Technische Universität Ilmenau

Fakultät für Wirtschaftswissenschaften

Fachgebiet Wirtschaftsinformatik 2

Univ.-Prof. Dr. Stefan Kirn

Zielabbildung in Multiagentensystemen

Projektarbeit

an der Fakultät für Wirtschaftswissenschaften

der Technischen Universität Ilmenau

Betreuender Hochschullehrer: Christian Heine

Bearbeiter: René Drießel

Ausgabetermin:

Abgabetermin:

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**

2.1 Agenten 3

2.2 Agentenumwelt 6

2.3 Multiagentensysteme (MAS) 6

2.4 Ziele und Wissen 7

2.5 Agentenarchitekturen 10

2.6 Methode und Vorgehen der Entwicklung 13

3 Entwicklung des Klassenmodells **14**

3.1 Anwendungsfälle 14

3.2 Grobdesign 16

3.3 Klassenmodell 17

3.3.1 Objekttypen der Zielabbildung 18

3.3.2 Linktypen der Zielabbildung 21

3.4 Algorithmus 22

4 Beispiele **29**

4.1 Fertigungssteuerung 29

4.2 Aquarium 31

5 Schlussfolgerungen **34**

A Diagramme **35**

Literaturverzeichnis **40**

Abkürzungsverzeichnis **41**

Abbildungsverzeichnis **42**

1 Einleitung

Multiagentensysteme (MAS) werden immer häufiger zur Lösung komplexer informationstechnischer Probleme eingesetzt. Dies liegt in den zunehmend komplexeren Entscheidungsprozessen begründet.

In unserer heutigen vernetzten Welt kann ein einzelner Mensch gar nicht mehr alle Variablen, welche ihm für einen solchen Entscheidungsprozess zur Verfügung stehen, verarbeiten. Genau an dieser Stelle setzt die Agenten- und Multiagententechnologie an. Mit Hilfe von verschiedenen Ansätzen, wie zum Beispiel BDI (vgl.Kap. 2.5), wird versucht, genau diese Komplexität zu beherrschen. Im Vordergrund steht hierbei die Unterstützung des Menschen durch personalisierte Agenten.

Die Grundidee dieser neuen Technologie ist dabei allerdings eine alte: man versucht, durch Zerlegung des scheinbar komplexen Problems in viele kleinere, leicht zu lösende Probleme, das Ausgangsproblem zu lösen. Die konkrete Idee bei der Multiagententechnologie besteht darin, dass einzelne Agenten kleine Aufgaben erledigen und in der Summe dabei das Problem lösen. Phänomene aus der Natur werden hierzu oft als Beispiel genommen. Franziska Klügl [Klue2001] beispielsweise hatte 2001 einen Ameisenstaat mit Hilfe der Multiagententechnologie simuliert.

Der Begriff des Agenten immer noch umstritten. Die Frage, ob man es mit einem „normalen“ Programm oder einem Agenten zu tun hat, ist schon seit einigen Jahren umstritten [Frank1996]. Trotzdem wird der Bedarf an diesen neuen Technologie wachsen. Man erhofft sich, die Komplexität der heutigen Anwendungen mit vertretbarem Aufwand und minimiertem Fehlerpotential zu bewältigen.

Eines der Hauptprobleme, mit dem die heutige Agententechnologie noch zu kämpfen hat, ist die Frage der allgemeingültigen Abbildung und Bewertung von Zielen. Speziell mit Hinblick auf die Wiederverwendbarkeit von Agenten ist es wünschenswert, Ziele allgemeingültig abzubilden, da dadurch ein Agent auch in anderen Domänen eingesetzt werden kann. Es existieren zwar schon verschiedene Protokolle mit denen sich die Agenten verständigen können, jedoch bieten diese nur ein Rahmenwerk, um Informationen austauschen zu können. Die „Knowledge Query and Manipulation Language“ (KQML)

[kqml2002] und das „Query Interaction Protocol“ (QIP) der „Foundation for Intelligent Physical Agents“ (FIPA) [fipa00027] stellen einen solchen Ansatz dar. Die Zielabbildung selbst ist bisher allerdings in keiner allgemeinen Art implementiert worden.
Obgleich dies ein wichtiger Punkt innerhalb eines MAS ist, wird diese Arbeit aus Umfangsgründen nur den Bereich der Zielabbildung innerhalb eines Agenten betrachten und nicht den Transport von Zielen zwischen den Agenten. Auch werden die Themen Kooperation und Koordination, welche zweifelsfrei wichtig sind, nicht näher behandelt. Für den interessierten Leser sei hier auf [Kirn2001] verwiesen, welches ein gutes Überblickswissen zu diesem Thema bietet.
Ziel dieser Arbeit ist die *Entwicklung eines Klassenmodells*, mit dem verschiedene Ziele innerhalb eines Agenten abgebildet werden können. Mit Hilfe dieses Klassenmodells soll eine allgemein einsetzbare Lösung geschaffen werden, um unscharfe Ziele, Zielkonflikte, Zielhierarchien usw. abbilden zu können. Gleichzeitig wird darauf zu achten sein, dass dieses Modell mit den bestehenden Standards zur Interagentenkommunikation verwendet werden kann.
Ferner wird der Entwurf eines beispielhaften Algorithmus’ versucht, mit dem eine Verarbeitung dieses Modells für einen lernenden (adaptiven) Agenten möglich ist. Das Hauptaugenmerk wird jedoch auf der Entwicklung des *Klassenmodells* liegen.
Die Idee dabei ist, dass ein Agent mit Hilfe seines Zielsystems in der Lage ist, eine Lösung für ein abstrakt formuliertes Problem selbständig zu finden. Die Implementation soll also nicht in einer Programmiersprache geschehen, sondern man soll sich auf die Definition von Zielen konzentrieren, die der Agent zu erfüllen versucht. Hierdurch wird es möglich, dass der Teil der Arbeit, welcher vom Fachkonzept erledigt werden muss, von dieser Seite aus dem Agenten auch gleich eingegeben werden kann [vgl. Kapitel 2.4].
Diese Art der Problembeschreibung ist auch als deklaratives Programmierparadigma bekannt. Vertreter dieses Paradigmas sind die Sprachen SQL (Structured Query Language) und Prolog. Die Vorteile dieser Art der Programmierung liegen auf der Hand. Anstatt sich damit zu beschäftigen, *wie* das Problem gelöst wird, reicht es aus, das Problem zu *beschreiben*.

2 Grundlagen

2.1 Agenten

Wie bereits erwähnt, ist der Begriff des Agenten in der Literatur sehr umstritten (vgl. Kapitel 1). Speziell der Unterschied zwischen einem „normalen“ Programm und einem Agenten wird stark diskutiert [Frank1996]. Problematisch ist die Abgrenzung deswegen, da man von Agenten so etwas wie Intelligenz erwartet, also etwas, das den Agenten ein wenig „menschlicher“ macht. Da der Begriff der Intelligenz auch nach mehr als 2000 Jahren Philosophie immer noch nicht richtig greifbar ist, tut sich die Agentenforschung schwer damit, eine allgemein aktzeptierte Definition zu finden. So wird versucht, den Begriff Intelligenz indirekt zu umschreiben. Eine Agentendefinition, welche große Beachtung gefunden hat, wurde von Stan Franklin und Art Graesser 1997 vorgestellt:

> „An agent is a system situated within and a part of an environment that senses that environment and acts on it, over time. . . and so as to effect what it senses in the future. “ [Frank1996]

Es existieren jedoch noch wesentlich mehr Definitionsversuche. Pattie Maes vom MIT beispielsweise stellte 1995 folgende Definition vor:

> „Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed.“

Wie man bereits, sieht gibt es sowohl verschiede Aspekte des „Agentenseins“, als auch verschiedene Agententypen. Viele Definitionen beschränken sich daher darauf, die Eigenschaften aufzuzählen, welche einen Agenten auszeichnen. Die Eigenschaften reichen von reaktiv über autonom, rational und proaktiv bis hin zu einer Art Menschenähnlichkeit [Frank1996] [Klue2001].
Im Wesentlichen kann man die nachfolgenden Attribute eines Agenten zusammenfassen [Wool1995] [Klue2001]. Ein Agent

- „lebt“ in einer *digitalen Umgebung*, die in irgendeiner Weise die reale Welt abbildet.
- *handelt autonom:* Lösungswege für nur vage vorgegebene Ziele erarbeitet er selbständig und reagiert unaufgefordert auf Veränderungen seiner Umwelt (reaktiv).
- arbeitet *zielorientiert* und kann *Pläne* zur Lösung eines Problems entwerfen und setzt diese später auch um (proaktiv).
- *kommuniziert* mit Menschen und anderen Agenten. Die Kommunikation mit einem Menschen erfolgt heutzutage mit größter Wahrscheinlichkeit über eine graphische Nutzerschnittstelle. Die Kommunikation zwischen den Agenten muss über eine standardisierte Sprache (z. B. KQML [kqml2002]) erfolgen.
- ist *lernfähig*. In einer Datenbank sammelt er Wissen über seinen Benutzer und arbeitet daher immer zuverlässiger. Für diese Adaptionsfähigkeit wurden bereits viele Lernmechanismen entwickelt und erprobt. So kann der Agent beispielsweise seinem Benutzer „über die Schulter gucken“, oder direkt oder indirekt von ihm lernen (indirekt, wenn ein Vorschlag vom Benutzer abgelehnt wird [Sutt1998]). Der Newsagent Gnus (vgl. [Inge1999]) ist ein gutes Beispiel für direktes Lernen. Er bewertet zum Beispiel die Wichtigkeit eines Artikels nach dem Leseverhalten des Nutzers.
- ist über einen längeren Zeitraum aktiv (*Kontinuität*). Erst dadurch hat der Agent Gelegenheit, „Erfahrungen“ zu sammeln, d. h. zu lernen.

