Jadex

User Guide

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

Alexander Pokahr
Lars Braubach

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[1] and Intellij IDEA[2]. 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 world of passive objects. In this respect agents represent active components with individual resoning capabilities. This means that agents can exhibit reactive behavior (responding to external events) as well as pro-active behavior (motivated by the agents own goals).

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[3], 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)[4].
- **Third-Party Libraries.** The Jadex distribution includes a number of third-party libraries. For an accurate list please consult the Appendix E, *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 libraries described below to your class path. The following assumes for simplicity, that you are running Jadex from the console. If you prefer, you can also use your favourite IDE to enter classpath settings and run configurations.

- **`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

[1] http://www.eclipse.org/
[2] http://www.jetbrains.com/idea/
[3] http://vsis-www.informatik.uni-hamburg.de/projects/jadex/
[4] http://www.javasoft.com/

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[5], 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

[5] http://jibx.sourceforge.net/

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 loaded or 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. 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.

2.1. 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 knowledge about 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. For example a game playing agent may choose between a safe action or an action, which is 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 of an 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 facilitates using the BDI model in the context mainstream programming, 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 behavior. 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 2.1, “Jadex Abstract Architecture”.

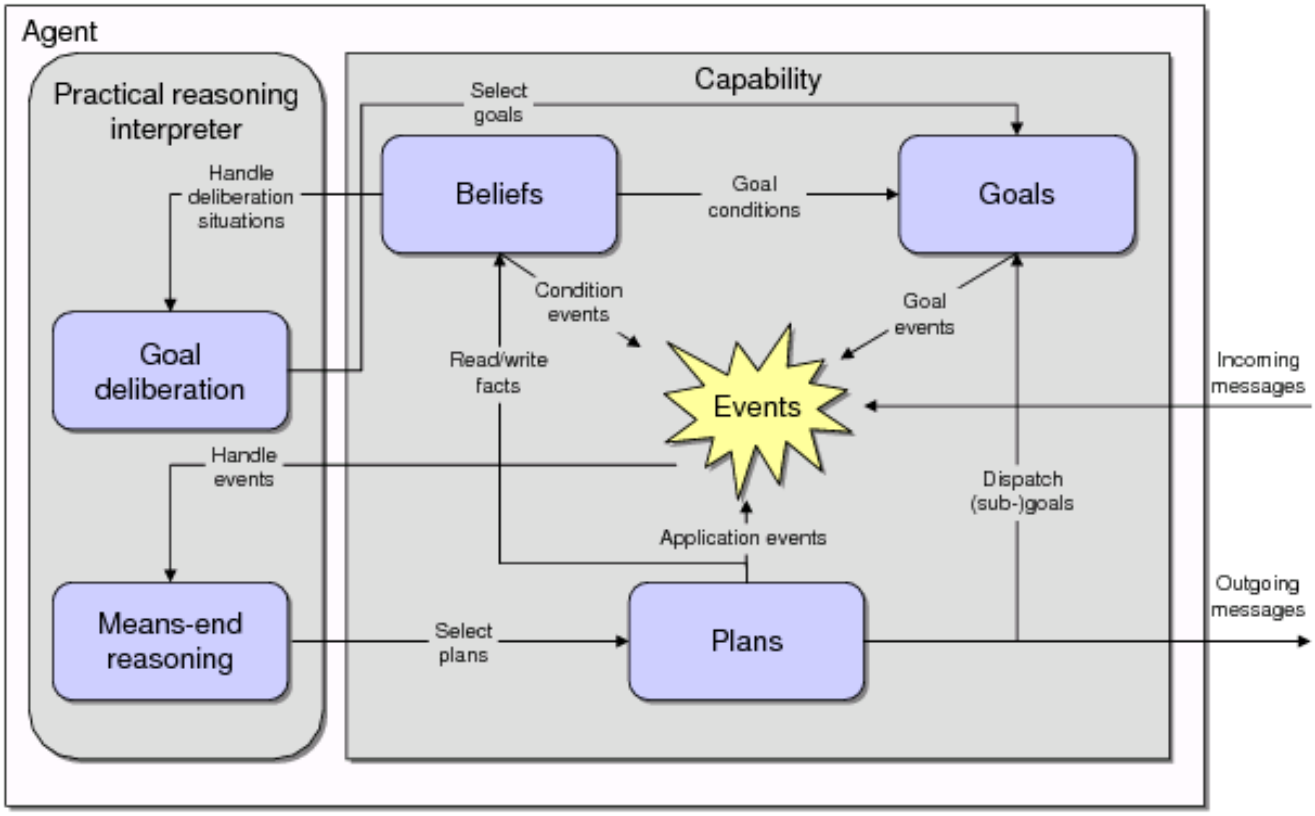

Figure 2.1. Jadex Abstract Architecture

Reasoning in Jadex is a process consisting of two interleaved components. On the one hand, the agent reacts to incoming messages, internal events and goals by selecting and executing plans (means-end reasoning). On the other hand, the agent continuously deliberates about its current goals, to decide about a consistent subset, which should be pursued.

The main concepts of Jadex are beliefs, goals and plans. The beliefs, goals and plans of the agent are defined by the programmer and prescribe the behavior of the agent. E.g., the current beliefs influence the deliberation and means-end reasoning processes 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.

2.1.1. 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 eval-

uated dynamically on demand.

2.1.2. 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 2.2, “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.

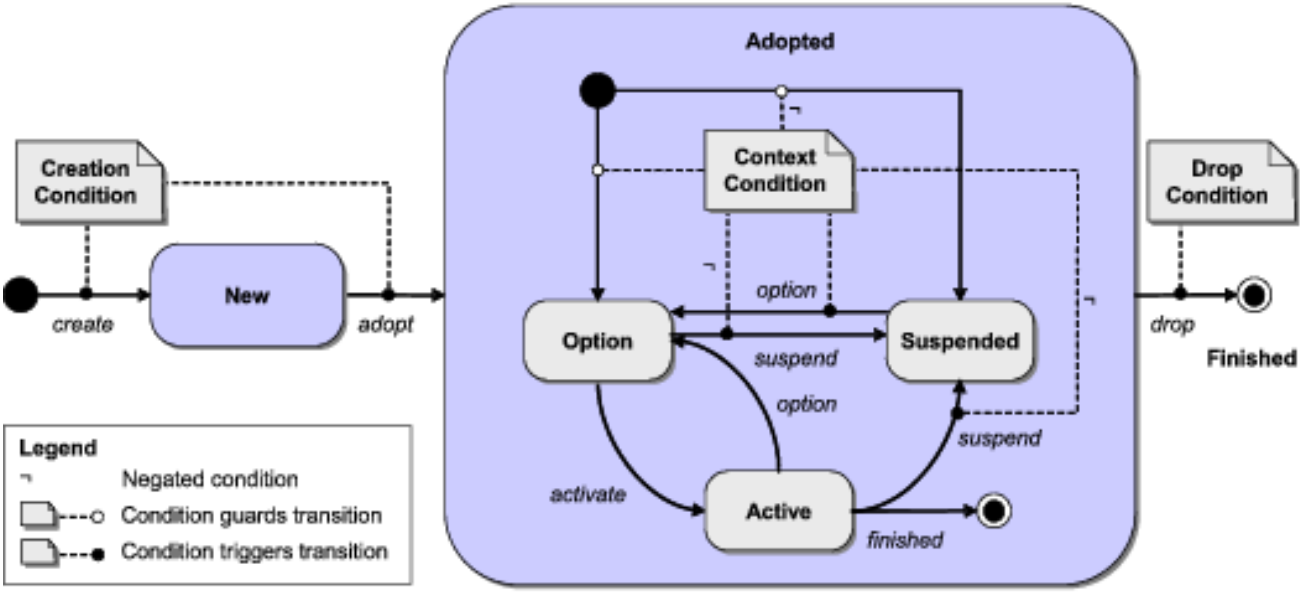

Figure 2.2. Goal Lifecycle

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.

2.1.3. Plan Specification

The concrete actions an agent may carry out to reach its goals are described 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 or goal occurs. In addition, so called initial plans get executed when the agent is born. During the execution of the plan body, running plans may not only execute arbitrary Java code but can also dispatch subgoals and wait for events to occur.

2.2. Agent Definition

The complete definition of an agent is captured in a so called *agent definition file* (ADF). The ADF is an XML file, which contains all relevant properties of an agent (e.g. the beliefs, goals and plans). In addition to the XML tags for the agent elements, the developer can use expressions in a Java-like syntax for specifying belief values and goal parameters. 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 BlackJack (`src/jadex/examples/blackjack`) share `Player.agent.xml` as their definition file.

For each element ADF in the all important properties can be defined as attributes or subtags. For example, plans are declared by specifying how to instantiate them from their Java class (body tag), and a trigger (e.g. event) can be stated, that determines under which conditions a plan gets executed. Moreover, in the ADF, the initial state of an agent (how the agent should look like, when it is born) is determined in a so called configuration, which defines the initial beliefs, initial goals, and initial plans.

2.3. Execution Model of a Jadex Agent

This sections shows the operation of the reasoning component, given the Jadex BDI concepts (see Figure 2.3, “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 cus-

tomizable 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.

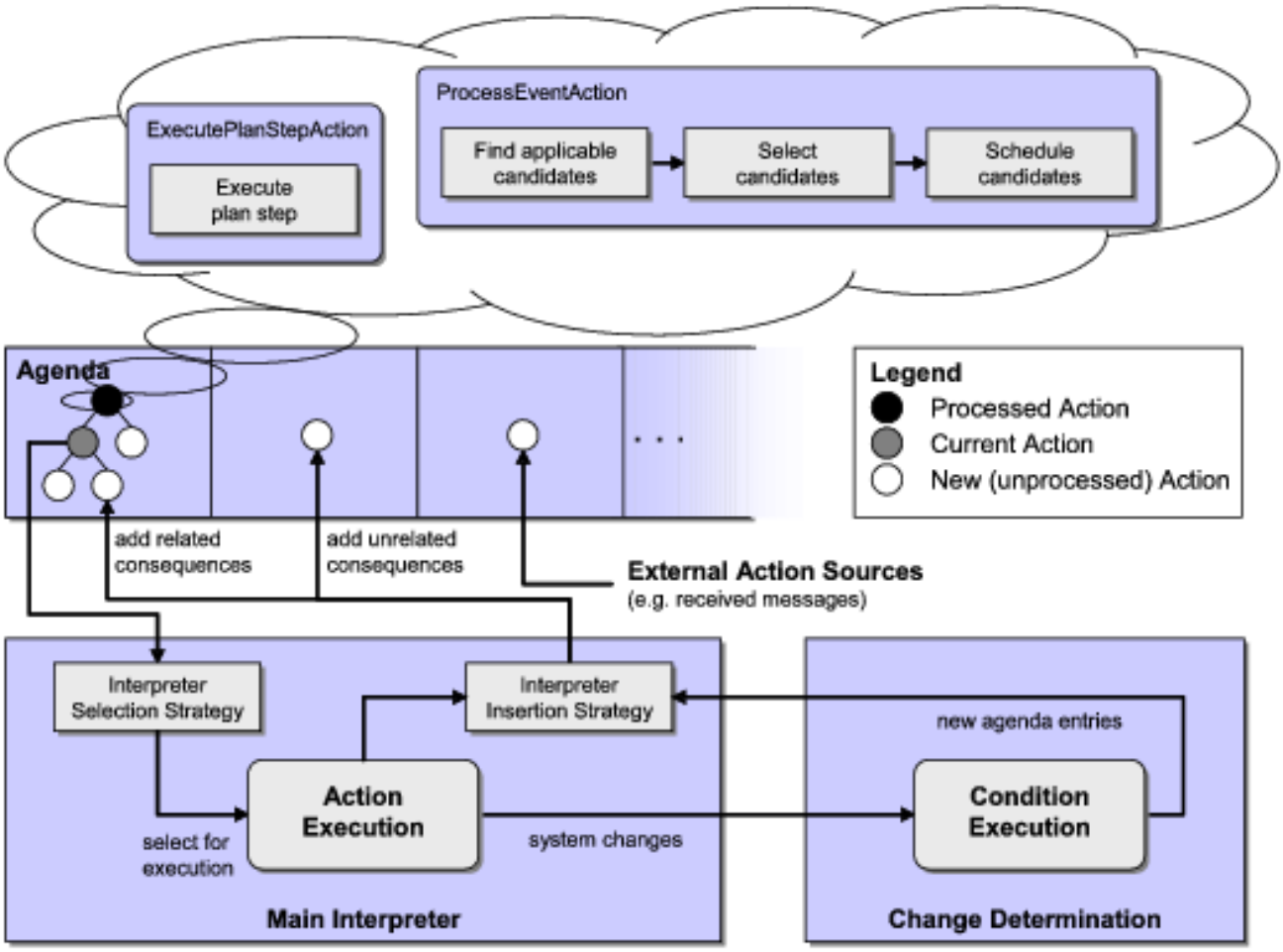

Figure 2.3. Jadex Execution Model

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 2.3, “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 7.7, “ Goal Deliberation with "Easy Deliberation" ” which is realized with extended meta-actions.

Chapter 3. 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].

3.1. 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 Buyer agents (from the booktrading example) are defined by the `Buyer.agent.xml` file, and use plans implemented, e.g. in the file `PurchaseBookPlan.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 3.1, “ 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.

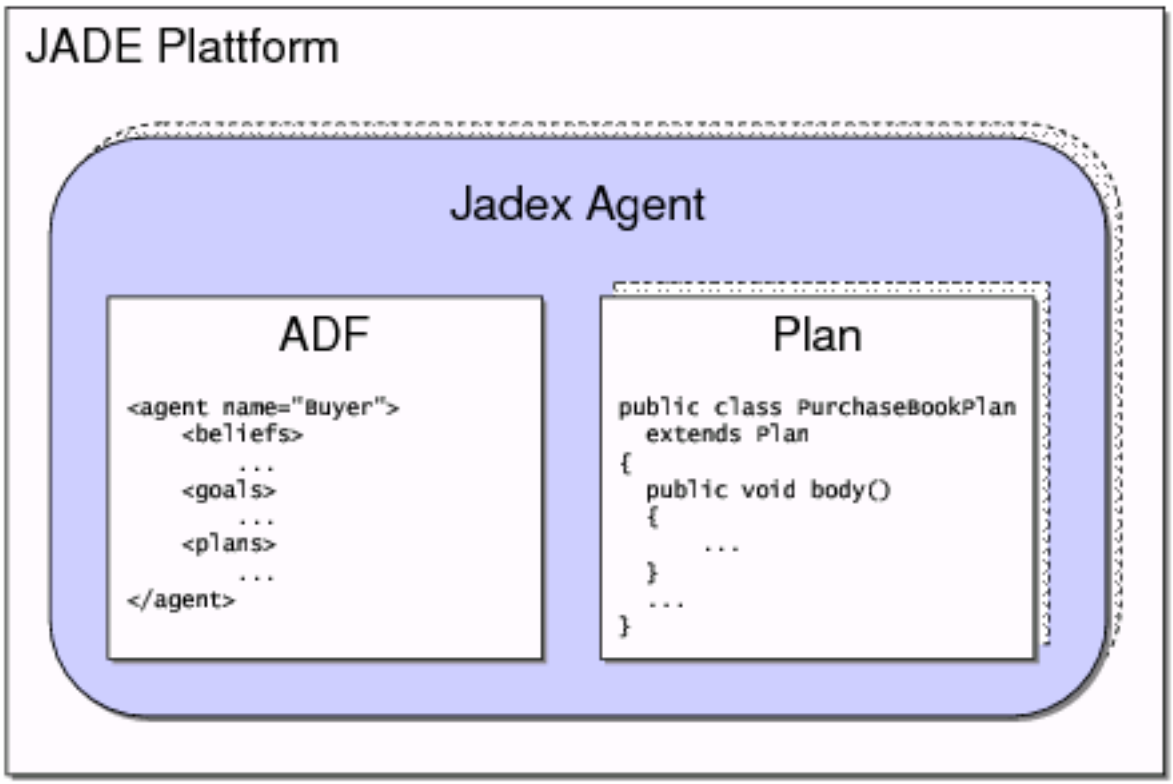

Figure 3.1. Components of a Jadex agent

3.2. Structure of Agent Definition Files (ADFs)

The head of an ADF looks like shown in Figure 3.2, “Header of an agent definition file”. First, the agent tag specifies that the XML document follows the `jadex-0.96.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 specified 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 4, *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 behavior of the Jadex engine can use the properties attribute to provide their own property XML file with customized settings (see Chapter 12, *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-0.96.xsd"
       name="Buyer"
       package="jadex.examples.booktrading.buyer">
       ...
</agent>
```

Figure 3.2. Header of an agent definition file

Figure 3.3, “ 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 <configurations> section, predefined configurations containing, e.g., initial beliefs, goals, and plans, as well as end goals and plans are specified.

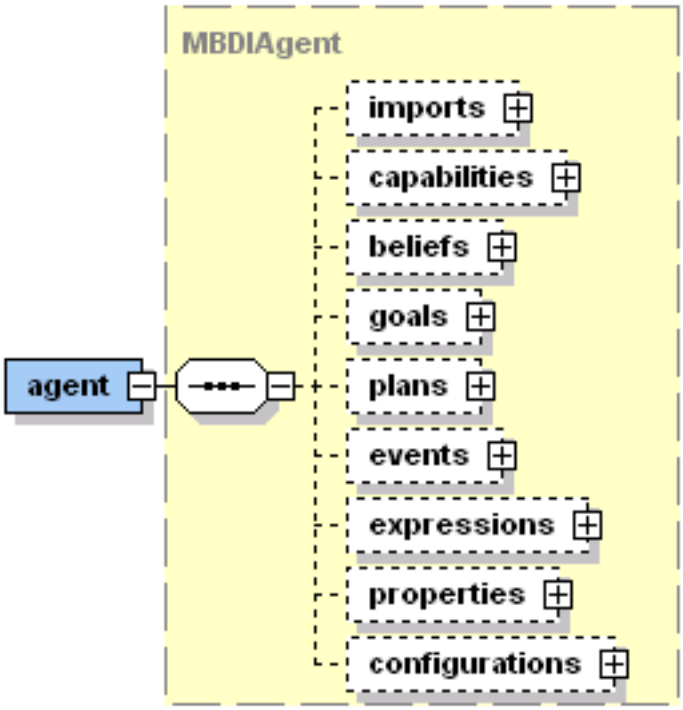

Figure 3.3. Jadex agent XML schema

It should be noted that, unless otherwise stated, the order of occurrence of the elements is prescribed by the underlying XML Schema. Therefore, you cannot, e.g., declare plans before beliefs. Throughout this user guide figure like Figure 3.3, “ Jadex agent XML schema ” will always denote the correct order of element appearence (from top to bottom). Of course, it is possible to omit those elements, which are not required for your agent.

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 buyer agent will instantiate new purchase book plans (`IPlan`) for each book to be bought, 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 4. 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.

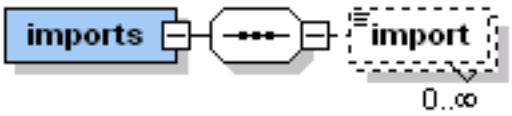

Figure 4.1. The Jadex imports XML schema part

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.

4.1. 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 4.2. Example import declaration and usage

Chapter 5. 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 5.1, “Capability concept”).

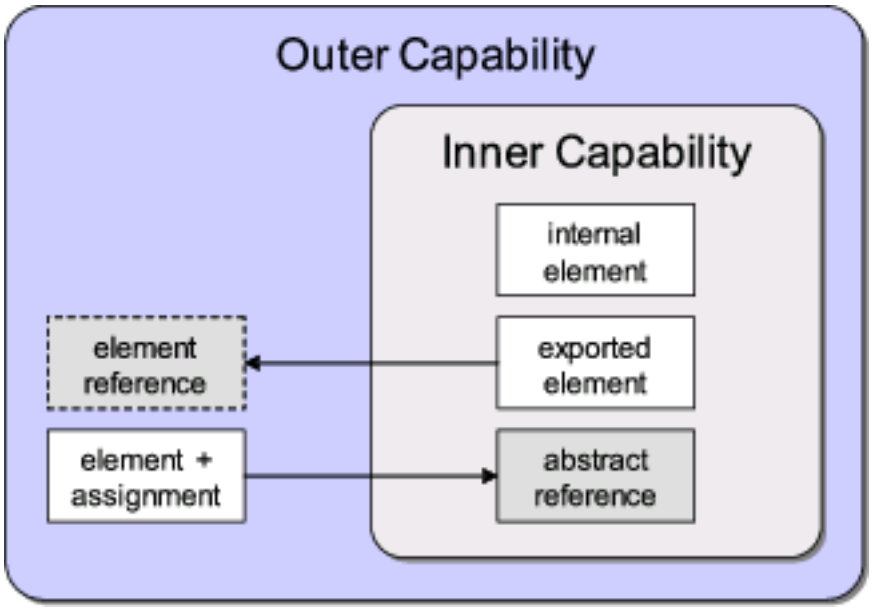

Figure 5.1. Capability concept

5.1. 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 3.2, “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-0.96.xsd"
       name="MyCapability"
       package="mypackage">
```

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

Figure 5.2. Capability XML file header

5.2. 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 5.3. Including subcapabilities

5.3. 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 5.1, “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 5.4, “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.

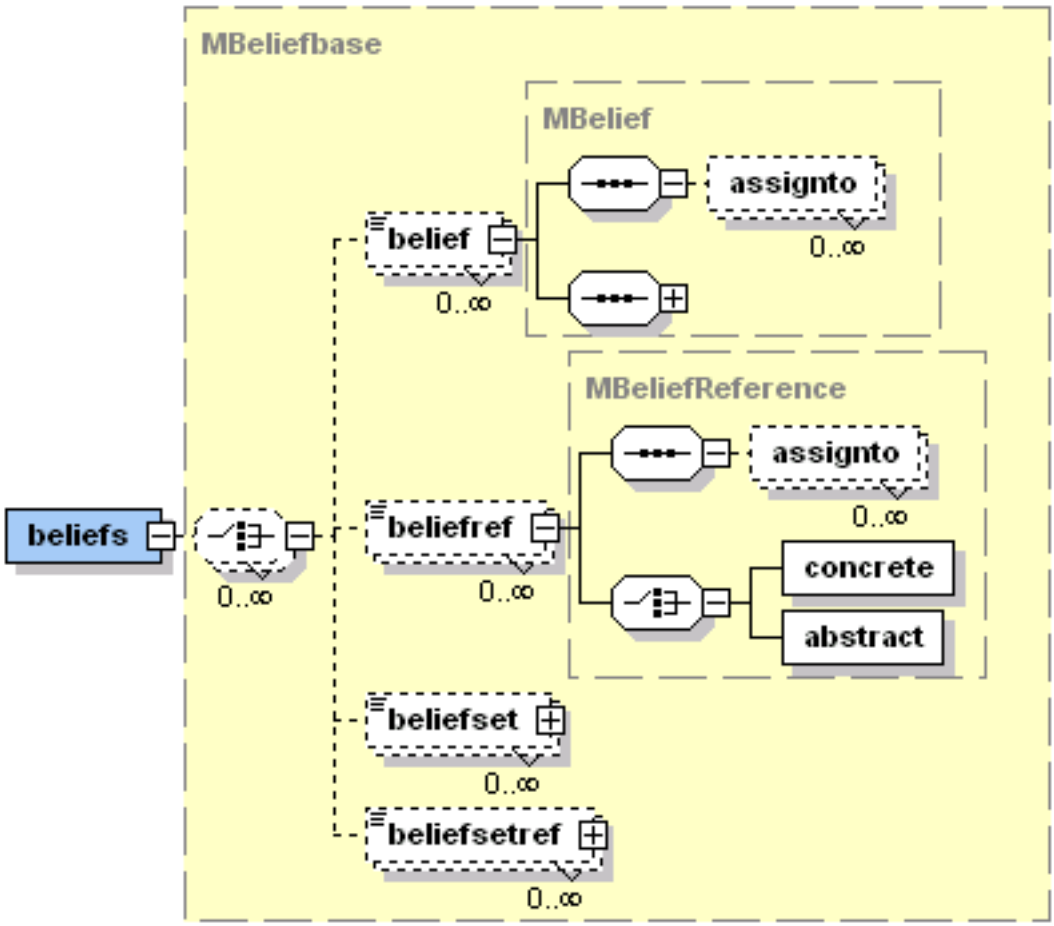

Figure 5.4. The Jadex references XML schema elements (using beliefs as example)

5.3.1. 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" exported="true" class="MyFact"/>
```

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

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

5.3.2. 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" exported="true" class="MyFact">
    <abstract/>
</beliefref>
```

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

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

5.4. Weak vs. Strong Export

By default, elements of an outer capability behave the same, regardless if they are references to concrete inner elements or concrete elements assigned to abstract inner elements. Sometimes one wants to distinguish between e.g. goals, created inside a capability and goals created from the outside. When using concrete elements in the inner capability (and therefore references in the outer capability) you can choose between strong export (`exported="true"`) and weak export (`exported="shielded"`). When a weakly exported element is instantiated inside a capability references in the outer capability will *not* be created. The strong export, on the other hand, will always instantiate the complete reference structure of an element. For most use cases, the strong export will be appropriate.

Chapter 6. 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.

6.1. 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 6.1, “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 6.2, “Example belief definition”). References to beliefs and belief sets from inner capabilities can be defined using the `<beliefref>` and `<beliefsetref>` tags (cf. Section 5.3, “Elements of a Capability”).

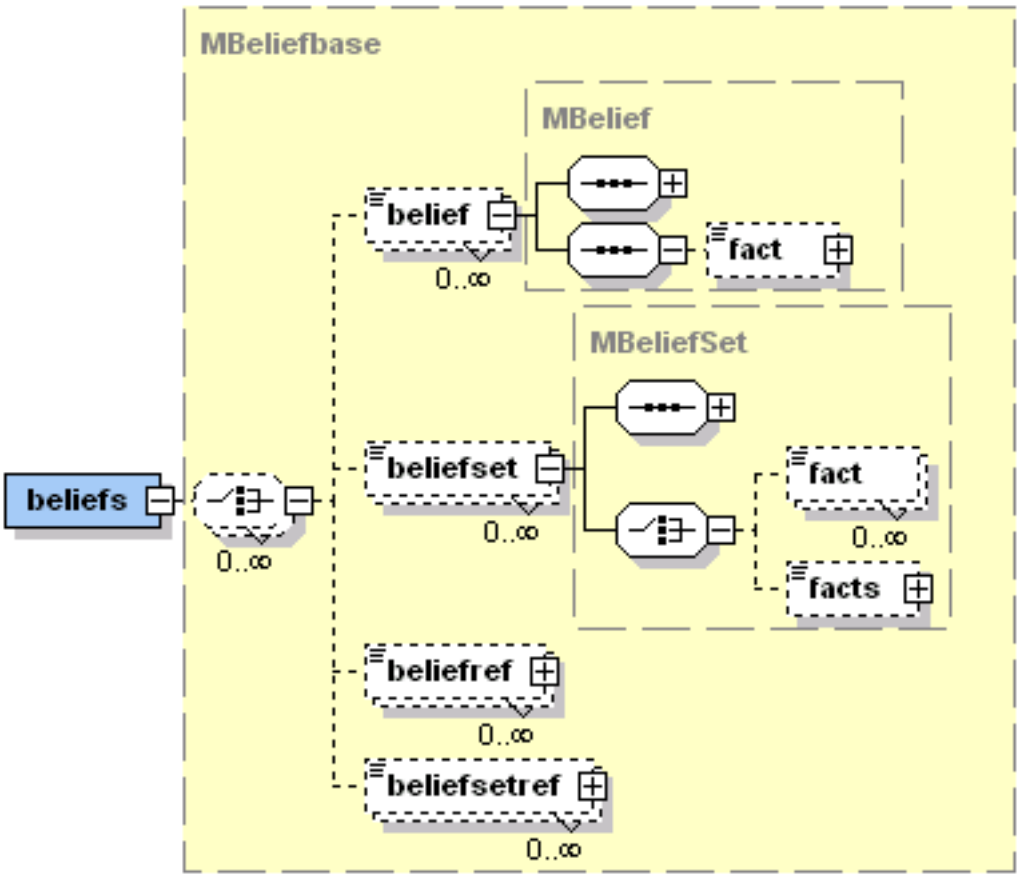

Figure 6.1. The Jadex beliefs XML schema part

```
<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 6.2. Example belief definition

6.2. 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 6.3. A simple example of using a boolean belief

6.3. 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 behavior of the fact expression can be adjusted using the `evaluationmode` attribute as further described in Section 10.2, “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 in continuous intervals. 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 6.4. Examples of dynamically evaluated beliefs

6.4. Propagation of Belief Changes

To monitor conditions (cf. Chapter 11, *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 how to implement this functionality inside a Java bean is shown below.

