Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Ausarbeitung

Sven Tennstedt

Veranstalltung AW2
Domain Specific Languages für die Beschreibung von Agentenverhalten

Fakultät Technik und Informatik
Department Informatik

Inhaltsverzeichnis

1 Motivation

Unter anderem durch meine Teilnahme am Robocup, einem Wettbewerb in dem Hardware-Agenten (Roboter) bzw. Software-Agenten gegeneinander im Fußball antreten, und dem Roboter-Projekt an der HAW Hamburg, ein im Bachelorstudiengang der Informatik angebotenes Projekt, in dem mit c't bots (Heise Zeitschriften Verlag GmbH & Co. KG, 2006) Aufgaben gelöst werden sollen, konnte ich Erfahrung in der Entwicklung von Agentenverhalten sammeln. Ich wurde damit konfrontiert, dass Agentenverhalten in der Praxis anscheinend sehr häufig mit Hilfe von objektorientierten oder prozedualen Sprachen implementiert werden.

Diese Sprachen bilden allerdings gängige Agentenkonzepte wie Brooks Subsumption Architektur oder deliberative Modelle wie das BDI-Modell nur zum Teil oder gar nicht ab. Der Entwickler muss sich erst eine Umgebung in dieser Sprache schaffen, die das verwendete Konzept unterstützt. Je nach Komplexität des Konzeptes, kann dieser Aufwand erheblich sein. Eine Mischung der Konzepte der Hostsprache und der der Agentenmetapher ist zudem unvermeidlich und erschwert eine klare Definition eines Agentenverhaltens.

Da allerdings die Agentenmetapher die Entwicklung komplexer verteilter Systeme erleichtern soll, stellt sich die Frage, ob es speziell für die Programmierung von Agentenverhalten entwickelte Sprachen gibt? Um die Fragestellung zu konkretisieren, wird das iFlat der HAW Hamburg als Orientierung genommen. Dort ist das Ziel die Hausgeräte als Agenten zu modellieren. Zur Zeit werden auch Alternativen betrachtet, unter anderem Aufgaben, die von dem iFlat zu lösen sind, als Prozesse mit BPEL (WS-BPEL2.0) zu modellieren. Die Modellierung geschieht hierbei allerdings vor dem Betrieb und nicht dynamisch. Ein Multiagentensystem könnte dagegen diese Dynamik liefern. Zudem verspricht es Robustheit gegenüber Fehlern, d. h. wenn z. B. ein Gerät ausfällt, soll das Gesamtsystem stabil weiter arbeiten. Es verspricht Skalierbarkeit, d. h. beliebige neue Geräte sollen dem System möglichst ohne oder geringem Konfigurationsaufwand hinzugefügt werden können und das Gesamtsystem damit erweitern.

Im Hinblick auf das iFlat wäre ein hybrides Agentenkonzept, d. h. deliberativ und reaktiv, sehr praktikabel, da die deliberative Ebene sich zwar besonders für qualitative Entscheidungen eignet, aber schlechter für quantitative. Diese sind einfacher durch eine reaktive Ebene zu realisieren.

Ein sehr simples Szenario: Der Besuch von Freunden steht an. Für diese wird immer etwas besonderes gekocht, das bedeutet ein Gericht muss ermittelt werden. Das kann die deliberative Ebene entscheiden. Die reaktive Ebene ermittelt, was dafür noch eingekauft werden muss.

In dieser Arbeit konzentriere ich mich auf den Teilaspekt der Beschreibung von deliberativen Agenten. Alle hier vorgestellten Sprachen arbeiten daher nach dem BDI-Konzept (siehe dazu Abschnitt 3).

In meiner Ausarbeitung „Agentenbeschreibung: Expressiv AI (Façade) im Vergleich zu

traditionellen Ansätzen“ (Tennstedt, 2008) verglich ich das Vorgehen Agenten zu modellieren in Façade mit dem in Jadex (Jadex, 2008). Der Schwerpunkt lag dort darauf, die Beschreibung von Agentenverhalten auch Nicht-Programmierern zu ermöglichen. Mein Fazit war, dass dies eine Agentenplattform wie Jadex nicht leisten kann. Stattdessen muss für einen konkreten Anwendungsfall eine eigene Sprache entwickelt werden. In dieser Arbeit möchte ich allerdings den Blick auf den gelernten Programmierer lenken.

In der Bachelorarbeit „Aktuelle Entwicklungstrends im Bereich von Programmiersprachen“ (Schempp, 2006) wird der Trend zu Domain Specific Languages (DSLs) beschrieben. Ebenso in „Auf dem Weg zu idealen Programmierwerkzeugen Bestandsaufnahme und Ausblick“ (Brauer u. a., 2008) nachzulesen.

Davon ausgehend möchte ich im Folgenden Agentensprachen als DSLs ansehen und die vorgestellten Sprachen u. a. anhand ihrer Orthogonalität zur Hostsprache beurteilen. Da dies eine Art Robustheit der Sprache widerspiegelt, mit anderen Worten die Wartbarkeit des Programmcodes verbessert.