In der weiteren Arbeit wird die knappe, jedoch treffende Definition von Alan Kay verwendet, die er bereits 1984 formulierte ([Kay1984]):

> „[An agent is] a system that when given a goal could carry out the details of the appropriate computer operations and could ask for and receive advice, offered in human terms, when it was stuck.“

Rein technisch betrachtet verhält sich ein Agent wie ein Regelkreis. Er kann über gewisse Sensoren Informationen aus seiner Umwelt aufnehmen und über verschiedene Aktionen

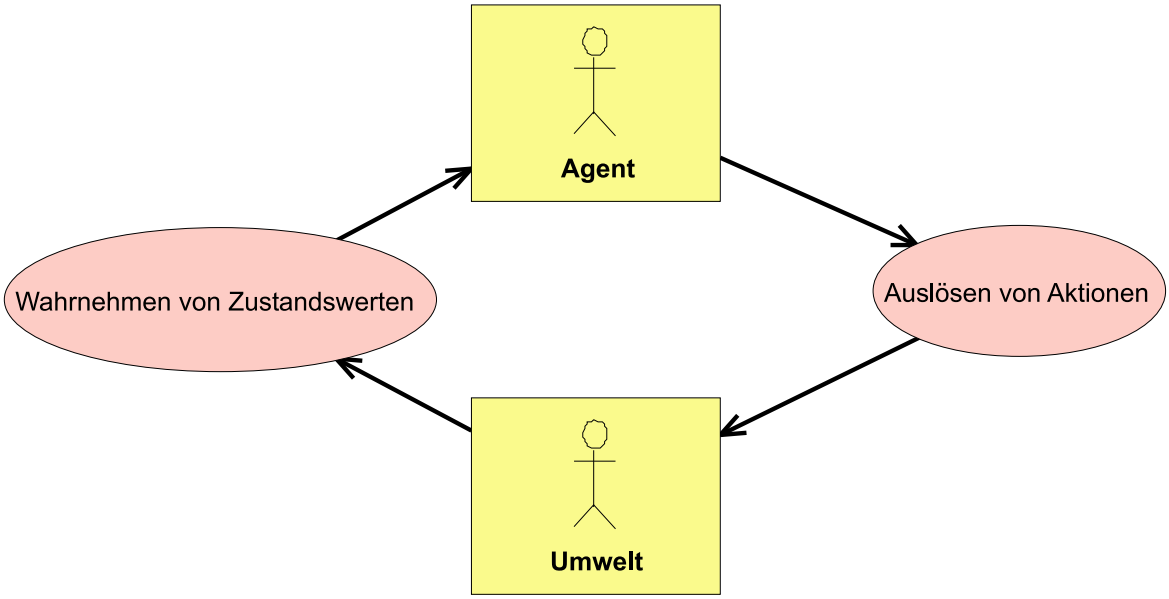

Bild 1: Agent - Umwelt

die Umwelt beinflussen. (Bild 1). Das ganze Geheimnis liegt nun in der sinnvollen Verknüpfung dieser Umweltzustände mit den entsprechenden Aktionen. Je komplexer diese Verknüpfung ist, umso „intelligenter“ – umso „menschenähnlicher“ – erscheint uns der Agent.

2.2 Agentenumwelt

Für die Entwicklung eines Agenten sind die Eigenschaften seiner Umwelt relevant. Unter anderem zählen dazu die folgenden Faktoren [Klue2001]:

- *Zugänglichkeit:* Unter Zugänglichkeit versteht man, wie leicht es für den Agenten ist, an Informationen seiner Umgebung zu gelangen. Daraus resultiert auch, wie berechenbar die Umwelt für den Agenten ist.

- *Dynamik:* Die Dynamik der Umwelt wird durch folgenden Fragen näher beleuchtet: wie veränderlich ist die Umwelt? Wie lang braucht eine Aktion des Agenten, um eine messbare Reaktion in der Umwelt hervorzurufen?

- *Determinierbarkeit:* Wie berechenbar (vorhersagbar) ist die Umwelt? Unter diesem Gesichtspunkt stellt sich auch die Frage nach der Vollständigkeit der Information, welche der Agent aus seiner Umwelt beziehen kann. Bei herkömmlichen Softwareentwicklungen kann man im Regelfall davon ausgehen, dass die Umwelt berechenbar ist und man alle relevanten Informationen für eine Entscheidung hat. Bei Agentensystemen sind solche Begriffe wie „Entscheidung unter Risiko“ und „Entscheidung unter Unsicherheit“ eher die Regel als die Ausnahme. So ist die Umwelt des Textdurchsuchungsprogramm grep klar definiert. Bei einem Roboteragenten, wie der Pathfinder-Mission der NASA, kann man die Umwelt in ihrer Komplexität nie vollständig erfassen.

2.3 Multiagentensysteme (MAS)

Innerhalb eines Multiagentensystems arbeiten mehrere Agenten gemeinsam an der Lösung eines Problems. Im Prinzip unterscheiden sich Multiagentensysteme daher auch nicht sonderlich von herkömmlichen verteilten Softwaresystemen. Der einzige Unterschied ist hier nur, dass nicht mehr verschiedene Objekte miteinander kommunizieren, sondern Agenten. Dadurch ergeben sich die folgenden grundlegenden Eigenschaften von Multiagentensystemen [Klue2001]:

- *dezentrale Datenhaltung:* Innerhalb eines Multiagentensystems ist die Dezentralität der Datenhaltung mit von entscheidender Bedeutung. Dadurch wird das gesamte System skalierbarer und ausfallsicherer.

- *dezentrale Berechnungen:* Aber nicht nur die Daten, sondern auch die Berechnungen sind im Idealfall verteilt. Dadurch lassen sich auch NP-vollständige Probleme mit vertretbarem Aufwand lösen. Selbstverständlich setzt das voraus, dass das entsprechende Problem auch parallelisierbar ist.

- *keine zentrale Kontrolle:* Dieser Punkt ist von ganz entscheidenem Interesse für die Agentenforschung, denn eine zentrale Kontrolle würde die Skalierbarkeit des Systems beeinträchtigen. Leider ist aber gerade dieser Punkt in vielen Systemen nicht realisierbar, da die selbständige Koordination verschiedener Agenten bei vielen Problemstellungen nicht möglich ist.

Daraus ergeben sich jedoch Probleme, welche schon fast philosophischen Charakter haben:

- Ist es gesichert, dass in einer verteilten Umgebung ohne zentrale „Gottinstanz“ das Gesamtoptimum erreicht wird?

- Man kann nicht immer davon ausgehen, dass die Informationen, welche ein anderer Agent liefert, richtig sind.

2.4 Ziele und Wissen

Nach Sicht des Autors ist ein Ziel der Wunsch nach dem Eintreten eines bestimmten Umweltzustandes. Die Beziehung zwischen zwei Zielen wird auch Zielelastizität genannt. Sie drückt aus, inwieweit die Zielerfüllung des einen Ziels von der Erfüllung des anderen Ziels abhängt. Daraus ergeben sich drei grundlegende Zielbeziehungen: Zielkonflikt, Zielneutralität (Indifferenz) und Zielkomplementarität. (Bild 2).

- *Zielkonflikt*: Wenn bei Ziel A der Zielerfüllungsgrad steigt, so sinkt selbiger bei Ziel B.

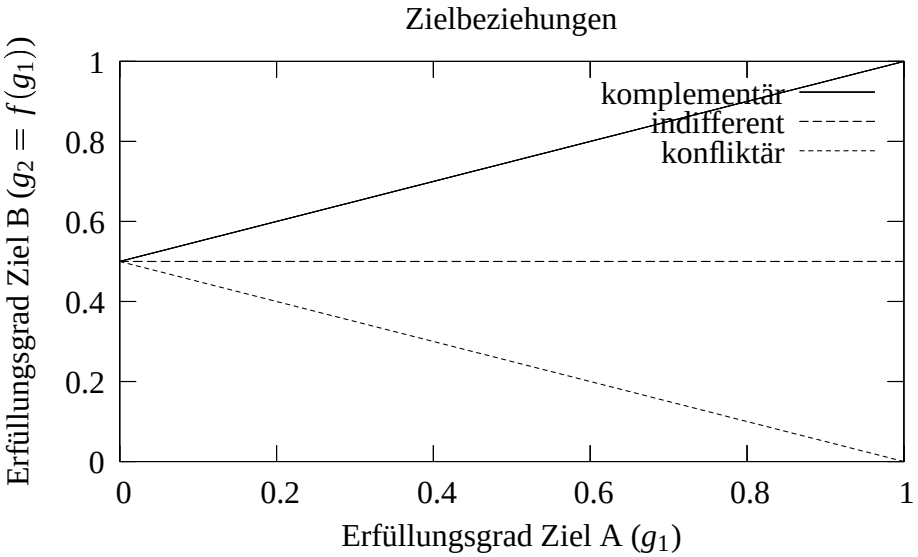

Bild 2: Zielbeziehungen [Hein1992]

- *Zielindifferenz*: Die Erfüllung des Zieles A beeinträchtigt die Erreichung des Zieles B nicht.

- *Zielkomplementarität*: Beide Ziele bestärken sich in ihren Erreichungsgraden, d. h. wenn der Erfüllungsgrad für Ziel A steigt, so steigt er auch bei Ziel B.

In der Praxis existieren verschiedene *Ziele*, welche auch immer domänenabhängig sind. So kann man zum Beispiel die Gewinn- und Umsatzmaximierung sowie die Kostenreduktion als betriebswirtschaftliche Formalziele auf jedes Unternehmen anwenden. Die einzelnen Ziele innerhalb einer speziellen Unternehmung können jedoch auch ganz anders aussehen. So kann zum Beispiel im speziellen Fall das Ziel des Umweltschutzes mit hinzu kommen.

Ein weiterer interessanter Aspekt ist die Möglichkeit von Hierarchiebildungen. Es ist möglich, Ziele in sogenannten Zielhierarchien aufzubauen. Durch diese ist es zum Beispiel machbar, abstrakte Ziele wie Gewinn- und Umsatzmaximierung auf konkretere Ziele, wie die Erhöhung des Umsatzes bei Produkt X um 20%, heruntergebrochen werden.

Daraus ergibt sich eine weitere Unterscheidung zwischen verschiedenen Zielen. Innerhalb einer Zielhierarchie müssen gewisse Ziele existieren, deren Erfüllungsgrade man messen

kann. So ist das Ziel einer Unternehmung, eine langfristige Qualitätssteigerung zu erreichen, nicht messbar. Dem gegenüber sind solche Ziele wie die Senkung des Kohlenstoffgehalt in Stahl sehr wohl messbar. Wenn ein Agent eine Umweltsituation bewerten soll, so braucht er auf jeden Fall messbare Ziele. Im Weiteren wird ein nicht messbares Ziel als Ziel (Goal) bezeichnet. Ein messbares Ziel jedoch als *Zustand* (State).
Wissen kann man als eine Menge von Informationen auffassen, welche man benötigt, um die einzelnen Ziele zu erfüllen (vgl. [Kirn2001]). In dieser Arbeit werden die Ziele, welche ein Agent hat, als Teil seines Wissens aufgefasst.
Im Prinzip kann man Wissen in drei verschiedene Wissensarten unterteilen. Diese sind Domänenwissen, strategisches Wissen sowie kognitives Wissen.

1. *Domänenwissen*: Domänen sind Wissensgebiete. Demzufolge ist Domänenwissen das Wissen über ein bestimmtes Gebiet. Sämtliches Domänenwissen kann in Form von Fakten (als Aussagen) abgebildet werden: „In der Nacht ist es dunkel.“.

2. *Strategisches Wissen*: Strategisches Wissen ist meistens domänenunabhängig. Unter dieser Wissensart versteht man im Prinzip Methodenwissen, welches in Form von Algorithmen abgebildet werden kann. So ist zum Beispiel Quicksort als Sortieralgorithmus in den verschiedensten Domänen anwendbar.

3. *Kognitives Wissen*: Kognitives Wissen ist Wissen über Wissen: „Man weiß, was man weiß.“

Eine einzelne Wissensdomäne kann man unter dem Begriff Ontologie zusammenfassen. Ein Agent besitzt nach dieser Sichtweise Wissen über ein bestimmtes Fachgebiet. Dadurch, dass er dieses Gebiet kennt (kognitives Wissen), ist er in der Lage, mit anderen Agenten zu kommunizieren. So ist es zumindest theoretisch ausgeschlossen, dass man sich missversteht und Begriffe verwendet, die unterschiedliche Bedeutungen (in unterschiedlichen Ontologien) haben können.