```
import java.beans.PropertyChangeSupport;
import java.beans.PropertyChangeListener;

public class Location {
    private int x, y;
    private PropertyChangeSupport pcs;

    public Location(int x, int y) {
        this.x = x;
        this.y = y;
        this.pcs = new PropertyChangeSupport(this);
    }

    public int getX() {
        return this.x;
    }
    public void setX(int x) {
        int old = this.x;
        this.x = x;
```

```
        this.pcs.firePropertyChange("X", old, this.x);
    }

    public int getY() {
        return this.y;
    }
    public void setY(int y) {
        int old = this.y;
        this.y = y;
        this.pcs.firePropertyChange("Y", old, this.y);
    }

    public void addPropertyChangeListener(PropertyChangeListener listener) {
        pcs.addPropertyChangeListener(listener);
    }
    public void removePropertyChangeListener(PropertyChangeListener listener) {
        pcs.removePropertyChangeListener(listener);
    }
}
```

Chapter 7. 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 2.1.2, “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 behavior 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 7.1, “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 5.3, “Elements of a Capability”).

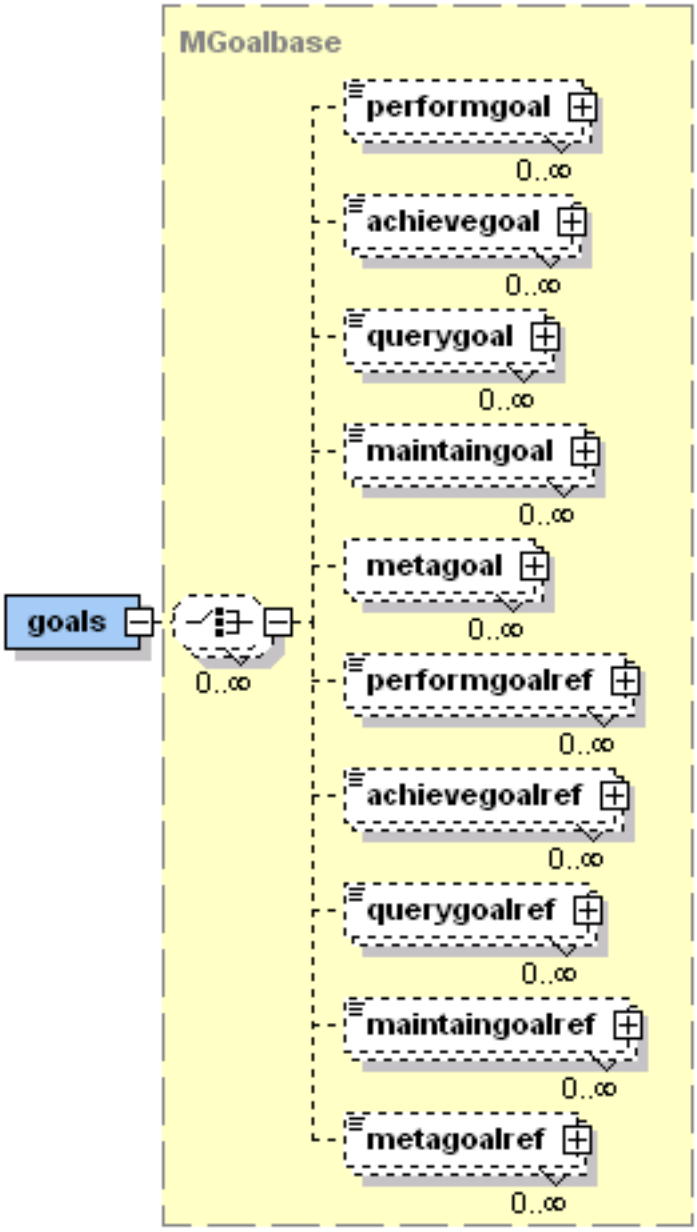

Figure 7.1. The Jadex goals XML schema part

At runtime an agent may have any number of top-level goals, as well as subgoals (i.e. belonging to some plan). 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 a running plan. 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.

7.1. Common Goal Features

In Figure 7.2, “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.

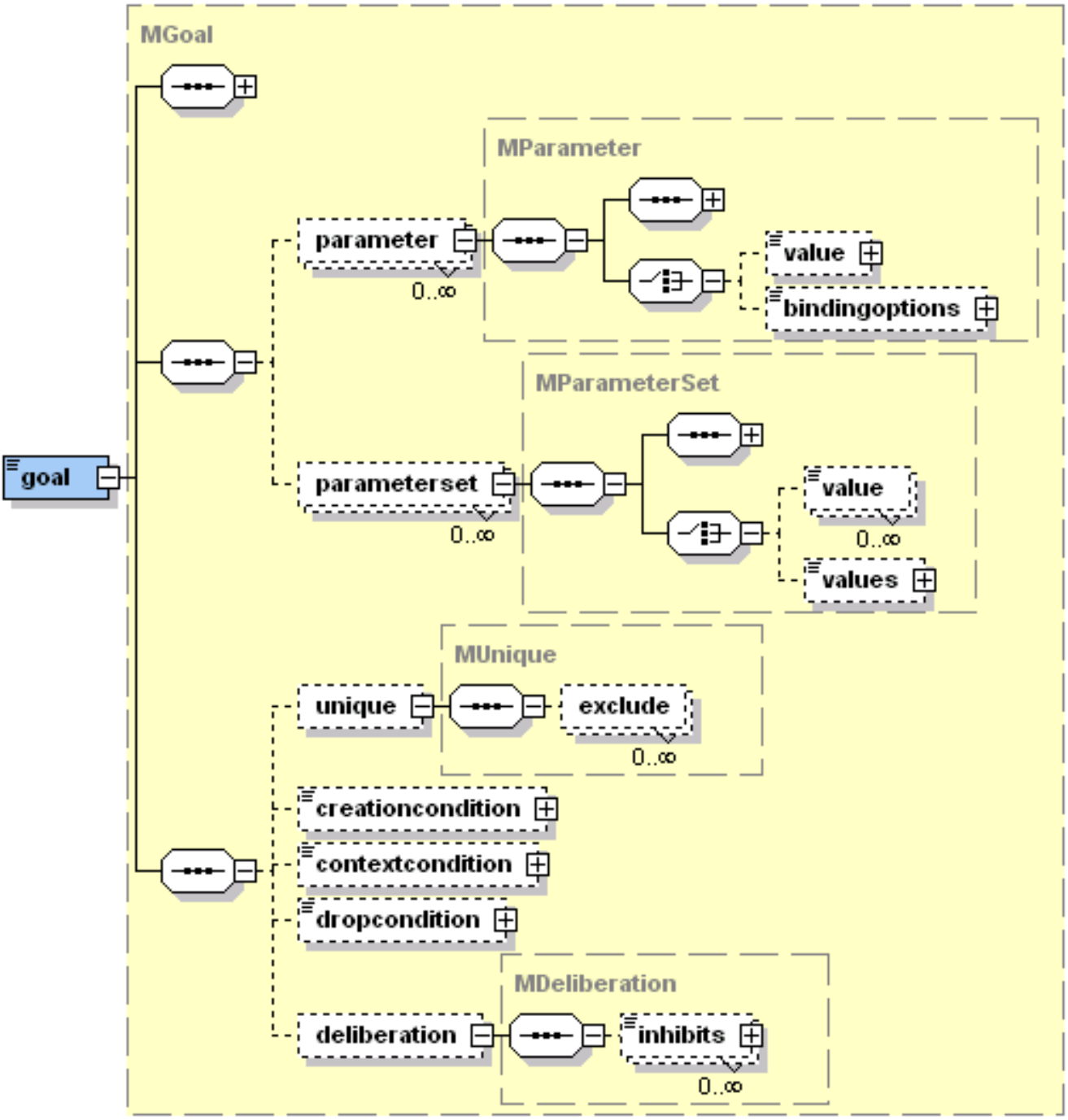

Figure 7.2. The Jadex common goal features

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 behavior 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 8.2.1, “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 7.7, “ Goal Deliberation with "Easy Deliberation" ”.

7.1.1. Example Goal

Figure 7.3, “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 7.3. Example goal (taken from Hunter-Prey scenario)

7.1.2. BDI Flags

The handling and the exposed behavior of goals can be adapted to the requirements of your application using the so called BDI flags as depicted in Table 7.1, “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 7.1. Common goal attributes (BDI flags)

Name	Default	Possible Values
`retry`	`true`	{ `true`, `false`}
`retrydelay`	`0`	positive `long` value
`exclude`	`"when_tried"`	{ `"when_tried"`, `"when_succeeded"`, `"when_failed"`, `"never"`}
`posttoall`	`false`	{ `true`, `false`}
`randomselection`	`false`	{ `true`, `false`}

Name	Default	Possible Values
metalevelreasoning	true	{ true, false}
recur	false	{ true, false}
recurdelay	0	positive long value

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 7.8, “Meta Goal”.

Furthermore the goal introduce the recur flag and the recurdelay (in milliseconds) option as further BDI settings. Consider a goal to have failed after trying all available plans. Setting recur to true, this goal will not be dropped but try to execute again, when the specified delay has elapsed. For recurring goals, all plans are considered again, i.e. recur allows to completely restart the reasoning for a goal after some delay.

7.2. 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 obtain the desired behavior. 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 7.4. Example perform goal

7.3. 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 7.5. Example achieve goal

7.4. Query Goal

Query goals can be used to retrieve specified information. From the specification and runtime behavior'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 7.6. Example query goal

7.5. Maintain Goal

Maintain goals allow a specific state to be monitored and whenever this state gets violated, the goal has the pur-

pose 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.

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 `"&gt;"` 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 7.7. Example maintain goal

7.6. 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 configuration of an agent or capability, and are directly created and dispatched as top-level goals when an agent is born or terminated (cf. Section 13.3, “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 interactions (cf. Chapter 15, *External Interactions*).

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 7.8. 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 8.2.1, “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 7.9. Handling of failed subgoals

7.7. 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 2.2, “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 7.10, “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.

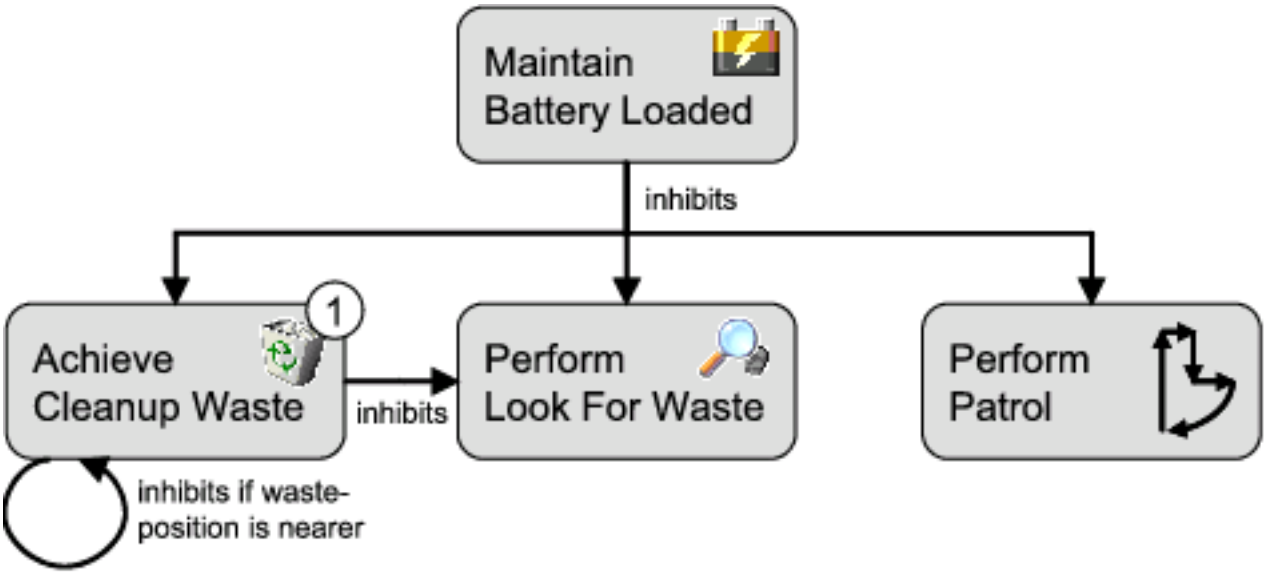

Figure 7.10. Example goal dependencies (taken from Cleanerworld scenario)

The dependencies can be naturally mapped to the goal specifications in the ADF (see Figure 7.11, “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 7.11. Example goals (taken from Cleanerworld scenario)

7.8. 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.

With the trigger tag, it is specified for which kind of event or goal the meta goal should be activated. Possible meta goal triggers are shown in Figure 7.12, “Meta goal trigger tag”. As can be seen, meta goals can be used to select among applicable plans for an internal event, message event, a goal finished event, and a new goal to process. Any number of these triggers can appear inside a meta goal specification, i.e., a meta goal can be used to control meta-level reasoning of more than one event type. Each triggering element can be further described using a match expression, which can be used, e.g., to match only elemens with given parameter values. For backwards compatibility it is also possible to specify the triggering events in form of a single filter expression. This should only be needed in very special cases and should otherwise be avoided, because support for filter expressions might be dropped in future releases of Jadex.

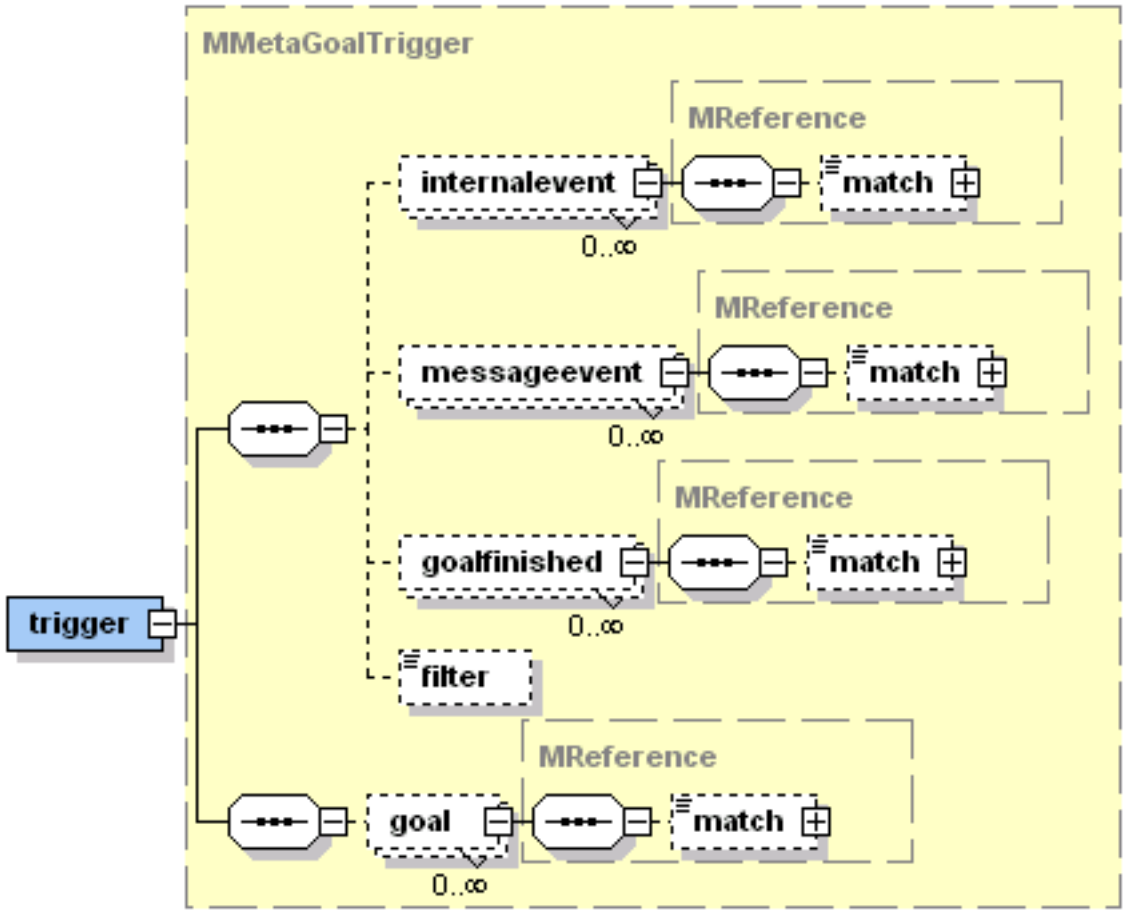

Figure 7.12. Meta goal trigger tag

Besides the declaration of a triggering goal or event, the specification of a meta goal requires including 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 7.14, “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">
        <parameterset name="applicables" class="ICandidateInfo"/>
        <parameterset 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 7.13. 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().getParameter("move").getValue();
        ...
    }

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

Figure 7.14. Body of the ChooseMovePlan

Chapter 8. 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.

8.1. Defining Plan Heads in the ADF

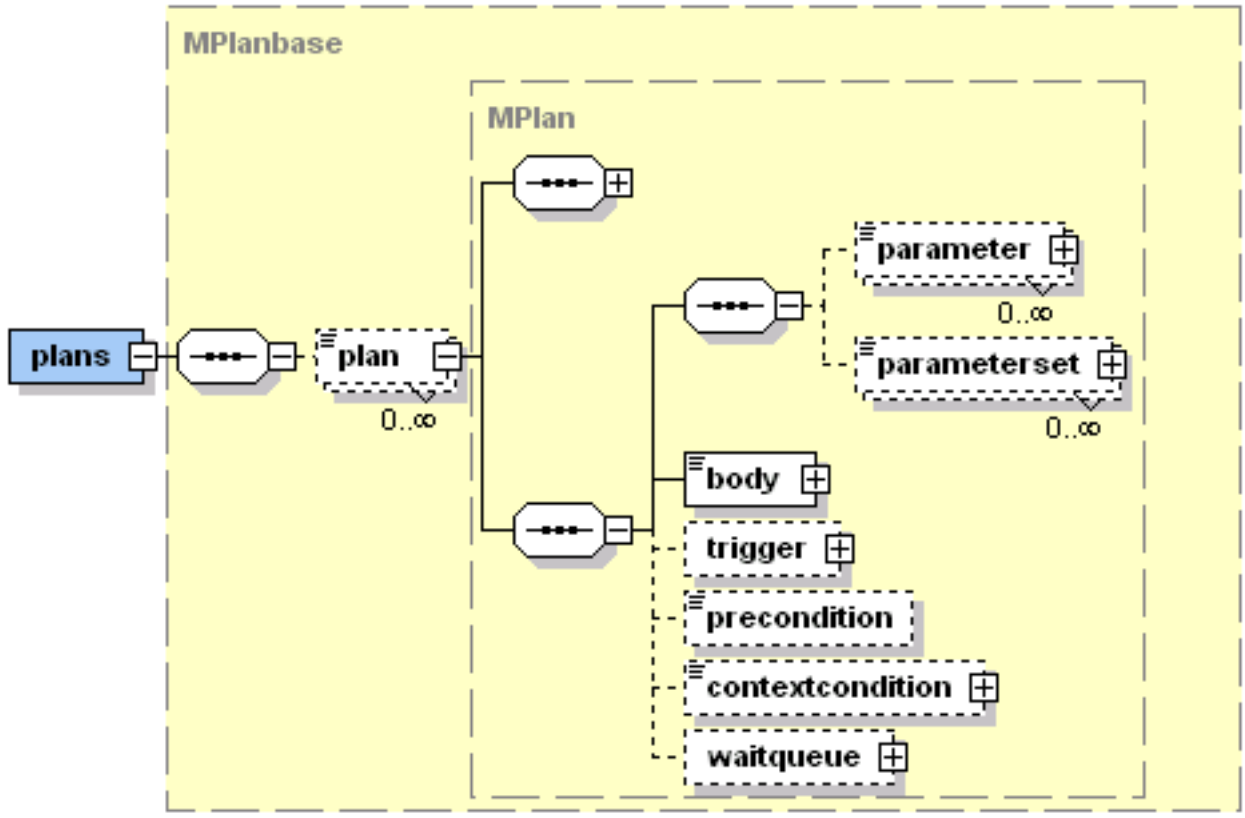

Figure 8.1. The Jadex plans XML schema part

In Figure 8.1, “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 8.1, “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 8.1. 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 8.2, “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 8.2. A plan reacting on a ping message

8.1.1. 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 8.3, “The Jadex plan trigger XML schema part”). Its subtags specify the internal-, message, or goal events for which the plan is applicable. For goals, it is distinguished between plans triggered for handling a goal (`<goal>` tag) and plans reacting on the completion of a goal (`<goalfinished>` tag). The `<goal>` tag indicates that a goal has been newly activated, while the `<goalfinished>` tag corresponds to a goal that has been dropped. These events or goals can be further restricted, by specifying a match expression. When a match expression is included in the trigger element, the plan will only be selected for those goal or event instances, for which the expression evaluates to true. 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.

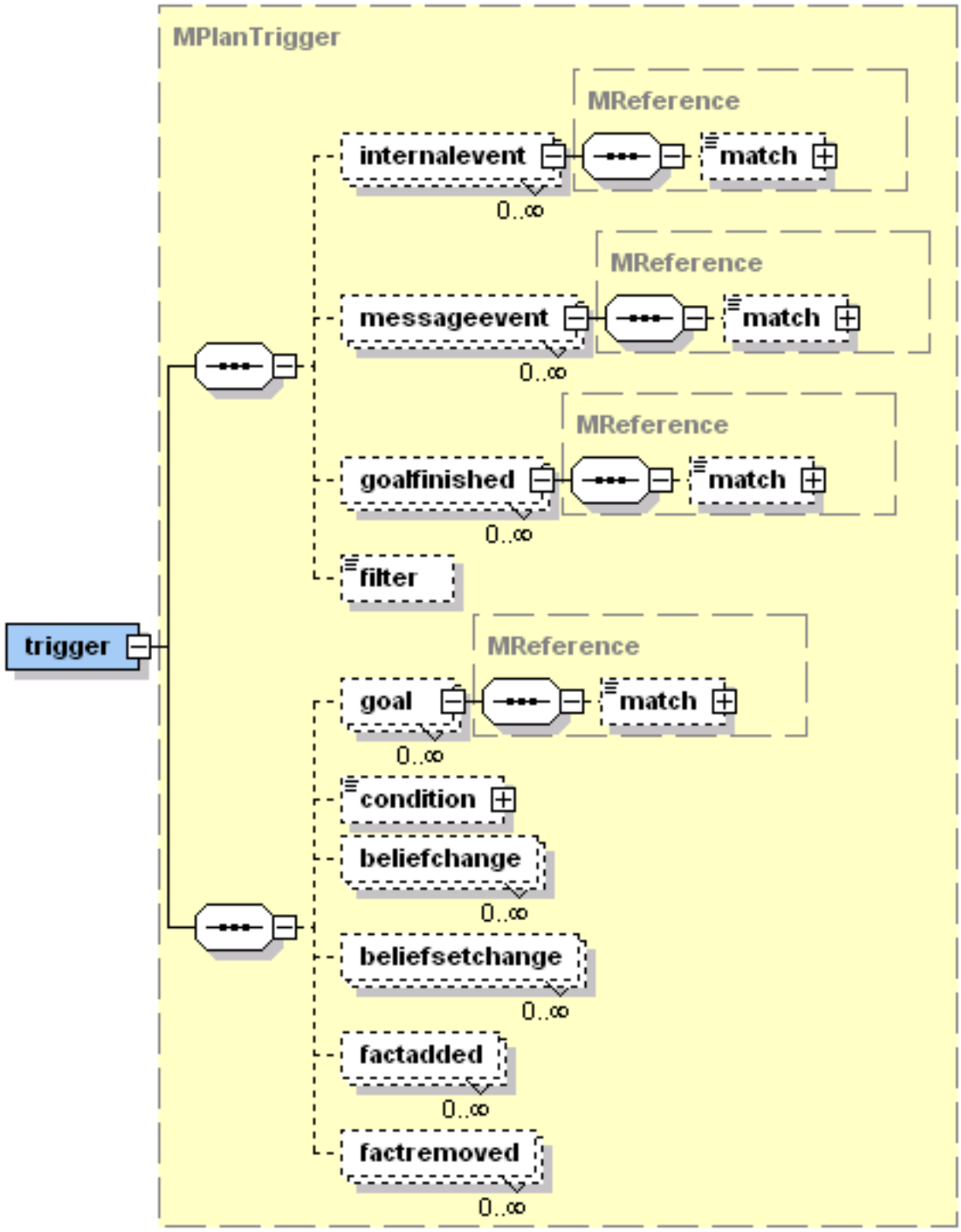

Figure 8.3. The Jadex plan trigger XML schema part

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.

8.1.2. 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 8.4. Example of a plan with context condition

8.1.3. 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 still has to execute 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.

8.1.4. 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 7.1, “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 with exception of the binding parameters. As binding parameters are evaluated before plan instantiation the value of a binding parameter can be accessed directly via its name (without prepending $plan).

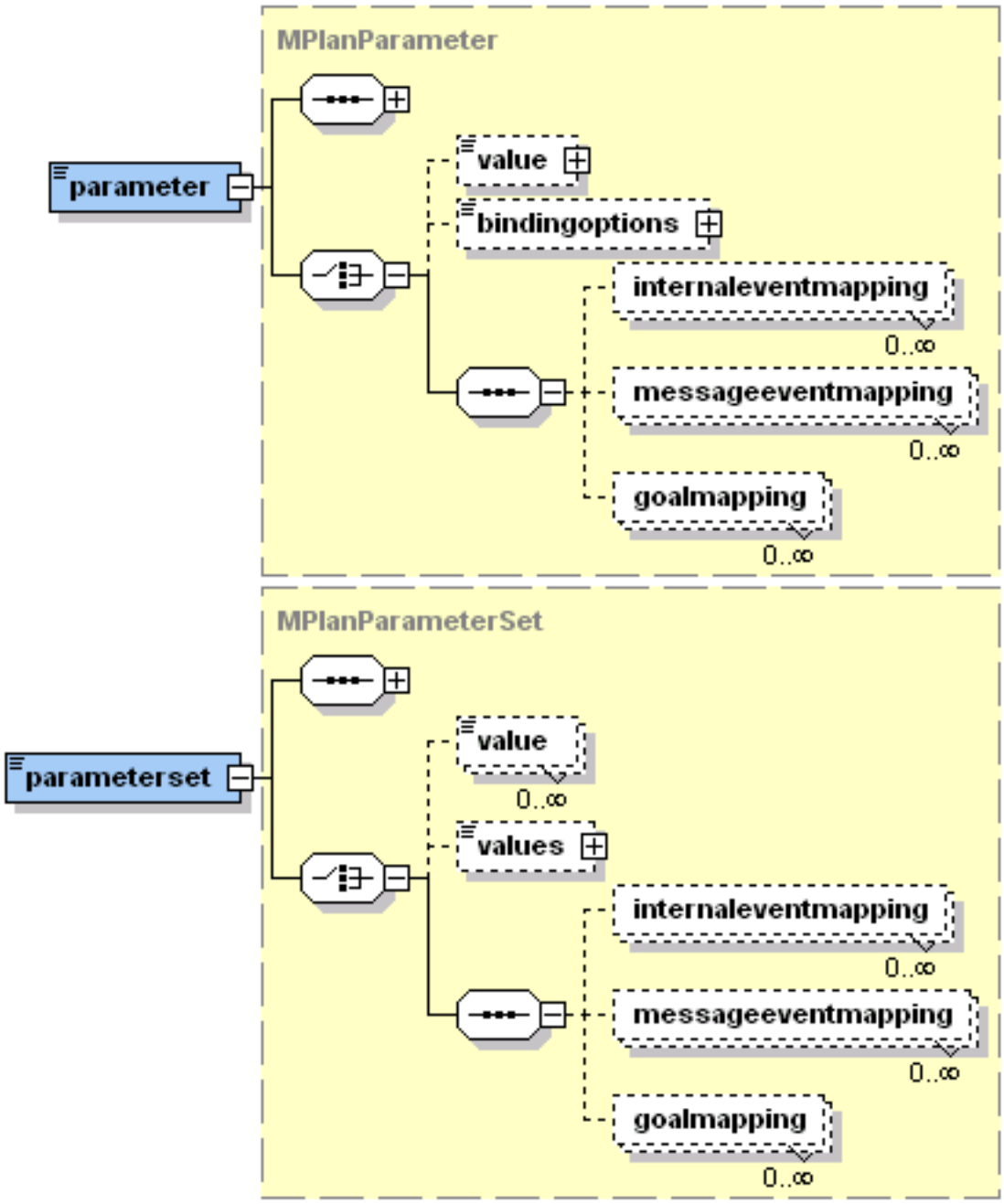

Figure 8.5. The Jadex plan parameters XML schema part

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 8.6, “ 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 [http://en.wikipedia.org/wiki/Cartesian_product])

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 8.6. 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 can also automatically be written back to a goal (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.

8.2. 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 2.3, “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() {
    // Send request.
    ...

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

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

        ...
        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 8.7. Programming style of the different plan types

8.2.1. 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 8.8, “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, but expect an `IEvent` parameter like the `action()` method. 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 8.8. 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 7.6, “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 15, *External Interactions*).

8.2.2. 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 8.9. How to establish an atomic block

Chapter 9. Events

An important property of agents is the ability to react timely to different kinds of events. Jadex supports two kinds of application-level events, which can be defined by the developer in the ADF. Internal events can be used to denote an occurrence inside an agent, while message events represent a communication between two or more agents. Events are usually handled by plans. For example the ping plan gets triggered when a ping request message arrives. 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 12, *Properties*).

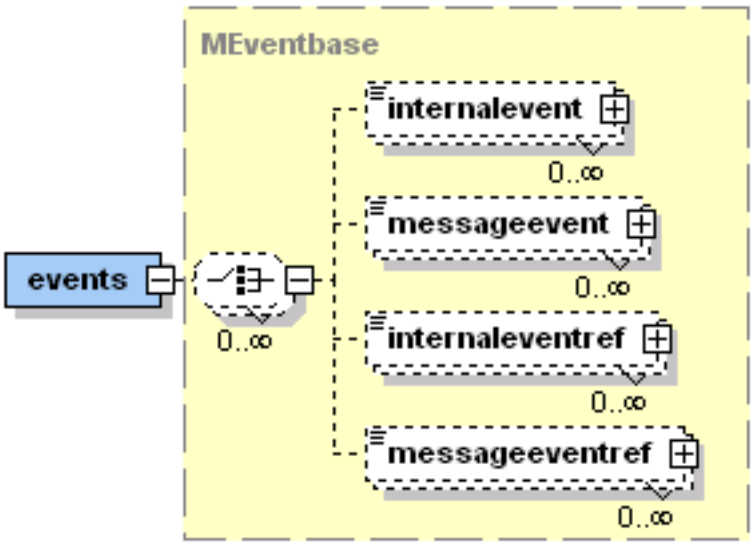

Figure 9.1. The Jadex events XML schema part

Two kinds of events are supported in Jadex: Message events and internal events, as well as references to these types, as shown in Figure 9.1, “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. Unlike goals, events are single points in time and therefore only support "in" parameters, which denote the "source to consumer"-direction of value passing. 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. 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 9.1. 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` and `jadex.runtime.IInternalEvent` interfaces. The following sections will de-

scribe these different types of events in more detail.

9.1. 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 9.2. 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 9.2, “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, because they will be passed to the plan body's action method (information about mobile plans can be found in Section 8.2, “Implementing a Plan Body in Java”)

Table 9.2. Predefined Internal Event Types

Predefined Types	Description
jadex.model.IMEventbase.TYPE_TIMEOUT	A timeout has occurred (generated, when waiting for a fixed time)
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

9.2. 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.

The message passing mechanism is based on using `jadex.adapter.fipa.AgentIdentifiers` for the unambiguous identification of agents. An agent identifier hence contains an agents globally unique name which consists of a local part followed by the "@" character and the platforms name (schema: <agentname>@<platformname>, example: ams@lars). In addition to the name an agent identifier can contain additional information. On the one hand arbitrary many transport addresses might be present. These addresses can be used to contact the agent from a remote platform and normally represent the address of platform wide transport mechanisms (schema: <transportname>://<address>, example: nio-mtp://mypc18:9876). Besides the transport addresses also so called resolvers can be part of an agent identifier. An agent resolver is itself another agent identifier which can be used for name resolution, i.e. if an agent wishes to send a message to another agent and wants to fetch up-to-date transport addresses of the receiving agent it can query the resolver(s) for additional transport addresses. Please refer to the FIPA Agent Management Specification[1]. An agent identifier of another agent can be obtained in two ways. If all details about the agent are known an agent identifier can directly be created using a local or global name. E.g., `new AgentIdentifier("Heinz", true)` would create an identifier to contact agent "Heinz" on the local platform. If the details of an agent are not known in advance, an agent may search for other agents using either the AMS listing all agents on a platform or the DF, which allows to search for agents providing a given service. Searching AMS and DF is explained in detail in Chapter 16, *Using Predefined Capabilities*

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 9.3, “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 FIPA ACL Message Structure Specification[2]. In addition to the FIPA parameters, Jadex introduces the `content-start`, `content-class`, and `action-class` parameters. The meanings of all of these parameters are explained in the following subsections.

Table 9.3. Reserved FIPA message event parameters

Name	Class	Meaning
`performative`	`String`	Speech act of the message that expresses the senders intention towards the message content. You can use the constants from `jadex.adapter.SFipa.{ACCEPT_PROPOSAL, AGREE, ...}`
`sender`	`AgentIdentifier`	The senders agent identifier, which contains besides other things its globally unique name.

[1] http://www.fipa.org/specs/fipa00023/SC00023J.html
[2] http://www.fipa.org/specs/fipa00061/SC00061G.html

Name	Class	Meaning
reply-to	AgentIdentifier	The agent identifier of the agent to which should be replied.
receivers [set]	AgentIdentifier	Arbitrary many (at least one) agent identifier of the intended receiver agents.
content	Object	The content (string or object) of the message. If the content is an object it has to be specified how the content can be marshalled for transmission. For this puropose codecs are used. Jadex has built in support for marshalling arbitrary Java beans via setting the language of the message to jadex.adapter.fipa.SFipa.NUGGETS_XML.
language	String	The language in which the content of the message should be encoded.
encoding	String	The encoding of the message.
ontology	String	The ontology that can be used for understanding the message content. Can also be used for deciding how to marshal the content.
protocol	String	The interaction protocol of the the message if it belongs to a conversation. There are constants available for the predefined FIPA interaction protocols in jadex.adapter.SFipa.PROTOCOL_{REQUEST, QUERY, ...}
reply-with	String	Reply-with is used for assigning a reply to a original message. The receiver of the message should respond to this message by putting the reply-with value in the in-reply-to field of the answer. Unique ids can e.g. be generated via the method SFipa.createUniqueId().
in-reply-to	String	Used in reply messages and should contain the reply-with content of the answered message.
conversation-id	String	The conversation-id is used in interactions for identifying messages that belong to a specific conversation. All messages of one interaction should share the same conversation-id. Unique ids can e.g. be generated via the method. SFipa.createUniqueId().
reply-by	Date	The reply-by field can contain the latest time for a response message.
content-start [non-fipa]	String	Can be used for easy matching of string value. It is checked if the value of the content start is equal to the beginning of the string content of the message.
content-class [non-fipa]	Class	The content-class can be used for matching and tests if the class of the content object equals the specified class.
action-class [non-fipa]	Class	The action-class can be used for matching and checks if the class of the inner agent action (which might be contained in another action concept e.g. in FIPA SL) matches the given class.

9.2.1. 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. In addition to fixed parameter values, message matching can be fine-tuned by using a match expression that can be specified for each message event. As shown in the second example below, in the match expression the parameters of a message can be accessed by prepending a "$" before the parameter name. Additionally, it is not allowed having variable names in Java that contain a "-" character as this is interpreted as minus. Therefore, in all parameter(set)s names the "-" characters have been replaced by a "_" character. This means you need to write e.g. "$reply_with" instead of "$reply-with".

A message event type matches an incoming message if all fixed parameter values are the same in the received message and the match expression evaluates to true.