Zunächst erläutere ich die Bezeichnung „Domain Specific Language“ und grenze **interne** und **externe** DSLs voneinander ab (Abschnitt 2, S. 4) . Darauf aufbauend unterscheide ich anschließend die Agentensprachen als interne und externe Sprachen (Abschnitt 3, S. 6). Am Ende erfolgt eine Zusammenfassung der Ergebnisse (Abschnitt 4, S. 11).

2 Domain Specific Languages

> *„The basic idea of a domain specific language (DSL) is a computer language that's targeted to a particular kind of problem, rather than a general purpose language that's aimed at any kind of software problem. Domain specific languages have been talked about, and used for almost as long as computing has been done.*
> *[...] DSLs are very common in computing: examples include CSS, regular expressions, make, rake, ant, SQL, HQL, many bits of Rails, expectations in JMock, graphviz's dot language, FIT, strut's configuration file. [...]"*
>
> — Fowler (2009)

Domain Specific Languages (DSLs) sind nach Fowler nicht neu. Seitdem es Computer gibt, werden DSLs benutzt. Sie werden erschaffen um spezielle Probleme zu bewältigen, im Gegensatz zu allgemeinen Sprachen wie z. B. C, Java, Smalltalk oder Ruby, deren Zweck darauf abzielt für alle Arten von Software-Problemen eingesetzt zu werden.

Er unterscheidet dabei interne und externe DSLs (siehe Fowler (2009) bzw. Fowler (a)), auf die ich im Folgenden eingehen werde.

2.1 Interne DSLs

Eine interne DSL bedient sich der Hostsprache und nutzt deren Möglichkeiten und Paradigmen um ein domänenspezifisches Problem zu bewältigen. Als wahrscheinlich populärste Erscheinung gilt hier Ruby on Rails. In Ruby hat sich das Arbeiten mit Symbolen besonders durchgesetzt, was das Rails Framework stark prägt. So werden im Rails Framework z. B. die Stellvertreterobjekte für Datenbanktabellen wie folgt zu einem internen Schema verknüpft:

```
1 class Tablename << ActiveRecord::Base
2     belongs_to :OtherTablename, :through => :RelationshipTablename
3 end
```

belongs_to ist hierbei eine Klassenmethode und :OtherTablename ist ein Symbol, das im Allgemeinen den Klassennamen des Stellvertreterobjekts einer anderen Datenbanktabelle enthält. Der zweite Parameter ist ein Schlüssel-Wert-Paar, kann aber auch eine Hashtabelle sein, und sagt aus, dass die beiden Tabellen über eine dritte Tabelle verbunden sind. Verallgemeinert nennt Fowler dies *Literal Collection Expression* (Fowler (b)). Ein weiteres Beispiel:

```
1 computer(processor(:cores => 2, :type => :i386),
2     disk(:size => 150),
3     disk(:size => 75, :speed => 7200, :interface => :sata))
```

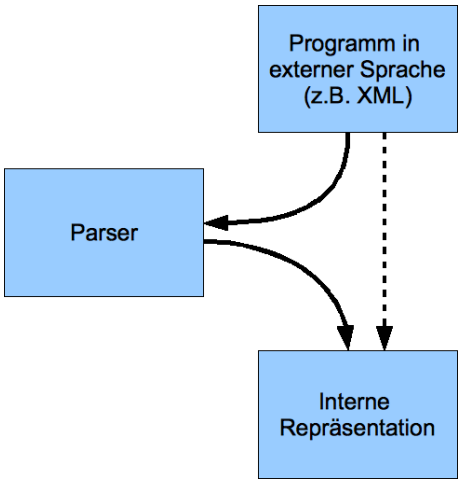

Abbildung 1: Verarbeitung von externen Domain Specific Languages: Schematische Darstellung

Dies ist ein komplexeres Beispiel in dem ein Computer konfiguriert werden soll. Der Methode computer wird eine Collection übergeben, die aus den Rückgabewerten der Methoden processor und disc besteht. Den Mehtode processor und disc werden wie in dem oberen Beispiel Schlüssel-Wert-Paare übergeben (key => value). Als Schlüssel und Werte kommen wieder Symbole zum Einsatz.

Als weitere Beispiele wie interne DSLs arbeiten nennt Fowler u. a. das *Method Chaining* und die *Nested Function*, nachzulesen im Anhang A ab Seite 13.

2.2 Externe DSLs

Eine externe DSL hingegen ist von der Hostsprache entkoppelt und bildet eine eigenständige Sprache.

Konfigurationsdateien z. B. in XML bezeichnet Fowler schon als externe DSL, da mit ihnen das Verhalten der Zielapplikation verändert werden kann.

