Jadex

User Guide

Release 0.941
24. May 2006
http://vsis-www.informatik.uni-hamburg.de/projects/jadex/

Alexander Pokahr
Lars Braubach
Andrzej Walczak

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 an agent-oriented reasoning engine for writing rational agents with XML and the Java programming language. Thereby, Jadex represents a conservative approach towards agent-orientation for several reasons. One main aspect is that no new programming language is introduced. Instead, Jadex agents can be programmed in the state-of-the art object-oriented integrated development environments (IDEs) such as Eclipse and Intellij IDEA. The other important aspect concerns the middleware independence of Jadex. As Jadex is loosely coupled with its underlying middleware, Jadex can be used in very different scenarios on top of agent platforms as well as enterprise systems such as J2EE.

Similar to the paradigm shift towards object-orientation agents represent a new conceptual level of abstraction extending well-known and accepted object-oriented practices. Agent-oriented programs add the explicit concept of autonomous actors to the the world of passive objects. In this respect agents represent active components with individual resoning capabilities. This means that agents can exhibit reactive as well as pro-active behaviour.

1.1. Requirements and Installation

If you want to run the Jadex examples to get a quick overview of the system, you may download one of the read-to-run installer bundles available from the project homepage, or run the system directly from the web.

If you intend to develop agent software with Jadex it is necessary to set up Jadex on your system Only few steps are necessary. It is recommended to do these steps by hand to see how the required components fit together.

Software. The following describes the 3rd party software required to run Jadex.

- **Java.** Jadex has been developed for use with the Java 2 Standard Edition (J2SE), Version 1.4 or any later version. If not already done, download and install a recent Java Development Kit (JDK).
- **Third-Party Libraries.** The Jadex distribution includes a number of third-party libraries. For an accurate list please consult the Appendix C, *Legal Notice*.

Installation. If you have not already done so, download the Jadex distribution .zip and unpack it to a directory of your choice. Afterwards, add at least the following libraries to your class path:

- **`jadex_rt.jar`:** The Jadex runtime jar includes the kernel of the Jadex reasoning engine.
- **`jadex_standalone.jar`:** The Jadex stanalone jar contains the recommended basic agent middleware for Jadex. It represents a fast and efficient agent environment with a minimal memory footprint.
- **`jadex_tools.jar`:** The Jadex tools jar contains all available Jadex tools, namely the Jadex Control Center (JCC) which allows administration of agents and represents the central access point to all other runtime tools: The introspector for viewing the internal state of an agent and also for debugging it via stepwise execution, the tracer for creating visual execution traces that can be used to determine if an agent behaves as intended, the Message Center for fast and easy messsage composition, and the Jadexdoc tool that allows to generate API docs for agents in the spirit of Javadoc.
- **`jibx-run.jar` and `xpp3.jar`:** These jars belong to the JiBX XML databinding framework, which is used to read agent and capability XML files.

Besides these standard libraries, which are needed for the execution of agents, some extra libraries are included

for certain features. The control center uses the JavaHelp system, which requires the `jhall.jar`. The Jadex tracer tool requires additionally the also contained `GraphLayout.jar`. The introspector detail view output can be visually improved (html instead of plain text) by also using a `velocity.jar` (not included, see http://jakarta.apache.org/velocity).

1.2. Getting Started

This chapter describes how to run the examples provided with the Jadex distribution. Following the instructions below you can test if your Jadex installation works correctly.

1.2.1. Compile an Example

When you have downloaded and unpacked the full distribution, you already have available the sources for the examples. If you do not have the sources or you do not want to compile them now, you can skip this section and instead use the precompiled `jadex_examples.jar`.

In the Jadex `src` directory there is an examples directory, which contains subdirectories with different example agents or multi-agent applications built with Jadex. Open a shell or console window, change to the `src` directory and compile the Java source files of the HelloWorld agent by entering

javac jadex/examples/helloworld/*.java
If it doesn't work, check if you have at least `jadex_rt.jar` in your classpath when compiling.

1.2.2. Start an Example Agent

Jadex comes with the Jadex Control Center (JCC) useful for loading and starting Jadex agents. You can start a standalone platform together with the Control Center with the following command:

java jadex.adapter.standalone.Platform
If the platform does not start or the Control Center user interface does not show up, check if you have all necessary libraries, at least `jadex_rt.jar`, `jadex_standalone.jar`, `jadex_tools.jar`, `jibx-run.jar`, `xpp3.jar`, and `jhall.jar` in the classpath.

Once the Control Center has started, you can select agents by using the browse button (named “"..."”) and locating some agent ADF from the examples directory. Besides file selection via a selection dialog you can also add new content root folders and jars to the tree model explorer (using the “+” button). When you want to use the precompiled examples, add the `jadex_examples.jar` from the lib directory. To load a model from the tree, it sufficient to click on a model contained in the folder. If a model was successfully loaded the starter dialog on the right-hand side shows details about the model and allows to enter an agent instance name and additional arguments. To start the HelloWorld agent browse to the `src/jadex/examples/helloworld` folder and select the `HelloWorld.agent.xml`. After loading the agent model, the details panel shows descriptions of the currently loaded agent model.

When you have loaded an agent definition file, all required values to start the agent will be filled in, so you just have to hit the “Start” button to create the agent. In the case of success, the HelloWorld agent will print out a welcome message to the console. When the example agent cannot be started, check if you have started the Standalone platform from the Jadex src directory, and that you have the current directory (“`.`”) in the classpath. (This is necessary for the example classes and XMLs to be found. Alternatively you can add the `jadex/src` directory to the classpath or the tree.)

We recommend to try out the other examples to get an impression of Jadex. You can use either the ExampleStarter agent (in the `src/jadex/examples/starter` directory) to easily start the agent applications or you can

use the provided manager agents in each application. The manager agents start all agents needed for a special application in correct order. Explanations of the examples can be found in the corresponding `readme.txt` files and at `docs/examples/index.html`. For further information on starting agents see the [Jadex Tool Guide].

Chapter 2. Platform Adapters

Jadex is realized as pure reasoning engine. This means that Jadex agents can potentially run on any middleware platform that fulfills some basic services concerning agent management and messaging. Currently, adapters for Jadex have been realized for the agent platforms JADE , for a Standalone platform and an experimental (not released) adapter for Diet.

Note

There is currently no dedicated manual availabe explaining how to build a new middleware adapter. If you are interested in developing a new adapter consider looking into the `jadex.adapter` package, which contains a handful of interfaces that every adapter has to implement. If you have any problems feel free to contact us directly. The same applies if you already developed a new adapter. We would be glad to know about it and possibly getting the chance to announce/link it on the Jadex web site.

In the following it is explained how you can configure and start Jadex using the Standalone and the JADE adapter.

2.1. The Jadex Standalone Adapter

The Jadex Standalone adapter is a fast and efficient execution environment for Jadex agents with a small memory footprint. The Standalone adapter is already contained in the normal Jadex distribution and needs to be put into the classpath (jadex_standalone.jar).

2.1.1. Starting the Jadex Standalone Adapter

You can start the Standalone adapter via the following command line:

java jadex.adapter.standalone.Platform [-conf filename] [-transport classname:port] [-notransport] [-nogui] [-noamsagent] [-nodfagent] [-autoshutdown]

Alternatively you can also use the following command to start the platform directly from the jar:

java -jar jadex_standalone.jar [options]

-*conf*: The property `-conf` can be used to configure the Jadex system. Per default it will search the current directory for the file `jadex.properties` and if not found the classpath will be searched (in each adapter jar a configuration is contained).

-*platformname*: The unique platform name. If more than one Jadex platform should run on the same machine it is useful to start them with different platform names (and with transports at different ports).

-*transport*: The standard transport mechanism for remote communication. As value the Java class name of a class that implements `jadex.adapter.standalone.ITransport` should be supplied. Optionally the port for this transport can also be supplied. As an example one could start Jadex with the "nsmt" (TCP/IP) based transport layer at port 9876 via `-transport jadex.adapter.standalone.transport.nsmt.NSMTransport:9876`

-*notransport*: Starts the platform without a remote transport mechanism.

-*nogui*: Starts the platform without user interface and the corresponding Jadex Control Center agent.

-noamsagent: Starts the platform without creating an Agent Management Service (AMS) agent. Note, this does only prevent remote agent access to the AMS as the AMS service is always available.

-nodfagent: Starts the platform without creating a Directory Facilitator (DF) agent. Note, this does only prevent remote agent access to the DF as the DF service itself is always available.

-autoshutdown: Automatically shut down the platform when the last agent is killed.

2.2. The JADE Adapter

The JADE adapter is not contained in the standard Jadex distribution and needs to be downloaded separately from the Jadex sourceforge download page. It contains the adapter jar (`jadex_jadeadapter.jar`) that should be added to the classpath. In addition (compatibility tested) official JADE jars (Base64.jar, http.jar, iiop.jar, jade.jar, jadeTools.jar) and additionally Crimson (crimson.jar) are contained and should be also added to the classpath.

2.2.1. Starting the JADE Adapter

You can start Jadex with JADE simply by starting JADE with a modified Remote Management Agent (RMA) instead of using the -gui option which activates the default JADE RMA you should use:

java [-Dconf=<conf.properties>] jade.Boot [JADE options] [rma:jadex.adapter.jade.tools.rma.rma]

Alternatively you can also start the platform from jar via:

java -jar jadex_jadeadapter.jar [JADE options] [rma:jadex.adapter.jade.tools.rma.rma]

The system property `-Dconf` can be used to configure the Jadex system. Per default it will search the current directory for the `jadex.properties` and if not found the classpath will be searched (in each adapter jar a configuration is contained).

2.2.2. Agent Migration and Persistence

Among the nice features of JADE is the ability to migrate agents between hosts, and to persist the state of an agent such that it can later be restored. These features are based on Java's serialization mechanism. Jadex has been designed in order to support serialization of agents at runtime. Nevertheless, there are some issues, the application developer has to be aware of:

- Only mobile plans are supported for agents which need to be serialized for migration or persistence. Standard plans are backed by separate threads for each plan, and therefore cannot be serialized.
- All objects that are used from plans or stored as facts in the beliefbase or as parameters (e.g., in goals) also have to be serializable.
- Due to a bug in JADE, you cannot use basic Java types such as `int` or arrays (e.g. `Object[]` or `String[]`) as class of your beliefs or parameters. For details on the bug or to see if it has been fixed, have a look at its entry in the JADE bug database. A simple workaround is to use the wrapper types such as `java.lang.Integer` instead of the basic types, and `Object` instead of array types (or even better, use a belief set instead of an array).

```
<!-- This does not work. -->
<belief name="my_int_belief" class="int">
```

```
    <fact>42</fact>
</belief>
<belief name="my_stringarray_belief" class="String[]">
        <fact>new String[]{"value1", "value2"}</fact>
</belief>

<!-- This works. -->
<belief name="my_integer_belief" class="Integer">
    <fact>42</fact>
</belief>
<belief name="my_arrayobject_belief" class="Object">
        <fact>new String[]{"value1", "value2"}</fact>
</belief>
<beliefset name="my_string_beliefset" class="String">
        <fact>"value1"</fact>
        <fact>"value2"</fact>
</beliefset>
```

- If you want to use SL-based communication or the DF or AMS capabilities (which use SL-based communication internally), you need to update to the latest development version of JADE, available from the JADE subversion repository. The reason for this is, that some bugs in the serialization, e.g., of the `SLCodec` and `SearchConstraints` classes have only been fixed after the last version of JADE (3.3) was released.

The Jadex distribution contains two larger examples which explicitly support migration: Cleanerworld and Puzzle. See packages `jadex.examples.cleanerworld.multi.cleanermobile` and `jadex.examples.puzzle.mobile`. For a simple example supporting migration see the ping example (package `jadex.examples.ping.mobile`).

2.2.3. Using JADE Behaviours

If you have some legacy JADE code that you want to use in a Jadex agent, but you do not want to convert your JADE behaviours to Jadex plans, you can still use them in the old fashioned way. From a plan, you can get a reference to the `jade.core.Agent` which is executing the BDI reasoning engine. You may add your own additional JADE behaviours to this agent, which will then be executed concurrent to your BDI goals and plans. Note, that this programming style is meant for easy porting of legacy JADE applications. In general, you should avoid hybrid JADE/Jadex agents, because the these mixed-style agents may easily become incomprehensible. Moreover, hybrid agents will not be portable to other middleware platforms.

To add a behaviour to a Jadex agent just call the `addBehaviour()` method of the JADE agent, which is accessible using `getScope().getPlatformAgent()`, and casting the result to `jade.core.Agent`:

```
// Add a JADE behaviour to the agent from a plan.
jade.core.Agent agent = (jade.core.Agent)getScope().getPlatformAgent();
agent.addBehaviour(new MyJADEBehaviour());
```

Per default all incoming messages are handled by Jadex. To enable custom JADE behaviours to handle incoming messages, these have to be ignored by the Jadex system. You can specify a message template as an agent property in the ADF to identify those messages that should not be handled by Jadex:

```
<agent ...>
    ...
    <properties>
        <!-- Setup a filter for messages which are handled by JADE behaviours. -->
        <property name="jadefilter">
            MessageTemplate.MatchPerformative(ACLMessage.QUERY_REF)
        </property>
     </properties>
    ...
</agent>
```

Chapter 3. Concepts of the Jadex BDI Reasoning Engine

This chapter shortly sketches the scientific background of Jadex and describes the concepts, and the execution model of Jadex agents.

3.1. The BDI Model of Jadex

Rational agents have an explicit representation of their environment (sometimes called world model) and of the objectives they are trying to achieve. Rationality means that the agent will always perform the most promising actions (based on the knowledgeabout itself and the world) to achieve its objectives. As it usually does not know all of the effects of an action in advance, it has to deliberate about the available options. Forexample a game playing agent may choose between a safe action or an action, whichis risky, but has a higher reward in case of success.

To realise rational agents, numerous deliberative agent architectures exist (e.g. BDI [Bratman 1987], AOP [Shoham 1993], 3APL [Hindriks et al. 1999] and SOAR [Lehman et al. 1996] to mention only the most prominent ones). In these architectures, the internal structure ofan agent and therefore its capability of choosing a course of action is based on mental attitudes. The advantage of using mental attitudes in the design and realisation of agents and multi-agent systems is the natural (human-like) modelling and the high abstraction level, which simplifies the understanding of systems [McCarthy et al. 1979].

Regarding the theoretical foundation and the number of implemented and successfully applied systems, the most interesting and widespread agent architecture is the Belief-Desire-Intention (BDI) architecture, introduced by Bratman as a philosophical model for describing rational agents ([Bratman 1987]). It consists of the concepts of *belief*, *desire* and *intention* as mental attitudes, that generate human action. Beliefs capture *informational* attitudes, desires *motivational* attitudes, and intentions *deliberative* attitudes of agents. [Rao and Georgeff 1995] have adopted this model and transformed it into a formal theory and an execution model for software agents, based on the notion of beliefs, goals, and plans.

Jadex incorporates this model into JADE agents, by introducing beliefs, goals and plans as first class objects, that can be created and manipulated inside the agent. In Jadex, agents have beliefs, which can be any kind of Java object and are stored in a beliefbase. Goals represent the concrete motivations (e.g. states to be achieved) that influence an agent's behaviour. To achieve its goals the agent executes plans, which are procedural recipes coded in Java. The abstract architecture of a Jadex agent is depicted in Figure 3.1, “Jadex Abstract Architecture”.

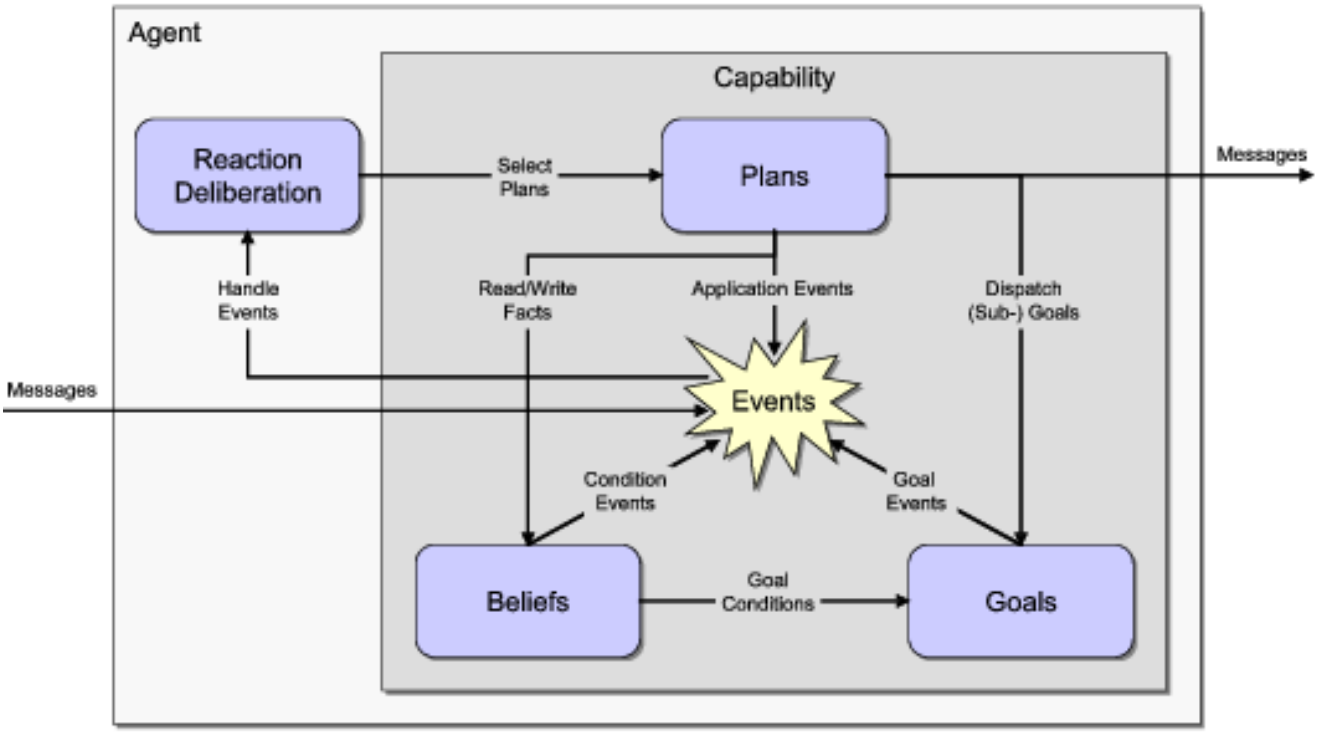

Figure 3.1. Jadex Abstract Architecture

The agent reacts to incoming messages and internal events, and deliberates about its goals. To handle messages and events, and to achieve its goals, the agent selects and executes plans. The current beliefs influence the deliberation process of the agent, and the plans may change the current beliefs while they are executed. Changed beliefs in turn may cause internal events, which may lead to the adoption of new goals and the execution of further plans. In the following the realisation of each of these main concepts in Jadex will be shortly described.

3.1.1. The Beliefbase

The beliefbase stores believed facts and is an access point for the data contained in the agent. Therefore, it provides more abstraction compared to e.g. attributes in the object-oriented world, and represents a unified view of the knowledge of an agent. In Jadex, the belief representation is very simple, and currently does not support any (e.g., logic-based) inference mechanism. The beliefbase contains strings that represent an identifier for a specific belief (similar to table names in relational databases). These identifiers are mapped to the beliefs values, called facts, which in turn can be arbitrary Java objects. Currently two classes of beliefs are supported: simple single-fact beliefs, and belief sets. Beliefs and belief sets are strongly typed, and the beliefbase checks at runtime, that only properly typed objects are stored.

On top of this simple belief representation, Jadex adds several advanced features, such as an OQL-like query language (adopted from the object-relational database world), conditions that trigger plans or goals when some beliefs change (resembling a rulebased programming style), and beliefs that are stored as expressions and evaluated dynamically on demand.

3.1.2. The Goal Structure

Unlike traditional BDI systems, which treat goals merely as a special kind of event, goals are a central concept in Jadex. Jadex follows the general idea that goals are concrete, momentary desires of an agent. For any goal it has, an agent will more or less directly engage into suitable actions, until it considers the goal as being reached,

unreachable, or not desired any more. Unlike most other systems, Jadex does not assume that all adopted goals need to be consistent to each other. To distinguish between just adopted (i.e. desired) goals and actively pursued goals, a goal lifecycle is introduced which consists of the goal states *option*, *active*, and *suspended* (see Figure 3.2, “Goal Lifecycle”). When a goal is adopted, it becomes an option that is added to the agent's desire structure. Application specific goal deliberation mechanisms are responsible for managing the state transitions of all adopted goals (i.e. deciding which goals are active and which are just options). In addition, some goals may only be valid in specific contexts determined by the agent's beliefs. When the context of a goal is invalid it will be suspended until the context is valid again.

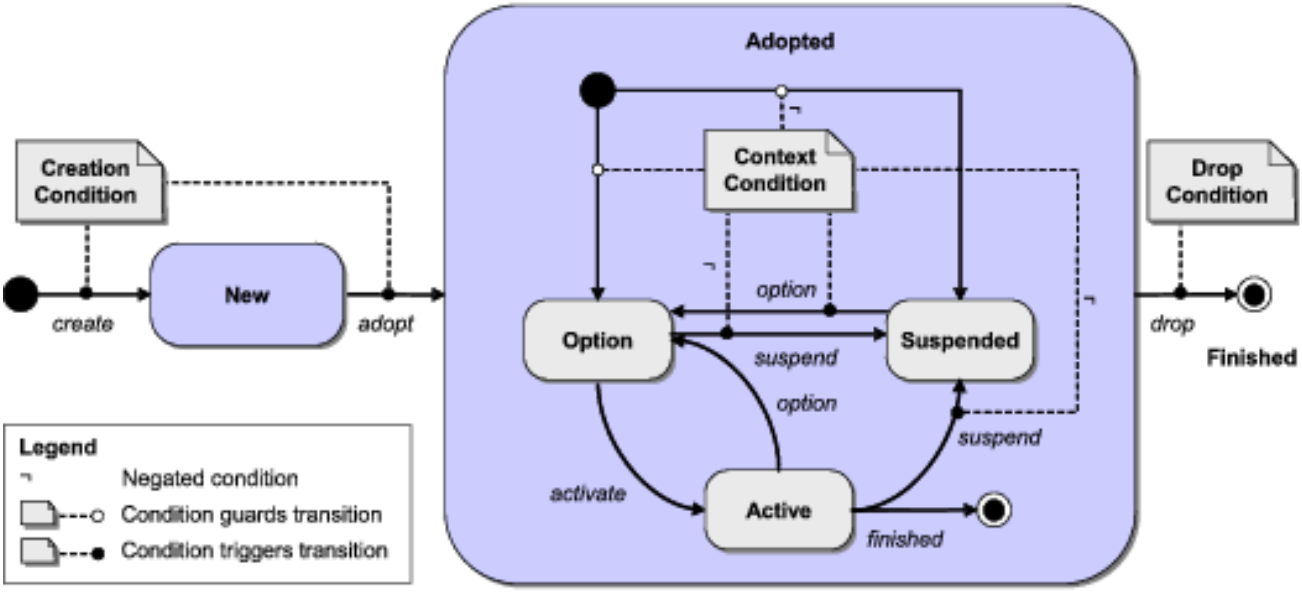

Figure 3.2. Goal Lifecycle

Four types of goals are supported by the Jadex system: Perform, achieve, query, and maintain goals as introduced by JAM [Huber 1999]. A *perform goal* states that something should be done but may not necessarily lead to any specific result. For example, a waste-pickup robot may have a generic goal to wander around and look for waste, which is done by a specific plan for this functionality. The *achieve goal* describes an abstract target state to be reached, without specifying how to achieve it. Therefore, an agent can try out different alternatives to reach the goal. Consider a player agent that needs certain resources in a strategy game: It could choose to negotiate with other players or try to find the required resources itself. The *query goal* represents a need for information. If the information is not readily available, plans are selected and executed to gather the needed information. For example a cleaner robot that has picked up some waste needs to know where the next waste bin is located. If it already knows the location it can directly head towards the waste bin, otherwise it has to find one, e.g by executing a search plan. The *maintain goal* specifies a state that should be kept (maintained) once it is achieved. It is the most abstract goal in Jadex. Not only does it abstract from the concrete actions required to achieve the goal, but also it decouples the creation and adoption of the goal from the timepoint when it is executed. For example the goal to keep a reactor temperature below a certain level is a maintain goal that gets triggered whenever the temperature exceeds the normal operating level. As with achieve and query goals, to (re)establish the desired target state of a maintain goal, the agent may try out several plans, until the state is reached.

In the Jadex System, goals are represented as objects with several attributes. The target state of achieve goals can be explicitly specified by an expression (e.g., referring to beliefs), which is evaluated to check if the goal is achieved. Attributes of the goal, such as the name, facilitate plan selection, e.g. by specifying that a plan can handle all goals of a given name. Additional (user-defined) goal parameters guide the actions of executing plans. For example in a goal to search for services (e.g. using the FIPA directory facilitator service), additional

search constraints could be specified (such as the maximum cardinality of the result set). The structure of currently adopted goals is stored in the goalbase of an agent. The agent has a number of top-level goals, which serve as entry points in the goalbase. Goals in turn may have subgoals, forming a hierarchy or tree of goals.

3.1.3. Plan Specification

The main functionality of agents is captured in plans. An agent developer has to define the head and the body of a plan. The head contains the conditions under which the plan may be executed and is specified in the agent definition file. The body of the plan is a procedural recipe describing the actions to take in order to achieve a goal or react to some event. The current version of Jadex supports plan bodies written in Java, providing all the flexibilities of the Java programming language (object-oriented programming, access to third party packages, etc.).

At runtime, plans are instantiated to handle events and to achieve goals. Activation triggers in the plan headers are used to specify if a plan should be instantiated when a certain event occurs. In addition, so called initial plans get executed when the agent is born. Running plans create additional filters to wait for specific events, which trigger subsequent plan steps.

3.2. Agent Definition

To create and start an agent, the system needs to know the properties of the agent to be instantiated. The state of an agent is determined by the beliefs, the goals, the running plans, as well as the library of known plans. The complete definition of an agent is captured in a so called *agent definition file* (ADF). The ADF is kind of a class description for agents: From the ADF agents get instantiated like Objects get instantiated from their class. For example, the different player agents from Black Jack (`src/jadex/examples/blackjack`) share `Player.agent.xml` as their definition file.