```
<imports>
    <import>jadex.adapter.fipa.SFipa</import>
</imports>
...
<events>
    <!-- A query-ref message with content "ping" -->
    <messageevent name="query_ping" type="fipa" direction="receive">
        <parameter name="performative" class="String" direction="fixed">
            <value>SFipa.QUERY_REF</value>
        </parameter>
        <parameter name="content" class="String" direction="fixed">
            <value>"ping"</value>
        </parameter>
    </messageevent>

    <!-- An inform message where content contains the word "hello" -->
    <messageevent name="inform_hello" type="fipa" direction="receive">
        <parameter name="performative" class="String" direction="fixed">
            <value>SFipa.INFORM</value>
        </parameter>
        <match>((String)$content).indexOf("hello")!=-1</match>
    </messageevent>
</events>
```

Figure 9.3. Examples for receiving messages

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 9.4, “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 and the presence of a match expression are 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 (e.g., 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 9.4. Possible problems when matching messages

Level	Output
INFO	<agentname> multiple events matching message, using message event with highest specialization degree
WARNING	<agentname> cannot process message, no message event matches
SEVERE	<agentname> 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 valid FIPA SL. The parameter is matched against the action expression contained in the message. As the effect of these special parameters can also be achieved using a match expression, support for these parameters might be dropped in future releases.

9.2.2. Sending Messages

Messages to be sent also have to be declared in the ADF. The actual sending is usually done inside a plan, which instantiates the declared message event, fills in desired parameter values, and dispatches the message using one of the sendMessage...() methods. 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 as described in Section 9.2.3, “Using Ontologies and Content Languages”.

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. This is described in Section 9.2.4, “Using Conversations for Managing Sequences of Messages”

```
<imports>
    <import>jadex.adapter.fipa.SFipa</import>
</imports>
...
<events>
    <!-- A query-ref message with content "ping" -->
    <messageevent name="query_ping" type="fipa" direction="send">
        <parameter name="performative" class="String">
            <value>SFipa.QUERY_REF</value>
        </parameter>
        <parameter name="content" class="String">
            <value>"ping"</value>
        </parameter>
    </messageevent>
</events>
```

```
// Plan snippet showing the creation and sending of the message.
public void body() {
    IMessageEvent me = createMessageEvent("query_ref");
    me.getParameterSet(SFipa.RECEIVERS).addValue(new AgentIdentifier("Ping", true));
```

```
    // me.setContent("ping 2"); // Set/change content if necessary
    sendMessage(me);
}
```

Figure 9.4. Example of sending a message

9.2.3. Using Ontologies and Content Languages

Message based communication allows that agents can communicate even when they are distributed across the network. One important property in the context of message based communication is the separaration of address spaces, i.e., that agents do not have direct access to the data inside other agents. Therefore data needs to be encoded into a message before sending and decoded from a message after receival. In the context of multi-agent systems, so called content languages and ontologies are responsible for describing how data should be encoded into messages. A content language defines the syntactical mechanism used to represent data and an ontology specifies the meaning of the concepts used in the message. Together, content language and ontology assure a shared common understanding among agents.

The data inside a Jadex agent is usually represented as a collection of Java objects referencing each other. The Jadex framework provides some useful features that allow to encode/decode object structures, such that they can be used directly for the communication between agents. For this purpose, the agent knows about so called *content codecs*, some of which are available by default, but can also be extended with custom codecs by the agent programmer. These codecs are selected automatically, when sending and receiving messages and are used to encode or decode the content of a message. From the viewpoint of an agent programmer, the agent is just sending or receiving messages containing Java objects. All the encoding and decoding works behind the scenes.

Two simple examples for sending and receiving a Java object inside a message are shown below (taken from the marsworld example). These examples use a `Target` object from package `jadex.examples.marsworld`. On the sender side, the message defines to use the language `SFipa.NUGGETS_XML`, which is per default available in each agent (see Figure 9.5, “Example of sending an object inside a message”). The corresponding nuggets codec can handle arbitrary JavaBeans (i.e. Java objects, which provide public getter and setter methods for their properties). For detailed information about JavaBean you should have a look at the JavaBeans Specification[3].

```
<!-- Message declaration in the ADF -->
<messageevent name="inform_target" type="fipa" direction="send">
    <parameter name="performative" class="String" direction="fixed">
        <value>SFipa.INFORM</value>
    </parameter>
    <parameter name="language" class="String" direction="fixed">
        <value>SFipa.NUGGETS_XML</value>
    </parameter>
    <parameter name="ontology" class="String" direction="fixed">
        <value>MarsOntology.ONTOLOGY_NAME</value>
    </parameter>
</messageevent>
```

```
public void body() {
    // Message sending in the plan.
    AgentIdentifier receiver = ...
    Target target = ...
    IMessageEvent me = createMessageEvent("inform_target");
    me.getParameterSet(SFipa.RECEIVERS).addValue(receiver);
    me.setContent(target); // The Java object is directly used as content.
    sendMessage(me);
}
```

[3] http://java.sun.com/products/javabeans/docs/spec.html

Figure 9.5. Example of sending an object inside a message

As the decoded object is already availble for matching an incoming message, on the receiver side, the `content-class` parameter can be used to only match messages containing a `Target` object (see Figure 9.6, “Example of receiving an object inside a message”).

```
<!-- Message declaration in the ADF -->
<messageevent name="target_inform" type="fipa" direction="receive">
    <parameter name="performative" class="String" direction="fixed">
        <value>SFipa.INFORM</value>
    </parameter>
    <parameter name="content-class" class="Class" direction="fixed">
        <value>Target.class</value>
    </parameter>
    <parameter name="ontology" class="String" direction="fixed">
        <value>MarsOntology.ONTOLOGY_NAME</value>
    </parameter>
</messageevent>
```

```
public void body() {
    // Message receiving in the plan.
    IMessageEvent msg = (IMessageEvent)getInitialEvent();
    Target target = (Target)msg.getContent();
    ...
}
```

Figure 9.6. Example of receiving an object inside a message

Two content languages are predefined in Jadex itself and therefore are available on all platforms. These languages are defined in the constants `SFipa.JAVA_XML` and `SFipa.NUGGETS_XML`. The Java XML language uses the bean encoder available in the JDK, to convert Java objects adhering to the JavaBeans specification to standardized XML files. The nuggets XML language is a proprietary language in Jadex, that works similar to the Java XML language but the encoding and decoding is much faster. Both languages allow marshalling content objects independently from the underlying ontology as they rely completely on the Java Bean specification. Using these languages 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[4]) if you are interested in converting an ontology to Java beans including the necessary bean infos. Other content languages are available depending on the underlying platform (e.g. the JADE platform supports the FIPA SL language). The usage of these platform-specific languages is described in Appendix B, *Platform Adapters*.

If you want to use your own mechanism for encoding and decoding of message contents, you can implement the interface `IContentCodec` from package `jadex.runtime`. The interface requires you to implement three methods. The `match()` method is used by Jadex, to determine if your codec applies to a given message. For this decision, the important message properties (e.g. langauge and ontology) are supplied. The other two methods are called to `encode()` and object to a string for sending and to `decode()` a string back to an object, when receiving a message. To register a custom content codec in an agent, it is sufficient to add a property starting with `contentcodec.` in the properties section of an agent:

```
<properties>
    <property name="contentcodec.my_codec">new MyContentCodec()</property>
</properties>
```

[4] http://vsis-www.informatik.uni-hamburg.de/projects/jadex/

9.2.4. Using Conversations for Managing Sequences of Messages

Normally messages are not sent in isolation, but occur inside a conversation of many messages that are sent and received. Because of this, you often want to identify a certain message as belonging to a specific conversation or being a direct reply to some other message sent before. In the FIPA message structure, three parameters are responsible for this kind of conversation management. A unique `conversation-id` can be used to group together several messages belonging to a single conversation. In addition the `in-repy-to` parameter allows to identify a message as being an answer to a previous message with a corresponding `reply-with` parameter value.

In Jadex, the relation between messages is used to achieve two things: First, it allows to wait for a specific message while ignoring other messages that do not belong to an ongoing conversation or are a reply to another message. Thanks to this, e.g., when two plans simultaneously wait for the same type of message, a received message will automatically be posted to the correct plan, from which the previous message of the conversation was sent. Second, it allows to restrict message receival to a certain capability, namely the capability from which an earlier message was sent. This means, e.g., that if an agent defines two similar message events in two different capabilities (as is commonly the case, when the same capability is included twice in an agent), the message will automatically be routed to the correct capability where the corresponding conversation originated.

In both cases, the mechanism is based on the usage of the `conversation-id` and/or `in-repy-to` and `reply-with` parameters. The developer has to make sure that, when sending an initial message a useful value has been set to one or more of these parameters. When replying to an initial message (by using `msg.createReply(...)`), the parameter values are set automatically, based on the values of the initial message (i.e. the conversation-id is retained while the reply-with is copied to the in-reply-to parameter). The setting of initial parameter values can directly be done in the message declaration as shown in Figure 9.7, “Example of an Initial Conversation Message”. In the example, the method `createUniqueId()` is used to generate a unique id for the conversation, whenever an instance of the message is created. The plan can send the message using `dispatchMessageAndWait()`. and directly receive the correponding reply message. When using a timout in `dispatchMessageAndWait()` and the message is not received before the timeout has elapsed, a `jadex.runtime.TimeoutException` is thrown (see also Section 8.2.1, “Plan Success or Failure and BDI Exceptions”). For a reply message (e.g. the inform below) no special settings have to defined in the ADF.

```
<events>
    <messageevent name="request" type="fipa" direction="send">
        <parameter name="performative" class="String">
            <value>SFipa.REQUEST</value>
        </parameter>
        <parameter name="conversation-id" class="String">
            <value>SFipa.createUniqueId($scope.getAgentName())</value>
        </parameter>
    </messageevent>
    <messageevent name="inform" type="fipa" direction="receive">
        <parameter name="performative" class="String" direction="fixed">
            <value>SFipa.INFORM</value>
        </parameter>
    </messageevent>
<events>
```

```
public void body() {
    IMessageEvent me = createMessageEvent("request");
    ... // Set other parameters as desired
    IMessageEvent reply = sendMessageAndWait(me);
    ... // Handle reply message
}
```

Figure 9.7. Example of an Initial Conversation Message

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 or in-reply-to also appears in the answer. Moreover, message properties, which should not change during a conversation (e.g. protocol, language and ontology) are also automatically copied into the reply. 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 also has to be present in your ADF as shown in Figure 9.8, “Example for Replying to a Message”.

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

```
public void body() {
    // Message receiving in the plan.
    IMessageEvent msg = (IMessageEvent)getInitialEvent();
    Object content = ... // Prepare content for reply
    IMessageEvent reply = msg.createReply("inform", content);
    sendMessage(reply); // Take care to send 'reply' and not 'msg'!
}
```

Figure 9.8. Example for Replying to a Message

The way of handling conversations described in this section is pretty different to programming agents based on abstract goals, as the programmer has to directly deal with all alternatives of the interaction flow. This process can be tedious and error-prone. Therefore, in Jadex a predefined capability is available, that already implements common use cases of interactions as specified in standardized FIPA interaction protocols (e.g. request, contract-net, auctions). The protocols capability allows to focus on the goals of the agents participating in a conversation. The protocols capability is described in detail in Chapter 16, *Using Predefined Capabilities*. Even if you want to implement your own custom interaction protocol, you should have a look at the protocols capability, because it introduces helpful patterns that can be applied to other interactions as well.

9.3. Goal Events

Besides internal and message events, there is a third kind of event in Jadex, that is sometimes of interest to an agent developer. So called goal events (`IGoalEvent`) are used to manage the processing of goals. For standard plans, goal events can usually be ignored, but when developing mobile plans, the results of goal processing are passed as goal events to the `action()` method of the plan. Therefore, understanding goal events is quite important, if you want to develop mobile plans.

Goal events are not declared in the ADF, as they are used only internally for the processing of goals. This means that the agent will automatically 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). To explicitly decide which kind of event has happened (as commonly necessary inside mobile plans) the `IGoalEvent.isInfo()` method can be used. Below, an excerpt of the `PickUpWastePlan` from the cleanerworld mobile example is shown to illustrate the usage

of goal events. In contrast to standard plans a mobile plan will be always be invoked through calling its `action()` method regardless of the state of that plan. Hence the programmer has to supply case distinction for the different plan steps and can use the event provided parameter for this decision. In the example in the first step a moveto subgoal is dispatched and in the second step (when the position has been reached) the desired clean-up operation can be performed.

```
public void action(IEvent event)
{
    Waste waste = (Waste)getParameter("waste").getValue();

    // First event is a process event (=!isInfo)
    if(event instanceof IGoalEvent && !((IGoalEvent)event).isInfo())
    {
        IGoal moveto = createGoal("achievemoveto");
        moveto.getParameter("location").setValue(waste.getLocation());
        dispatchSubgoalAndWait(moveto);
    }

    // Second event is info event from achievemoveto goal dispatched above.
    else if(event instanceof IGoalEvent
        && ((IGoalEvent)event).getGoal().getType().equals("achievemoveto"))
    {
       ...
    }
}
```

Figure 9.9. Usage of goal events in the cleanermobile example

Chapter 10. 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).

10.1. 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 4, *Imports*) that allow to use unqualified class names. The imports are defined once, and can be used for all expressions throughout the ADF.

10.2. 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 10.1, “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.

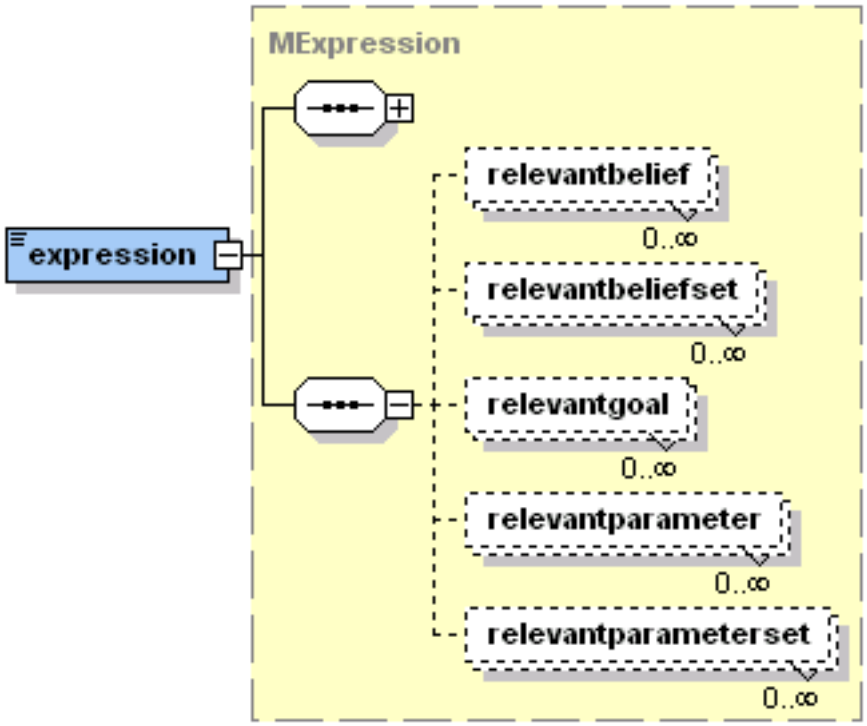

Figure 10.1. The Jadex expressions relevant settings XML schema part

10.3. 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 10.1, “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. (In future releases this will change so that only the interfaces are accessible in expressions, too).

Table 10.1. Reserved expression variables

Name	Class	Accessibiliy
`$agent`	`RBDIAgent`	In any agent expression
`$scope`	`RCapability`	In any expression
`$beliefbase`	`RBeliefbase`	In any expression
`$planbase`	`RPlanbase`	In any expression
`$goalbase`	`RGoalbase`	In any expression
`$eventbase`	`REventbase`	In any expression
`$expressionbase`	`RExpressionbase`	In any expression
`$propertybase`	`RPropertybase`	In any expression
`$goal`	`RGoal`	In any goal expression (except cre-

Name	Class	Accessibiliy
		ation condition and binding options)
`$plan`	`RPlan`	In any plan expression (except trigger and pre condition and binding options)
`$event`	`REvent`	In any event expression (except binding otpions)
`$ref`	`RGoal`	In any inhibition arc expression.
`$messagemap`	`Map`	In match expressions of message events.

10.4. Expressions Examples

In Figure 10.2, “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 10.2. Example expressions

10.5. 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 10.3, “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.

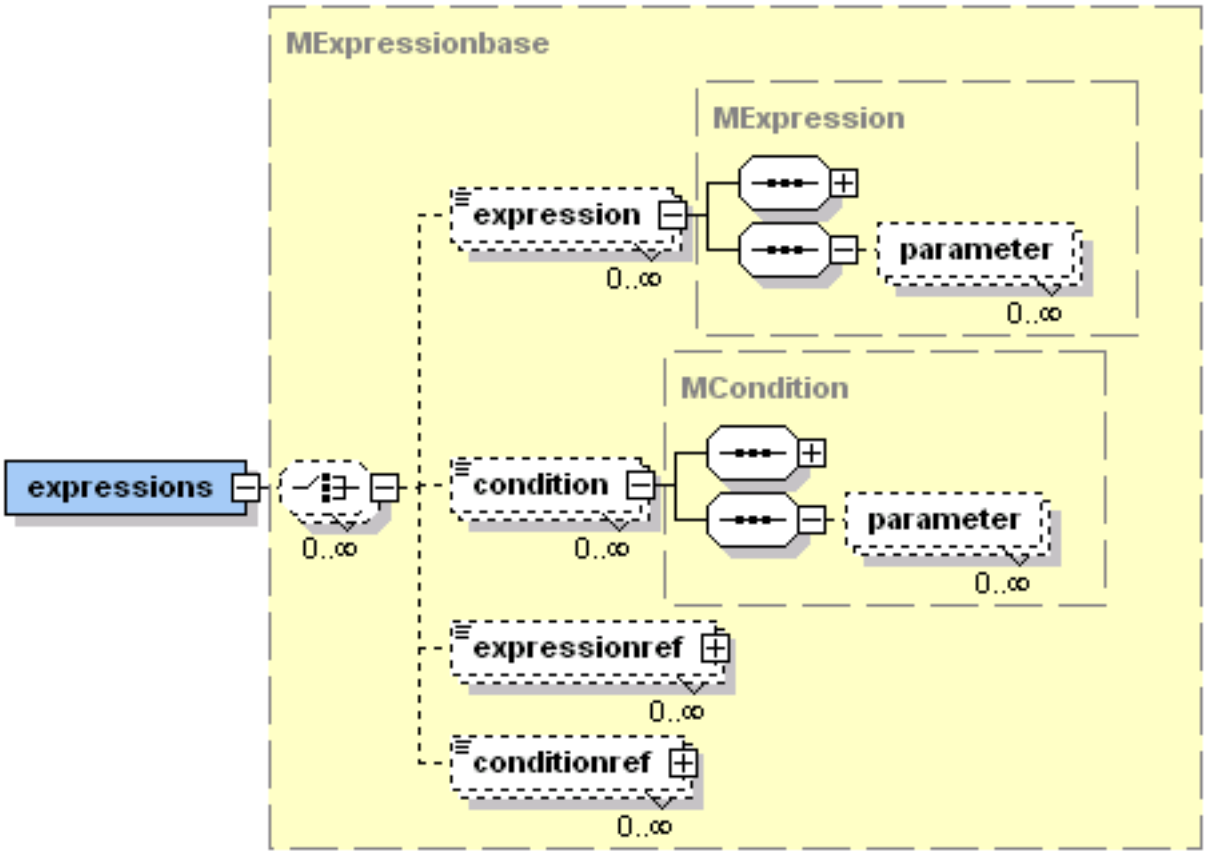

Figure 10.3. The Jadex expressions XML schema part

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 that are defined as a `String` and an `Object`array containing parameter names and values separately. 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 10.4. Defining an expression in the ADF

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

Figure 10.5. Evaluating an expression from a plan

10.6. 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 of 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 10.6, “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 10.6. 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. 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 10.7, “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 10.7. 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 11. 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 behavior 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 11.1, “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 11.1. 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)

11.1. ADF Conditions

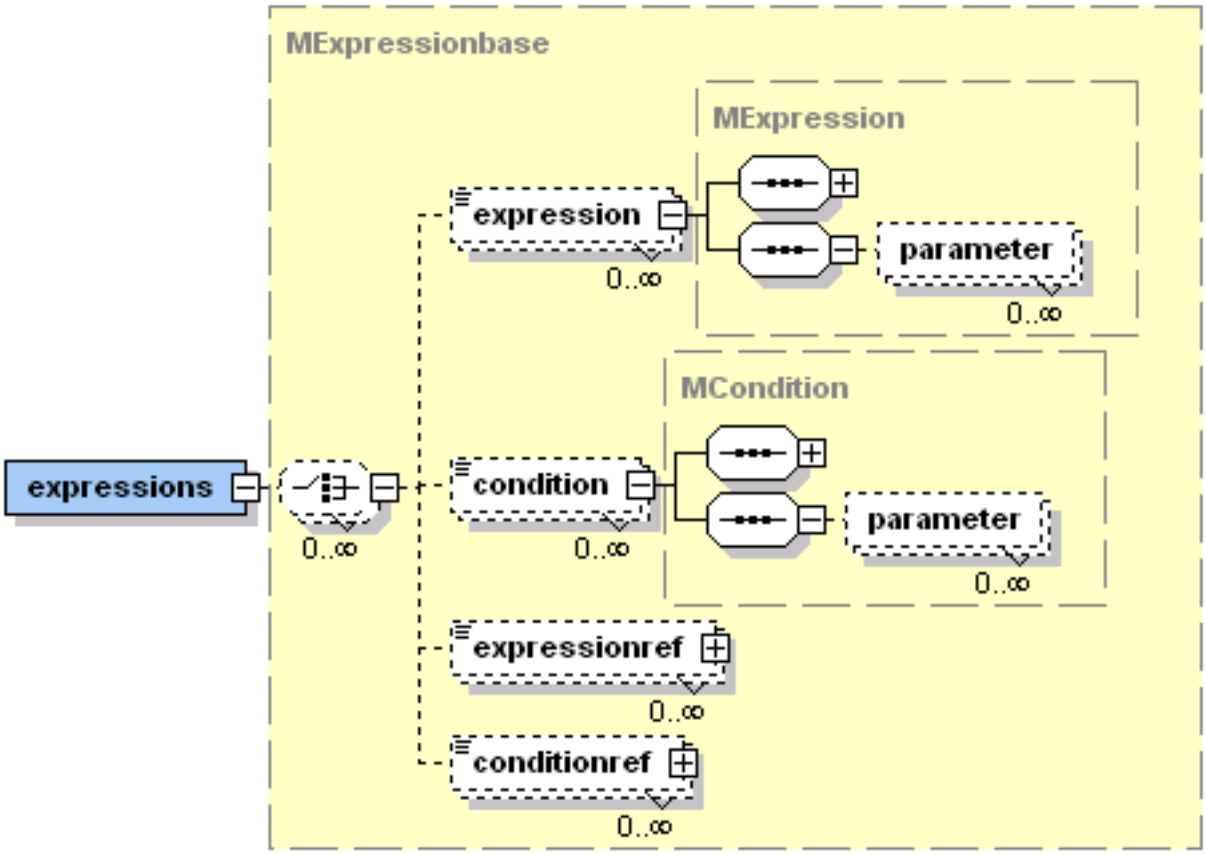

Figure 11.1. The Jadex conditions XML schema part

When programming plans, it is also possible to explicitly wait for certain conditions using the `waitForCondition(ICondition cond)` method. Conditions are obtained in a similar fashion to expressions, either by instantiating a predefined condition from the ADF (see Figure 11.1, “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 6.3, “Dynamically Evaluated Beliefs” to execute some action when the alarmtime has reached (belief not shown here).

```
<agent ...>
    ...
    <expressions>
        <condition name="alarmtime_reached">
            $beliefbase.timer >= $beliefbase.alarmtime
        </condition>
        ...
    </expressions>
    ...
</agent>
```

Figure 11.2. Defining a condition in the ADF

```
public void body {
    ICondition condition = getCondition("alarmtime_reached");
    ...
```

```
    // Wakeup every full hour.
    IEvent event = waitForCondition(condition);
    ...
}
```

Figure 11.3. Using a condition inside a plan

Chapter 12. Properties

This chapter contains an overview about the usage of agent and capability properties, that allow to change the behavior of the agent. In general, properties represent static expressions, i.e. they are interpreted but only once when an agent instance is loaded. 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.

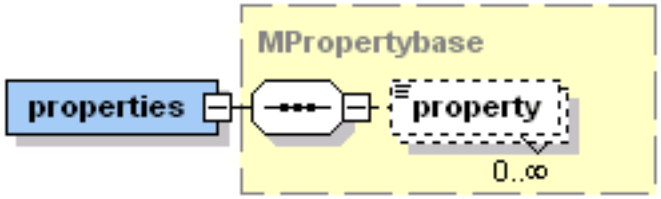

Figure 12.1. The Jadex properties XML schema part

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 12.1, “Available properties” gives an overview of all 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 12.1. 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 an ongoing 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 behavior.

The logging properties can be used to adjust the logging behavior according to the Java Logging API[1]. 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 expect to throw exceptions. The following concrete subclasses of the abstract `jadex.runtime.BDIFailureException` may occur:

- `jadex.runtime.AgentDeathException`
  The agent death exception denotes that an agent has dies. It is e.g. used by the reasoning engine to wake up waiting agent listeners and the agent is killed in the meantime.

- `jadex.runtime.GoalFailureException`
  A goal failure exception indicates that a goal could not successfully pursued. It is thrown by the reasoning engine when e.g. dispatchSubgoalAndWait() is called and the goal does not succeed.

- `jadex.runtime.MessageFailureException`
  Indicates that a message could not be delivered. Is e.g. thrown by the message transport system of the Standalone platform when a message should be sent that has no receivers.

- `jadex.runtime.PlanFailureException`
  Can be thrown from user code in plans for indicating that a normal plan failure has occurred. Also calling the fail() method will lead to throwing a plan failure exception.

- `jadex.runtime.TimeoutException`
  Occurs, when any waitFor...() method is called with eexception of the basic waitFor(time) method, which will only block until the given time interval elpased.

Figure 12.2, “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>
```

[1] http://java.sun.com/j2se/1.4/docs/guide/util/logging/overview.html

```
</properties>
```

Figure 12.2. Example properties section

Chapter 13. Configurations

Configurations represent both the initial and/or end states of an agent type. Initial instance elements can be declared that are created when the agent (resp. the capability) is started. This means that initial elements such as goals or plans are created immediately when an agent is born. On the conatrary, end elements can be used to declare instance elements such as goals or plans that will be created when an agent is going to be terminated. After an agent has been urged to terminate (e.g. by calling `killAgent()` from within a plan or by an AMS `ams_destroy_agent` goal), all normal goals and plans will be aborted (except plans that perform their cleanup code, i.e. execute one of the `passed()`, `failed()` or `aborted()` methods) and the declared end elements will be created and executed.

Instance and end elements always have to refer to some original element via the "ref" attribute. Additionally, an optional instance name can be provided via the "name" attribute. This can be useful if the element should be accessible later on via this name. Besides the name also bindings can be used in combination with initial/end elements. If (at least one) binding parameter is declared instance elements will be created for all possible bindings.

It is possible to declare any number of configurations for a single agent or capability. When starting an agent or including a capability you can choose among the available configurations In the XML portion for specifying configurations is depicted. Each configuration must have a name for identification purposes. The default configuration can be set up by using the `default` attribute of the `<configurations>` base tag. If no explicit default configuration is specified, the first one declared in the ADF is used.

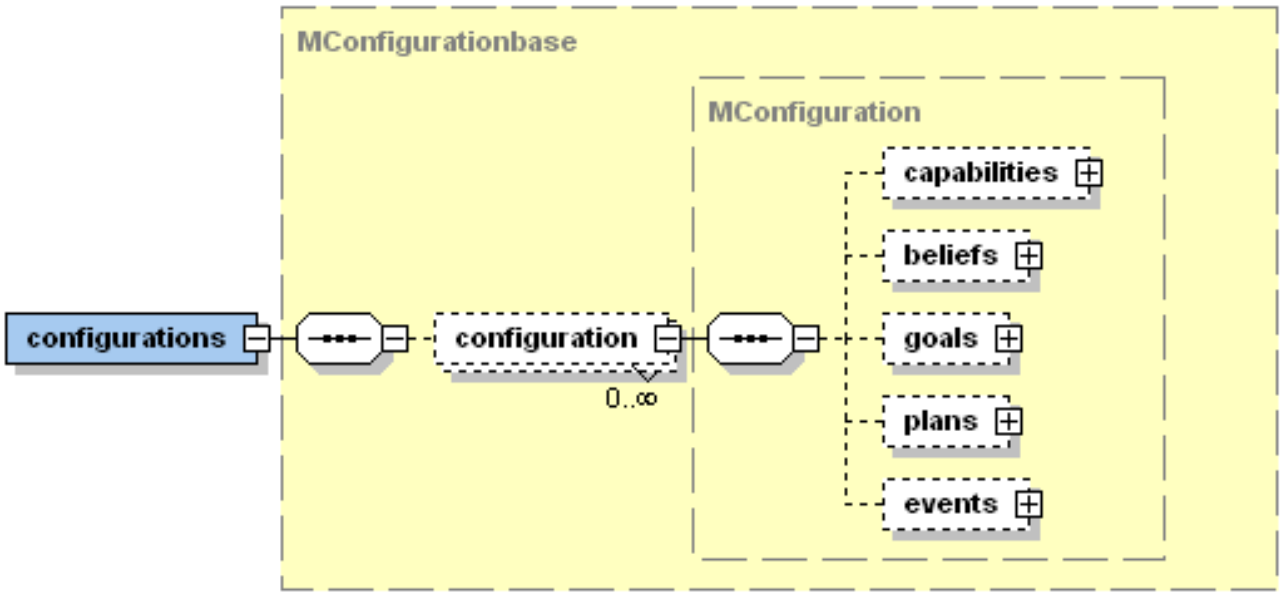

Figure 13.1. The Jadex configurations XML schema part

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

13.1. 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 configuration to be used by the included capability can be set by using the configuration attribute of the initial capability tag.

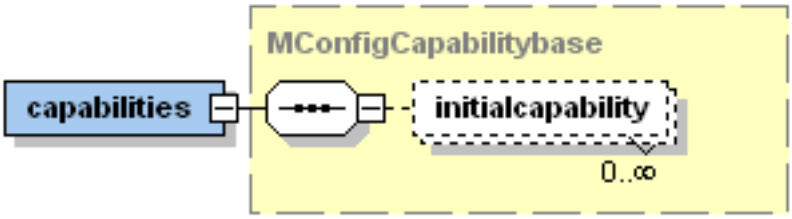

Figure 13.2. The Jadex initial capabilities XML schema part

In Figure 13.3, “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>
    ...
    <configurations default="two">
        <configuration name="one">
            <capabilities>
                <initialcapability ref="mycap" configuration="a"/>
            </capabilities>
        </configuration>
        <configuration name="two">
            <capabilities>
                <initialcapability ref="mycap" configuration="b"/>
            </capabilities>
        </configuration>
    </configurations>
</agent>
```

Figure 13.3. Initial capability configuration

13.2. 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.

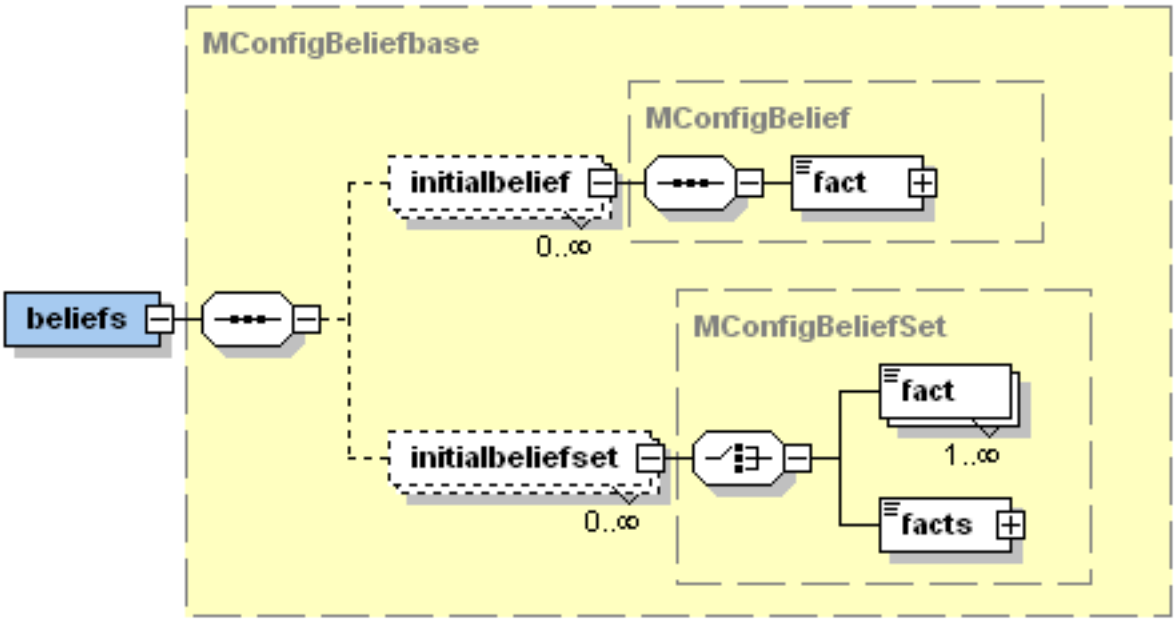

Figure 13.4. The Jadex initial beliefs XML schema part

The example in Figure 13.5, “Initial belief configuration” shows how a configuration 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>
    ...
    <configurations>
        <configuration name="one">
            <beliefs>
                <initialbelief ref="name">
                    <fact>"John"</fact>
                </initialbelief>
                <initialbelief set ref="names">
                    <facts>DB.queryNames()</facts>
                </initialbelief set>
            </beliefs>
        </configuration>
    </configurations>
</agent>
```

Figure 13.5. Initial belief configuration

13.3. Goals

In the <goals> section initial and end goals can be specified. Initial goals will be instantiated when an agent is born whereas end goals are created when an agent is beginning the termination phase. This means that a new goal instance is created for each declared initial resp. end goal at the mentioned points in time. The specification of an <initialgoal> and an <endgoal> 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.