Die schematische Abbildung 1 zeigt den Grundgedanken bei der Verwendung von externen DSLs. Es gibt eine externe Repräsentation einer Aufgabe bzw. einer Problemlösung in Form eines Programms in der DSL, ein Parser verarbeitet dieses Programm und generiert daraus die interne Darstellung in der Zielsprache. Der Parser kann je nach Aufgabe von einem einfachen Textparser für Konfigurationsdateien bis zu einem eigenen Compiler oder Interpreter einer spezialisierten Hochsprache reichen.

2.3 Interne- und Externe-DSLs gegenüber gestellt

Heutige IDEs sind sehr komplex und bieten eine Vielzahl an Werkzeugen, um die Software-Entwicklung zu vereinfachen, wie etwa Refactoring und Debugging Werkzeuge. Diese für DSLs nutzen zu können ist wünschenswert. Bei internen DSLs ist diese Unterstützung prinzipbedingt stets gegeben, da sie ein Subset der Hostsprache darstellen. Hingegen müssen externe DSLs diese Werkzeuge mitliefern, z. B. in Form von PlugIns für Eclipse (Eclipse, 2009). Dies ist leider häufig nicht der Fall, was unter anderem das Debugging erschwert.

Die Semantik von internen DSLs ist gegenüber der ihrer jeweiligen Hostsprache nicht klar abgegrenzt, da sie, wie oben erwähnt, ein Subset der Hostsprache darstellen. Die Ausdruckmöglichkeiten der Hostsprache bleiben dadurch erhalten. Es liegt am Entwickler sich an die Semantik zu halten. Bei externen DSLs ist die Semantik dagegen klar abgegrenzt und muss keine Gemeinsamkeiten mit der Hostsprache aufweisen. Eine externe DSL kann daher in jeder beliebigen allgemeinen Programmiersprache implementiert werden.

Je nach Domäne muss der Entwickler einer DSL Vor- und Nachteile beider Varianten abwägen. Die gleiche Abwägung muss ein Entwickler vornehmen, um sich für eine geeignete DSL für seine Zwecke zu entscheiden.

3 Sprachen zur Beschreibung von Agentenverhalten

Agenten-Programmiersprachen können als DSLs betrachtet werden. Im Folgenden werden einige Agenten-Programmiersprachen vorgestellt und dabei die vorherige Unterscheidung von interner und externer Sprache vorgenommen.

Alle aufgeführten Sprachen arbeiten nach dem so genannten Belief-Desire-Intention-Konzept (BDI-Konzept). Hierbei handelt es sich um ein Agentenkonzept für deliberative Agenten. Unter *Beliefs* versteht man die Wissensbasis des Agenten, d. h. seine interne Repräsentation der Welt. Da dieses auch falsch sein kann, heißt es nicht *Fact* oder *Knowledge*, sondern *Belief*. *Desires* sind Ziele die ein Agent verfolgt und erfüllen möchte. *Intentions* sind Pläne, die zur Erfüllung eines oder mehrere Ziele führen sollen. Näheres zu Agentenkonzepten findet man in Wooldridge (2002) und Ferber (2001).

Im Folgenden wird die Bezeichnung „mobile Geräte“ synonym für Smartphones und PDAs gebraucht.

3.1 interne Agentenprogrammiersprachen

Wie in Abschnitt 2 erläutert wurde, basieren interne Agentenprogrammiersprachen auf den Konzepten ihrer jeweiligen Hostsprache. Die beiden folgenden Systeme Jadex und JACK basieren auf Java.

3.1.1 Jadex

Jadex (Jadex, 2008) wurde an der Universität Hamburg entwickelt und wird dort aktiv gepflegt. Inzwischen ist die Version 2.0 in Arbeit. Zusätzlich wurde die Rule Engine (JadexRuleEngine) als separates Projekt ausgegliedert, so dass sie separat und in eigenen Projekten verwendbar ist. Das Jadex System ist Open Source.

In Jadex liegen die Pläne als Javaklassen vor und können von unterschiedlichen Agententypen verwendet werden, sofern sie in ihrer Konzeption austauschbar sind. Das eigentliche Agentenverhalten wird in der so genannten „Agent Description Language“ (ADL), die in XML implementiert ist, beschrieben. Auch wenn diese Trennung den Anschein macht, dass es sich um eine externe DSL für Agenten handelt, so muss die fehlende Orthogonalität des Konzeptes kritisiert werden. In der ADL wird explizit auf Java-Objekte verwiesen oder sie werden erstellt. Aus diesem Grund ordne ich Jadex zu den internen Agenten-Programmiersprachen. Ein gekürzter Jadex-Beispielcode, der diese Eigenschaft demonstriert, ist im Anhang B.1 (S. 14) zu finden.

Für die Kommunikation der Agenten kommt die Kommunikationsplattform JADE zum Einsatz (JADE). Dadurch ist es Jadex möglich mit Nicht-Jadex-Agenten zusammen zu arbeiten.