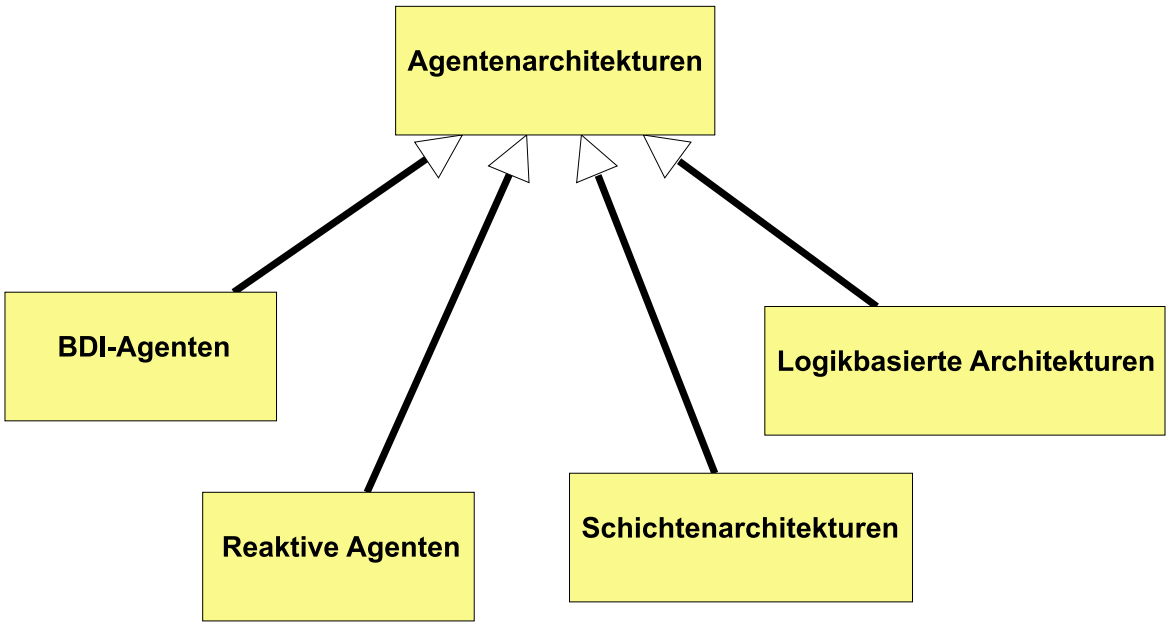

Bild 3: Agentenarchitekturen

2.5 Agentenarchitekturen

Für die technische Umsetzung werden heutzutage vier verschiedene Architekturtypen von Agenten verwendet. Diese sind danach klassifiziert, welche Eigenschaften sie implementieren (Bild 3) (vgl. [Kirn2001] und [Klue2001]).

1. *Reaktive Agenten* basieren auf einfachen Regeln, wie auf gewisse Umweltzustände reagiert werden soll. Dieser Agententyp kann beispielsweise zur Steuerung eines Heizungsthermostats verwandt werden.

2. *BDI-Agenten* stellen den Versuch dar, einen Agenten mit proaktivem Verhalten auszustatten. Während reaktive Agenten eben nur auf die Umwelt reagieren, sollen BDI-Agenten selbst aktiv werden. BDI steht dabei für „Believe, Desire and Intention“ (Überzeugung, Wunsch und Absicht). BDI-Agenten arbeiten mit Hilfe von Entscheidungsbäumen. Hierbei sind die Knoten die Umweltzustände und die Kanten die Aktionen des Agenten oder Ereignisse der Umwelt. Bei letzteren handelt es sich um sogenannte „Chancenknoten“, da der Agent hier keinen Einfluss hat.

3. *Schichtenarchitekturen* basieren auf der Idee, dass sowohl geplantes als auch reaktives Verhalten häufig sinnvoll ist. Der Agent wird demzufolge in mehrere Schichten unterteilt, welche die einzelnen Aspekte berücksichtigten. Über ein Kontrollsystem werden die Schichten miteinander synchronisiert. Die Schichtenarchitekturen werden noch einmal in vertikale und horizontale Schichtenarchitekturen untergliedert.

4. *Logikbasierte Architekturen* basieren auf demselben Prinzip wie Expertensysteme. Auf Basis von Aussagen, welche Fakten und Regeln definieren, hat der Agent ein Bild über den Problembereich. Bei diesem Architekturtyp beweist der Agent Theoreme. Mit Hilfe des deduktiven Schließens werden die Aktionen bestimmt, welche ausgeführt werden sollen.

Sprachen zur Entwicklung von Agenten

Um einen Agenten implementieren zu können und handlungsfähig zu machen, sind verschiedene Sprachen notwendig (vgl. [Kirn2001]). Folgend nun eine Auswahl der benötigten Sprachentypen:

- *Implementierungssprachen* – da die Agententechnologie aus der Objektorientierung hervorgegangen ist, werden hier meist auch objektorientierte Sprachen wie Smalltalk, C++, Python oder Java verwendet.

- *Kommunikationssprachen* dienen der Interaktion zwischen Agenten. In diesem Bereich ist die KQML [kqml2002], die auf der Sprechakttheorie beruht, angesiedelt.

- *Wissensrepräsentationssprachen* werden benötigt, um das Wissen des Agenten über sich und seine Umwelt zu hinterlegen. Hierfür existieren je nach Anwendungsfall verschiedene Möglichkeiten – So zum Beispiel regelbasierte oder „Blackboard"-basierte Sprachen. Als ein Teil dieser Arbeit wird ein semantisches Netz entwickelt, um die Wissens- und Zielabbildung zu ermöglichen.

Methoden der künstlichen Intelligenz

Um die Zielorientierung eines Agenten gewährleisten zu können, ist eine gewisse Lernfähigkeit (Adaptionsfähigkeit) desselben erforderlich. Für diese Probleme existieren bereits einige Ansätze in der künstlichen Intelligenz.

Neuronale Netze beispielsweise sind ein Versuch, ein menschliches Gehirn nachzubilden. Hierbei steht die Adaptionsfähigkeit im Vordergrund. Mittels verschiedener Lernregeln wird versucht, die Fehler, welche das System macht, zu minimieren. Mit Hilfe von neuronalen Netzen sind auf beeindruckende Art und Weise auch komplexe Klassifikationsaufgaben lösbar. Bei den Lernmethoden muss man jedoch die Art des Lernens unterscheiden. Speziell die Art der Klassifikationsprobleme wird haupsächlich über Trainingsdaten gelernt, welche als konkrete Ein- und Ausgabeparameter formuliert sind. Bei diesen Problemen arbeitet man mit einzelnen Neuronen, welche in Schichten angeordnet sind. Zwischen den Neuronen gibt es gewichtete Verbindungen, welche die Aktivierung des Folgeneurons bestimmen.

Bei Problemen, bei denen diese konkreten Ein- und Ausgabedaten nicht bestimmt werden können, kann man einen Algorithmus namens „Reinforcement Learning“ verwenden [Sutt1998]. Hierbei werden in einer Tabelle die einzelnen Aktionen für gewisse Zustände nach ihrem Erfolg gewichtet. Je höher das Gewicht ist, umso wahrscheinlicher ist die Wiederholung der Aktion bei denselben Zuständen.

Bei beiden Methoden gibt es jedoch das Problem, dass man sich in dem Fehlertal in einem lokalen Optimum verlaufen kann.

Genetische Algorithmen hingegen versuchen, mittels Evolution die beste Lösung zu finden. Wie in der Natur wird mit Hilfe von Mutation und Selektion versucht, das Optimum zu finden. Der interessanteste Punkt hierbei ist sicherlich die Mutation, da dadurch die Möglichkeit besteht, aus dem lokalen Optimum wieder „herauszuspringen“.

2.6 Methode und Vorgehen der Entwicklung

Da es sich bei der Agententechnologie um eine Art der Softwareentwicklung handelt, welche auf objektorientierten Prinzipien aufbaut, liegt der Schluss nahe, für die Entwicklung des Zielabbildungsmodells, auch objektorientierte Entwicklungstools zu nutzen. Das heißt,dass alle Anforderungen und konzeptionellen Entwürfe mit Hilfe der „Unified Modelling Language“ (UML) [OMG2001] dokumentiert werden. Diese Arbeit versteht sich als der erste Teil in einem iterativen Vorgehensmodell, mit dem die Ziele innerhalb eines Agenten *allgemein* abgebildet werden können. [1]

[1] Auf Grund der Verfügbarkeit und der universitären Historie der Objekt Technologie Werkbank (OTW) der Firma OWiS wird dieses Programm für den Entwicklungsprozess intensiv eingesetzt [Burk1999].

3 Entwicklung des Klassenmodells

3.1 Anwendungsfälle

Um die einzelnen konkreten Anforderungen an ein Agentensystem darzustellen, wurden die Anwendungsfälle als *Use Cases* [Burk1999] [OMG2001] dargestellt.
Wie bereits erwähnt, kann man das Verhältnis zwischen Agent und Umwelt als einen einfachen Regelkreis beschreiben [vgl. Kapitel 2.1] (Bild 1). Der Agent nimmt über gewisse Sensoren *Umweltzustände* auf und manipuliert gleichzeitig die Umwelt über verschiedene Aktionen. Da er allerdings als Dienstleister für einen Nutzer existiert, muss man die Anforderungen entsprechend erweitern (Bild 4).
Es muss dem Nutzer möglich sein, dem Agenten Anweisungen zu geben. Der Nutzer erzeugt eine Zielhierachie, mit der er dem Agenten seine Wünsche und die Umwelt beschreibt. Anhand der Bewertungen des Nutzers kann der Agent lernen und seine Aktionen noch optimaler an die Zielerfüllung anpassen. Gleichzeitig soll der Agent allerdings auch unabhängig vom Nutzer lernen, indem er überprüft, ob sich die Aktionen, welche er in der Vergangenheit ausgeführt hat, positiv auf die Umwelt ausgewirkt haben oder nicht.

- *Eingabe von Zielhierarchien*: Mit Hilfe dieser Hierarchie ist es dem Agenten möglich, seine Aktionen selbständig zu bewerten. Aus der Zielhierarchie ergeben sich direkt die Aktionen, welche der Agent auf bestimmte Umweltzustände ausführt (vgl. Kap. 2.4).

- *Ergebnisse bewerten*: Dieser Anwendungsfall ist relativ wichtig, da er die Lernfähigkeit des Agenten impliziert. Nur wenn die Ergebnisse der Aktionen des Agenten bewertet werden, kann er daraus Rückschlüsse ziehen, um seine Aktionen besser den Umweltbedingungen anzupassen.

- *Modifikationen der Zielhierarchie*: Mit Hilfe der Bewertungen ist es dem Agenten möglich, seine Zielhierarchie anzupassen. Diese Anpassungen können zum Beispiel über Änderungen in der Gewichtung erfolgen. Wie sie im Detail aussehen hängt von dem Algorithmus ab, welcher über dieser Datenstruktur operiert.