In the ADF, the developer defines the initial beliefs and goals using a Java-like syntax for initial facts and goal parameters. Plans are declared by specifying how to instantiate them from their Java class. For plans to be instantiated on demand (called passive plans) a trigger (e.g. event) has to be stated. The trigger can be omitted in the case of a plan to be executed, when the agent starts (initial plan). In addition to the BDI components some other information is stored in the ADF, e.g. service descriptions for registering the agent at a directory facilitator.

3.3. Execution Model of a Jadex Agent

This sections shows the operation of the reasoning component, given the Jadex BDI concepts (see Figure 3.3, “Jadex Execution Model”). Since version 0.93 Jadex does not employ the classical BDI-interpreter cycle as described in the literature [Rao and Georgeff 1995] but uses a new agenda based execution scheme (described more extensive and formally in [Pokahr et al. 2005b]). The interpreter consists of an agenda component holding the scheduled meta-actions to execute. The basic mode of operation is simple: The agent selects a meta-action from its agenda and executes it when the the action's preconditions hold. Otherwise the action is simply dropped. The execution of the action may produce further actions that are added to the agenda following a customizable insertion strategy. Currently, the insertion strategy mainly distinguishes between related and unrelated actions, whereby related actions are added as child nodes to the current node.

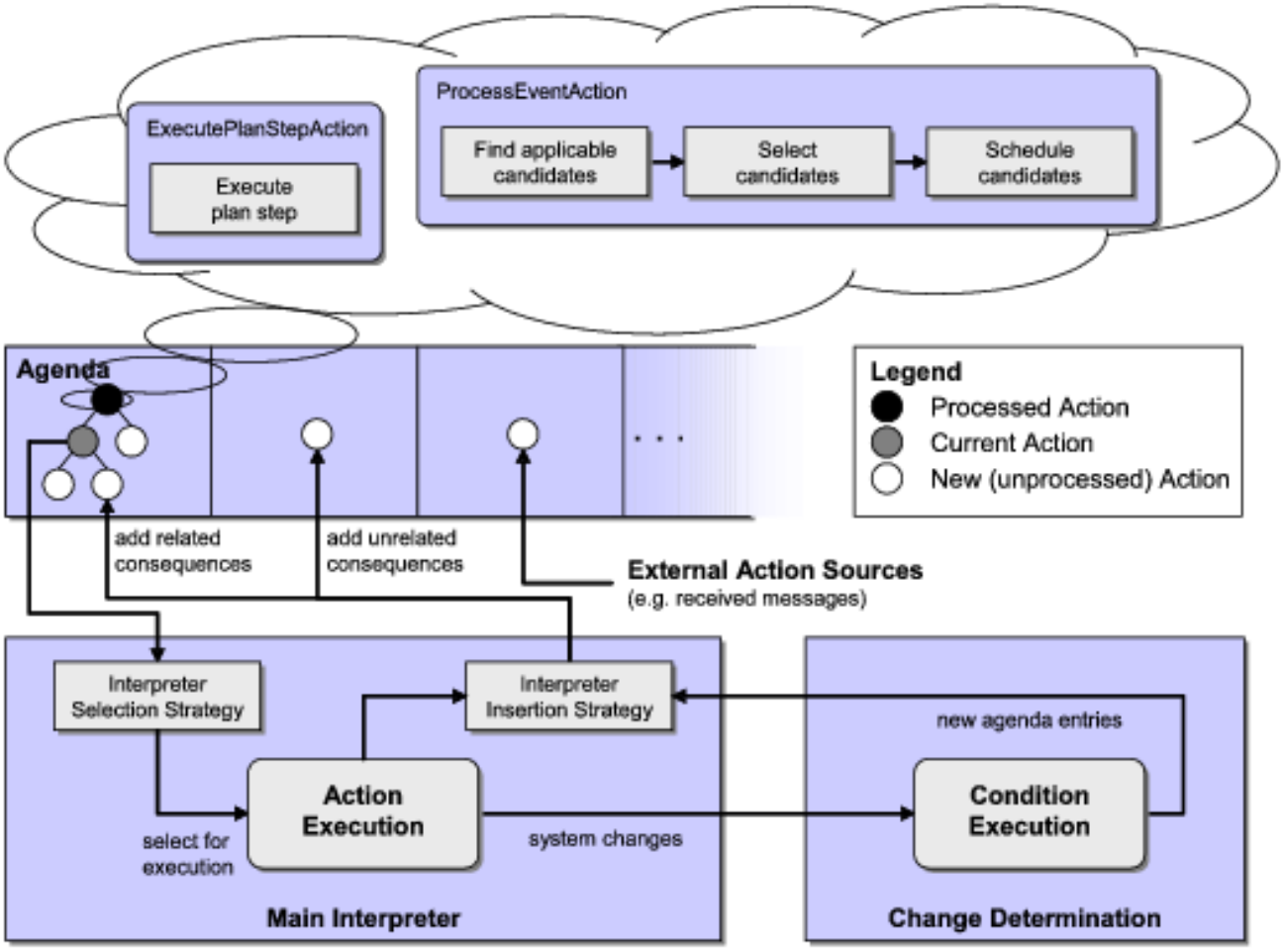

Figure 3.3. Jadex Execution Model

Besides the creation of new agenda entries, the execution of actions can have further side-effects that are of importance for the agent, e.g. when a belief is changed or a goal is dropped. These occurrences are captured within `jadex.runtime.SystemEvents` and may cause system changes which are computed by a change determination component accordingly. To determine which effects certain SystemEvents have, the component evaluates affected conditions. If a condition triggers, new agenda actions may be produced in turn and are added to the agenda.

Having outlined the mode of operation the question arises which kinds of actions are contained within the agenda? These action are not application level actions, but are inter alia derived from the classical BDI interpreter cycle and represent BDI-meta actions. Two typical BDI-meta actions are displayed at the top of Figure 3.3, “Jadex Execution Model” namely the `ProcessEventAction` and the `ExecutePlanStepAction`. The ProcessEventAction encapsulates the well-known BDI plan finding process. The meta action searches for applicable plans matching to an event or goal occurence, selects candidates from the list and schedules them for execution by creating `ExecutePlanStepActions` for each candidate. An `ExecutePlanStepAction` simply execute one step of its plan and produces a new `ExecutePlanStepAction` when further steps for this plan are necessary. (All meta-actions are implemented in the `jadex.runtime.impl.agenda` package).

Advantages of the new approach are that the new mechanism offers a much higher degree of extensibility and flexibility as new BDI-meta actions can be easily added to the system if desired. One concrete effect already contained in this version is the support for goal deliberation via the "Easy Deliberation" strategy Section 8.7, “ Goal Deliberation with "Easy Deliberation" ” which is realized with extended meta-actions.

Chapter 4. Agent Specification

The programmer's guide is a reference to the concepts and constructs available, when programming Jadex agents. It is not meant as a step-by-step introduction to the programming of Jadex agents. For a step-by-step introduction consider working through the tutorial [Jadex Tutorial].

4.1. Overview

To develop applications with Jadex, the programmer has to create two types of files: XML agent definition files (ADF) and Java classes for the plan implementations. The ADF can be seen as a type specification for a class of instantiated agents. For example ping agents (`src/jadex/examples/ping`) that are capable of answering ping requests are defined by the `Ping.agent.xml` file, and use a plan implemented in the file `PingPlan.java`. The user guide describes both aspects of agent programming, the XML based ADF declaration and the plan programming Java API, and highlights the interrelations between them. Detailed reference documentation for the XML definition as well as the plan programming API is also separately available in form of the generated XML schema documentation and the generated Javadocs. Figure 4.1, “ Components of a Jadex agent ” depicts how XML and Java files together define the functionality of an agent. To start an agent, first the ADF is loaded, and the agent is initialized with beliefs, goals, and plans as specified.

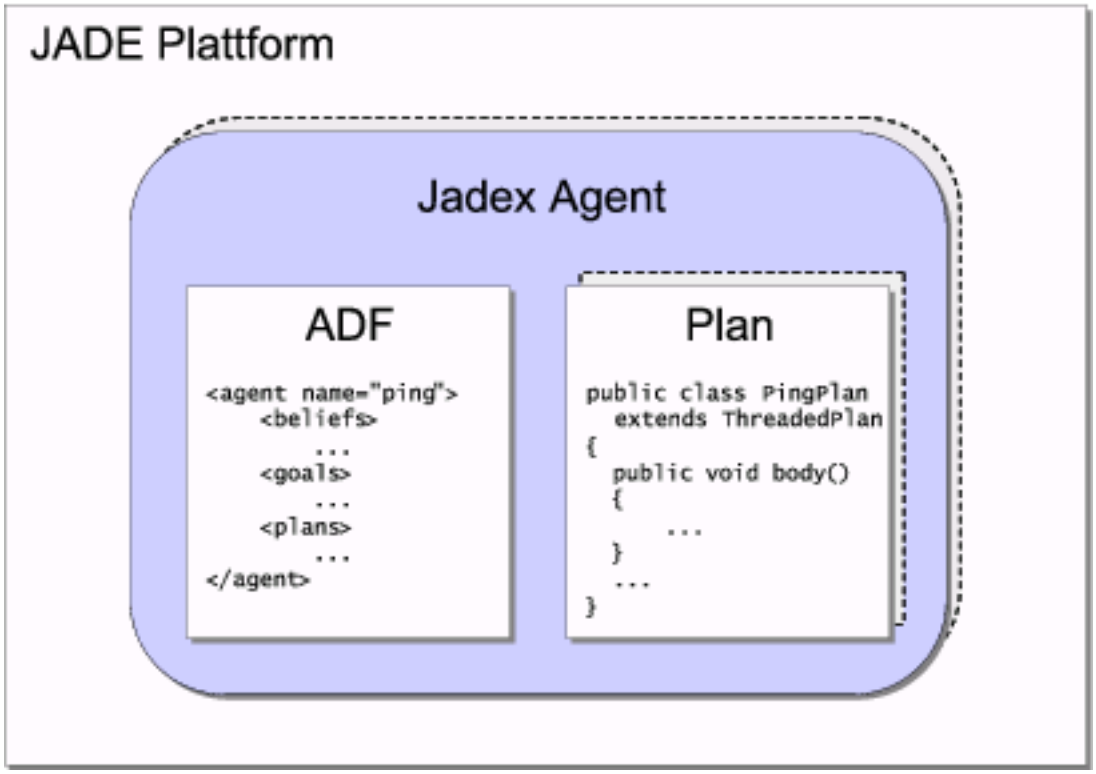

Figure 4.1. Components of a Jadex agent

4.2. Structure of Agent Definition Files (ADFs)

The fully declared head of an ADF looks like shown in Figure 4.2, “Header of an agent definition file”. First, the agent tag specifies that the XML document follows the `jadex-0.94.xsd` schema definition which allows to verify that the document is not only well formed XML but also a valid ADF. The name of the agent type is spe-

cified in the name attribute of the agent tag, which should match the file name without suffix (`.agent.xml`). It is also used as default name for new agent instances, when the ADF is loaded in the starter panel of the Jadex Control Center (see [Jadex Tool Guide]). The package declaration specifies where the agent first searches for required classes (e.g., for plans or beliefs) and should correspond to the directory, the XML file is located in. Additionally required packages can be specified using the `<imports>` tag (see Chapter 5, *Imports*). The Jadex engine requires some properties for initialization, which are by default taken from the file `jadex/config/runtime.properties.xml`. Normally, this is not of interest for agent developers, as it is only concerned with system internals, but developers who whish to change the behaviour of the Jadex engine can use the properties attribute to provide their own property XML file with customized settings (see Chapter 13, *Properties* for a detailed description).

```
<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-0.94.xsd"
       name="Ping"
       package="jadex.examples.ping"
       properties="jadex.config.runtime">
       ...
</agent>
```

Figure 4.2. Header of an agent definition file

Figure 4.3, “ Jadex agent XML schema ” shows which elements can be specified inside an agent definition file (please refer also to the commented schema documentation generated from the schema itself in `docs/schemadoc`). The `<imports>` tag is used to specify, which classes and packages can be used by expressions throughout the ADF. To modularize agent functionality, agents can be decomposed into so called capabilities. The capability specifications used by an agent are referenced in the `<capabilities>` tag. The core part of the agent specification regards the definition of the beliefs, goals, and plans of the agent, which are placed in the `<beliefs>`, `<goals>`, and `<plans>` tag, respectively. The events known by the agent are defined in the `<events>` section. The `<expressions>` tag allows to specify expressions and conditions, which can be used as predefined queries from plans. The `<properties>` tag is used for custom settings such as debugging and logging options. Finally, in the <initialstates> section, predefined configurations containing, e.g., initial beliefs, goals, and plans, are specified.

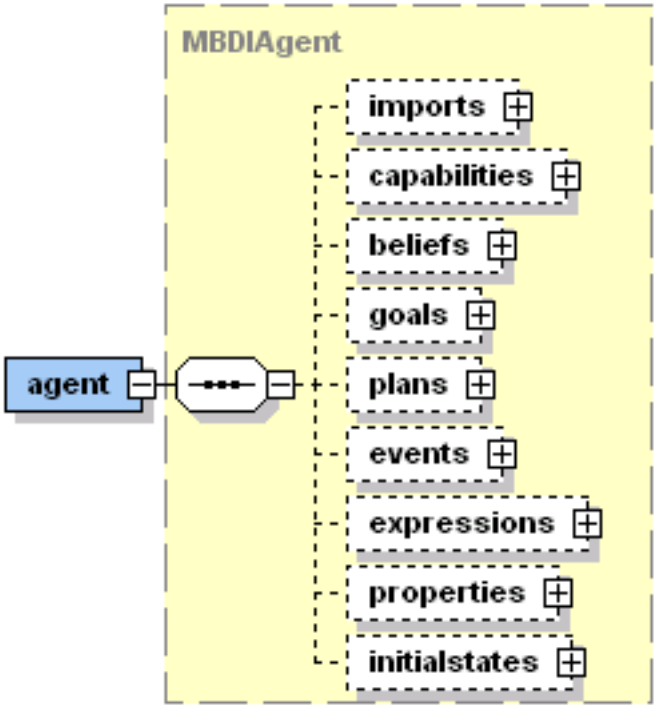

Figure 4.3. Jadex agent XML schema

When an ADF is loaded, Java objects are created for the XML elements (e.g., beliefs, goals, plans) defined in the ADF. The interfaces for these so called model elements reside in the package `jadex.model`. Examples are `IMBelief`, `IMGoal`, `IMPlan`. In most cases, you do not need to access these elements. When the agent is executed, instances of the model elements are created; so called runtime elements (package `jadex.runtime`, e.g., `IBelief`, `IGoal`, `IPlan`). This ensures that for modelled elements (e.g., `IMPlan` objects) at runtime several instances (`IPlan` objects) can be created. For example, the ping agent will instantiate a ping plan (`IPlan`) for each received ping request message, based on the plan specification in the ADF (`IMPlan`). Think of the relation between model elements and runtime elements as corresponding to the relation between `java.lang.Class` and `java.lang.Object`. When programming plans, you are mostly concerned with the runtime elements, unless the agent model should be changed dynamically at runtime. In this case you can fetch model elements by calling `getModelElement()` on a runtime element.

Chapter 5. Imports

The `<imports>` tag is used to specify, which classes and packages can be used by Java expressions throughout an agent or capability definition file. The import section with an ADF resembles very much the Java import section of a class file. A Jadex import statement has the same syntax as in Java allowing single classes as well as whole packages being included.

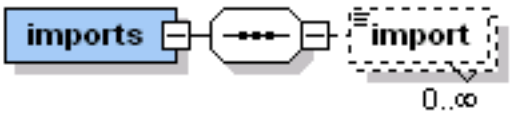

Figure 5.1. The Jadex imports XML schema part

The imports are used for searching Java classes as well as non-Java agent artifacts such as `agent.xml` or `capability.xml` files. It is not necessary to declare an import statement for the actual package of the ADF as this is automatically considered.

5.1. Import Examples

In the following some simple code snippets from an ADF are shown that demonstrate how import statements are declared and subsequently used, e.g., in facts of beliefs, or to include a capability from another package.

```
...
<imports>
    <!-- Import only the HashMap class. -->
    <import>java.util.HashMap</import>

    <!-- Import all classes of the awt package. -->
    <import>java.awt.*</import>

    <!-- Import a movement package containing, e.g., a Move capability. -->
    <import>movement.*</import>
    ...
</imports>

<capabilities>
    <!-- Use the imported movement.Move capability. -->
    <capability name="movecap" file="Move"/>
</capabilities>

<beliefs>
    <!-- Use the imported java.util.HashMap. -->
    <belief name="data">
        <fact>new HashMap()</fact>
    </belief>

    <!-- Use the imported java.awt.Frame. -->
    <belief name="gui">
        <fact>new Frame()</fact>
    </belief>
</beliefs>
...
```

Figure 5.2. Example import declaration and usage

Chapter 6. Capabilities

The term “capability” is used for different purposes in the agent community. In the context of Jadex, the term is used to denote an encapsulated agent module composed of beliefs, goals, and plans. The concept of an agent module (and the usage of the term “capability”) was proposed by Busetta et al. [Busetta et al. 2000] and first implemented in JACK Agents [Winikoff 2005]. Capabilities allow for packaging a subset of beliefs, plans, and goals into an agent module and to reuse this module wherever needed. Capabilities can contain subcapabilities forming arbitrary hierarchies of modules. In Jadex, a revised and extended capability model has been implemented as described in [Braubach et al. 2005b]. In this model, the connection between a parent (outer) and a child (inner) capability is established by a uniform visibility mechanism for contained elements (see Figure 6.1, “Capability concept”).

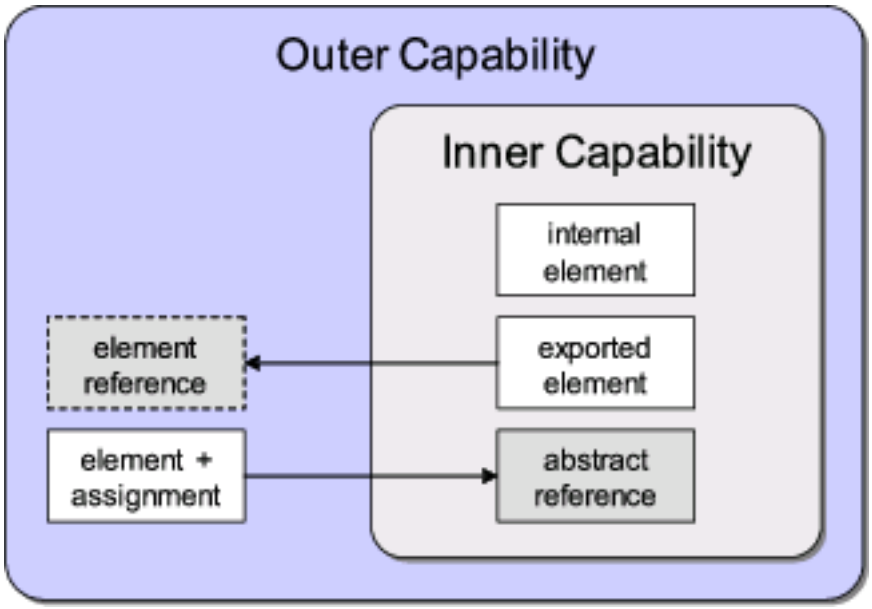

Figure 6.1. Capability concept

6.1. Capability Definition

A capability is basically the same as an agent, but without its own reasoning process. On the other hand, an agent can be seen as a collection (i.e. subcapability hierarchy) of capabilities plus a separate reasoning process shared by all its capabilities. Each agent has at least one capability (sometimes called “root capability”) which is given by the beliefs, goals, plans, etc. contained in the agent's XML file. To create additional capabilities for reuse in different agents, the developer has to write capability definition files. A capability definition file is similar to an agent definition file, but with the `<agent>` tag replaced by `<capability>`. The `<capability>` tag has the same substructure as the `<agent>` tag described in Section 4.2, “Structure of Agent Definition Files (ADFs)”. Note that the `<capability>` tag has `name` and `package` attributes, but no `propertyfile` attribute. As there are so many similarities between agent definition files and capability definition files, we commonly use the term “ADF” to denote both.

```
<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-0.94.xsd"
       name="MyCapability"
       package="mypackage">
```

```
       <beliefs> ... </beliefs>
       <goals> ... </goals>
       <plans> ... </plans>
       ...
</capability>
```

Figure 6.2. Capability XML file header

6.2. Using Capabilities

Agents and capabilities may be composed of any number of subcapabilities which are referenced in a `<capabilities>` tag. To reference a capability, a local `name` and the location of the capability definition has to be supplied in the `file` attribute as absolute or relative file name or capability type name. Type names are resolved using the package and import declarations, and can therefore be unqualified or fully qualified. Capabilities from the `jadex.planlib` package, such as the DF capability, which have platform-specific implementations, must always be referenced using a fully qualified type name.

```
<agent ...>
    <capabilities>
        <!-- Referencing a capability using a filename. -->
        <capability name="mysubcap" file="mypackage/MyCapability.capability.xml"/>

        <!-- Referencing a capability using a fully qualified type name. -->
        <capability name="dfcap" file="jadex.planlib.DF"/>
        ...
    </capabilities>
    ...
</agent>
```

Figure 6.3. Including subcapabilities

6.3. Elements of a Capability

The capability introduces a scoping of the BDI concepts. By default all beliefs, goals, and plans have local scope (i.e., are not `exported`), that is they can only be used in the capability where they have been defined. This restriction can be relaxed by declaring elements as exported or abstract for making them accessible from the outer capability (cf. Figure 6.1, “Capability concept”). In the outer capability such elements can be used when an explicit reference (with its own possibly different name) to those elements is established. In Figure 6.4, “Jadex references XML schema elements” this reference mechanism, which applies to all elements in the same manner, is exemplarily depicted for beliefs. In the following the possible use cases are described.

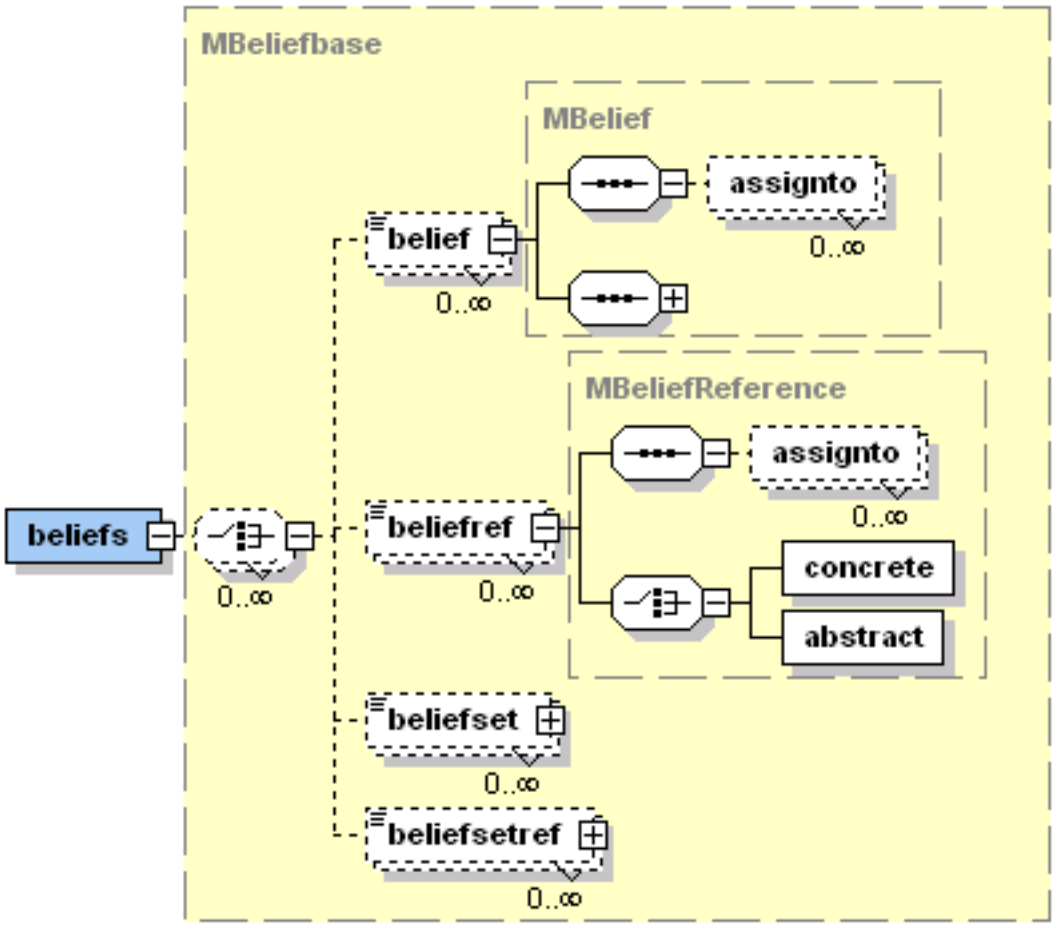

Figure 6.4. The Jadex references XML schema elements (using beliefs as example)

6.3.1. Making an Element Accessible for the Outer Capability

For this purpose the element must declare itself as exported (using the `exported="true"` attribute) in the inner capability. In the outer capability, a reference (e.g., `<beliefref>`) has to be declared, which directly references the original element (using dot notation `"capname.belname"`) within the concrete tag. An example for an exported belief is shown below.

Inner Capability `A`.

```
<belief name="myexportedbelief" type="exported" class="MyFact"/>
```

Outer Capability `B` includes `A` under the name `mysubcap`.

```
<beliefref name="mysubbelief">
    <concrete ref="mysubcap.myexportedbelief"/>
</beliefref>
```

6.3.2. Defining an Abstract Element

This means the element itself provides no implementation and needs to be assigned from an outer capability.

For this purpose an abstract element reference (e.g., `<beliefref>`) has to be declared. An outer capability can provide an implementation for this abstract element by defining a concrete element (or another reference) and assigning it to the abstract reference (using the `<assignto>` tag). In addition, the abstract element can be declared as optional (using the `optional="true"` attribute of the abstract tag) requiring no outer element assignment. At runtime, such unassigned abstract elements are not accessible, and trying to use them will result in runtime exceptions. For some of the elements (e.g., beliefs) it can be tested at runtime with the `isAccessible()` method from within plans, if a reference is connected.

