Einführung in das Jadex-System

Stefan Tittel
Universität Dortmund

Projektgruppenseminar: Wissen in Multiagentensystemen,
März 2006

Inhaltsverzeichnis

1 Einführung

Jadex ist eine agentenorientierte Schlussfolgerungs-Engine, welche auf JAVA und XML basiert und zur Realisierung rationaler Agenten dient. Unter einem Agenten wird dabei ein autonomes, proaktives, reaktives, soziales sowie lern- und anpassungsfähiges Programm verstanden.

1.1 Ausführung

Jadex selbst stellt keine über die Schlussfolgerungs-Engine hinausgehenden Dienste bereit. Um einen Agenten ausführen und mit ihm kommunizieren zu können, bedarf es daher einer Middleware-Plattform. Gegenwärtig existieren zwei Implementierungen einer solchen Middleware-Plattform, eine dritte (für die DIET-Agenten-Plattform[1]) ist in Entwicklung.

- Der *Jadex Standalone Adapter* liegt Jadex bei und läuft ohne weitere Abhängigkeiten. Er verfügt über eine eigene graphische Benutzerschnittstelle, die sich in vier Bereiche gliedert:

  1. *Starter* – dient dazu, Agenten zu laden und auszuführen
  2. *Inspektor* – bietet die Möglichkeit, die Menge der beliefs, plans und goals zu betrachten und beinhaltet den Debugger
  3. *Conversation Center* – dient der Interaktion zwischen Agenten und Nutzer in Form von Nachrichten
  4. *Tracer* – ermöglicht die Ablaufverfolgung von Agenten

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

- Der *Jade Adaper* liegt Jadex nicht bei und muss separat heruntergeladen werden. Er dient primär zur Einbindung von Jadex in das JAVA Agent Development Framework (JADE)[2].

1.2 Agentenmodell

Das Agentenmodell von Jadex basiert auf dem BDI-Modell, dem wohl bekanntesten Vertreter deliberativer Agentenarchitekturen.

1.2.1 BDI-Modell

Dem BDI-Modell zufolge verfügt jeder Agent über *beliefs*, *desires* und *intentions*.

1. Beliefs spiegeln dabei das Weltwissen – den *informational state* – des Agenten wider.
2. Desires geben die Hauptziele – den *motivational state* – des Agenten an.
3. Intentions sind die konkreten Absichten und Pläne eines Agenten und beschreiben den *deliberative state*.

1.2.2 Beliefbase, Beliefs, Beliefsets und Facts

Der informational state eines Agenten wird in Jadex durch genau eine beliefbase modelliert. Eine beliefbase beinhaltet *beliefs*[3] und *beliefsets*. Ein belief besteht genau aus einem *fact*, ein beliefset aus einer Menge von facts. Ein fact enthält tatsächliche Information und kann durch eine beliebige JAVA-Klasse implementiert werden. Der direkte Zugriff erfolgt entweder durch Identifikationsstrings (z. B. „gib mir den belief mit dem Namen ‚foo' aus der beliefbase und gib mir den fact dieses belief") oder durch Anfragen an die beliefbase in einer OQL-ähnlichen Sprache.

Eine logikbasierte Darstellung von Wissen und daraus erwachsende Möglichkeiten zur Inferenz bietet Jadex von sich aus nicht, wenngleich natürlich eine Implementierung in Form geeigneter JAVA-Klassen möglich ist. Ferner können Änderungen im Bereich der beliefs als Ausführungs- und Triggerbedingungen für andere Jadex-Komponenten dienen.

1.2.3 Goals

Goals realisieren in Jadex den motivational state eines Agenten. Ein goal beschreibt dabei einen seiner konkreten momentanen Wünsche. Für jedes goal wählt der Agent früher oder später geeignete Aktionen aus, solange bis das goal entweder erreicht wird, als unerreichbar gilt oder nicht länger aktuell ist. Ein goal kann sich in drei Zuständen befinden.

[2] `http://jade.tilab.com/`
[3] Hier sind beliefs im Kontext von Jadex, nicht beliefs im Sinne des BDI-Modells gemeint.

