Thomas Weiser

Ein Berechnungsmodell für situatives Agieren

Institut für Informatik
der Technischen Universität München

Lehrstuhl für Echtzeitsysteme und Robotik

Ein Berechnungsmodell für situatives Agieren

Thomas Weiser

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. J. Schlichter
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. H.-J. Siegert
	2.	Univ.-Prof. Dr. Dr. h.c. W. Brauer

Die Dissertation wurde am 11. Juli 2000 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 18. Oktober 2000 angenommen.

Diese Arbeit entstand während meines Aufenthaltes am Deutschen Forschungsinstitut für Künstliche Intelligenz in Saarbrücken sowie während meiner Tätigkeit als wissenschaftlicher Angestellter am Institut für Informatik der Technischen Universität München.

An erster Stelle möchte ich mich bei meinem Doktorvater Prof. Dr. Hans-Jürgen Siegert für die wissenschaftliche Betreuung und seine konstruktive Unterstützung bedanken. Er hat mich in zahlreichen Diskussionen dazu ermutigt, meine Ideen frühzeitig zu konkretisieren. Prof. Dr. Dr. h.c. Wilfried Brauer danke ich für seine Anmerkungen und die Übernahme des Zweitgutachtens.

Mein besonderer Dank geht an Dr. Klaus Fischer, der meine wissenschaftliche Arbeit auf den Weg gebracht und mich auf vielfältige Weise gefördert hat.

Ich bedanke mich bei Dr. Gerhard Schrott für seine stete Unterstützung, bei Andrea Baumann für ihre Freundschaft und ihre Hilfe in der Endphase der Arbeit sowie bei Anke Unger und Peter Perez für ihren Beitrag zur Implementierung des Sita-Systems. Nicht zuletzt möchte ich mich bei meinen Kollegen der vergangenen Jahre bedanken, für ihr Interesse, für viele Diskussionen sowie für die freundschaftliche Atmosphäre: Dr. Boris Baginski, Dr. Martin Buchheit, Florian Fuchs, Dr. Stefan Hahndel, Dr. Andreas Koller, Dr. Jörg Müller, Markus Pischel, Oliver Schmid, Dr. Peter Stöhr und Susanne Stöhr.

Kurzfassung

Die Arbeit entwickelt ein neues Modell zur Programmierung flexibler Agenten basierend auf dem Prinzip des situativen Agierens. Demnach konstituiert sich reaktives Verhalten aus den beiden Vorgängen Erkennen und Handeln. Üblicherweise wird eine Form von Bedingungs-Aktions-Regeln verwendet, um reaktives Verhalten programmiersprachlich zu beschreiben. Problematisch hierbei ist die oft geringe Ausdrucksstärke sowohl der Bedingungs- als auch der Aktionssprache, sowie der Mangel an Strukturierungsmitteln, die über das Regelkonstrukt hinausgehen.

Das vorgestellte Berechnungsmodell Sita betrachtet das Erkennen und das Handeln als zwei verschiedenartige und zunächst getrennte Berechnungsmodi, deren Spezifikation nach zwei unterschiedlichen Beschreibungsformalismen verlangt. Die Situationserkennung bedient sich einer Hornklausellogik, die es ermöglicht, komplexe Situationsklassen auf deklarative Weise zu beschreiben. Die Handlungen des Agenten werden dagegen auf imperative Weise in einem nebenläufigen Prozeßkalkül beschrieben. Durch das Konstrukt der überwachten Auswahl können Prozesse auf das Eintreffen bestimmter Situationen warten und ihr weiteres Vorgehen von der Art der aufgetretenen Ereignisse abhängig machen. Umgekehrt können Prozesse ihrerseits neue Ereignisse generieren und so das Wissen um die aktuelle Situation modifizieren.

Die verwendete Hornklausellogik schließt Funktionssymbole, Negation und Rekursion als Sprachmittel mit ein, und geht damit in ihrer Ausdrucksstärke deutlich über die Bedingungssprachen der aus der Literatur bekannten regelbasierten Systeme hinaus. Das Logikprogramm wird durch ein inkrementelles, vorwärtsverkettendes und datengetriebenes Verfahren ausgewertet, den IDC-Algorithmus (*Incremental Deductive Closure*). Dieser bezieht seine Effizienz aus einer Reihe neu entwickelter Optimierungstechniken.

INHALTSVERZEICHNIS

1 Einleitung **1**
1.1 Motivation und Zielsetzung 1
1.2 Ansatz von Sita 3
1.3 Beiträge der Arbeit 4
1.4 Überblick 4

2 Stand der Technik **7**
2.1 Einleitung 7
2.2 Agentenarchitekturen 8
2.2.1 Logikbasierte Architekturen 9
2.2.2 Rein reaktive Architekturen 10
2.2.3 BDI-Architekturen 10
2.2.4 Hybride Architekturen 12
2.3 Programmierung situativen Verhaltens 13
2.3.1 Bedingungs-Aktions-Regeln: Magsy 13
2.3.2 Ausführbare Temporallogik: Cuncurrent METATEM 14
2.3.3 Prozedurales Reasoning: AgentSpeak(L) 16
2.4 Zusammenfassung 20

3 Das Sita-Berechnungsmodell **21**
3.1 Erkennen und Handeln 21
3.2 Basisarchitektur von Sita 22
3.3 Sita als Programmiersprache 25

3.3.1 Logikprogrammierung 25
3.3.2 Koordination 26
3.3.3 Verwandte Programmiermodelle 27
3.4 Sita als Agentenmodell 29

4 Wissensbasis **31**
4.1 Syntax 31
4.2 Semantik 35
4.2.1 Programmklassen 35
4.2.2 Semantiken für stratifizierbare Programme 39
4.2.3 Wohl-fundierte Semantik 41
4.3 Magic-Set Transformation 44

5 Handlungsbasis **47**
5.1 Syntax 47
5.2 Verwendung von Variablen 49
5.3 Semantik 50
5.3.1 Überwachte Auswahl 50
5.3.2 Koordinierter Wissensbasis-Zugriff 51
5.3.3 Thread-Scheduling 52
5.3.4 Builtin-Prozeduren 53
5.4 Erweiterungen 53

6 Auswertungsalgorithmen **57**
6.1 Differentielle Fixpunktiteration 57
6.2 Relationen-Netze 59
6.3 Netz-Konsistenz 62
6.4 Einfaches Ausgleichen 66
6.4.1 Unausgeglichene Zustände 69
6.4.2 Reduktion 70
6.5 Vermeidung von Zirkelschlüssen 76
6.5.1 Entstehen und Auflösen von Zyklen 76

6.5.2 Zwei-Phasen-Löschen 77
6.5.3 Selektives Löschen mit Höheninformationen 81
6.5.4 Vergleich der beiden Löschverfahren 83
6.6 Scheduling-Strategien 85
6.6.1 Horizontale Priorisierung 86
6.6.2 Vertikale Priorisierung 87
6.6.3 Experimentelle Bewertung 88
6.7 Weitere Optimierungen 89
6.7.1 Join-Order-Optimierung 90
6.7.2 Aggregate 90
6.8 Nicht-stratifizierbare Programme 91
6.8.1 Modulare Stratifikation 91
6.8.2 Normale Programme 94
6.9 Verwandte Arbeiten 94

7 Anwendungsbeispiel 97

7.1 Szenario: Reaktive Wegeplanung 97
7.2 Realisierung in Magsy 98
7.3 Realisierung in AgentSpeak(L) 100
7.4 Realisierung in Sita 101

8 Schlußbemerkungen 105

8.1 Zusammenfassung 105
8.2 Ausblick 106
8.2.1 Weitergehende Strukturierungsmittel 107
8.2.2 Updates abgeleiteter Prädikate 108

A Ergebnisse der Performanz-Messungen 111

A.1 Löschverfahren 111
A.2 Scheduling-Strategie 112

Literaturverzeichnis 113

KAPITEL 1

Einleitung

In diesem Kapitel wird der Entwurf des Berechnungsmodells Sita *als Weiterentwicklung regelbasierter Sprachen motiviert. Der Ansatz des Modells wird eingeführt, die wesentlichen Beiträge der Arbeit werden genannt und ein Überblick über die Arbeit wird gegeben.*

1.1 Motivation und Zielsetzung

Der Agent ist das ausgewiesene Modellierungskonzept für flexibles, autonomes Handeln in dynamischen Umgebungen. Der Begriff faßt Anforderungen zusammen, die zunehmend an Anwendungen der Informationstechnologien gestellt werden: Die Systeme müssen flexibel, reaktiv, kooperativ, verteilt, „intelligent“ sein, Anforderungen, die nach entsprechenden Softwaretechnologien verlangen. Ein wichtiger Faktor hierbei ist die programmiersprachliche Ausstattung der Softwareentwicklung.

Da Agentensysteme ein noch relativ junges Gebiet sind, gibt es bisher kaum allgemein anerkannte Methoden zur Entwicklung entsprechender Software. Die Entwicklung hat daher oft experimentellen Charakter. Wir sehen hier einen Bedarf nach Programmiersprachen, die den Entwurf und die Realisierung von prototypisch konzipierten und experimentell ausgerichteten Systeme unterstützen.