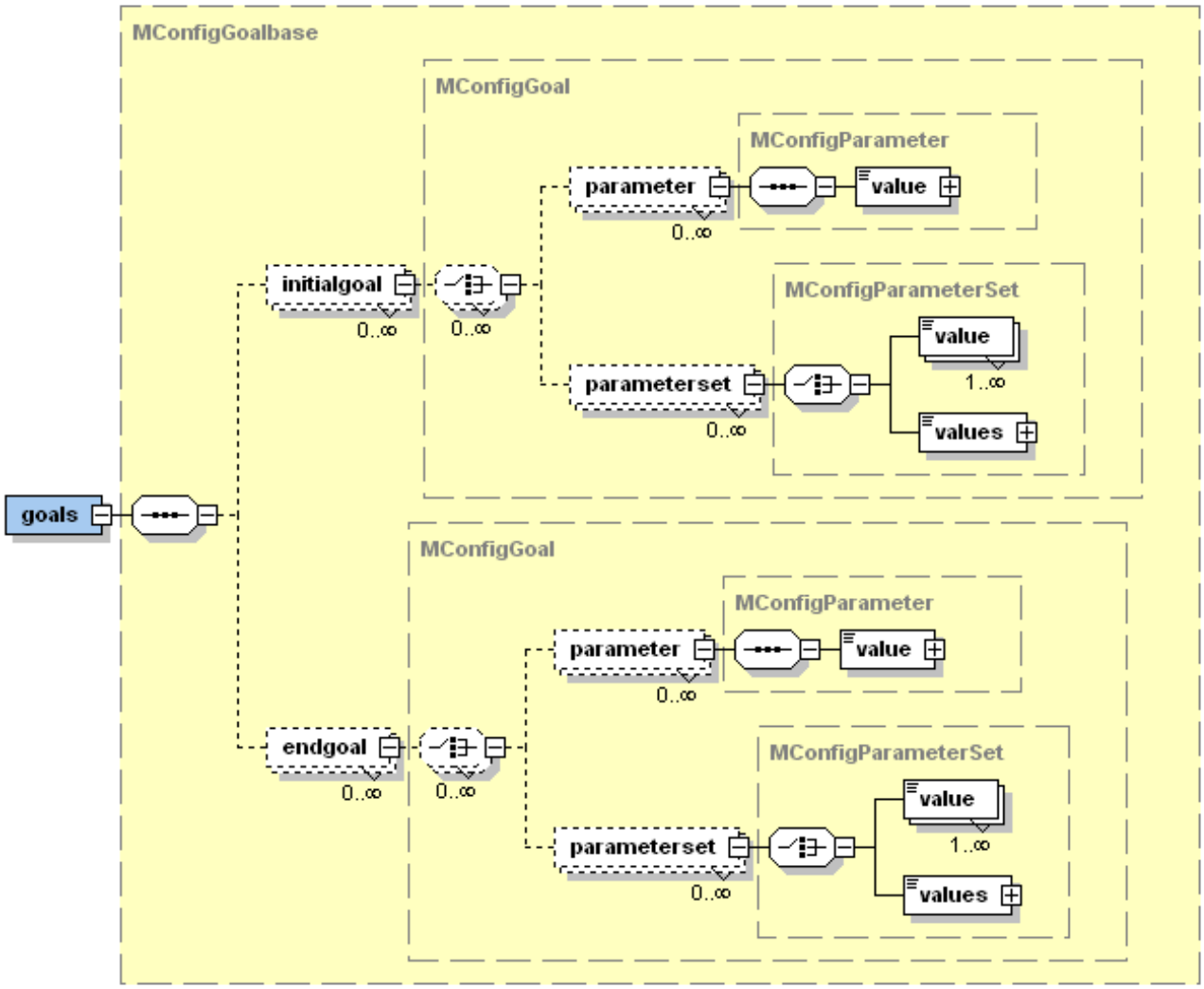

Figure 13.6. The Jadex initial and end goals XML schema part

In Figure 13.7, “Initial and end goals” an example is depicted how an initial and end goal can be created. Both, the initial and end goal refer 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. On the other hand the agent will also play a good bye jingle when it is terminated by creating the corresponding end goal.

```
<agent ...>
    ...
    <goals>
        <performgoal name="play_song">
            <parameter name="song" class="URL"/>
        </performgoal>
```

```
    </goals>
    ...
    <configurations>
        <configuration name="one">
            <goals>
                <initialgoal name="welcome" ref="play_song">
                    <parameter ref="song">
                        <value>new URL("http://someserver/welcome.mp3")</value>
                    </parameter>
                </initialgoal>
                <endgoal name="goodbye" ref="play_song">
                    <parameter ref="song">
                        <value>new URL("http://someserver/goodbye.mp3")</value>
                    </parameter>
                </endgoal>
            </goals>
        </configuration>
    </configurations>
</agent>
```

Figure 13.7. Initial and end goals

13.4. Plans

In the `<plans>` section initial and end plans can be specified. This means that a new plan instance is created for each declared initial and end plan. The specification of an `<initialplan>` and `<endplan>` 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.

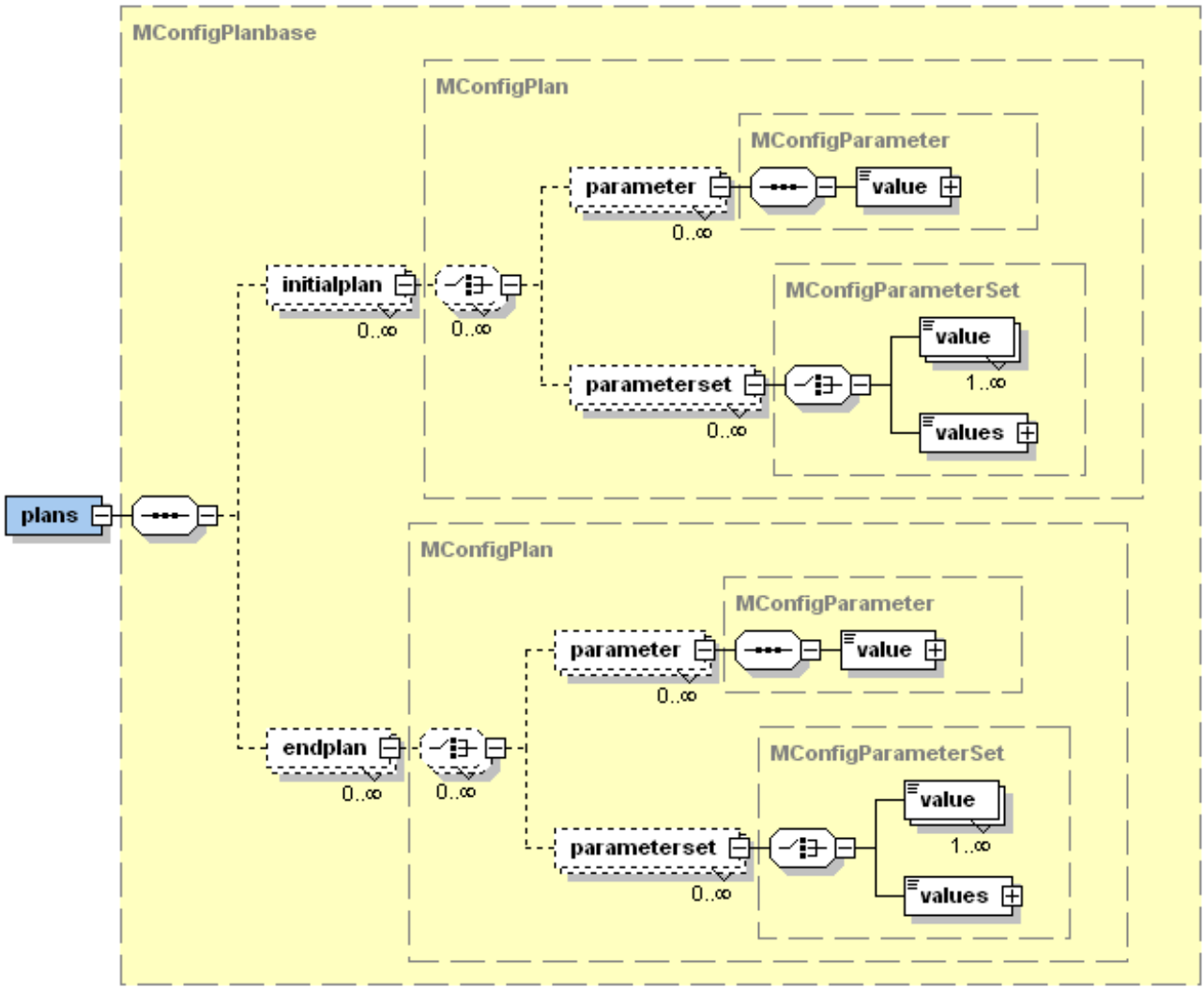

Figure 13.8. The Jadex initial and end plans XML schema part

In Figure 13.9, “Initial and end plans” an example is depicted how an initial and end 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. On the contrary it will also print "Goodbye World" when the agent gets terminated by creating the corresponding end plan.

```
<agent ...>
    ...
    <plans>
        <plan name="print_plan">
            <parameter name="text" class="String"/>
            <body class="PrintOnConsolePlan" />
        </plan>
    </plans>
    ...
    <configurations>
        <configuration name="one">
            <plans>
                <initialplan ref="print_hello">
                        <parameter name="text">"Hello World!"</parameter>
                </initialplan>
                <endplan ref="print_goodbye">
                        <paramter name="text">"Goodbye World!"</parameter>
                </endplan>
```

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

Figure 13.9. Initial and end plans

13.5. Events

Finally, in the `<events>` section initial and end events can be specified. This means that a new event instance is created for each declared initial event after startup of the agent. Additionally, new event instances are created for all declared end events whenever the agent is shutdowned. It is possible to define initial/end internal and initial/end message events (goal events are not necessary as initial goals can be declared). The specification of an `<initialinternalevent>` resp. an `<endinternalevent>` or an `<initialmessageevent>` resp. an `<endmessageevent>` 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.

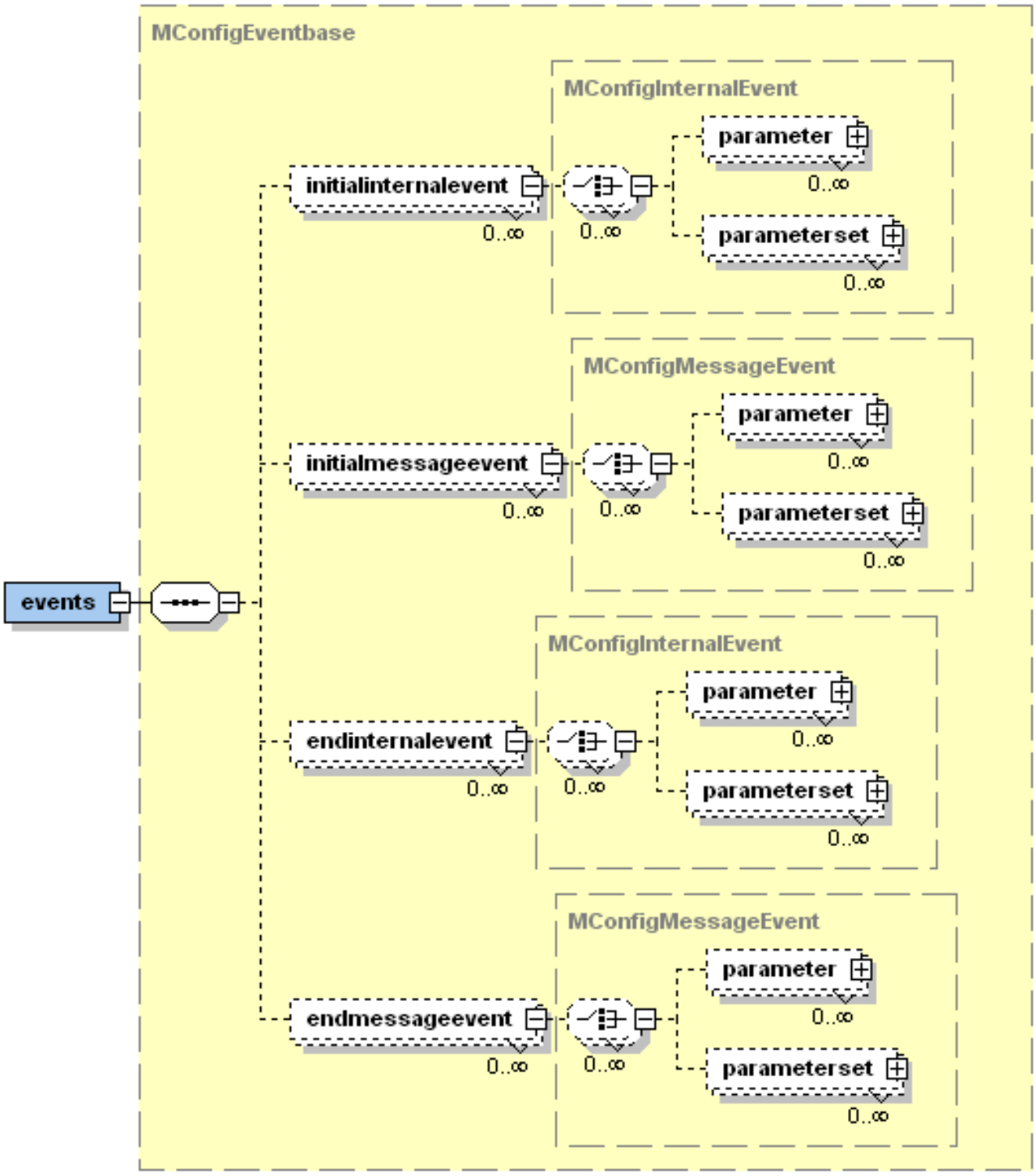

Figure 13.10. The Jadex initial and end events XML schema part

In Figure 13.11, “Initial events” an example is shown how an initial and end message event can be created. The intial/end message events refer to the underlying message event template "inform_state" and set the parameter values for the content as well as for the receiver accordingly. When an agent named "Harry" is started, it sends an initial message event with the content "Harry is born" to an agent named "Uncle" on the same platform. Likewise it sends the message "Harry is terminating" to "Uncle" when the agent shuts down.

```
<events>
    <messageevent name="inform_state" type="fipa" direction="send">
```

```
        <parameter name="performative" class="String" direction="fixed">
            <value>SFipa.INFORM</value>
        </parameter>
    </messageevent>
</events>
...
<configurations>
    <configuration name="one">
        <events>
            <initialmessageevent ref="inform_state">
                <parameter ref="content">
                    <value>$agent.getAgentName()+" is born."</value>
                </parameter>
                <parameterset ref="receivers">
                    <value>new AgentIdentifier("Uncle", true)</value>
                </parameterset>
            </initialmessageevent>
            <endmessageevent ref="inform_state">
                <parameter ref="content">
                    <value>$agent.getAgentName()+" is terminating."</value>
                </parameter>
                <parameterset ref="receivers">
                    <value>new AgentIdentifier("Uncle", true)</value>
                </parameterset>
            </endmessageevent>
        </events>
    </configuration>
</configurations>
```

Figure 13.11. Initial events

Chapter 14. 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 layer 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.

14.1. 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 5.3, “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 14.1, “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 14.1. Adding a capability at runtime

The removal of a capability can be done likewise. In Figure 14.2, “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 14.2. Removing a capability at runtime

14.2. 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 6.1, “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 14.3, “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 14.3. Creating a belief at runtime

In Figure 14.4, “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 14.4. Deleting a belief at runtime

14.3. 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 7.1, “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 14.5, “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 14.5. Creating a new goal type at runtime

An example for the deletion of a goal type at runtime is shown in Figure 14.6, “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 14.6. Deleting a goal type at runtime

14.4. 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 14.7. Creating a new plan type at runtime

In the following the deletion of a plan type is sketched (see Figure 14.8, “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 14.8. Deleting a plan type at runtime

14.5. 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.

In Figure 14.9, “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 14.9. Creating a new internal event type at runtime

In Figure 14.10, “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 14.10. Deleting an internal event type at runtime

Chapter 15. External Interactions

In this chapter it is explained how the interaction of Jadex agents with other system components that are not necessarily agents can be done. For this purpose it is shown how agent internals can be accessed from other (non-agent) threads (cf. Section 15.1, “External Processes”) and additionally how agent listeners can be employed to get notified whenever changes within the agent happen (cf. Section 15.2, “Agent Listeners”).

15.1. 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 one may 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. On the contrary, it is allowed to call the external access from any thread (including plan resp. agent threads).

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 blocking 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 15.1. External process example

15.2. Agent Listeners

Agent listeners can be used to get informed whenever agent state changes happen. Normally, listeners will be employed in agent external components such as a GUI for getting information about declared elements. A GUI e.g. could use a listener to update its view with respect to belief changes in the agent. Generally, for all important agent attitudes such as belief, plans and goals as well as the agent itself different listener types exist (see Table 15.1, “Available listeners”).

Depending on the listener type different callback methods are provided that are automatically invoked when relevant changes happen. Whenever a callback method is invoked a so called `AgentEvent` is passed and contains relevant information about the change that happened. It basically offers the two methods `getSource()` and `getValue()`. The source here is the originating element of the change event. For belief and beliefset changes, the agent event additionally contains the changed fact object, accessible by `getValue()`.

The invocation of listener methods can happen either on the agent thread or on a separate thread. If the notification is performed on the agent thread it is not possible to use blocking calls such as `dispatchTopLevelGoalAndWait()`. This is only allowed if asynchronous listeners are used. The decision to use a synchronous or an asynchronous listener is made when the listener is added.

The addition and removal of listeners can be done either on the instance elements themselves (e.g. a goal) or on the bases (e.g. the goalbase). In case the listener shall be added on an instance element it is only necessary to pass the listener object itself and the asynchronous flag as parameters of the call (e.g. `addBeliefListener( IBeliefListener listener, boolean async)`). In case a type-based listener shall be used e.g. for getting informed about new goal instances in addition to the parameters aforementioned also the type needs to be declared (e.g. `addGoalListener( String type, IGoalListener listener, boolean async)`).

In the listener example below (see Figure 15.2, “Agent listener example”) it is shown how a belief listener can be directly added to a "name" belief via the external access interface. It is used to update the value of a textfeld whenever the belief value changes.

```
IExternalAccess agent = ...
agent.getBeliefbase().getBelief("name").addBeliefListener(new IBeliefListener() {
    public void beliefChanged(AgentEvent ae) {
        textfield.setText("Name: ["+ae.getValue()+"]");
    }
}, false);
```

Figure 15.2. Agent listener example

Table 15.1. Available listeners

Listener	Element	Listener Methods
`IAgentListener`	`ICapability`	`agentTerminating()`[a]
`IBeliefListener`	`IBelief`	`beliefChanged()`
`IBeliefSetListener`	`IBeliefSet`	`factAdded()`, `factRemoved()`, `beliefSetChanged()`[b]
`IConditionListener`	`ICondition`, `IExpressionbase`	`conditionTriggered()`
`IGoalListener`	`IGoal`, `IGoalbase`	`goalAdded()`, `goalFinished()`
`IInternalEventListener`	`IEventbase`	`internalEventOccurred()`
`IMessageEventListener`	`IMessageEvent`, `IEventbase`	`messageEventReceived()`, `messageEventSent()`
`IPlanListener`	`IPlan`, `IPlanbase`	`planAdded()`, `planFinished()`

[a] The event source will always be the terminating agent itself (i.e., the root capability) even when registering the listener on a subcapability.
[b] The value of the agent event will be null when all facts have changed, e.g., due to a dynamic facts expression.

Chapter 16. Using Predefined Capabilities

Jadex uses capabilities for the modularization of agents (see Chapter 5, *Capabilities*), whereby capabilities contain ready to use functionalities. The Jadex distribution contains several ready-to-use predefined capabilities for different purposes. Besides the basic management capabilties for using the AMS (agent management system, see Section 16.1, “The Agent Management System (AMS) Capability”) and the DF (directory facilitator, see Section 16.2, “The Directory Facilitator (DF) Capability”) also a Protocols capability (Section 16.3, “The Interaction Protocols Capability ”) is available for the efficient usage of some predefined FIPA interaction protocols. The interface of a capability mainly consists of a set of exported goals which is similar to an object-oriented method-based interface description. This chapter aims at depicting their usage by offering the application programmer an overview and explanation of their functionalities and additionally a selection of short code snippets that can directly be used in your applications.

The test capability for writing agent-based unit test is explained in the “Jadex Tool Guide”, which also illustrates the usage of the corresponding Test Center user interface.

16.1. The Agent Management System (AMS) Capability

The Agent Management System (AMS) capability provides goals, that allow the application programmer to use functionalties of the local or some remote AMS. Basically the AMS is responsible for managing the agent lifecycle and for interacting with the platform. Concretely this means the AMS capability can be used for

- Section 16.1.1, “Creating an Agent”
- Section 16.1.2, “Starting an Agent”
- Section 16.1.3, “Destroying an Agent”
- Section 16.1.4, “Suspending an Agent”
- Section 16.1.5, “Resuming an Agent”
- Section 16.1.6, “Searching for Agents”
- Section 16.1.7, “Shutting Down a Platform”

16.1.1. Creating an Agent

The goal “ams_create_agent” creates a new agent via the AMS on the platform.

This goal has the following parameters:

Table 16.1. Parameters for ams_create_agent

Name	Type	Description
`ams` *	`AgentIdentifier`	The AMS agent identifier (only needed for remote requests).
`arguments` *	`Map`	The arguments as name-value pairs for the new agent. Depending on the platform, Java objects (for Jadex Standalone

Name	Type	Description
		or local JADE requests) or string expressions (for remote JADE requests) have to be supplied for the argument values.
configuration *	String	The initial agent configuration to use.
name *	String	The name of the instance to create. If no name is specified, a name will be generated automatically.
type	String	The type (e.g. XML file name) of the agent you want to create.
start	Boolean	True, when the agent shoudl be directly started after creation (default). Note that some platforms will not support decoupling of agent creation and starting (e.g. for remote requests in JADE).
agentidentifier [out]	AgentIdentifier	Output parameter containing the agent identifier of the created agent.

**: optional parameter*

To use the “ams_create_agent”-goal, you must first of all include the AMS-capability in your ADF (if not yet done in order to use other goals of the AMS-capability) and set a reference to the goal as described in Figure 16.1, “ Including the AMS capability and the ams_create_agent-goal ”. The name of the goal reference can be arbitrarily chosen, but it will be assumed here for convenience that the same as the original name will be used.

```
...
<capabilities>
    <capability name="amscap" file="jadex.planlib.AMS" />
    ...
</capabilities>
...
<goals>
    <achievegoalref name="ams_create_agent">
        <concrete ref="amscap.ams_create_agent" />
    </achievegoalref>
    ...
</goals>
...
```

Figure 16.1. Including the AMS capability and the ams_create_agent-goal

Now you can use this goal to create an agent in your plan:

```
public void body()
{
    ...
    IGoal ca = createGoal("ams_create_agent");
    ca.getParameter("type").setValue("mypackage.MyAgent");
    dispatchSubgoalAndWait(ca);
    AgentIdentifier createdagent = (AgentIdentifier)
        ca.getParameter("agentidentifier").getValue();
    ...
```

```
}
```

Figure 16.2. Creating an agent on a local platform

In listing Figure 16.2, “Creating an agent on a local platform” - in order to create an agent - you instantiate a new goal using the `createGoal()`-method with the paramter “ams_create_agent”. Then you set its parameters to the desired values, dispatch the subgoal and wait. After the goal has succeeded, you can fetch the `AgentIdentifier` of the created agent by calling the `getValue()`-method on the parameter “agentidentifier”.

The same goal is used for remote creation of an agent:

```
public void body()
{
    AgentIdentifier ams = new AgentIdentifier("ams@remoteplatform",
        new String[]{"nio-mtp://134.100.11.232:5678"});
    IGoal ca = createGoal("ams_create_agent");
    ca.getParameter("type").setValue("mypackage.MyAgent");
    ca.getParameter("ams").setValue(ams);
    dispatchSubgoalAndWait(ca);
    AgentIdentifier createdagent = (AgentIdentifier)
        ca.getParameter("agentidentifier").getValue();
   ...
}
```

Figure 16.3. Creating an agent on a remote platform

In the listing Figure 16.3, “Creating an agent on a remote platform” you can see how to create an agent on a remote platform using its remote AMS. In order to do so, it's of course crucial that you know at least one address of the remote AMS. Moreover, the corresponding transport must be available on the local platform. The transport used by the other platform can be recognized by the prefix of the address (ending with the ://). In this case the prefix is `nio-mtp://`, which represents the transport `jadex.adapter.standalone.transport.niotcpmtp.NIOTCPTransport`.

If you know the address of the remote AMS and you're sure that the local platform supports its transport, you must instantiate an `AgentIdentifier` and set its name and address to that of the AMS that shall create a new agent.

Thereafter you can instantiate a new goal using the `createGoal()`-method with the paramter “ams_create_agent”. Then you set its parameters to the desired values, dispatch the subgoal and wait. After the goal has succeeded, you can fetch the `AgentIdentifier` of the created agent by calling the `getValue()`-method on the parameter “agentidentifier”.

16.1.2. Starting an Agent

The AMS offers the goal “ams_start_agent” to give the application programmer the possibility to start agents, both on the local or a remote platform. This goal is useful if the creation and starting of agents should be decoupled, meaning that you want first to create an agent and start it afterwards.

The goal has the following parameters:

Table 16.2. Parameters for ams_start_agent

Name	Type	Description
agentidentifier	AgentIdentifier	The identifier of the agent that should be started.
ams *	AgentIdentifier	The AMS agent identifier (only needed for remote requests).

**: optional parameter*

To use the “ams_start_agent”-goal, you must first of all include the AMS-capability in your ADF (if not yet done in order to use other goals of the AMS-capability) and set a reference to the goal as described in Figure 16.4, “ Including the AMS capability and the ams_start_agent-goal ”.

```
...
<capabilities>
    <capability name="amscap" file="jadex.planlib.AMS" />
    ...
</capabilities>
...
<goals>
    <achievegoalref name="ams_start_agent">
        <concrete ref="amscap.ams_start_agent" />
    </achievegoalref>
    ...
</goals>
...
```

Figure 16.4. Including the AMS capability and the ams_start_agent-goal

Thus you can start an agent in your plan:

```
public void body()
{
   // Fetching the agent identifier after creating
   AgentIdentifier createdagent = (AgentIdentifier)
       ca.getParameter("agentidentifier").getValue();

   IGoal sa = createGoal("ams_start_agent");
   // sa.getParameter("ams").setValue(ams);  // Set ams in case of remote platform
   sa.getParameter("agentidentifier").setValue(createdagent);
   dispatchSubgoalAndWait(sa);
   ...
}
```

Figure 16.5. Starting an agent on a local/remote platform

In listing Figure 16.5, “Starting an agent on a local/remote platform” - in order to start an agent - you instantiate a new goal using the `createGoal()`-method with the paramter “ams_start_agent”. Then you set its agentidentifier-parameter to the desired value, dispatch the subgoal and wait for success. If you want to start the agent on a remote platform the only difference is that you need to supply the agent identifier of the remote AMS.

16.1.3. Destroying an Agent

The AMS offers the goal “ams_destroy_agent” to give the application programmer the possibility to destroy agents, both on a local as well as on remote platforms.

The goal has the following parameters:

Table 16.3. Parameters for ams_destroy_agent

Name	Type	Description
`agentidentifier`	`AgentIdentifier`	The identifier of the agent that should be destroyed.
`ams` *	`AgentIdentifier`	The AMS agent identifier (only needed for remote requests).

**: optional parameter*

To use the “ams_destroy_agent”-goal, you must first of all include the AMS-capability in your ADF (if not yet done in order to use other goals of the AMS-capability) and set a reference to the goal as described in Figure 16.6, “ Including the AMS capability and the ams_destroy_agent-goal ”.

```
...
<capabilities>
    <capability name="amscap" file="jadex.planlib.AMS" />
    ...
</capabilities>
...
<goals>
    <achievegoalref name="ams_destroy_agent">
        <concrete ref="amscap.ams_destroy_agent" />
    </achievegoalref>
    ...
</goals>
...
```

Figure 16.6. Including the AMS capability and the ams_destroy_agent-goal

Thus you can destroy an agent in your plan:

```
public void body()
{
   IGoal da = createGoal("ams_destroy_agent");
   da.getParameter("agentidentifier").setValue(createdagent);
   // da.getParameter("ams").setValue(ams);  // Set ams in case of remote platform
   dispatchSubgoalAndWait(da);
   ...
}
```

Figure 16.7. Destroying an agent on a local/remote platform

In listing Figure 16.7, “Destroying an agent on a local/remote platform” - in order to destroy an agent - you instantiate a new goal using the `createGoal()`-method with the paramter “ams_destroy_agent”. Then you set its agentidentifier-parameter to the desired value, dispatch the subgoal and wait for success. The same goal is used to destroy a remote agent. In this case you only have to additionally supply the remote AMS agent identifier.

16.1.4. Suspending an Agent

The AMS offers the goals “ams_suspend_agent” and “ams_resume_agent” in order to suspend the execution of an agent and later resume. When an agent gets suspended the platform will not process any actions of this agent. Nevertheless, the agent is able to receive messages from other agents and will process them when its execution is resumed.

The “ams_suspend_agent”-goal has the following input parameters:

Table 16.4. Parameters for ams_destroy_agent

Name	Type	Description
`agentidentifier`	`AgentIdentifier`	The identifier of the agent that should be suspended.
`ams` *	`AgentIdentifier`	The AMS agent identifier (only needed for remote requests).
`agentdescription [out]`	`AMSAgentDescription`	This output parameter contains the possibly changed AMSAgentDescription of the suspended agent.

**: optional parameter*

To use the “ams_suspend_agent”-goal, you must first of all include the AMS-capability in your ADF (if not yet done in order to use other goals of the AMS-capability) and set a reference to the goal as described in Figure 16.8, “ Including the AMS capability and the ams_suspend_agent-goal ”.

```
...
<capabilities>
    <capability name="amscap" file="jadex.planlib.AMS" />
    ...
</capabilities>
...
<goals>
    <achievegoalref name="ams_suspend_agent">
        <concrete ref="amscap.ams_suspend_agent" />
    </achievegoalref>
    ...
</goals>
...
```

Figure 16.8. Including the AMS capability and the ams_suspend_agent-goal

Thus you can suspend an agent in your plan:

```
public void body()
{
   AgentIdentifier agent;  // The agent to suspend
   ...
   IGoal sa = createGoal("ams_suspend_agent");
   sa.getParameter("agentidentifier").setValue(agent);
   // sa.getParameter("ams").setValue(ams);  // Set ams in case of remote platform
   dispatchSubgoalAndWait(sa);
   ...
}
```

Figure 16.9. Suspending an agent on a local/remote platform

In listing Figure 16.9, “Suspending an agent on a local/remote platform” - in order to suspend an agent - you instantiate a new goal using the `createGoal()`-method with the paramter “ams_suspend_agent”. Then you set its agentidentifier-parameter to the desired value, dispatch the subgoal and wait for success. As result the goal returns a possibly modified AMS agent description of the suspended agent. The same goal is used to suspend a remote agent. In this case you only have to additionally supply the remote AMS agent identifier.