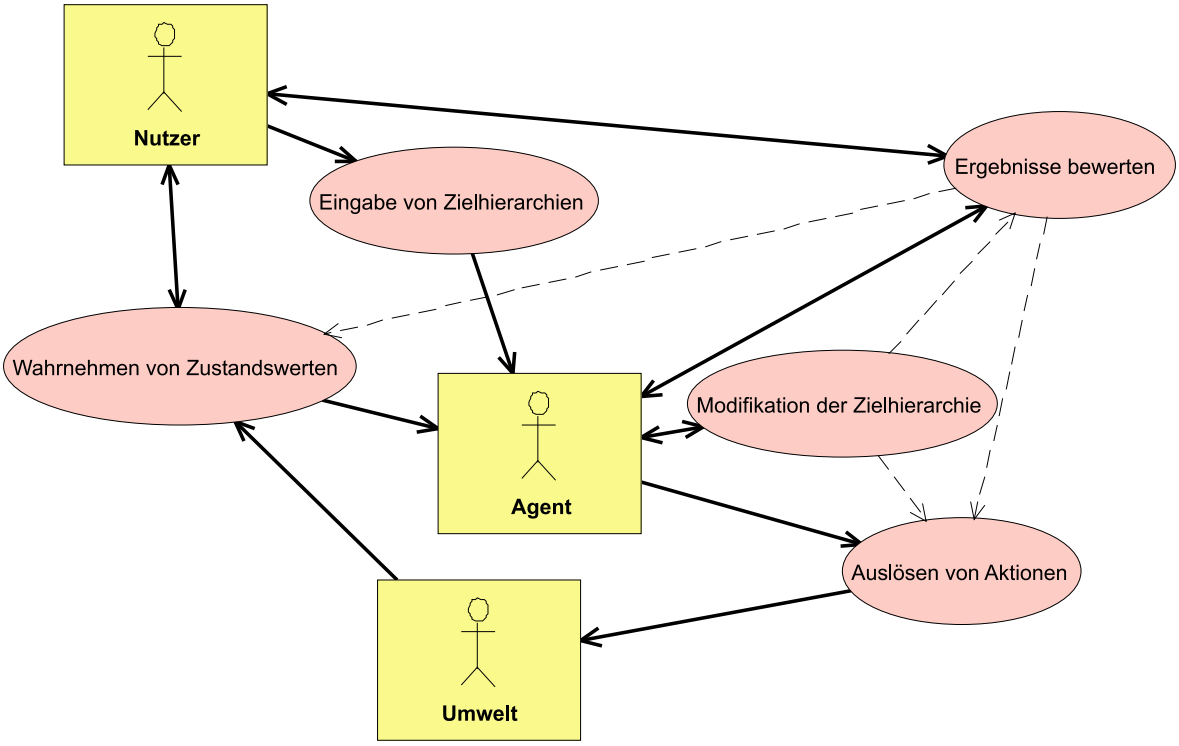

Bild 4: Nutzer - Agent - Umwelt

- *Auslösen von Aktionen*: Aktionen sind die einzige Möglichkeit des Agenten, auf seine Umwelt Einfluss zu nehmen. Im Prinzip sind diese Aktionen einfache Methodenaufrufe, um beispielsweise eine Heizung anzustellen.
- *Wahrnehmen von Zustandswerten*: Über Zustandswerte kann man die Umwelt des Agenten beschreiben. Diese Zustandswerte sind meistens direkt an ein Ziel gebunden und geben damit indirekt den Zielerfüllungsgrad dieses Zieles wieder.

Für die Zielabbildung, die Thema der Arbeit ist, wird ebenfalls ein objektorientierter Ansatz gewählt. Bevor man sich direkt mit der Zielabbildung beschäftigt, muss man sich darüber klar werden, in welcher Form man die Wissensabbildung realisiert werden soll. In dieser Arbeit wurde eine Struktur gewählt, welche der eines semantischen Netzes ähnlich ist, da sich damit nach Meinung des Autors die Strukturen der Umwelt besonders einfach darstellen lassen.

3.2 Grobdesign

Um eine möglichst flexible Darstellung der Wissensbasis erreichen zu können, wurde eine Ansatz gewählt, der sehr nahe an der Struktur angelehnt ist, mit der das menschliche Gehirn Informationen speichert. Die grundlegende Idee ist die folgende: alle Informationen werden durch sogenannte *Wissensobjekte* dargestellt. Diese Objekte sind durch sogenannte *Objektlinks* verbunden. Die semantischen Informationen können dadurch abgebildet werden, dass sowohl die *Wissensobjekte* als auch die *Objektlinks* typisiert sind. Jedes Wissensobjekt (und jeder Link) hat eine Priorität ($Prio(O)$), welche die Wichtigkeit des Objektes gegenüber anderen Objekten ausdrückt. Welche konkrete Interpretation die Priorität erfährt, ist selbstverständlich vom jeweiligen Objekt- oder Linktyp abhängig.

$$\text{Objekt} \rightarrow \text{Link} \rightarrow \text{Objekt}$$

Auf Basis dieser Grundstruktur ist es möglich, alle drei Wissensarten (Domänen-, Strategisches und kognitives Wissen) (vgl. Kapitel 2.4) elegant abzubilden. Beispielsweise würde eine „ist ein“ - Beziehung folgendermaßen abgebildet werden:

$$\text{Mensch} \rightarrow \text{ist ein} \rightarrow \text{Primat}$$

Domänenwissen (Strukturwissen) wird meist in Form solcher Hierarchien dargestellt. Die typisierten Links bieten aber auch die Möglichkeit, Regelwissen in der Form „Wenn <Bedingung> dann <Aktion>“ abzubilden:

$$\text{Wenn Bedingung} \rightarrow \text{dann} \rightarrow \text{Aktion}$$

$$\text{Wenn Bedingung} \rightarrow \text{sonst} \rightarrow \text{Aktion}$$

Der große Vorteil, der sich aus dieser Darstellung ergibt, ist, dass es aufgrund der Objektstruktur des Wissens leichter ist, einen Interpreter zu implementieren, als direkt einen Parser – für beispielsweise Prolog oder Lisp – zu schreiben.
Es ergibt sich somit ein Graph, welcher über gerichtete Kanten verfügt. Dadurch das die einzelnen Objekte als auch die Beziehungen(Links) typisiert sind, ist es möglich Plausibilätsprüfungen zu implementieren.

Nach Meinung des Autors existieren nur eine Handvoll dieser Linktypen, mit denen man die meisten Wissensstrukturen abbilden kann. So ist es zum Beispiel einleuchtend, dass man auf jeden Fall eine *„ist ein“-Beziehung* abbilden können muss. Ebenso ist eine *Aggregations-Beziehung* notwendig.
Der größte Vorteil dieser Art der Wissensabbildung ist jedoch der, dass die ganze Semantik eines Netzes in den Typinformationen sowohl der Objekte als auch der Links steckt. Dies ist auch der Unterschied zu reinen semantischen Netzen, bei denen die Links nur beschriftet sind. Innerhalb eines Agenten ist es dadurch auch möglich, Beziehungen zwischen zwei Onthologien herzustellen. Die Zielabbildung selbst ist innerhalb dieser Struktur abgebildet.

3.3 Klassenmodell

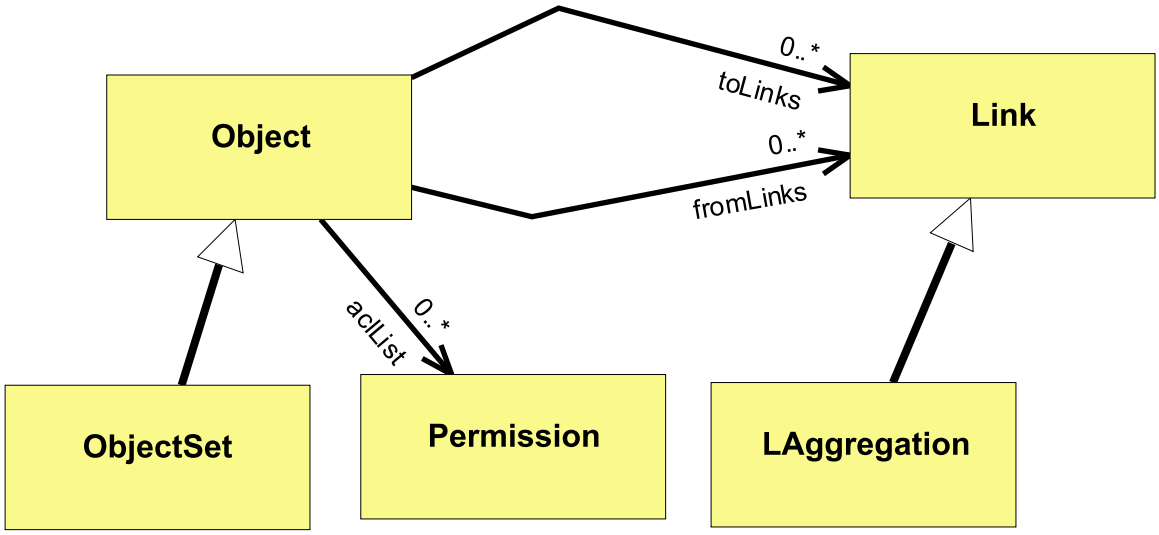

Bild 5: Grundstruktur der Wissensabbildung

Auf der Grundlage des Grobdesigns ergeben sich prinzipiell folgende Klassen (Bild 5):

- *Object*: Ein *Objekt* dient als „Lager“ eines Wissensobjektes. Dies kann zum Beispiel ein Text, eine Zahl oder eine beliebige andere Information sein. Jedes *Objekt*

besitzt außerdem eine Priorität, welche seine Wichtigkeit innerhalb des Wissensgraphen angibt. Die Interpretation dieser Priorität hängt vom jeweiligen Objekttypen ab. Die Priorität ist prinzipiell nach oben offen. So kann Wissensobjekt A eine Priorität von 1000 haben und Wissensobjekt B eine Priorität von 1.

- *Link*: Diese Klasse repräsentiert die Verlinkung zwischen verschiedenen Wissensobjekten. Dadurch ist eine hierarchische Gliederung der Wissensobjekte möglich. Durch die einzelnen Link-Typen (abgeleitete Klassen) ist es möglich, verschiedene Sichten zu realisieren. In dieser Klasse werden nur die Objektidentitäten gespeichert.

- *LAggregation*: Dieser Link bezeichnet eine *ist Teil von-Beziehung*. Beispielsweise besteht diese Arbeit aus einzelnen Kapiteln.

- *ObjectSet*: Ein *ObjectSet* stellt eine Menge von Wissensobjekten dar. Ein Objekt kann in mehreren Objektmengen enthalten sein. Diese Objektmengen sind entscheidend, um das Wissen des Agenten zu strukturieren. Alle Teile müssen über eine *Teil-Ganze-Beziehung* angebunden werden.

- *ObjectID*: Ein Objekt hat eine *ObjectID*, welche dazu dient, das Objekt eindeutig zu identifizieren.

- *Permission*: Da jeder Agent eindeutig identifizierbar ist, können ihm mit Hilfe der Klasse *Permission* entsprechende Zugriffrechte auf die Wissensobjekte zugestanden werden.

Der Autor ist überzeugt, dass sich mit Hilfe von entsprechenden Objekt- (Bild 10) und Linktypen (Bild 11) alle Wissensabbildungen leicht realisieren lassen.

3.3.1 Objekttypen der Zielabbildung

Für die Zielabbildung existieren auch entsprechende Klassen. Ein abstraktes Ziel wird durch die Klasse *OGoal* repräsentiert. Davon abgeleitet existiert eine Klasse *OState*, welche ein messbares Ziel, einen Zustand, darstellt. Über ein Flag wird festgelegt, ob das

Ziel oder der Status für die Gesamtbewertung relevant sind oder nur Teilziele sind. Eine einzelne Aktion (*OAction*) des Agenten setzt sich aus einer Zustandfolge (*OStateValue*) und einer Aktionsfolge (Folge von Methodenaufrufen (*OMethodCall*)) zusammen. Die Methoden sind durch die Klasse *OMethod* beschrieben.