Inner Capability `A`.

```
<beliefref name="myabstractbelief" type="exported" class="MyFact">
    <abstract/>
</beliefref>
```

Outer Capability `B` includes `A` under the name `mysubcap`.

```
<belief name="mybelief" class="MyFact">
    <assignto ref="mysubcap.myabstractbelief"/>
</belief>
```

6.4. Predefined Capabilities

The package `jadex.planlib` contains two capabilities with predefined functionality, useful for many kinds of applications. The `DF` capability allows for easy handling of directory facilitator related issues such as (de)registration and searching of agent services. The `AMS` capability defines goals for interaction with platform services (as provided by the AgentManagementSystem of FIPA-compliant platforms), allowing to create and destroy agents, to search for agents on the platform, as well as to shutdown the whole platform. To use the goals defined in the capabilities, these have to be referenced as described above. Details of the available goals can be found in the Jadexdoc documentation of the `planlib` package. Moreover, you can have a look at the tutorial or the examples to learn how to use the capabilities.

Chapter 7. Beliefs

Beliefs represent the agent's knowledge about its enviroment and itself. In Jadex the beliefs can be any Java objects. They are stored in a belief base and can be referenced in expressions, as well as accessed and modified from plans using the beliefbase interface.

7.1. Defining Beliefs in the ADF

The beliefbase is the container for the facts known by the agent. Beliefs are usually defined in the ADF and accessed and modified from plans. To define a single valued belief or a multi-valued belief set in the ADF the developer has to use the corresponding `<belief>` or `<beliefset>` tags (Figure 7.1, “The Jadex beliefs XML schema part”) and has to provide a name and a class. The name is used to refer to the fact(s) contained in the belief. The class specifies the (super) class of the fact objects that can be stored in the belief. The default fact(s) of a belief may be supplied in enclosed `<fact>` tags. Alternatively, for belief sets a collection of initial facts can be directly specified using a `<facts>` tag. This is useful, when you do not know the number of initial facts in advanvce, e.g., when invoking a static method or retrieving values from a database (see Figure 7.2, “Example belief definition”). References to beliefs and belief sets from inner capabilities can be defined using the `<beliefref>` and `<beliefsetref>` tags (cf. Section 6.3, “Elements of a Capability”).

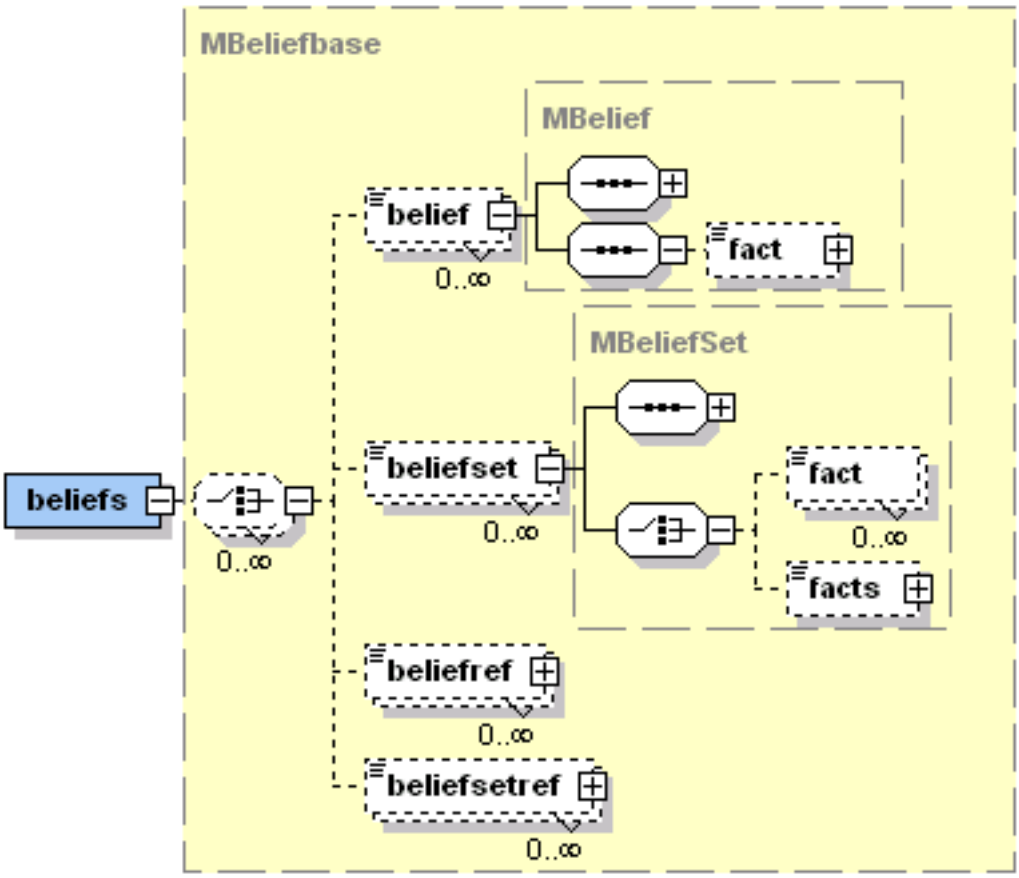

Figure 7.1. The Jadex beliefs XML schema part

```
<agent ...>
    ...
    <beliefs>
        <belief name="my_location" class="Location">
            <fact>new Location("Hamburg")</fact>
```

```
        </belief>
        <beliefset name="my_friends" class="String">
            <fact>"Alex"</fact>
            <fact>"Blandi"</fact>
            <fact>"Charlie"</fact>
        </beliefset>
        <beliefset name="my_opponents" class="String">
            <facts>Database.getOpponents()</facts>
        </beliefset>
        ...
    </beliefs>
    ...
</agent>
```

Figure 7.2. Example belief definition

7.2. Accessing Beliefs from within Plans

From within a plan, the programmer has access to the beliefbase (class `IBeliefbase`) using the `getBeliefbase()` method. The beliefbase provides `getBelief()` / `getBeliefSet()` methods to get the current beliefs and belief sets by name, as well as methods to create new beliefs and belief sets or remove old ones. The content of a belief (class `IBelief`) can be accessed by the `getFact()` method. A belief set (class `IBeliefSet`) is accessed through the `getFacts()` method and will return an appropriately typed array of facts. To check if a fact is contained in a belief set the `containsFact()` method can be used.

The contents of a single fact belief are modified using the `setFact()` method. Setting a fact on a belief will result in overwriting the previous value, if any. For deleting the fact of a single fact belief, you can set the belief value to `null`. Belief sets are manipulated using the `addFact(fact)` / `removeFact(fact)` methods. When removing facts that do not exist from the belief set, the belief set remains unchanged and a warning message will be produced. For the remove operation, the beliefbase relies on the implementation of the `equals()` method of the fact objects. Additionally, `updateFact(fact)` can be used to replace an existing fact value.

```
public void body {
    ...
    IBelief hungry = getBeliefbase().getBelief("hungry");
    hungry.setFact(new Boolean(true));
    ...
    Food[] food = (Food[])getBeliefbase().getBeliefSet("food").getFacts();
    ...
}
```

Figure 7.3. A simple example of using a boolean belief

7.3. Dynamically Evaluated Beliefs

In the ADF the initial facts of beliefs are specified using expressions. Normally, the fact expressions are evaluated only once: When the agent is born. The evaluation behaviour of the fact expression can be adjusted using the `evaluationmode` attribute as further described in Section 11.2, “Expression Properties”. Additionally, an `updaterate` may be specified as attribute of the belief that will cause the fact to be continuously evaluated and updated in the given time interval (in milliseconds).

In the example, the first belief `"time"` is evaluated on access, and will therefore always contain the exact current time as returned by the Java function `System.currentTimeMillis()`. The second belief `"timer"` is not only evaluated on access (i.e., when accessed), but also every 10 seconds (10000 milliseconds). The advantage of using an updaterate for continuously evaluating a belief is that the fact value changes even when it is not accessed, and therefore may trigger conditions referring to that belief. For example, using the `"timer"` belief you could define a condition to invoke a plan that has to be executed every hour. Both options also provide an easy and effective way for making an agent aware of external input (e.g., sensory data available through a Java API).

```
<beliefs>
    <!-- A belief holding the current time (re-evaluated on every access). -->
    <belief name="time" class="long">
        <fact evaluationmode="dynamic">
            System.currentTimeMillis()
        </fact>
    </belief>

    <!-- A belief continuously updated every 10 seconds. -->
    <belief name="timer" class="long" updaterate="10000">
        <fact> System.currentTimeMillis() </fact>
    </belief>
</beliefs>
```

Figure 7.4. Examples of dynamically evaluated beliefs

7.4. Propagation of Belief Changes

To monitor conditions (cf. Chapter 12, *Conditions*), an agent observes the beliefs and automatically reacts to changes of these beliefs, as necessary. Jadex is aware of manipulation operations that are executed directly on beliefs, e.g., by setting the fact of a belief, and of changes due to belief dependencies (i.e., a dynamically evaluated fact expression referencing another belief).

On the other hand, when you retrieve a complex fact object from a belief or belief set and perform operations on it subsequently, the system cannot detect the changes made. To enable the system detecting these changes the standard Java beans event notification mechanism can be used. This means that the bean has to implement the `add/removePropertyChangeListener()` methods and has to fire property change events, whenever an important change has occurred. The belief will automatically add and remove itself as a property change listener on its facts. An example on how to utilize this mechanism is contained in the testcases folder (`src/jadex/testcases/beliefs/BeanChanges.agent.xml`).

Chapter 8. Goals

Goals make up the agent's motivational stance and are the driving forces for its actions. Therefore, the representation and handling of goals is one of the main features of Jadex. The concepts that make up the basis for the representation of goals in Jadex are described in Section 3.1.2, “The Goal Structure” and in more detail in [Braubach et al. 2004] and [Pokahr et al. 2005a]. Currently Jadex supports four different goal kinds and a meta-level goal kind. A perform goal specifies some activities to be done. Therefore the outcome of the goal depends only on the fact, if activities were performed. In contrast, an achieve goal can be seen as a goal in the classical sense by representing a target state that needs to be achieved. Similar to the behaviour of the achieve goal is the query goal, which is used to enquire information about a specified issue. The maintain goal has the purpose to observe some desired world state and actively reestablishes this state when it gets violated. Meta-level goals can be used in the plan selection process for reasoning about events and suitable plans. Figure 8.1, “The Jadex goals XML schema part” shows that a specific tag for each goal kind exists. Additionally, the `<...goalref>` tags allow for the definition of references to goals from other capabilities (cf. Section 6.3, “Elements of a Capability”).

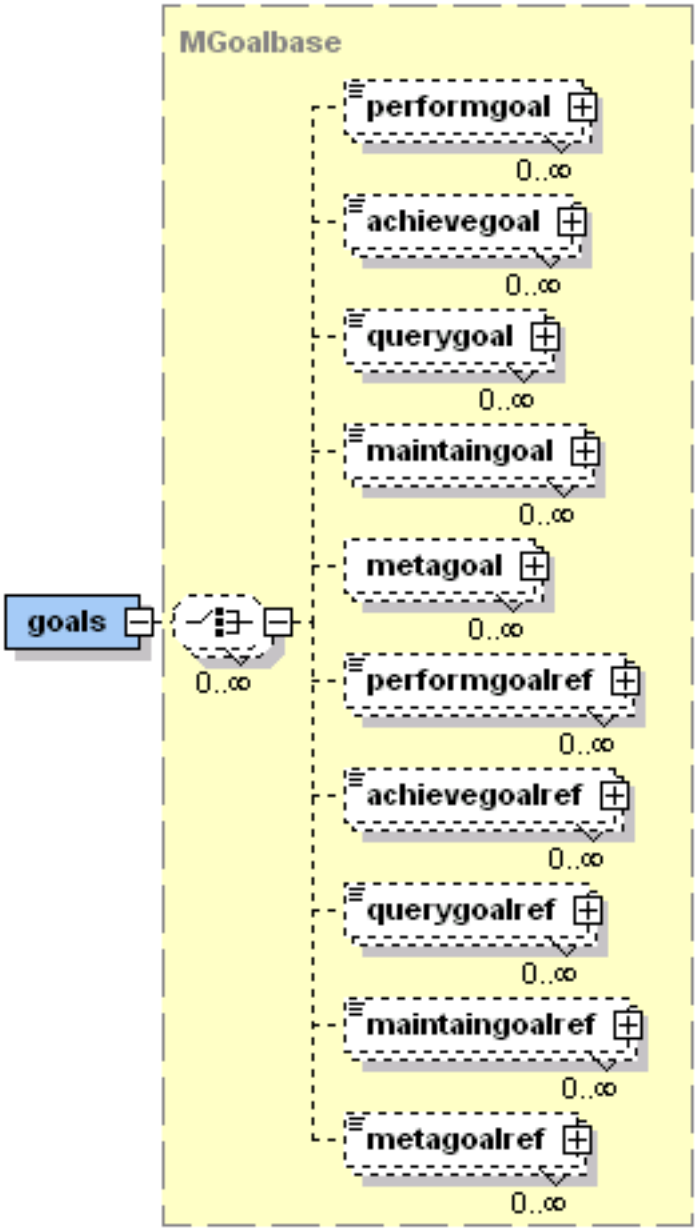

Figure 8.1. The Jadex goals XML schema part

At runtime an agent may have any number of top-level goals, as well as subgoals. Top-level goals may be created when the agent is born (contained in an initial state in the ADF) or will be later adopted at runtime, while subgoals can only be dispatched by running plans. Regardless of how a goal was created, the agent will automatically try to select appropriate plans to achieve all of its goals. The properties of a goal, specified in the ADF, influence when and how the agent handles these goals. In the following, the features common to all goal kinds will be described, thereafter the special features of the specific goal kinds will be explained.

8.1. Common Goal Features

In Figure 8.2, “The Jadex common goal features” the base type of all four goal kinds is depicted to illustrate the shared goal features. In Jadex, goals are strongly typed in the sense that all goal types can be identified per name and all parameters of a goal have to be declared in the XML. The declaration of parameters resembles very much the specification of beliefs. Therefore it is distinguished between single-valued parameters and multi-valued parameter sets. As with beliefs, arbitrary expressions can be supplied for the parameter values. The system distingiushes `in`, `out`, and `inout`, parameters, specified using the `direction` attribute. `in` parameters are set before the goal is dispatched, while `out` parameters are set by the plan processing the goal, and can be read when the goal returns. Additionally, it can be specified that a parameter is not mandatory by using the `optional` attribute. Whenever a goal instance of the declared type is created and dispatched to the system it will be checked with respect to its parameters, and when no value has been supplied for a mandatory `in` parameter or parameter set a runtime exception will be thrown. The creation of new goals can be further influenced by using binding options for parameters via the `<bindingoptions>` tag instead of the `<value>` tag. All possible combinations of assignments of binding parameters will be calculated when the creation condition is affected from a change. For those bindings that fullfil the creation condition new goals are instantiated. The bound variables can be referenced in the creation condition directly via their name attribute.

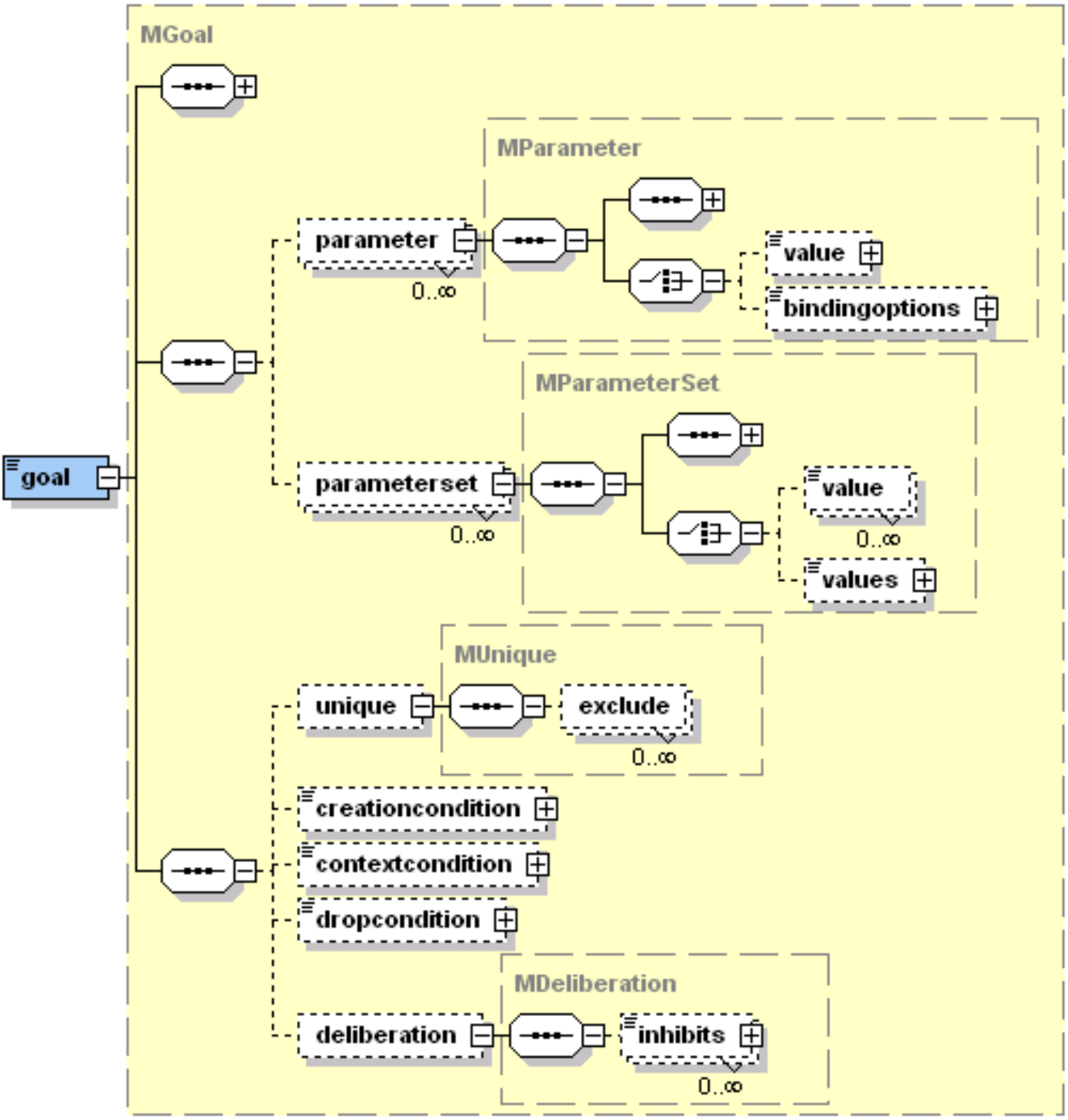

Figure 8.2. The Jadex common goal features

The `<unique>` settings influence if a goal is adopted or ignored. When the unique tag is present, the agent does not adopt two equal instances of the goal at once. By default two goal instances of the same type are equal, when all parameters and parameter sets have the same values. Using the `<exclude>` tag this default behaviour can be overriden by specifying which parameter(set)s should not be considered in the comparison. When a plan tries to adopt a goal that already exists and is declared as unique, a goal failure exception is thrown (see Section 9.2.1, “Plan Success or Failure and BDI Exceptions”.

To describe the situations in which a new goal of the declared user type will be automatically instantiated, the `<creationcondition>` may be used. For adopted goals, it can be specified under which conditions such a goal has to be suspended or dropped using the `<contextcondition>` and `<dropcondition>` respectively. The suspension of a goal means that all currently executing plans for that goal and all subgoals are terminated at once. If the suspension is cancelled, new means for achieving the goal will be inited. On the other hand, when a goal

is dropped it is removed from the agent, and cannot be reactivated.

The `<deliberation>` settings, which influence which of the possible (i.e., not suspended) goals get pursued, will be explained in Section 8.7, “ Goal Deliberation with "Easy Deliberation" ”.

8.1.1. Example Goal

Figure 8.3, “Example goal (taken from Hunter-Prey scenario)” shows an example goal using most of the features described above. It is a simplified example taken from the Hunter-Prey scenario (package `jadex.examples.hunterprey`) from the `BasicBehaviour` capability, common to all prey creatures. The goal is named `eat_food` and has one parameter `$food`, which is assigned from binding options taken from the `food` belief set. It is created whenever there is food (in the `$food` parameter) and the creature is allowed to eat (see creation condition). The goal is `<unique/>` meaning that the creature will not pursue two goals to eat the same food at the same time. Moreover, the `<deliberation>` settings specify that the `eat_food` goal is more important than the `wander_around` goal.

```
<achievegoal name="eat_food">
    <parameter name="$food" class="Food">
        <bindingoptions>$beliefbase.food</bindingoptions>
    </parameter>
    <unique/>
    <creationcondition>
         $beliefbase.eating_allowed
    </creationcondition>
    <deliberation>
        <inhibits ref="wander_around"/>
    </deliberation>
</achievegoal>
```

Figure 8.3. Example goal (taken from Hunter-Prey scenario)

8.1.2. BDI Flags

The handling and the exposed behaviour of goals can be adapted to the requirements of your application using the so called BDI flags as depicted in Table 8.1, “Common goal attributes (BDI flags)”. The flags can be specified as attributes of the different goal tags in the ADF, or individually for each goal instance using the `set...()` methods. The `retry` flag indicates that the goal should be retried or redone, until it is reached, or no more plans are available, which can handle the goal. An optional waiting time (in milliseconds) can be specifed using the `retrydelay`. The `exclude` flag is used in conjunction with `retry` and indicates that, when retrying a goal, only plans should be called that where not already executed for that goal.

Table 8.1. Common goal attributes (BDI flags)

Name	Default	Possible Values
`retry`	`true`	{ `true`, `false`}
`retrydelay`	`0`	positive `long` value
`recur`	`false`	{ `true`, `false`} (for maintain goals only)
`recurdelay`	`0`	positive `long` value (for maintain goals only)
`exclude`	`"when_tried"`	{ `"when_tried"`, `"when_succeeded"`, `"when_failed"`, `"never"`}

Name	Default	Possible Values
posttoall	false	{ true, false}
randomselection	false	{ true, false}
metalevelreasoning	true	{ true, false}

The posttoall flag enables parallel processing of a goal by dispatching the goal to all applicable plans at once. The first plan to reach the goal "wins" and all other plans are terminated. When all plans terminate without achieving the goal, it is regarded as failed. The randomselection flag can be used to choose among applicable plans for a given goal randomly. Using this flag, the order of plan declarations within the ADF becomes unimportant, i.e., random selection is only applied to plans of the same priority and rank.

The metalevelreasoning flag activates the meta-level reasoning for processing of that goal. Meta-level reasoning means, that the selection among the applicable plans for a given goal (or event) is shifted to a meta-level. This is done by the system by creating a meta-level goal which subsequently needs to be handled by a meta-level plan, which actually has to make the decision and return the result. As the description indicates this process could be made recursive to further meta-meta levels if more than one meta-plan is applicable for the meta-goal, but in our experience this is only a theoretical issue without practical relevance. In Jadex the meta-goals and plans need to be explicitly defined within an ADF. From this circumstance the Meta Goal type is derived which will be explained in more detail in Section 8.8, “Meta Goal”.

8.2. Perform Goal

Perform goals are conceived to be used when certain activities have to be done. Below, an example declaration from the cleaner world example is shown. You can see that the perform goal "patrol" refines some BDI flags to achieve the desired behaviour. By allowing the goal to redo activities (retry= "true"), it is assured that the agent does not conclude to knock off after having performed one patrol round, but instead patrols as long as it is night and it does not need to recharge its battery as described in the context condition. Even when the agent only knows one patrol plan, it will reuse this plan and perform the same patrol rounds, because it is not allowed to exclude a plan (exclude="never"). Note that in the example the "&" entity is used to escape the AND character ("&") in XML.

```
<performgoal name="patrol" retry="true" exclude="never">
    <contextcondition>
        !$beliefbase.is_loading &amp;&amp; !$beliefbase.daytime
    </contextcondition>
</performgoal>
```

Figure 8.4. Example perform goal

8.3. Achieve Goal

Achieve goals are used to reach some desired world state. Therfore, they extend the presented common goal features by adding a <targetcondition> and a <failurecondition>. With the target condition it can be specified in what cases a goal can be considered as achieved, whereas the failure condition is useful to describe the opposite. Therefore the failure condition is very similar to the drop condition that can be found in all goal types. In contrast to the drop condition the final state of the achieve goal is guaranteed to be failed when the failure condition triggers. If target and failure condition are not specified, the results of the plan executions are used to

decide if the goal is achieved. In contrast to a perform goal, an achieve goal without target condition is completed when the first plan completes without error, while the perform goal would continue to execute as long as more applicable plans are available. Below another goal specification from the cleaner world example is shown. The `"moveto"` goal tries to bring the agent to a target position as specified in the location parameter. The goal has been reached, when the agent's position is near the target position as described in the target condition.

```
<achievegoal name="moveto">
    <parameter name="location" class="Location"/>
    <targetcondition>
        $beliefbase.my_location.isNear($goal.location)
    </targetcondition>
</achievegoal>
```

Figure 8.5. Example achieve goal

8.4. Query Goal

Query goals can be used to retrieve specified information. From the specification and runtime behaviour's point of view they are very similar to achieve goals with one exception. Query goals exhibit an implicit target condition by requesting all `out` parameters to have a value other than `null` and `out` parameter sets to contain at least one value. Therefore, a query goal automatically succeeds, when all `out` parameter(set)s contain a value. The agent will engage into actions by performing plans only, when the required information is not available. Below, the `"query_wastebin"` example realizes a query goal to find the nearest not full waste bin. It defines an out parameter, which contains a query expression. If one or more not full waste bins are already known by the agent and therefore contained in the wastebins belief set, the result will be set to the nearest waste bin calculated from the agent's current position (as described in the order by clause). Otherwise the agent does not know any not full waste bin and will try to reach the goal by using matching plans.

```
<querygoal name="query_wastebin" exclude="never">
    <parameter name="result" class="Wastebin" direction="out">
        <value evaluationmode="dynamic">
            select one $wastebin from $beliefbase.wastebins
            where !$wastebin.isFull()
            order by $beliefbase.my_location.getDistance($wastebin.getLocation())
        </value>
    </parameter>
</querygoal>
```

Figure 8.6. Example query goal

8.5. Maintain Goal

Maintain goals allow a specific state to be monitored and whenever this state gets violated, the goal has the purpose to reestablish its original maintain state. Hence it adds a mandatory `<maintaincondition>` tag for the specification of the state to observe. Sometimes it is desirable to be able to refine the maintain state for being able to define more accurately what state should be achieved on a violation of the maintained state. Therefore the optional `<targetcondition>` can be declared. Furthermore the maintain goal introduces the recur flag and the recurdelay (in milliseconds) option as further BDI settings. Consider a maintain goal to have failed to reestablish the state to maintain. Setting recur to true, this maintain goal will try to satisfy the maintain condition

again, when the specified delay has elapsed.