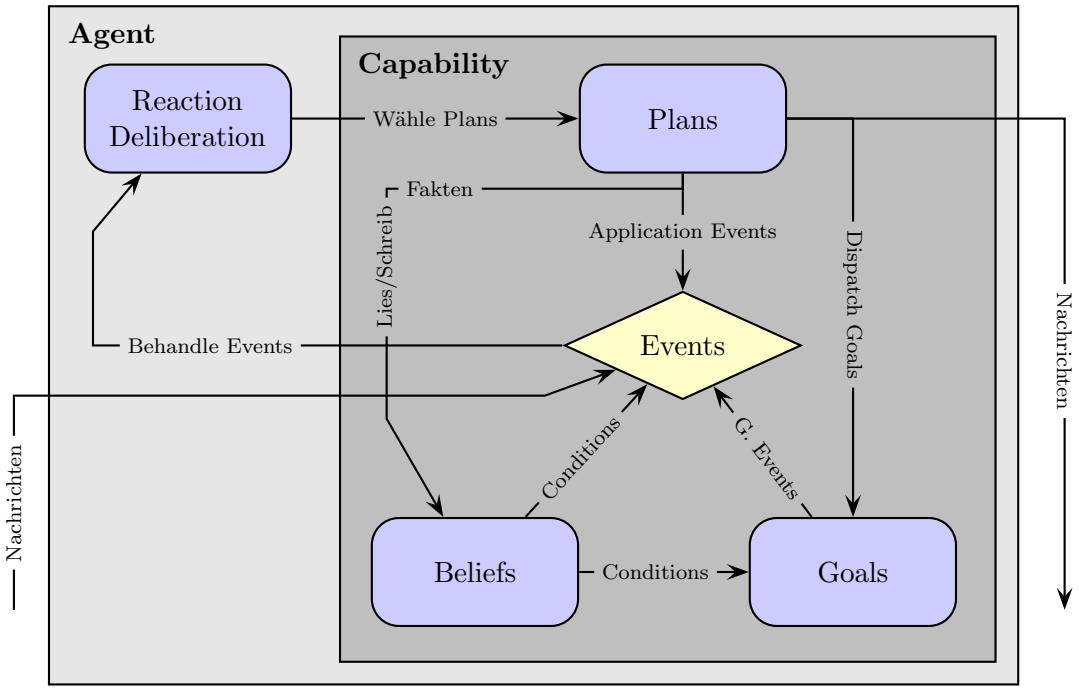

Abbildung 1: Abstrakte Architektur

1. *option* – Das goal wurde angenommen, wird aber derzeit nicht verfolgt.
2. *active* – Das goal wird aktiv verfolgt.
3. *suspended* – Das goal verfügt über eine derzeit nicht erfüllte Kontextbedingung.

Zu Beginn seines Lebens wird jedes goal erzeugt, sei es, weil seine Triggerbedingungen erfüllt sind (z. B. „immer wenn die Sonne scheint, erzeuge ein Autowaschgoal“), weil es explizit dispatcht wird oder weil es sich um ein initiales goal handelt, mit welchem der Agent geboren wird. Für ein erzeugtes goal überprüft Jadex, ob seine Kontextbedingungen (so existent) erfüllt sind. Ist dies nicht der Fall, gelangt das goal in den Zustand suspended, anderenfalls wird es zur option. Aus den sich im Zustand „option“ befindlichen goals wählt der Agent goals zur aktiven Verfolgung aus. Geschieht dies, gelangt das entsprechende goal in den Zustand „active“ und zwar solange, bis der Agent die aktive Verfolgung aussetzt oder bis die Kontextbedingungen nicht mehr erfüllt sind. In ersterem Fall wird das goal erneut zur option, in letzterem Fall wird es suspended. Dieser Zyklus setzt sich solange fort, bis das goal endgültig (erfolgreich oder nicht) verworfen wird.

Eine Besonderheit von Jadex besteht darin, dass goals zueinander nicht konsistent sein müssen. Es ist also durchaus möglich, dass ein Agent über zwei oder mehrere widersprüchliche Ziele verfügt, deren geeignete Behandlung dem (hoffentlich weitblickenden) Entwickler überantwortet wird.

1.2.4 Plans

Konkrete Absichten und Pläne werden in Jadex durch *plans* implementiert. Ein plan ist dabei eine beliebige JAVA-Prozedur zzgl. Informationen darüber, wann diese ausgeführt werden soll und welche Ereignisse sie behandeln kann.

1.2.5 Realisierung in XML und Java

Ein Agent besteht grundsätzlich aus einer XML-Datei – dem Agent Definition File (ADF) – und JAVA-Klassen, wobei das ADF die eigentliche Spezifikation des Agenten enthält, während die JAVA-Klassen die Funktionalität von plans implementieren sowie ggf. zur Erzeugung weiterer benötigter Objekte (z. B. zur Realisierung von beliefs) dienen.

Plans verfügen über einen *head* und einen *body*. Der head befindet sich im ADF und gibt an, wann der plan auszuführen ist. Der body ist Teil einer JAVA-Klasse und implementiert die eigentliche Funktionalität des plan. Der grundlegende Aufbau eines Jadex-Agenten auf Implementierungsebene wird durch die folgende sich selbst erklärende Spezifikation eines Hello-World-Agenten verdeutlicht.

HelloWorldAgent.xml

```
<agent (Schemaangaben) name="HelloWorld">
   <plans>
      <plan name="hello">
         <body>new HelloWorldPlan()</body>
      </plan>
   </plans>
   <initialstates>
      <initialstate name="default">
         <plans>
            <initialplan ref="hello"/>
         </plans>
      </initialstate>
   </initialstates>
</agent>
```

HelloWorldPlan.java

```
import jadex.runtime.Plan;
public class HelloWorldPlan extends Plan {
   public void body() {
      System.out.println("Hello world!");
      killAgent();
   }
}
```

2 Komponenten

Jadex besteht aus verschiedenen miteinander interagierenden Komponenten, von denen einige schon im letzten Abschnitt kurz vorgestellt wurden. Diese und weitere Komponenten werden nun in diesem Abschnitt vertieft behandelt. Um den Rahmen dieser Arbeit einzuhalten, werden jedoch nur die wichtigsten Jadex-Komponenten besprochen.