16.1.5. Resuming an Agent

If you want to resume a suspended agent you can use the goal “ams_resume_agent”. It offers the following input parameters:

Table 16.5. Parameters for ams_resume_agent

Name	Type	Description
`agentidentifier`	`AgentIdentifier`	The identifier of the agent that you want to resume its execution.
`ams` *	`AgentIdentifier`	The AMS agent identifier (only needed for remote requests).
`agentdescription [out]`	`AMSAgentDescription`	The output parameter of this goal contains the possibly changed AMSAgentDescription.

**: optional parameter*

To use the “ams_resume_agent”-goal, you must first of all include the AMS-capability in your ADF (if not yet done in order to use other goals of the AMS-capability) and set a reference to the goal as described in Figure 16.10, “ Including the AMS capability and the ams_resume_agent-goal ”.

```
...
<capabilities>
    <capability name="amscap" file="jadex.planlib.AMS" />
    ...
</capabilities>
...
<goals>
    <achievegoalref name="ams_resume_agent">
        <concrete ref="amscap.ams_resume_agent" />
    </achievegoalref>
    ...
</goals>
...
```

Figure 16.10. Including the AMS capability and the ams_resume_agent-goal

Thus you can resume an agent in your plan:

```
public void body()
{
   AgentIdentifier agent;  // The agent to resume
   ...
   IGoal ra = createGoal("ams_resume_agent");
   ra.getParameter("agentidentifier").setValue(agent);
```

```
   // ra.getParameter("ams").setValue(ams);  // Set ams in case of remote platform
   dispatchSubgoalAndWait(ra);
   ...
}
```

Figure 16.11. Resuming an agent on a local/remote platform

In listing Figure 16.11, “Resuming an agent on a local/remote platform” - in order to resume an agent - you instantiate a new goal using the `createGoal()`-method with the paramter “ams_resume_agent”. Then you set its agentidentifier-parameter to the desired value, dispatch the subgoal and wait for success. As result the goal returns a possibly modified AMS agent description of the resumed agent. The same goal is used to resume a remote agent. In this case you only have to additionally supply the remote AMS agent identifier.

16.1.6. Searching for Agents

The goal "ams_search_agents" allows you to search for agents, both on the local platform and on remote platforms, thereby determining if the agent is available at all and learning about its state (e.g. active or suspended).

The goal has the following parameters:

Table 16.6. Parameters for ams_search_agents

Name	Type	Description
`ams` *	`AgentIdentifier`	The AMS agent identifier (only needed for remote requests).
`constraints` *	`SearchConstraints`	Representation of a set of constraints to limit the search process. See FIPA Agent Management Specification[1].
`description`	`AMSAgentDescription`	The AMSAgentDescription of the agent that you search for.
`result [set][out]`	`AMSAgentDescription`	This output parameter set contains the agent descriptions that have been found.

**: optional parameter*

To use the “ams_search_agents”-goal, you must first of all include the AMS-capability in your ADF (if not yet done in order to use other goals of the AMS-capability) and set a reference to the goal as described in Figure 16.12, “ Including the AMS capability and the ams_search_agents-goal ”.

```
...
<capabilities>
    <capability name="amscap" file="jadex.planlib.AMS" />
    ...
</capabilities>
...
<goals>
    <achievegoalref name="ams_search_agents">
        <concrete ref="amscap.ams_search_agents" />
```

[1] http://www.fipa.org/specs/fipa00023/XC00023H.html#_Toc526742642

```
    </achievegoalref>
    ...
</goals>
...
```

Figure 16.12. Including the AMS capability and the ams_search_agents-goal

To search for agents in your plan use the goal in the following manner:

```
public void body()
{
   AMSAgentDescription desc = new AMSAgentDescription(new AgentIdentifier("a1", true));
   IGoal search = createGoal("ams_search_agents");
   search.getParameter("description").setValue(desc);
   // search.getParameter("ams").setValue(ams);  // Set ams in case of remote platform
   dispatchSubgoalAndWait(search);
   AMSAgentDescription[] result = (AMSAgentDescription[])
           search.getParameterSet("result").getValues();
   ...
}
```

Figure 16.13. Searching an agent on a local/remote platform

In listing Figure 16.13, “Searching an agent on a local/remote platform” - in order to search for an agent - you instantiate a new goal using the `createGoal()`-method with the paramter “ams_search_agents”. The search is constrained by an AMS agent description that need to be provided. You could e.g. create an `AMSAgentDescription` with a new `AgentIdentifier` and the boolean `true` for a local agent as parameter, that is defined only by its name. Then you set its description-parameter to that just created `AMSAgentDescription`, dispatch the subgoal and wait for success. Supplying an empty agent description with agent identifier of null allows to perform an unconstrained search, i.e. returning all agents on the platform. In case of a remote request you have to set the agent identifier of the remote AMS well.

16.1.7. Shutting Down a Platform

The goal “ams_shutdown_platform” shuts down the platform on which a given AMS is running.

This goal has the following parameters:

Table 16.7. Parameters for ams_shutdown_platform

Name	Type	Description
`ams` *	`AgentIdentifier`	The AMS agent identifier (only needed for remote requests).

**: optional parameter*

To use the “ams_shutdown_platform”-goal, you must first of all include the AMS-capability in your ADF (if not yet done in order to use other goals of the AMS-capability) and set a reference to the goal as described in Figure 16.14, “ Including the AMS capability and the ams_shutdown_platform-goal ”.

```
...
<capabilities>
    <capability name="amscap" file="jadex.planlib.AMS" />
    ...
</capabilities>
...
<goals>
    <achievegoalref name="ams_shutdown_platform">
        <concrete ref="amscap.ams_shutdown_platform" />
    </achievegoalref>
    ...
</goals>
...
```

Figure 16.14. Including the AMS capability and the ams_shutdown_platform-goal

```
public void body()
{
   IGoal sd = createGoal("ams_shutdown_platform");
   // sd.getParameter("ams").setValue(ams);  // Set ams in case of remote platform
   dispatchSubgoal(sd);
}
```

Figure 16.15. Shutting down a local/remote platform

In order to shutdown the local platform you instantiate a new goal using the `createGoal()`-method with the paramter “ams_shutdown_platform” and then dispatch the goal (see listing Figure 16.15, “Shutting down a local/remote platform”). If you want to shutdown a remote platform you have additionally to set the value of the ams parameter to the agent identifier of the remote platform AMS.

16.2. The Directory Facilitator (DF) Capability

The directory facilitator allows the agents to register their services and search for the services offered by other agents. In order to avoid that there are registered services in the DF, that are no longer available, e.g. because the agent offering the service has been destroyed without deregistering its services, Jadex can use “lease times”, that is, the agent's service is only kept registered for that lease time and has to be refreshed by the agent before the lease time expires.

So if you're working with lease times, you have two possibilities of registering an agent. Either you register the agent's service only for the lease time by using the achieve-goal “df_register” with the option to manually refresh the registration with a “df_modify” achieve-goal, or you use the maintain-goal “df_keep_registered” that refreshes the registration automatically.

Depending on the kind of registration you chose, you can deregister an agent, either by using the “df_deregister”-goal as counterpart of “df_register” or you must first call the method drop() on the instance of the maintain-goal “df_keep_registered” and afterwards use a “df_deregister”-goal or keep the (now outdated registration) until the lease time expires.

The DF works with service descriptions. They are specified in the FIPA Agent Management Specification[2]. Generally the agent services are described in form of DF service descriptions which belong to a DF agent de-

[2] http://www.fipa.org/specs/fipa00023/XC00023H.html#_Toc526742641

scription that additionally might contain information about the agent itself.

Concretely this means the DF capability can be used for

- Section 16.2.1, “Registering an Agent Description”
- Section 16.2.2, “Keeping an agent description registered”
- Section 16.2.3, “Modifying a registration”
- Section 16.2.4, “Deregistration of services”
- Section 16.2.5, “Searching for agents and services”

16.2.1. Registering an Agent Description

For the registration of an agent description at a DF the “df_register”-goal can be used. This goal is immediately finished after the registration has been performed (or failed). When using “lease times” you can register a service temporarily by using the “df_register”- goal (see Section 16.2, “The Directory Facilitator (DF) Capability”). When not using “lease times” you can use this goal to register the service permanently.

The goal has the following parameters:

Table 16.8. Parameters for df_register

Name	Type	Description
`description`	`AgentDescription`	The agent description containing an arbitrary number of service descriptions to beregistered.
`df` *	`AgentIdentifier`	The DF at which the service shall be registered. Can be used for remote registration.
`leasetime` *	`Long`	The duration of the registration in ms. When a permanent registration is desired, the lease time can be ommitted.
`result [out]`	`AgentDescription`	This output parameter contains the AgentDescription hat has been registered.

**: optional parameter*

To use the “df_register”-goal, you must first of all include the DF-capability in your ADF (if not yet done in order to use other goals of the DF-capability) and set a reference to the goal as described in Figure 16.16, “ Including the DF capability and the df_register-goal ”.

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

```
    <achievegoalref name="df_register">
        <concrete ref="dfcap.df_register"/>
    </achievegoalref>
    ...
</goals>
...
```

Figure 16.16. Including the DF capability and the df_register-goal

To register an agent with the above mentioned goal proceed in the following way:

```
public void body()
{
    // Create an agent description with service descriptions

    ServiceDescription service1 = new ServiceDescription(
        "service_1", "type of service_1", "owner of service_1");

    // Define some characteristics of the services
    String[] languages = new String[]{"language_1", "language_2"};
    String[] ontologies = new String[]{"ontology_1", "ontology_2"};
    String[] protocols = new String[]{"protocol_1", "protocol_2"};

    Property[] properties = new Property[]{
        new Property("property_1", "value_1"),
        new Property("property_2", "value_2")};

    ServiceDescription service2 = SFipa.createServiceDescription(
        "service_2", "type of service_2", "owner of service_2",
        languages, ontologies, protocols, properties);

    ServiceDescription[] services = new ServiceDescription[]{service1, service2};

    AgentDescription desc = SFipa.createAgentDescription(null, services, null, null, null);

    IGoal register = createGoal("df_register");
    register.getParameter("description").setValue(desc);

    // For a remote DF
    // AgentIdentifier df = ...
    // register.getParameter("df").setValue(df);

    dispatchSubgoalAndWait(register);
   ...
}
```

Figure 16.17. Registering an agent description with the df_register goal at a DF

To register a service, you first have to create service descriptions.

When creating a new service description you can define the service name, the service type, the ownership of the service, an array of the languages understood by the service, an array of the ontologies known by the service, an array of the protocols used by the service and last any additional service properties in an array of `jadex.adapter.fipa.Property`. The property is used to specify parameter/value-pairs and the specification can be found at FIPA Agent Management Specification[3]

After having defined the service descriptions it is necessary to add those to an agent description for the agent.

[3] http://www.fipa.org/specs/fipa00023/XC00023H.html#_Toc526742645

For the registration a “df_register” goal needs to be created and dispatched. It is necessary to set the parameter "description" to the agent description that should be registered. If you want to register the services to a DF on a remote platform, you also have to set the parameter “df” to the agent identifier of the remote DF.

Finally, you can dispatch the goal and the service should be registered if the goal is successful.

As long as the agent's services are registered at the DF, the only way to add or remove a part of the services is to use the “df_modify”-method (see Section 16.2.3, “Modifying a registration” [107]. Other attempts of registration by the yet registered agent will fail unless you deregister the agent before.

Instead of instantiating the goals in a plan, you can also use an initial goal in your ADF:

```
...
<configurations>
    <configuration name="default">
        ...
        <goals>
            <initialgoal ref="df_register">
                <parameter ref="description">
                    <value>
                        SFipa.createAgentDescription(null,
                        SFipa.createServiceDescription("offered_service",
                        "my_agent", "me"))
                    </value>
                </parameter>
                <parameter ref="leasetime">
                    <value>100000</value>
                </parameter>
            </initialgoal>
        </goals>
        ...
    </configuration>
</configurations>
...
```

Figure 16.18. Registering an agent description with an initial goal

16.2.2. Keeping an agent description registered

In contrast to a “df_register”-goal a “df_keep_registered”-goal can be used when one wants to ensure that an agent description remains available at the DF as long as the goal is present in the agent. As it is a maintain goal it will persist unless it will be dropped intentionally. The goal tries to register the contained agent description at the DF an renews its registration whenever it threatens to expire. This means you can using you can register an agent description permanently even with “lease times” by using the “df_keep_registered”-goal (see Section 16.2, “The Directory Facilitator (DF) Capability”).

The goal has the following parameters:

Table 16.9. Parameters for df_keep_registered

Name	Type	Description
`description`	`AgentDescription`	The AgentDescription to be registered.
`df` *	`AgentIdentifier`	The (remote) DF at which the service shall be registered.

Name	Type	Description
`leasetime` *	`Long`	The duration of the registration in ms.
`buffertime`	`Long`	This parameter specifies when to refresh a registration with lease time again, e.g. a buffertime of 5000 ms means that 5 seconds before the lease time will expire the new registration process restarts. This shall guarantee that the agenent won't be deregistered before it is newly registered. The default is 3000 ms.
`result [out]`	`AgentDescription`	This output parameter contains the AgentDescription that has been registered at the DF.

**: optional parameter*

Please note that the buffertime is per default set to 3000 ms, so any lease time must be greater than 3000 ms if you don't change the buffertime.

To use the “df_keep_registered”-goal, you must first of all include the DF-capability in your ADF (if not yet done in order to use other goals of the DF-capability) and set a reference to the goal as described in Figure 16.19, “ Including the DF capability and the df_keep_registered-goal ”.

```
...
<capabilities>
    <capability name="dfcap" file="jadex.planlib.DF" />
    ...
</capabilities>
...
<goals>
    <achievegoalref name="df_keep_registered">
        <concrete ref="dfcap.df_keep_registered"/>
    </achievegoalref>
    ...
</goals>
...
```

Figure 16.19. Including the DF capability and the df_keep_registered-goal

Now you can register an agent:

```
public void body()
{
    AgentDescription desc = ...

    // Alternative for setting the lease time as goal-parameter
    //desc.setLeaseTime(new Date(System.currentTimeMillis() + 20000));

    IGoal keep = createGoal("df_keep_registered");
    keep.getParameter("description").setValue(desc);
    keep.getParameter("leasetime").setValue(new Long(20000));
    // For a remote DF
    // AgentIdentifier df = ...
    // keep.getParameter("df").setValue(df);
    dispatchSubgoalAndWait(df_keep_registered, 2000);
    AgentDescription resdesc = ((AgentDescription)df_keep_registered
```

```
        .getParameter("result").getValue());
    ServiceDescription[] descriptions = resdesc.getServices();
    ...
}
```

Figure 16.20. Registering an agent description with the df_keep_registered goal at a DF

In order to keep an agent description registered, you have to create the goal using the `createGoal()`-method and then set its parameters to the agent description that you want to register and the lease time to the preferred value. If you want to use a remote DF you have additionally to specify its agent identifier in the corresponding parameter. Thereafter, you dispatch the goal and wait for until it is registered and the wait returns. Now the agent description will be registered as long as the agent is alive and the goal is not dropped. The current agent description that was registered can be fetched using the result parameter as shown in the code snippet above.

When dispatching the “keep registered” goal as a subgoal, it will only be adopted as long the corresponding plan is running. If you want the agent to maintain its registration even after the plan has finished, you should dispatch the goal as a top-level goal. Instead of disptaching the top-level goal from a plan, you can also use an initial goal in your ADF:

```
...
<configurations>
    <configuration name="default">
        ...
        <goals>
            <initialgoal ref="df_keep_registered">
                <parameter ref="description">
                    <value>
                        SFipa.createAgentDescription(null,
                        SFipa.createServiceDescription("offered_service",
                        "my_agent", "me"))
                    </value>
                </parameter>
                <parameter ref="leasetime">
                    <value>120000</value>
                </parameter>
            </initialgoal>
        </goals>
        ...
    </configuration>
</configurations>
...
```

Figure 16.21. Registering services as initial goal

16.2.3. Modifying a registration

In order to change an agent description that is already registered at the DF the “df_modify”-goal can be used, e.g. if you want to register more services or - in case of temporary registration - you want to increase the lease time.

The goal has the following parameters:

Table 16.10. Parameters for df_modify

Name	Type	Description
description	AgentDescription	The changed agent description.
df *	AgentIdentifier	The DF at which the service is registered.
leasetime *	Long	The duration of the registration in ms.
result [out]	AgentDescription	This output parameter contains the current agent description that is registered at the DF.

**: optional parameter*

To use the “df_modify”-goal, you have to include the DF-capability in your ADF (if not yet done in order to use other goals of the DF-capability) and set a reference to the goal as described in Figure 16.22, “ Including the DF capability and the df_modify-goal ”.

```
...
<capabilities>
    <capability name="dfcap" file="jadex.planlib.DF" />
    ...
</capabilities>
...
<goals>
    <achievegoalref name="df_modify">
        <concrete ref="dfcap.df_modify"/>
    </achievegoalref>
    ...
</goals>
...
```

Figure 16.22. Including the DF capability and the df_modify-goal

To modify a registration use the df_modify-goal that way:

```
public void body()
{
    AgentDescription desc = ...
    IGoal modify = createGoal("df_modify");
    modify.getParameter("description").setValue(desc);
    // Set the leasetime up to one day
    modify.getParameter("leasetime").setValue(new Long(86400000));
    // For a remote DF
    // AgentIdentifier df = ...
    // modify.getParameter("df").setValue(df);
    dispatchSubgoalAndWait(modify);
    AgentDescription resdesc = ((AgentDescription)
        modify.getParameter("result").getValue());
    ...
}
```

Figure 16.23. Modifying a DF-registration

The modification of an already registered agent description requires that a “df_modify”-goal is created and its description parameter is set to the modified agent description. If the lease time should be refreshed you can set the lease duration in the leasetime parameter. If you want to use a remote DF you have to specify its agent iden-

tifier in the corresponding parameter.

Using the “result”-parameter, you can get the agent description with the services that have been registered and the current absolute lease time.

16.2.4. Deregistration of services

How you deregister services depends on the way that registered them. If you used the “df_register”-goal to register the services, you can use its pendant “df_deregister”. Whereas if you used the “df_keep_registered”-goal, you must first invoke the `drop()`-method on that goal (see Section 16.2.4, “Deregistration of services” [110]) and can then deregister the agent description by using the “df_deregister” goal.

The goal “df_deregister” has the following parameters:

Table 16.11. Parameters for df_deregister

Name	Type	Description
`description` *	`AgentDescription`	The AgentDescription to be removed from the DF.
`df` *	`AgentIdentifier`	The DF at which the service to remove is registered.

**: optional parameter*

To use the “df_deregister”-goal, you must first of all include the DF-capability in your ADF (if not yet done in order to use other goals of the DF-capability) and set a reference to the goal as described in Figure 16.24, “ Including the DF capability and the df_deregister-goal ”.

```
...
<capabilities>
    <capability name="dfcap" file="jadex.planlib.DF" />
    ...
</capabilities>
...
<goals>
    <achievegoalref name="df_deregister">
        <concrete ref="dfcap.df_deregister"/>
    </achievegoalref>
    ...
</goals>
...
```

Figure 16.24. Including the DF capability and the df_deregister-goal

```
public void body()
{
    IGoal deregister = createGoal("df_deregister");
    //AgentDescription desc = ...
    //deregister.getParameter("description").setValue(descript);

    dispatchSubgoalAndWait(deregister);
   ...
}
```

Figure 16.25. Deregistering a DF-registration

If you want to deregister the agents own agent description the goal just needs to be created and can directly be dispatched. In case of a deregistration of an agent description of another agent it is necessary to set the optional “description”-parameter of the goal to the agent description of that agent. This agent description should at least contain the agent identifier of the agent to deregister. If you only want to remove some of the registered services, you need to use the “df_modify”-goal.

As mentioned above, if you used the “df_keep_registered”-goal, you first have to get rid of that goal by invoking the method `drop()` on it (see Figure 16.26, “Dropping the df_keep_registered-goal”). Otherwise it will register the agent again.

```
public void body()
{
    IGoal keep = createGoal("df_keep_registered);
    ...
    dispatchSubgoal(keep);
    ...
    keep.drop();
    ...
}
```

Figure 16.26. Dropping the df_keep_registered-goal

16.2.5. Searching for agents and services

The “df_search”-goal offers the possibility to search for agents and services registered at a DF.

It has the following parameters:

Table 16.12. Parameters of the df_search-goal

Name	Type	Description
`constraints` *	`SearchConstraints`	Representation of a set of constraints to limit the searching. See FIPA Agent Management Specification[4].
`description`	`AgentDescription`	The AgentDescription that shall be searched for.
`df` *	`AgentIdentifier`	The DF that shall be searched. Necessary for remote search requests.
`result [set][out]`	`AgentDescription`	This output parameter set contains all the agent descriptions that have been found.

**: optional parameter*

To use the “df_search”-goal, you have to include the DF-capability in your ADF (if not yet done in order to use other goals of the DF-capability) and set a reference to the goal as described in Figure 16.27, “ Including the

[4] http://www.fipa.org/specs/fipa00023/XC00023H.html#_Toc526742642

DF capability and the df_search-goal ".

```
...
<capabilities>
    <capability name="dfcap" file="jadex.planlib.DF" />
    ...
</capabilities>
...
<goals>
    <achievegoalref name="df_search">
        <concrete ref="dfcap.df_search"/>
    </achievegoalref>
    ...
</goals>
...
```

Figure 16.27. Including the DF capability and the df_search-goal

```
public void body()
{
    // Search for a service with given type.
    IGoal df_search = createGoal("df_search");
    AgentDescription desc = new AgentDescription();
    desc.addService(new ServiceDescription(null, "type of service_1", null))
    df_search.getParameter("description").setValue(desc);
    // For a remote DF
    // AgentIdentifier df = ...
    // modify.getParameter("df").setValue(df);
    dispatchSubgoalAndWait(df_search);
    AgentDescription[] result = (AgentDescription[])df_search
        .getParameterSet("result").getValues();
    if(result.length != 0)
    {
        AgentIdentifier[] serviceproviders = new AgentIdentifier[result.length];
        for(int i = 0; i < result.length; i++)
            serviceproviders[i] = result[i].getName();
    }
    ...
}
```

Figure 16.28. Searching at a DF

In order to search for agents, you first have to create the "df_search"-goal, then you have to create the agent description you search for and set the goal's parameter to it. Optionally you can set some search constraints on the goal that can be used to limit the number of search results (per default this number is not limited). Thereafter you can dispatch the goal and wait. When the goal returns you can fetch the results by invoking `getParameterSet("result").getValues()` on the goal and cast it to an array of agent descriptions. If you are interested in the agent identifiers of the service providers you can retrieve them via calling `getName()` on an agent description.

16.3. The Interaction Protocols Capability

Interaction protocols are predefined patterns of allowed message sequences that are designed for different interaction purposes. As certain kinds of interaction objectives are useful in different kinds of application domains FIPA has standardized several domain-independent protocols. Jadex offers built-in support for most of these FIPA protocols and hence facilitates the task of building standardized interactions.

The FIPA specifications define interaction protocols by using AUML sequence diagrams [Bauer et al. 2001]. These sequence diagrams specify the roles of the participating agents, their cardinality and the allowed message occurrences. The specifications do not consider how these protocols should be systematically related to necessary domain activities. Therefore, the protocol- domain interactions have been analyzed and led to the extended specification of goal-oriented interaction protocols. These protocol descriptions devide each role into two distinct parts: the domain and the protocol layer. Together both layers encapsulate the whole functionality of a role. This separation has the objective to make explicit the interaction point between both. The interaction points are defined by the start and the end of the activity (denoted by arrows in the AUML diagrams) and also by the goal signature which shows the purpose of the activity. In addition, the goal signatures contain detailed information about the in- and out-parameters that are used to parametrize the invocations.

The protocols capability offers implementations of the following FIPA interaction protocol specifications:

- Section 16.3.1, “FIPA Request Interaction Protocol (RP)” (SC00026H[5])
- Section 16.3.2, “FIPA Contract Net Interaction Protocol (CNP)” (SC00029H[6])
- Section 16.3.3, “FIPA Iterated Contract Net Protocol (ICNP)” (SC00030H[7])
- Section 16.3.4, “FIPA English Auction Interaction Protocol (EA)” (XC00031F[8])
- Section 16.3.5, “FIPA Dutch Auction Interaction Protocol (DA)” (XC00032F[9])
- Section 16.3.6, “Abnormal Termination of Protocols” (see e.g. SC00026H[10])

The protocols capability contains the functionality of the initiator as well as the participant sides of the interactions. To use the functionality of the protocols capability it is necessary to include it within the capability section of the ADF. Generally, if the agent should be enabled to initiate protocols it is also required to reference the corresponding initiator goal types, e.g. for the request protocol the rp_initiate goal type. Within plans a protocol can be started by creating a goal instance of a initiator goal type, setting the needed parameter value and dispatching it. After the protocol has ended the results can be directly read out of the out-parameters of the goal.

On the other hand the participant side of the protocols capability depends not only referencing necessary receiver goals but also on belief settings. Per default the participant role of the capability is turned off meaning that no messages will be handled by it. If you want to use it for handling protocols it is necessary to determine as exactly as possible which kinds of initiator messages should be accepted. For that purpose for each of the supported protocols a filter belief has been defined, e.g. the belief rp_filter is used to determine which request messages should trigger the capability. If the initial fact is set to IFilter.ALWAYS all request messages will be delegated to the capability. Normally, it is advisable to further constrain the kinds of messages that should be routed into the capability by using a custom filter expression.

In the following the meaning and usage of the different protocols will be described in more detail.

16.3.1. FIPA Request Interaction Protocol (RP)

The Request Interaction Protocol (SC00026H[11]) manages the interaction consisting of one initiator and one participant agent. The initiator wants the participant to perform some action.

[5] http://www.fipa.org/specs/fipa00026/SC00026H.html
[6] http://www.fipa.org/specs/fipa00029/SC00029H.html
[7] http://www.fipa.org/specs/fipa00030/SC00030H.html
[8] http://www.fipa.org/specs/fipa00031/XC00031F.html
[9] http://www.fipa.org/specs/fipa00032/XC00032F.html
[10] http://www.fipa.org/specs/fipa00026/SC00026H.html#_Toc26669020
[11] http://www.fipa.org/specs/fipa00026/SC00026H.html

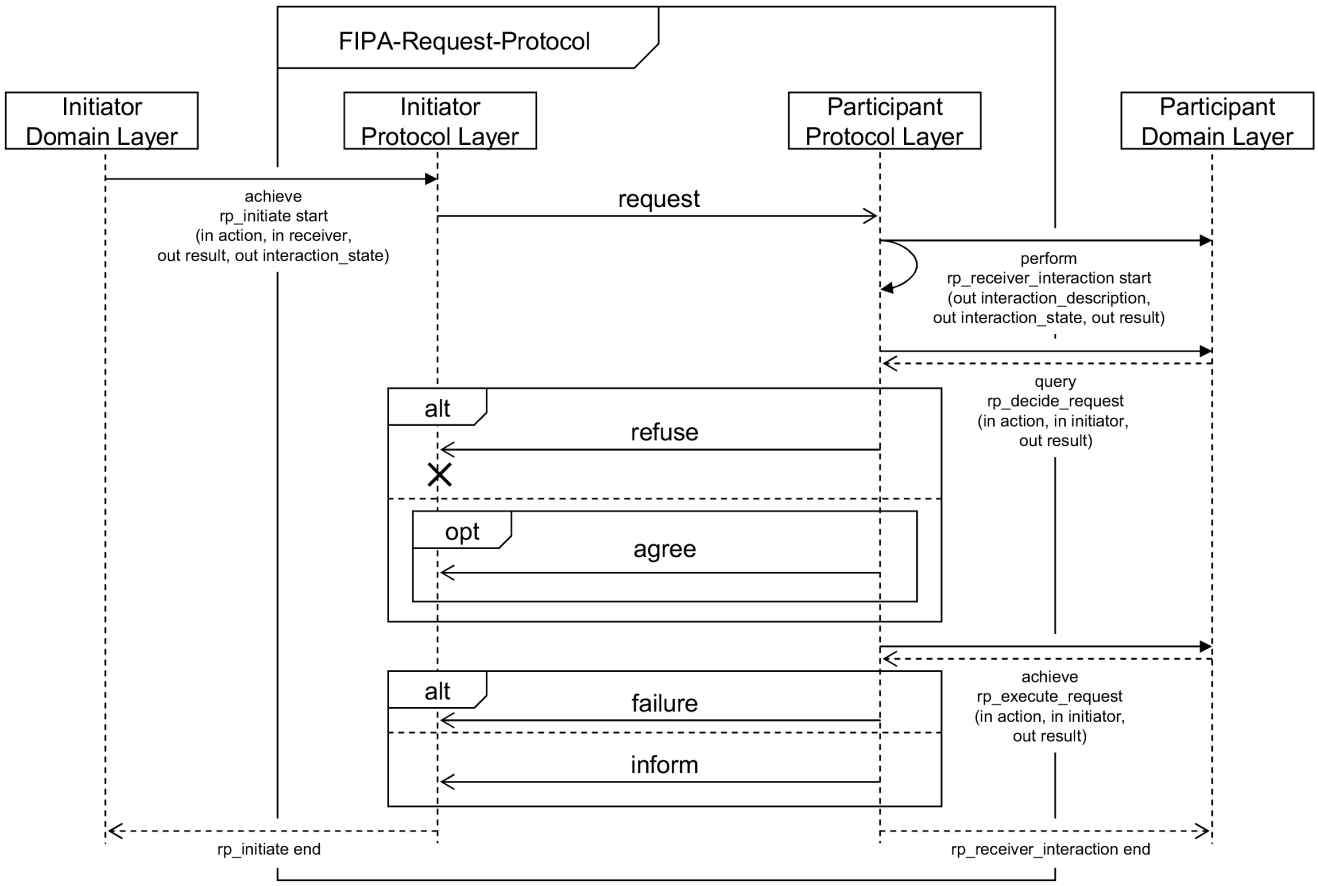

Figure 16.29. The Request Protocol

The protocol consists of a initiator and a participant side (cf. Figure 16.29, “ The Request Protocol ”). The initiator asks the participant to perform an action by sending a request message. When the participant receives this message it accepts or refuses to perform the action and dependent on that decision it either sends an optional agree message or a refuse message. If it has agreed, the participant subsequently performs the action and when it has finished, the participant sends a failure or an inform message. The inform message may be just a notification that the task was done or contain a result of the task execution.

In Jadex, the Protocols capability offers four corresponding goals. One for the initiator and three for the participant side. Namely “rp_initiate”, “rp_receiver_interaction”, “rp_decide_request” and “rp_execute_request”.

16.3.1.1. Initiator Side

From the view of the initiator side the protocol can be executed completely with dispatching the “rp_initiate”-goal and waiting for its completion.

It has the following parameters:

Table 16.13. Parameters for rp_initiate

Name	Type	Description
`action`	`Object`	The action to be performed by the requested agent.
`receiver`	`AgentIdentifier`	The receiver of the request.

Name	Type	Description
language *	String	The language to be used in the interaction (for the marshalling of the action and result objects).
ontology *	String	The ontology to be used in the interaction (for the marshalling of the action and result objects).
timeout *	Long	The timeout for the request.
interaction_state [out]	InteractionState	An object allowing to inspect the interaction state after protocol execution.
result [out]	Object	The result of the protocol execution (if sent in the inform message).

**: optional parameter*

In the code snippets below it is shown what needs to be done for starting a request protocol. First, the capability and the rp_initiate goal need to be included as depicted in Figure 16.30, “Inclusion of the rp_initiate goal”. From within a plan the protocol can be started by creating a corresponsing goal instance, setting its parameters to the desired values and dispatching it (cf. Figure 16.31, “Using the rp_initiate goal”). When the goal has finished the results of the protocol execution can be fetched.

```
...
<capabilities>
    <capability name="procap" file="jadex.planlib.Protocols"/>
    ...
</capabilities>
...
<goals>
    ...
    <achievegoalref name="rp_initiate">
        <concrete ref="procap.rp_initiate"/>
    </achievegoalref>
</goals>
...
```

Figure 16.30. Inclusion of the rp_initiate goal

```
public void body()
{
    AgentIdentifier rec = ...
    Object action = ...
    IGoal request = createGoal("rp_initiate");
    request.getParameter("action").setValue(action);
    request.getParameter("receiver").setValue(rec);
    dispatchSubgoalAndWait(request);
    Object res = request.getParameter("result").getValue();
    ...
}
```

Figure 16.31. Using the rp_initiate goal

16.3.1.2. Participant Side

On the participant side a request protocol can be triggered when the rp_filter belief value allows this (per default it is turned off). When a request message arrives that passes the rp_filter a new “rp_receiver_interaction” goal is created. This goal is present during the whole lifetime of the interaction and will contain the results and interaction state of the conversation. If the participant side wants to be informed about the outcome of its conversations it can do so by waiting for the completion of this goal (using the goalfinished trigger type). For the implementation of the requesters domain functionality two goals can be used. If the conversation is handled by the capability it will request domain activities by dispatching subgoals which should be handled by custom user plans.