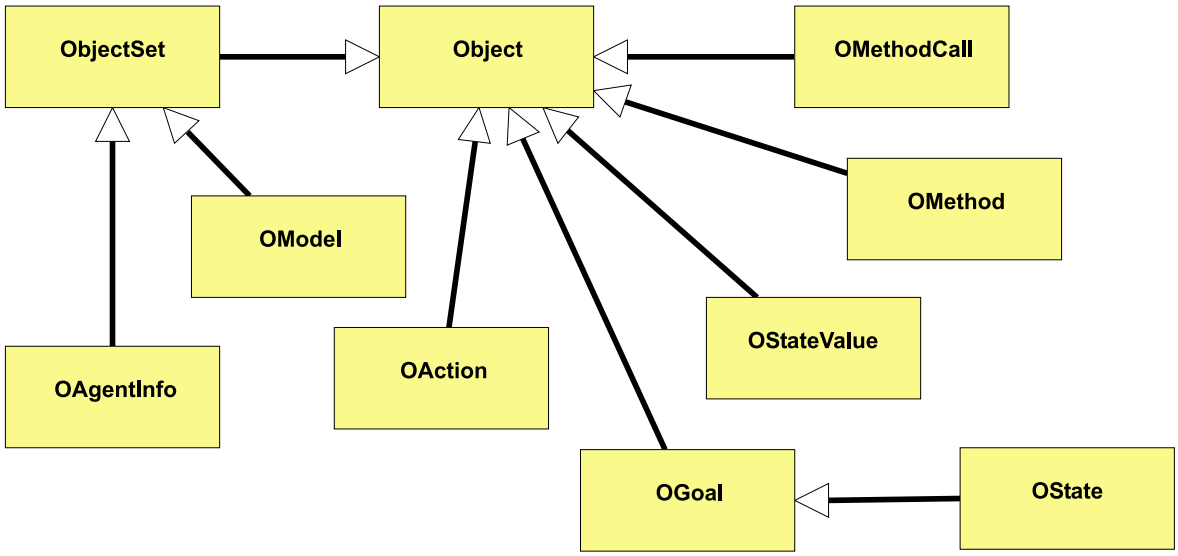

Bild 6: Objekttypen für Agenten

Es folgt eine Erläuterung der jeweiligen Objekttypen (Bild 6):

- *OModel*: Diese Klasse dient dazu, das Modell, welches der Agent von der Umwelt hat, zu strukturieren. Da es im Prinzip auch nur eine Objektmenge ist, ist es von dieser Klasse abgeleitet.

- *OGoal*: Ein Ziel ist für den Agenten etwas ganz Entscheidendes. Durch dieses Ziel ist es erst möglich, das Handeln (die Methodenaufrufe) des Agenten zu bewerten. Mit Hilfe eines Flags (useForBewertung) werden die Ziele markiert, welche für die Zielbewertung relevant sind. Dieses Flag ist eine Tribut an den Algorithmus, welcher in Kapitel 3.4 entwickelt wird.

- *OState*: Ein Zustand ist ein messbares Ziel. Er kann in der Umwelt verschiedenste Ausprägungen haben, jedoch ist für jeden Zustand eine Zugehörigkeit zwischen

null und eins bestimmbar. Wenn der Wert null ist, so ist der Zustand schlecht für die Zielerfüllung. Ist er eins, so ist die Zielerfüllung sehr gut. Jeder einzelne Zustand muss eine Funktion *double getValidation(CStateValue v)* bereitstellen, mit der es möglich ist, die Zielerfüllung dieses Zustandes als Wert zwischen null und eins auszudrücken. Ferner muß noch eine Funktion *double getAehnlichkeit(CStateValue v1, CStateValue v2)* existieren, welche die Ähnlichkeit zwischen zwei Zustandwerten als Wert zwischen null und eins zurückliefert. Bei einem Wert von eins sind die beiden Werte identisch.

- *OStateValue*: Ein Zustandswert ist die Realisierung eines Zustandes.

- *OMethod*: Diese Klasse beschreibt eine Methode eines Agenten. Nur mit Hilfe von Methoden können Aktionen ausgeführt werden. Wie in der Klasse *OState* existiert auch hier eine Funktion *double getAehnlichkeit(CMethodCall c1, CMethodCall c2),* welche die Ähnlichkeit zwischen zwei Methodenaufrufen berechnen kann.

- *OAgentInfo*: Alle Informationen, die der Agent über einen anderen Agenten hat, sind über die Objektmenge, die diese Klasse bereitstellt, zu erreichen. Dazu zählen unter anderem auch die Methoden dieses Agenten.

- *OMethodCall*: Diese Klasse repräsentiert einen Methodenaufruf mit den entsprechenden Parameterwerten. Sie definiert eine Funktion $time(t_1, t_2)$, welche angibt, inwieweit der Methodenaufruf, wenn er in t_1 ausgeführt wurde, Auswirkungen auf den Zeitpunkt t_2 hat. Eine Temperaturerhöhung tritt zum Beispiel auch erst nach einer geraumen Weile nach dem Heizbeginn ein.

$$\{time(t_1, t_2) \in 0..1\}$$

- *OAction*: Diese Klasse fasst einen Methodenaufruf mit seinen Vorbedingungen (Zustandswerten) zusammen. In dieser Klasse wird auch die Vor- oder Nachteilhaftigkeit des Methodenaufrufs gespeichert.

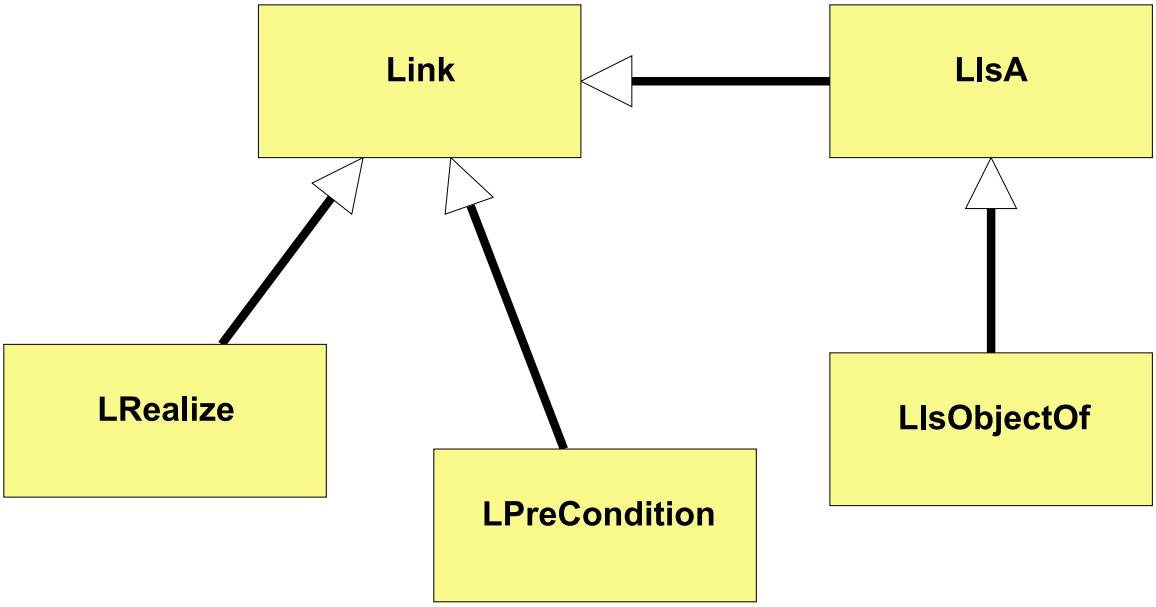

Bild 7: Linktypen für Agenten

3.3.2 Linktypen der Zielabbildung

Im weiteren folgt eine nähere Erläuterung der Linktypen (Bild 7):

- *LRealize*: Dieser Link verbindet ein Ziel mit einem Ziel sowie einen Zustand mit einem Ziel. Durch diesen Link wird auch die Elastizitätsfunktion zwischen Zielen abgebildet.
- *LPreCondition*: Durch diesen Linktypen werden die Vorbedingungen für einen Methodenaufruf (*OMethodCall*) innerhalb einer Aktion (*OAction*) definiert. Dieser Link darf von der Klasse *OAction* auf verschiedene Zustände verweisen.
- *LIsA*: Die Beziehung „ist ein“ drückt, wie bereits erwähnt, eine Beziehung zwischen Klassen aus.
- *LIsObjectOf*: Diese Beziehung ist eine Verstärkung der „ist ein“-Beziehung. Hier wird sie benötigt, um einen Zustandswert an seinen Zustand und einen Methodenaufruf an seine Methode zu binden.

Die in Kapitel 2.4 eingeführte Zielhierarchie wird innerhalb dieser Datenstruktur auf

einen Graphen erweitert. Dies hat nach Meinung des Autors den Vorteil, dass die Datenabbildung keiner Einschränkung unterliegt. Die jeweiligen Algorithmen, die über dieser Datenstruktur operieren können und müssen selbstverständlich diese Einschränkungen treffen.

3.4 Algorithmus

Dieser Abschnitt wird sich der Entwicklung eines beispielhaften Algorithmus widmen, mit dem es möglich ist, einen lernenden und adaptiven Agenten zu entwickeln, welcher auf Basis dieser Ziel- und Wissensabbildung Entscheidungen treffen kann. Sämtliche mathematische Strukturen, welche im folgenden verwendet werden, können innerhalb des Wissensgraphen abgebildet werden. Sie stellen dabei auch die entsprechende Einschränkungen des Graphen dar.
Gegeben sind dem Agenten für einen Algorithmuslauf die folgenden Parameter:

$$s := \{s_0, s_1 \dots s_k\} \quad \text{– Ausprägungen der Umweltzustände}$$

$$a := \{a_0, a_1 \dots a_m\} \quad \text{– Menge der möglichen Aktionen}$$

Die Menge der Umweltzustände S wird durch Objekte des Typs *OStateValue* abgebildet. Die Menge der Aktionen A enthält alle denkbaren Methodenaufrufe, welche durch Objekte der Klasse *CMethodCall* abgebildet werden können (vgl. Kapitel 3.3).
Gesucht ist eine Aktion, welche die Menge S so verändert, dass die Zielbewertung des Gesamtsystems maximal wird. Um dies zu erreichen, werden die Erfolge der einzelnen Aktionen gespeichert. Hierzu wurde im Kapitel 3.3 der Objekttyp OAction eingeführt.

Der Algorithmus wird, wie folgender Pseudocode zeigt, in mehrere Phasen zerlegt:

```
ObjectSet lerne(ObjectSet states; ObjectSet actions) {
  double B = getZielBewertung(states);
  bewerteAktionen(states, actions, B, getTime);
  return waehleAktion(states);
}
```

Diese Methode führt einen Lernschritt durch und liefert die gewählte Aktion als Objektmenge zurück. Für die folgende mathematische Darstellung bleibt nur noch zu erwähnen, dass alle verwendeten Matrizen innerhalb des Datenmodells über die entsprechenden Objekte und Objektlinks abgebildet werden.