2.1 Capabilities

Im Rahmen der objektorientierten Programmierung ist es üblich, Funktionalität zu kapseln, indem Klassen von anderen Klassen erben bzw. abstrakte Methoden anderer Klassen implementieren. Ein solches Vorgehen ist auch bei der Entwicklung von Jadex-Agenten wünschenswert und ihm wird durch das Vorhandensein von *capabilities* Rechnung getragen. Eine capability in Jadex ist dabei ein Agent ohne Schlussfolgerungsprozess, also eine Menge von beliefs, goals und plans, welche Funktionalität beschreibt, ohne selbst ausführbar zu sein. Ein Agent kann eine capability einbinden, um so die dort definierte Funktionalität zu erhalten. Ebenso können capabilities weitere subcapabilities enthalten, die ihrerseits wieder subcapabilities enthalten können und so weiter.

2.1.1 Spezifikation

Da der Schlussfolgerungsprozess nicht Bestandteil der Spezifikation des Agenten durch den Entwickler ist, sondern durch Jadex bereitgestellt wird, erfolgt die Spezifikation einer capability analog zur Spezifikation eines Agenten. Es wird lediglich im ADF[4] das äußere `<agent>` tag samt Spezifikation des XML-Schemas durch ein entsprechendes `<capability>` tag substituiert.

Das folgende ADF gehört zu einem Agenten, welcher zwei capabilities einbindet. Zum einen eine im ADF `MyCap.capability.xml` definierte capability unter dem Namen `mysubcap`, zum anderen eine capability des Jadex-Paketes `jadex.planlib.DF` unter dem Namen `dfcap`.

```
<agent ...> ...
   <capabilities>
      <capability name="mysubcap" file="MyCap.capability.xml"/>
      <capability name="dfcap" file="jadex.planlib.DF"/>
   </capabilities> ...
</agent>
```

2.1.2 Sichtbarkeit

Ein Agent bzw. eine äußere capability hat auf die Bestandteile einer eingebundenen capability standardmäßig keinen direkten Zugriff. Soll dies geändert werden, so muss das

[4] Der Einfachheit halber wird auch bei capabilities von ADFs (und nicht etwa CDFs) gesprochen.

Element der capability, auf welches von außen zugegriffen werden soll, im ADF der capability das Attribut `exported=true` gesetzt haben. Ferner muss im ADF des Agenten bzw. der äußeren capability das eingebundene Element per Referenz aufgerufen und benannt werden.

Angenommen es wurde gemäß des letzten ADF-Auszuges eine capability unter dem Namen `mysubcap` in den Agenten eingebunden. Um nun auf ein belief in dieser capability vom Agenten aus zugreifen zu können, muss dieses zuerst im ADF der capability für den Zugriff von außen freigegeben werden:

```
<belief name="example" exported="true" class="MyFact"> ...
```

Nun kann der Agent das belief `example` der eingebundenen capability `mysubcap` unter dem Namen `mysubbelief` einbinden und so darauf zugreifen.

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

2.1.3 Abstrakte Elemente

Ein abstraktes Element ist ein Element, welches an der Stelle, an der es definiert wird, nicht implementiert wird. Die Implementierung kann oder muss durch den Agenten bzw. die äußere capability erfolgen, welche die das abstrakte Element enthaltende capability einbindet. Abstrakte Elemente werden durch eine abstrakte Elementreferenz der Form `<beliefref ...><abstract/></beliefref>` spezifiziert. Die Zuweisung einer Implementierung des abstrakten Elementes im Agenten bzw. in der äußeren capability erfolgt durch `<assignto>`. Ein abstraktes Element muss implementiert werden, solange das `<abstract>` tag nicht über das Attribut `required` mit Wert `false` verfügt.

Beispiel: Innere Capability (Auszug)

```
<beliefref name="abs" exported="true" class="MyF">
   <abstract/>
</beliefref>
```

Beispiel: Äußere Capability (Auszug)

```
<belief name="mybelief" class="MyF">
   <assignto ref="mysubcap.abs">
</belief>
```

2.2 Beliefs

Beliefs werden im ADF spezifiziert und können von plans gelesen und modifiziert werden. Beliefs dienen in Jadex dabei nicht nur der Implementierung von beliefs im Sinne des BDI-Modells, sondern auch zur Realisierung von weiteren agentenspezifischen Daten. Denkbar wäre z. B. ein allgemeiner motivation plan, der abhängig von in der beliefbase des Agenten hinterlegten Charaktereigenschaften goals für den Agenten dispatcht.

2.2.1 Spezifikation

Der Beliefabschnitt des ADFs wird durch ein `<beliefs>` tag eingeleitet. Innerhalb dieses Abschnittes können einfache single-fact beliefs mittels `<belief>` und beliefsets mittels `<beliefset>` spezifiziert werden. Beide tags verfügen über das Attribut `name`, welches den Namen des belief bzw. beliefset angibt, um an anderer Stelle darauf Bezug nehmen zu können, und über das Attribut `class`, welches festlegt, welche JAVA-Klasse zur Realisierung des bzw. der facts dient. Im Falle eines belief instantiiert Jadex ein JAVA-Objekt dieser Klasse, im Falle eines beliefset ein entsprechend typisiertes `Array`. Sollen facts bekannter Anzahl im ADF spezifiziert werden, so geschieht dies durch Angabe des rechts vom Gleichheitszeichen stehenden Teils der entsprechenden JAVA-Zuweisung für den gewählten Objekttyp inmitten von `<fact></fact>`. Soll ein beliefset mit einer unbekannten Anzahl von facts initialisiert werden, so dient dazu `<facts> someMethod() </facts>`, wobei `someMethod()` eine Methode sein muss, die ein dem Objekttyp des beliefset entsprechend typisiertes `Array` zurückliefert.

