with Jadex agents as these applications often use other means of communications such as sockets or RMI. A Jadex agent executes behavior sequentially and does not allow any parallel access to its internal structures due to integrity constraints. For this reason it is disallowed and discouraged to block the active plan thread e.g. by opening sockets and waiting for connections or simply by calling `Thread.sleep()`. This can cause the whole agent to hang because the agent waits for the completion of the current plan step. It will possibly abort the plan when the maximum plan step execution time has been exceeded (if the maximum execution is restricted within the agent runtime.properties). When external processes need to interact directly with the agent, they have to use methods from the so called `jadex.runtime.IExternalAccess` interface, which offers the most common agent methods.

8.1. Exercise G1 - Socket Communication

We extend the simple translation agent from exercise C2 with a plan that sets up a server socket which listens for translation requests. Whenever a new request is issued (e.g. from a browser) a new goal containing the client connection is created and dispatched. The translation plan handles this translation goal and sends back some HTML content including some text and the translated word.

Create a new file for the ServerPlanG1.

- Declare the ServerSocket as attribute within the plan

```
protected ServerSocket server;
```

- Create a constructor which takes the server port as argument and creates a the server within it:

```
try {
    this.server = new ServerSocket(port);
}
catch(IOException e) {
    throw new RuntimeException(e.getMessage());
}
getLogger().info("Created: "+server);
```

- Additionally create a close method that can be used for shutting down the server socket:

```
public void close() {
    try {
        getExternalAccess().getLogger().info("Closing: "+server);
        server.close();
    }
    catch(IOException e) {
        e.printStackTrace();
    }
}
```

- In the body simply start a new thread that will handle client request in the run method. Additionally add an agent listener that gets invoked when the agent will be terminating. In this case the server is shut down:

```
new Thread(this).start();
getScope().addAgentListener(new IAgentListener() {
    public void agentTerminating(AgentEvent ae) {
        close();
    }
}, false);
```

- In the threads run method create and dispatch goals for every incoming request:

```
while(true)
{
    Socket    client    = server.accept();
    IGoal goal = getExternalAccess().getGoalbase().createGoal("translate");
    goal.getParameter("client").setValue(client);
    getExternalAccess().getGoalbase().dispatchTopLevelGoal(goal);
}
```

Modify the EnglishGermanTranslationPlanG1 to handle translation goals.

- Extract the socket from the goal and read the English word:

```
Socket client = (Socket)getParameter("client").getValue();
BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));
String request = in.readLine();
// Read the word to translate from the input string
```

- Translate the word as usual by using the query
- Send back answer to the client:

```
PrintStream out = new PrintStream(client.getOutputStream());
out.print("HTTP/1.0 200 OK\r\n");
out.print("Content-type: text/html\r\n"); out.println("\r\n");
out.println("<html><head><title>TranslationG1 - "+eword
    +"</title></head><body>");
out.println("<p>Translated from english to german: "+eword+" = "+gword+".");
out.println("</p></body></html>");
client.close();
```

Create a file TranslationG1.agent.xml by copying TranslationC2.agent.xml.

- The addword plan and event declarations are not used and can be removed for clarity.
- Introduce the translation goal type:

```
<achievegoal name="translate">
    <parameter name="client" class="java.net.Socket"/>
</achievegoal>
```

- Introduce the new plan for setting up the server and start the plan initially:

```
<plan name="server">
    <body>new ServerPlanG1(9099)</body>
</plan>
...
<configurations>
    <configuration name="default">
        <plans>
            <initialplan ref="server"/>
        </plans>
    </configuration>
</configurations>
```

- Modify the trigger of the translation plan to react on translation goals and add a parameter for the client:

```
<plan name="egtrans">
    <parameter name="client" class="Socket">
```

```
        <goalmapping ref="translate.client"/>
    </parameter>
    <body>new EnglishGermanTranslationPlanG1()</body>
    <trigger>
        <goal ref="translate"/>
    </trigger>
</plan>
```

Start and test the agent.