Betrachtet man in dieser Hinsicht heutige Ansätze, Agentenanwendungen zu implementieren, so sind im wesentlichen drei Gruppen von Lösungen auszumachen: Da ist zum ersten die Verwendung erprobter Allzwecksprachen, wie C++ oder Java. Ihre Allgemeinheit und Maschinennähe schließt allerdings jede Design-Unterstützung auf Sprachebene aus, so daß Unterstützung lediglich in Form von Bibliotheken oder Komponenten etwa für Kommunikationsschnittstellen zu erwarten ist.

Als zweite Möglichkeit stehen symbolische Hochsprachen zur Verfügung, wie etwa die klassische „KI“-Sprache Prolog oder nebenläufige Constraint-Sprachen wie Oz

[SHW95]. Diese legen ihren Schwerpunkt zumeist auf ein mathematisches, funktionales Berechnungskonzept und haben dadurch oft Schwierigkeiten mit Reaktivität oder beim Umgang mit veränderlichen Zuständen.

Zuletzt gibt es noch die Möglichkeit, speziell an das Agentenkonzept angepaßte Programmiersprachen einzusetzen oder neu zu entwickeln. Hier geht man entweder von einer ausgewählten Agentenarchitektur oder einer besonders geeigneten Spezifikationsmethodik (z.B. Modallogiken) aus. Daraufhin sind folgende Fragen zu klären: Wie kann der Zustand eines Agenten strukturiert und repräsentiert werden? Mit welchen sprachlichen Mitteln können Verhaltensweisen spezifiziert werden? Welche Inferenzmechanismen sind zur Interpretation notwendig?

Beispiele spezieller Agentensprachen sind Agent0 [Sho93], PLACA [Tho95], Concurrent METATEM [Fis93c] und AgentSpeak(L) [Rao96]. Diese Sprachen, von denen einige im zweiten Kapitel vorgestellt werden, sind zumeist regelorientiert: Wenn der Agent in einer bestimmten Situation ist, soll er in einer bestimmten Weise darauf reagieren. Die Sprachen unterscheiden sich darin, wie die aktuelle Situation repräsentiert wird, wie die reaktionswürdigen Situationen beschrieben werden und wie schließlich die Reaktionen selbst spezifiziert werden.

Auch Produktionensysteme wie OPS5 [For81] und CLIPS [Cul89] verwenden Bedingungs-Aktions-Regeln. Obwohl ursprünglich zum Bau von Expertensystemen konzipiert, fanden sie dank ihrer reaktiven Arbeitsweise auch als Programmiersprache in Multiagentensystemen Anwendung. Ein Beispiel hierfür ist das regelbasierte Multiagentensystem Magsy [Fis93a], das für Planungs- und Steuerungsaufgaben in flexiblen Fertigungsumgebungen entwickelt wurde [Fis93b]. Die bei der Implementierung und den Anwendungen von Magsy gewonnenen Erfahrungen waren ein wichtiger Ausgangspunkt bei der Entwicklung des in dieser Dissertation vorgestellten Berechnungsmodells Sita.

Die Vorteile von Produktionensystemen liegen in ihrer Eignung, reaktive und nebenläufige Verhaltensweisen auf symbolischer Ebene zu beschreiben. Dies wird vor allem durch die vorwärtsverkettende Arbeitsweise des Regelinterpreters erreicht, die eine daten- und somit ereignisgetriebene Situationserkennung ermöglicht. Das Funktionsprinzip der Produktionensysteme liefert daher eine geeignete Grundlage für den Entwurf von Agentensprachen.

Auf der anderen Seite bringen Produktionensysteme eine Reihe von Problemen mit sich: Die einzige Kontrollstruktur in der Sprache ist die Produktion: Eine Konjunktion von Bedingungsmustern führt zu einer Sequenz atomarer Aktionen. Darüberhinaus gibt es keine Abstraktions- oder Strukturierungskonzepte. Es gibt insbesondere keine Möglichkeit, komplexere Aktionen als Prozeduren oder komplexere Bedingungen als Situationsklassen zu definieren. Dies führt insgesamt zu großen, unstrukturierten Regelprogrammen, die mühsam zu erstellen und schwer zu warten sind.

Ausgangspunkt der vorliegenden Arbeit war es daher, auf der Basis der vorwärtsver-

kettenden, ereignisgetriebenen Regelprogrammierung ein Berechnungmodell zu entwickeln, das die genannten Schwierigkeiten mit Hilfe geeigneter Abstraktions- und Strukturierungskonzepte überwindet.

Es ergeben sich folgende Punkte als Ziele für die Entwicklung von Sita:

- Situationsklassen sollen deklarativ beschrieben werden. Es wird ein Konzept zur Definition abstrakter Situationsklassen benötigt.
- Der Formalismus zur Situationserkennung muß eine Balance zwischen Ausdrucksstärke und effizienter Berechenbarkeit finden.
- Der Formalismus soll einfach verständlich sein. Dazu ist insbesondere eine klare Semantik notwendig.
- Prozedurale Vorgänge sollen im prozeduralen Stil programmierbar sein.
- Das Modell soll nebenläufige Handlungsstränge unterstützen.
- Die aus den Produktionensystemen bekannte Reaktivität muß erhalten bleiben.
- Die Sprache ist als symbolische Hochsprache (etwa auf der Ebene von Prolog) auf das „Rapid-Prototyping“ entsprechender Agentenanwendungen ausgerichtet.
- Und schließlich sollte wie bei jeder Programmiersprache der Kern aus einer möglichst kleinen Menge orthogonaler Konstrukte aufgebaut sein.

1.2 Ansatz von Sita

Um sowohl die Situationssprache als auch die Aktionssprache in der geforderten Weise zu erweitern, werden beide zunächst aus dem engen Zusammenhang, der bei Produktionensystemen durch das Regelkonstrukt gegeben ist, herausgenommen. Das Erkennen und das Handeln des Agenten werden dazu als zwei getrennte, wenngleich auch ständig miteinander interagierende Berechnungsmodi betrachtet. Demgemäß setzt sich die Architektur von Sita aus zwei Modulen zusammen, der *Wissensbasis* und der *Handlungsbasis*.

Die Wissensbasis verwendet Hornklausellogik (mit Negation und Funktionssymbolen) als ausdrucksstarken Formalismus zur Beschreibung der Situationen, auf die reagiert werden soll. Die geforderte ereignisgetriebene Auswertung wird durch einen inkrementellen, vorwärtsverkettenden Algorithmus erreicht.

Auf der anderen Seite dient ein nebenläufiger Prozeßkalkül zur Spezifikation der prozeduralen Handlungen des Agenten. Prozeduren können dabei ihren weiteren Verlauf

jederzeit vom Eintreffen bestimmter Situationen abhängig machen. Andererseits können Prozeduren selbst die Wissensbasis ändern, um das Situationswissen des Agenten fortzuschreiben.

Diese geteilte, deklarative und imperative Modellierung führt zu einer neuen Programmiermethodik, die Agentenverhalten als *situatives Agieren* betrachtet und beschreibt.

Der Ausdruck „situated actions" wurde ursprünglich von Suchman [Suc87] im Zusammenhang mit ihrer Arbeit an Mensch-Maschine-Schnittstellen eingeführt. Sie argumentiert, daß (menschliche) Aktionen niemals in einem strengen Sinne vorausgeplant werden. Vielmehr sind alle Aktionen in starkem Maße durch die aktuelle, spezifische Situation bestimmt. Diese Beobachtungen unterstreichen die Bedeutung einer ausdrucksstarken Situationsbeschreibungssprache.

1.3 Beiträge der Arbeit

Die vorliegende Arbeit trägt in folgenden Punkten dazu bei, die Methodik des Programmierens situativ agierender Softwaresysteme zu bereichern:

Situationserkennung: Zur Spezifikation der Situationsklassen, auf die der Agent reagieren soll, steht die Ausdrucksstärke der Hornklausellogik zur Verfügung. Die rein deklarative Programmierung kommt einer intuitiven Beschreibung von Situationen auf hohem Abstraktionsniveau entgegen.

Effiziente Auswertung: Es wird ein inkrementeller, vorwärtsverkettender Algorithmus zur Auswertung von Hornklauselprogrammen entwickelt. Wir zeigen einige neue Optimierungstechniken auf, die wesentlich zur Effizienz des Verfahrens beitragen.

Programmiersprache: Zusammen ergeben Situationserkennung und prozedurale Handlungsbasis ein konkretes Berechnungsmodell situativen Agierens. Als Programmiersprache entsteht damit ein neues Werkzeug zur Realisierung reaktiver und zielgerichteter Agenten.

1.4 Überblick

Das folgende Kapitel gibt zunächst einen kurzen Überblick über grundlegende Agentenarchitekturen und beleuchtet damit den Hintergrund, vor dem Agentenprogrammierung betrachtet werden muß. Daraufhin werden drei konkrete Agentensprachen vorgestellt und bewertet, und so Bezugspunkte zur Einordnung der Konzepte von Sita geschaffen.