Der folgende ADF-Ausschnitt soll die Möglichkeiten zur Spezifikation von beliefs und beliefsets verdeutlichen.

```
<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>"Charly"</fact>
   </beliefset>
   <beliefset name="my_opponents" class="String">
      <facts>Database.getOpponents()</facts>
   </beliefset>
</beliefs>
```

Beim ersten Element handelt es sich um ein single-fact belief der Klasse `Location`, welches mit einer neuen `Location` „Hamburg“ initialisiert wird. Das zweite Element ist ein beliefset der Klasse `String`, welches Jadex als `String-Array` der Länge 3 mit Inhalt `Alex`,

`Blandi`, `Charly` instantiiert. Ein Beispiel für ein beliefset der Klasse `String` mit einer Menge von facts unbekannter Anzahl stellt das dritte Element dar. Dort wird eine Methode `Database.getOpponents()` aufgerufen, welche ein `String-Array` zum Spezifikationszeitpunkt nicht notwendig bekannter Länge returniert.

2.2.2 Zugriff

Um aus plans auf beliefs zugreifen zu können, steht eine Reihe von Methoden bereit. Zunächst wird die beliefbase durch Aufruf der Methode `getBeliefbase()` alloziert, welche ein Objekt vom Typ `IBeliefbase` zurückliefert. Dieses Objekt verfügt wiederum über Methoden `getBelief()` und `getBeliefset()`, welche als Parameter einen String akzeptieren, der den Namen des gewünschten belief bzw. beliefset angibt, und Objekte vom Typ `IBelief` bzw. `IBeliefSet` returniert. Um auf die Fakten eines belief bzw. beliefset zugreifen zu können, steht die Methode `getFact()` bzw. `getFacts()` zur Verfügung. Erstere liefert im Falle eines belief das Factobjekt, zweitere ein entsprechend typisiertes `Array`.

Ferner stehen die Methoden `containsFact()`, `update -Fact()`, `removeFact(fact)`, `addFact(fact)` und `setFact()` bereit, um ein belief bzw. beliefset auf das Vorhandensein eines bestimmten facts zu überprüfen sowie die facts von beliefs und beliefsets zu modifizieren.

2.2.3 Auswertung und Propagation

Die Methodenaufrufe zu im ADF spezifizierten initialen Fakten werden von Jadex standardmäßig nur beim Start des Agenten getätigt. Dieses Verhalten ist jedoch in vielen Fällen nicht wünschenswert, z. B. wenn ein belief immer die aktuelle Uhrzeit enthalten soll. Es ist daher möglich durch Angabe des Attributes `evalutionmode="dynamic"` im `<fact>` tag Jadex den fact bei jedem Zugriff auswerten zu lassen.

Wie schon in Abschnitt 1.2.2 erläutert, können beliefs als Triggerbedingungen für andere Jadex-Komponenten dienen. Deshalb kann es erforderlich werden, dass Fakten regelmäßig aktualisiert werden müssen, selbst wenn nicht auf sie zugegriffen wird. Dies ist durch Angabe des Attributes `updaterate="`n`"` im `<belief>`- bzw. `<beliefset>` tag möglich, wobei n die Länge des Aktualisierungsintervalles in Sekunden angibt. Die Angabe einer updaterate für ein belief bzw. beliefset impliziert dabei dynamische Evaluation für die im belief bzw. beliefset enthaltenen Fakten.

Aus der Tatsache, dass Änderungen von beliefs andere Jadex-Komponenten triggern können, erwächst eine weitere Notwendigkeit: Die Propagation von Beliefänderungen innerhalb von Jadex. In Fällen, in denen Fakten neue Objekte zugewiesen werden, geschieht dies automatisch. Wird hingegen ein komplexes zu einem Fakt gehörendes Objekt modifiziert, so müssen explizit events gefeuert werden, um Jadex von den Änderungen in Kenntnis zu setzen.

2.3 Goals

Wie schon im ersten Abschnitt erläutert, dienen goals der Modellierung von Zielen eines Agenten.

2.3.1 Typen

In Jadex gibt es vier Arten von tatsächlichen goals sowie *meta-level goals*, welche in Abschnitt 2.3.7 behandelt werden.

- Ein *perform goal* gibt eine zu erledigende Tätigkeit an, wobei die Ausführung der Tätigkeit und nicht das Erzielen eines Ergebnisses für das Erreichen des goal entscheidend ist (z. B. die Tätigkeit „Rasenmähen“ im Gegensatz zum Zustand „wohlgeschnittener Rasen“).
- Abstrakte Zielzustände werden hingegen von *achieve goals* beschrieben. Im Falle des Zielzustandes „wohlgeschnittener Rasen“ wäre es dem Agenten völlig gleichgültig, ob dieser Zustand statt durch Rasenmähen etwa durch eine gleichmäßig grasende Karnickelpopulation erreicht würde.
- Ein Bedarf an Information wird durch ein *query goal* ausgedrückt. Ein am Zustand seines Rasens interessierter Agent könnte sich z. B. für die Essgewohnheiten der nächstgelegenen Karnickelpopulation interessieren, um geeignete Maßnahmen zum Schutze seines Rasens treffen zu können.
- Ist es das Ziel, einen Zustand zu halten, so werden dazu *maintain goals* verwendet. Plans zur Behandlung eines solchen goal werden genau dann dispatcht, wenn die maintain condition verletzt wird.