The “rp_decide_request” goal is used by the capability to decide if the request should be accepted and the optional agree message should be sent. This goal is optional, which means that it needs not to be handled by the user. Per default the request is accepted and no agree message is sent. To accept and send the agree message the goal has be to handled and the accept return value should be set to true. If set to false a refuse message is sent and the interaction is immediately terminated.

Table 16.14. Parameters for rp_decide_request

Name	Type	Description
`initiator`	`AgentIdentifier`	The initiator of the protocol.
`action`	`Object`	The action to be performed by the requested agent.
`accept [out]`	`Boolean`	True, if the request should be accepted and the optional agree message should be sent.

The “rp_execute_request” goal is used by the capability to request the action being executed by the domain layer. The handling of this goal is mandatory for a successful protocol execution. If the domain plan for handling this goal has executed the requested action without problems it can write back to results of this execution to the result parameter of the goal. If the action execution fails this should be indicated by letting the plan fail (by raising an exception).

Table 16.15. Parameters for rp_execute_request

Name	Type	Description
`initiator`	`AgentIdentifier`	The initiator of the protocol.
`action`	`Object`	The action to be performed by the requested agent.
`result [out]` *	`Object`	The result of the action execution.

In the following code cutouts it is shown what needs to be done for handling a request protocol as receiver. First, the capability and the participant goals need to be included as indicated in Figure 16.32, “Inclusion of the participant goals and beliefs”. In the example also the optional rp_decide_request goal is used. It is shown that the agent in the example wants to handle all request messages via its protocol capability, because the rp_filter is set to IFilter.ALWAYS. In addition three plans have been declared, one for each of the protocol goals. The optional rp_finished_plan is used to detect whenever a request interaction has ended. The corresponding plan body can evaluate the results and may perform action in response to them.

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

<beliefs>
    ...
    <beliefref name="rp_filter" class="IFilter">
        <concrete ref="procap.rp_filter"/>
    </beliefref>
</beliefs>

<goals>
    ...
    <performgoalref name="rp_receiver_interaction">
        <concrete ref="procap.rp_receiver_interaction"/>
    </performgoalref>
    <achievegoalref name="rp_decide_request">
        <concrete ref="procap.rp_decide_request"/>
    </achievegoalref>
    <achievegoalref name="rp_execute_request">
        <concrete ref="procap.rp_execute_request"/>
    </achievegoalref>
</goals>

<plans>
    ...
    <plan name="rp_decide_request_plan">
        <parameter name="action" class="Object">
            <goalmapping ref="rp_decide_request.action"/>
        </parameter>
        <parameter name="accept" class="Boolean" direction="out">
            <goalmapping ref="rp_decide_request.accept"/>
        </parameter>
        <body>new MyDecideRequestPlan()</body>
        <trigger><goal ref="rp_decide_request"/></trigger>
    </plan>

    <plan name="rp_execute_request_plan">
        <parameter name="action" class="Object">
            <goalmapping ref="rp_execute_request.action"/>
        </parameter>
        <parameter name="result" class="Object" direction="out" optional="true">
            <goalmapping ref="rp_execute_request.result"/>
        </parameter>
        <body>new MyExecuteActionPlan()</body>
        <trigger><goal ref="rp_execute_request"/></trigger>
    </plan>

    <plan name="rp_finished_plan">
        <parameter name="interaction_state" class="InteractionState">
            <value>(InteractionState)((IRGoalEvent)$event).getGoal()
                .getParameter("interaction_state").getValue()</value>
        </parameter>
        <parameter name="result" class="Object">
            <value>((IRGoalEvent)$event).getGoal()
                .getParameter("result").getValue()</value>
        </parameter>
        <body>new MyResultPlan()</body>
        <trigger>
            <goalfinished ref="rp_receiver_interaction"/>
        </trigger>
    </plan>
</plans>
...
<configurations>
    <configuration name="default">
        <beliefs>
            <initialbelief ref="rp_filter">
```

```
                <fact>IFilter.ALWAYS</fact>
            </initialbelief>
            ...
        </beliefs>
        ...
    </configuration>
</configurations>
...
```

Figure 16.32. Inclusion of the participant goals and beliefs

In Figure 16.33, “ Plan body of the MyDecideRequestPlan ” the schematic content of the MyDecideRequestPlan body is depicted. The result of the acceptance test is stored in the `accept` parameter. Similarly, in Figure 16.34, “ Plan body of the MyExecuteRequestPlan ” an outline of the MyExecuteActionPlan is given. It reads out the action description, performs some operation for executing this action and then writes back the result to the `result` parameter.

```
public void body()
{
    Object action = getParameter("action").getValue();
    boolean accept = ... // Determine acceptance
    getParameter("accept").setValue(new Boolean(accept));
}
```

Figure 16.33. Plan body of the MyDecideRequestPlan

```
public void body()
{
    Object action = getParameter("action").getValue();
    // Perform the action
    result = ...
    // Store the result
    getParameter("result").setValue(result);
}
```

Figure 16.34. Plan body of the MyExecuteRequestPlan

16.3.2. FIPA Contract Net Interaction Protocol (CNP)

The Contract Net Protocol (SC00029H[12]) is used for negotiations between one initiator and an arbitrary number of participant agents. Its purpose is to delegate a task to other agents and let them execute the tasks on the initiator agent's behalf.

[12] http://www.fipa.org/specs/fipa00029/SC00029H.html

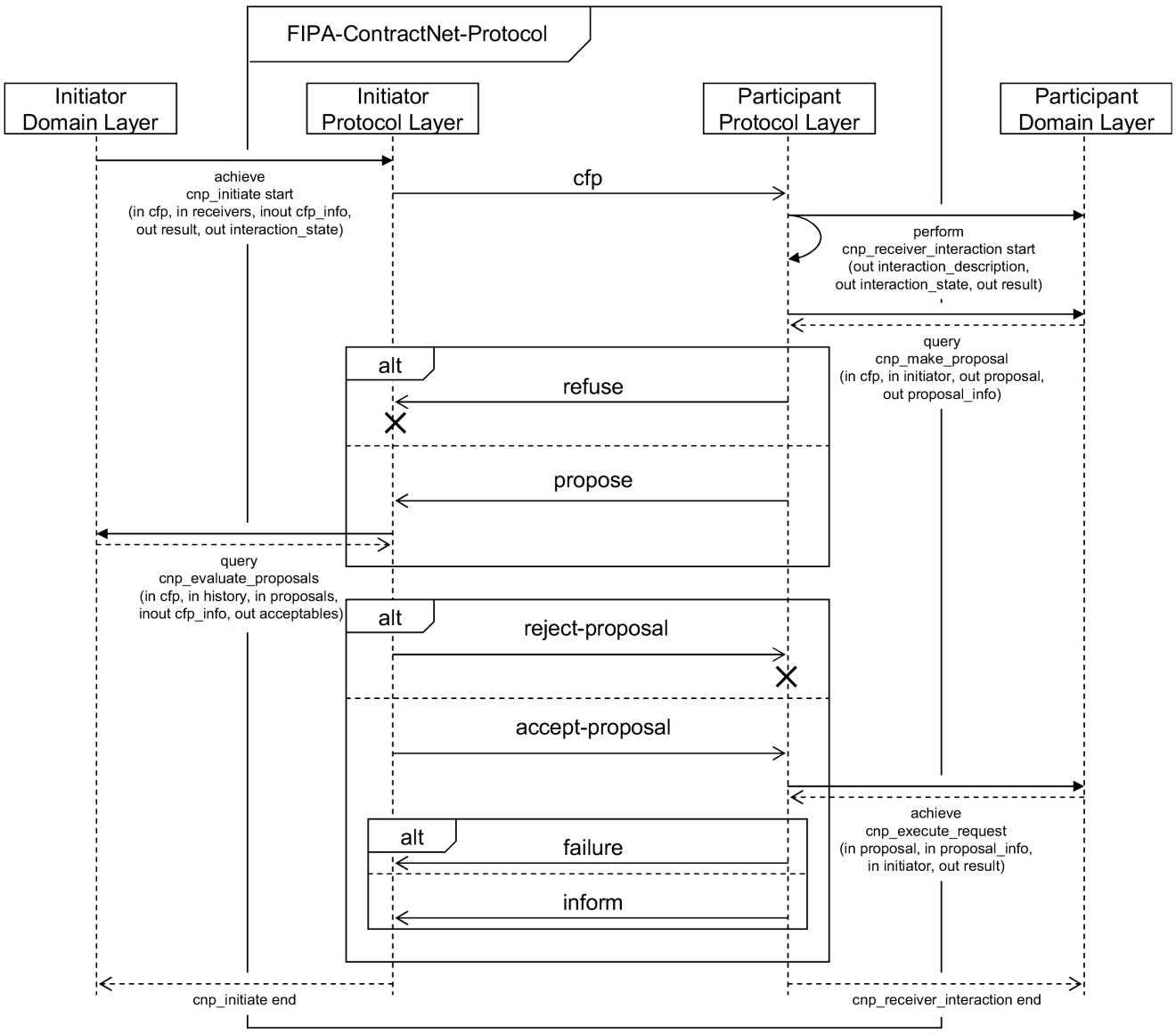

Figure 16.35. The Contract Net Protocol

There is an initiator and a participant side of the protocol. The initiator sends a call for proposal to an arbitrary number of participant agents. Then it waits for proposals of the participants until a given deadline has passed. Up to this deadline the participants can either send a propose message or a refuse message. When the deadline has passed, the initiator decides about all of the received proposals and either sends reject- or accept-proposal messages to the participants. The elected participants now have the commitment to perform the task. Thereafter they send send an inform message with the result (if any), or if the execution has failed, they send a failure message.

In Jadex, there are five corresponding goals, called “cnp_initiate”, “cnp_evaluate_proposals”, “cnp_receiver_interaction”, “cnp_make_proposal” and “cnp_execute_task”. The first two implement the initiator side, the other ones belong to the participant side.

16.3.2.1. Initiator Side

From the view of the initiator side the protocol can be started via dispatching the “cnp_initiate”-goal. During execution of the goal the protocol engine will automatically dispatch a “cnp_evaluate_proposals”-goal. This goal should be handled to decide wich proposals should be accepted.

The cnp_initiate-goal has the following parameters:

Table 16.16. Parameters for cnp_initiate

Name	Type	Description
`cfp`	`Object`	The call for proposals. Will be sent to all participants.
`cfp_info` *	`Object`	Some local information about the cfp. Will not be sent to the participants but is available as input in other initiator goals.
`ontology` *	`String`	The ontology to be used in the interaction (e.g. for the marshalling of the cfp and proposal objects).
`language` *	`String`	The language to be used in the interaction (e.g. for the marshalling of the cfp and proposal objects).
`timeout` *	`Long`	The timeout for the call for proposal.
`executeall` *	`Boolean`	Set to true, when all acceptable proposals should be executed. Defaults to false, such that only one successful proposal is accepted per default.
`interaction_state [out]`	`InteractionState`	An object allowing to inspect the interaction state after protocol execution.
`receivers [set]`	`AgentIdentifier`	The receivers of the call for proposal.
`result [set][out]`	`Object`	The results of the task executions.
`history [set][out]`	`NegotiationRecord`	The negotiation history.

**: optional parameter*

The second important goal on initiator side is the "cnp_evaluate_proposals" goal. Its in- and out-parameters are described below.

Table 16.17. Parameters for cnp_evaluate_proposals

Name	Type	Description
`cfp`	`Object`	The original call for proposals as initially sent to participants.
`cfp_info [inout]` *	`Object`	Local information about the cfp if set in the initiate goal.
`proposals [set]`	`ParticipantProposal`	The proposals received from the participants.
`history [set]` *	`NegotiationRecord`	Details about the negotiation.

Name	Type	Description
`acceptables [set][out]`	`ParticipantProposal`	The acceptable proposals that should be selected.

**: optional parameter*

In the code that follows it is depicted what needs to be done for executing a contract-net negotiation. First, the capability, the cnp_initiate and the cnp_evaluate_proposals goals need to be included as shown in Figure 16.36, “ Including the protocols capability and the goals for the contract-net protocol ”. In a plan the cnp_initiate goal needs to be created and its mandatory parameters should be set to the desired values. After dispatching the goal (cf. Figure 16.37, “Using the cnp_initiate goal”) one can wait for its completion and read out the results of the protocol execution. During the execution the engine will raise a cnp_evaluate_proposals goal which should be handled by a custom plan.

```
...
<imports>
    ...
    <import>jadex.planlib.*</import>
</imports>

<capabilities>
    ...
    <capability name="procap" file="jadex.planlib.Protocols"/>
</capabilities>
...
<goals>
    ...
    <achievegoalref name="cnp_initiate">
        <concrete ref="procap.cnp_initiate"/>
    </achievegoalref>
    <querygoalref name="cnp_evaluate_proposals">
        <concrete ref="procap.cnp_evaluate_proposals"/>
    </querygoalref>
    ...
</goals>
<plans>
    ...
    <plan name="evaluator_plan">
        <parameter name="cfp" class="Object">
            <goalmapping ref="cnp_evaluate_proposals.cfp"/>
        </parameter>
        <parameter name="cfp_info" class="Object" optional="true">
            <goalmapping ref="cnp_evaluate_proposals.cfp_info"/>
        </parameter>
        <parameterset name="proposals" class="ParticipantProposal">
            <goalmapping ref="cnp_evaluate_proposals.proposals"/>
        </parameterset>
        <parameterset name="history" class="NegotiationRecord" optional="true">
            <goalmapping ref="cnp_evaluate_proposals.history"/>
        </parameterset>
        <parameterset name="acceptables" class="ParticipantProposal" direction="out">
            <goalmapping ref="cnp_evaluate_proposals.acceptables"/>
        </parameterset>
        <body>new MyEvaluatorPlan()</body>
        <trigger>
            <goal ref="cnp_evaluate_proposals"/>
        </trigger>
    </plan>
</plans>
...
```

Figure 16.36. Including the protocols capability and the goals for the contract-net protocol

```
public void body()
{
    AgentIdentifier[] recs = ...
    Object cfp = ...
    IGoal cnpini = createGoal("cnp_initiate");
    cnpini.getParameterSet("receivers").addValues(recs);
    cnpini.getParameter("cfp").setValue(cfp);
    dispatchSubgoalAndWait(cnpini);
    Object res = request.getParameterSet("result").getValues();
    ...
}
```

Figure 16.37. Using the cnp_initiate goal

Not shown here is the content of the custom evaluator plan body which has the purpose to weight the proposals and choose those which should be performed. This can either be one or many and the acceptable proposals (if any) should be stored in the out-parameterset result. Depending on the value of the "executeall" parameter, the initiator will either sequentially accept the accpetable proposals in turn, until the first participant answers with an inform message and reject the remaining proposals ("executeall=false") or the initiator will accept all acceptable proposals in parallel rejecting only inacceptable proposals, thereby collecting potentially many results at once ("executeall=true"). The results of the execution are available in the "result" parameter set of the "cnp_initiate" goal. Additional information can be found in the negotiation history (parameter set "history"), which contains one `NegotiationRecord` object for the negotiation phase containg all proposals and their evaluations and one for the execution phase containing the executed proposals and the received results from the participant.

16.3.2.2. Participant Side

On the participant side a cnp protocol is triggered when the cnp_filter belief value allows this (per default it is turned off). When a cfp message arrives that passes the cnp_filter a new “cnp_receiver_interaction” goal is created. This goal is present during the whole lifetime of the interaction and will contain the results and interaction state of the conversation. If the participant side wants to be informed about the outcome of its conversations it can do so by waiting for the completion of this goal (using the goalfinished trigger type). For the implementation of the participant domain functionality two goals need to be supported. If the conversation is handled by the capability it will automatically request domain activities by dispatching subgoals which should be handled by custom user plans.

The “cnp_make_proposal” goal is used by the capability to indicate that a proposal for a cfp should be generated. The custom user plan that handles this goal finds all necessary information about the cfp in the parameters of the goal. On the basis of this knowledge it can decide to participate in the contract-net negotiation by creating a proposal and storing it into the proposal out-parameter.

The “cnp_make_proposal”-goal has the following parameters:

Table 16.18. Parameters for cnp_make_proposal

Name	Type	Description
`cfp`	`Object`	The call-for-propsal that can be handled.
`initiator`	`AgentIdentifier`	The agent identifier of the protocol initiator.
`proposal [out]`	`Object`	The proposal for the cfp that will be sent

Name	Type	Description
		back to the initiator.
`proposal_info [out]` *	`Object`	Some optional local data for the proposal. It will be available in the cnp_execute_task goal.

**: optional parameter*

If a proposal has been made and the initiator selects the participant as one of the winners a "cnp_execute_task" goal will be raised. A custom user plan should be provided that handles the goal, executes the task and reports back the results of the processing.

The “cnp_execute_task”-goal has the following parameters:

Table 16.19. Parameters for cnp_execute_task

Name	Type	Description
`proposal`	`Object`	The proposal that was sent to the initiator and has won.
`proposal_info` *	`Object`	The local proposal data that was generated by the cnp_make_proposal goal
`initiator`	`Object`	The initiator of the cfp.
`result [out]` *	`Object`	Information about the task execution.

**: optional parameter*

In the following code snippets it is shown what needs to be done for handling a contract-net protocol as receiver. First, the capability and the cnp_make_proposal, cnp_execute_task goals need to be included as indicated in Figure 16.38, “Inclusion of the participant goals and beliefs”. It is shown that the agent in the example wants to handle all cfp messages via its protocol capability, because the cnp_filter is set to IFilter.ALWAYS. In addition two plans have been declared, one for each of the protocol goals.

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

<beliefs>
    <beliefref name="cnp_filter" class="IFilter">
        <concrete ref="procap.cnp_filter"/>
    </beliefref>
    ...
</beliefs>

<goals>
    <querygoalref name="cnp_make_proposal">
        <concrete ref="procap.cnp_make_proposal"/>
    </querygoalref>
    <achievegoalref name="cnp_execute_task">
        <concrete ref="procap.cnp_execute_task"/>
    </achievegoalref>
    ...
</goals>
```

```
<plans>
    <plan name="cnp_make_proposal_plan">
        <parameter name="cfp" class="Object">
            <goalmapping ref="cnp_make_proposal.cfp"/>
        </parameter>
        <parameter name="initiator" class="AgentIdentifier">
            <goalmapping ref="cnp_make_proposal.initiator"/>
        </parameter>
        <parameter name="proposal" class="Object">
            <goalmapping ref="cnp_make_proposal.proposal"/>
        </parameter>
        <parameter name="proposal_info" class="Object">
            <goalmapping ref="cnp_make_proposal.proposal_info"/>
        </parameter>
        <body>new MyMakeProposalPlan()</body>
        <trigger>
            <goal ref="cnp_make_proposal"/>
        </trigger>
    </plan>

    <plan name="cnp_execute_task_plan">
        <parameter name="proposal" class="Object">
            <goalmapping ref="cnp_execute_task.proposal"/>
        </parameter>
        <parameter name="proposal_info" class="Object" optional="true">
            <goalmapping ref="cnp_execute_task.proposal_info"/>
        </parameter>
        <parameter name="initiator" class="AgentIdentifier">
            <goalmapping ref="cnp_execute_task.initiator"/>
        </parameter>
        <parameter name="result" class="Object" optional="true">
            <goalmapping ref="cnp_execute_task.result"/>
        </parameter>
        <body>new MyExecuteTaskPlan()</body>
        <trigger>
            <goal ref="cnp_execute_task"></goal>
        </trigger>
    </plan>
    ...
</plans>
...
<configurations>
    <configuration name="default">
        <beliefs>
            <initialbelief ref="cnp_filter">
                <fact>IFilter.ALWAYS</fact>
            </initialbelief>
            ...
        </beliefs>
        ...
    </configuration>
</configurations>
...
```

Figure 16.38. Inclusion of the participant goals and beliefs

Besides the inclusion of the relevant goals and beliefs the main task of the agent programmer consists in developing the two custom domain plans. Here in the example code they are named MyMakeProposalPlan and the MyExecuteTaskPlan. As the basic responsibilities of the two plans consists in performing domain tasks and writing back the results to the goals no further code snippets will be given here.

16.3.2.3. Simplified Protocol Usage

In many cases, the decisions an agent has to make during the execution of a protocol are quite simple and do

not necessarily have to be defined in terms of goals and plans. Therefore, the protocols capability offers some simple Java interfaces that can be implemented to describe these decisions instead of having to define plans. For the contract-net usually a plan has to be defined, to evaluate the received proposals and to decide about acceptable proposals. Alternatively, the `jadex.planlib.IProposalEvaluator` interface can be used for this purpose. An object implementing this interface can be passed in the "cfp_info" parameter of the "cnp_initiate" goal and will then automatically be used by a default plan in the protocols capability to handle the "evaluate_proposals" goal.

The `IProposalEvaluator` interface requires the `evaluateProposals()` method to be implemented, which features in essence the same parameters as the "evaluate_proposals" goal. To simplify the usage of the contract-net protocol even more, a default implementation is available (`jadex.planlib.ProposalEvaluator`), which handles some very common cases. The default implementation allows to determine acceptable proposals by comparing proposals or evaluations to a limit value, given in the constructor. Moreover, the evaluation process is implemented in three methods, which can be separately overwritten if needed, while reusing functionality of the other methods.

- The proposals are evaluated by calling the the `evaluateProposal()` method for each of the proposals. The evaluation result is written back into the proposal. The default implementation just checks, if the proposal object itself is suitable as an evaluation (i.e. if it is comparable).
- For each of the proposals, the acceptability is determined. By default, if a limit value is given, the proposal evaluations are compared to the limit value.
- Finally, the acceptable proposals are ordered by preference. As a default, the proposals are compared to each other and sorted according to a given ordering.

The usage of the default implementation is illustrated in Figure 16.39, “Simplified usage of the cnp_initiate goal”, in which a newly instantiated `ProposalInitiator` is used as cfp_info, to automatically accept all proposals greater or equal to 5.

```
public void body()
{
    AgentIdentifier[] recs = ...
    Object cfp = ...
    IProposalEvaluator cfp_info = new ProposalEvaluator(new Integer(5), false);
    IGoal cnpini = createGoal("cnp_initiate");
    cnpini.getParameterSet("receivers").addValues(recs);
    cnpini.getParameter("cfp").setValue(cfp);
    cnpini.getParameter("cfp_info").setValue(cfp_info);
    dispatchSubgoalAndWait(cnpini);
    Object res = request.getParameterSet("result").getValues();
    ...
}
```

Figure 16.39. Simplified usage of the cnp_initiate goal

16.3.3. FIPA Iterated Contract Net Protocol (ICNP)

The Iterated Contract Net Protocol (SC00030H[13]) is used for negotiations between one initiator and an arbitrary number of participant agents. Its purpose is to delegate a task to other agents and let them execute the tasks on the initiator agent's behalf. The only difference to the normal contract net protocol described in the last section is that more than one negotiation round may be performed.

[13] http://www.fipa.org/specs/fipa00030/SC00030H.html

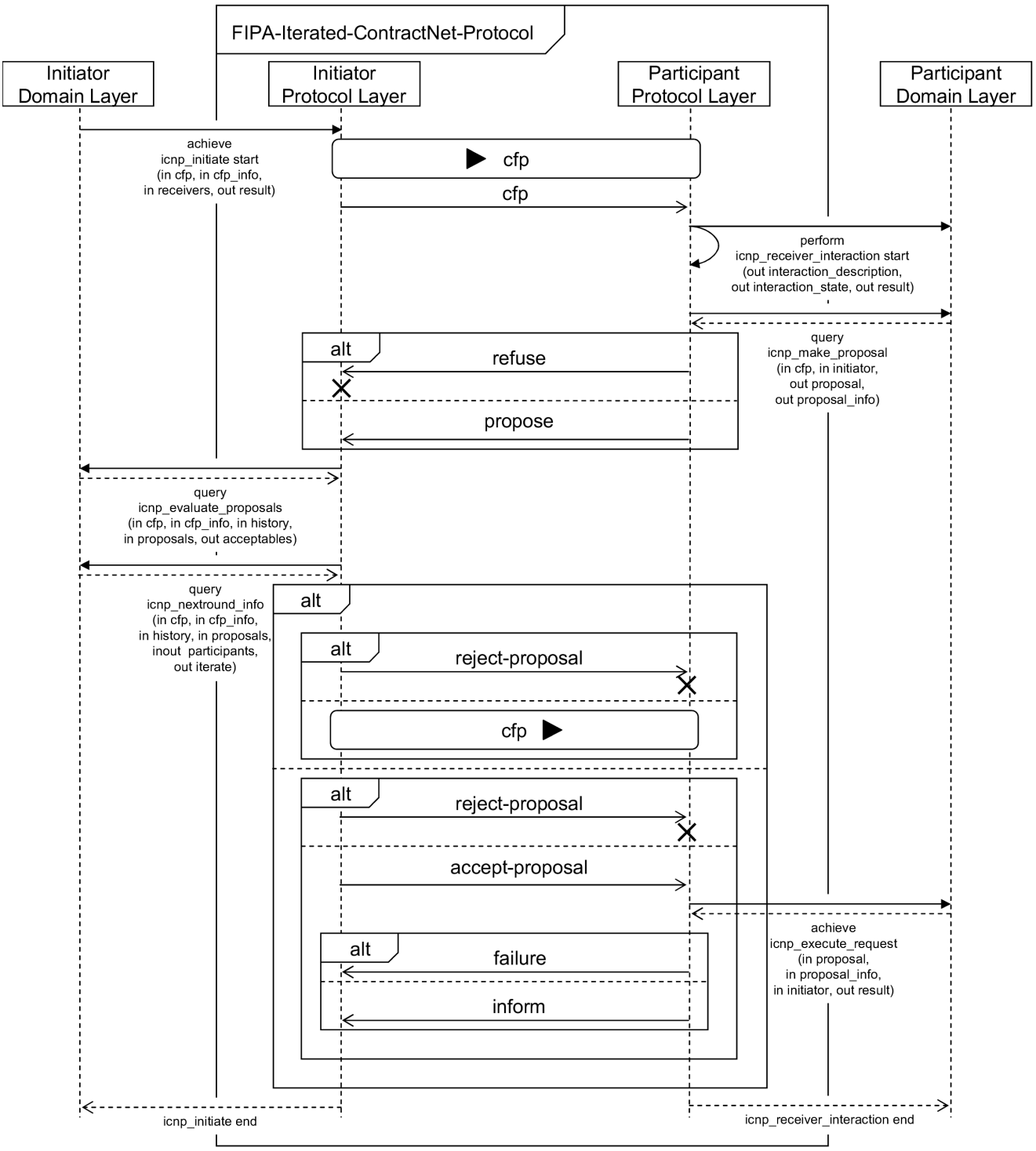

Figure 16.40. The Iterated Contract Net Protocol

There is an initiator and a participant side of the protocol. The initiator sends a call for proposal to an arbitrary number of participant agents. Then it waits for proposals of the participants until a given deadline has passed. Up to this deadline the participants can either send a propose message or a refuse message. When the deadline has passed, the initiator decides about all of the received proposals and evaluates if some proposals are acceptable (like in the non-iterated contract-net discussed in the last section). In a second step, the initiator decides, if another negotiation round should be initiated with a possibly refined cfp. New negotiation rounds can be performed with the same or any subset of the original participants. If a subset is chosen the excluded participants will be rejected immediately. When the initiator decides to accept some proposals (i.e. performing no further negotiation round), the winning participants have the commitment to perform the task. Thereafter they send

send an inform message with a result (if any), or if the execution has failed, they send a failure message.

In Jadex, there are six corresponding goals, called “icnp_initiate”, “icnp_evaluate_proposals”, “icnp_nextround_info”, “icnp_receiver_interaction”, “icnp_make_proposal” and “icnp_execute_task”. The first three implement the initiator side, the other ones belong to the participant side. The goals are very similar to the non-iterated version of the contract net protocol. The only new goal is the query “icnp_nextround_info” goal, which decides if another negotiation round is necessary and which settings should apply to the next round (if any). As the other goals are identical to their original non-iterated counterparts here only the “icnp_nextround_info” will be explained in detail. For details about the other goals please refer to the explanations from the last section.

16.3.3.1. Initiator Side

From the view of the initiator side the protocol can be started via dispatching the "icnp_initiate" goal. During execution of the goal the protocol engine will automatically dispatch a "icnp_evaluate_proposals" goal. This goal should be handled to decide wich proposals are acceptable. After the evaluation has taken place the engine raises a "icnp_nextound_info" goal which has the purpose to decide if another negotiation round should be performed and if yes, which settings should be used, concerning e.g. the participants, the possibly refined cfp and additional only locally availabe cfp data.

The "icnp_nextound_info" goal has the following parameters:

Table 16.20. Parameters for icnp_nextround_info

Name	Type	Description
`cfp [inout]`	`Object`	The original call for proposals as initially sent to participants. Can be refined for the next round.
`cfp_info [inout]` *	`Object`	Local information about the cfp if set in the "icnp_initiate" or "icnp_evaluate_proposals" goal. Can be refined for the next round.
`participants [set][inout]`	`AgentIdentifier`	The participants of the negotiation. Can be refined for the next round.
`proposals [set]`	`ParticipantProposal`	The proposals received from the participants including the evaluation value possibly created while handling the "icnp_evaluate_proposals" goal.
`history [set]`	`NegotiationRecord`	Contains details about all the negotiation rounds that have been performed so far.
`iterate [out]`	`Boolean`	Flag indicating the decision to iterate (set to true or false to end goal).

**: optional parameter*

In the code that follows it is depicted what needs to be done for executing an iterated contract-net negotiation. First, the capability, the "icnp_initiate" goal, the "icnp_evaluate_proposals" goal and the "icnp_nextround_info" goal need to be included as shown in Figure 16.41, “ Including the protocols capability and the goals for the iterated contract-net protocol ”. In a plan the "icnp_initiate" goal needs to be created and its mandatory paramet-

ers should be set to the desired values. After dispatching the goal (cf. Figure 16.42, “Using the icnp_initiate goal”) one can wait for its completion and read out the results of the protocol execution. During the execution the engine will raise one "icnp_evaluate_proposals" goal in every negotiation round when the collected proposals need to be evaluated. Additionally an "icnp_nextround_info" goal is created in every round to decide if another round is necessary.

```
...
<imports>
    ...
    <import>jadex.planlib.*</import>
</imports>

<capabilities>
    ...
    <capability name="procap" file="jadex.planlib.Protocols"/>
</capabilities>
...
<goals>
    ...
    <achievegoalref name="icnp_initiate">
        <concrete ref="procap.icnp_initiate"/>
    </achievegoalref>
    <querygoalref name="icnp_evaluate_proposals">
        <concrete ref="procap.icnp_evaluate_proposals"/>
    </querygoalref>
    <querygoalref name="icnp_nextround_info">
        <concrete ref="procap.icnp_nextround_info"/>
    </querygoalref>
</goals>
<plans>
    ...
    <plan name="evaluator_plan">
        <parameter name="cfp" class="Object">
            <goalmapping ref="icnp_evaluate_proposals.cfp"/>
        </parameter>
        <parameter name="cfp_info" class="Object" optional="true">
            <goalmapping ref="icnp_evaluate_proposals.cfp_info"/>
        </parameter>
        <parameterset name="proposals" class="ParticipantProposal">
            <goalmapping ref="icnp_evaluate_proposals.proposals"/>
        </parameterset>
        <parameterset name="history" class="NegotiationRecord" optional="true">
            <goalmapping ref="icnp_evaluate_proposals.history"/>
        </parameterset>
        <parameterset name="acceptables" class="ParticipantProposal" direction="out">
            <goalmapping ref="icnp_evaluate_proposals.acceptables"/>
        </parameterset>
        <body>new MyEvaluatorPlan()</body>
        <trigger>
            <goal ref="icnp_evaluate_proposals"/>
        </trigger>
    </plan>

    <plan name="icnp_nextround_plan">
        <parameter name="cfp" class="Object">
            <goalmapping ref="icnp_nextround_info.cfp"/>
        </parameter>
        <parameter name="cfp_info" class="Object" optional="true">
            <goalmapping ref="icnp_nextround_info.cfp_info"/>
        </parameter>
        <parameter name="iterate" class="Boolean" direction="out">
            <goalmapping ref="icnp_nextround_info.iterate"/>
        </parameter>
        <parameterset name="participants" class="AgentIdentifier" direction="inout">
            <goalmapping ref="icnp_nextround_info.participants"/>
        </parameterset>
        <parameterset name="proposals" class="ParticipantProposal">
            <goalmapping ref="icnp_nextround_info.proposals"/>
        </parameterset>
```

```
        <parameterset name="history" class="NegotiationRecord" optional="true">
            <goalmapping ref="icnp_nextround_info.history"/>
        </parameterset>
        <body>new MyNextroundPlan()</body>
        <trigger>
            <goal ref="icnp_nextround_info"/>
        </trigger>
    </plan>
</plans>
...
```

Figure 16.41. Including the protocols capability and the goals for the iterated contract-net protocol

```
public void body()
{
    AgentIdentifier[] recs = ...
    Object cfp = ...
    IGoal icnpini = createGoal("icnp_initiate");
    icnpini.getParameterSet("receivers").addValues(recs);
    icnpini.getParameter("cfp").setValue(cfp);
    dispatchSubgoalAndWait(icnpini);
    Object res = request.getParameterSet("result").getValues();
    ...
}
```

Figure 16.42. Using the icnp_initiate goal

Not shown here is the content of the custom evaluator plan body for the "icnp_evaluate_proposals" plan, which has the purpose to rate the proposals. It may identify one or many acceptable proposals (if any) which should be stored in the out-parameterset result. This information can also be helpful for deciding which agent should participate in the next negotiation round. The information is available in the "icnp_decide_iteration" goal, which decides if another negotiation round should be performed (out-parameter "iterate") and which settings to use (inout-parameters "cfp", "cfp_info", "participants").