Kapitel 3 befaßt sich mit der Architektur von Sita. Die Fähigkeit zu reaktivem Verhalten wird auf das Zusammenspiel zweier komplementärer Teilleistungen (Erkennen und Handeln) zurückgeführt, woraus der zweiteilige Aufbau von Sita abgeleitet wird. Die Programmiersprache wird informell eingeführt und der Bezug zu verwandten Programmiermodellen hergestellt. Schließlich wird das Modell als reaktive Agentenarchitektur betrachtet.

In den folgenden zwei Kapiteln werden die zwei Teilsprachen von Sita formal eingeführt. Den Anfang macht in Kapitel 4 die Wissensbasis, deren Syntax und Semantik definiert werden. Dabei stützen wir uns auf die aus der Literatur bekannte Begrifflichkeit der wohl-fundierten Semantik ab. Schließlich erläutern wir die ebenfalls aus der Literatur übernommene magic-set Transformation, die wesentlich zur Praktikabilität des Formalismus beiträgt.

Kapitel 5 definiert Syntax und Semantik der in der Handlungsbasis eingesetzten prozeduralen Sprache. Zusätzlich zu den essentiellen Konstrukten werden verschiedene Erweiterungsmöglichkeiten der Sprache diskutiert.

Kapitel 6 beschäftigt sich mit der algorithmischen Auswertung des Hornklauselprogrammes. Das Problem wird zunächst in die Begrifflichkeit der relationalen Algebra übertragen. Hierdurch stellt sich das Logikprogramm als Gleichungssystem dar, welches durch ein Netz aus Relationen und Operationsknoten modelliert wird. Für den Zustand des Netzes wird ein Konsistenzbegriff definiert, der mit der gewünschten Programmsemantik korrespondiert. Sodann wird beschrieben, wie die inkrementelle Auswertung durch Propagieren von Veränderungen durch das Netz erreicht wird. Besondere Aufmerksamkeit kommt hierbei der Vermeidung unerwünschter Zirkelschlüsse zu. Anschließend werden verschiedene Optimierungstechniken besprochen und experimentell bewertet. Schließlich geben wir eine Programmklasse als Grenze der Anwendbarkeit des Verfahrens an. Es sind dies Programme, die eine bestimmte Eigenschaft bei der kombinierten Verwendung von Negation und Rekursion erfüllen.

In Kapitel 7 zeigen wir die Anwendung von Sita an einem Beispielproblem auf. Wir vergleichen die Formulierung in Sita mit der in zwei alternativen Sprachen und demonstrieren so die durch unseren Ansatz erzielten Verbesserungen.

Das abschließende Kapitel 8 faßt die wesentlichen Ergebnisse der Arbeit zusammen und zeigt mögliche Weiterentwicklungen auf.

KAPITEL 2

Stand der Technik

Ein Überblick über grundlegende Agentenarchitekturen beleuchtet den Hintergrund, vor dem wir Agentenprogrammierung betrachten. Anschließend werden mit Magsy, Cuncurrent METATEM und AgentSpeak(L) drei konkrete Agentensprachen vorgestellt, die als Bezugspunkte für die Einordnung und Bewertung von Sita dienen.

2.1 Einleitung

Situatives Agieren in dynamischen Umgebungen ist ein wesentliches Grundthema jener Forschung, die seit fünfzehn Jahren unter dem Stichwort intelligente Agenten stattfindet. Die Arbeiten auf diesem Gebiet lassen sich nach [WJ95] in drei Bereiche einteilen:

Theorien sind im wesentlichen Spezifikationen von „Agentenschaft“: Wie kann der Begriff des Agenten konzeptualisiert werden. Welche Eigenschaften sind relevant und wie können diese formal definiert werden?

Architekturen sind Modelle zur Konstruktion von Agenten: Wie können Computer-Systeme strukturiert und modularisiert werden, um die in den Theorien spezifizierten Eigenschaften zu erhalten?

Sprachen umfassen Entwicklungssysteme und Programmiersprachen, die den theoretischen und architektonischen Konzepten eine konkrete Form geben: Wie können Agenten programmiert werden? Welches sind die geeigneten Primitive? Wie können Agentenprogramme effizient übersetzt und ausgeführt werden?

Das in dieser Arbeit vorgestellte Sita ist als Berechnungsmodell gemäß dieser Einteilung zunächst den Agentensprachen zuzurechnen. Gleichzeitig ist Sita aber auch als

eine bestimmte Agentenarchitektur aufzufassen. Zudem ist die Thematik des situativen Agierens für praktisch alle Agentenarchitekturen relevant, so daß die mit Sita entwickelten Konzepte auch in den Entwurf komplexerer Architekturen einfließen können.

Um einen Rahmen zur Einordnung von Sita zu schaffen, geben wir im folgenden einen kurzen Überblick grundlegender Agentenarchitekuren und stellen darauf einige Systeme bzw. Sprachen vor, die ähnlich wie Sita der Spezifikation oder Programmierung situativen Verhaltens dienen. Zunächst stellen wir aber ganz kurz dar, was wir dem Konzept des Agenten zurechnen wollen.

Für diese Begriffsbestimmung lehnen wir uns an [WJ95] an und verstehen unter einem Agenten ein Computersystem, das sich in der Situation einer veränderlichen Umgebung befindet und unter eigener Kontrolle mit dieser Umgebung interagiert, um seine Entwurfsziele zu erfüllen. Wesentliche Attribute eines intelligenten Agenten sind demnach:

Autonomie: Der Agent handelt selbständig ohne direkten Eingriff seitens einer Kontrollinstanz.

Reaktivität: Der Agent reagiert auf aktuelle Ereignisse in der Umgebung, handelt also in Echtzeit.

Proaktivität: Der Agent zeigt zielgerichtetes Verhalten und initiiert dementsprechend geeignete Aktionen.

Als weitere Eigenschaften, die jedoch in unseren Augen weniger wesentlich sind, werden öfters genannt: soziale Fähigkeiten (im Sinne einer Interaktion mit anderen, in der Umgebung handelnden Agenten oder Menschen), Anpassungsfähigkeit (im längerfristigen Bereich Lernfähigkeit) sowie Mobilität (entweder physisch, zumeist aber im Sinne migrierender Berechnungsprozesse verstanden).

2.2 Agentenarchitekturen

Wir übernehmen aus [Woo99] die Einteilung in vier Architekturklassen: Logikbasierte Agenten, rein reaktive Agenten, BDI-Agenten[1] und hybride Agenten[2].

[1] „Beliefs, Desires, Intentions“

[2] in [Woo99] als Layered Agents bezeichnet

2.2.1 Logikbasierte Architekturen

Dieser Ansatz entspringt der traditionellen „künstlichen Intelligenz“, wo die Problemdomäne mit Hilfe eines geeigneten Logik-Kalküls beschrieben wird und Problemlösen als Inferenz in diesem Kalkül realisiert wird (vergleiche etwa [GN87]).

Der Agent erhält aufgrund seiner Wahrnehmung eine Datenbasis logischer Formeln, die sein direktes Wissen über seine Umwelt darstellen. Zudem existiert eine Theorie des gewünschten Verhaltens, etwa in Form von Deduktionsregeln. Entscheidungsfindung heißt nun, aus diesen Daten und Regeln Aktionen logisch abzuleiten, bzw. ein Modell zu konstruieren, welches die vorgegebene Theorie erfüllt.

Als formale Werkzeuge kommen hier unter anderem der Situationenkalkül [MH69] und der Ereigniskalkül [KS86] zum Einsatz, letztere oft in Verbindung mit abduktivem Schlußfolgern [Esh88] [Sha89]. Formale Basis bildet im allgemeinen die Prädikatenlogik, oft um temporale Modalitäten oder solche um Wissen und Glauben erweitert.

Diese Ansätze haben ihren Reiz in einer klaren Semantik. Sie sind von großer Bedeutung zur Formulierung von Agententheorien. Bekanntestes Beispiel ist die Charakterisierung von Agentenschaft mittels der BDI-Theorien [BIP88] [RG91], siehe auch Abschnitt 2.2.3.

Als praktische Architektur haben viele dieser Ansätze das Problem, daß sich die erforderlichen logischen Schlußfolgerungen einer effizienten Berechnung widersetzen. Die eingesetzten Logiken sind zu mächtig, als daß ihre vollständige algorithmische Auswertung möglich wäre. In der Praxis müssen daher Abstriche gemacht werden mit der Folge, daß die Semantik des realisierten Systems mehr oder weniger stark von der sauberen Semantik der Theorie abweicht.

Ähnliche Probleme hat aber bereits die klassische Logikprogrammierung, die dennoch als wichtiges Programmierparadigma erhalten blieb, und die nun auch als Grundlage einiger logikbasierter Agentenarchitekturen dient.

Beispiele für logikbasierte Architekturen sind Congolog [LLL+96] (basierend auf dem Situationenkalkül), Eve [JFB96] (basierend auf dem Ereigniskalkül und Abduktion), sowie das in Abschnitt 2.3.2 dargestellte METATEM [Fis93c] (basierend auf einer Temporallogik).