Neben diesen Goal-Typen wird ferner zwischen top-level goals und subgoals unterschieden. Subgoals stellen Teilziele auf dem Weg zur Erreichung eines top-level goal dar und können im Gegensatz zu diesem nur von laufenden plans dispatcht, nicht aber initial im ADF spezifiziert werden.

2.3.2 Parameter

Goals können Parameter haben, die z. B. dazu dienen können, allgemein gehaltene goals zu konkretisieren (man denke an ein Gehe-zu-Ort-Goal mit einer Ortsangabe als Parameter) oder bei einem query goal die erlangte Information zu beinhalten. Analog zur Unterscheidung zwischen beliefs und beliefsets wird bei Parametern zwischen `<parameter>` und `<parameterset>` unterschieden.

Ob ein Parameter vom goal genutzt oder vom goal gefüllt wird (oder beides), spezifiziert das Attribut `direction`, welche Werte aus {`in`, `out`, `inout`} annehmen kann. Das Attribut `optional` gibt an, ob eine Füllung des Parameters verpflichtend ist. Der eigentliche Parameterwert wird mittels `<value>` bzw. `<bindingoptions>` spezifiziert.

Name	Default	mögliche Werte
`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}`
`metalevelreasoning`	`true`	`{true,false}`

Tabelle 1: allen Goal-Typen gemeine BDI-Flags

2.3.3 Exklusivität und Bedingungen

Beinhaltet ein goal das `<unique/>`-Element, so wird sichergestellt, dass es immer nur ein goal gleichen Typs und gleicher Parameter gleichzeitig gibt. Mittels `<exclude>` kann zusätzlich angegeben werden, welche Parameter beim Vergleich dabei nicht berücksichtigt werden sollen.

Bedingungen, wann ein goal automatisch instantiiert werden soll, können durch Angabe von `<creationcondition>` spezifiziert werden. Weiterhin besteht die Möglichkeit mittels `<contextcondition>` Kontextbedingungen festzulegen (siehe auch Abschnitt 1.2.3). Wird ein goal suspended, weil seine Kontextbedingungen nicht erfüllt sind, so werden alle zum goal gehörigen plans und subgoals terminiert und bei Wiederaufnahme des goal ggf. neu instantiiert. Eine endgültige Verwurfbedingung kann durch `<dropcondition>` angegeben werden. Einmal verworfene Goals können nach Auslösen einer solchen Bedingung nicht reaktiviert werden.

2.3.4 BDI-Flags

BDI-Flags sind Attribute von Jadex-Komponenten. Sie können entweder im ADF oder zur Laufzeit mittels einer `set`-Methode für einzelne Goal-Instanzen spezifiziert werden. Die für alle Goal-Typen verfügbaren BDI-Flags sind in Tabelle 1 dargestellt.

`retry` gibt an, ob ein goal nach Beendigung automatisch wieder aufgenommen werden soll. Ist dies der Fall, so kann mittels `retrydaily` eine Wartezeit in Sekunden zwischen Beendigung und Wiederaufnahme angegeben werden. Durch `exclude` wird dabei spezifiziert, ob für die erneute Behandlung des goal bereits versuchte (`when_tried`), bereits erfolgreiche (`when_succeeded`), bereits fehlgeschlagene (`when_failed`) oder gar keine Pläne (`never`) ausgenommen werden sollen.

Normalerweise werden geeignete Pläne zur Behandlung eines goal vom Agenten seriell ausprobiert. Durch Setzen von `posttoall` auf `true` kann parallele Bearbeitung erzwungen werden. Hat einer der parallelen Pläne Erfolg, so werden alle anderen für dieses goal

gestarteten Pläne terminiert.

Durch Angabe von `randomselection` wählt der Agent zwischen den zur Behandlung eines goal geeigneten Plänen gleicher Priorität zufällig aus.

Das `metalevelreasoning`-Flag kann benutzt werden, um den Reasoningprozess auszuschalten, welcher bei Vorhandensein mehrerer plans zur Behandlung eines goal zwischen diesen auswählt.

2.3.5 Spezielle Bedingungen

Abhängig vom Goal-Typ sind spezielle Bedingungen zu spezifizieren:

Achieve Goals

- Die `<targetcondition>` gibt den zu erreichenden abstrakten Zielzustand an.
- Mittels `<failurecondition>` kann angegeben werden, wann das goal als nicht mehr erreichbar gelten soll.
- Fehlt die Angabe von Bedingungen, so bestimmt der Erfolg der zur Behandlung des goal ausgeführten Pläne den Erfolg des goal.

Query Goals Query goals verfügen über die implizite Zielbedingung, dass alle Goal-Parameter mit `direction=out` (also diejenigen Parameter, welche die zur Stillung des Informationsbedarfs erlangten Informationen erhalten sollen) Werte enthalten. Liegt statt eines einfachen Parameters ein `<parameterset>` vor, so gilt für dieses, dass es mindestens einen Wert beinhalten muss.