Zielbewertung

Gesucht ist die Gesamtbewertung des Zielsystems. Um diese zu ermitteln, berechnet man zunächst die fehlenden Zielerfüllungsgrade:

$$g := \{g_0, g_1 \dots g_n\}$$

Hierfür ist folgendes gegeben:

$s := \{s_0, s_1 \dots s_k\}$ – Ausprägungen der Umweltzustände

$s \subseteq g$ – Die Zustände sind eine echte Teilmenge der Ziele

Die Zielelastizitäten (Eg), welche durch die *LRealize*-Links (vgl. Kapitel 3.3) gegeben sind:

$$Eg := \begin{vmatrix} 1 & g_{10} & g_{20} & \cdots & g_{n0} \\ 0 & 1 & g_{21} & \cdots & g_{n1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & g_{nn-1} \\ 0 & 0 & 0 & \cdots & 1 \end{vmatrix}$$

$$\forall i \in 0(1)n, j \in 0(1)n : g_{ij} = f_i(g_i) \in 0 \ldots 1$$

Da nun $s \subseteq g$ gilt, kann man davon ausgehen, dass ein Teil der Zielerfüllungsgrade schon gegeben ist. Zur Berechnung kann demzufolge ein einfacher Gaußalgorithmus dienen. Das dazu benötigte Lineare Gleichungssystem sieht wie folgt aus:

$$\{\forall j = 1(1)m \; : \; g_j := \frac{\sum_{i=1}^{m} g_{ij}}{m}\}$$

Die Gesamtbewertung des Zielsystems (B) lässt sich nun mittels eines gewichteten arithmetischen Mittels der einzelnen Zielerfüllungsgrade berechnen. Für die Gewichtung wird dabei die prozentuale Priorität (vgl. Klasse *OGoal*) verwendet. Hierbei ist jedoch zu beachten, dass nur die Ziele verwendet werden, in denen das Flag *useForBewertung* gesetzt ist.

$$B := \frac{g_i.priority}{\sum g_i.priority} * g_i$$

$$\forall i = 1(1)m; g_i.useForBewertung == true$$

Aktionenbewertung

Für die Bewertung der Aktionen muss man die soeben gewonnene Bewertung des Zielsystems auf die einzelnen Aktionen herunterbrechen. Damit dies möglichst realitätsnah geschehen kann, müssen folgende Informationen gegeben sein:

Eine Matrix S, welche die Ähnlichkeit der Zustände beschreibt (vgl. Klasse *OState* und *OStateValue*):

$$S := \begin{vmatrix} 1 & s^*_{10} & s^*_{20} & \cdots & s^*_{n0} \\ 0 & 1 & s^*_{21} & \cdots & s^*_{n1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & s^*_{nn-1} \\ 0 & 0 & 0 & \cdots & 1 \end{vmatrix}$$

$$\left\{\forall i \in 0(1)n; j \in 0(1)n : s^*_{ij} \in 0..1\right\}$$

Eine Matrix A, welche die Ähnlichkeit der Aktionen beschreibt (vgl. Klasse *OMethod* und *OMethodCall*):

$$A := \begin{vmatrix} 1 & a_{10} & a_{20} & \cdots & a_{m0} \\ 0 & 1 & a_{21} & \cdots & a_{m1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{mm-1} \\ 0 & 0 & 0 & \cdots & 1 \end{vmatrix}$$

$$\left\{\forall i \in 0(1)m; j \in 0(1)m : a_{ij} \in 0..1\right\}$$

Eine Matrix T, welche die letzten Zeiten einer Aktion-Zustand-Kombination speichert(vgl. Klasse *OAction*).

$$T := \begin{vmatrix} t_{00} & t_{10} & \cdots & t_{m0} \\ t_{01} & t_{11} & \cdots & t_{m1} \\ \vdots & \vdots & \ddots & \vdots \\ t_{0n} & t_{1n} & \cdots & t_{mn} \end{vmatrix}$$

Gesucht sind nun die Einzelbewertungen der Aktion-Status-Kombination, welche über die Entscheidungtabelle (ETA) abgebildet werden können (vgl. Klasse *OAction*):

	a_0	a_1	...	a_m
s_0				
s_1				
$\vdots$				
s_n				

Mit Hilfe dieser Werte lässt sich relativ genau beschreiben, wie sich die Aktionen auf die Umweltveränderungen auswirken. Zunächst multipliziert man die Gesamtbewertung mit dem Zugehörigkeitswert der Zeit; gleichzeitig wird die Ähnlichkeit der Zustandswerte sowie die Ähnlichkeit der Aktionen berücksichtigt.

$$\left\{ \forall i, r \in 0(1)m; \forall j, l \in 0(1)n : ETA_{ij} = ETA_{ij} + B \cdot \frac{a_i.time\left(t_{ij}, gettime()\right) + s_{jl} + a_{ir}}{3} \right\}$$

Aktionenauswahl

Mit Hilfe der Entscheidungstabelle muss nun eine Aktion gewählt werden. Da allerdings verhindert werden soll, dass sich der Agent „totlernt“, muss man eine gewisse „Mutationsrate“ respektive „Lernrate“ haben, damit der Agent aus einem lokalen Optimum „herausspringen“ kann (vgl. Kap. 2.5).

Gegeben ist in diesem Teil des Algorithmus' die Mutationsrate z ($z \in 0..1$), sowie die schon bekannte Tabelle *ETA*, als auch die Ähnlichkeitsmatrix S der Zustandswerte. Die Mutationsrate *z* wird, hierbei für zweierlei Zwecke verwendet. Zum einem gibt sie diejenige Ähnlichkeit von Zustandswerten an, bei der ein Vergleich mit den schon gespeicherten Zustandswerten keinen Sinn mehr ergibt. Zum anderen dient sie dazu, dass der Agent neue Aktionen ausprobiert.

Gesucht ist nun eine Aktion, welche den meisten Erfolg für den Agenten verspricht.

Zunächst wird anhand der Ähnlichkeit die „Ähnlichste“ schon bestehende Zustandsfolge herausgesucht. Wenn die Ähnlichkeit unter die Mutationsrate z fällt, so wird eine neue Zustandsfolge angelegt. Für diese Zustandsfolge wird nun mit Hilfe eines Zufallsgenerators und der Mutationsrate eine passende Aktion gewählt:

$$y = \left(\sum_{i=0}^{m} ETA_{ij} \right) + \left(\sum_{i=0}^{m} ETA_{ij} \right) \cdot z; \quad x = rand\,(y)$$

$$j = const;\ z \in 0..1;\ x \in 0..y$$

```
double s := 0;
for (int i = 0; i <= m; i++) {
  s = s + ETA[i,j];
  if s >= x then
    break;
}
return a[i]; // gefundene Aktion zurueckliefern
```

Wenn keine Aktion gefunden wurde, so wird eine neue Aktion zufällig angelegt.

Nutzereingabe und Komplexität

Ein hauptsächliches Problem liegt darin, dem Agenten alle Informationen zur Verfügung zu stellen. Eine Eingabe der Ähnlichkeiten aller Zustände sowie der Aktionen ist dem Nutzer in den wenigsten Fällen zumutbar. Dies kann man jedoch dadurch umgehen, dass die Ähnlichkeiten, wie in diesem Beispiel hart implementiert werden. Ähnlichkeiten zwischen zwei Methodenaufrufe (derselben Methode) können beispielsweise berechnet werden, indem man die Wertunterschiede der Parameter heranzieht. So kann zum Beispiel eine Methode, die das Thermostat einer Heizung regelt mit verschiedenen Parameterwerten aufgerufen werden. Eine Ähnlichkeit zwischen den Stufen 1 und 2 lässt sich hier durchaus berechnen.

Ein weiteres Problem sind die Laufzeiten der Algorithmen. In dem Beispielalgorithmus ist die Herunterbrechung der Gesamtbewertung auf eine Aktionenbewertung der kritische

Teil. Die Komplexität liegt hier im ungünstigsten Fall bei $(n+m)^2$. Wie stark jedoch der Anstieg dieser Kurve ist, hängt hauptsächlich davon ab, inwieweit die einzelnen Objekte miteinander verbunden sind. Je komplexer das Umweltmodell ist, umso kritischer wird das Laufzeitverhalten. Dadurch, dass im eigentlichen Quelltext anhand der Objektlinks, und nicht mit Hilfe von Matrizen, navigiert wird, dürfte das Laufzeitverhalten im Mittel sogar fast linear sein.

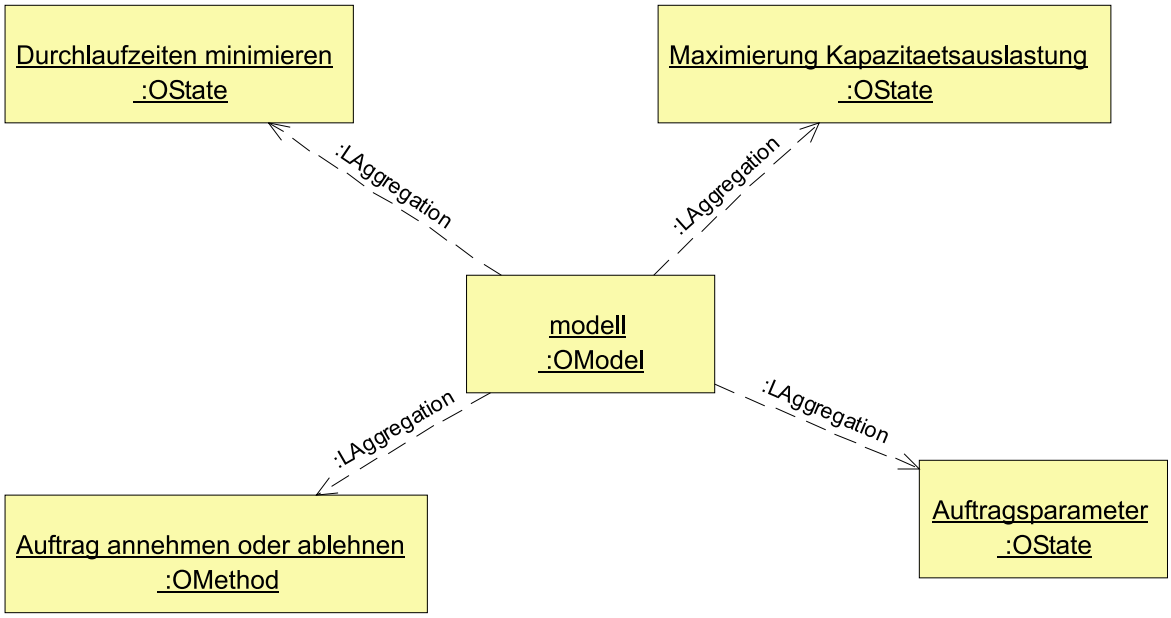

Bild 8: Zielsystem Fertigungssteuerung

4 Beispiele

4.1 Fertigungssteuerung

Aufbauend auf der entwickelten Ziel- und Wissensabbildung (vgl. Kapitel 3.3) soll ein kleines Beispiel für eine Fertigungssteuerung entwickelt werden. Die Möglichkeiten des Agenten sind sehr beschränkt (durch das Zielsystem). Er ist nur in der Lage, einen neuen Auftrag anzunehmen bzw. abzulehnen. Gleichzeitig versucht er, mit dieser Aktion sein Zielsystem bestmöglich zu erfüllen.