Auch Sita verwendet eine logikbasierte Komponente (die Sita-Wissensbasis), die sich allerdings funktional auf die Situationserkennung des Agenten beschränkt. Durch Einschränkungen in Mächtigkeit und Funktion bietet diese Logik beides, saubere Semantik und effiziente Auswertbarkeit.

2.2.2 Rein reaktive Architekturen

Als Antwort auf die Schwierigkeiten des logikbasierten Zuganges vermeiden rein reaktive Ansätze jede komplexere symbolische Repräsentation von Wissen. Stattdessen werden (zumeist physische) Agenten mit einer Menge relativ einfacher Verhaltensweisen ausgestattet, etwa in Form von Sensor-Aktor-Regeln.

Am meisten Beachtung hat hier die *Subsumption-Architektur* [Bro86] von Brooks gefunden.

Hier wird der Entwurf eines Agenten zerlegt in eine Menge aufgabenorientierter Verhaltensmodule, welche alle unabhängig voneinander Zugriff auf die Sensorik haben und jede für sich eine einfache und effizient zu berechnende Funktionalität aufweisen. Insbesondere wird jedes symbolische Schlußfolgern vermieden. Die Module konkurrieren untereinander um die Kontrolle der Aktoren des Agenten. Sie sind dazu in einer priorisierten Hierarchie angeordnet, in der tiefere Ebenen die einfacheren und dringlicheren Verhaltensweisen darstellen und Vorzug gegenüber den höheren Ebenen erhalten.

Eine solche Architektur kann sehr effizient implementiert werden und stellt sich zudem als sehr robust heraus. Aufgrund der fehlenden Repräsentation von Umgebungswissen kann sich der Agent bei der Entscheidungsfindung allerdings nur auf die lokal wahrnehmbare Information stützen. Die fehlende Erinnerungsfähigkeit bedingt eine kurzfristige Sichweise des Agenten.

Es wird argumentiert, daß durch das Zusammenspiel einfacher Verhaltensweisen bzw. der so ausgestatteten Agenten intelligentes Gesamtverhalten *emergiert*, und dieses plötzliche Auftauchen von Intelligenz wird auch in einigen Szenarien eindrucksvoll demonstriert (siehe etwa [Ste90]). Allerdings dürfte es zumindest schwierig sein, zu einem geforderten komplexeren Verhalten rein reaktive Agenten zu entwerfen, die dann eben jenes Verhalten zeigen werden. Für viele Aufgaben scheint dann doch eine interne Repräsentation von Wissen notwendig zu sein.

Sita unterstützt die Formulierung reaktiven Verhaltens in einer Form ähnlich zu Situations-Aktions-Regeln, wobei symbolische Repräsentation und logisches Schlußfolgern nicht abgelehnt werden, sondern vielmehr dazu dienen, eine größere Ausdrucksmächtigkeit in der Situationserkennung zu erzielen. Andererseits erfordert diese Ausdrucksstärke einen größeren Rechenaufwand, so daß Sita nicht so enge Echtzeitanforderungen erfüllen kann, wie dies etwa in der Subsumption-Architektur möglich ist.

2.2.3 BDI-Architekturen

BDI-Architekturen (Beliefs, Desires, Intentions) verwenden die in ihrem Namen bereits angesprochenen *mentalen Kategorien*, um ein Modell praktischen Schließens zu

entwickeln. Als zentrale Kategorie greifen wir hier die Intentionen heraus, mit welchen jene Ziele bezeichnet werden, auf die sich der Agent aktuell festgelegt hat. Intentionen entstehen oder werden fallengelassen auf der Basis der Optionen, die der Agent aufgrund seines gegenwärtigen Weltbildes zusammen mit seinen langfristigen Zielen (Desires) für sich sieht. Die Menge der aktuellen Intentionen treibt wiederum eine Mittel-Ziel-Analyse an, in der schließlich die Aktionen ausgewählt werden, um den Intentionen nachzukommen. Wesentlich an dem Modell ist, daß der Agent bestehende Intentionen nicht ohne Grund aufgibt, andererseits aber nicht an Intentionen festhält, die inzwischen keine Aussicht mehr auf Erfolg haben. Hier muß eine Balance zwischen proaktivem (zielgerichteten) und reaktiven Verhalten gefunden werden.

Der BDI-Ansatz hat einen großen Einfluß auf die Agenten-Forschung, insbesondere auch auf seiten von Agententheorien, wo die mentalen Kategorien mittels eines modal-logischen Kalküls axiomatisiert werden. Dadurch wird es möglich, Begriffe wie „rationales Verhalten“ und bestimmte Agenteneigenschaften auf einer formalen Basis zu diskutieren (vergleiche Abschnitt 2.2.1).

Andererseits liefert die BDI-Terminologie auch ein Modell zum funktionalen Aufbau von Agenten. Bekannteste Architektur in dieser Richtung ist das *Procedural Reasoning System* (PRS) [GL87], sowie die davon abgeleiteten Systeme *dMARS* [dKLW98] und *AgentSpeak(L)* [Rao96]. Auf das letztere wird im Abschnitt 2.3.3 näher eingegangen.

BDI-Architekturen wie das PRS sind dabei nicht als semantiktreue Implementierungen von BDI-Theorien zu verstehen. Vielmehr werden die in den Theorien verwendeten Modalitäten wie Überzeugungen, Ziele und Intentionen auf konkrete Datenstrukturen entsprechender Funktionsmodule abgebildet. So werden etwa Überzeugungen als Prolog-ähnliche Fakten repräsentiert, Intentionen als Threads und Pläne als Prozeduren.

BDI-Architekturen liegen konzeptionell auf einer abstrakteren Modellierungsebene als die Sita-Architektur. So könnte Sita als Implementierungsplattform für eine BDI-Architektur dienen, wobei die Situationserkennung von Sita in verschiedenen Funktionen eingesetzt werden kann: Bei der Revision und Abstraktion des Agentenwissens, zur Erkennung von Handlungsoptionen, bei der Pflege der Intentionen sowie bei der Auswahl konkreter Aktionen.

Andererseits lassen sich gewisse Konstrukte von Sita durchaus selbst als mentale Konzepte interpretieren. In Abschnitt 3.4 werden wir etwa die Fakten der Sita-Wissensbasis als Beliefs des Agenten und die Threads der Sita-Handlungsbasis als Intentionen betrachten. Allerdings stellt Sita allein damit noch keine BDI-Architektur dar.

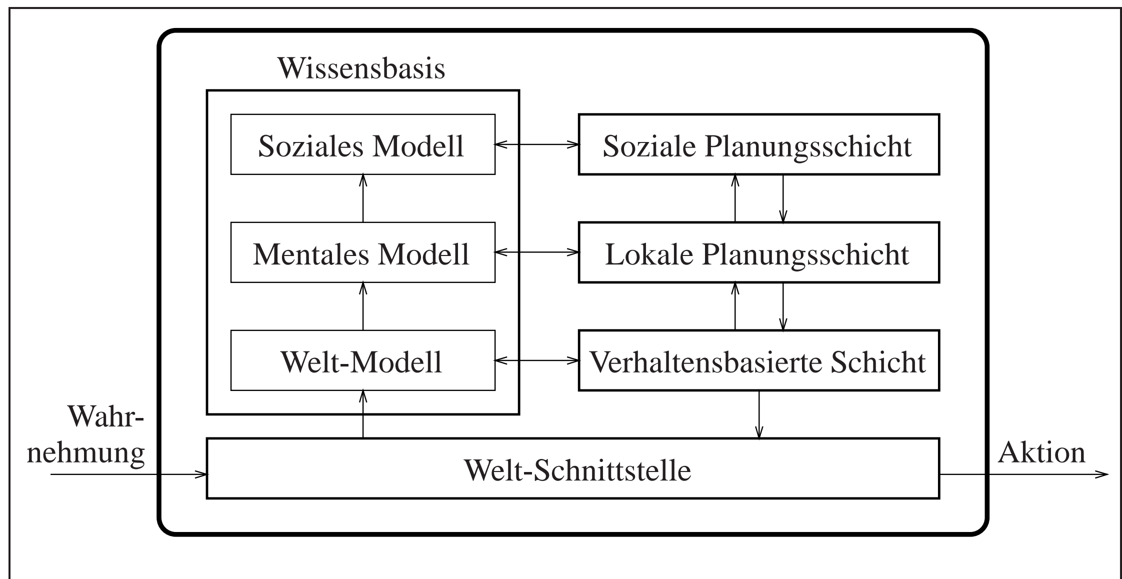


Abbildung 2.1: InteRRaP-Architektur

2.2.4 Hybride Architekturen

Die Anforderung, sowohl reaktives wie zielgerichtetes Handeln zu unterstützen, legt eine Zerlegung in entsprechende Funktionsblöcke nahe.

Beispiel einer solchen hybriden Architektur ist InteRRaP („Integration of Reactivity and RAtional Planning“) [MP93], das sich im wesentlichen aus drei Schichten konstruiert, siehe Abbildung 2.1: Verhaltensbasierte Schicht, lokale Planungsschicht und soziale Planungsschicht. Diese sind jeweils für reaktives, deliberatives bzw. kooperatives Verhalten zuständig. Jede Schicht ist einem entsprechenden Ausschnitt aus einer Wissensbasis zugeordnet, wobei das Abstraktionsniveau in Richtung höherer Schichten ansteigt.