Note that maintain goals differ from the other kinds of goals in that they do not necessary lead to actions at once, but start processing automatically on demand. In addition, maintain goals are never finished according to actions or state, so the only possibility to get rid of a maintain goal, is to drop it either by specifying a drop condition or by dropping it from a plan.

The maintain goal `"battery_loaded"` shown below, makes sure that the cleaner agent recharges its battery whenever the charge state drops under 20%. To avoid the agent moving to the charging station and loading only until 21% (which satisfies the maintain condition), the extra target condition is used. It ensures that the agent stays loading until the battery is fully recharged. Note that in the example the `">"` entity is used to escape the greater-than character (">") in XML.

```
<maintaingoal name="battery_loaded">
    <maintaincondition>
        $beliefbase.my_chargestate &gt; 0.2
    </maintaincondition>
    <targetcondition>
        $beliefbase.my_chargestate == 1.0
    </targetcondition>
</maintaingoal>
```

Figure 8.7. Example maintain goal

8.6. Creating and Dispatching New Goals

Jadex distinguishes between top-level goals and subgoals. Subgoals are created in the context of a plan, while top-level goals exist independently from any plans. When a plan terminates or is aborted, all its not yet finished subgoals are dropped automatically. There are four ways to create and dispatch new goals: Goals can be contained in the initial states of an agent or capability, and are directly created and dispatched as top-level goals when an agent is born (cf. Section 14.3, “Goals”). In addition, goals are automatically created and dispatched as top-level goals, when the goal's creation condition triggers. Subgoals may be created inside plans only, while top-level goals may be created manually from plans, as well as from external processes (cf. Chapter 16, *External Processes*).

When a plan wants to dispatch a subgoal or make the agent adopt a new top-level goal it also has to create an instance of some `IMGoal`. For convenience a method `createGoal()` is provided in `jadex.runtime.AbstractPlan` that automatically performs the necessary goal lookup for the model element of the new goal instance. The name therefore specifies the `IMGoal` to use as basis for the new `IGoal`.

A subgoal is dispatched as child of the root goal of the plan, and remains in the goal hierarchy until it is finished or aborted. To start processing of a subgoal, the plan has to dispatch the goal using the `dispatchSubgoal()` method. When the subgoal is finished (e.g., failed or succeeded), an appropriate goal event (type `info`) will be generated, which can be handled by the plan that created the subgoal. A `dispatchSubgoalAndWait()` method is provided in the `Plan` class, which dispatches the goal an waits until the goal is completed. Alternatively to subgoals, the plan can make the agent adopt a new top-level goal by using the `dispatchTopLevelGoal()` method. Further on, a plan may at any time decide to abort one of its subgoals or a top-level goal by using the `drop()` method of the goal. Note, that a goal cannot be dropped when it is already finished.

```
public void body() {
    // Create new top-level goal.
    IGoal goal1 = createGoal("mygoal");
    dispatchTopLevelGoal(goal1);
```

```
    ...
    // Create subgoal and wait for result.
    IGoal goal2 = createGoal("mygoal");
    IGoalEvent ge = dispatchSubgoalAndWait(goal2);
    ...
    // Drop top-level goal.
    goal1.drop();
}
```

Figure 8.8. Dispatching goals from plan bodies

When dispatching and waiting for a subgoal from a standard plan, a goal failure will be indicated by a `GoalFailureException` being thrown. Normally, this exception need not be catched, because most plans depend on all of their subgoals to succeed. If the plan may provide alternatives to failed subgoals, you can use try/catch statements to recover from goal failures (see also Section 9.2.1, “Plan Success or Failure and BDI Exceptions”):

```
public void body() {
    ...
    // Goal failure will cause plan to fail.
    dispatchSubgoalAndWait(goal1);

    // Goal failure will not cause plan to fail.
    try {
        dispatchSubgoalAndWait(goal2);
    }
    catch(GoalFailureException e) {
        // Recover from goal failure.
        ...
    }
}
```

Figure 8.9. Handling of failed subgoals

8.7. Goal Deliberation with "Easy Deliberation"

One aspect of rational behavior is that agents can pursue multiple goals in parallel. Unlike other BDI systems, Jadex provides an architectural framework for deciding how goals interact and how an agent can autonomously decide which goals to pursue. This process is called goal deliberation, and is facilitated by the goal lifecycle (cf. Figure 3.2, “Goal Lifecycle”), which introduces the *active*, *option*, and *suspended* states. The context condition of a goal specifies which goals can possibly be pursued, and which goals have to be suspended. A goal deliberation strategy then has the task to choose among the possible (i.e., not suspended) goals by activating some of them, while leaving the others as options (for later processing).

The current release of Jadex includes a goal deliberation strategy called *Easy Deliberation*, which is designed to allow agent developers to specify the relationships between goals in an easy and intuitive manner. It is based on goal cardinalities, which restrict the number of goals of a given type that may be active at once, and goal inhibitions, which prohibit certain others goal to be pursued in parallel. More details and scientific background about the Easy Deliberation strategy and goal deliberation in general can be found in [Pokahr et al. 2005a].

The goal deliberation settings are included in the goal specification in the ADF Using the `<deliberation>` tag. The cardinality is specified as an integer value in the `cardinality` attribute of the `<deliberation>` tag. The default is to allow an unlimited number of goals of a type to be processed at once. Inhibition arcs between goal

types are specified using the `ref` attribute of the `<inhibits>` tag, which specifies the name of the goal to inhibit. Per default, any instance of the inhibiting goal type inhibits any instance of the referenced goal type. An expression can be included as content of the inhibits tag, in which case the inhibition only takes effect when the expression evaluates to true. Using the expression variables `$goal` and `$ref`, fine-grained instance-level inhibition releationships may be specified. Some goals, such as idle maintain goals, might not alway be in conflict with other goals, therefore it is sometimes required to restrict the inhibition to only take effect when the goal is in process. This can be specified with the `inhibit` attribute of the `<inhibits>` tag, using `"when_active"` (default) or `"when_in_process"` as appropriate. For a better understanding of the goal deliberation mechanism in the following the deliberation settings of the cleanerworld example will be explained.

Figure 8.10, “Example goal dependencies (taken from Cleanerworld scenario)” shows the dependencies between the goals of a cleaner agent (cf. package `jadex.examples.cleanerworld`. The basic idea is that the cleaner agent (being an autonomous robot) has at daytime the task to look for waste in some environment and clean up the located pieces by bringing them to a near waste-bin. At night it should stop cleaning and instead patrol around to guard its environment. Additionally, it always has to monitor its battery state and reload it at a charging station when the energy level drops below some threshold.

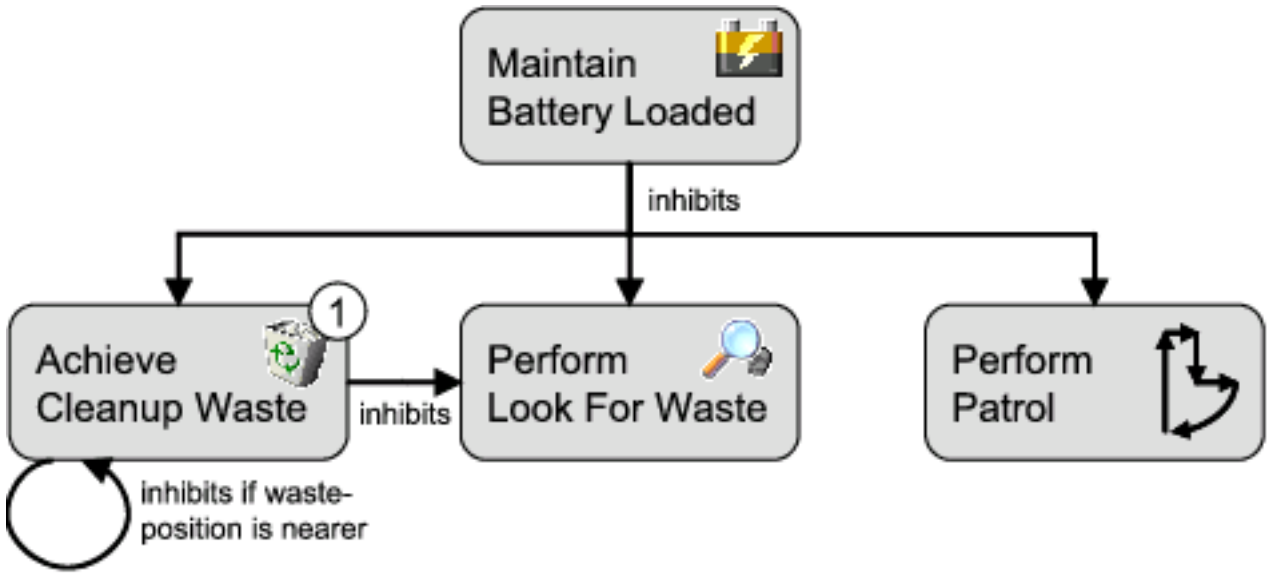

Figure 8.10. Example goal dependencies (taken from Cleanerworld scenario)

The dependencies can be naturally mapped to the goal specifications in the ADF (see Figure 8.11, “Example goals (taken from Cleanerworld scenario)”). `<inhibits>` tags are used to specify that the “maintainbatteryloaded” goal is more important than the other goals. As the “maintainbatteryloaded” is a maintain goal, it only needs to precede the other goals when it is in process, i.e., the cleaner is currently recharging its battery. The cardinality of the “achievecleanup” goal specifies, that the agent should only pursue one cleanup goal at the same time. The goal inhibits the “performlookforwaste” goal and additionally introduces a runtime inhibition relationship to other goals of its type. The expression contained in the inhibits declaration means that one “achievecleanup” goal should inhibit other instances of the “achievecleanup” goal, when its waste location is nearer to the agent.

```
<maintaingoal name="maintainbatteryloaded">
    <!-- Omitted conditions for brevity. -->
    <deliberation>
        <inhibits ref="performlookforwaste" inhibit="when_in_process"/>
        <inhibits ref="achievecleanup" inhibit="when_in_process"/>
        <inhibits ref="performpatrol" inhibit="when_in_process"/>
    </deliberation>
</maintaingoal>
```

```
<achievegoal name="achievecleanup" retry="true" exclude="when_failed">
    <parameter name="waste" class="Waste" />
    <!-- Omitted conditions for brevity. -->
    <deliberation cardinality="1">
        <inhibits ref="performlookforwaste"/>
        <inhibits ref="achievecleanup">
            $beliefbase.my_location.getDistance($goal.waste.getLocation())
            &amp;lt; $beliefbase.my_location.getDistance($ref.waste.getLocation())
        </inhibits>
    </deliberation>
</achievegoal>
```

Figure 8.11. Example goals (taken from Cleanerworld scenario)

8.8. Meta Goal

Meta Goals are used for meta-level reasoning. This means, whenever an event or goal is executed and it is determined that meta-level resoning needs to be done (i.e., because there are multiple matching plans) the corresponding meta-level goal of the goal or event is created and dispatched. Corresponding meta-level plans are then executed to achieve the meta goal (i.e., find a plan to execute). When the meta goal is finished the result contains the selected plans, which are afterwards scheduled for execution.

The specification of a meta goal requires the declaration of a triggering goal or event and has at least to include the `in` parameter set `"applicables"` and the `out` parameter set `"result"` (both of type `jadex.runtime.ICandidateInfo`). The applicables are filled in by the system, while the result is set by the meta-level plan executed to achieve the meta goal. Furthermore, a failure condition can specified (similar to query goals) as meta goals are also used for information retrieval (to find a plan to execute for a goal resp. event). Meta-goals are only created internally by the system when the demand for meta-level reasoning arises. Therefore, in contrast to the query goal and the other goal types presented here, meta-goals exhibit several restrictions, as for these kinds of goals creation condition, unique settings and binding parameters are not allowed. On the other hand, meta-plans do not differ from other plans (there is no a separate tag for meta plans). A plan is a meta plan, when its plan trigger contains a meta goal.

In the example below, adapted from the `jadex.examples.puzzle` example, for every “makemove” goal a large number of plan instances might be applicable, as the “move_plan” has a binding option which always contains all possible moves. Therefore, the “choosemove” meta goal is used to decide which of the applicable “move_plan” instances should be executed. In turn, handling the “choosemove” meta goal another plan is executed (“choose_move_plan”). As you can see in Figure 8.13, “Body of the ChooseMovePlan”, the “choose_move_plan” has access to the parameters of applicable plans and may use this informations to decide which plan(s) to execute. The selected plans are placed in the “result” parameter of the “choose_move_plan” goal.

```
<goals>
    <achievegoal name="makemove">
        ...
     </achievegoal>

    <metagoal name="choosemove">
        <parameter name="applicables" class="ICandidateInfo"/>
        <parameter name="result" class="ICandidateInfo" direction="out"/>
        <trigger>
            <goal ref="makemove"/>
        </trigger>
    </metagoal>
</goals>
```

```
<plans>
    <plan name="move_plan">
        <parameter name="move" class="Move">
            <bindingoptions>$beliefbase.board.getPossibleMoves()</bindingoptions>
        </parameter>
        ...
        <trigger>
            <goal ref="makemove"/>
        </trigger>
    </plan>

    <plan name="choose_move_plan">
        <parameterset name="applicables" class="ICandidateInfo">
            <goalmapping ref="choosemove.applicables"/>
        </parameterset>
        <parameterset name="result" class="ICandidateInfo" direction="out">
            <goalmapping ref="choosemove.result"/>
        </parameterset>
        <body>new ChooseMovePlan()</body>
        <trigger>
            <goal ref="choosemove"/>
        </trigger>
    </plan>
</plans>
```

Figure 8.12. Example meta goal and corresponding plan

```
public void body()
{
    ICandidateInfo[] apps = (ICandidateInfo[])getParameterSet("applicables").getValues();

    ICandidateInfo sel = null;

    for(int i=0; i<apps.length; i++)
    {
        // Decide which plan to select, e.g. using the move parameter of the move_plan.
        Move move = (Move)apps[i].getPlan(this).getParameter("move").getValue();
        ...
    }

    getParameterSet("result").addValue(sel);
}
```

Figure 8.13. Body of the ChooseMovePlan

Chapter 9. Plans

Plans represent the agent's means to act in its environment. Therefore, the plans predefined by the developer compose the library of (more or less complex) actions the agent can perform. Depending on the current situation, plans are selected in response to occuring events or goals. The selection of plans is done automatically by the system and represents one main aspect of a BDI infrastructure. In Jadex, plans consist of two parts: A plan head and a corresponding plan body. The plan head is declared the the ADF whereas the plan body is realized in a concrete Java class. Therfore the plan head defines the circumstances under which the plan body is instantiated and executed.

9.1. Defining Plan Heads in the ADF

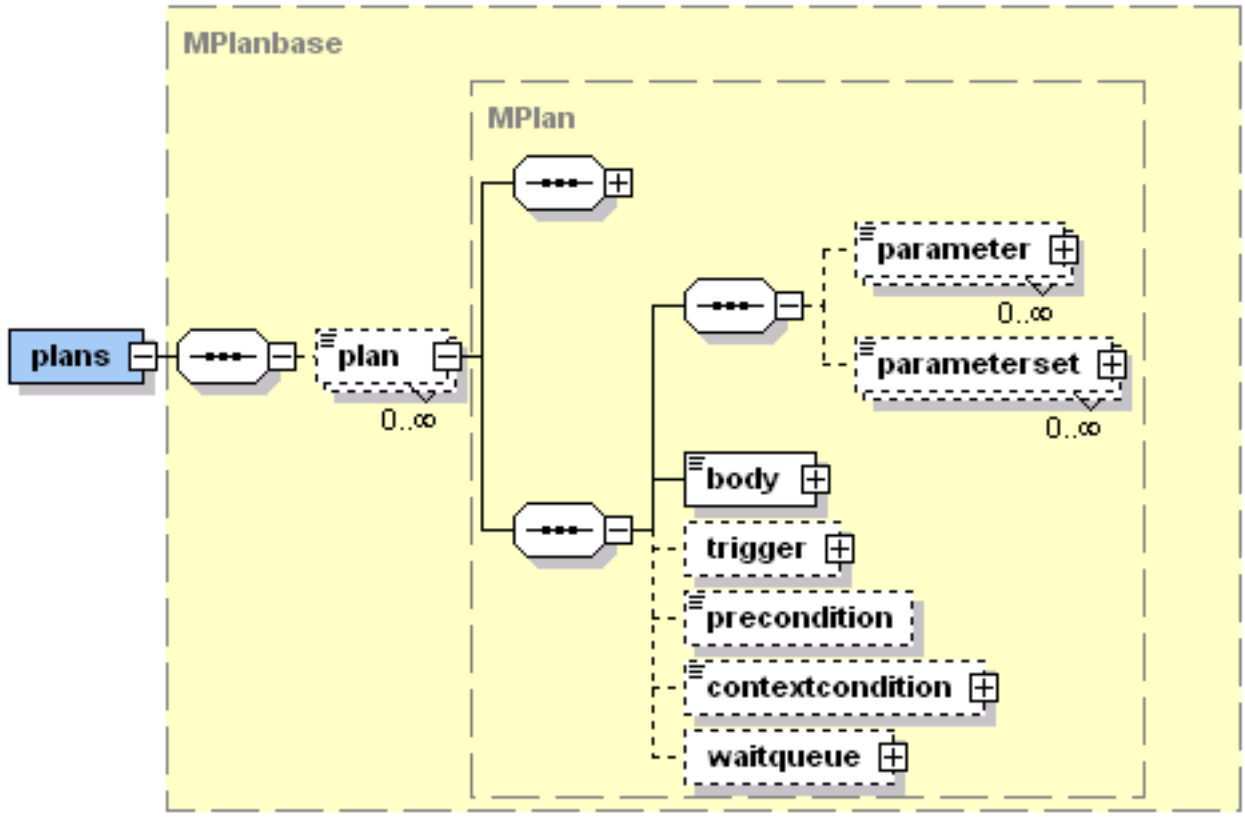

Figure 9.1. The Jadex plans XML schema part

In Figure 9.1, “The Jadex plans XML schema part” the XML schema part for the plans section is shown. Inside the `<plans>` tag an arbitrary number of plan heads denoted by the `<plan>` tag can be declared. For each plan head several attributes (as shown in Table Table 9.1, “Important attributes of the plan and the body tag”) and contained elements can be defined. For each plan a name has to be provided. The priority of a plan describes its preference in comparison to other plans. Therefore it is used to determine which candidate plan will be chosen for a certain event occurence, favouring higher priority plans (random selection, if activated, applies only to plans of equal priority). Per default all applicable plans have a default priority of 0 and are selected in order of appearance (or randomly when the corresponding BDI flag is set).

Table 9.1. Important attributes of the plan and the body tag

Tag	Attribute	Required	Default	Possible Values
plan	name	yes		
plan	priority	no	0	any positive or negative integer
body	type	no	standard	{standard, mobile}

For each plan the corresponding plan body has to be declared using the `<body>` element. Within this element a Java expression has to be provided for the creation of the plan body (in most cases a simple constructor call like `new PingPlan()` is used). The type attribute determines which kind of plan body is used. Currently, the options are standard vs. mobile plan bodies as further described in Section 9.2, “Implementing a Plan Body in Java”. To clarify things, a simple example ADF is given below that shows the declaration of a plan reacting on a ping message.

```
<agent ...>
    ...
    <plans>
        <plan name="ping">
            <body>new PingPlan()</body>
            <trigger>
                <messageevent ref="query_ping"/>
            </trigger>
        </plan>
    </plans>
    ...
    <events>
        <messageevent name="query_ping" type="fipa">
            ...
        </messageevent>
    </events>
    ...
</agent>
```

Figure 9.2. A plan reacting on a ping message

9.1.1. Plan Triggers

To indicate in what cases a plan is applicable and a new plan instance shall be created the `<trigger>` tag can be used (see Figure 9.3, “The Jadex plan trigger XML schema part”). Its subtags specify the internal-, message, or goal events for which the plan is applicable. These events or goals can be further restricted, by requiring certain parameter values. When parameter specifications are included in the trigger tag, the plan will only be selected for those goals or events matching all parameter specifications. Restrictions of parameter set values are currently not supported. For backwards compatibility to older Jadex versions, additionally, a filter instance can be used, although its use is discouraged, because of the lack of declarativeness and readability.

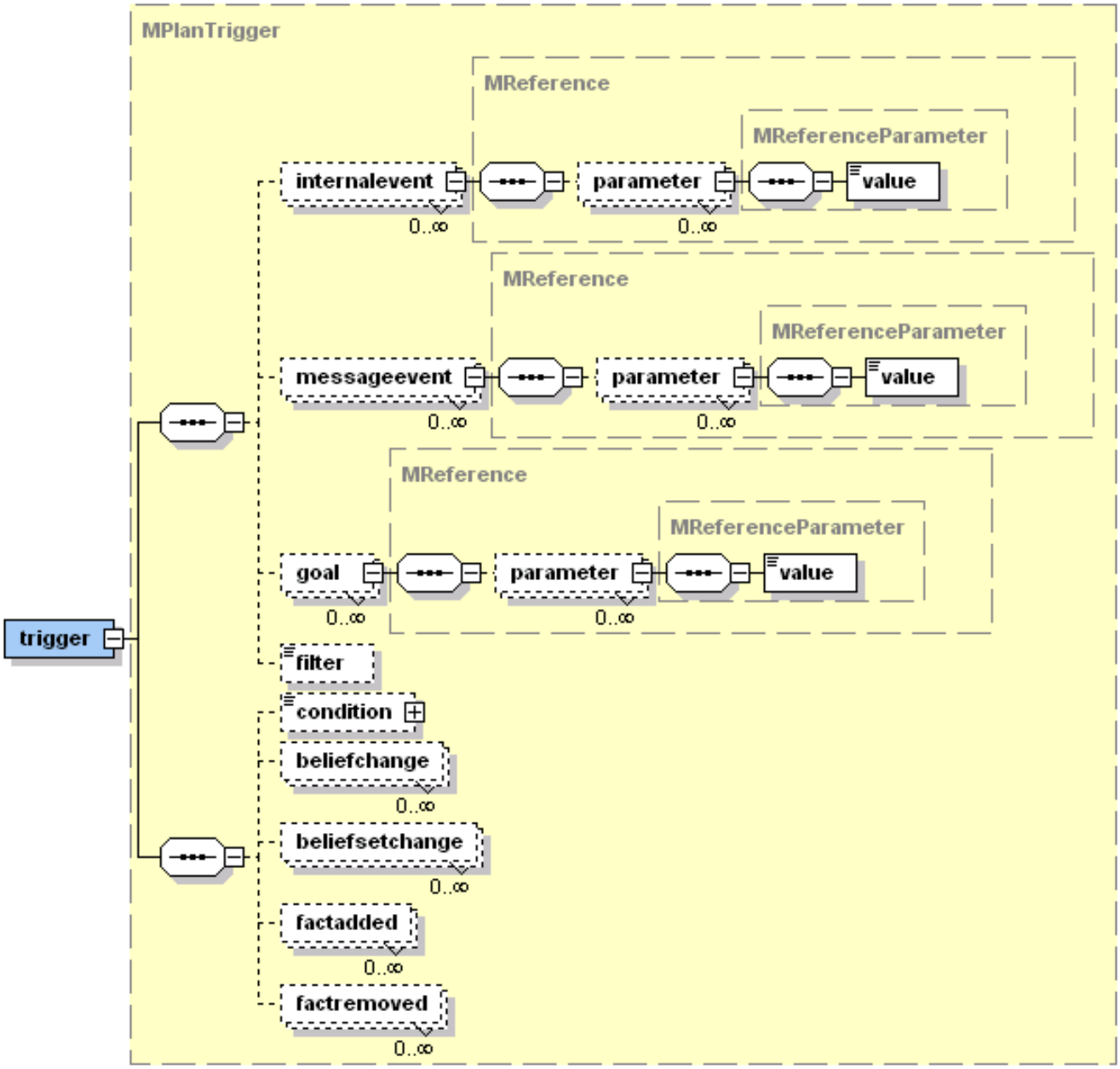

Figure 9.3. The Jadex plan trigger XML schema part

In addition to the reaction on certain event or goal types, it is also possible to define data-driven plan execution by using the `<condition>` tag. A trigger condition can consist of arbitrary boolean Jadex expressions, which may refer to certain beliefs when their states needs to be supervised. If only some specific belief needs to be monitored the `<beliefchange>` tag can be used. In this respect a belief change is reported whenever the belief's new fact value is different from the value held before. Similarly, belief sets can be monitored with the `<beliefsetchange>` tag, or more specifically for addition or removal of facts by using the tags `<factadded>` and `<factremoved>` respectively.