Steckbrief von Jadex:

DSL-Typ intern

Sprache Java und XML

Sprachkonzept objektorientiert / imperativ

Agentenplattform JADE[1], implementiert FIPA-ACL (FIPA:ACL, 2002).

3.1.2 JACK

JACK (AOS:JACK) ist das einzige hier vorgestellte kommerzielle Produkt. Es wird von der Firma AOS (AOS) entwickelt und vertrieben. Das System ist Closed Source und proprietär.

JACK wird mit einer eigenen IDE ausgeliefert, dem so genannten *JACK Development Environment* (JDE), das eine graphische Programmierumgebung enthält, sowie Tools bietet um die Entwicklung von JACK Agenten zu unterstützen. AOS verspricht, dass JACK auf allen gängigen Java-Plattformen läuft, also auch auf mobilen Geräten. Die JACK-Agenten sollen sehr leichtgewichtig sein, d. h. dass selbst auf kleinen Geräten hunderte bis tausende von Agenten gleichzeitig ausgeführt werden können. Allerdings geht aus den Produkt-Informationen zu JACK nicht hervor, ob JACK mit Agenten aus anderen Systemen kommunizieren kann. Es scheint, dass dies nicht vorgesehen ist.

[1] JADE wurde von der Telecom Italia entwickelt und bietet eine auf den FIPA Standards basierende Plattform um Multiagentensysteme zu entwickeln

Ich bezeichne Jack salopp als eine Java-Syntax-Erweiterung und zwar, weil die Java-Syntax um Elemente ergänzt wird, die es ermöglichen in Java BDI-basierte Agenten zu entwickeln. Ein Precompiler erzeugt daraus dann Java-Code. Ein Jack-Codegerüst ist im Anhang B.2 (S. 15) zu sehen.

Steckbrief von JACK:

DSL-Typ intern

Sprache eine erweiterte Java-Syntax

Sprachkonzept objektorientiert und imperativ

Agentenplattform proprietär, keine konkreten Hersteller-Angaben

3.2 externe Agentenprogrammiersprachen

Als externe Agenten DSLs werden hier 3APL und 2APL von der Universität Utrecht aus den Niederlanden vorgestellt, sowie Jason, ein AgentSpeak(L)-Interpreter.

3.2.1 3APL: An Abstract Agent Programming Language

Die Entwicklung von 3APL (3APL) ist anscheinend inzwischen eingestellt. Auch wenn 3APL der Vorgänger von 2APL ist, soll es hier dennoch vorgestellt werden. Das System wird anscheinend ausschließlich für Forschungs- und Lehrzwecken entwickelt, die an der Utrecht Universität stattfinden, der Zweck besteht anscheinend nicht darin dieses für reale agentenbasierte Softwareprojekte einzusetzen. Der Source Code steht nicht auf der Website zur Verfügung, was daran liegen kann, dass 3APL auf einen kommerziellen Prologinterpreter (Chirico) aufbaut. Dieser bildet den Kern von 3APL und daher sieht die Syntax der Sprache der Prolog-Syntax sehr ähnlich. Wie in STRIPS werden in der Agentenbeschreibung Vor- und Nachbedingungen für das Auslösen bestimmter Aktionen definiert. Ein Codebeispiel ist im Anhang B.3 (S. 16) zu finden.

Aus unbekannten Gründen läuft 3APL nur unter Windows XP[2] oder Mac OS X, unter Windows Vista scheiterten die Startversuche. Da zudem das System mit Java 6 kompiliert wurde, ist ein Betrieb auf mobilen bzw. embedded Geräten ausgeschlossen.

[2] ältere Systeme nicht berücksichtigt

Steckbrief von 3APL:

DSL-Typ extern

Sprache 3APL, prologähnliche Syntax

Sprachkonzept logisch und imperativ

Agentenplattform Agent Managment System (AMS): proprietär, an FIPA Standard orientiert

3.2.2 2APL: A Practical Agent Programming Language

2APL (2APL) ist der Nachfolger und die Weiterentwicklung von 3APL. Genauso wie sein Vorgänger wird 2APL anscheinend nur zu Forschungs und Lehrzwecken an der Universität Utrecht eingesetzt. Da bei 2APL wieder JIProlog (Chirico) den Kern bildet, steht der Source Code der Sprache ebenfalls nicht zur Verfügung.

Im Gegensatz zu 3APL kann 2APL über die Agentenkommunikationsplattform JADE mit anderen auf JADE basierenden Agenten zusammen arbeiten. Ansonsten ist 2APL vom Prinzip und der Syntax her seinem Vorgänger sehr ähnlich (siehe Anhang B.4, S. 17). Jedoch gibt es die STRIPS-artigen Klauseln nicht mehr und die Syntax wurde etwas vereinfacht.