Sieht sich eine Schicht einer Situation gegenüber, welche ihre Kompetenz übersteigt, so wird die nächsthöhere Schicht aktiviert. Umgekehrt beauftragt eine höhere Schicht die nächsttiefere mit der Ausführung einfacherer Aufgaben.

Geschichtete Architekturen wie InteRRaP stellen sehr pragmatische Ansätze dar, Agenten funktional zu beschreiben. Ihre komplexe innere Struktur macht es allerdings schwierig, exakte formale Beschreibungen insbesondere ihrer Semantik anzugeben. In diese Richtung geht die Weiterentwicklung InteRRaP-R [JF98] [GJ98]. Dort wird der Schichtenaufbau auf ein formal spezifiziertes Zusammenspiel nebenläufiger Prozesse abgebildet, die ihrerseits als Inferenzprozeduren einer übergreifenden Zeit- und Aktionslogik realisiert sind.

Insgesamt gilt auch hier: Die hybriden Funktionsmodelle liegen auf einer anderen Modellierungsebene als Sita, welches ein Berechnungsmodell und kein Funktionsmodell

darstellt. Die Konzepte von Sita unterstützen auf direkte Weise die Formulierung reaktiven Verhaltens. Für weitergehende Funktionen kann Sita als reaktive Hochsprache eine komfortable Entwicklungsplattform darstellen.

2.3 Programmierung situativen Verhaltens

Wir stellen mit Magsy, Concurrent METATEM und AgentSpeak(L) drei Programmiermodelle vor, die alle zur Beschreibung situativen Verhaltens ausgelegt sind, dies aber durch sehr unterschiedliche Konzepte realisieren.

2.3.1 Bedingungs-Aktions-Regeln: Magsy

Im regelbasierten Multiagentensystem Magsy („Multi-AGent SYstem“) [Fis93a] wird, wie in der Einleitung bereits angesprochen, der Kern eines Agenten aus einem Produktionensystem, also einem vorwärtsverkettenden Regelinterpreter gebildet. Jeder Agent besitzt damit die potentielle Problemlösefähigkeit eines Expertensystems.

Durch Nachrichtentransfer tauschen die Agenten eines Systems Wissen aus und nehmen gegenseitig ihre in einer Schnittstellendefinition beschriebenen Dienste in Anspruch. Des weiteren sind Magsy-Agenten in der Lage, dynamisch neue Agenten zu erzeugen und in das System einzubinden.

Das Verhalten eines Agenten wird durch eine Menge von Bedingungs-Aktions-Regeln spezifiziert, die als Bibliothek der dem Agenten zur Verfügung stehenden Pläne betrachtet werden können. Jeder Agent verfügt über eine interne Wissensbasis. Das darin in Faktenform repräsentierte lokale Wissen kann in den Bedingungsteilen der Regeln getestet und in den Aktionsteilen modifiziert werden.

Zur Ausführung vergleicht der Interpreter zyklisch zunächst die vorhandenen Fakten mit den Bedingungsteilen der Regeln. Dann wird gemäß einem Prioritätenschema eine erfüllte Regelinstanz ausgewählt und der zugehörige Aktionsteil zur Ausführung gebracht. Zudem werden eintreffende Nachrichten oder Sensorwerte asynchron als Fakten in die lokale Wissensbasis eingetragen.

Das eingesetzte Berechnungsmodell steht in einem engen Zusammenhang zu Petri-Netzen. Dabei werden Fakten auf Marken abgebildet, Bedingungen auf Stellen und Aktionen auf Transitionen. Dadurch erhält man ein formales Modell der Berechnung als nebenläufiges System. Das Regelsystem wird damit der formellen und bis zu einem gewissen Grade auch maschinellen Analyse[3] zugänglich, etwa zur Verifikation bestimmter Eigenschaften.

[3] Vergleiche etwa [Sta90].

Sita hat wichtige konzeptionelle (und auch historische) Bezüge zu Magsy und dem in dessen Implementierung zugrundegelegten Produktionensystem OPS5 [For81]. Insbesondere übernehmen wir in Sita das Prinzip der vorwärtsverkettenden, datengetriebenen Auswertung. Sie liefert den Schlüssel für das reaktive, logik-basierte Berechnungsmodell von Sita. OPS5 verwendet den inkrementell arbeitenden Rete-Algorithmus [For82], dessen Prinzip auch die Grundlage des für Sita (in Kapitel 6) entwickelten Auswertungsalgorithmus liefert.

Aus programmiersprachlicher Sicht besteht ein Magsy-Programm aus einer flachen, unstrukturierten Menge von Bedingungs-Aktions-Regeln. Dadurch werden größere Regelprogramme schnell unübersichtlich [Boc89], „nahezu unverständlich“ [Tic87] und sind schwer wartbar. Die ursprüngliche Vision von Produktionensystemen (vergleiche [BS84]), nach der jede Regel eine unabhängige Wissensquelle ist und sich ihr Zusammenspiel ohne zusätzlichen Aufwand von selbst ergeben würde, stellte sich somit als nicht sehr tragfähig heraus. Diese Schwierigkeiten haben sich bereits beim Einsatz von Regelprogrammen in Expertensystemen gezeigt [BFKM85], so etwa beim Konfigurationsprogramm XCON [SBJ87].

Sita begegnet diesem Mißstand mit der Einführung zweier sich ergänzender Teilsprachen, welche die Formulierung komplexer Bedingungen und strukturierter Aktionsfolgen erlauben. Zum weiteren Vergleich der Konzepte wird auf Abschnitt 3.3.3 verwiesen.

2.3.2 Ausführbare Temporallogik: Cuncurrent METATEM

Concurrent METATEM [Fis93c] ist eine Sprache zur Spezifikation und Prototyping reaktiver Systeme. Sie beruht auf der Verwendung und direkten Ausführung temporallogischer Formeln, welche als Regeln vom vergangenen und gegenwärtigen Agentenzustand auf den zukünftigen Zustand schließen.

Es wird eine lineare, diskrete Temporallogik mit endlicher Vergangenheit zugrunde gelegt. Die herkömmlichen prädikatenlogischen Konstrukte werden um zwei binäre temporale Operatoren $\mathcal{U}$ („until“) und $\mathcal{S}$ („since“) erweitert:

$\varphi\,\mathcal{U}\,\psi$ ist zu einem Zeitpunkt s wahr, wenn ψ zu einem späteren Zeitpunkt t wahr ist und φ zu allen Zeitpunkten zwischen s und t wahr ist. $\varphi\,\mathcal{S}\,\psi$ ist zu einem Zeitpunkt s wahr, wenn ψ zu einem früheren Zeitpunkt t wahr ist und φ zu allen Zeitpunkten zwischen t und s wahr ist.

Aus diesen Operatoren lassen sich weitere ableiten:

$\bullet\,\varphi$	$:= \mathbf{false}\,\mathcal{S}\,\varphi$	φ ist im direkt vorhergehenden Zustand wahr.
$\Diamond\,\varphi$	$:= \mathbf{true}\,\mathcal{U}\,\varphi$	φ wird in einem zukünftigen Zustand wahr sein.
$\mathbf{start}$	$:= \neg\bullet\mathbf{true}$	es gibt keinen vorhergehenden Zustand.

Hiermit lassen sich nun METATEM-Regeln angeben, welche als Implikation

$$P(X) \;\Rightarrow\; Q(X)$$

notiert werden. (Hier und im folgenden sei X ein Vektor von Variablen. k_a, l_b, m_j und l seien Literale.) Die Prämisse $P(X)$ einer Regel ist eine Bedingung an den vorhergehenden und an den aktuellen Zustand und besitzt einer der beiden Formen:

$$\mathbf{start} \wedge \bigwedge_b l_b(X) \quad \text{oder} \quad \bullet \bigwedge_a k_a(X) \wedge \bigwedge_b l_b(X)$$

Die Konklusion $Q(X)$ einer Regel fordert eine Eigenschaft des aktuellen Zustandes oder eines zukünftigen Zustandes. Hierbei sind die folgenden beiden Formen zulässig:

$$\bigvee_j m_j(X) \quad \text{oder} \quad \Diamond\, l(X)$$

Die Regeln haben somit immer die Form „Vergangenheit und Gegenwart implizieren Gegenwart und Zukunft“[4].

Zur Kommunikation mit der Umgebung (insbesondere mit weiteren METATEM-Prozessen, daher auch der Name *Concurrent* METATEM) gibt es einen Mechanismus zum Nachrichtenaustausch, der über Prädikate spezieller Funktionalität in die Logik eingebunden ist. Eingangsprädikate repräsentieren eingehende Nachrichten und werden zu dem Zeitpunkt wahr, zu dem eine entsprechende Nachricht eintrifft. Ausgangsprädikate repräsentieren ausgehende Nachrichten. Zu dem Zeitpunkt, zu dem sie erfüllt sind, wird als Seiteneffekt eine entsprechende Nachricht verschickt.