9.1.2. Defining Plan Applicability with Pre- and Context Conditions

To find out if the plan is applicable not only with respect to the current event or belief change but also considering the current situation, the pre- and context conditions can be used. The precondition is evaluated before a plan is instantiated and when it is not fulfilled this plan is excluded from the list of applicable plans. In contrast, the context condition is evaluated during the execution of a plan and whenever it is violated the plan execution

is aborted and the plan has failed. Both conditions can be specified in the corresponding tags supplying some boolean Jadex expression. The following example shows how to execute a "repair" plan whenever the belief `"out_of_order"` becomes `true`, and as long as the agent believes to be repairable.

```
<plans>
    <plan name="repair">
        <body> new RepairPlan() </body>
        <trigger>
            <condition> $beliefbase.out_of_order </condition>
        </trigger>
        <contextcondition> $beliefbase.repairable </contextcondition>
    </plan>
</plans>
```

Figure 9.4. Example of a plan with context condition

9.1.3. Waitqueue

When an event occurs, and triggers an execution step of a plan, it may take a while, before the plan step is actually executed, due to many plans being executed concurrently inside an agent. Therefore, it is sometimes possible, that a subsequent event, which might be relevant for a plan, is not dispatched to that plan, because it is still executing a previous plan step, and does not yet wait for the event. To avoid this, each plan has a waitqueue to collect such events. The waitqueue for a plan is set up using the `<waitqueue>` tag or the `getWaitqueue()` method in plan bodies. The waitqueue of a plan is always matched against events, although the plan may not currently wait for that specific event. The `<waitqueue>` tag provides the same event and goal options as the `<trigger>` tag described above, but does not support the `<condition>` and the belief(set) change tags. Events that match against the waitqueue of a plan are added to the plans internal waitqueue. They will be dispatched to the plan later, when it calls `waitFor()` or `getWaitqueue().getEvents()` with optionally a matching filter that restricts the returned events. You may have a look at the `jadex.runtime.IWaitqueue` interface for more details.

9.1.4. Parameters, Binding, and Parameter Mapping

Similar to goals, plans may have parameters and parameter sets, which can store local values, required or produced during the execution of the plan. Plan parameters can be accessed from plan bodies for read and write access depending on the parameter direction attribute: `in` parameters (cf. Section 8.1, “Common Goal Features”) allow only read access, `out` parameters can only be written, while `inout` parameters allow both kinds of access. Default values for any of these parameters and parameter sets can be provided in the ADF. Just like facts for belief sets, initial values for parameter sets can be either specified as a sequence of `<value>` tags, or as a single `<values>` tag. The parameter(set)s of a plan can also be accessed from the body tag or the context condition, by referencing the plan via the reserved variable `$plan` concatenated with the parameter(set) name, e.g. `$plan.result`. The precondition and the trigger condition are evaluated before the plan is instantiated, therefore from these conditions no parameters and parameter sets can be accessed.

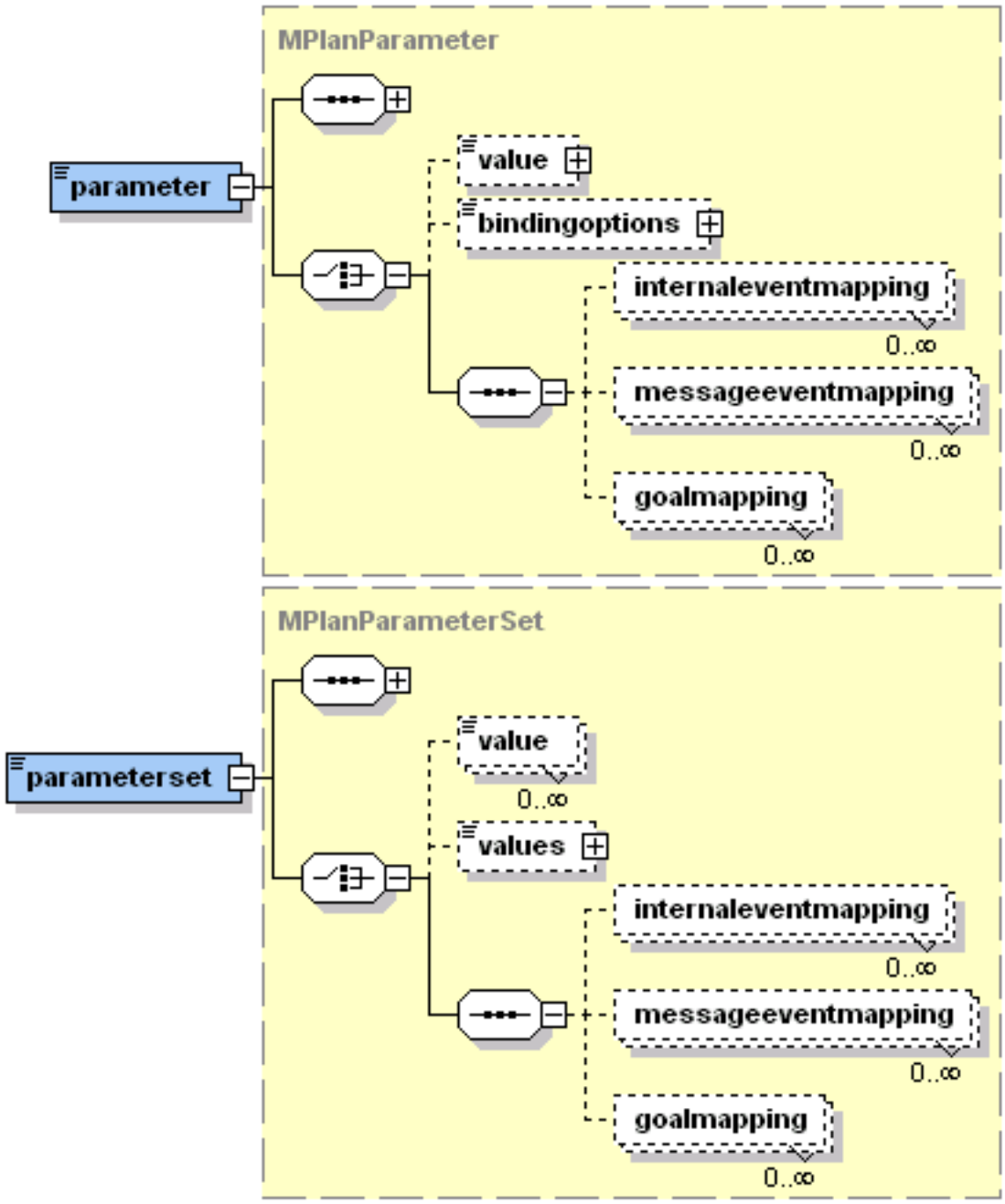

Figure 9.5. The Jadex plan parameters XML schema part

For (single valued) parameters it is possible to use binding options instead of an initial value. A binding option is an expression, that will be evaluated to a collection of values (supported are arrays or an object implementing `Iterator`, `Enumeration`, `Collection`, or `Map`). The binding options of a parameter therefore represent a set of possible initial values for that parameter. The cartesian product [1] of all binding parameters (if there is more than one parameter with binding otpions) determines the number of candidate plans that is considered in the event dispatching process. Please note that the calculation of the cartesian product can easily lead to large numbers of applicable plans so that binding options should always be used with care. For example Figure 9.6, “ Example binding parameter (from the puzzle example) Example binding parameter ” shows a plan from the “puzzle” example, where for each possible move a plan instance is created. In addition to accessing the binding values like other parameters by writing `$plan.paramname`, it is also possible to access the binding value directly via its

[1] In mathematics, the Cartesian product (or direct product) of two sets X and Y, denoted X x Y, is the set of all possible ordered pairs whose first component is a member of X and whose second component is a member of Y. Example: The cartesian product of {1,2}x{3,4} is {(1,3),(1,4),(2,3),(2,4)}. (cf. Wikipedia)

name via paramname. This allows binding values also to be considered for evaluating the pre- and trigger condition, before the plan instance is created.

```
<plan name="move_plan">
    <parameter name="move" class="Move">
        <bindingoptions>$beliefbase.board.getPossibleMoves()</bindingoptions>
    </parameter>
    ...
</plan>
```

Figure 9.6. Example binding parameter (from the puzzle example) Example binding parameter

A common use case for plan parameter(set)s is to capture parameter(set)s from a goal or event that triggered the plan. To make this relationship between event and plan parameters explicit, the <internaleventmapping>, <messageeventmapping>, and <goalmapping> tags can be used. A mapping definition contains a ref attribute denoting the event or goal parameter to be mapped. The reference is given in the form type.param, where type is the name of the goal or event, and param is the name of the goal or event parameter. When a plan parameter is mapped, the parameter properties like class and direction are ignored, as the values from the mapped parameter are used. Depending on the direction of the parameter, the default values of the plan parameter are automatically assigned from the event or goal (direction in, inout), and/or written back to the goal or event (direction out, inout), when the plan has finished. Note that when a plan reacts to more than one goal or event, you cannot just provide a mapping for one of these events. If you want to use a mapping for a parameter, you have to provide mappings for all events or goals handled by the plan.

9.2. Implementing a Plan Body in Java

A plan body represents a part of the agent's functionality and encapsulates a recipe of actions. In Jadex, plan bodies are written in pure Java and therfore it is easily possible to write plans that access any available Java libraries, and to develop plans in your favourite Java Integrated Development Environment (IDE). The connection between a plan body and a plan head is established in the plan head, thus plan bodies can be reused within different plan declarations. For developing reusable plans, plan parameters in combination with parameter mappings from some triggering event or goal to/from the plan should be used.

As mentioned earlier, currently two types of plan bodies are supported in Jadex, which are both implemented as conventional Java classes. The standard plans inherit from jadex.runtime.Plan. The mobile plans inherit from jadex.runtime.MobilePlan and allow agents to be migrating, even while plans are executing (e.g., supported by the JADE platform). The code of standard plans is placed in the body() method, while the code of mobile plans goes into the action(IEvent) method.

Plans that are ready to run are executed by the main interpreter (cf. Section 3.3, “Execution Model of a Jadex Agent”). The system takes care that only one plan step is running at a time. The length of a plan step depends on the plan itself. For mobile plans the action() method is always executed as a whole, for the first and again for all subsequent steps. The body() method of standard plans is called only once for the first step, and runs until the plan explicitly ends its step by calling one of the waitFor() methods, or the execution of the plan triggers a condition (e.g., by changing belief values). For subsequent steps the body() method is continued, where the plan was interrupted.

The API of both plan types is very similar (both inherit from the same super class jadex.runtime.AbstractPlan), the only difference regards the waiting for events. Both plans provide several variations of the waitFor...() method, but only the standard plan will block when it is called. The different execution style is shown in a code example implementing the initiator side of a FIPA-request protocol. Remember, the body() method of a standard plan is called only once, while the action() method of a mobile plan is

called for each event. The standard plan can use nested `if-then-else` blocks to naturally handle all cases of the protocol, while the mobile plan has no state information of previous messages, and has to handle all events at the top level of the `action()` method in one large `if-then-else` statement.

Standard Plan

```
public void body() {
    boolean agreed, informed;

    // Send request.
    ...

    // Wait for agree/refuse.
    IEvent e1 = waitFor(...);
    ...

    // Wait for inform/failure.
    if(agreed) {
        IEvent e2 = waitFor(...);
        ...
        if(informed) {
            ...
        }
        else {
            ...
        }
    }
    else {
        ...
    }
}
```

Mobile Plan

```
public void action(IEvent e) {
    boolean agreed, refused;
    boolean informed, failed;
    ...
    if(initial_event) {
        // Send request.
        ...
        // Wait for agree/refuse.
        waitFor(...);
    }
    else if(agreed) {
        ...
        // Wait for inform/failure.
        waitFor(...);
    }
    else if(refused) {
        ...
    }
    else if(informed) {
        ...
    }
    else if(failed) {
        ...
    }
}
```

Figure 9.7. Programming style of the different plan types

9.2.1. Plan Success or Failure and BDI Exceptions

If a plan completes without producing an exception it is considered as succeeded. Completion means for standard plans that the `body()` method returns. For mobile plans it means that the `action()` method returns and the plan does not wait for any more events. To perform cleanup after the plan has finished, you can override the `passed()`, `failed()`, and `aborted()` methods, which are called when the plan succeeds (runs through without exception), fails (e.g., due to an exception), or was aborted during execution (e.g., because the root goal was dropped or has been achieved before the plan reached its end). In Figure 9.8, “Standard plan skeleton” a plan skeleton of a standard Jadex plan is depicted including all predefined methods. The cleanup methods are also available in mobile plans. In the `failed()` method, a plan may call the `getException()` method to see which problem occured. To find out whether the plan was aborted, because its root goal was achieved, you can call the `isAbortedOnSuccess()` method inside the `aborted()` method.

```
public class MyPlan extends Plan {

    public void body() {
        // Application code goes here.
        ...
    }

    public void passed() {
        // Clean-up code for plan success.
        ...
```

```
    }

    public void failed() {
        // Clean-up code for plan failure.
        ...
        getException().printStackTrace();
    }

    public void aborted() {
        // Clean-up code for an aborted plan.
        ...
        System.out.println("Goal achieved? "+isAbortedOnSuccess());
    }
}
```

Figure 9.8. Standard plan skeleton

Regardless if standard or mobile plans are used, a plan is considered as failed if it produces an exception. To aid debugging, occurring exceptions are (by default) printed on the console (logging level `SEVERE`), although the agent continues to execute. Subclasses of `jadex.runtime.BDIFailureException` are not printed, because they are produced by the system and indicate "normal" plan failure. If you want your plan explicitly to fail without printing an exception, you can throw a `PlanFailureException` or, as a shortcut, call the `fail()` method. Other subclasses of the `BDIFailureException` are generated automatically by the system, to denote certain failures during plan execution. All of these exceptions can be explicitly handled if desired, or just ignored (causing the plan to fail). The `GoalFailureException`, already introduced in Section 8.6, “Creating and Dispatching New Goals”, is thrown, when a subgoal of a plan could not be reached or if the subgoal could not be adopted due to its uniqueness settings (i.e. there exists already a goal that is considered equal to the new one). The `MessageFailureException` indicates that a message could not be sent, e.g., because the receiver is unknown. A `TimeoutException` occurs when calling `waitFor()` with a timout, and the awaited event does not happen. Finally, the `AgentDeathException` is thrown when an operation could not be performed, because the agent has died. This usually does not occur inside plans, but only when accessing an agent from external processes (see Chapter 16, *External Processes*).

9.2.2. Atomic Blocks

For mobile plans, each call to the `action()` method is executed as an atomic block, i.e., the agent will not do other things until the `action()` method returns. Standard plans on the other hand, might be interrupted whenever the agent regards it as necessary, e.g., when a belief has been changed leading to the adoption of a new goal. Sometimes it is desireable that a sequence of actions is considered as a single atomic action. For example when you change multiple beliefs at once, which might trigger some conditions, you may want to perform all changes before the conditions are evaluated. In standard plans, this can be achieved by using a pair of `startAtomic()` / `endAtomic()` calls around the code you want to execute as a whole. Note that you are not allowed to end the plan step inside an atomic block (e.g., by calling `waitFor()`).

```
public void body() {
    ...
    startAtomic();
    // Atomic code goes here.
    ...
    endAtomic();
    ...
}
```

Figure 9.9. How to establish an atomic block

Chapter 10. Events

Jadex is an event-based system, that is, nothing happens inside a Jadex agent unless it is triggered by some event. For example the ping plan gets triggered when a ping request message arrives. Nevertheless, Jadex agents are not purely reactive, because Jadex not only supports external events (e.g., messages), but also different types of internal events. For example the adoption of a new goal will generate a goal event, leading to plans being executed to achieve the goal. When an event occurs in Jadex and no plan is found to handle this event a warning message is generated, which can be printed to the console depending on the logging settings (see Chapter 13, *Properties*).

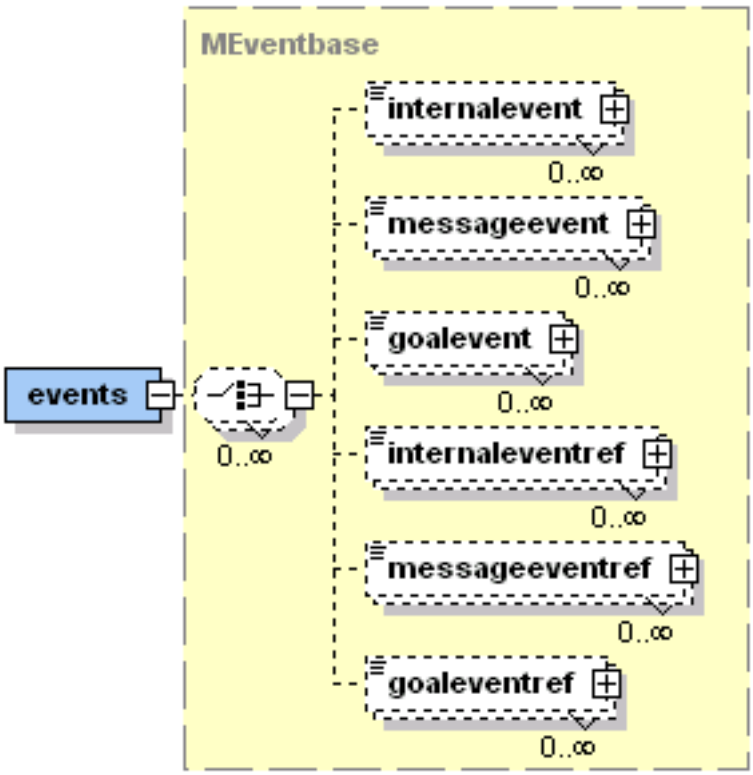

Figure 10.1. The Jadex events XML schema part

Three types of events are supported in Jadex: Message events, goal events, and internal events, as well as references to these types, as shown in Figure 10.1, “The Jadex events XML schema part”. Generally, all event types are parameter elements meaning that any number of parameter(set)s can be specified for a user-defined kind of event. A parameter itself can be used for passing values from the source to the consumer of the event and possibly back. For making explicit in which direction the data flow should occur the parameter's direction attribute can be used. The possible values are "in", "out" or "inout" similar to parameters of procedure calls known from conventional programming languages. In addition all kinds of events share the common attributes: "posttoall", "metalevelreasoning" and "randomselection". The "posttoall" flag determines if the event should be dispatched to a single receiver or to all applicable plans (note, that for post-to-all goals no retry is performed). The "metalevelreasoning" flag can be used to turn off the process of reasoning about plan candidates if more than one candidate is applicable for a given event. Finally, the "randomselection" flag can be used to turn off the importance of the declaration order of plans for the plan selection process. Nevertheless, the priority of a plan is still respected.

Table 10.1. Event Flags

Flags	Default Value
posttoall	internal event=true, otherwise false
metalevelreasoning	true
randomselection	false

At runtime, e.g., when accessed from plans, instances of the elements are represented by the `jadex.runtime.IMessageEvent`, `jadex.runtime.IGoalEvent`, and `jadex.runtime.IInternalEvent` interfaces. The following sections will describe these different types of events in more detail.

10.1. Goal Events

Goal events are used only internally for the processing of goals. This means that the agent will create a goal event in response to an active goal it wants to perform (process event) and for a goal that finished its processing (info event). *A user never has to define custom goal events, but instead can concentrate on goal modelling.* To explicitly decide which kind of event has happened (as commonly necessary inside mobile plans) the `IGoalEvent.isInfo()` method can be used.

10.2. Internal Events

Internal events are the typical way in Jadex for explicit one-way communication of an interesting occurrence inside the agent. The usage of internal events is characterized by the fact that an information should be passed from a source to some consumers (similar to the object oriented observer pattern). Hence, if an internal event occurs within the system, e.g., because some plan dispatches one, it is distributed to all plans that have declared their interest in this kind of event by using a corresponding trigger or by waiting for this kind of internal event. The internal event can transport arbitrary information to the consumers if custom parameter(set)s are defined in the type for that purpose. A typical use case for resorting to internal events is, e.g., updating a GUI.

```
<!-- ADF snippet showing the internal event declaration. -->
...
<events>
        <!-- Specifies an internal event for updating the gui.-->
        <internalevent name="gui_update">
            <parameter name="content" class="String"/>
        </internalevent>
</events>
...
```

```
// Plan snippet showing the creation and dispatching of the internal event.
...
public void body() {
    String update_info;
    ...
    // "gui_update" internal event type must be defined in the ADF
    IInternalEvent event = createInternalEvent("gui_update");
    // Setting the content parameter to the update info
    event.getParameter("content").setValue(update_info);
    dispatchInternalEvent(event);
    ...
}
```

Figure 10.2. Dispatching an internal event example

In addition to user-defined internal events there are also some already predefined internal events used by the Jadex system itself. In Table 10.2, “Predefined Internal Event Types” these types are explained shortly. They should not be of much importance for most agent developers. Only, if you intend to write mobile plans you may need to know them.

Table 10.2. Predefined Internal Event Types

Predefined Types	Description
jadex.model.IMEventbase.TYPE_TIMEOUT	A timeout has occurred (e.g., while waiting for some other event)
jadex.model.IMEventbase.TYPE_EXECUTEPLAN	A plan should be executed (e.g., initial or conditional plan)
jadex.model.IMEventbase.TYPE_CONDITION_TRIGGERED	A condition has been triggered

10.3. Message Events

All message types an agent wants to send or receive need to be specified within the ADF. The message event (class `jadex.runtime.IMessageEvent`) denotes the arrival or sending of a message. The direction of the message (arrived or to be sent) can be checked with the `isIncoming()` method. Note, that only incoming messages are handled by the event dispatching mechanism, while outgoing messages are just sent. The native underlying message object (which is platform dependent) can be retrieved using the `getMessage()` method. In addition, the message content, which may be a String or some content object, can be retrieved using the `getContent()` method.

Templates for message events are defined in the ADF in the `<events>` section. The direction attribute can be used to declare whether the agent wants to receive, send or do both (default) for a given event. Possible values for that attribute are "send", "receive" and "send_receive" respectively. The type of the message constrains the available parameters of a message. Currently, the only available type is "fipa" which automatically creates parameter(set)s according to the FIPA message specification (e.g., parameters for the receivers, content, sender, etc. are introduced). Through this message typing Jadex does not require that only FIPA messages are being sent, as other options may be added in future. In the following Table 10.3, “Reserved FIPA message event parameters”, all available parameter(set)s are itemized. For details about the meaning of the FIPA parameters, see the FIPA specifications available at http://www.fipa.org. In addition to the FIPA parameters, Jadex introduces the `content-start`, `content-class`, and `action-class` parameters. These are explained below.

Table 10.3. Reserved FIPA message event parameters

Name	Class
`performative`	`String`
`sender`	`jadex.adapter.fipa.AgentIdentifier`
`reply-to`	`jadex.adapter.fipa.AgentIdentifier`
`content`	`Object`
`language`	`String`
`encoding`	`String`

Name	Class
ontology	String
protocol	String
reply-with	String
in-reply-to	String
conversation-id	String
reply-by	java.util.Date
receivers	jadex.runtime.adapter.AgentIdentifier
content-start	String
content-class	Class
action-class	Class

10.3.1. Receiving Messages

Typically in the ADF of an agent a number of message event types for sending and receiving message events are declared for the application domain. Examples for such user-defined message event types might be "inform_time", "request_vision", etc. As those message types are defined for each agent separately there are consequently no global message types. So how does an agent know the message type of a newly received message? For this purpose a simple matching process is used. This means that all locally known message types of an agent and its subcapabilities (with direction "receive" or "send_receive") are matched against the newly received message and the best fitting is selected. For the matching process the parameter values of a message type are checked against the values in the received message. For this purpose only parameters with direction="fixed" are considered important, as they represent fixed type information. *A message event type matches an incoming message if all fixed parameter values are the same in the received message.*

There are several reasons why an agent may fail to correctly process an incoming message. These are indicated by different logging outputs at different logging levels (see Table 10.4, “Possible problems when matching messages”). In the first case, if more than one message event type has a match with the incoming event the most specific match will be used. The number of fixed parameters is used as indicator for the specificity. As this is a common case, it is only logged at level `INFO`. When a message is received, which does not match any of the declared message events of the agent, a `WARNING` is generated, indicating that this message is ignored by the agent. Finally, when there are two or more message events, which all match an incoming message to the same degree (i.e., all have the same number of fixed parameters) the system cannot decide which message event to use, and has to choose one arbitrarily. As this probably indicates a programming error in the ADF, a `SEVERE` ouput is produced.

Table 10.4. Possible problems when matching messages

Level	Output
INFO	Multiple events matching message, using message event with highest specialization degree
WARNING	...cannot process message, no message event matches
SEVERE	Cannot decide which event matches message, using first

The content-start, content-class, and action-class parameters (if present) are treated specially in the matching process. The content-start parameter matches to all messages with a content starting with the given string value. The content-class parameter is matched against the class of the object sent as message content (see below). Finally, the action-class parameter is useful for messages encoded in FIPA SL. The parameter is matched against the action expression contained in the message.

10.3.2. Sending Messages

The super class of both plan types (jadex.runtime.AbstractPlan) provides several convenience methods to create message events. To send a message, a message event has to be created using the createMessageEvent(Strint type) method supplying the declared message event type name as parameter. The receivers of fipa messages are specified by agent identifiers (class jadex.adapter.fipa.AgentIdentifier). The message content can be supplied as String or as Object with setContent(Object content). If the content is provided as Object it must be ensured that the agent can encode it into a transmissable representation. Currently the only language supported for all adapters is the jadex.runtime.adapter.SFipa.JAVA_XML language which employs the JDK built-in Java bean mechanism to encode and decode the content. Using this language requires that Java bean information about the content object can be found or inferred by the Java bean introspector. Please have a look at the Beanynizer tool (available from the Jadex homepage) if you are interested in converting an ontology to Java beans including the necessary bean infos.

Note

If you are using the Jadex adapter for the JADE platform you can rely on the JADE specific codecs for content transmission. In this case the content object must be instance of a jade.content.ContentElement, which will be automatically filled into the message using JADEs content manager mechanism. To use this mechanism you have to ensure several things.

- The language and ontology need to be known by the agent. Therefore the following predefined properties "jade.language.xxx" and "jade.ontology.yyy" can be used to supply the JADE codec and the ontology instance (replacing "xxx"/"yyy" with a suitable name of your choice).
- The language and ontology names need to be set in the message, so that JADE can determine at runtime which codec to use for the given message event.