Als ein betriebswirtschaftliches Formalziel der Fertigung kann man die Verringerung der Durchlaufzeiten annehmen. In diesem Fall steht jedoch ebenfalls noch das Ziel einer optimalen Kapazitätsauslastung an. Wie man bereits sieht, sind beides konkurrierende Ziele. Wenn die Kapazitätsauslastung maximiert wird, so steigen die Durchlaufzeiten (Bild 8).

Für dieses Beispiel wird davon ausgegangen, dass die Verminderung der Durchlaufzeiten doppelt so wichtig ist wie das Ziel der maximalen Kapazitätsauslastung. Ferner wird noch

angenommen, dass die Auftragsparameter entsprechend gesetzt sind.

$$Prio(DLZ) = 2$$

$$Prio(Kap) = 1$$

Diese beiden Zustände werden auch zur Berechnung des Zielerfüllungsgrades verwendet(Flag *useForBewertung* ist gesetzt). Für den ersten Durchlauf des Algorithmus werden folgende Erfüllungsgrade angenommen:

$$g_{DLZ} = \frac{3}{5}$$

$$g_{Kap} = \frac{2}{5}$$

Der Erfüllungsgrad des Gesamtsystems ergibt sich nun folgendermaßen.

$$B = \frac{\frac{2}{2+1} \cdot \frac{3}{5} + \frac{1}{2+1} \cdot \frac{2}{5}}{2} = 0,2\overline{6}$$

Die Aktionsbewertung würde jetzt selbstverständlich ins Leere laufen, da es bisher noch keine Aktionen bis dahin gegeben hat (sie wird demzufolge nicht ausgeführt). Aus diesem Grund wählt der Agent eine Aktion zufällig aus. In diesem Beispiel wäre es die Aktion „Auftrag annehmen“.
Beim zweiten Durchlauf des Algorithmus' ermittelt er den Zielerfüllungsgrad neu(vgl. Funktion *getValidation* der Klasse *OState*) und verteilt die Bewertung auf die jeweiligen Methodenaufrufe.

$$g_{DLZ} = \frac{4}{5}$$

$$g_{Kap} = \frac{2}{5}$$

$$B = \frac{\frac{2}{2+1} \cdot \frac{4}{5} + \frac{1}{2+1} \cdot \frac{2}{5}}{2} = 0,\overline{3}$$

Es existiert diesmal schon eine Aktion-Zustands-Kombination mit dem Zustand $s_0 = \{g_{DLZ} = \frac{3}{5}, g_{Kap} = \frac{2}{5}\}$ und der Aktion $a_0 = \{auftragAnnehmen(Param)\}$ ($Param = \{Zeit = 5h, Wert = 10000EUR\}$).

Die Gewichtung der Methode ergibt sich nun wie folgt:

$$ETA_{00} = ETA_{00} + B \cdot \frac{a_i.time(t_{ij}, gettime()) + s_{00} + a_{0,0}}{3} = 0 + 0,\overline{3} \cdot \frac{0,7+1+1}{3} = 0,3$$

Die Zeitzugehörigkeit ist in diesem Fall willkürlich gewählt (0,7). Sie würde sich jedoch aus den Auftragsparametern berechnen lassen.
Bei einem erneuten Auftreten dieser Zustandskombination oder einer ähnlichen Kombination wird sich der Agent demzufolge gleich verhalten.

4.2 Aquarium

In diesem Abschnitt wird mit Hilfe der Wissens- und Zielabbildungssprache ein Zielsystem modelliert, mit dem man ein Aquarium steuern kann. Mit Rücksicht auf die Verständlichkeit des Beispiels wird das Modell, welches von dem Aquarium gezeichnet wird, sehr einfach sein und wahrscheinlich nur in geringem Maße der Realität entsprechen.
Das Aquarium beinhaltet auf jeden Fall Wasser, bei dem es drei mögliche Stellgrößen gibt. Diese wären die *Wassertemperatur*, der *PH-Wert* sowie der *Nährstoffgehalt*. Als globale Oberziele existieren auf der einen Seite selbstverständlich das Ziel, dass es den Fischen gut gehen soll und auf der anderen Seite möglichst geringe Kosten des Aquariums. Die Zustandswerte, mit denen die Kosten gemessen werden, sind zum einen der Stromverbrauch (zum Heizen des Wassers) und zum anderen die Kosten für die Nahrung, welche ebenfalls an deren Verbrauch gemessen werden können. Auf Grund dieser Überlegungen kann man beispielsweise folgendes Modell für den Steueragenten aufbauen (Bild 5).

- *gut*: Dieses Ziel ist eines der Oberziele. Da es selbst nicht direkt messbar ist, muss es durch andere (Unter-) Ziele näher beschrieben werden.

- *billig*: Dieses Ziel ist das zweite große Ziel. Es steht im Prinzip mit dem ersten Ziel in Konflikt, da der Pflegeaufwand für die Fische Geld kostet. Obwohl es im Prinzip durchaus möglich wäre diese Ziel direkt zu messen, wurde entschieden es noch in weitere Unterziele aufzuteilen.

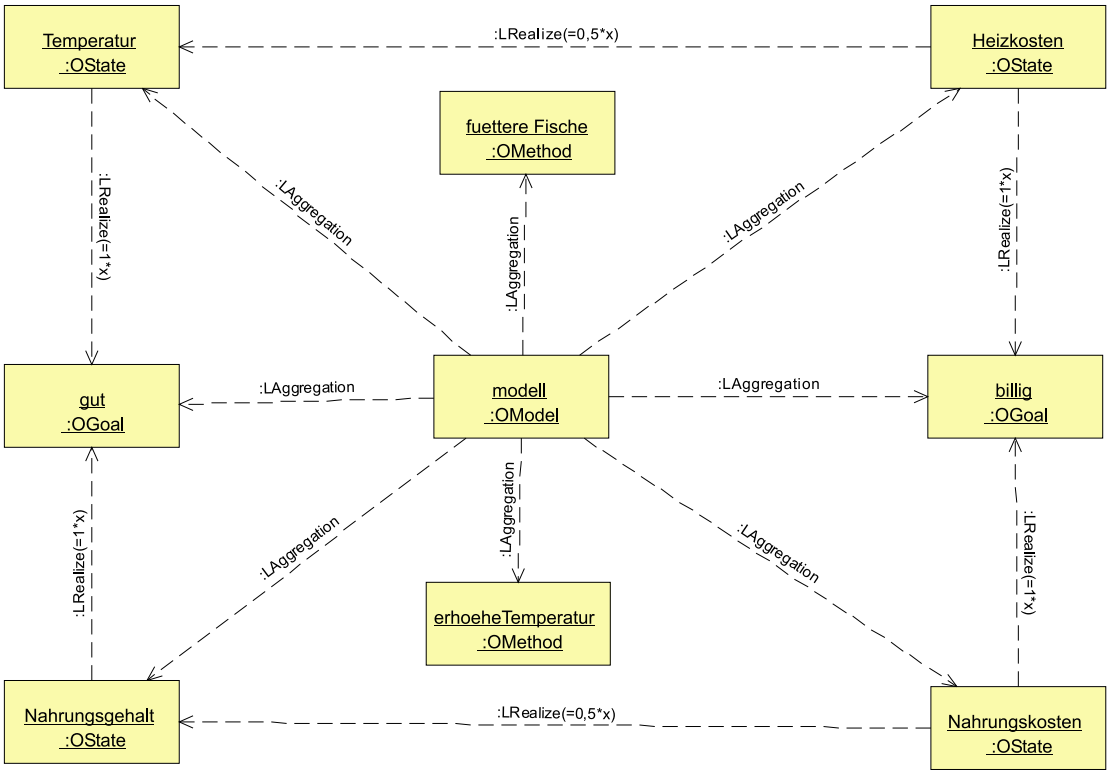

Bild 9: Zielsystem Aquarium

- *Temperatur*: Die Temperatur ist ein messbares Ziel. Innerhalb einer bestimmten Spanne ist die Temperatur für die Fische ideal (z. B. 20°C +/- 2K).

- *Heizkosten*: Die Heizkosten gehören zu denjenigen Zuständen, welche das Ziel *billig* messbar machen. Da diese Kosten im Prinzip immer steigen, muss man selbstverständlich dafür sorgen, dass diese Kosten immer in einer bestimmten Zeiteinheit (z. B. 1h) gemessen werden.

- *Nährstoffgehalt*: Der Nährstoffgehalt des Wassers gibt die Nahrungsmittelversorgung der Fische wider.

- *Nahrungskosten*: Ebenso wie die *Heizkosten* das Äquivalent zu der *Temperatur* sind, so sind die *Nahrungskosten* das Äquivalent zum *Nährstoffgehalt*.

- *PH-Wert*: Der *PH*-Wert sollte möglichst zwischen 5 und 7 liegen. Für den *PH*-

Wert wurde diesmal kein Kostenäquivalent angegeben, da diese Kosten gegen Null gehen. Außerdem wird dadurch deutlich, dass der Agent nur mit den Informationen arbeiten kann, die er besitzt. Wenn der Agent etwas falsch macht, so liegt es im Prinzip an mangelndem Wissen über die Umwelt.

Wie sich der Agent beim Lernen verhält, hängt hier natürlich in entscheidendem Maße von der Korrektheit der Beziehungen ab. Um das Modell möglichst wartbar zu halten, sollte man zu viele Beziehungen vermeiden. Sie machen das System unübersichtlich und fehleranfällig.

5 Schlussfolgerungen

Mit Hilfe des entwickelten Datenmodells ist es möglich, die Wissens- und Zielabbildung innerhalb eines Agenten und damit auch innerhalb eines Multiagentensystems abzubilden. Ferner ist die Anbindung an bestehende Standards prinzipiell möglich. Das Datenmodell, welches hier vorgestellt wurde, kann man als ein erweiterbares Rahmenwerk auffassen auf dem verschiedene Algorithmen operieren können.

Auf Grund der vielen möglichen *Links* zwischen den einzelnen *Wissensobjekten* kann die *Wissensbasis* eines Agentens sehr schnell sehr komplex werden. Hieraus können selbstverständlich *Debuggingprobleme* resultieren. Dies sind jedoch *prinzipielle Probleme* sowohl der KI als auch der VKI und keine Schwäche des Ansatzes.

So ist es im Prinzip zwar möglich, eine Wissensbasis aufzubauen, die dem Wissen eines Menschen entspricht. Im Moment existiert jedoch nicht genügend Rechenleistung, um solch komplexe Modelle berechnen zu können.

Es wurden einige potentielle Wege aufgezeigt um die Komplexität einschränken zu können und gleichzeitig die Wissensabbildung möglichst flexibel zu halten.

In den weiterfolgenden Schritten müsste dieses Datenmodell implementiert werden. Ferner müsste eine Liste mit allgemeingültigen Objekt- und Linktypen aufgestellt werden, welche die Semantik eines möglichst breiten Gebietes abdecken.