Der vorgestellte Mechanismus erlaubt es, reaktives Verhalten auf einer deklarativen Ebene zu beschreiben. Anwendungsbeispiele sind die Modellierung einfacher Eisenbahn-Netzwerke (Strecken, Signale, Zugbewegungen) [FFO93], die Simulation verschiedener Verhaltensweisen in einer handeltreibenden Gesellschaft [Fis94] und die Spezifikation kooperativer Protokolle [FW94].

Zum Prototyping solcher Modellierungen lassen sich METATEM-Programme ausführen, indem das System schrittweise ein Modell zu konstruieren versucht, welches die logischen Formeln des Programmes erfüllt. Hierzu werden die Regeln vorwärts verkettet, um aus den bereits bekannten bisherigen Zuständen einen neuen konsistenten Zustand abzuleiten. Aus den Regeln, deren Konklusionen $\Diamond$-Formeln sind, ergeben sich Anforderungen an zukünftige Eigenschaften des Modells. In jedem Schritt wird versucht, möglichst viele der noch ausstehenden Anforderungen zu erfüllen. Die in einem Schritt nicht erfüllbaren Anforderungen werden auf einen späteren Schritt verschoben.

[4] Die Regelform ist auch theoretisch motiviert, da sie eine eingeschränkte Variante einer Normalform darstellt, auf die sich alle Formeln der dargestellten Logik transformieren lassen [Fis92].

Sowohl durch die Disjunktion als auch durch den $\Diamond$-Operator ergeben sich bei der Modellkonstruktion Auswahlpunkte. Läuft die Auswertung in eine Sackgasse (durch Erreichen eines inkonsistenten Zustandes), wird ein Backtracking über diese Auswahlpunkte durchgeführt. Backtracking darf allerdings nicht hinter das Verschicken einer Nachricht zurücksetzen, da eine Nachricht als von außen beobachtbares Ereignis nicht zurückgenommen werden kann. Das Versenden einer Nachricht bedeutet also eine Festlegung der Ausführung auf den gegenwärtig eingeschlagenen Pfad.

Bereits ohne den temporalen Operatoren stellt ein METATEM-Programm ein disjunktives Logikprogramm mit Negation dar, so daß sich hier neben Fragen der Semantik auch solche der Effektivität und Effizienz der Modellkonstruktion stellen. Hierzu wird in [Fis95] darauf hingewiesen, daß in Anbetracht der hohen Ausdrucksstärke der beschriebenen Temporallogik der Auswertungsmechanismus nicht als vollständiger Theorembeweiser verstanden werden soll, sondern lediglich als *Versuch*, ein Modell des Programmes zu erzeugen. In [Fis94] wird angedeutet, daß eine bisherige Implementierung nur den aussagenlogischen Fall abdeckt, eine Unterstützung für die volle Prädikatenlogik aber noch aussteht.

Aufgrund der Form der Regeln (Vergangenheit impliziert Zukunft) und der Art ihrer Ausführung wird das Paradigma in [Gab89] als „Declarative Past and Imperative Future“ benannt. Dieses Bezeichnung könnte wörtlich auch auf Sita angewandt werden, da hier ein deklarativer Formalismus zur Situationserkennung mit einem imperativen Formalismus zur Beschreibung von Handeln kombiniert wird. Sita kodiert Handeln jedoch explizit mit Hilfe prozeduraler Kontrollstrukturen, während METATEM hierfür einen Schlußfolgerungsalgorithmus bemüht, der die in die Zukunft reichenden logischen Formeln in Handlungen umsetzt. Imperativ ist hier also weniger die temporallogische Darstellung als vielmehr ihr Interpretationsmechanismus.

Während METATEM eher als Spezifikationssprache anzusehen ist, versteht sich Sita mehr als Programmiersprache. Durch die prozedurale Handlungssprache wird das Problem unvollständiger oder ineffizienter Inferenzalgorithmen vermieden. Zudem erscheint die prozedurale Kodierung von Handlungsfolgen oft die natürlichere, intuitivere Methodik zu sein.

2.3.3 Prozedurales Reasoning: AgentSpeak(L)

AgentSpeak(L) [Rao96] und dMARS [dKLW98] sind zwei konkrete Formulierungen des Procedural Reasoning System (PRS, siehe Abschnitt 2.2.3), für die eine formale, operationelle (und im Falle von AgentSpeak(L) auch beweistheoretische) Semantik angegeben wurde. Während dMARS als kommerzielles C++-Framework entwickelt wurde, stellt AgentSpeak(L) dessen Abstraktion als textuelle Sprache dar. Als Realisation von PRS folgen beide dem Paradigma der BDI-Architekturen.

Der Zustand eines AgentSpeak(L)-Agenten ist gegeben durch eine Menge von Beliefs, einer Menge zu bearbeitender Ereignisse und einer Menge aktueller Intentionen.

Die dem Agenten zur Verfügung stehenden Verhaltensweisen sind durch eine Planbibliothek gegeben[5]. Außerdem ist das Verfahren noch in drei Selektionsfunktionen parametrisiert, die der Auswahl von Ereignissen, Optionen bzw. Intentionen dienen.

Die genannten Strukturen sind folgendermaßen aufgebaut:

- **Beliefs** werden als variablenfreie Prolog-ähnliche Fakten repräsentiert.
- Es gibt zwei Arten von **Zielen**: Das Erreichen eines Zustandes, in dem ein bestimmter Belief gilt, (Zustandsziele) und die Abfrage, ob eine bestimmter Belief-Ausdruck gilt (Abfrageziele).
- Als **Ereignisse** können auftreten: Externe Ereignisse (z.B Nachrichtenempfang), das Einfügen und Löschen von Beliefs sowie die Aufnahme von Zustandszielen.
- Jeder **Plan** der Planbibliothek besteht aus drei Komponenten: Ein triggerndes Ereignis, ein Plankontext, sowie als Rumpf eine Sequenz von Planschritten.
- Ein **Plankontext** ist eine logische Bedingung, gegen die die aktuelle Beliefmenge getestet werden kann. Als Konnektive stehen Konjunktion, Disjunktion und Negation zur Verfügung. Sowohl das triggernde Ereignis als auch der Plankontext dürfen Variablen erhalten, die bei einer Instantiierung des Planes entsprechend gebunden werden.
- **Planschritte** können sein: Externe Aktionen (z.B. Nachrichtenversand), Anweisungen zum Einfügen oder Löschen von Beliefs, sowie Zustands- und Abfrageziele.
- **Intentionen** schließlich sind als Keller organisierte Sequenzen von partiell instantiierten Plänen. Der oberste Plan einer Intention ist der als nächstes auszuführende Plan entsprechend dieser Intention.

Der Berechnungsprozeß eines AgentSpeak(L)-Agenten ergibt sich aus der Interaktion von Ereignissen und Intentionen. Ereignisse triggern Pläne, von denen ausgewählte in die Intentionen aufgenommen werden. Intentionen ihrerseits generieren bei ihrer Ausführung neben externen Aktionen wiederum neue Ereignisse.

[5] Die Planbibliothek ist zur Laufzeit unveränderlich. Wie die meisten anderen BDI-Architekturen auch besitzen AgentSpeak(L)-Agenten keine Möglichkeit, zur Laufzeit Pläne von Grund auf neu zu erstellen.