- Start the agent and open a browser to issue translation request. This can be done by entering the server url and appending the word to translate, e.g. http://localhost:9099/dog. The result should be printed out in the returned web page.

Chapter 9. Conclusion and Outlook

We hope you enjoyed working through the tutorial and now are equipped at least with a basic understanding of the Jadex BDI reasoning engine. Nevertheless, this tutorial does not cover all important aspects about agent programming in Jadex. Most importantly the following topics have not been discussed:

9.1. Ontologies

Ontologies can be used for describing message contents. In more complex applications you usually want to transfer objects instead of simple strings. In Jadex for this purpose you could use arbitrary Java beans in connection with the SFipa.NUGGETS_XML language. If this language is specified for a message event the nuggets XMLEncoder/Decoder will be used to encode/decode the message content. If you don't want to write the beans by hand you also could use the beanynizer tool [Jadex Tool Guide] to generate beans directly from an ontology description defined in Protégé. Further information about ontologies you can find in the [Jadex User Guide], on the Protégé homepage http://protege.stanford.edu/ and in the source code of various examples shipped with the Jadex distribution.

9.2. Protocols Capability

The protocols capability, which belongs to the Jadex planlib provides ready-to-use implementations of some common interaction protocols. In Section 7.4, “Exercise F4 - A Multi-Agent Scenario” you could already see how to use the initiator side of the FIPA request protocol. For more details on the protocols capability, there is a separate section in the [Jadex User Guide].

9.3. Goal Deliberation

This tutorial only mentions the different goal types available in Jadex (perform, achieve, query and maintain). It does not cover aspects of goal deliberation, i.e. how a conflict free pursuit of goals can be ensured. Jadex offers the built-in *Easy Deliberation* strategy for this purpose. The strategy allows to constrain the *cardinality* of active goals. Additionally, it is possible to define *inhibition links* between goals that allow to establish an ordering of goals. Inhibited goals are suspended and can be reactivated when the reason for their inhibition has vanished, e.g. another goal has finished processing. Please refer also to the [Jadex User Guide] for an extended explanation. Background information is available in the paper [Pokahr et al. 2005a].

9.4. Plan Deliberation

If more than one plan is applicable for a given goal or event the Jadex interpreter has to decide which plan actually will be given a chance to handle the goal resp. event. This decision process called plan deliberation can be customized with *meta-level reasoning*. This means that a custom defined meta-level goal is automatically raised by the system in case a plan decision has to be made. This meta goal can be handled by a corresponding meta-level plan which has the task to select among the candidate plans. Further details about meta-level reasoning can be found in the [Jadex User Guide] and by looking into the source code of the "puzzle" agent included in the Jadex release.

9.5. Jadex BDI Architecture

During some of the exercises you may have used the Jadex debugger for executing Jadex agents step-by-step. But what makes-up one such step in the debugger? All steps represent *BDI meta-actions* meaning that they are not at the application-level but on the architecture level. Examples for such meta- actions are "selecting plans

for a given event", "executing a plan step", "creating a new goal" and many more. Basically, the Jadex interpreter selects one meta-action after another and executes them when they are applicable in the current situation. This new architecture makes the Jadex framework efficient and also extensible as new meta-actions can be added to the system easily. Details about the architecture are described in the [Jadex User Guide] and the paper [Pokahr et al. 2005b].

Bibliography

[Bauer et al. 2001] B. Bauer, J. Müller, and J. Odell. *Agent UML: A Formalism for Specifying Multiagent Interaction*. P. Ciancarini and M. Wooldridge. *Proceedings of the First International Workshop on Agent-Oriented Software Engineering (AOSE 2000)*. Springer. Berlin, New York. 2001. pp.91-103.

[Bellifemine et al. 2007] F. Bellifemine, G. Caire, and D. Greenwood. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons. New York, USA. 2007.