To actually send the message event it is sufficient to call the sendMessage(IMessageEvent me) method with the prepared message event as parameter. It is also possible to send a message and directly wait for a reply with an optional timeout by using the sendMessageAndWait(IMessageEvent me [, timeout]) method. In this case, make sure you have supplied a conversation-id or a reply-with parameter, because otherwise the reply cannot be related to the original message. The reply-with resp. conversation-id parameter need to be filled with an appropriate unique id. For this purpose you may consider using the helper method SFipa.createUniqueId() as shown in Figure 10.3, “Sending a message example”. When using a timout and the message is not received before the timeout has elapsed, a jadex.runtime.TimeoutException is thrown (see also Section 9.2.1, “Plan Success or Failure and BDI Exceptions”).

```
<!-- ADF snippet showing the message declaration. -->
...
<events>
    <messageevent name="request_carry" type="fipa" direction="send">
        <parameter name="performative" class="String" direction="fixed">
            <value>SFipa.REQUEST</value>
        </parameter>
        <parameter name="language" class="String" direction="fixed">
```

```
            <value>SFipa.JAVA_XML</value>
        </parameter>
        <parameter name="ontology" class="String" direction="fixed">
            <value>MarsOntology.ONTOLOGY_NAME</value>
        </parameter>
        <parameter name="reply-with" class="String">
            <value>SFipa.createUniqueId($scope.getAgentName())</value>
        </parameter>
    </messageevent>
    ...
</events>
...
```

```
// Plan snippet showing the creation and sending of the message.
public void body() {
    ...
    jadex.adapter.fipa.AgentIdentifier receiver;
    CarryRequest action;  // Onotology Java bean.
    ...
    // "request_carry" message event type must be defined in the ADF
    IMessageEvent me = createMessageEvent("request_carry");
    me.getParameterSet(jadex.adapter.fipa.SFipa.RECEIVERS).addValue(receiver);
    me.setContent(action);
    IMessageEvent reply = sendMessageAndWait(event);
    ...
}
```

Figure 10.3. Sending a message example

On the other hand, if you have received a message event and want to reply to the sender you don't have to create a new message event from scratch but can directly create a reply. This ensures that all important information such as the conversation id also appears in the answer. A reply can be created by calling `createReply(String type [, Object content])` method directly on the received message event. This method takes the message event type for the reply as parameter. Note that the message type with which you are replying still has to be present in your ADF.

Chapter 11. Expressions

For many elements (plan bodies, default and initial facts of beliefs, etc.) the developer has to specify expressions in the ADF. The most important part of an expression is the expression string. In addition, some meta information can be attached to expressions, e.g., to specify if the expression should be evaluated once (static) or dynamically for every access (dynamic).

11.1. Expression Syntax

The expression language follows a Java-like syntax. In general, all of the *operators* of the Java language are supported (with the exception of assignment operators), while no other constructs can be used. Operators are, for example, math operators (+-*/%), logical operators (&&, ||, !), and method, or constructor invocations. Other unsupported constructs are loops, class declarations, variable declarations, if-then-else blocks, etc. As a rule you can use every Java code that can be contained in the right hand side of a variable assignment (e.g., `var = <expression>`). There are just two exceptions to this rule: Declarations of anonymous inner classes are not supported. Assignment operators (=, +=, *=...) as well as de- and increment operators (++, --) are not allowed, because they would violate declarativeness.

In addition to the Java-like syntax, the language has some extensions: Parameters give access to specific elements depending on the context of the expression. OQL-like select statements allow to create complex queries, e.g., for querying the beliefbase. To simplify the Java statements in the expressions, imports can be declared in the ADF (see Chapter 5, *Imports*) that allow to use unqualified class names. The imports are defined once, and can be used for all expressions throughout the ADF.

11.2. Expression Properties

Expressions have properties which can be specified as attributes of the enclosing XML tag. The optional class attribute can be specified for any expression, and is used for cross checking the expression string against the expected return type. This allows to detect errors in the ADF already at load time, which could otherwise only be reported at runtime. The evaluation mode influences when and how often the expression is evaluated at runtime. A "static" expression caches the value once the expression created, while the value of a "dynamic" expression is reevaluated for ervery access. The default values of these properties depend on the context in which the expression is used. E.g. initial facts of beliefs are usually static, while conditions are dynamic.

Expressions that are derived from other elements as well as traced conditions (see below) need to know, which changes inside the agent affect the value of the expression. For example, an expression referring to a belief would have to be updated, when the belief has changed. The expression parser is able to autodetect most of these dependencies: References of belief(set)s, to goal parameters, and to the content of the goalbase (e.g., to react to the addition or removal of a goal). But sometimes you may want to react to changes that cannot be detected automatically. The `<relevant...>` tags (see Figure 11.1, “The Jadex expressions relevant settings XML schema part”) allow to specify an arbitrary number of elements on which an expression depends. These tags require the specification of a reference to an element, i.e., a belief(set), goal, or parameter(set) by using the "ref" attribute. Parameter references take the form "goalname.parametername" unless the goal can be determined from the expression context (e.g., in a goal condition). In this case and for the other references, the name of the element suffices. Optionally, a system event type can be given (see interface `jadex.model.ISystemEventTypes` for available event types) to further restrict the dependency to only specific changes, such as `BSFACT_ADDED` using the "eventtype" attribute.

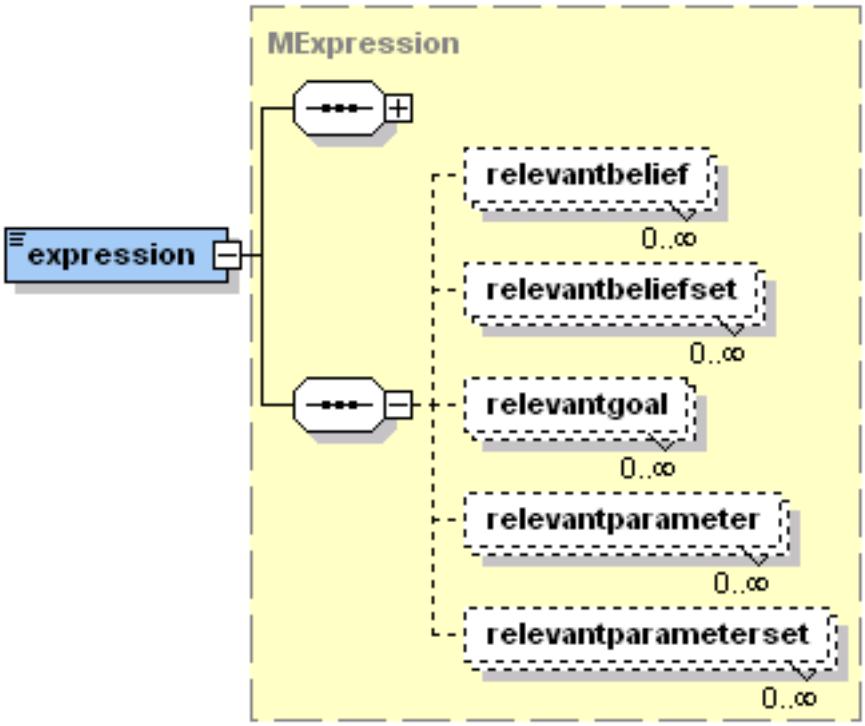

Figure 11.1. The Jadex expressions relevant settings XML schema part

11.3. Reserved Variables

Within expressions, several variables can be accessed depending on the context the expression is used in. Generally, the following variable names are reserved for agent components and can be accessed directly by their name. In table Table 11.1, “Reserved Expression variables” the reserved variables, their type and accessibility settings are summarized. Values of beliefs and belief sets (from the $beliefbase) and parameter(set)s (from $plan, $event, $goal, and $ref) can be accessed using a shortcut notation (allowing to write statements like "$beliefbase.mybelief"). Note that these variables do not refer to the usual interfaces from `jadex.runtime`, but to implementation classes from `jadex.runtime.impl`. In general, you should use these objects as if they were the interfaces.

Table 11.1. Reserved Expression variables

Name	Class
`$args`	`Object[]`
`$agent`	`RBDIAgent`
`$scope`	`RCapability`
`$beliefbase`	`RBeliefbase`
`$planbase`	`RPlanbase`
`$goalbase`	`RGoalbase`
`$eventbase`	`REventbase`
`$expressionbase`	`RExpressionbase`
`$propertybase`	`RPropertybase`

Name	Class
$goal	RGoal
$plan	RPlan
$event	REvent
$ref	RGoal

11.4. Expressions Examples

In Figure 11.2, “Example expressions”, two example expressions are shown. Here the expressions are used to specifiy the facts of some beliefs. In fact there are many places besides beliefs in the ADF where expressions can be used. In the first case, the "starttime" fact expression is evaluated only once when the agent is born. The second belief represents the agent's lifetime and is recalculated on every access.

```
<belief name="starttime" class="long">
    <fact>
        System.currentTimeMillis()
    </fact>
</belief>

<belief name="lifetime" class="long">
    <fact evaluationmode="dynamic">
        System.currentTimeMillis() - $beliefbase.starttime
    </fact>
</belief>
```

Figure 11.2. Example expressions

11.5. ADF Expressions

The expression language cannot only be used to specify values for beliefs, plans, etc. in the ADF but also for dynamic evaluation, e.g., to perform queries on the state of the agent, most notably the current beliefs. Expressions (`jadex.runtime.IExpression`) can be created at runtime by providing an expression string. A better way is to predefine expressions in the ADF in the expression base (see Figure 11.3, “The Jadex expressions XML schema part”). Because predefined expressions only have to be parsed and precompiled once and can be reused by different plans, they are more efficient. The following example shows a predefined expression for searching the beliefbase for a certain person contained in the belief persons, using the OQL-like language extension described in more detail below. Moreover, this example uses a custom parameter $surname to specify which person to retrieve from the belief set.

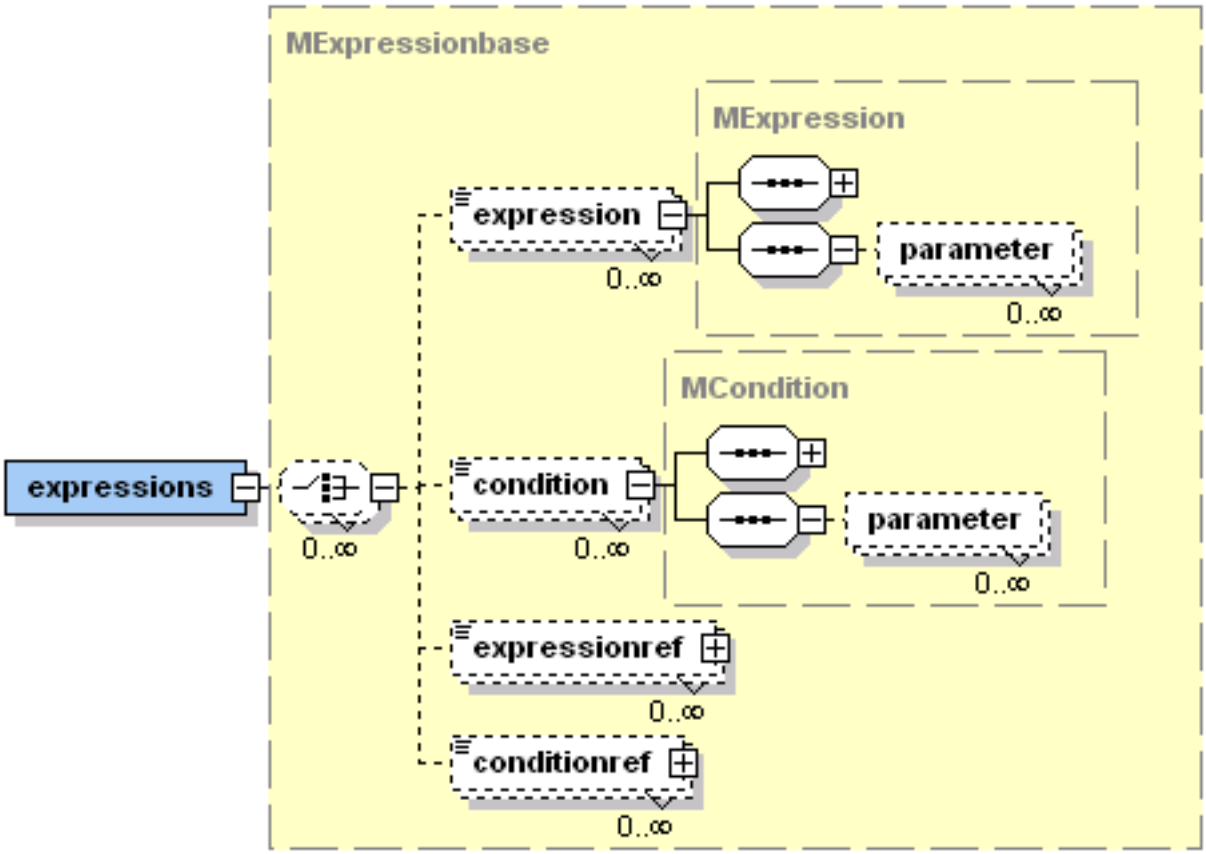

Figure 11.3. The Jadex expressions XML schema part

Primary usage of predefined expression is to perform queries, when executing plans. The `getExpression(String name)` method creates an expression object based on the predefined expression with the given name. In addition, the `createExpression(String exp [, String[] paramnames, Class[] paramtypes])` method is used to create an expression directly by parsing the expression string (without referring to a predefined expression). Custom parameters can be optionally be defined for such queries by additionally providing the parameter names and classes. Values for these parameters have to be supplied when executing the query. The expression object provides several `execute()` methods to evaluate a query specifying either no parameters, a single parameter as name/value pair, or a set of parameters in a `jadex.util.Tuple` array, where each tuple represents a name/value pair. You can also pre-set parameters before executing the query using the `setParameter()` method. For example, one can execute the person query with a given surname.

```
<agent ...>
    ...
    <expressions>
        <expression name="find_person" class="Person">
            select one Person $person
            from $person in $beliefbase.persons
            where $person.getSurname().equals($surname)

            <parameter name="$surname" class="String"/>
        </expression>
        ...
    </expressions>
    ...
</agent>
```

Figure 11.4. Defining an expression in the ADF

```
public void body {
    IExpression query = getExpression("find_person");
    ...
    Person person = (Person)query.execute("$surname", "Miller");
    ...
}
```

Figure 11.5. Evaluating an expression from a plan

11.6. OQL-like Select Statements

Jadex provides an OQL-like query syntax, which can be used in conjunction with any other expression statements. OQL (Object-Query-Language) is an extension to SQL (Structured-Query-Language) for object-oriented databases. The generic query syntax as supported by Jadex is very similar to OQL (note that until now only select statements are supported). The syntax is shown in Figure 11.6, “Syntax of OQL-like select statements”.

```
select (one)? <class>? <result-expression>
from (<class>? $<element> in)? <collection-expression>
    (, <class>? $<element> in <collection-expression>)*
(where <where-expression>)?
(order by <ordering-expression> (asc | desc)? )?
```

Figure 11.6. Syntax of OQL-like select statements

Unlike OQL and SQL, the keywords (select etc.) are currently case sensitive and have to be written in lower case. Also unlike OQL, the query variable $<element> has to start with the '$' character. The <collection-expression> has to evaluate to an object that can be iterated (an array or an object implementing `Iterator`, `Enumeration`, `Collection`, or `Map`). In the other expressions (result, where, ordering) the query variables can be accessed using $<element>. When using "$<element>" as result expression, the second "$<element> in" part can be omitted for readability. While you are free to use any expression for the result and the ordering, the where clause, of course, has to evaluate to a boolean value.

Some simple example queries (assuming that the beliefbase contains a belief set "persons", where each person has attributes "forename", "surname", "age", and "address") are shown in Figure 11.7, “Examples of OQL-like select statements”. The first query returns a `java.util.List` of all persons in the order they are contained in the belief set. The second query only returns persons that are older than 21. In this case a cast is used to invoke the `getAge()` method. The third example orders the returned list by the addresses of the persons, using a type declaration at the beginning of the query, and therefore does not need a cast for accessing the `getAddress()` method. The order-by implementation relies on the `java.lang.Comparable` interface. In the example, the addresses have to be comparable for the query to work. The next query shows that it is possible to use complex expressions to create the result elements. Note, that in this case, the "$person in" part cannot be ommited. The last example shows how to do a join. The expression returns a list of strings of any two (distinct) persons, which have the same address.

```
select $person from $beliefbase.persons

select $person from $beliefbase.persons where ((Person)$person).getAge()>21

select Person $person from $beliefbase.persons order by $person.getAddress()
```

```
select $person.getSurname()+", "+$person.getForename()
from Person $person in $beliefbase.persons

select $p1+", "+$p2 from Person $p1 in $beliefbase.persons,
                         Person $p2 in $beliefbase.persons
where $p1!=$p2 && $p1.getAddress().equals($p2.getAddress())
```

Figure 11.7. Examples of OQL-like select statements

An extension to OQL is the support of the "one" keyword. The default (without "one") is standard OQL semantics to return all objects matching the query. The "one" keyword is used to select a single element. For queries without ordering, this returns the first found element that matches the query. When using ordering, the query is evaluated for all input elements and returns the first element after having applied the ordering. In both cases null is returned, when no element matches the query. Without the "one" keyword, an empty collection is returned, when no element matches the query.

Chapter 12. Conditions

In essence, a condition is a monitored boolean expression, what means that whenever some of the referenced entities (e.g., beliefs) change the expression of the condition is evaluated. Associated with a condition is an action, that gets executed whenever the condition is triggered. Context-specific conditions as defined in the ADF have special associated actions (e.g., for activating goals). For custom conditions created by plans the default action is to generate an internal event of type `jadex.model.IMEventbase.TYPE_CONDITION_TRIGGERED`.

The behaviour of a custom condition can be adjusted with the trigger attribute. Note, that the trigger types of predefined conditions such as goal or plan creation conditions cannot be changed. Several trigger types are available: A condition can be triggered, e.g., whenever it is evaluated to true, or only triggered when its value first changes to true, but not when it stays true for some time. The list of available trigger types is given in Table 12.1, “Condition Trigger Types”. The default trigger type of a predefined condition depends on the context, for example the maintain condition of a maintain goal is triggered when the expression value changes to false, because the goal should be processed whenever the maintain condition is violated.

Table 12.1. Condition Trigger Types

Name	Description
`changes_to_true`	Execute action when expression value changes to true
`changes_to_false`	Execute action when expression value changes to false
`changes`	Execute action when the expression value changes
`is_true`	Execute action whenever expression evaluates to true
`is_false`	Execute action whenever expression evaluates to false
`always`	Execute action whenever expression is evaluated (regardless of value)

12.1. ADF Conditions

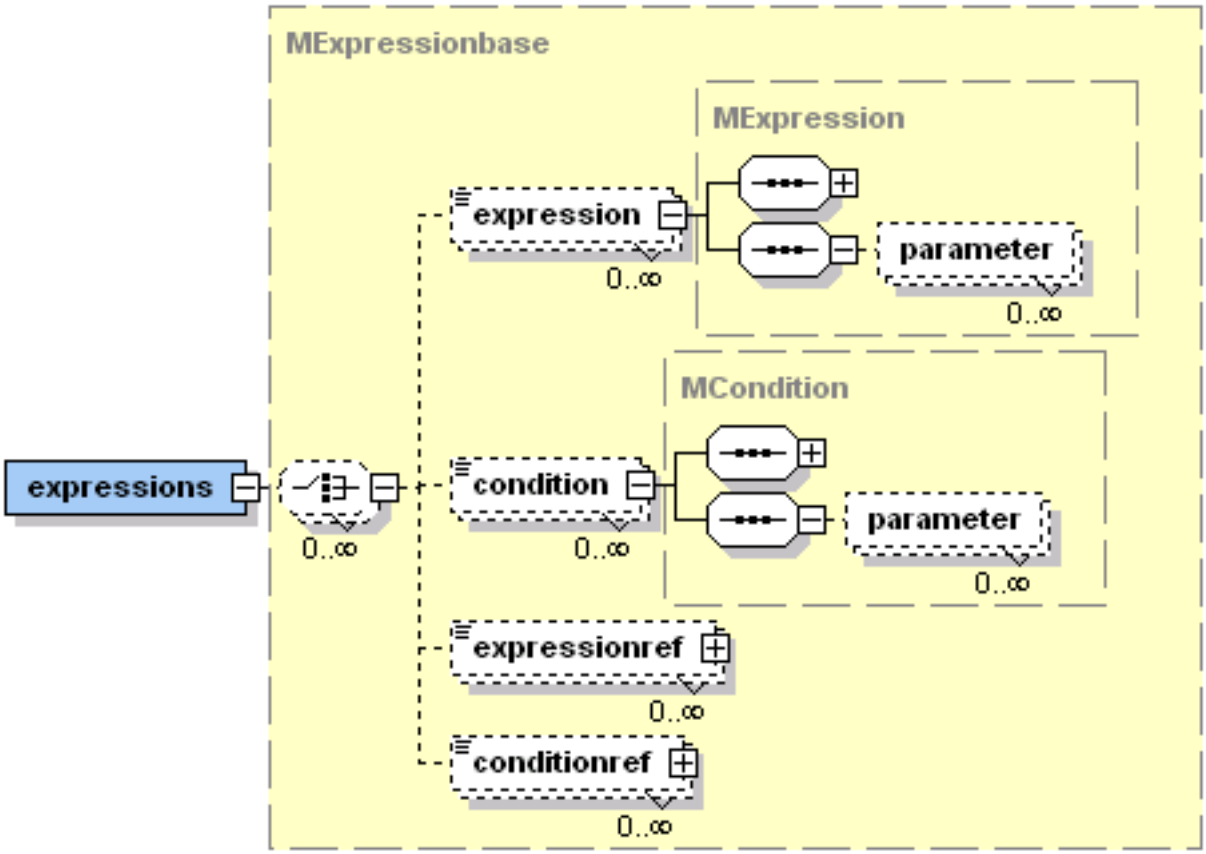

Figure 12.1. The Jadex conditions XML schema part

When programming plans, it is also possible to explicitly wait for certain conditions using the `waitFor(ICondition cond)` method. Conditions are obtained in a similar fashion to expressions, either by instantiating a predefined condition from the ADF (see Figure 12.1, “The Jadex conditions XML schema part”), or by creating a new condition from an expression string. When waiting for a condition, the plan will be blocked until the condition triggers, which by default means that its value changes to true. The condition is monitored automatically by the agent, by considering all internal state changes that may affect the condition value, e.g., when some other plan changes a belief. The following example uses the "timer" belief from Section 7.3, “Dynamically Evaluated Beliefs” to execute some code at every full hour.

```
<agent ...>
    ...
    <expressions>
        <condition name="full_hour">
            $beliefbase.timer%3600000==0
        </condition>
        ...
    </expressions>
    ...
</agent>
```

Figure 12.2. Defining a condition in the ADF

```
public void body {
    ICondition condition = getCondition("full_hour");
    ...
    while(true) {
        // Wakeup every full hour.
```

```
        IEvent event = waitForCondition(condition);
        ...
    }
}
```

Figure 12.3. Using a condition inside a plan

Chapter 13. Properties

This chapter contains an overview about the usage of agent and capability properties, that allow to change the behaviour of the agent. Properties can be defined in two different ways. First, you can use the properties section of the agent (and capability) XML file and add an arbitrary number of properties. Secondly, the agent tag has an optional attribute "propertyfile" which refers to an XML file containing important definitions. The default value of this attribute is the `jadex/config/runtime.properties.xml` file which specifies basic Jadex agent properties and normally can be used for all agents, but if you would like to provide the same set of properties to several agents, you can define your own XML property file and set the properties attribute of your agents accordingly.

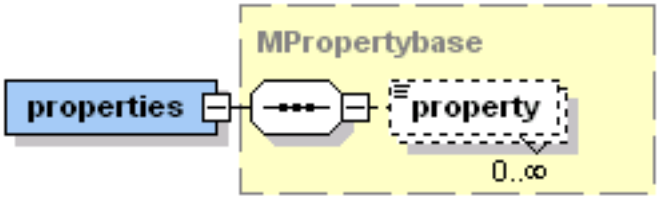

Figure 13.1. The Jadex properties XML schema part

Properties specified in the properties section override values loaded from the included property file. In addition, some properties can be defined individually for each capability, which otherwise inherits the properties of the outer capability or agent. Table 13.1, “Available properties” gives an overview of the available properties. The scope denotes, if the property can only be specified for the agent as a whole, or can be adjusted to different values for individual capabilities.

Table 13.1. Available properties

Scope	Property	Default	Possible Values
agent	`max_planstep_time`	unlimited	Positive `long` or `0` for unlimited
agent	`storedmessages.size`	unlimited	Positive `int` or `0` for unlimited
agent	`debugging`	`false`	{`true`, `false`}
capability	`logging.level`	`SEVERE`	`java.util.logging.Level` instances
capability	`logging.useParentHandlers`	`true`	{`true`, `false`}
capability	`logging.addConsoleHandler`		`java.util.logging.Level` instances
capability	`logging.level.exceptions`	`SEVERE`	`java.util.logging.Level` instances

The Jadex sytem has to take care that only one plan step is executed at a time, therefore it waits until a plan step returns. With the help of the "max_planstep_time" property it is possible to set the maximum execution time for a single planstep in milliseconds. Per default the execution time is not limited and a plan might execute as long plan steps as it want to (note that long plan steps are not recommended, because they hinder the agent in responding to urgent events). A plan running longer than the maximum plan step time will be forcefully aborted by the system. This feature is only available for standard, but not for mobile plans.

The "storedmessages.size" property can be used to restrict the number of monitored conversations. Generally, an agent has to keep track of its sent messages for being able to associate an incoming message to already sent

messages. This means an agent has to know what it sent to determine if it received some reply of a previous message. When restricting the number of conversations, and a message arrives belonging to a conversation that was removed from the cache, the agent might not be able to route the message to the correct capability.

The "debugging" property influences the execution mode of the agent. When setting debugging to `true` the agent is halted after startup and set to single-step mode. You can then use the debugger tab of the introspector tool execute the agent step-by-step and observe its behaviour.

The logging properties can be used to adjust the logging behaviour according to the Java Logging API. The level influences the amount of logging information produced by the agent (logging information below the level will be completely ignored). Setting "useParentHandlers" to "true" will forward logging information to the parent handler, which by Java default causes logging output up to the INFO level to be displayed on the console. If you want to direct more detailed logging output to the console use the "addConsoleHandler" property, which creates a custom logging handler for console ouput with the specified logging level. More about logging settings can be found in [Jadex Tool Guide].

The "logging.level.exceptions" property can be used to specify the logging level for uncatched exceptions occurring in plan bodies. Using the default settings for logging (non-BDI specific) exceptions are printed out as SEVERE log messages to the console. You can adjust the level settings to suppress exception log messages from plans that you except to throw exceptions.