Wie bei 3APL läßt sich 2APL nur unter Windows XP[3] oder Mac OS X starten. 2APL ist wie 3APL mit Java 6 kompiliert worden. Ein Einsatz auf mobilen bzw. embedded Systemen ist daher zur Zeit ausgeschlossen.

Steckbrief von 2APL:

DSL-Typ extern

Sprache 2APL, prologähnliche Syntax

Sprachkonzept logisch und imperativ

Agentenplattform JADE

3.2.3 Jason: Ein Java AgentSpeak(L) Interpreter

Jason ist ein auf Java basierender Interpreter für die Agentensprache AgentSpeak(L) (Rao, 1996) und wurde maßgeblich von Hübner und Bordini entwickelt (Hübner und Bordini). Jason ist Open Source und wird aktiv gepflegt.

Die Agenten in Jason kommunizieren über SACI (Hübner und Sichman) miteinander. Dabei handelt es sich um eine Agentenkommunikationsplattform, die von Hübner mitentwickelt

[3] ältere Systeme nicht berücksichtigt

wurde. Die Kommunikation findet in KQML statt. Jason ist jedoch so aufgebaut, dass diese Komponente auch gegen eine andere Plattform ausgetauscht werden kann, z. B. mit JADE.

Jason liefert eine eigene IDE auf Basis von jEdit (jEdit, 2009) mit, die Syntax Highlighting beherrscht und in der man ein Projekt mit einem Mausklick kompilieren und starten kann.

Das Agentenverhalten wird im AgentSpeak(L)-Dialekt definiert. So genannte *Actions* werden jedoch in Java implementiert, die dann in Jason als Prädikate zur Verfügung stehen. Zudem ist es möglich das System auf Javaseite anzupassen. So können eigene Agentenklassen erstellt werden, auf die dann die Jasonagenten aufbauen, oder die Umgebung[4] definiert werden, wenn man Umwelteigenschaften in dem System abbilden möchte.

Jason setzt Java 5 voraus, ist damit auf mobilen bzw. embedded Systemen zur Zeit nicht einsetzbar.

Steckbrief von Jason:

DSL-Typ extern

Sprache AgentSpeak(L)-Dialekt

Sprachkonzept logisch

Agentenplattform SACI

[4] Eine Java Klasse die von Environment erbt.

4 Fazit

Nachdem die Sprachen einzeln vorgestellt wurden, möchte ich im folgenden die Ergebnisse auswerten.

Jadex ist ein erprobtes System, das von mehreren Entwicklern aktiv gepflegt und weiterentwickelt wird. Die Rules-Engine ist sogar separat erhältlich, so dass man eigene Systeme damit entwickeln kann. Bei der Entwicklung mit Jadex kann man auf Jadex-eigene Tools zurückgreifen, die die Entwicklung gut unterstützen. Der Hauptkritikpunkt an Jadex stellt jedoch die fehlende Orthogonalität der Sprache dar. Der Einsatz in embedded oder mobilen Systemen ist mit Jadex wahrscheinlich nicht möglich.

Wenn man den Leitsatz Alan Kays **„Simple things should be simple, complex things should be possible“** zugrunde legt, so erfüllt Jadex diesen nicht. Bis ein einfacher Agent erstellt ist, bedarf es viel Schreibarbeit, was nicht nur die Einarbeitung in Jadex erschwert, sondern ebenso die Wartung eines in Jadex entwickelten Gesamtsystems.

JACK orientiert sich sehr an den Bedürfnissen der Industrie und führt keine neue Sprache ein, sondern ergänzt die bestehende Java-Syntax. Nach Angaben des Herstellers ist ein Einsatz in mobilen Geräten möglich, was dadurch unterstrichen wird, dass JACK-Agenten sehr leichtgewichtig sind. Theoretisch wäre ein Einsatz im iFlat der HAW Hamburg möglich. Leider macht der Hersteller auf seiner Website keinerlei Angaben zu Lizenzkosten. Außerdem sehe ich es eher kritisch, dass das System, im Gegensatz zu Jadex, 2APL und Jason, von Haus aus nur mit JACK-Agenten kooperieren kann.

3APL und **2APL** zeigen eine relativ einfach zu erlernende Syntax, sofern man sich schon einmal mit Prolog auseinander gesetzt hat, doch werden beide Sprachen anscheinend ausschließlich für interne Lehrzwecke der Utrecht University verwendet. Die Quellen sind nicht verfügbar, somit sind eigene Anpassungen und Veränderungen nicht möglich.

Jason implementiert die BDI-Agentensprache AgentSpeak(L) und ist mit dem oben erwähnten Leitsatz eher in Verbindung zu bringen als Jadex. Einfache Agenten erfordern nur wenige Zeilen Code und es ist möglich ein komplexes System zu erstellen. Jason wird zwar mit einer IDE auf jEdit-Basis geliefert, bringt aber nicht so viele Werkzeuge wie Jadex mit, um das Multiagentensystem während des Betriebs zu überwachen.