Maintain Goals

- Eine Angabe der zu erhaltenden Bedingung mittels `<maintaincondition>` ist bei maintain goals verpflichtend.
- `<targetcondition>` gibt optional an, welcher Zustand bei Verletzung der maintain condition angestrebt werden soll. Dies ist z. B. dann sinnvoll, wenn ein nur knappes Erreichen der zu erhaltenden Bedingung wenig Sinn ergibt (man denke an einen Roboter mit Akku und eine maintain condition „Akkuladung $\geq 0,2$“, der im Falle eines niedrigen Akkustandes seinen Akku nicht nur bis 0,2 sondern vollständig aufladen soll).

2.3.6 Easy Deliberation

Aus dem im Abschnitt 1.2.3 beschriebenen Goal-Lebenszyklus erwächst die Notwendigkeit zu entscheiden, wann ein goal mit Zustand „option“ aktiviert und wann ein aktiviertes goal wieder zur option werden soll. Der von Jadex unterstützte Entscheidungsprozess heißt

Easy Deliberation und basiert auf zwei Komponenten: *Goal-Kardinalitäten* und *Sperrbeziehungen*.

Goal-Kardinalitäten geben an, wie viele goals eines Goal-Typs gleichzeitig verfolgt werden dürfen. Die Spezifikation geschieht durch `<deliberation cardinality="`*n*`">`.

Eine Sperrbeziehung regelt, welche goals von einem aufgenommenen goal blockiert werden. Sperrbeziehungen werden innerhalb des `<deliberation>`-Abschnittes mittels `<inhibits ref="`*gn*`">` spezifiziert, wobei *gn* den Namen des goal angibt, dessen Aufnahme gesperrt werden soll solange das entsprechende goal selbst aufgenommen ist. Soll die Sperrung nur dann gelten, wenn das sperrende goal gerade verarbeitet wird, kann dies durch das Attribut `inhibit="when_in_process"` angegeben werden.

2.3.7 Meta-Level Goals

Ein meta-level goal beschreibt kein tatsächliches vom Agenten zu erreichendes Ziel, sondern – wie der Name vermuten lässt – ein Meta-Ziel. Ein solches Meta-Ziel besteht i. d. R. darin, bei Vorliegen mehrerer zur Behandlung eines goal geeigneter plans den tatsächlich auszuführenden plan auszuwählen. Tritt eine solche Situation auf und soll sie durch meta-level reasoning behandelt werden, so muss ein entsprechendes meta-level goal („Finde einen plan!“) dispatcht werden. Das meta-level goal wird nun durch einen meta-level plan behandelt, der den tatsächlich auszuführenden plan bestimmt. Die zur Auswahl stehenden plans erhält der meta-level plan dabei aus dem `in`-Parameter `applicables` des meta-level goal. Das Ergebnis wird im `out`-Parameter `result` des goal hinterlegt.

Die Definition eines meta-level goal geschieht mittels `<metagoal>` und muss mindestens einen `<trigger>` beinhalten.

2.3.8 Beispiele

Perform Goal Das nachfolgende perform goal lässt den Agenten immer dann patrollieren, wenn gerade sein Akku nicht lädt (`!beliefbase.is_loading`) und es nicht Tag ist. Wegen `retry="true"` und `exclude="never"` terminiert es nie. Man beachte ebenfalls, dass für XML reservierte Zeichen durch XML-Entitäten angegeben werden müssen (`&amp;&amp;` statt `&&`).

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

Achieve Goal Das nachfolgende achieve goal enthält als Parameter einen Zielort und erhält über die beliefbase den Ort, an dem sich der Agent gerade befindet. Die abstrakte Zielbedingung ist dann erfüllt, wenn der Rückgabewert der `isNear`-Methode für den aktuellen Ort mit dem Zielort als Parameter `true` wird.

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

Query Goal Das nachfolgende query goal liefert ein Beispiel für die Abfrage der beliefbase in der bereits in Abschnitt 1.2.2 erwähnten OQL-ähnlichen Sprache („wähle den ersten Mülleimer aus einer nach Entfernung aufsteigend sortierten Liste, die aus den Mülleimern der beliefbase besteht, die nicht voll sind“). Das Ergebnis dieser Abfrage wird in den `out`-Parameter namens `result` geschrieben. Die Abfrage wird bei jedem Zugriff durchgeführt (`evalutionmode="dynamic"`) und das goal wird fortwährend wiederholt (`retry="true"`, `exclude="never"`).

```
<querygoal name="qw" exclude="never" retry="true">
   <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>
```

Maintain Goal Das nachfolgende maintain goal lässt den Agenten danach streben, seinen Akkuladezustand immer größer als 0,2 zu halten. Wird diese Bedingung verletzt, so wird es jedoch zum Ziel des Agenten, den Akku wieder vollständig aufzuladen und nicht etwa nur soweit aufzuladen, dass die maintain condition erfüllt wird.

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

2.4 Plans

Wie bereits im ersten Abschnitt erläutert, zeigen sich plans für das Verhalten von Agenten hauptverantwortlich. Ihre (im ADF spezifizierten) heads geben an, unter welchen Bedingungen der jeweilige plan auszuführen ist, ihre (in Form von JAVA-Klassen vorliegenden) bodies beinhalten JAVA-Prozeduren, die das Verhalten des zugehörigen Agenten beschreiben.