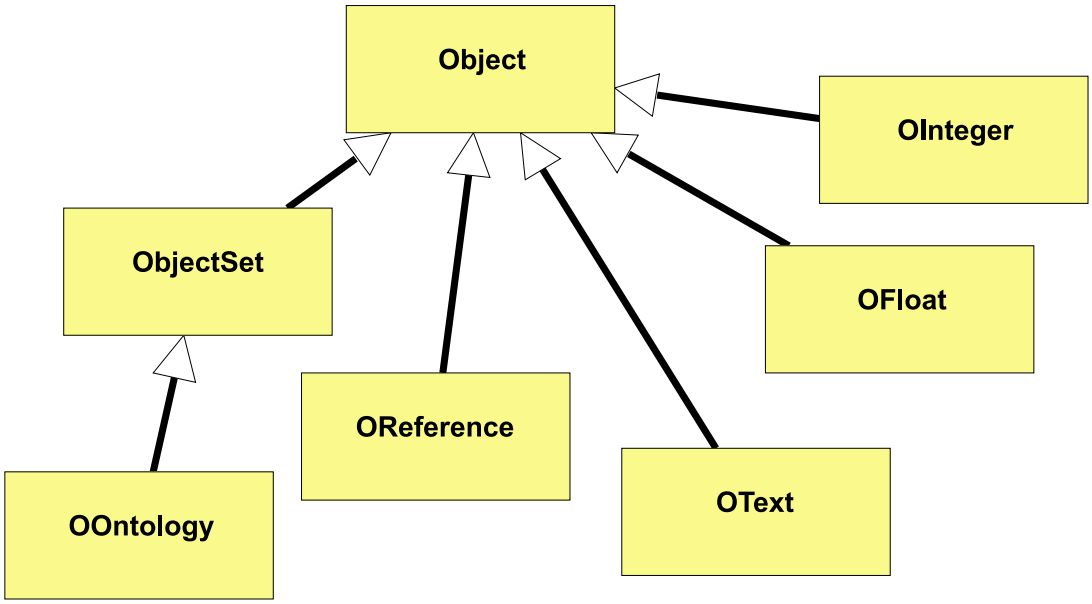

Bild 10: grundlegende Objekttypen

A Diagramme

In diesem Kapitel werden die Grundklassen, welche zur Wissensabbildung verwendet werden können, näher erläutert. Es soll dazu dienen, die Grundidee der Wissensabbildung näher zu verstehen.

- *Object*: Ein *Object* dient als „Lager“ eines Wissensobjektes. Dies kann zum Beispiel ein Text, eine Zahl oder eine beliebige andere Information sein.
- *Link*: Diese Klasse repräsentiert die Verlinkung zwischen verschiedenen Wissensobjekten. Dadurch ist eine hierarchische Gliederung der Wissensobjekte möglich. Durch die einzelnen Link-Typen (abgeleitete Klassen) ist es möglich, verschiedene Sichten zu realisieren. In dieser Klasse werden nur die Objektidentitäten gespeichert.
- *ObjectSet*: Ein *ObjectSet* stellt eine Menge von Wissensobjekten dar. Ein Objekt kann in mehreren Objektmengen enthalten sein. Diese Objektmengen sind entscheidend, um das Wissen des Agenten zu strukturieren.

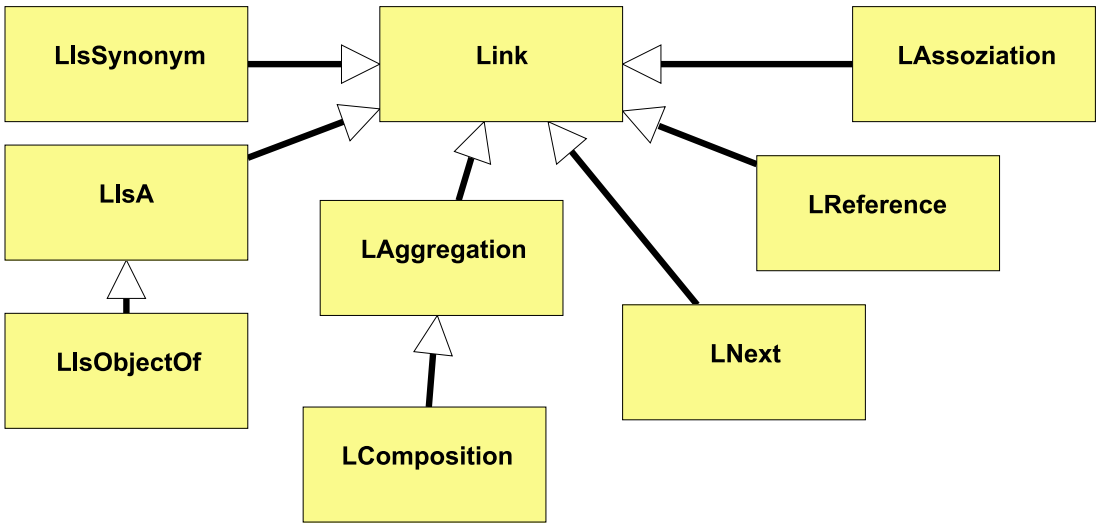

Bild 11: grundlegende Linktypen

- *OOntology*: Eine Ontologie repräsentiert ein Wissensgebiet. Mittels Ontologien werden diese Gebiete voneinander abgegrenzt.

- *Agent*: Ein Agent ist die zentrale Instanz des ganzen Systems. Er kommuniziert mit anderen Agenten über ein Protokoll, mit dem Anweisungen und Informationen an andere Agenten weitergegeben werden können.

  Technisch gesehen ist jeder Agent ein Servent, das heißt sowohl Client als auch Server

- *OText*: Diese Klasse wird verwendet um einfache Texte aufzunehmen. Diese dienen dann der textuellen Beschreibung von anderen Objekten.

- *OReference*: Objekte dieser Art referenzieren eigentlich auf anderen Objekte. Ein CReference-Objekt muss mittels eines LReference-Links das zu referenzierende Objekt angegeben haben.

- *LAggregation*: Die Aggregation drückt eine „Teil-Ganze“-Beziehung aus. Im Unterschied zur Komposition werden die Teilobjekte jedoch nicht gelöscht, wenn das

Oberobjekt gelöscht wird. Die Aggregation kann immer nur von einer Objektmenge her verlinken.

- *LAssoziation*: Eine Assoziation ist ein Link, welcher keine spezielle semantische Bedeutung hat. Sie drückt einfach nur eine Zusammengehörigkeit zwischen zwei Wissensobjekten aus.

- *LIsA*: Diese Klasse realisiert eine „ist ein“-Beziehung zwischen zwei Wissensobjekten. Mittels der Ist ein Beziehung lassen sich die Zugehörigkeiten von Objekten zu Klassen und von Klassen zu Klassen ausdrücken.

  Mensch $\rightarrow$ ist ein $\rightarrow$ Tier

  Tier $\rightarrow$ ist ein $\rightarrow$ Lebewesen

  Pflanze $\rightarrow$ ist ein $\rightarrow$ Lebewesen

- *LIsSynonym*: Diese Beziehung drückt aus, dass zwei Begriffe im Prinzip identisch sind. Dies ist relativ wichtig, da man mit Hilfe dieser Beziehung Dinge koppeln kann, welche von unterschiedlichen Personen verschieden genannt wurden, jedoch eigentlich dasselbe sind.

- *LComposition*: Die Komposition ist eine verstärkte Assoziation. Bei ihr werden die Teilobjekte gelöscht, wenn das Oberobjekt gelöscht wird.

- *LReference*: Dieser Linktyp stellt eine Verbindung zwischen einem CReferenz-Objekt und einem anderen Objekt her. Die einzige Bedeutung, die dieser Link hat, ist, dass in verschiedenen Objektmengen auch verschiedene Links gepflegt werden können.

- *LIsObjectOf*: Stellt eine Beziehung zwischen einem Modellobjekt und einer Ausprägung dieses Modellobjektes her.

  René $\rightarrow$ ist Objekt von $\rightarrow$ Mensch

- *LNext*: Dieser Linktype dient dazu, um Reihenfolgen auszudrücken. So können beispielsweise Lieder in einem Album sortiert werden.

Literatur

[Burk1999] *Burkhardt, Dr. Rainer*: UML - Unified Modelling Language : Objektorientierte Modellierung für die Praxis. Addison-Wesley, München, 1999.

[fipa00027] *Foundation for Intelligent Physical Agents*: FIPA Query Interaction Protocol Specification. http://www.fipa.org/specs/fipa00027/, 2001, Abruf am 2002-04-11.

[Frank1996] *Graesser, Art; Franklin, Stan*: Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents in Agent Theories, Architectures, and Languages(21-35). 1996.

[Hein1992] *Heinen, Dr. Dr. h.c. Edmund*: Grundlagen betriebswirtschaftlicher Entscheidung. Dr. Th. Gabler, Wiesbaden 1976.

[Inge1999] *Ingebrigtsen, Lars Magne*: Gnus Manual. http://www.gnus.org/manual.html, 1999, Abruf am 2002-05-10.

[Kay1984] *Kay, Alan F.*: Computer Software in Scientific American(41-47). Sep 1984.

[Kirn2001] *Kirn, Prof. Dr. Stefan*: Multiagentensysteme. Addison-Wesley, München, 2001.

[Klue2001] *Klügl, Franziska*: Multiagentensimulation. Addison-Wesley, München, 2001.

[kqml2002] *Weber, Jay; Finin, Tim*: DRAFT Specification of the KQML. http://www.cs.umbc.edu/kqml/kqmlspec.ps, 1993, Abruf am 2002-04-10.

[OMG2001] *Group, Object Managment*: OMG Unified Modelling Language Specification.

http://www.omg.org/technology/documents/formal/uml.htm, 2002, Abruf am 2002-07-20.

[Sutt1998] *Barto, Andrew G.; Sutton, Richardt S.*: Reinforcement Learning: An Introduction. MIT Press, Cambridge, Massachusetts 1998.

[Wool1995] *Jennings, Nicholas R.; Wooldridge, Michael*: Intelligent Agents: Theory and Practice. http://citeseer.nj.nec.com/article/wooldridge95intelligent.html, 1995, Abruf am 2002-06-24.

Abkürzungsverzeichnis

A	Ähnlichkeitsmatrix der Aktionen
a	Menge der Aktionen
a_i	Eine einzelne Aktion
B	Gesamtbewertung des Zielsystems
E_g	Matrix der Zielelastizitäten
ETA	Entscheidungstabelle
$FIPA$	Foundation for Intelligent Physical Agents
g	Menge der Zieleerfüllungsgrade
KI	Künstliche Intelligenz
$KQML$	Knowledge Query and Manipulation Language
MAS	Multiagentensysteme
MIT	Massachusetts Institute of Technology
OTW	Objekt Technologie Werkbank
QIP	FIPA Query Interaction Protocol
S	Ähnlichkeitsmatrix der Zustände
s	Menge der Zustände
s_i	Ein einzelner Zustandswert
T	Matrix, welche die letzen Zeiten einer Aktion-Zustand-Kombination speicher
UML	Unified Modelling Language - Vereinheitlichte Modellierungssprache
VKI	Verteilte künstliche Intelligenz

Abbildungsverzeichnis

1 Agent - Umwelt 5
2 Zielbeziehungen [Hein1992] 8
3 Agentenarchitekturen 10
4 Nutzer - Agent - Umwelt 15
5 Grundstruktur der Wissensabbildung 17
6 Objekttypen für Agenten 19
7 Linktypen für Agenten 21
8 Zielsystem Fertigungssteuerung 29
9 Zielsystem Aquarium 32
10 grundlegende Objekttypen 35
11 grundlegende Linktypen 36