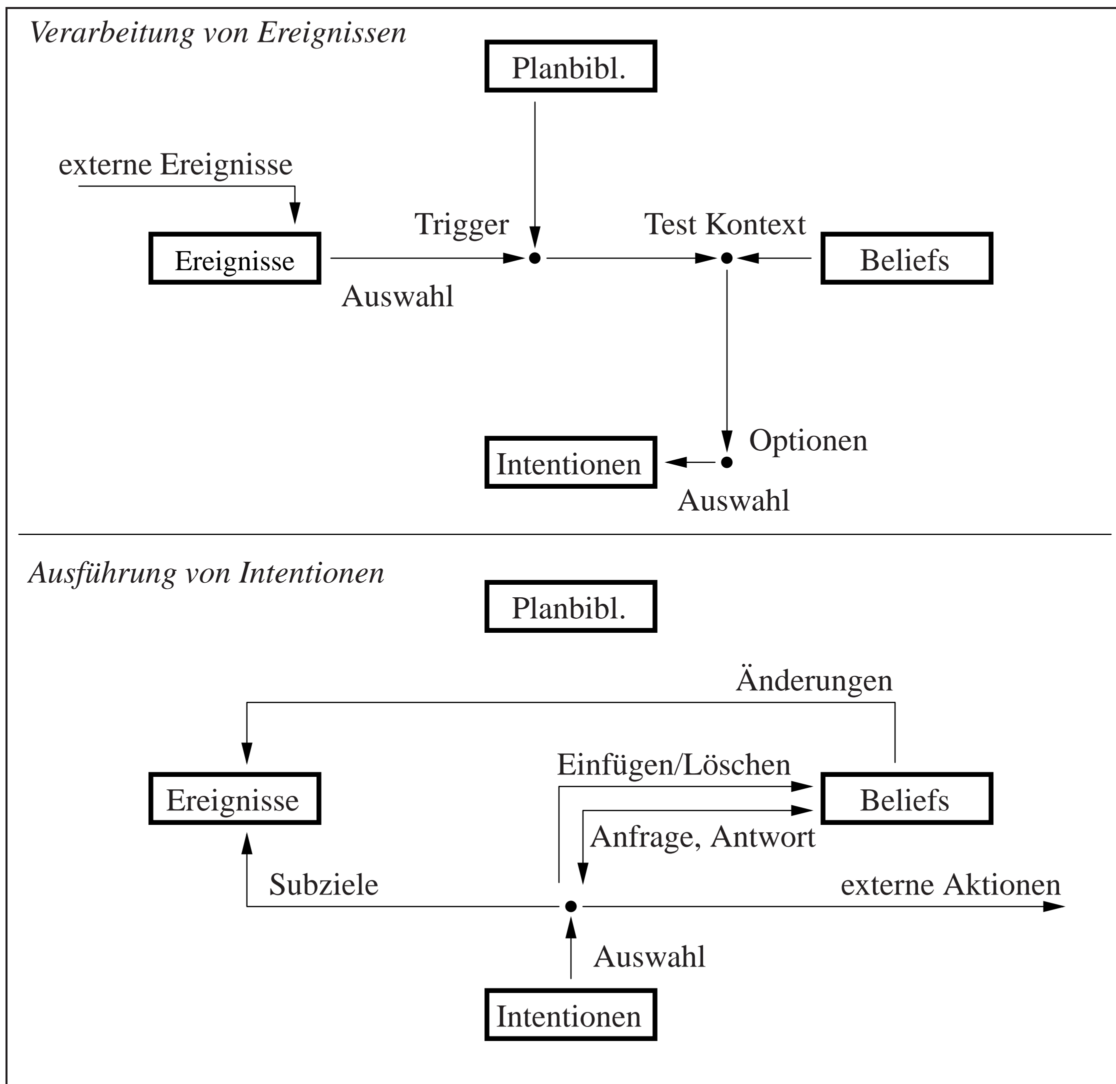


Abbildung 2.2: Operationen in AgentSpeak(L)

Dementsprechend werden zwei Arbeitsweisen unterschieden: Solange die Ereignismenge nicht leer ist, werden die anstehenden Ereignisse verarbeitet[6], siehe auch Abbildung 2.2 oben:

1. Wähle gemäß einer Selektionsfunktion ein noch unbearbeitetes Ereignis aus.

2. Suche in der Planbibliothek die gemäß den Trigger-Bedingungen passenden Pläne.

3. Teste, ob die Plankontexte dieser Pläne bei der vorliegenden Belief-Menge er-

[6] Dies entspricht der Darstellung in [dKLW98]. Im ursprünglichen Artikel [Rao96] wird immer abwechselnd ein Ereignis verarbeitet und ein Planschritt ausgeführt. In einer weiteren Darstellung [d'I98] bleibt dieser Punkt unspezifiziert.

füllbar sind. In diesem und im vorherigen Schritt werden gegebenenfalls in den Plänen vorkommende Variablen gebunden.

4. Von den anwendbaren Plänen (den sogenannten Optionen) wähle eine aus. Hierzu wird wiederum eine passende Selektionsfunktion vorausgesetzt.

5. Der ausgewählte Plan wird wird nun den Intentionen hinzugefügt:

   - Falls das auslösende Ereignis ein Zustandsziel war, so wird der ausgewählte Plan auf den Intentionenkeller gelegt, der das Ereignis ursprünglich erzeugt hat. (Mit anderen Worten: Ein Subziel wird expandiert.)
   - Andernfalls wird der ausgewählte Plan auf einen neu erzeugten Intentionenkeller gelegt.

Jeder der Intentionenkeller stellt einen nebenläufigen Ausführungskontext dar. Nachdem alle Ereignisse verarbeitet worden sind, wird mit der Ausführung der Intentionen fortgefahren, vergleiche Abbildung 2.2 unten.

1. Wähle gemäß einer weiteren Selektionsfunktion einen Intentionenkeller aus.

2. Führe den nächsten Planschritt der obersten Intention im Keller aus. Das Ergebnis hängt vom Typ des Planschrittes ab:

   - Ein Zustandsziel wird als neues Ereignis der Ereignismenge zugeschlagen.
   - Ein Abfrageziel wird als Anfrage an die aktuelle Beliefmenge gestellt und führt zu einer entsprechenden Variablenbelegung.
   - Eine Anweisung zur Modifikation der Beliefs wird ausgeführt. Ändert sich dadurch die Belief-Menge, so wird dies dementsprechend als Ereignis vermerkt.
   - Eine externe Aktion wird direkt zur Ausführung gebracht.

Damit ist die operationelle Semantik von AgentSpeak(L) erklärt. Eine formale Darstellung findet sich in [dL98].

AgentSpeak(L) enthält nur die essentiellen Konstrukte von PRS, welche in unterschiedlichen Aspekten erweitert werden können (vergleiche [Rao96]), z.B.: Aufgeben von Intentionen, Scheitern von Intentionen, dazu passende Ereignistypen. Neben der sequentiellen Anordnung von Planschritten könnten auch andere Konnektive eingeführt werden, etwa Operatoren zur alternativen oder parallelen Ausführung von Planzweigen.

Aus programmiersprachlicher Sicht ergibt sich eine große Nähe von Sita zu AgentSpeak(L). Die in AgentSpeak(L) zur Anwendung kommenden Repräsentations- und

Berechnungsmechanismen werden auch von Sita unterstützt, dort in einer etwas allgemeineren Form. Dabei ist der Abstraktionsgrad von AgentSpeak(L) nur wenig höher als der von Sita. Zusammen bedeutet dies, daß zum einen die mit Sita entwickelten Konzepte und Algorithmen als programmiersprachliche Basistechnologie zur Implementierung von PRS-Architekturen eingesetzt werden können, und daß zum anderen die PRS-Konzepte ihrerseits eine Methodik zur Realisierung von BDI-Agenten in Sita liefern können. Wir werden in Kapitel 7 auf die Gemeinsamkeiten und Unterschiede von AgentSpeak(L)- und Sita-Konstrukten zurückkommen.

2.4 Zusammenfassung

Die im letzten Abschnitt vorgestellten Systeme beinhalten jeweils verschiedene Konzepte zur Spezifikation von Situationen bzw. zur Repräsentation von Intentionen.

Wir betrachten zunächst den Abstraktionsgrad der Beschreibung reaktionswürdiger Situationen. In AgentSpeak(L) gibt es nur elementare Ereignisse zum Triggern von Plänen, nämlich Nachrichtenempfang, das Einfügen und Löschen von Beliefs sowie die Aufnahme von Zustandszielen. Die Plankontexte testen lediglich die Anwendbarkeit eines Planes, können aber nicht von sich aus einen Plan triggern. Mächtiger ist die Bedingungssprache von Magsy-Regeln. Hier stehen logische Operatoren wie Konjunktion und Negation zur Verfügung, um von elementaren Ereignissen auf abstraktere Handlungsauslöser zu schließen. Die von METATEM verwendete Logik schließlich ist in dieser Hinsicht die mächtigste Sprache, zu mächtig allerdings, um effizient implementierbar zu sein. Ein wichtiges Ziel bei der Entwicklung von Sita war es deshalb, eine Balance zwischen Mächtigkeit und Effizienz der Situationsbeschreibung zu erzielen.

Auf der anderen Seite stehen sich verschieden strukturierte Repräsentationen der Intentionen des Agenten gegenüber. Magsy gibt hier keinerlei Strukturen vor, der Programmierer muß den Ausführungskontext strukturierter Pläne selbst verwalten. In METATEM werden die eingegangenen Absichten des Agenten in einer Menge von Zukunftszeit-Formeln gesammelt. Ein Ausführungsmechanismus versucht sie zu erfüllen, kann dies jedoch nicht garantieren. Die Intentionenstruktur von AgentSpeak(L) schließlich wendet das übliche Verfahren (nebenläufiger) imperativer Programmiersprachen an, den Ausführungskontext im Zusammenhang mit Unterprogrammaufrufen mit Hilfe von Kellern zu verwalten. Dies ist zugleich auch das für Sita gewählte Modell. Es folgt damit der einfachen Idee, prozedurales Wissen auch prozedural zu beschreiben.

Kapitel 3

Das Sita-Berechnungsmodell

Dieses Kapitel motiviert die Architektur des Sita-Berechnungsmodells, welche anschließend aus programmiersprachlicher Sicht erklärt und eingeordnet wird. Schließlich wird Sita aus einer anderen Perspektive als eine einfache reaktive Agentenarchitektur dargestellt.

3.1 Erkennen und Handeln

Reaktivität ist eines der Hauptmerkmale von Agentenmodellen und daher auch gestaltgebendes Prinzip bei der Entwicklung von Sita. In der Forderung nach Reaktivität liegt insbesondere die zweiteilige Architektur von Sita mit Wissensbasis und Prozeßkalkül begründet. Dies wird im folgenden näher ausgeführt.