Abschließend läßt sich sagen, dass die Einstiegshürde bei Systemen wie Jadex bzw. Jason relativ niedrig ist. Beide Systeme holen die meisten Entwickler aus einem gewohnten Umfeld ab, die objektorientierte und imperative Entwicklung in Java. Selbst wenn man bei Jadex den Mangel an Orhtogonalität mit einbezieht.

Bei Systemen wie 3APL, 2APL und Jason ist dagegen vorher eine Auseinandersetzung mit logischer Programmierung notwendig.

JACK wirkt insgesamt recht aufgeräumt und klar strukturiert, nicht nur im direkten Vergleich mit Jadex. Erwartungsgemäß hinterlassen die externen Agentensprachen einen klareren Eindruck und wirken strukturierter.

Der Einsatz in mobilen Geräten ist nur bei JACK gesichert. Diese Einschränkung der anderen Systeme erschwert deren Einsetzbarkeit in einer heterogenen Umgebung bzw. schränkt ihren Nutzen ein.

Nach reichlicher Recherche komme ich zu dem Schluß, dass es keine Sprachen gibt, die ein hybrides Konzept unterstützen. Nun stellt sich die Frage, ob es unbedingt notwendig ist, eine Sprache zu haben, die die Konzepte der deliberativen und reaktiven Agenten vereinigt? Genauso könnte es praktikabel sein, ein hybrides Multiagentensystem zu entwerfen, das aus deliberativen und reaktiven Agenten besteht, die Teams bilden um Aufgaben zu lösen. Diese Aufgaben können dann von der Art sein, wie ich sie in dem Szenario des Besuchs der Freunde zu Beginn dieser Ausarbeitung gezeigt habe (Abschnitt 1, S. 2). Die deliberativen Agenten übernehmen die qualitativen Entscheidungsprozesse, die reaktiven die quantitativen. Das hat den Vorteil, dass die einzelnen Agenten dadurch simpler werden und einfacher zu entwerfen sind.

Diese Arbeit zeigt zudem, dass für die Entwicklung von deliberativen Agenten Sprachen und Werkzeuge existieren, bzw. es Erfahrungen gibt, wie diese Sprachen entworfen werden können. Auf diese Erfahrungen kann bei der Entwicklung eines Agentenbasierten iFlat aufgebaut werden.

Entsprechend dem BDI-Konzept bei den deliberativen Agenten gibt es bei den reaktiven Agenten die Subsumption Architektur von Brooks. Für diese gibt es die verschiedensten Ausprägungen in unterschiedlichen Programmiersprachen, aber keine explizite DSL. Dazu wäre ein Blick auf die Arbeiten von Maes (Maes, MIT) interessant. Wie in „MultiAgent Systems“ (Wooldridge, 2002, Kap 5.1) beschrieben, hat sie das Konzept von Kompetenz-Modulen mit STRIPS-ähnlichen Vor- und Nachbedingungen vorgeschlagen.

A Domain Specific Languages

A.1 Method Chaining

Method Chaining ist eine Methode, wie sie zum Beispiel in Smalltalk oder Ruby sehr leicht verwendet werden kann, da diese Sprachen standardmäßig das aufgerufene Objekt als Rückgabewert eine Methode liefern.

So kann man nach Beispiel von Fowler einen Computer folgender maßen mit Method Chaining konfigurieren:

```
computer()
  .processor()
    .cores(2)
      .i386()
  .disk()
    .size(150)
  .disk()
    .size(75)
    .speed(7200)
    .sata()
.end();
```

A.2 Nested Function

Ist gegenüber der Method Chaining leicht in jede funktionale, prozeduale oder objektorientierte Sprache zu implementieren. Es werden Funktionen ineinander verschachtelt aufgerufen. Der Rückgabewert der einen Funktion liefert einen Parameter für die Übergeordnete Funktion.

```
computer(
    processor(
      cores(2),
      Processor.Type.i386
    ),
    disk(
      size(150)
    ),
    disk(
      size(75),
      speed(7200),
      Disk.Interface.SATA
    )
  );
```

B Codebeispiele der Agentensprachen

B.1 Jadex

Ein Jadex Plan in Java:

```
1  public class BroadcastVisitedPositionsPlan extends Plan {
2    /**
3    * The plan body.
4    */
5    @Override
6    public void body() {
7      // fetch parameters:
8      String teamID = (String) this.getBeliefbase().getBelief("teamID").getFact();
9
10     // fetch parameter:
11     MapPoint[] mps = (MapPoint[]) this.getParameterSet("visited_positions").getValues();
12
13     // create message content:
14     VisitedPositionsMessage msg_content = new VisitedPositionsMessage();
15     msg_content.setMps(mps);
16
17     AgentDescription[] agents = getTeamMembers(teamID);
18     for (AgentDescription ad : agents) {           // send message
19       if (! ad.getName().getLocalName().contains(this.getAgentName())) {
20         // don't send to self
21         IMessageEvent mevent = createMessageEvent("visited_position_inform");
22         mevent.setContent(msg_content);
23         mevent.getParameterSet(SFipa.RECEIVERS).addValue(ad.getName());
24         sendMessage(mevent);
25       }
26     }
27     System.out.println(this.getAgentName() + " broadcasted visited positions");
28   }
29 }
```