Figure 13.2, “Example properties section” shows an example property section setting logging and plan step options.

```
<properties>
    <property name="logging.level">Level.WARNING</property>
    <property name="scheduler.max_planstep_time">5000</property>
</properties>
```

Figure 13.2. Example properties section

Chapter 14. Initial States

Initial states of an agent or a capability represent their configuration at creation time. In an initial state, instance elements can be declared that are created when the agent (resp. the capability) is started. This means an initial state is the right place to specify which plans should initially run or which goals the agent shall pursue after creation. It is possible to declare any number of initial states for a single agent or capability. When starting an agent or including a capability you can choose among the available initial states (called configurations). In Figure 14.1, “The Jadex initialstates XML schema part” the XML portion for specifying initial states is depicted. Each initial state must have a name for identification purposes. The default initial state can be set up by using the `default` attribute of the `<initialstates>` base tag. If no explicit default initial state is specified, the first one declared in the ADF is used.

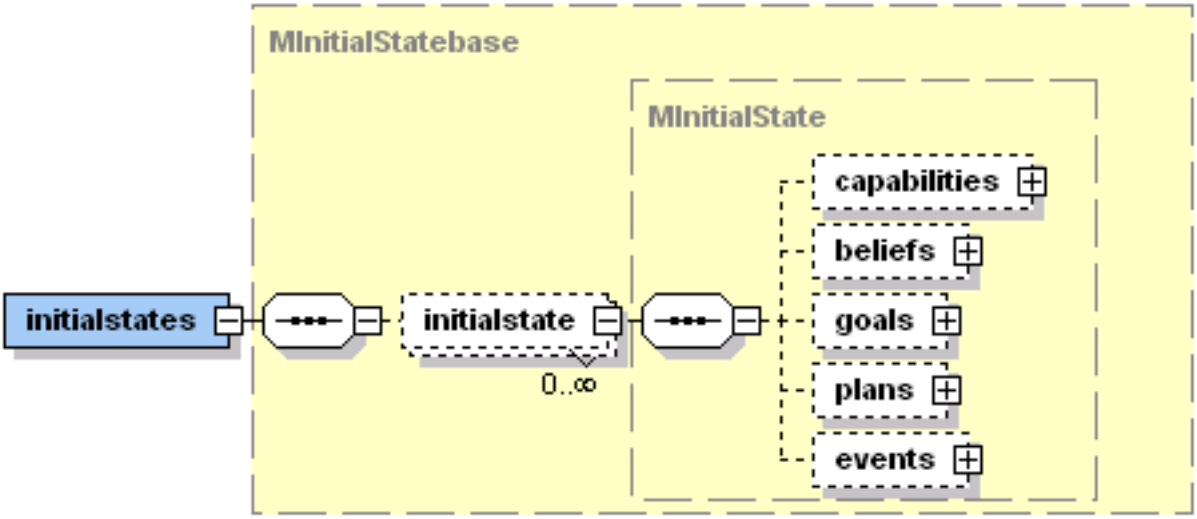

Figure 14.1. The Jadex initialstates XML schema part

An initial state allows to specify various properties. Generally, the initial state allows two different kinds of adaptations. The first one is the creation of instance elements for declared types, e.g., initial goals or plans. The second one is the configuration of instance elements such as beliefs or capabilities. In the following, the possible settings will be discussed.

14.1. Capabilities

The `<capabilities>` tag allows to configure included capabilities. For this purpose a reference to an included `<initialcapability>` must be declared. The reference to the capability is established by setting the `ref` attribute to the symbolic name of the capability specified within the `<capabilities>` section of the agent/capability (i.e., not the type name but the instance name). The initial state to be used by the included capability can be set by using the `initialstate` attribute of the initial capability.

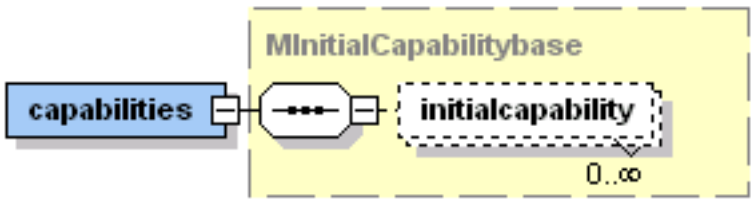

Figure 14.2. The Jadex initial capabilities XML schema part

In Figure 14.3, “Initial capability configuration” an example is shown how the initial state can be used to declare two different initial states. In state "one" the included capability "mycap" is configured to use its initial state "a", while in state "two" "b" is used. Per default the agent would start using initial state "two" as it is declared as default.

```
<agent ...>
    ...
    <capabilities>
        <capability name="mycap" file="SomeCapability"/>
    </capabilities>
    ...
    <initialstates default="two">
        <initialstate name="one">
            <capabilities>
                <initialcapability ref="mycap" initialstate="a"/>
            </capabilities>
        </initialstate>
        <initialstate name="two">
            <capabilities>
                <initialcapability ref="mycap" initialstate="b"/>
            </capabilities>
        </initialstate>
    </initialstates>
</agent>
```

Figure 14.3. Initial capability configuration

14.2. Beliefs

In the `<beliefs>` section the initial facts of beliefs and belief sets can be altered or newly introduced. In order to set the initial fact(s) of a belief or belief set an `<initialbelief>` resp. an `<initialbelief set>` tag should be used. The connection to the "real" belief is again established via the `ref` attribute and the facts can be declared in the same way as default values of beliefs and belief sets. The initial state does not distinguish between original beliefs and references to beliefs from other capabilities, therefore the same tags can also be used to change initial facts of belief references and belief set references as well.

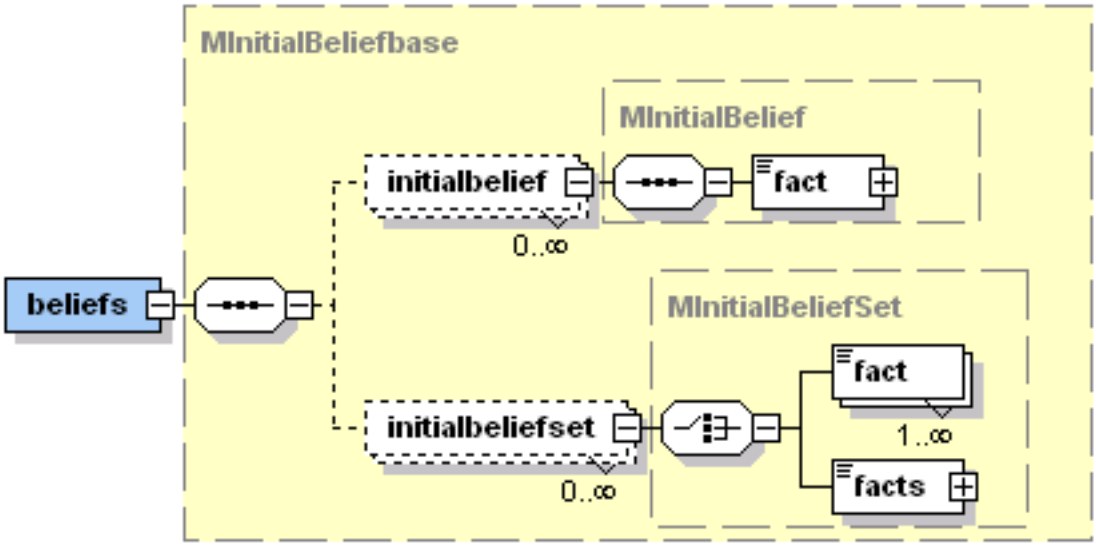

Figure 14.4. The Jadex initial beliefs XML schema part

The example in Figure 14.5, “Initial belief configuration” shows how an initialstate can be used to change belief facts. Belief "name" has a default value of "Jim" which is overridden by the initial belief fact "John". The belief set "names" has no default values. In the initial state it is filled with some data from a database. This means that for all results that the method `DB.queryNames()` produces, a new fact is added to the belief set.

```
<agent ...>
    ...
    <beliefs>
        <belief name="name" class="String">
            <fact>"Jim"</fact>
        </belief>
        <beliefset name="names" class="String"/>
    </beliefs>
    ...
    <initialstates>
        <initialstate name="one">
            <beliefs>
                <initialbelief ref="name">
                    <fact>"John"</fact>
                </initialbelief>
                <initialbelief set ref="names">
                    <facts>DB.queryNames()</facts>
                </initialbelief set>
            </beliefs>
        </initialstate>
    </initialstates>
</agent>
```

Figure 14.5. Initial belief configuration

14.3. Goals

In the `<goals>` section initial goals can be specified. This means that a new goal instance is created for each declared initial goal. The specification of an `<initialgoal>` requires the connection to the underlying goal template which is used for instantiation. For this purpose the `ref` attribute is used. Optionally, further parameter(set) values can be declared by using the corresponding `<parameter>` and `<parameterset>` tags.

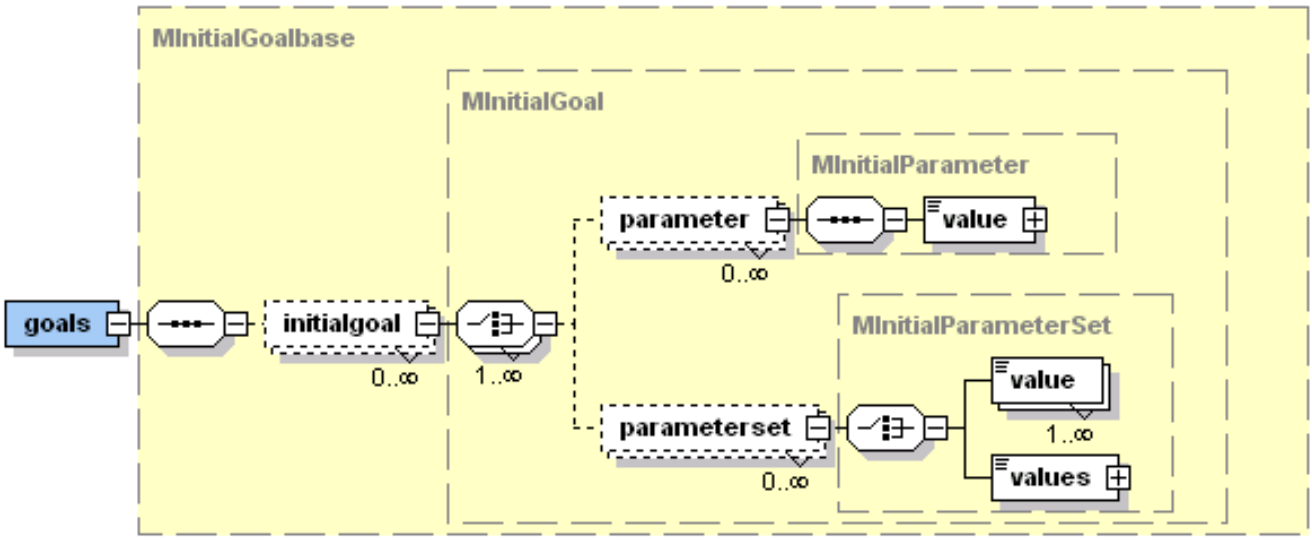

Figure 14.6. The Jadex initial goals XML schema part

In Figure 14.7, “Initial goals” an example is depicted how an initial goal can be created. The initial goal refers to the declared "play_song" perform goal of the agent and provides a new parameter value for the song parameter. When the agent is started in this initial state it creates the initial goal and pursues it. So, given that the agent has some plan to play an mp3 file, it will play a welcome song in this example.

```
<agent ...>
    ...
    <goals>
        <performgoal name="play_song">
            <parameter name="song" class="URL"/>
        </performgoal>
    </goals>
    ...
    <initialstates>
        <initialstate name="one">
            <goals>
                <initialgoal ref="play_song">
                    <parameter ref="song">
                        <value>new URL("http://someserver/welcome.mp3")</value>
                    </parameter>
                </initialgoal>
            </goals>
        </initialstate>
    </initialstates>
</agent>
```

Figure 14.7. Initial goals

14.4. Plans

In the `<plans>` section initial plans can be specified. This means that a new plan instance is created for each declared initial plan. The specification of an `<initialplan>` requires the connection to the underlying plan template which is used for instantiation. For this purpose the `ref` attribute is used. Optionally, further parameter(set) values can be declared by using the corresponding `<parameter>` and `<parameterset>` tags.

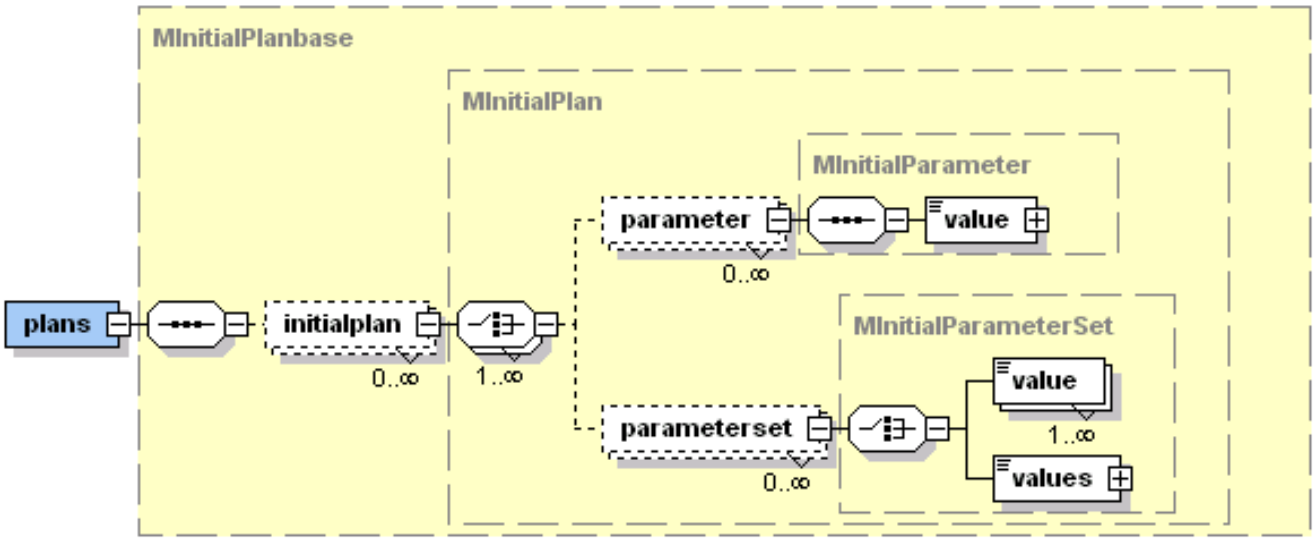

Figure 14.8. The Jadex initial plans XML schema part

In Figure 14.9, “Initial plans” an example is depicted how an initial plan can be used. In this case an initial

"print_hello" plan is declared which refers to the "print_hello" plan template of the agent. As result the agent will print "Hello World!" to the console on start-up.

```
<agent ...>
    ...
    <plans>
        <plan name="print_hello">
            <body>new PrintOnConsolePlan("Hello World!")</body>
        </plan>
    </plans>
    ...
    <initialstates>
        <initialstate name="one">
            <plans>
                <initialplan ref="print_hello"/>
            </plans>
        </initialstate>
    </initialstates>
</agent>
```

Figure 14.9. Initial plans

14.5. Events

Finally, in the `<events>` section initial events can be specified. This means that a new event instance is created for each declared initial event. It is possible to define initial internal and initial message events (goal events are not necessary as initial goals can be declared). The specification of an `<initialinternalevent>` or an `<initialmessageevent>` requires the connection to the underlying event template which is used for instantiation. For this purpose the `ref` attribute is used. Optionally, further parameter(set) values can be declared by using the corresponding `<parameter>` and `<parameterset>` tags.

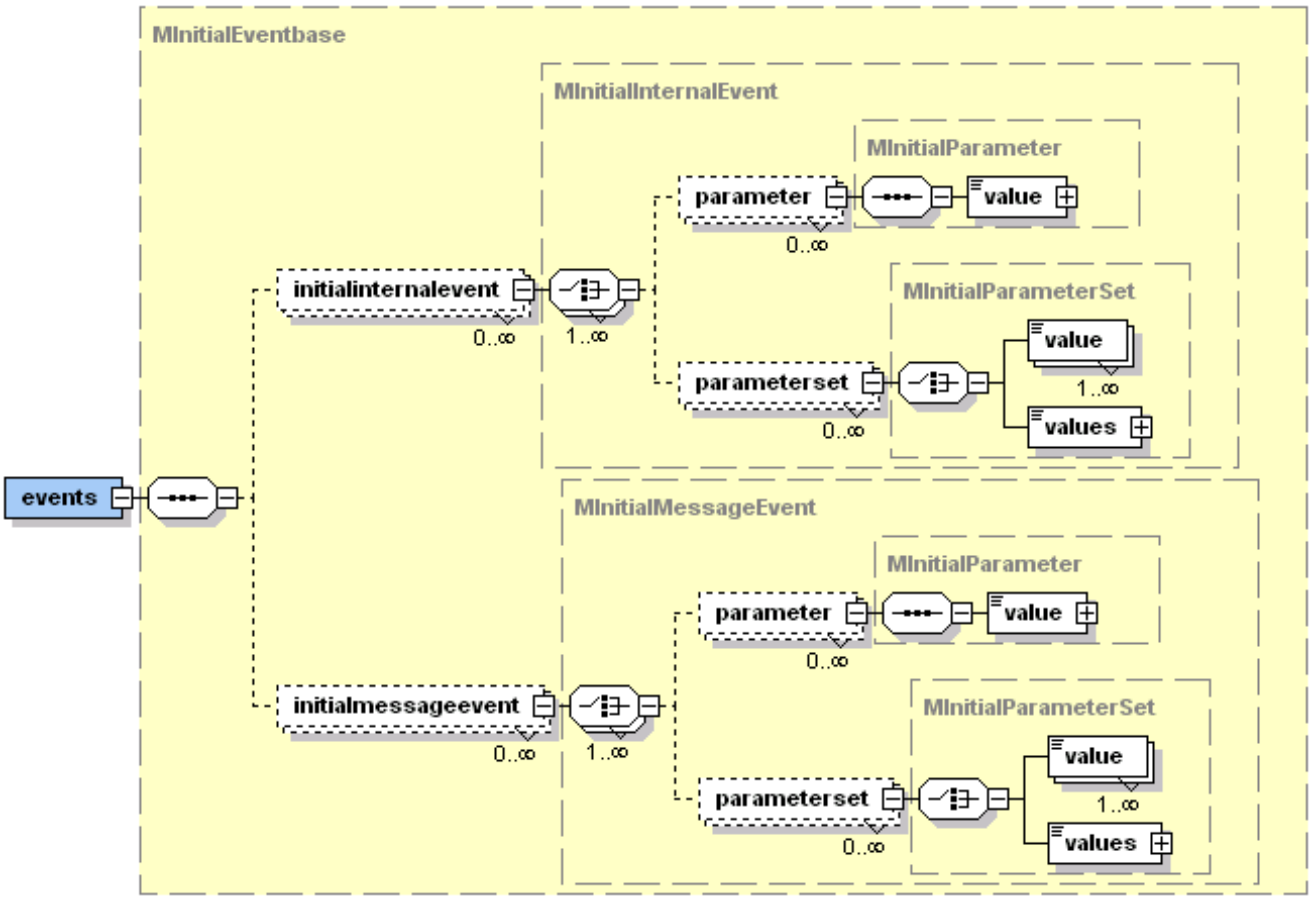

Figure 14.10. The Jadex initial events XML schema part

In Figure 14.11, “Initial events” an example is shown how an initial message event can be created. The intial message event refers to the underlying message event template "inform_born" and sets the parameter values for the content as well as for the receiver. When the agent "Harry" is started, it sends an initial message event with the content "Harry is born." to an agent named "Uncle" on the same platform.

```
<events>
    <messageevent name="inform_born" type="fipa" direction="send">
        <parameter name="performative" class="String" direction="fixed">
            <value>SFipa.INFORM</value>
        </parameter>
    </messageevent>
</events>
...
<initialstates>
    <initialstate name="one">
        <events>
            <initialmessageevent ref="inform_born">
                <parameter ref="content">
                    <value>$agent.getAgentName()+" is born."</value>
                </parameter>
                <parameterset ref="receivers">
                    <value>new AgentIdentifier("Uncle", true)</value>
                </parameterset>
            </initialmessageevent>
        </events>
    </initialstate>
</initialstates>
```

Figure 14.11. Initial events

Chapter 15. Dynamic Models

Jadex offers the possibility to change the underlying agent and capability models at runtime, i.e. it is e.g. easily possible to define new beliefs, goals and plans and the like at runtime. For this purpose every capability and agent instance owns it own copy of the underlying model. In this respect changes to a model remain local to the corresponding capability or agent instance, and the original model contained in the ADF will never be changed this way.

The process for creating model elements at runtime is very simple in principle. It consists of two steps:

- *Create the model element in the model.* Regardless, which element should be created it is necessary to fetch the corresponding model element that belongs to the current scope (capability):

```
IMCapability model = (IMCapability)getScope().getModelElement();
```

  Depending on which element you want to create you can access e.g. the different bases such as the planbase and create a new model element via the various `create...()` methods. For a detailed overview consider looking into the API docs of the Jadex model accessible through the interfaces contained in the `jadex.model` package.

- *Register the model element at the runtime.* To make the runtime aware of the new element it is necessary to call one of the `register...()` methods at the corresponding runtime bases (or the capability itself).

Similarly, it is possible to delete elements from the model in two steps:

- *Deregister the model element at the runtime.* To clean-up the element at runtime the suitable `deregister...()` method should be called.

- *Delete the model element from the model.* Again, it is necessary to have access to the model laver via:

```
IMCapability model = (IMCapability)getScope().getModelElement();
```

  Depending on which element you want to delete you can access e.g. the different bases such as the planbase and delete an existing model element via the various `delete...()` methods.

15.1. Adding/Removing Capabilities at Runtime

Using the model API (package `jadex.model`) plans can dynamically load and unload capabilities. To load a capability, you first have to create a so called capability reference in the agent (or capability) model. The `createCapabilityReference()` method works the same as the `<capability>` tag shown in Figure 6.3, “Including subcapabilities”, and therefore expects the local name of the subcapability and a filename. After creating the reference, which also loads the given XML file, you can register the new capability at runtime (this will initialize the capability by creating initial goals, plans, etc.). To use the features of the capability you can also dynamically add references to the exported beliefs and goals of the capability. Figure 15.1, “Adding a capability at runtime” shows how to include a capability at runtime.

```
public void body() {
    // Create reference in the model.
    IMCapability model = (IMCapability)getScope().getModelElement();
    IMCapabilityReference subcap = model.createCapabilityReference("dfcap",
        "jadex.planlib.DF");
```

```
    // Register capability at runtime.
    getScope().registerSubcapability(subcap);
    ...
}
```

Figure 15.1. Adding a capability at runtime

The removal of a capability can be done likewise. In Figure 15.2, “Removing a capability at runtime” an example code is depicted.

```
public void body() {
    ...
    IMCapabilityReference subcap = ((IMCapability)getScope().getModelElement())
        .getCapabilityReference("subcap_name");

    // Deregister subcapability at runtime.
    getScope().deregisterSubcapability(subcap);

    // Delete subcapability from the model.
    ((IMCapability)getScope().getModelElement()).deleteCapabilityReference(subcap);
    ...
}
```

Figure 15.2. Removing a capability at runtime

15.2. Creating/Deleting Beliefs at Runtime

Usually all agent beliefs are defined in the ADF. At runtime only the facts contained in the beliefs change, not the beliefs themselves. The model API (package `jadex.model`) allows to dynamically create new beliefs and belief sets at runtime, if this self-modifying functionality is required. To create a new belief, it first has to be defined in the agent or capability model. The `createBelief...()` methods of the `IMBeliefbase` work the same as the belief/set/reference tags shown in Figure 7.1, “The Jadex beliefs XML schema part”. For beliefs and belief sets a name, class, update rate and exported flag have to be specified. For belief(set) references instead of an update rate the path of the referenced belief(set) has to be provided.

After creating a belief(set) or reference in the model, you have to register the new element at runtime (this will also evaluate the initial facts, if you have supplied some in the model). Figure 15.3, “Creating a belief at runtime” shows how to create a belief at runtime.

```
public void body() {
    ...
    // Create belief in the model.
    IMBeliefbase model = (IMBeliefbase)getBeliefbase().getModelElement();
    IMBelief belief = model.createBelief("name", String.class, -1, false);

    // Register belief at runtime.
    getBeliefbase().registerBelief(belief);
    ...

    // Access the belief as usual.
    getBeliefbase().getBelief("name").setFact("Hugo");
    ...
}
```

Figure 15.3. Creating a belief at runtime

In Figure 15.4, “Deleting a belief at runtime” it is shown how a belief can be deleted at runtime.

```
public void body() {
    ...
    IBelief belief = getBeliefbase().getBelief("belief_name");
    IMBelief mbelief = (IMBelief)belief.getModelElement();

    // Deregister the belief at runtime.
    getBeliefbase().deregisterBelief(mbelief);

    // Delete the belief in the model.
    IMBeliefbase model = (IMBeliefbase)getBeliefbase().getModelElement();
    model.deleteBelief(mbelief);
    ...
}
```

Figure 15.4. Deleting a belief at runtime

15.3. Creating/Deleting Goal Types at Runtime

Usually all goal types (like, e.g., “performpatrol”) are defined in the ADF. At runtime instances of these types are created, but the set of available goal types remains the same. The model API (package `jadex.model`) allows to dynamically create new goal types and goal references at runtime, if this self-modifying functionality is required. To create a new goal type, it first has to be defined in the agent or capability model. The `create...Goal()` and `create...GoalReference()` methods of the `IMGoalbase` work the same as the tags shown in Figure 8.1, “The Jadex goals XML schema part”. For goals, a name, exported flag, retry, retry delay, and exclude mode are required. Maintain goals require in addition the specification of recur and recur delay. For references, the name, the exported flag, and the path to the referenced goal have to be specified.