2.4.1 Erzeugung

Hinsichtlich der Erzeugung von plans wird zwischen *active plans* und *passive plans* unterschieden. Ein active plan wird bei der Instantiierung des Agenten erzeugt, terminiert typischerweise nicht und stellt Dienstleistungen bereit. Eine Passive-Plan-Instanz wird instantiiert, wenn die Trigger-Bedingungen des plan erfüllt sind, und behandelt normalerweise nur den Vorfall, der zu seiner Instantiierung geführt hat. Eine geeignete Analogie findet sich im Bereich der UNIX-Netzwerkdienste, die entweder (wie ein active plan) als daemon laufen oder (wie ein passive plan) vorfallabhängig vom inetd aufgerufen werden.

Active plans werden im (hier nicht näher erläuterten) `<initialstates>`-Abschnitt des ADF als Referenz spezifiziert. Passive plans verfügen über ein `<trigger>`-Element, innerhalb dessen angegeben wird, welche Bedingungen zu ihrer Instantiierung führen. Es wird dabei zwischen einer eventgetriebenen und datengetriebenen Auslösung unterschieden.

Elementgetriebenes Triggern kann durch die tags `<internalevent>`, `<messageevent>` und `<goal>` spezifiziert werden. Diese Trigger können durch Angabe von Parametern (durch `<parameter>`) weiter verfeinert werden. Ein plan wird in diesem Fall genau dann getriggert, wenn alle vorgegebenen Parameter mit den tatsächlichen Parametern übereinstimmen.

Datengetriebenes Triggern bezieht sich auf Änderungen bzw. den Zustand der beliefbase. Mögliche Bedingungen können durch `<condition>`, `<factadded>`, `<factremoved>`, `<beliefchange>` und `<beliefsetchange>` angegeben werden.

Ein einfacher datengetriebener Trigger eines Reparaturplanes, der dann getriggert wird, wenn etwas laut beliefbase „out of order“ ist, könnte z. B. so ausschauen:

```
<trigger>
   <condition>
      $beliefbase.out_of_order
   </condition>
</trigger>
```

2.4.2 Vor- und Kontextbedingungen

Die Angabe von Vorbedingungen erfolgt durch `<precondition>`, die Angabe von Kontextbedingungen durch `<contextcondition>`. Die Erfüllung der Vorbedingungen wird nur beim Start des Agenten überprüft, während Kontextbedingungen während der gesamten Lebensdauer des plan erfüllt sein müssen.

Angenommen der Reparaturplan des letzten Beispiels soll nur dann aktiv sein, wenn das zu reparierende Teil auch tatsächlich reparierbar ist, so könnte man diesen Umstand durch folgende Kontextbedingung ausdrücken:

```
<contextcondition>
   $beliefbase.repairable
</contextcondition>
```

2.4.3 Waitqueues

Angenommen ein plan funktioniere wie folgt:

1. Warte auf Nachricht A.
2. Führe Berechnungen durch.
3. Warte auf Nachricht B.
4. Verschicke das Ergebnis der Berechnungen.

Ein nur aus einem einzelnen Thread bestehender plan wird – da die Durchführung von Berechnungen Zeit verbraucht – auf eine während des zweiten Schrittes eintreffende B-Nachricht nicht reagieren können, da das zugehörige message event ausgelöst wird, während der plan nicht darauf wartet. Angenommen die Nachricht B signalisiert das Interesse eines anderen Agenten am Ergebnis der Berechnungen. In diesem Falle wäre es wünschenswert, das eintreffende Ereignis zwischenspeichern und bei Bedarf abrufen zu können. Dies wird in Jadex durch *waitqueues* realisiert.

Eine waitqueue wird durch das `<waitqueue>`-Element im ADF spezifiziert, innerhalb dessen all diejenigen Event-Typen angegeben werden, deren zugehörige events in die waitqueue aufgenommen werden sollen. Mittels `getWaitqueue()` erfolgt der Zugriff auf die waitqueue aus dem body eines plan heraus.

2.4.4 Cleanup

Abhängig vom Erfolg eines plan kann es notwendig sein, Aufräumcode zu spezifizieren, welcher nach Beendigung des plan ausgeführt wird.

Ein plan ist erfolgreich, wenn er ohne exception terminiert. Sein Aufräumcode wird im plan body durch Angabe der Methode `passed()` spezifiziert. Das Werfen einer exception kennzeichnet das Fehlschlagen eines plan. In diesem Fall erfolgt die Angabe des Aufräumcodes durch die Methode `failed()`. Es kann ferner vorkommen, dass ein plan vorzeitig terminiert wird, z. B. weil das von ihm behandelte goal nicht mehr aktuell ist. Ein solches Vorkommnis kennzeichnet einen abgebrochenen plan, für dessen Aufräumcode die `aborted()`-Methode zuständig ist.

2.5 Events

Jadex ist eventbasiert, nichts geschieht ohne events. Die einzelnen Event-Typen und ihre Eigenschaften werden in diesem Abschnitt beschrieben.