Die Agentenkonfiguration in der *Agent Description Language*:

```
1  <agent xmlns="http://jadex.sourceforge.net/jadex"
2    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://jadex.sourceforge.net/jadex
4  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣http://jadex.sourceforge.net/jadex-0.96.xsd"
5    name="Cleaner"
6    package="jadex.examples.cleanerworld.multi.cleaner">
7
8    <beliefs>
9      <!-- The known wastes. -->
10     <beliefset name="wastes" class="Waste" />
```

```
11
12      <!-- The statistics of visited map points. -->
13      <beliefset name="visited_positions" class="MapPoint">
14        <!-- The facts tag generates several facts retrieved by the stated expression -->
15        <facts>
16          MapPoint.getMapPointRaster($beliefbase.raster[0].intValue(), $beliefbase.raster[1].intV
17        </facts>
18      </beliefset>
19
20      <!-- The points used for patrolling at night. -->
21      <beliefset name="patrolpoints" class="Location">
22        <fact>new Location(0.1, 0.1)</fact>
23        <fact>new Location(0.1, 0.9)</fact>
24        <fact>new Location(0.3, 0.9)</fact>
25      </beliefset>
26
27    </beliefs>
28
29    <goals>
30      <!-- Broadcast waste object to team members. -->
31      <achievegoal name="broadcast_waste" exported="true">
32        <parameter name="waste" class="Waste" />
33      </achievegoal>
34    </goals>
35
36    <plans>
37      <!-- Broadcast waste-object to team members. -->
38      <plan name="broadcast_waste">
39        <parameter name="waste" class="Waste">
40          <goalmapping ref="broadcast_waste.waste" />
41        </parameter>
42        <body class="BroadcastWastePlan"></body>
43        <trigger>
44          <goal ref="broadcast_waste"/>
45        </trigger>
46      </plan>
47    </plans>
48  </agent>
```

B.2 JACK

Eine Agentklasse:

```
1  agent ExampleAgent extends Agent {
2    // #-statements
3    // data members
4
```

```
5    ExampleAgent (String name)
6    {
7      super(name);
8    ...
9    }
10 }
```

Eine Plandefinition:

```
1  plan MovementResponse extends Plan {
2
3    #handles event RobotMoveEvent moveResponse;
4    #uses agent implementing RobotInterface robot;
5
6    static boolean relevant (RobotMoveQEvent ev) {
7      return (ev.ID == RobotMoveQEvent.REPLY_SAFE ||
8          ev.ID == RobotMoveQEvent.REPLY_DEAD);
9    }
10
11   Body() {
12     if (moveResponse.ID==REPLAY_SAFE)
13     {
14       System.err.println("Robot␣Safe␣to␣Move");
15       robot.updatePosition(moveResponse.Lane,
16         moveResponse.Displacement);
17     } else {
18       // robot didn't make it
19       System.err.println("Robot␣is␣Dead");
20       robot.die();
21     }
22   }
23 }
```

B.3 3APL

```
1  PROGRAM "cleaning"
2  CAPABILITIES{
3    { pos(P) }     Goto(R) { NOT pos(P) , pos(R) },
4    { pos(P) AND dirty(R) } Vacuum(R) { NOT dirty(R) },
5    { pos(P1) AND box(P1)}  Movebox(P1,P2)  { NOT pos(P1), NOT box(P1), pos(P2), box(P2)},
6    {TRUE}      IsClean() {clean()},
7    {TRUE}      Transported() {transport()}
8  }
9  BELIEFBASE{
10   dirty(room1).
11   dest(room1).
```

```
12    box(room2).
13    pos(room3).
14  }
15  GOALBASE{ clean(), transport() }
16  PLANBASE{ }
17  PG-RULES{
18    clean() <- dirty(Room) |
19    { Goto(Room);
20      Vacuum(Room);
21      if not dirty(R) then IsClean()
22    },
23    transport() <- box(Room) AND dest(Dest) |
24    { Goto(Room);
25      Movebox(Room,Dest);
26      if box(Dest) then Transported()
27    }
28  }
29  PR-RULES{}
```

B.4 2APL

```
1  Include: person.2apl
2
3  Goals:
4    search( blockWorld )
5
6  Plans:
7    @blockworld( enter( 8, 8, red ), L )
8
9  PG-rules:
10   search( blockWorld ) <- true |
11   {
12     B(is( X, int( random( 15 ) ) ));
13     B(is( Y, int( random( 15 ) ) ));
14
15     goto( X, Y );
16
17     @blockworld( senseBombs(), BOMBS );
18
19     if B( BOMBS = [ [default,X1,Y1] | REST ] ) then
20     {
21       send( harry, inform, La, On, bombAt( X1, Y1 ) )
22     }
23   }
```