As already described for the non-iterated contract-net, the execution of acceptable proposals can be influenced by the value of the "executeall" parameter. When set to false (default) the initiator will either sequentially accept the accpetable proposals in turn, until the first participant answers with an inform message and reject the remaining proposals. When set to true, initiator will accept all acceptable proposals in parallel rejecting only inacceptable proposals, thereby collecting potentially many results at once.

16.3.3.2. Participant Side

On the participant side an iterated contract-net protocol is triggered when the icnp_filter belief value allows this (per default it is turned off). When a cfp message arrives that passes the icnp_filter a new “icnp_receiver_interaction” goal is created. This goal is present during the whole lifetime of the interaction and will contain the results and interaction state of the conversation. If the participant side wants to be informed about the outcome of its conversations it can do so by waiting for the completion of this goal (using the goalfinished trigger type). For the implementation of the participant domain functionality two goals need to be supported. If the conversation is handled by the capability it will automatically request domain activities by dispatching subgoals which should be handled by custom user plans.

The "icnp_make_proposal" goal is used by the capability to indicate that a proposal for a cfp should be generated. The custom user plan that handles this goal finds all necessary information about the cfp in the parameters of the goal. On the basis of this knowledge it can decide to participate in the contract-net negotiation by creat-

ing a proposal and storing it into the proposal out-parameter. As the iterated version of the contract-net protocol can consist of an arbirary number of negotiation rounds the protocol engine will raise one "icnp_make_proposal" goal in each round.

If the initiator finally selects the participant as one of the winners a "icnp_execute_task" goal will be raised. A custom user plan should be provided that handles the goal, executes the task and reports back the results of the processing.

16.3.3.3. Simplified Protocol Usage

To simplify the usage of protocols for cases, where no complex plans are needed, the protocols capability offers some simple Java interfaces that can be implemented to describe the required decisions instead of having to define plans. The iterated contract-net usually requires one plan for evaluating proposals and another plan for preparing the settings for the next negotiation round. Alternatively to an evaluate proposals plan, the `IProposalEvaluator` interface can be used as already described for the non-iterated contract-net protocol in Section 16.3.2.3, “Simplified Protocol Usage”. For determing settings of a further negotiation round the interface `IQueryNextroundInfo` comes into play for the iterated contract-net. As described before, an object implementing one of this interfaces can be passed in the "cfp_info" parameter of the "cnp_initiate" goal and will then automatically be used by a default plan in the protocols capability to handle the "evaluate_proposals" or the "nextround_info" goal. If you want to supply implementations of both interfaces, you can use the wrapper class `ICNPHandler` designed specifically for this purpose.

The `IQueryNextroundInfo` interface requires the `queryNextroundInfo()` method to be implemented, which resembles the signature of the "nextround_info" goal. As the "nextround_info" goal contains several inout parameters, which can not be mapped directly to method paramters, a helper class `NextroundInfo` is used to hold the current cfp, cfp_info and participants and allows these settings to be changed for the next round. The `queryNextroundInfo()` should return true, when another negotiation round is required. Currently, no default implementation for the `IQueryNextroundInfo` interface is available, so you have to implement the required method yourself. The usage of both interfaces is illustrated in Figure 16.43, “Simplified usage of the icnp_initiate goal”, in which a newly instantiated `ICNPHandler` is used as cfp_info containing an `IProposalEvaluator` and an `IQueryNextroundInfo` object. The proposal evaluator will automatically accept all proposals greater or equal to 5 and the `IQueryNextroundInfo`, implemented as anonymous inner class, leads to three negotiation rounds being performed.

```
public void body() {
    AgentIdentifier[] recs = ...
    Object cfp = ...
    IProposalEvaluator pe = new ProposalEvaluator(new Integer(5), false);
    IQueryNextroundInfo qnri = new IQueryNextroundInfo() {
        public boolean queryNextroundInfo(NextroundInfo info, NegotiationRecord[] history, Participant
            return history.length < 3;
        }
    };
    Object cfp_info = new ICNPHandler(pe, qnri);
    IGoal icnpini = createGoal("icnp_initiate");
    icnpini.getParameterSet("receivers").addValues(recs);
    icnpini.getParameter("cfp").setValue(cfp);
    icnpini.getParameter("cfp_info").setValue(cfp_info);
    dispatchSubgoalAndWait(icnpini);
    Object res = request.getParameterSet("result").getValues();
    ...
}
```

Figure 16.43. Simplified usage of the icnp_initiate goal

16.3.4. FIPA English Auction Interaction Protocol (EA)

The FIPA English Auction Interaction Protocol (XC00031F[14]) describes an auction with steadily increased offers. The auction consists of an auctioneer and a set of bidders. The auctioneer chooses a start offer below the supposed market value. Round-by-round the offer is increased by the auctioneer as long as at least one bidder accepts the proposal. When no bidder is willing to accept the current offer of the auctioneer the auction has finished and the winner is the agent that has accepted the last offer. As the auctioneer starts the auction below the true market value it has the choice to not sell the good when the offered proposal is below his secret limit value.

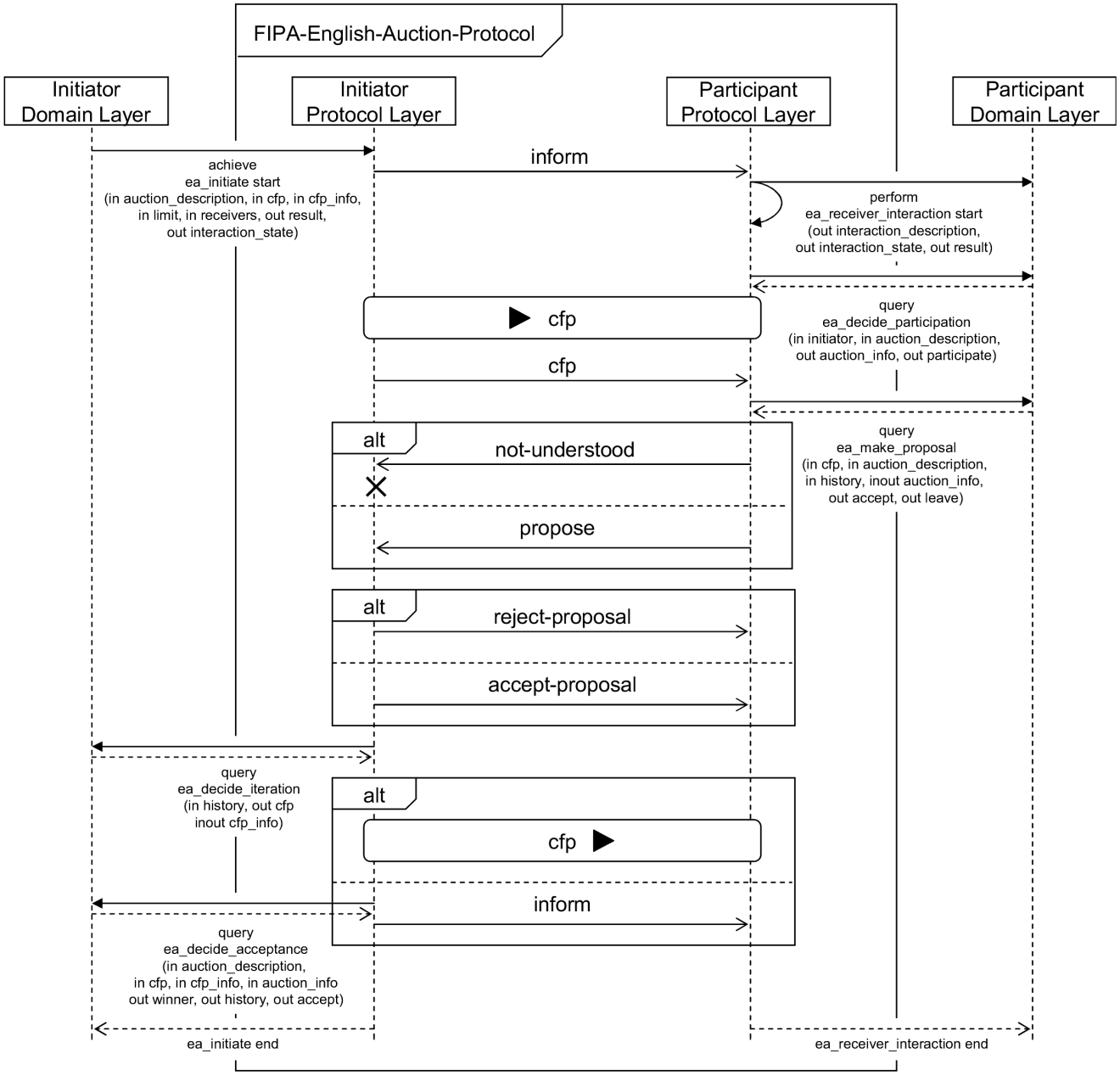

Figure 16.44. The English Auction Protocol

Technically the auction is specified as follows:

The auctioneer starts the auction by sending an “inform-start-of-auction”-message that contains information such as the topic, start time etc. about the planned auction. When the start time of auction has been reached the

[14] http://www.fipa.org/specs/fipa00031/XC00031F.html

auctioneer sends a call for proposal (CFP) to all bidders and waits. The bidders can react in three different ways. Firstly, a bidder can send a “not-understood”-message in order to signalize that it is not able to understand or not interested in the CFP. The auctioneer will subsequently exclude this bidder from the auction. Secondy, the bidder can decide to make an offer by sending a propose message. Thirdly, e.g., when the CFP exceeds the price the bidder is currently willing or allowed to pay, it may decide not to answer. In this case, the bidder will stay in the auction and may decide to bid again in a later negotiation round. Of course, the bidder, who won the last round, will usually not bid in the current round. The auctioneer waits for bids until the timeout of a negotiation round. If there is more than one proposal message, only the first one is accepted.[15] (Even if they arrive at the same time, they will technically not be processed concurrently. This is similar to a real English auction, because if two bidders raise their hands at the same time it is also nondeterministic which one of the bidders is selected, as it depends on who is first noticed by the auctioneer.) So the first proposal is answered with an “accept_proposal”-message, any other proposal with a “reject_proposal”-message. The auction lasts as long as at least one bidder answers the current cfp. When in one round no bidder is willing to accept the current cfp, the auction ends. Finally, the auctioneer will send a “inform”-message to all participants.

There are six predefined goals in this capability that offer the possibility to implement an English auction without much coding effort. These goals are “ea_initiate”, “ea_decide_iteration”, “ea_decide_acceptance” “ea_receiver_interaction”, “ea_decide_participation” and “ea_make_proposal”. The first three goals belong to the initiator side while the others are used on receiver side.

16.3.4.1. Initiator Side

From the perspective of the initiator the protocol can be started via dispatching the “ea_initiate”-goal. The protocol engine will raise a “ea_decide_iteration”-goal for every further negotiation round. This goal gives the initior the chance to decide if a next negotiation round should be performed (e.g. if there are time constraints) and what offer should be made to the participants.

The da_initiate goal needs to be provided with information about the auction (auction_description), the initial call-for-proposal (cfp) and the receivers that shall participate in the negotiation (receivers). Optionally, some local cfp data (cfp_info) as well as a language and ontology for marshalling can also be specified. As result the goal contains the global interaction state (iteraction_state) and the domain output (result).

The ea_initiate goal parameters are further explained in the following table:

Table 16.21. Parameters for ea_initiate

Name	Type	Description
`auction_description`	`AuctionDescription`	A detailed description about the auction.
`cfp`	`Object`	The initial offer of the initiator.
`cfp_info` *	`Object`	Optional locally available further cfp details.
`limit` *	`Comparable`	The secret limit offer of the auctioneer. When specified the protocol engine will test if the last offer is greater or equal than the limit. If this is not the case the good will not be sold to any bidder. When no limit is specified a ea_decide_acceptance goal will be raised

[15]The implementation of the protocol currently only supports a single winner per auction. In future releases, support for multiple winners might be added, e.g., supporting several items of a good to be auctioneered at once.

Name	Type	Description
		and can be used to decide upon acceptance of the final offer.
`ontology` *	`String`	The ontology for marshalling.
`language` *	`String`	The language for marshalling.
`receivers [set]`	`AgentIdentifier`	The agent identifiers of the participants to which the auction will be advertised.
`interaction_state [out]`	`InteractionState`	The interaction state after protocol execution.
`result [out]` *	`Object`	The result of the protocol execution.

**: optional parameter*

The ea_decide_iteration goal is used to determine if the next round should be performed and which offer should be used in that round. For that purpose it provides the history of all made CFPs (history) and the only locally available additional cfp information (cfp_info). If a new round shall be performed the new CFP should be written to the cfp out-parameter.

The ea_decide_iteration parameters are further explained in the following table:

Table 16.22. Parameters for ea_decide_iteration

Name	Type	Description
`cfp [out]`	`Object`	The cfp for the next round.
`cfp_info [inout]` *	`Object`	Optional locally available further cfp details. Changes of the cfp_info can be stored in the parameter again.
`history [set]`	`Object`	The negotiation history. Contains all cfps made so far.

**: optional parameter*

The ea_decide_acceptance goal comes into play when the auction has ended and the auctioneer needs to decide if he, e.g., wants to sell the good for the reached price. There are two alternative ways how one can influence this decision. Firstly, the limit offer can be set directly in the ea_initiate goal. If this is too inflexible and the limit may vary over time it should be left open in the ea_initiate goal. Secondly, it can be determined by the automatically raised ea_decide_acceptance goal. When the goal is not handled by a custom plan and a winner exists it will be accepted. Otherwise the custom plan can decide upon acceptance.

The ea_decide_acceptance parameters are further explained in the following table:

Table 16.23. Parameters for ea_decide_acceptance

Name	Type	Description
`auction_description`	`AuctionDescription`	The cfp for the next round.
`cfp`	`Object`	The current cfp.

Name	Type	Description
cfp_info *	Object	Optional locally available further cfp details.
winner	AgentIdentifier	The auction winner.
accept [out]	Boolean	True, if the current offer should be accepted.
history [set]	Object	The negotiation history. Contains all cfps made so far.

**: optional parameter*

In the code below it is described what needs to be done for executing an English auction. First, the capability, the ea_initiate, the ea_decide_iteration (and optionally the ea_decide_acceptance) goals need to be included as shown in Figure 16.45, “ Including the protocols capability and the goals for the English auction protocol ”. In a plan the ea_initiate goal needs to be created and its mandatory parameters should be set to the desired values. After dispatching the goal (cf. Figure 16.46, “Using the ea_initiate goal”) one can wait for its completion and read out the results of the protocol execution. During the execution the engine will raise a ea_decide_iteration goal in every negotiation round which should be handled by a custom user plan. At the end of the auction a ea_decide_acceptance goal is raised when no limit was specified in the ea_initiate goal.

```
...
<capabilities>
    <capability name="procap" file="jadex.planlib.Protocols"/>
    ...
</capabilities>
...
<goals>
    ...
    <achievegoalref name="ea_initiate">
        <concrete ref="procap.ea_initiate"/>
    </achievegoalref>
    <querygoalref name="ea_decide_iteration">
        <concrete ref="procap.ea_decide_iteration"/>
    </querygoalref>
    <querygoalref name="ea_decide_acceptance">
        <concrete ref="procap.ea_decide_iteration"/>
    </querygoalref>
    ...
</goals>
<plans>
    <plan name="decide_iteration_plan">
        <parameter name="cfp" class="Object">
            <goalmapping ref="ea_decide_iteration.cfp"/>
        </parameter>
        <parameter name="cfp_info" class="Object">
            <goalmapping ref="ea_decide_iteration.cfp_info"/>
        </parameter>
        <parameterset name="history" class="Object">
            <goalmapping ref="ea_decide_iteration.history"/>
        </parameterset>
        <body>new MyDecideIterationPlan()</body>
        <trigger>
            <goal ref="ea_decide_iteration"/>
        </trigger>
    </plan>

    <plan name="decide_acceptance_plan">
        <parameter name="auction_description" class="Object">
            <goalmapping ref="ea_decide_acceptance.auction_description"/>
        </parameter>
        <parameter name="cfp" class="Object">
```

```
            <goalmapping ref="ea_decide_acceptance.cfp"/>
        </parameter>
        <parameter name="cfp_info" class="Object">
            <goalmapping ref="ea_decide_acceptance.cfp_info"/>
        </parameter>
        <parameter name="winner" class="AgentIdentifier">
            <goalmapping ref="ea_decide_acceptance.winner"/>
        </parameter>
        <parameter name="accept" class="Boolean">
            <goalmapping ref="ea_decide_acceptance.accept"/>
        </parameter>
        <parameterset name="history" class="Object">
            <goalmapping ref="ea_decide_acceptance.history"/>
        </parameterset>
        <body>new MyDecideAcceptancePlan()</body>
        <trigger>
            <goal ref="ea_decide_acceptance"/>
        </trigger>
    </plan>
    ...
</plans>
...
```

Figure 16.45. Including the protocols capability and the goals for the English auction protocol

```
public void body()
{
    AgentIdentifier[] recs = ...
    Object cfp = ...
    long starttime = ...
    long roundtimeout = ...
    IGoal eaini = createGoal("ea_initiate");
    eaini.getParameterSet("receivers").addValues(recs);
    eaini.getParameter("cfp").setValue(cfp);
    eaini.getParameter("auction_description").setValue(new AuctionDescription(
        starttime, roundtimeout, "Test auction 1"));
    // Comparable limit = ...
    // eaini.getParameter("limit").setValue(limit);
    dispatchSubgoalAndWait(eaini);
    Object res = request.getParameter("result").getValues();
    ...
}
```

Figure 16.46. Using the ea_initiate goal

Not shown here is the content of the custom decide iteration plan body which has the purpose to generate a new cfp if the next auction round should be performed A convenient way to do this consists in using an automatic offer generator following the interface of `jadex.planlib.IOfferGenerator`. Currently there are two implementation of this interface that allow automatic price generation. For linear price intervals you can use the `LinearPriceCalculator` and for exponential intervals the `ExponentialPriceCalculator`. In addition the custom plan body of the decide acceptance is not further illustrated. It only has to check if the offer should be accepted and write back the boolean value to the corresponding out-parameter (accept). Alternatively, the outcommented lines can be used if a fixed limit is admissable.

16.3.4.2. Participant Side

On the participant side an English auction protocol is triggered when the ea_filter belief value allows this (per default it is turned off). When a inform start-auction message arrives that passes the ea_filter a new

“ea_receiver_interaction” goal is created. This goal is present during the whole lifetime of the interaction and will contain the results and interaction state of the conversation. If the participant side wants to be informed about the outcome of its conversations it can do so by waiting for the completion of this goal (using the goalfinished trigger type). For the implementation of the participant domain functionality two goals need to be supported. If the conversation is handled by the capability it will automatically request domain activities by dispatching subgoals which should be handled by custom user plans.

The “ea_decide_participation” goal is raised when an inform start-auction message has been received. It is used to determine if the participant wants to participate at the auction. If no plan is provided for handling the goal this is regarded as acceptance and the agent will participate at the auction.

Table 16.24. Parameters for ea_decide_participation

Name	Type	Description
`initiator`	`AgentIdentifier`	The agent identifier of the initiator.
`auction_description`	`AuctionDescription`	A detailed description about the auction.
`auction_info [out]` *	`Object`	Optional details about the auction that are only available locally.
`participate [out]` *	`Boolean`	True if the agent wants to participate.

**: optional parameter*

If the agent has decided to particpate and the auction has started it will be requested to react on call-for-proposals of the initiator. For this purpose a “ea_make_proposal” goal will be raised in every negotiation round. A custom user plan should be provided that handles the goal by evaluating the cfp and deciding about its acceptance. Additionally, the agent can also decide to leave the auction in every round via the goal if circumstances have changed.

Table 16.25. Parameters for ea_make_proposal

Name	Type	Description
`cfp`	`Object`	The call-for-proposal represents the current offer of the auctioneer.
`auction_description`	`AuctionDescription`	A detailed description about the auction.
`auction_info [inout]` *	`Object`	Optional details about the auction that are only available locally.
`history [set]` *	`Object`	The history of earlier CFPs.
`accept`	`Boolean`	True if the agent wants to accept the current cfp.
`leave` *	`Boolean`	True if the agent wants to leave the auction.

**: optional parameter*

To use these goals, you must first of all include the Protocols capability in your ADF (if not yet done in order to use other goals of the Protocols capability) and set a reference to the goals as described in Figure 16.47, “ Including the Protocols capability and the participant goals for the English auction ”.

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

<beliefs>
    <beliefref name="ea_filter" class="IFilter">
        <concrete ref="procap.ea_filter"/>
    </beliefref>
    ...
</beliefs>

<goals>
    <performgoalref name="ea_receiver_interaction">
        <concrete ref="procap.ea_receiver_interaction"/>
    </performgoalref>
    <achievegoalref name="ea_decide_participation">
        <concrete ref="procap.ea_decide_participation" />
    </achievegoalref>
    <querygoalref name="ea_make_proposal">
        <concrete ref="procap.ea_make_proposal" />
    </querygoalref>
    ...
</goals>

<plans>
    <plan name="decide_participation_plan">
        <parameter name="initiator" class="AgentIdentifier">
            <goalmapping ref="ea_decide_participation.initiator" />
        </parameter>
        <parameter name="participate" class="Boolean" direction="out">
            <goalmapping ref="ea_decide_participation.participate" />
        </parameter>
        <parameter name="auction_description" class="AuctionDescription">
            <goalmapping ref="ea_decide_participation.auction_description" />
        </parameter>
        <parameter name="auction_info" class="Object">
            <goalmapping ref="ea_decide_participation.auction_info" />
        </parameter>
        <body>new MyDecideParticipationPlan()</body>
        <trigger>
            <goal ref="ea_decide_participation"></goal>
        </trigger>
    </plan>

    <plan name="make_proposal_plan">
        <parameter name="accept" class="Boolean" direction="out" optional="true">
            <goalmapping ref="ea_make_proposal.accept" />
        </parameter>
        <parameter name="leave" class="Boolean" direction="out" optional="true">
            <goalmapping ref="ea_make_proposal.leave" />
        </parameter>
        <parameter name="cfp" class="Object">
            <goalmapping ref="ea_make_proposal.cfp" />
        </parameter>
        <parameter name="auction_description" class="AuctionDescription">
            <goalmapping ref="ea_make_proposal.auction_description" />
        </parameter>
        <parameter name="auction_info" class="Object">
            <goalmapping ref="ea_make_proposal.auction_info" />
        </parameter>
        <parameterset name="history" class="Comparable" optional="true">
            <goalmapping ref="ea_make_proposal.history" />
        </parameterset>
```

```
        <body>new MyMakeProposalPlan()</body>
        <trigger>
            <goal ref="ea_make_proposal"/>
        </trigger>
    </plan>
</plans>
...
<configurations>
    <configuration name="default">
        <beliefs>
            <initialbelief ref="ea_filter">
                <fact>IFilter.ALWAYS</fact>
            </initialbelief>
            ...
        </beliefs>
        ...
    </configuration>
</configurations>
...
```

Figure 16.47. Including the Protocols capability and the participant goals for the English auction

Besides the inclusion of the relevant goals and beliefs the main task of the agent programmer consists in developing the custom domain plan(s). Here in the example code they are named MyDecideParticipationPlan and MyMakeProposalPlan. As the basic responsibilities of the two plans consists in performing domain tasks and writing back the results to the goals no further code cutouts are shown here.

16.3.5. FIPA Dutch Auction Interaction Protocol (DA)

The FIPA Dutch Auction Interaction Protocol (XC00032F[16]) describes an auction with steadily decreased offers. The auction consists of an auctioneer and a set of bidders. The auctioneer chooses a start offer much higher than the supposed market value. Round-by-round the offer is decreased by the auctioneer until there is a proposal from a bidder that accepts the offer. In this case the auction has finished successfully. Otherwise the auctioneer can stop the auction if his limit offer has been reached and further bids may not be acceptable for him.

[16] http://www.fipa.org/specs/fipa00032/XC00032F.html

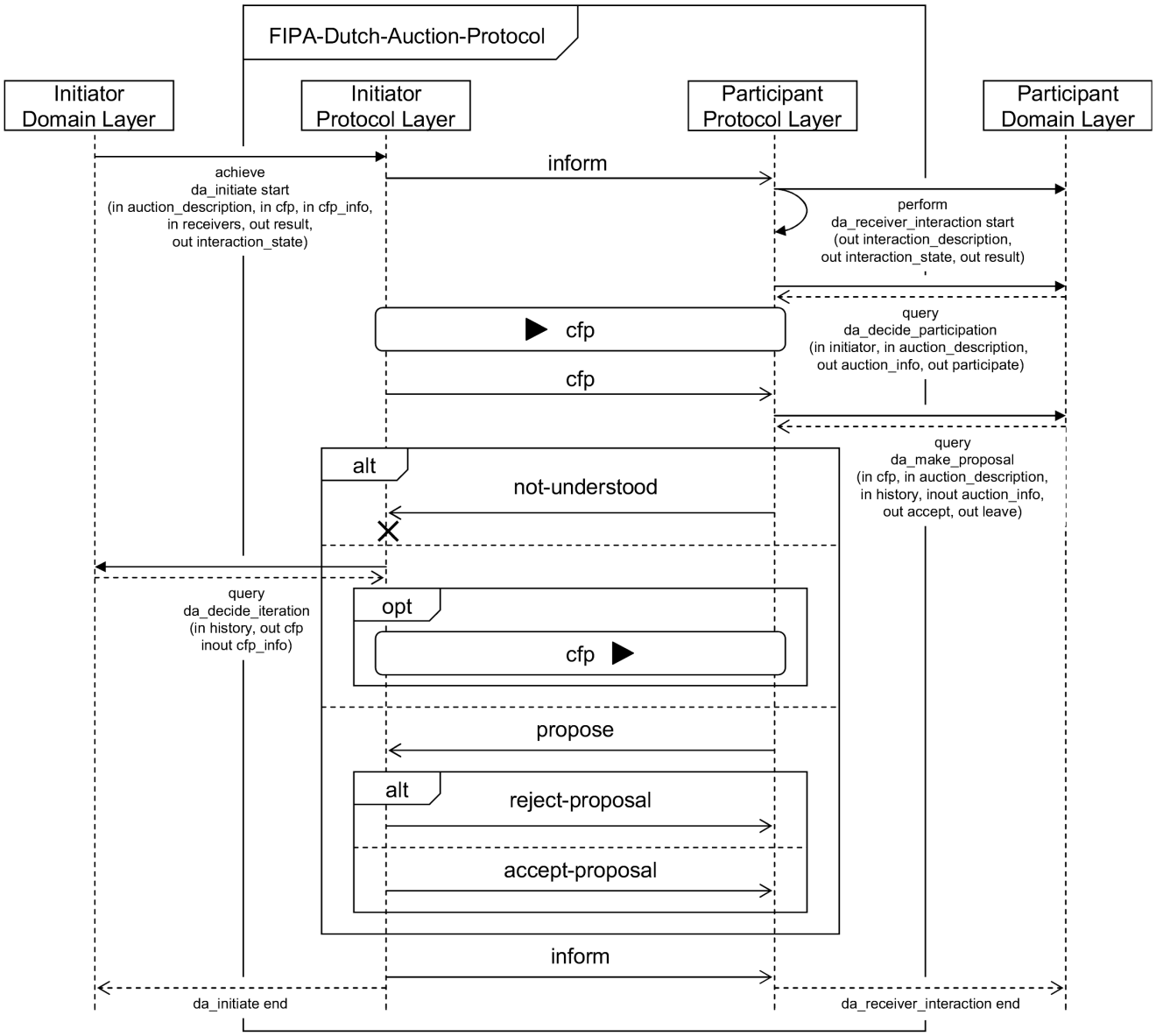

Figure 16.48. The Dutch Auction Protocol

Technically the auction is specified as follows:

The auctioneer starts the auction by sending an “inform-start-of-auction”-message that contains information such as the topic, start time etc. about the planned auction. When the start time of auction has been reached the auctioneer sends a call for proposal (CFP) to all bidders and waits. The bidders can react in three different ways. Firstly, a bidder can send a “not-understood”-message in order to signalize that it is not able to understand the CFP. The auctioneer will subsequently exclude this bidder from the auction. Secondly, a bidder considers the CFP as inacceptable. In this case it will do nothing and wait for the next negotiation round in which a new CFP will be available. Thirdly, a bidder can send a proposal, i.e. it can accept the offer contained in the CFP from the auctioneer. If there is more than one proposal, only the first one is accepted.[17] (Even if they arrive at the same time, they will technically not be processed concurrently. This is similar to a real Dutch auction, because if two bidders raise their hands at the same time it is also nondeterministic which one of the bidders is selected, as it depends on who is first noticed by the auctioneer.) So the first proposal is answered with an “accept_proposal”-message, any other proposal with a “reject_proposal”-message. At the end of the auction

[17]The implementation of the protocol currently only supports a single winner per auction. In future releases, support for multiple winners might be added, e.g., supporting several items of a good to be auctioneered at once.

the auctioneer will send a “inform”-message to all participants.

There are five predefined goals in this capability that offer the possibility to implement a Dutch auction with only few lines of code. These goals are “da_initiate”, “da_decide_iteration”, “da_receiver_interaction”, “da_decide_participation” and “da_make_proposal”. The first two goals belong to the initiator side while the others are used on receiver side.

16.3.5.1. Initiator Side

From the perspective of the initiator the protocol can be started via dispatching the “da_initiate”-goal. If the auctioneered good is not bought immediately, the protocol engine will raise a “da_decide_iteration”-goal for every further negotiation round. This goal gives the inititor the chance to decide if a next negotiation round should be performed and what offer should be made to the participants.

The da_initiate goal needs to be provided with information about the auction (auction_description), the initial call-for-proposal (cfp) and the receivers that shall participate in the negotiation (receivers). Optionally, some local cfp data (cfp_info) as well as a language and ontology for marshalling can also be specified. As result the goal contains the global interaction state (interaction_state) and the domain output (result).

The da_initiate goal parameters are further explained in the following table:

Table 16.26. Parameters for da_initiate

Name	Type	Description
`auction_description`	`AuctionDescription`	A detailed description about the auction.
`cfp`	`Object`	The initial offer of the initiator.
`cfp_info` *	`Object`	Optional locally available further cfp details.
`ontology` *	`String`	The ontology for marshalling.
`language` *	`String`	The language for marshalling.
`receivers [set]`	`AgentIdentifier`	The agent identifiers of the participants to which the auction will be advertised.
`interaction_state [out]`	`InteractionState`	The interaction state after protocol execution.
`result [out]` *	`Object`	The result of the protocol execution.

**: optional parameter*

The da_decide_iteration is used to determine if the next round should be performed and which offer should be used in that round. For that purpose it provides the history of all made CFPs (history) and the only locally available additional cfp information (cfp_info). If a new round shall be performed the new CFP should be written to the cfp out-parameter.

The da_decide_iteration parameters are further explained in the following table:

Table 16.27. Parameters for da_decide_iteration

Name	Type	Description
`cfp [out]`	`Object`	The cfp for the next round.
`cfp_info [inout]` *	`Object`	Optional locally available further cfp details. Changes of the cfp_info can be stored in the parameter again.
`history [set]`	`Object`	The negotiation history. Contains all cfps made so far.

**: optional parameter*

In the code below it is described what needs to be done for executing a Dutch auction. First, the capability, the da_initiate and the da_decide_iteration goals need to be included as shown in Figure 16.49, “ Including the protocols capability and the goals for the Dutch auction protocol ”. In a plan the da_initiate goal needs to be created and its mandatory parameters should be set to the desired values. After dispatching the goal (cf. Figure 16.50, “Using the da_initiate goal”) one can wait for its completion and read out the results of the protocol execution. During the execution the engine will raise a da_decide_iteration goal in every negotiation round which should be handled by a custom user plan.