2.5.1 Event-Typen

Es existieren die folgenden Event-Typen:

- `<goalevent>` – wird intern automatisch in Form eines process event (falls ein active goal verfolgt werden soll) oder in Form eines info event (falls die Verarbeitung eines goal beendet ist) ausgelöst
- `<internalevent>` – ermöglicht die explizite Einwege-Kommunikation innerhalb des Agenten
- `<messageevent>` – Nachricht von oder nach außen
- vollständig interne events, z. B. zur Propagation von Beliefänderungen

2.5.2 Parameter und Attribute

Es besteht bei events (wie schon bei den goals) die Möglichkeit, `<parameter>` bzw. `<parameterset>` zu spezifizieren, die über das Attribut `direction="`x`"` ($x \in$ {`in`, `out`, `inout`}) verfügen können.

Ferner ist bei events die Angabe der bereits aus Abschnitt 2.3.4 bekannten BDI-Flags `posttoall`, `metalevelreasoning` und `randomselection` möglich.

2.5.3 Message Events

Alle zu sendenden und zu empfangenden message event types müssen im ADF spezifiziert werden und sind derzeit vom Typ „FIPA“, da Jadex noch kein anderes Nachrichtenformat unterstützt. Ein message event type kann dabei über das Attribut `direction="`x`"` ($x \in$ {`send`, `receive`, `send_receive`}) verfügen, welches angibt, ob der Nachrichtentyp zum Versand oder zum Emfang von Nachrichten oder zu beidem dient.

Zu versendende Nachrichten werden einfach versandt, bei empfangenen Nachrichten muss jedoch erst festgestellt werden, zu welchem message event type die empfangene Nachricht gehört, da die Spezifikation der message event types für jeden Agenten lokal erfolgt. Dies geschieht durch einen Parametervergleich. Es werden dabei nur diejenigen Parameter verglichen, deren Inhalt geeignet ist den Nachrichtentyp zu beschreiben (denn es wäre offensichtlich sinnlos, z. B. den Inhalt eines Feldes `unique-id` zur Charakterisierung der Nachricht heranzuziehen). Ein solcher Parameter wird durch das Attribut `direction="fixed"` gekennzeichnet, welcher nicht mit dem oben genannten Message-Event-Attribut gleichen Namens verwechselt werden sollte.

Aus plans heraus können message events mittels `createMessageEvent(String` type) erzeugt werden, wobei „type“ den Namen eines im ADF spezifizierten Nachrichtentyps angibt. Der Empfänger einer Nachricht wird durch ein `AgentIdentifier`-Objekt angegeben, der Nachrichteninhalt mittels `setContent(Object` o) gesetzt und schließlich durch `sendMessage()` abgeschickt. Ein message event in Jadex wird immer durch eine Klasse instantiiert, die das Interface `IMessageEvent` implementiert.

Um wechselseitige Gespräche zwischen Agenten zu vereinfachen, können Nachrichten über den Parameter `conversation-id` bzw. `reply-with` verfügen. Eine konforme Antwort auf eine solche Nachricht kann dann einfach mittels `createReply()` erzeugt werden.

2.5.4 Beispiele

Internal Event Das nachfolgende internal event könnte dazu dienen, einer GUI-Klasse mitzuteilen, dass sie Inhalte aktualisieren soll. Der zu aktualisierende Inhalt wird dabei vom feuernden plan in den Parameter `content` geschrieben. Die GUI-Klasse muss des Weiteren auf das Eintreffen von events des Typs `gui_update` lauschen und bei Eintreffen eines solchen event den Parameter auslesen.

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

Message Event Die nachfolgende Spezifikation beschreibt einen Nachrichtentyp für ausgehende Nachrichten. Der Parameter `performative` gibt an, um was für eine Art von Nachricht es sich handelt (z. B. `REQUEST` oder `INFORM`); die hier möglichen Werte sind durch die FIPA-Spezifikation vorgegeben. Das Nachrichtenfeld `reply-with` wird mit einer eindeutigen ID gefüllt, um eine Antwort auf diese Nachricht zuordnen zu können.

```
<events>
   <messageevent name="request_carry" type="fipa" direction="send">
      <parameter name="performative" class="String" direction="fixed">
         <value>SFipa.REQUEST</value>
      </parameter>
      <parameter name="reply-with" class="String">
         <value>SFipa.createUniqueId(...)</value>
      </parameter>
   </messageevent>
   ...
</events>
```

3 Fazit

Jadex stellt alle grundlegenden Funktionen bereit, um komfortabel Multiagentensysteme entwickeln zu können. Die Verwendung von Standardtechnologien wie JAVA und XML sowie die Anlehnung an das bekannte BDI-Modell sorgen dabei für eine kurze Einarbeitungszeit. Einige im Rahmen der Projektgruppe notwendigen Funktionen, wie z. B. die Unterstützung von Logik und Inferenz oder die Modellierung einer gemeinsamen Umwelt, werden von Jadex nicht explizit unterstützt, so dass hier Erweiterungen vorgenommen werden müssen.

Literatur

[1] Alexander Pokahr, Lars Braubach, Andrzej Walczak. *Jadex User Guide (Release 0.941)*. Internet: `http://tinyurl.com/r4dly`, 2005 (Stand: 2006–03–25).

[2] Lars Braubach, Alexander Pokahr, Andrzej Walczak. *Jadex Tutorial (Release 0.941)*. Internet: `http://tinyurl.com/h657v`, 2005 (Stand: 2006–03–25).

[3] Diverse Autoren. *Wikipedia*. Internet: `http://de.wikipedia.org/` (Stand: 2006–04–20).