Literatur

[2APL] *2APL: A Practical Agent Programming Language*. – URL `http://www.cs.uu.nl/2apl/`. – Zugriffsdatum: 19.02.2009

[3APL] *3APL: An Abstract Agent Programming Language*. – URL `http://www.cs.uu.nl/3apl/`. – Zugriffsdatum: 19.02.2009

[AOS] *AOS: Agent Oriented Software*. – URL `http://www.agent-software.com`. – Zugriffsdatum: 10.02.2009

[AOS:JACK] *JACK - Autonomous Software*. – URL `http://www.agent-software.com/products/jack/index.html`. – Zugriffsdatum: 10.02.2009

[JadexRuleEngine] *Jadex Rule Engine*. – URL `http://jadex.informatik.uni-hamburg.de/rules/bin/view/About/Overview`. – Zugriffsdatum: 18.02.2009

[JADE] *Java Agent DEvelopment Framework (JADE)*. – URL `http://jade.tilab.com/`. – Zugriffsdatum: 10.02.2009

[WS-BPEL2.0] *WS-BPEL 2.0*. – URL `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`. – Zugriffsdatum: 18.02.2009

[FIPA:ACL 2002] *Agent Communication Language Specification*. – URL `http://www.fipa.org/repository/aclspecs.html`. – Zugriffsdatum: 21.02.2009

[Jadex 2008] *Jadex BDI Agent System*. – URL `http://vsis-www.informatik.uni-hamburg.de/projects/jadex`. – Zugriffsdatum: 30.12.2008

[Eclipse 2009] *Eclipse IDE*. – URL `http://www.eclipse.org/`. – Zugriffsdatum: 10.02.2009

[jEdit 2009] *jEdit: Programmer's Text-Editor*. – URL `http://www.jedit.org/`. – Zugriffsdatum: 21.02.2009

[Brauer u. a. 2008] BRAUER, Johannes ; CRASEMANN, Christoph ; KRASEMANN, Hartmut: Auf dem Weg zu idealen Programmierwerkzeugen Bestandsaufnahme und Ausblick. In: *Informatik Spektrum* 31 (2008), Dezember, S. 580 – 590

[Chirico] *JIProlog: Java Internet Prolog*. – URL `http://www.ugosweb.com/jiprolog/index.aspx`. – Zugriffsdatum: 11.02.2009. CHIRICO, Ugo

[Ferber 2001] FERBER, Jacques: *Multiagentensysteme*. Addison-Wesley, 2001. – ISBN 3-8273-1679-0

[Fowler a] *Domain Specific Languages*. – URL `http://www.martinfowler.com/dslwip/`. – Zugriffsdatum: 12.01.2009. FOWLER, Martin

[Fowler b] *Literal Collection Expression*. – URL `http://martinfowler.com/dslwip/LiteralCollection.html`. – Zugriffsdatum: 10.02.2009. FOWLER, Martin

[Fowler 2009] *Domain Specific Languages*. – URL `http://www.martinfowler.com/bliki/DomainSpecificLanguage.html`. – Zugriffsdatum: 12.01.2009. FOWLER, Martin

[Heise Zeitschriften Verlag GmbH & Co. KG 2006] *c't-Bot und c't-Sim*. – URL `http://www.heise.de/ct/projekte/ct-bot/`. – Zugriffsdatum: 21.02.2009

[Hübner und Bordini] *Jason*. – URL `http://www.lti.pcs.usp.br/saci/`. – Zugriffsdatum: 12.01.2009. HÜBNER, Jomi F. ; BORDINI, Rafael H.

[Hübner und Sichman] *Simple Agent Communication Infrastructure*. – URL `http://www.lti.pcs.usp.br/saci/`. – Zugriffsdatum: 12.01.2009. HÜBNER, Jomi F. ; SICHMAN, Jaime S.

[Maes] *MIT Media Lab: Pattie Maes*. – URL `http://web.media.mit.edu/~pattie/`. – Zugriffsdatum: 20.02.2009. MAES, Pattie

[Rao 1996] RAO, Anand S.: AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In: *Agents Breaking Away*. 1996, S. 42–55

[Schempp 2006] SCHEMPP, Ruben: *Aktuelle Entwicklungstrends im Bereich von Programmiersprachen*. Berliner Tor 7, HAW Hamburg, Bachelorarbeit, August 2006

[Tennstedt 2008] TENNSTEDT, Sven: Agentenbeschreibung: Expressiv AI (Façade) im Vergleich zu traditionellen Ansätzen / HAW Hamburg. 2008. – Forschungsbericht

[Wooldridge 2002] WOOLDRIDGE, Michael: *MultiAgent Systems*. Wiley, 2002. – ISBN 0-471-49691-X