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

[Braubach et al. 2004] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf. *Goal Representation for BDI Agent Systems*. R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Proceedings of the Second Workshop on Programming Multiagent Systems: Languages, frameworks, techniques, and tools (ProMAS04)*. Springer. Berlin, New York. 2004. pp.9-20.

[Braubach et al. 2005a] L. Braubach, A. Pokahr, and W. Lamersdorf. *Jadex: A BDI Agent System Combining Middleware and Reasoning*. R. Unland, M. Klusch, and M. Calisti. *Software Agent-Based Applications, Platforms and Development Kits*. Birkhäuser. 2005. pp.143-168.

[Braubach et al. 2005b] L. Braubach, A. Pokahr, and W. Lamersdorf. *Extending the Capability Concept for Flexible BDI Agent Modularization*. R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Proceedings of the Third International Workshop on Programming Multi-Agent Systems (ProMAS'05)*. . 2005. pp.99-114.

[Busetta et al. 2000] P. Busetta, N. Howden, R. Rönnquist, and A. Hodgson. *Structuring BDI Agents in Functional Clusters*. N. Jennings and Y. Lespérance. *Intelligent Agents VI, Proceedings of the 6th International Workshop, Agent Theories, Architectures, and Languages (ATAL) '99*. Springer. Berlin, New York. 2000. pp.277-289.

[Hindriks et al. 1999] K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. Meyer. *Agent Programming in 3APL*. N. Jennings, K. Sycara, and M. Georgeff. *Autonomous Agents and Multi-Agent Systems*. Kluwer Academic publishers. 1999. pp. 357-401.

[Huber 1999] M. Huber. *JAM: A BDI-Theoretic Mobile Agent Architecture*. O. Etzioni, J. Müller, and J. Bradshaw. *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS-99)*. ACM Press. New York. 1999. pp. 236-243.

[Jadex Tutorial] L. Braubach and A. Pokahr. *Jadex Tutorial*. 2005.

[Jadex Tool Guide] A. Pokahr and L. Braubach. *Jadex Tool Guide*. 2005.

[Jadex User Guide] A. Pokahr and L. Braubach. *Jadex User Guide*. 2005.

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

[McCarthy et al. 1979] J. McCarthy. *Ascribing mental qualities to machine*. M. Ringle. *Philosophical Perspectives in Artificial Intelligence*. Humanities Press. Atlantic Highlands, NJ. 1979. pp. 161-195.

[Pokahr et al. 2005a] A. Pokahr, L. Braubach, and W. Lamersdorf. *A Goal Deliberation Strategy for BDI Agent Systems*. T. Eymann, F. Klügl, W. Lamersdorf, M. Klusch, and M. Huhns. *In Proceedings of the third German conference on Multi-Agent System TEchnologieS (MATES-2005)*. Springer-Verlag. Berlin

Heidelberg New York. 2005.

[Pokahr et al. 2005b] A. Pokahr, L. Braubach, and W. Lamersdorf. *A Flexible BDI Architecture Supporting Extensibility*. A. Skowron, J.P. Barthes, L. Jain, R. Sun, P. Morizet-Mahoudeaux, J. Liu, and N. Zhong. *Proceedings of The 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-2005)*. IEEE Computer Society. 2005. pp. 379-385.

[Pokahr et al. 2005c] A. Pokahr, L. Braubach, and W. Lamersdorf. *Jadex: A BDI Reasoning Engine*. R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Programing Multi-Agent Systems*. Kluwer Academic Publishers. 2005. pp.149-174.

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

[Shoham 1993] Y. Shoham. *Agent-oriented programming*. D. G. Bobrow. *Artificial Intelligence Volume 60*. Elsevier. Amsterdam. 1993. pp.51-92.

[Winikoff 2005] M. Winikoff. *JACK Intelligent Agents: An Industrial Strength Platform*. R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni. *Programing Multi-Agent Systems*. Kluwer Academic Publishers. 2005. pp.175-193.