```
...
<capabilities>
    <capability name="procap" file="jadex.planlib.Protocols"/>
    ...
</capabilities>
...
<goals>
    ...
    <achievegoalref name="da_initiate">
        <concrete ref="procap.da_initiate"/>
    </achievegoalref>
    <querygoalref name="da_decide_iteration">
        <concrete ref="procap.da_decide_iteration"/>
    </querygoalref>
    ...
</goals>
<plans>
    <plan name="decide_iteration_plan">
        <parameter name="cfp" class="Object">
            <goalmapping ref="da_decide_iteration.cfp"/>
        </parameter>
        <parameter name="cfp_info" class="Object">
            <goalmapping ref="da_decide_iteration.cfp_info"/>
        </parameter>
        <parameterset name="history" class="Object">
            <goalmapping ref="da_decide_iteration.history"/>
        </parameterset>
        <body>new MyDecideIterationPlan()</body>
        <trigger>
            <goal ref="da_decide_iteration"/>
        </trigger>
    </plan>
</plans>
...
```

Figure 16.49. Including the protocols capability and the goals for the Dutch auction protocol

```
public void body()
{
    AgentIdentifier[] recs = ...
```

```
    Object cfp = ...
    IGoal daini = createGoal("da_initiate");
    daini.getParameterSet("receivers").addValues(recs);
    daini.getParameter("cfp").setValue(cfp);
    daini.getParameter("auction_description").setValue(new AuctionDescription(
        System.currentTimeMillis()+1000, roundtimeout, "Test auction 1"));
    dispatchSubgoalAndWait(daini);
    Object res = request.getParameter("result").getValues();
    ...
}
```

Figure 16.50. Using the da_initiate goal

Not shown here is the content of the custom decide iteration plan body which has the purpose to generate a new cfp if the next auction round should be performed A convenient way to do this consists in using an automatic offer generator following the interface of `jadex.planlib.IOfferGenerator`. Currently there are two implementation of this interface that allow automatic price generation. For linear price intervals you can use the `LinearPriceCalculator` and for exponential intervals the `ExponentialPriceCalculator`.

16.3.5.2. Participant Side

On the participant side a Dutch auction protocol is triggered when the da_filter belief value allows this (per default it is turned off). When a inform start-auction message arrives that passes the da_filter a new “da_receiver_interaction” goal is created. This goal is present during the whole lifetime of the interaction and will contain the results and interaction state of the conversation. If the participant side wants to be informed about the outcome of its conversations it can do so by waiting for the completion of this goal (using the goalfinished trigger type). For the implementation of the participant domain functionality two goals need to be supported. If the conversation is handled by the capability it will automatically request domain activities by dispatching subgoals which should be handled by custom user plans.

The “da_decide_participation”-goal is raised when an inform start-auction message has been received. It is used to determine if the participant wants to participate at the auction. If no plan is provided for handling the goal this is valued as acceptance and the agent will participate at the auction.

Table 16.28. Parameters for da_decide_participation

Name	Type	Description
`initiator`	`AgentIdentifier`	The agent identifier of the initiator.
`auction_description`	`AuctionDescription`	A detailed description about the auction.
`auction_info [out]` *	`Object`	Optional details about the auction that are only available locally.
`participate [out]` *	`Boolean`	True if the agent wants to participate.

**: optional parameter*

If the agent has decided to particpate and the auction has started it will be requested to react on call-for-proposals of the initiator. For this purpose a “da_make_proposal” goal will be raised in every negotiation round. A custom user plan should be provided that handles the goal by evaluating the cfp and deciding about its acceptance. Additionally, the agent can also decide to leave the auction in every round via the goal if circum-

stances have changed.

Table 16.29. Parameters for da_make_proposal

Name	Type	Description
`cfp`	`Object`	The call-for-proposal represnts the current offer of the auctioneer.
`auction_description`	`AuctionDescription`	A detailed description about the auction.
`auction_info [inout]` *	`Object`	Optional details about the auction that are only available locally.
`history [set]` *	`Object`	The history of earlier CFPs.
`accept`	`Boolean`	True if the agent wants to accept the current cfp.
`leave` *	`Boolean`	True if the agent wants to leave the auction.

**: optional parameter*

To use these goals, you must first of all include the Protocols capability in your ADF (if not yet done in order to use other goals of the Protocols capability) and set a reference to the goals as described in Figure 16.51, “ Including the Protocols capability and the participant goals for the Dutch auction ”.

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

<beliefs>
    <beliefref name="da_filter" class="IFilter">
        <concrete ref="procap.da_filter"/>
    </beliefref>
    ...
</beliefs>

<goals>
    <performgoalref name="da_receiver_interaction">
        <concrete ref="procap.da_receiver_interaction"/>
    </performgoalref>
    <achievegoalref name="da_decide_participation">
        <concrete ref="procap.da_decide_participation" />
    </achievegoalref>
    <querygoalref name="da_make_proposal">
        <concrete ref="procap.da_make_proposal" />
    </querygoalref>
    ...
</goals>

<plans>
    <plan name="decide_participation_plan">
        <parameter name="participate" class="Boolean" direction="out">
            <goalmapping ref="da_decide_participation.participate" />
        </parameter>
        <parameter name="auction_description" class="AuctionDescription">
            <goalmapping ref="da_decide_participation.auction_description" />
        </parameter>
```

```
        <parameter name="auction_info" class="Object">
            <goalmapping ref="da_decide_participation.auction_info" />
        </parameter>
        <body>new MyDecideParticipationPlan()</body>
        <trigger>
            <goal ref="da_decide_participation"></goal>
        </trigger>
    </plan>

    <plan name="make_proposal_plan">
        <parameter name="accept" class="Boolean" direction="out" optional="true">
            <goalmapping ref="da_make_proposal.accept" />
        </parameter>
        <parameter name="leave" class="Boolean" direction="out" optional="true">
            <goalmapping ref="da_make_proposal.leave" />
        </parameter>
        <parameter name="cfp" class="Object">
            <goalmapping ref="da_make_proposal.cfp" />
        </parameter>
        <parameter name="auction_description" class="AuctionDescription">
            <goalmapping ref="da_make_proposal.auction_description" />
        </parameter>
        <parameter name="auction_info" class="Object">
            <goalmapping ref="da_make_proposal.auction_info" />
        </parameter>
        <parameterset name="history" class="Comparable" optional="true">
            <goalmapping ref="da_make_proposal.history" />
        </parameterset>
        <body>new MyMakeProposalPlan()</body>
        <trigger>
            <goal ref="da_make_proposal"/>
        </trigger>
    </plan>
</plans>
...
<configurations>
    <configuration name="default">
        <beliefs>
            <initialbelief ref="da_filter">
                <fact>IFilter.ALWAYS</fact>
            </initialbelief>
            ...
        </beliefs>
        ...
    </configuration>
</configurations>
...
```

Figure 16.51. Including the Protocols capability and the participant goals for the Dutch auction

Besides the inclusion of the relevant goals and beliefs the main task of the agent programmer consists in developing the custom domain plan(s). Here in the example code they are named MyDecideParticipationPlan and MyMakeProposalPlan. As the basic responsibilities of the two plans consists in performing domain tasks and writing back the results to the goals no further code cutouts are shown here.

16.3.6. Abnormal Termination of Protocols

As described in the specification of the FIPA protocols such as FIPA Request[18] and FIPA Contract Net[19], at any point in time, one side of an ongoing interaction, i.e. the initiator or one of the participants, may decide to

[18] http://www.fipa.org/specs/fipa00026/SC00026H.html
[19] http://www.fipa.org/specs/fipa00029/SC00029H.html

leave the interaction. When the initiator decides to leave the interation this will lead to the whole interaction to stop, while when a participant leaves, the initiator will usually continue to interact with the other participants. To indicate that it wishes to leave an interaction, a participant may send a not-understood message, while an initiator can decide to stop the whole protocol by sending a cancel message to all participants.

In the Protocols capability, all interactions are represented by an interaction goal. Therefore, when an agent whishes to leave an interaction it can naturally do so by dropping the corresponding goal. Generic mechanisms available in all implemented protocols will take care of the necessary coordination between initiator and participant(s). These mechanism are described in more detail in the following sections Section 16.3.6.1, “Leaving an Interaction (Participant Side)” and Section 16.3.6.2, “Cancelling an Interaction (Initiator Side)”.

16.3.6.1. Leaving an Interaction (Participant Side)

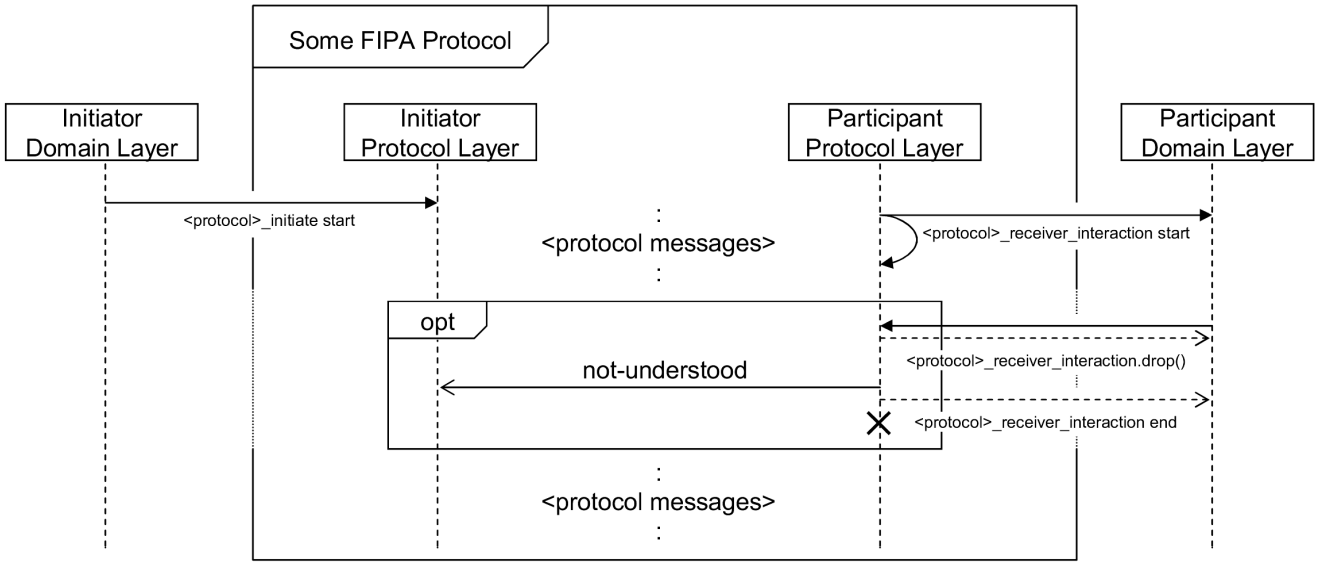

Figure 16.52. The Participant Termination Subprotocol

The process of leaving an interaction is depicted in Figure 16.52, “The Participant Termination Subprotocol”. At any time after the interaction has started, the domain layer of the participant may decide to drop the interaction goal (e.g. a "cnp_receiver_interation" goal). The protocol will be aborted at the participant side and as an indication, a not-understood message is sent to the initiator, which will exclude the participant from the rest of the interaction. The participant domain layer can check the state after the abortion of the interaction, by waiting for the goalfinished event of the "<protocol>_receiver_interaction" goal and inspecting the "interaction_state" parameter.

16.3.6.2. Cancelling an Interaction (Initiator Side)

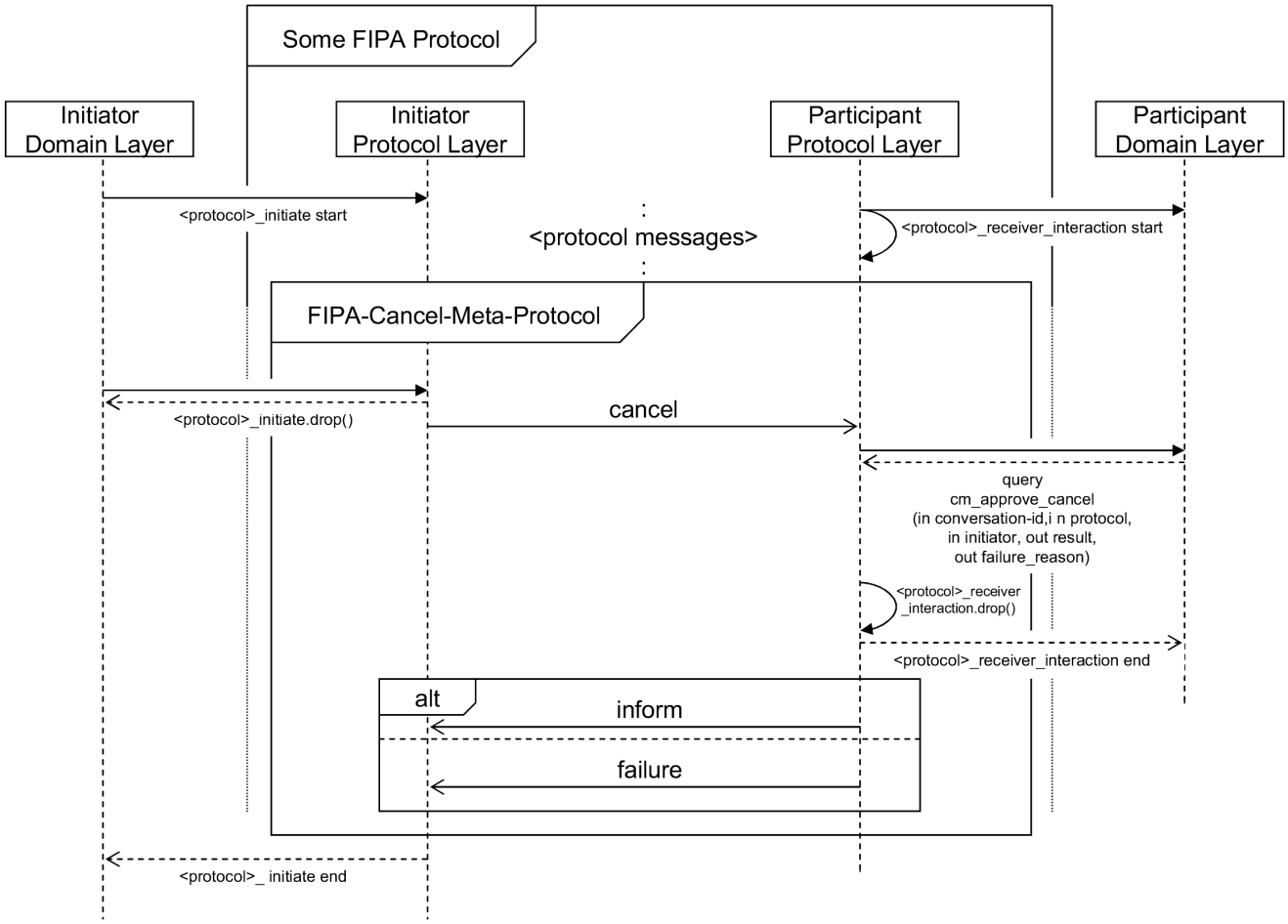

Figure 16.53. The FIPA Cancel Meta Protocol

In Figure 16.53, “The FIPA Cancel Meta Protocol” it is shown, how a protocol can be cancelled. At any time after the interaction has started, the domain layer of the initiator may decide to drop the interaction goal (e.g. a "cnp_initiate" goal). The protocol will be aborted at the initiator side and a cancel message is sent to all participants. According to the FIPA Cancel Meta Protocol, which is specified as part of several other FIPA protocol specifications[20], the participant has two choices. It may either answer with an inform message thereby indicating that it has no problem with the cancellation of the protocol. Otherwise, it might send a failure message containing some description of its objections (e.g. if the participant already performed some costly task it may demand a reimbursement). This decision is delegated to the participant domain layer in form of the query "cm_approve_cancel" goal. It contains the following input and output parameters:

Table 16.30. Parameters for cm_approve_cancel

Name	Type	Description
`conversation-id`	`String`	The id of the interaction to be cancelled. Can e.g. be used to find the corresponding goal in the goal base.
`protocol`	`String`	The name of the cancelled protocol (e.g. "fipa-request").
`initiator`	`AgentIdentifier`	The agent identifier of the initiator of the

[20] See e.g. http://www.fipa.org/specs/fipa00026/SC00026H.html

Name	Type	Description
		interaction.
`result [out]`	`Boolean`	Should be set to true if the interaction can be safely cancelled, and to false if there are some problems related to cancelling the interaction.
`failure_reason [out]` *	`Object`	An object describing the participants objections to the cancellation of the protocol.

**: optional parameter*

The "conversation-id", "protocol", and "initiator" parameters are used to identify the interaction to be cancelled. The boolean "result" parameter is used to store the decision and another parameter is available for the optional "failure_reason". A default plan exists in the protocols capability that will automatically acknowledge all "cm_approve_cancel" goals. Therefore custom plans for this goal are only necessary, when in some cases a failure message should be sent. Regardless of the goal result, the interaction will be terminated at the participant side, when the "cm_approve_cancel" goal returns. The handling of possibly required compensations is out of the scope of the cancel meta protocol and would have to be performed in a separate interaction between participant and initiatior.

The decision and potentially also the failure reason are communicated back to the sender, after the interaction has been terminated at the participant side. After all participants have answered to the cancel message or after the timout has passed, the protocol also ends at the initiator side. The participant and the initiator domain layer can check the state after the abortion of the interaction, by waiting for the goalfinished event of the "<protocol>_receiver_interaction" goal resp. the "<protocol>_initiate" goal and inspecting the "interaction_state" parameter, which will also contain references to the supplied failure reasons of the participants (if any). This information should be useful to initiate compensation actions, if necessary.

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 0.95 and 0.96

Protocols Capability. The protocols capability provides a goal-oriented way to deal with various interaction protocols such as FIPA-Request, Contract-Net, Dutch- or English-Auction. Through predefines goals for initiator and responder sides of each protocol, the message-based communication becomes completely transparent to the developer. The protocols capability is described in the new chapter Chapter 16, *Using Predefined Capabilities* of the user guide.

Agent Configurations. As an extension of the initial states known from previous Jadex releases, agents now may have so called configurations including initial elements as well as end elements. End elements (e.g. end goals or end plans) are executed when the agent gets killed. They provide an easy way to perform complex agent-oriented cleanup operations. Agent configurations are described in chapter Chapter 13, *Configurations* of the user guide.

Agent Listeners. The new release adds the possibility to add listeners to agents and BDI-elements (e.g. beliefs or goals). These listeners get called when certain events happen on an element (e.g. a fact was added or a goal finished). This callback style of programming is especially useful for developing GUIs. For more details see section Section 15.2, “Agent Listeners”.

Agent Arguments. To simplify the specification of agent arguments, now all exported beliefs of an agent are considered as arguments, which can be set using the name of the belief. Therefore, e.g., when a new agent is created, a map of arguments (name-value-pairs) can be provided.

Detail Improvements. The new strong export (which is now the default) allows that instances of referenced elements are created outside a capability, even when the creation was triggered inside the capability (see Section 5.3, “Elements of a Capability”). Recur and recurdelay can now be used in all goal types (previously only in maintain goals). Custom match expressions can now be specified for message event types as well as plan triggers. These allow to define complex expressions, e.g., including more than one parameter. The nuggets codec has been added as a fast replacement for the slow java xml codec. With regard to programming plans, it should be noted that it is now possible to dispatch subgoals, wait for messages etc. also in the passed/failed/aborted methods of plans.

New and Improved Tools. With the Test Center and the DF Browser, two new tools have been added. The Test Center allows to execute agent-based unit-like tests. The DF Browser allows to inspect the current registrations In addition, Javadoc/Jadexdoc generation is now provided in a separate perspective, which has been completely rewritten. Improvements of existing tools include the ability to specify arguments und to start agents by double-click in the Starter perspective, a new goal/plan-hierarchy view in the Introspector as well as updated icons for Introspector and Tracer views. Moreover, the JCC startup time has been reduced by loading plugins on demand.

New Example "Book Trading". The booktrading example contains buyer and seller agents, which negotiate prices for goods such as books. The example thereby specifically shows how to use the new protocols capability.

A.2. Incompatibilities to Release 0.941

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. The new schema definition carries version number 0.96, therefore you have to update your XML headers, if you want your IDE (or other validation tools) to recognize the new or changed tags (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-0.96.xsd"
       name=".."
       package="..">
```

Figure A.1. Header of an agent definition file

Initial States. The initial states have been renamed to configurations as they now also support end elements in addition to initial elements (see Figure A.2, “Example configurations section of an agent”).

```
<configurations>
        <configuration name="default">
                <plans>
                        <initialplan ref="initial_plan"/>
                </plans>
        </configuration>
        <configuration name="abc">
                <plans>
                        <initialplan ref="initial_plan"/>
                        <endplan ref="initial_plan"/>
                </plans>
        </configuration>
</configurations>
```

Figure A.2. Example configurations section of an agent

A.2.2. Capability Changes

AMS Capability

The ams_create_agent goal now has a parameter of type Map for holding the agent arguments (instead of the previous parameter set), due to arguments now being specified as name-value pairs.

DF Capability

The FIPA-Request protocol support (goal_request) has been moved to the new protocols capability (see goal rp_initiate in Section 16.3, “The Interaction Protocols Capability ”). Moreover the result parameter of the df_search goal has been changed to a parameter set (containing arbitrary AgentDescription objects).

A.2.3. API Changes

`MobilePlan.passed()/failed()/aborted()`

These methods now allow e.g. subgoals to be dispatched and therefore may be called several time sfor subsequent events. They introduce a new event parameter indicating which event occurred. The old methods without parameter have been changed to final, such that they can no longer be (accidentally) overwritten.

`Plan.xxxAndWait()`

Most of the xxxAndWait() methods of the standard plan class have been changed to return void instead of an event, because the programmer should not have to deal with explicit event when using elements such as goals or conditions. Moreover, these methods now always throw exception, when the expected event does not oocur (e.g. TimeoutException or GoalFailureExecption when a subgoal does not succeed).

`IExpression.execute(Tuple[] params)`

The signature of IExpression for executing queries with more than one parameter value has changed to execute(String[] names, Object[] values) as it is more natural than using a Tuple[].

`ICandidateInfo.getPlan()/getEvent()`

Removed the `caller` parameter, of `ICandidateInfo.getPlan()` and `CandidateInfo.getEvent()` as it is no longer necessary.

Appendix B. 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 platform JADE [1] and for a Standalone platform.

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.

B.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`).

B.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. To start the "niotcp" (TCP/IP) based transport layer at port 9876 use the setting: `-transport jadex.adapter.standalone.transport.niotcpmtp.NIOTCPTransport:9876` (Since the new I/O (NIO) is only provided in Java 1.4 or later, we also provide another Java 1.3 compatible transport, which can be

[1] http://jade.tilab.com/

selected as follows: `-transport jadex.adapter.standalone.transport.tcpmtp.TCPTransport: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.

B.1.2. Starting Agents from the Command Line

To start an agent from the command line, the name and properties of the agent are given after the platform start command. The syntax for each agent is given below. This way an arbitrary number of agents can be started when launching the platform.

name:model[(configuration[, arg1name=arg1, ..., argNname=argN])]

name: The name of the agent instance, which can be freely chosen, as long as no two agents would get the same name.

model: The file name or logical name of the agent model (e.g. `jadex.examples.helloworld.HelloWorld` or `jadex/examples/helloworld/HelloWorld.agent.xml`).

configuration: The name of the configuration to use as defined in the agent model. If the configuration is omitted, the default configuration of the agent is used.

argNname=argN: Agent arguments can be supplied following the configuration. Each argument is given as a name-value pair separated by an equals sign. The value part can be an arbitrary Java expression that is parsed before being given to the agent (e.g. `new java.awt.Color(255,0,0)`).

The following example starts the standalone platform and instantly starts a hello world agent using a custom argument. Note that the quotes for the string argument have to be escaped (using `\"`), because the agent specification as a whole is enclosed in quotes.

java jadex.adapter.standalone.Platform "hello:jadex.examples.helloworld.HelloWorld(default, msg=\"Hi!\")"

B.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. These jars are found automatically if they reside in the same directory as the `jadex_jadeadapter.jar`. The same is true for the other required Jadex jars (see Section 1.1, “Requirements and Installation”), i.e. you should copy all jars (except `jadex_standalone.jar`) from the Jadex main distribution into the jadeadapter's lib directory.

B.2.1. Starting the JADE Adapter

The JADE adapter contains a helper class, which can be used to start the JADE platform and open the Jadex Control Center. To start the platform, use the command:

java jadex.adapter.jade.tools.Starter [JADE options] [JADE/Jadex agents]

Alternatively you can also start the platform from jar via:

java -jar jadex_jadeadapter.jar [JADE options] [JADE/Jadex agents]

JADE options: For the JADE options, all flags and configuration options of JADE can be used. E.g. -container, -host, -port, etc.

JADE/Jadex agents: Agents can be started from the command line using the normal JADE syntax "agent-name:classname(arg1 arg2 ...)". If you want to start a Jadex agent from the command line, the classname needs to be `jadex.adapter.jade.JadeAgentAdapter`, the first argument to the agent must be the name of the agent model (the XML file) and the second argument is the name of the configuration to be used.

The JCC is also a normal Jadex agent, which can be started in JADE as any other agent. Therefore, JADE and the JCC can also be started as follows:

java jade.Boot [JADE options] jcc:jadex.adapter.jade.JadeAgentAdapter(jadex.tools.jcc.JCC default) [other JADE/Jadex agents]
JADE does its own processing of argument values before handing them to Jadex. Therefore, one needs an extra layer of escaping characters, which would otherwise get mangled by JADE. The helloworld example described before therefore needs to be started like this (note the three backslashes before the quotes of the string argument):

java jade.Boot jcc:jadex.adapter.jade.JadeAgentAdapter(jadex.examples.helloworld.HelloWorld default msg=\\\"Hi!\\\")

The system property `-Dconf` can be used in all cases desribed above, to configure the Jadex system (e.g. java -Dconf=myjadex.properties [...]). 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).

B.2.2. Using JADE Ontologies and Content Languages

This section assumes that you know how to use ontologies and content languages in JADE. More information about this is provided in the document "Creating and using applications-specific ontologies" available in the JADE distribution or from the JADE homepage at http://jade.tilab.com/doc/index.html. In summary, the JADE content management requires that the message content object must be instance of a `jade.content.ContentElement`. The concrete class of the object should belong to some ontology, which can be generated using the Beanynizer tool of Jadex (described in the tool guide) or the JADE ontology beangenerator available from the JADE homepage.

Making JADE content management available to a Jadex agent is achieved by a small helper class, that implements a JADE specific Jadex content codec. An instance of this class `JadeContentCodec` (from package `jadex.adapter.jade`) has to be created for each pair of language and ontology that you want to support. These instances are added to the properties section of an agent:

```
<properties>
    <property name="contentcodec.fipa-management-sl0">
        new JadeContentCodec(new SLCodec(0), FIPAManagementOntology.getInstance())
    </property>
    <property name="contentcodec.jade-management-sl0">
        new JadeContentCodec(new SLCodec(0), JADEManagementOntology.getInstance())
    </property>
```

```
</properties>
```

B.2.3. 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 (bug 0000107[2]). 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>
```

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`).

B.2.4. 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.

[2] http://avalon.cselt.it/mantis/bug_view_page.php?bug_id=0000107

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>
```

Appendix C. Add-Ons

Several extensions are available for Jadex from the Jadex add-ons page[1]. These are shortly presented in the following.

C.1. Expression Compiler

In the normal version Jadex utilizes a Java expression interpreter. The expression compiler is based on Janino[2] 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.

C.2. Webbridge

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.

C.3. 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.

C.4. Diet adapter (experimental)

Besides the JADE and Standalone adapter an experimental adapter for the Diet platform[3] 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.

[1] http://vsis-www.informatik.uni-hamburg.de/projects/jadex/addons.php
[2] http://www.janino.net/
[3] http://diet-agents.sourceforge.net/

Appendix D. FAQ+HOWTO

D.1.

When I start the Jadex platform together with the JADE adapter I get a strange error (e.g. `ClassCastException`) and the JADE RMA does not show up. What can be the reason for that?

Currently, you should set-up cleanly separated environments for running Jadex with the Standalone resp. the JADE platform. For setting up a clean environment you should consider the following aspects:

- Both versions should not be started from the same directory, as both use the `jadex.properties` and overwrite values in it (or should explicitly use different propoerties). As the settings are incompatible this may cause troubles.
- Secondly, you should also separate classpath settings of both Jadex versions. It causes troubles if more than adapter jar is included as the wrong service classes might be used (AMS/DF etc).

D.2.

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

D.3.

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.

D.4.

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.

D.5.

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[1]) are directly reflected whenever the model is laoded. The reason for this is that inline plan bodies can be compiled on demand at runtime.

D.6.

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.

D.7.

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

Since version 0.96 Jadex supports not only initial states but also end states. Whenever an agent is terminated its execution will not be immediately stopped. Instead the agent changes its state to "terminating", aborts all running goals and plans and activates elements declared in the end state. For details please have a look at Chapter 13, *Configurations*. If you want to be notified when an agent dies you can use an agent listener (cf. Section 15.2, “Agent Listeners”).

D.8.

How can I parametrize an agent and set parameter values before starting?

Since Jadex 0.96 the way agent arguments are specified has changed. All agent beliefs that are "exported" are automatically considered as agent parameters. The JCC gui automatically creates input fields for these exported parameters. Programmatically, the arguments can be directly referenced via their associated beliefs.

D.9.

Is there some preferred persistence mechanism for beliefs?

[1] http://vsis-www.informatik.uni-hamburg.de/projects/jadex/addons.php

In the current version Jadex does not provide a ready-to-use persistence mechanism for the beliefs of an agent. We have successfully used normal object-relational mapping frameworks such as Hibernate[2] in combination with Jadex. Nonetheless, the task of persisting data cannot be fully automated and needs to be done in plans. This topic should be an issue of further research and improvement.

D.10.

Can capabilities be used for group communication, i.e. are they some kind of tuple space, where one agent puts in data and other can read it?

No, this seems to be a common misunderstanding of the concept. A capability is comparabale to a module. Each agent that includes a capability get a separate instance of that module. For details have a look at Chapter 5, *Capabilities*.

[2] http://www.hibernate.org

Appendix E. Legal Notice

Licence. Jadex is distributed under the GNU Lesser General Public License[1] (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.

E.1. Third-Party Software

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

- JiBX
- GraphLayout
- JavaHelp

JiBX. The JiBX[2] XML binding framework is used to load agent models from an agent definition file (ADF). It is distributed under following licence:

[1] http://www.gnu.org/copyleft/lesser.html
[2] http://jibx.sourceforge.net/

GraphLayout. The GraphLayout[3] component is used in the Tracer Tool and is licenced under following terms:

TouchGraph LLC. Apache-Style Software License

Copyright (c) 2001-2002 Alexander Shapiro. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by TouchGraph LLC (http://www.touchgraph.com/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear.

4. The names "TouchGraph" or "TouchGraph LLC" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact alex@touchgraph.com

5. Products derived from this software may not be called "TouchGraph", nor may "TouchGraph" appear in their name, without prior written permission of alex@touchgraph.com.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL TOUCHGRAPH OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

JavaHelp System. The JavaHelp[4] system is used in all runtime tools.

[3] http://touchgraph.sourceforge.net/

[4] http://java.sun.com/products/javahelp/index.jsp

Sun Microsystems, Inc.
Binary Code License Agreement

READ THE TERMS OF THIS AGREEMENT AND ANY PROVIDED SUPPLEMENTAL LICENSE TERMS (COLLECTIVELY "AGREEMENT") CAREFULLY BEFORE OPENING THE SOFTWARE MEDIA PACKAGE. BY OPENING THE SOFTWARE MEDIA PACKAGE, YOU AGREE TO THE TERMS OF THIS AGREEMENT. IF YOU ARE ACCESSING THE SOFTWARE ELECTRONICALLY, INDICATE YOUR ACCEPTANCE OF THESE TERMS BY 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 REFUND 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 PARTICULAR PURPOSE OR NON-INFRINGEMENT ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT THESE DISCLAIMERS 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 OUT OF OR RELATED TO THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Heidelberg New York. 2005.

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

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

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

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

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