After creating a goal or reference in the model, you have to register the new element at runtime (this will also activate the creation condition, if you have supplied one in the model). Figure 15.5, “Creating a new goal type at runtime” shows how to create a new goal type at runtime.

```
public void body() {
    ...
    // Create goal type in the model.
    IMGoalbase model = (IMGoalbase)getGoalbase().getModelElement();
    IMGoal goal = model.createPerformGoal("performpatrol", false, true, -1,
        IMGoal.EXCLUDE_NEVER);
    goal.createContextCondition("!$beliefbase.is_loading && !$beliefbase.daytime");

    // Register goal at runtime.
    getGoalbase().registerGoal(goal);
    ...
}
```

Figure 15.5. Creating a new goal type at runtime

An example for the deletion of a goal type at runtime is shown in Figure 15.6, “Deleting a goal type at runtime”.

```
public void body() {
    ...
        // Assuming that mgoal is the model of the element to be deleted

    // Deregister goal at runtime.
    getGoalbase().deregisterGoal(mgoal);

    // Delete goal type from the model.
    IMGoalbase model = (IMGoalbase)getGoalbase().getModelElement();
    model.deleteAchieveGoal((IMAchieveGoal)mgoal);
    ...
}
```

Figure 15.6. Deleting a goal type at runtime

15.4. Creating/Deleting Plan Types at Runtime

Adding new plan specifications to the agent or capability at runtime is similar to what has already been described for beliefs and goals. First the plan head has to be created using the API of the `jadex.model` package. Afterwards, the plan has to be registered at runtime, mainly for activating the plan trigger.

```
public void body() {
    ...
    // Create plan type in the model.
    IMPlanbase model = (IMPlanbase)getPlanbase().getModelElement();
    IMPlan plan = model.createPlan("ping", 0, "new PingPlan()", IMPlanBody.BODY_STANDARD);
    plan.createTrigger().createMessageEvent("query_ping");

    // Register plan at runtime.
    getPlanbase().registerPlan(plan);
    ...
}
```

Figure 15.7. Creating a new plan type at runtime

In the following the deletion of a plan type is sketched (see Figure 15.8, “Deleting a plan type at runtime”).

```
public void body() {
    ...
        // Assuming that mplan is the model of the element to be deleted

    // Deregister plan at runtime.
    getPlanbase().deregisterPlan(mplan);

    // Delete plan type in the model.
    IMPlanbase model = (IMPlanbase)getPlanbase().getModelElement();
    model.deletePlan(mplan);
    ...
}
```

Figure 15.8. Deleting a plan type at runtime

15.5. Creating/Deleting Event Types at Runtime

In general it is possible to create custom events at runtime. This applies for goal, message as well as internal events. Nevertheless, as goal events are used only internally creating message and internal events should be the only relevant use case.

Note

The creation of message event references at runtime is not recommended in the current version and may lead to difficulties when conversations are used.

In Figure 15.9, “Creating a new internal event type at runtime” an example is depicted how a new internal event can be created and registered.

```
public void body() {
    ...
    // Create event type in the model.
    IMEventbase model = (IMEventbase)getEventbase().getModelElement();
    IMInternalEvent ievent = model.createInternalEvent("new_ievent", false);
    ievent.createParameter("param1", String.class, IMParameter.DIRECTION_IN, 0, null, null);

    // Register event at runtime.
    getEventbase().registerEvent(ievent);
    ...
}
```

Figure 15.9. Creating a new internal event type at runtime

In Figure 15.10, “Deleting an internal event type at runtime” the removal of an internal event at runtime is outlined.

```
public void body() {
    ...
    // Assuming that imevent is the model of the event to be deleted

    // Deregister event at runtime.
    getEventbase().deregisterEvent(imevent);

    // Delete event type in the model.
    IMEventbase model = (IMEventbase)getEventbase().getModelElement();
    model.deleteInternalEvent(imevent);
    ...
}
```

Figure 15.10. Deleting an internal event type at runtime

Chapter 16. External Processes

A Jadex agent is synchronized in the sense, that only one plan step at a time is executed (or none, if the agent is busy performing internal reasoning processes). Sometimes you want to access agent internals from external threads. A good example is when your agent provides a graphical user interface (GUI) to accept user input. When the user clicks a button your Java AWT/Swing event handler method is called, which is executed on the Java AWT-Thread (there is one AWT Thread for each Java virtual machine instance). To force that such external threads are properly synchronized with the internal agent execution, you are not allowed to call Jadex methods directly from those threads. If you try to do so, a runtime exception “Wrong thread calling plan interface” will be thrown.

The AbstractPlan class provides a method `getExternalAccess()` which returns an accessor which automatically does the necessary thread synchronization. This accessor implements the `ICapability` interface, providing access to all features of the capability (beliefbase, goalbase, etc.). In addition, some convenience methods are provided to wait for goals to be completed or messages to be received. These methods should be used with caution, as they could easily lead to deadlocks. To avoid at least one source of deadlocks, it is not possible to call methods on this accessor from the plan thread. Whenever you call the wrong object from the wrong thread, a RuntimeException will immediately identify the problem. The following code presents an example where a belief is changed when the user presses a button.

```
public void body() {
    ...
    JButton button = new JButton("Click Me");
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            // This code is executed on the AWT thread (not on the plan thread!)
            IBeliefbase bb = getExternalAccess().getBeliefbase();
            bb.getBelief("button_pressed").setFact(new Boolean(true));
        }
    });
    ...
}
```

Figure 16.1. External process example

Appendix A. Changes and Compatibility Issues

Jadex is a rapidly evolving project. If you have used a previous version of Jadex, you should read this section carefully, to learn what has changed since then. This section shortly introduces important features that have been added and changes that have been made to the API, which may cause incompatibilities to applications you may have developed with an older Jadex version. For a detailed description of the many single changes and numerous bug fixes, have a look at the changes document (`changes.txt`), where you will find a history of all important changes that have been made since the initial release.

A.1. New Features in 0.94

JADE separation. Jadex has been cleanly separated from JADE. From this version Jadex is realized as reasoning engine that can be easily combined with different underlying agent platforms or even other kinds of middleware. Of course, it is still possible to run Jadex over JADE. The corresponding JADE adapter can be downloaded from the website.

Jadex standalone adapter. The new default middleware layer for Jadex is the new Jadex Standalone platform that represents the most efficient environment for running Jadex agents. The Standalone adapter supports most of the Jadex tools that were already available (except the logger) and additionally offers a new way for administering agents via the new *Jadex Control Center* tool.

Initial states. Agents and capabilities can be configured with so called initial states. An initial state comprises all information about instances of elements that should be created at startup such as initial plans or goals.

Faster model loading times. The underlying XML-databinding framework for loading XML files to Java objects has been changed. Since this version Jadex uses JiBX as primary databinding framework which is one of the fastest tools available.

Improved performance and scalability. Jadex has been greatly improved with respect to performance and memory usage. The new version running on the Standalone adapter is up to 100% (depending heavily on the concrete application) faster than the old version.

Finer-grained triggers and waitForXXX() methods. In the new version Jadex supports more diverse trigger types and many specialized waitForXXX() methods in plans. Most notably it is now possible to wait directly for belief and belief set changes (cf. e.g. `waitForBeliefChange()`, `waitForFactAdded()`, <beliefchange ref="belname"/>, etc.).

Unification of binding options with parameters. A simpler way for handling plans and goals with binding options is provided. Plans and goals only have parameters. If a plan or goal should be customized via binding options you can now delcare a parameter to fetch its value via binding options.

Improved examples. The BlackJack example now offers a human player interface. You can play with a mixture of human players and agents at one dealer. Of course the players can be distributed accross the network.

Re-established support for mobile agents. The support for serialization of Jadex agents at runtime has been re-established. Using the JADE adapter and the JADE migration service it is possible the migrate Jadex agents as well (using mobile plans).

Note

Jadex Add-ons. Several new tools and extensions are available for Jadex from the Jadex add-ons page. These include:

- **Expression Compiler:** In the normal version Jadex utilizes a Java expression interpreter. The expression compiler is based on Janino and allows to compile Java expressions on demand at runtime. The expression compiler can be used to further improve the performance of agents significantly. In addition to the faster evaluation of expressions the Jadex expression compiler add-on also provides the possibility to define *inline plan bodies*. This means that a whole agent can be programmed in one XML file without any additional plan classes. The add-on includes also a precompiler tool that allows to generate precompiled expression files for ADFs that are stored on disk. Using the tool avoids the need for runtime compilations of expressions that would delay the first agent execution otherwise.
- **Webbridge tool:** It can be used to seamlessly integrate Jadex agents with Java Server Pages (JSPs) and servlets. Using the Webbridge tool it is possible to design agent-based applications that can be used from a web interface. Web-requests are forwarded to the Jadex system and can be processed by the agents as normal message events. The result for a web-request will be returned to the web layer. If the result is a complete web page it will be displayed directly. On the other hand a result object can be postprocessed by some designated JSP for producing the final html page.
- **Planner:** For certain problem domains reactive planning is not the best achievable soluation. Instead planning from first principles should be used. Jadex is available in an extended version with an integrated high-speed state-of-the-art planner.
- **Diet adapter (experimental):** Besides the JADE and Standalone adapter an experimental adapter for the Diet platform has been developed. It shows the applicability of Jadex even running on fundamentally different middleware platforms. The adapter is still experimental as it has not reached a level of maturity yet and also does not explore the full potential of the Diet platform. Nonetheless, we could achieve that all examples bundled with the normal Jadex release are executable under every adapter including Diet without changes.

A.2. Incompatibilities to Release 0.932

Jadex is still in beta stage and to facilitate effective further development backwards compatibility is currently not explicitly addressed. As the concepts and the API undergoes continuous changes, applications developed with older versions of Jadex may not directly work with the current release. We hope that design issues will settle down until the release of version 1.0, at least for the common features.

Until then, this section tries to highlight the issues which might cause problems with your old code. Use this section as a hint how to adapt your Jadex applications to the new release.

A.2.1. Changes in the ADF Definition

ADF XML header. Due to the changed XML binding framework the headers of the XML agent and capability files have to be adapted with correct schema specifications (see Figure A.1, “Header of an agent definition file”).

```
<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-0.94.xsd"
       name=".."
       package="..">
```

Figure A.1. Header of an agent definition file

Capabilities from planlib. The capabilities from the Jadex plan library are available for the different platform adapters in different implementations. All of these implementations follow the specifications of the abstract capabilities contained directly in the `jadex.planlib` package. Nevertheless, from an application view all these implementations provide the same functionality through the same set of beliefs, goals, and plans. To build applications independently of the underlying adapter the planlib capabilities should be declared with their full name, such as `jadex.planlib.DF`. The system will resolve these names by using the properties from the Jadex configuration to the concrete implementation names such as `jadex.adapter.standalone.planlib.DF_standalone`.

Initial State. Instead of specifying plans as "initial", and initial goals directly in the goalbase, the initial state of the agent is now specified in a separate initial states section.

Agent/Service Descriptions. These have been removed as explicit tags. For creating agent or service descriptions the `createServiceDescription(...)createAgentDescription(...)` methods in class `jadex.adaper.fipa.SFipa` can be used.

Languages/Ontologies. These have also been removed as explicit tags. IF you are using Jadex without JADE you don't need to define those languages and ontologies in the agent resp. capability scope. Using the JADE adapter you can define properties starting with jade.language.xxx and jade.ontology.yyy for that purpose. The JADE agent adapter registers automatically all languages and ontologies that are declared in this way. This allows to en- and decode messages using the standard JADE codecs.

```
<!-- Example language/ontology properties. -->
<properties>
    <property name="jade.language.sl0">new SLCodec(0)</property>
    <property name="jade.ontology.cleaner">CleanerOntology.getInstance()</property>
</properties>
```

Outgoing Messages. From now on, only messages defined in the ADF can be sent and received. Message declarations in the ADF require the `type` attribute being set to "fipa". This equips "fipa" messages automatically with all required FIPA parameters such as "sender", "receivers", "conversation-id" and the like. In addition to the type a `direction` attribute can be used to distinguish explicitly betweeen messages that can be sent ("send"), received ("receive") and both ("send_receive").

A.2.2. API Changes

`createAID()`
: This method has been moved to the new `jadex.runtime.adapter.jade.SJade` class.

`JadeWrapperAgent`
: This class has been renamed and moved to `jadex.runtime.adapter.jade.JadeAgentAdapter`.

`getJadeAgent()`
: This method has been renamed to `getPlatformAgent()`.

`getGUIWrapper()`

This method has been renamed to `getExternalAccess()` and returns an extended interface (see Chapter 16, *External Processes*).

`MessageEventFilter`

`MessageEventFilter` (formerly `MessageFilter`) has been refactored to allow matching for instances and model elements or message events (as opposed to ACL messages).

`util.jade`

Contents of this package have been moved to `jadex.runtime.adapter.jade`.

Messages

Direct links to `ACLMessage` have been removed, so explicit casts are required to access JADE specific message features. Moreover, creation methods such as `createMessageEvent()` and `IMessageEvent.createReply()` have been changed (cf. Chapter 10, *Events*).

`createConversationId()`

This method has been moved and renamed to `SFipa.createUniqueId()`.

Agent

The agent now requires an additional configuration (string or model) argument before the custom arguments.

Arguments

Command-line arguments now are available as Object[] "$args" in the agent.

Appendix B. FAQ+HOWTO

B.1.

I get the message while loading an agent in the Jadex-RMA console: `No model loaded: java.io.IOException Unable to access binding information for class jadex.model.jibiximpl.MBDIAgent`

Use developer version or run

org.jibx.binding.Compile -v kernel/src/jadex/model/jibximpl/binding.xml

B.2.

What does "retrydelay" flag mean?

Without retrydelay goal processing works as follows: goal -> plan 1 -> plan 2 -> plan 3 -> ... until the goal is failed or succeeded.

The retrydelay just specifies a delay in milliseconds before trying the next plan, when the previous plan has finished, i.e.: goal -> plan 1 -> wait -> plan 2 -> wait -> plan 3 -> ... until goal fails or succeeds. This is e.g. useful, when already tried plans are not excluded from the applicable plan set, leading to the same plan being tried over and over again.

B.3.

How can the environment of a Jadex MAS be programmed?

We tried out different approaches for realizing the environment of a MAS. In the most simple case we realized the environment as a singleton object for all agents. Of course this approach is limited in nature as it is not possible to distribute the application over more than one Java VM. In this case we used a simple belief with a fact expression that refers to that singleton object, e.g. you can look at the garbage-collector example in the ADF you can find:

```
<!-- Environment object as singleton.-->
<belief name="env" class="Environment">
    <fact>Environment.getInstance($agent.getType(), agent.getName())</fact>
</belief>
```

If distribution is needed we used the approach of a separate environment agent. This agent administers the environment and permits several actions being executed on the environment object. Therefore a domain specific ontology is defined, in which the actions are contained together with the FIPA stuff (agent action, agent identifier etc.). In principle each agent has to create the specific action it wants to perform on the environment (such as moveup) and encode it into an AgentAction (see FIPA spec). The environment agent tries to execute the contained action and sends back the result e.g. Done(AgentAction). As this procedure is cumbersome, we used following idea. For every primitive action a goal is defined with corresponding plans that do the message handling. The agent programmer can subsequently use just the goals for interaction with the environment.

B.4.

I have change the `.java` file, e.g. a plan. Why are my changes not reflected in the running Jadex system?

Jadex relies on the Java class loading mechanism. This means that normally Java classes are loaded only

once into the VM. You need to restart the Platform for taking the changes effect. Since Jadex 0.94 a plan class reloading at runtime is also possible when the Jadex expression interpreter is used and the corresponding option "plan reloading" option is turned on in the Jadex settings. In contrast to plan classes XML changes inlcuding inline plan bodies (only available with the expression compiler add-on available from the Jadex add-on page) are directly reflected whenever the model is laoded. The reason for this is that inline plan bodies can be compiled on demand at runtime.

B.5.
In my agents there is always one *plan* for a *goal*. Why do I need goals anyway?

You don't need to use goals for every problem. But, in our opinion using goals in many cases simplifies the development and allows for easier extensions of an application. The difference between plans and goals is fundamental. Goals represent the "what" is desired while plans are characterized by the "how" could things be accomplished. So if you e.g. use a goal "achieve happy programmers" you did not specify how you want to pursue this goals. One option might be the increase of salary, another might be to buy new TFT monitors. Genereally, the usefulness of goals depends on the concrete problem and its complexity at hand.

B.6.
How can the agent become aware of or react to its own death?

There is a system event `AGENT_DIED` that can be accessed using a system listener. There is currently no way to start a new plan in response to the event. The prefered way is to start an endless plan with an abort method. I.e for closing the GUI of an agent:

```
public void body(){
    waitFor(IFilter.NEVER);
}

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

B.7.
How do I access the arguments specified in the Agent Starter Dialog?

The arguments may be accessed in the ADF under the variable `$args` of the type `Object[]`. Each argument can be accessed using `$args[0]`, `$args[1]`, ..., `$args[n]`.

Appendix C. Legal Notice

Licence. Jadex is distributed under the GNU Lesser General Public License (LGPL). In essence the license allows you to freely use and distribute Jadex for any kind of project (including commercial projects), but requires you to make available the Jadex sources including all changes that you have done.

C.1. Third-Party Software

Following libraries and software products are distributed with the resoning engine.

- JiBX
- GraphLayout
- JavaHelp

JiBX. The JiBX XML binding framework is used to load agent models from an agent definition file (ADF). It is distributed under following licence:

GraphLayout. The GraphLayout component is used in the Tracer Tool and is licenced under following terms:

JavaHelp System. The JavaHelp system is used in all runtime tools.

ARE ACCESSING THE SOFTWARE ELECTRONICALLY, INDICATE YOUR ACCEPTANCE OF THESE TERMS
SELECTING THE "ACCEPT" BUTTON AT THE END OF THIS AGREEMENT. IF YOU DO NOT AGREE TO ALL
THESE TERMS, PROMPTLY RETURN THE UNUSED SOFTWARE TO YOUR PLACE OF PURCHASE FOR A RE
OR, IF THE SOFTWARE IS ACCESSED ELECTRONICALLY, SELECT THE "DECLINE" BUTTON AT THE END
OF THIS AGREEMENT.

1. LICENSE TO USE.
Sun grants you a non-exclusive and non-transferable license for the internal use only
of the accompanying software and documentation and any error corrections provided by
Sun (collectively "Software"), by the number of users and the class of computer hardware
for which the corresponding fee has been paid.

2. RESTRICTIONS.
Software is confidential and copyrighted. Title to Software and all
associated intellectual property rights is retained by Sun and/or its licensors.
Except as specifically authorized in any Supplemental License Terms, you may not make
copies of Software, other than a single copy of Software for archival purposes. Unless
enforcement is prohibited by applicable law, you may not modify, decompile, or reverse
engineer Software. You acknowledge that Software is not designed, licensed or intended
for use in the design, construction, operation or maintenance of any nuclear facility.
Sun disclaims any express or implied warranty of fitness for such uses. No right, title
or interest in or to any trademark, service mark, logo or trade name of Sun or its
licensors is granted under this Agreement.

3. LIMITED WARRANTY.
Sun warrants to you that for a period of ninety (90) days from the date of purchase, as
evidenced by a copy of the receipt, the media on which Software is furnished (if any)
will be free of defects in materials and workmanship under normal use. Except for the
foregoing, Software is provided "AS IS". Your exclusive remedy and Sun's entire
liability under this limited warranty will be at Sun's option to replace Software media
or refund the fee paid for Software.

4. DISCLAIMER OF WARRANTY.
UNLESS SPECIFIED IN THIS AGREEMENT, ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS
AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PA
PURPOSE OR NON-INFRINGEMENT ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT THESE DISCLAIM
ARE HELD TO BE LEGALLY INVALID.

5. LIMITATION OF LIABILITY.
TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR
ANY LOST REVENUE, PROFIT OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL
OR PUNITIVE DAMAGES, HOWEVER CAUSED REGARDLESS OF THE THEORY OF LIABILITY, ARISING OU
OR RELATED TO THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF T
POSSIBILITY OF SUCH DAMAGES. In no event will Sun's liability to you, whether in contract,
tort (including negligence), or otherwise, exceed the amount paid by you for Software under
this Agreement. The foregoing limitations will apply even if the above stated warranty
fails of its essential purpose.

6. Termination.
This Agreement is effective until terminated. You may terminate this Agreement at any time
by destroying all copies of Software. This Agreement will terminate immediately without
notice from Sun if you fail to comply with any provision of this Agreement. Upon Termination,

you must destroy all copies of Software.

7. Export Regulations.
All Software and technical data delivered under this Agreement are subject to US export control laws and may be subject to export or import regulations in other countries. You agree to comply strictly with all such laws and regulations and acknowledge that you have the responsibility to obtain such licenses to export, re-export, or import as may be required after delivery to you.

8. U.S. Government Restricted Rights.
If Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in Software and accompanying documentation will be only as set forth in this Agreement; this is in accordance with 48 CFR 227.7201 through 227.7202-4 (for Department of Defense (DOD) acquisitions) and with 48 CFR 2.101 and 12.212 (for non-DOD acquisitions).

9. Governing Law.
Any action related to this Agreement will be governed by California law and controlling U.S. federal law. No choice of law rules of any jurisdiction will apply.

10. Severability.
If any provision of this Agreement is held to be unenforceable, this Agreement will remain in effect with the provision omitted, unless omission would frustrate the intent of the parties, in which case this Agreement will immediately terminate.

11. Integration.
This Agreement is the entire agreement between you and Sun relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification of this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

JAVAHELP(TM) VERSION 2.0
SUPPLEMENTAL LICENSE TERMS

These supplemental license terms ("Supplemental Terms") add to or modify the terms of the Binary Code License Agreement (collectively, the "Agreement"). Capitalized terms not defined in these Supplemental Terms shall have the same meanings ascribed to them in the Agreement. These Supplemental Terms shall supersede any inconsistent or conflicting terms in the Agreement, or in any license contained within the Software.

1. Software Internal Use and Development License Grant.
Subject to the terms and conditions of this Agreement, including, but not limited to Section 4 (Java(TM) Technology Restrictions) of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license to reproduce internally and use internally the binary form of the Software complete and unmodified for the sole purpose of designing, developing and testing your Java applets and applications intended to run on the Java platform ("Programs").

2. License to Distribute Software.
In addition to the license granted in Section 1 (Software Internal Use and Development License Grant) of these Supplemental Terms, subject to the terms and conditions of this Agreement, including but not limited to Section 4 (Java Technology Restrictions), Sun grants you a non-exclusive, non-transferable,

limited license to reproduce and distribute the Software in binary form only, provided that you (i) distribute the Software complete and unmodified and only bundled as part of your Programs, (ii) do not distribute additional software intended to replace any component(s) of the Software, (iii) do not remove or alter any proprietary legends or notices contained in the Software, (iv) only distribute the Software subject to a license agreement that protects Sun's interests consistent with the terms contained in this Agreement, and (v) agree to defend and indemnify Sun and its licensors from and against any damages, costs, liabilities, settlement amounts and/or expenses (including attorneys' fees) incurred in connection with any claim, lawsuit or action by any third party that arises or results from the use or distribution of any and all Programs and/or Software.

3. License to Distribute Redistributables.
In addition to the license granted in Section 1 (Software
Internal Use and Development License Grant) of these Supplemental Terms, subject to the terms and conditions of this Agreement, including but not limited to Section 3 (Java Technology Restrictions) of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license to reproduce and distribute those files specifically identified as redistributable in the Software "README" file ("Redistributables") provided that: (i) you distribute the Redistributables complete and unmodified (unless otherwise specified in the applicable README file), and only bundled as part of your Programs, (ii) you do not distribute additional software intended to supersede any component(s) of the Redistributables, (iii) you do not remove or alter any proprietary legends or notices contained in or on the Redistributables, (iv) you only distribute the Redistributables pursuant to a license agreement that protects Sun's interests consistent with the terms contained in the Agreement, and (v) you agree to defend and indemnify Sun and its licensors from and against any damages, costs, liabilities, settlement amounts and/or expenses (including attorneys' fees) incurred in connection with any claim, lawsuit or action by any third party that arises or results from the use or distribution of any and all Programs and/or Software.

4. Java Technology Restrictions.
You may not modify the Java Platform Interface ("JPI", identified as classes contained within the "java" package or any subpackages of the "java" package), by creating additional classes within the JPI or otherwise causing the addition to or modification of the classes in the JPI.
In the event that you create an additional class and associated API(s) which (i) extends the functionality of the Java platform, and (ii) is exposed to third party software developers for the purpose of developing additional software which invokes such additional API, you must promptly publish broadly an accurate specification for such API for free use by all developers.
You may not create, or authorize your licensees to create, additional classes, interfaces, or subpackages that are in any way identified as "java", "javax", "sun" or similar convention as specified by Sun in any naming convention designation.

5. Java Runtime Availability.
Refer to the appropriate version of the Java Runtime Environment binary code license (currently located at http://www.java.sun.com/jdk/index.html) for the availability of runtime code which may be distributed with Java applets and applications.

6. Trademarks and Logos.
You acknowledge and agree as between you and Sun that Sun owns the SUN, SOLARIS, JAVA, JINI, FORTE, and iPLANET trademarks and all SUN, SOLARIS, JAVA, JINI, FORTE, and iPLANET-related trademarks, service marks, logos and other brand designations ("Sun Marks"), and you agree to comply with the Sun Trademark and Logo Usage Requirements currently located at http://www.sun.com/policies/tradem
Any use you make of the Sun Marks inures to Sun's benefit.

7. Source Code. Software may contain source code that is provided solely for reference purposes

pursuant to the terms of this Agreement. Source code may not be redistributed unless expressly provided for in this Agreement. Some source code may contain alternative license terms that apply only to that source code file.

8. Termination for Infringement.
Either party may terminate this Agreement immediately should any Software become, or in either party's opinion be likely to become, the subject of a claim of infringement of any intellectual property right.

For inquiries please contact: Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, California 95054.
(LFI#135834/Form ID#011801)

Bibliography

[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. . 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, A. Pokahr, and A. Walczak. *Jadex Tutorial*. 2005.

[Jadex Tool Guide] A. Pokahr, L. Braubach, R. Leppin, and A. Walczak. *Jadex Tool Guide*. 2005.

[Jadex User Guide] A. Pokahr, L. Braubach, and A. Walczak. *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. Bor-

dini, 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.