Reaktives Verhalten konstituiert sich aus den beiden Vorgängen *Erkennen* und *Handeln*: Worauf wird reagiert und worin besteht die Reaktion.

Dementsprechend ist auch die Modellierung von reaktiven Systemen zweiteilig: Es werden Klassen von Situationen beschrieben, auf die reagiert werden soll, und dazu Handlungsvorschriften, die zu jeder dieser Klassen eine entsprechende Reaktion spezifizieren.

Handeln heißt Verändern. Handeln kann sich nach außen richten, so daß der Agent die Umwelt, in der er lebt, verändert, etwa durch Kommunikation oder physisch durch den Einsatz von Effektoren. Handeln kann aber auch inneres Handeln sein, mit dem der Agent seine „innere Welt“ verändert, etwa indem er sein Weltmodell angleicht oder seine Vorhaben modifiziert. Wesentlich an Handlung ist, daß sie Festlegung bedeutet, daß sie Geschichte schreibt.

Der Übergang vom Erkennen zum Handeln ist Entscheidung. Aus der Dynamik der Umgebung und aus der Ressourcengebundenheit der Berechnung ergibt sich für den Agenten im allgemeinen eine Menge alternativer Handlungsweisen. Der Agent wird

sie bewerten, eine auswählen und ausführen, und dadurch seine Welt verändern. Aus der Existenz mehrerer Alternativen folgt die Notwendigkeit der Entscheidung.

Der Agent trifft seine Entscheidungen für bestimmte Handlungsweisen aufgrund seines Wissens um seine Situation. Dieses Situationswissen fließt ihm auf zwei Kanälen zu: Durch „Wahrnehmung“ der Umwelt (Sensorik, Kommunikation) einerseits, und durch „Erinnerungen“ an bisherige Handlungen andererseits. Beides zusammen ergibt das unmittelbare Faktenwissen, das dem Agenten zur Verfügung steht.

Das Vorgehen, das notwendig ist, um von diesen Basisfakten zu konkreten Entscheidungen zu kommen, wird zumeist in zwei Stufen beschrieben: Zunächst pflegt der Agent aufgrund neuer Fakten sein Weltmodell (Wissensassimilation). Daraufhin stellt er mögliche Handlungsalternativen auf und bewertet diese. Für die beste (weil z.B. dringlichste oder Erfolg versprechendste) Alternative wird er sich entscheiden. Dieser Prozeß des Erkennens der durchzuführenden Handlung wird als Entscheidungsfunktion bezeichnet.

Eine wesentliche Entwurfsentscheidung bei der Spezifikation von Agentenverhalten ist der Grenzverlauf zwischen Erkennen und Handeln. Eine perfekte, optimale Entscheidungsfunktion ist im allgemeinen unbekannt oder zu ineffizient.

Es ist eine Frage der Modellierungsmöglichkeiten, welche Funktionalitäten des Agenten als Erkennen realisiert werden können, und welche als Handlung zu beschreiben sind. Je mächtiger die Entscheidungsfunktion wird, desto aufwendiger ihre Berechnung. Da sich aber der Agent für das Erkennen des nächsten optimalen Schrittes nicht beliebig viel Zeit nehmen kann, muß reines Erkennen dort aufhören, wo Entscheidung und Handlung notwendig werden, um am Fortschreiten der Welt teilzunehmen.

Wir wollen dies am Beispiel des Schachspiels verdeutlichen. Zunächst ist klar, daß das Ausführen eines Zuges eine (unumkehrbare) Handlung darstellt. Dieser Entscheidung für einen Zug geht eine Bewertung alternativer Züge voraus. Auch klar ist allerdings, daß diese Entscheidung nicht perfekt sein kann, weil die vollständige Betrachtung des Suchraumes viel zu komplex ist, als daß sie in annehmbarer Zeit zu bewältigen wäre. Deshalb muß der spielende Agent auch innerhalb eines Zuges ständig Entscheidungen treffen, welche Äste des Suchraumes er weiterverfolgen soll, und welche er abschneidet. Auch diese Entscheidungen stellen Handlungen dar, insofern nämlich, als daß sich der Agent festlegt, in welchem Teil des Suchbaumes er seine knappe Zeit der Zugsuche verbringt. Man kann in diesem Zusammenhang von *innerem Handeln* sprechen.

3.2 Basisarchitektur von Sita

Kernidee von Sita ist es, Erkennen und Handeln als zwei verschiedene Berechnungsmodi zu betrachten, deren Formalisierung nach unterschiedlichen Spezifikationstechniken verlangt.

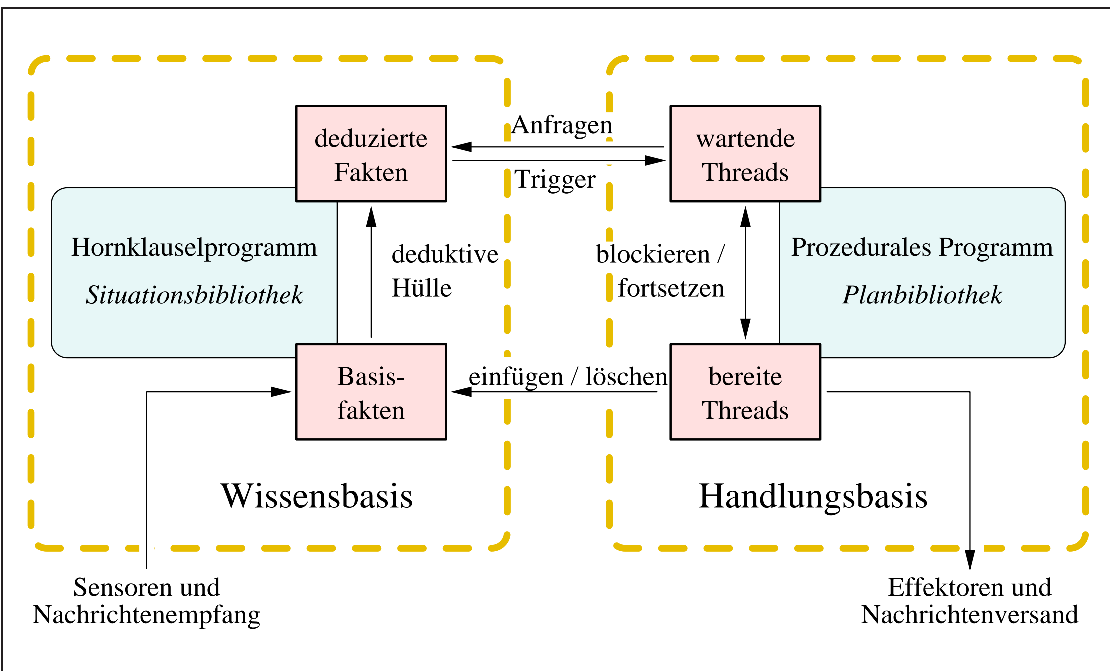


Abbildung 3.1: Übersicht Sita-Architektur

Der Verschiedenartigkeit dieser Funktionen wird durch die Verwendung zweier komplementärer Modellierungsparadigmen Rechnung getragen: Die Modellierung von Erkennen erfolgt *deklarativ* mit Hilfe eines logikbasierten Kalküls; Handeln dagegen wird *imperativ* modelliert durch die Verwendung eines Prozeßkalküls.

Wurde oben Reaktivität als Fähigkeit *erklärt* durch die komplementären Fähigkeiten des Erkennens und des Handelns, so wird nun reaktives Verhalten *realisiert* durch das Zusammenspiel zweier sich ergänzender Berechnungsformalismen.

Wir skizzieren im folgenden kurz diese Formalismen. Ausführlich werden sie in den Kapiteln 4 und 5 dargestellt.

Für die Situationserkennung ist ein *Wissensbasis* genanntes Modul zuständig (siehe Abbildung 3.1). Als Formalismus liegt diesem Hornklausel-Logik (mit Negation und Funktionssymbolen) zugrunde. Die zu erkennenden Situationen werden durch ein Logikprogramm (eine Menge von Hornklauseln) spezifiziert. Die Wissensbasis verwaltet zwei variable Mengen von Fakten. *Assertierte Fakten* werden von außen vorgegeben, das heißt sie werden eingefügt oder gelöscht, einerseits durch Wahrnehmung der Umwelt und andererseits durch spezielle Aktionen des prozeduralen Moduls. Assertierte Fakten repräsentieren das unmittelbare Faktenwissen. Im Gegensatz dazu stehen die *deduzierten Fakten*, welche sich als logische Folgerungen aus den assertierten Fakten zusammen mit den Klauseln des Logikprogrammes ergeben. Sie repräsentieren die abstrahierte Sicht des Faktenwissens und zeigen damit letztendlich jene Situationen an, auf die der Agent mit eigenem Handeln reagieren soll.

Realisiert wird die Wissensbasis durch eine Inferenzmaschine, welche die Menge der deduzierten Fakten in Abhängigkeit von der Menge der assertierten Fakten berechnet. Da die Wissensbasis typischerweise viele Situationsklassen gleichzeitig überwachen muß, sind an diesen Deduktionsprozeß spezielle Anforderungen zu stellen. Insbesondere wird die Auswertung nicht nur von den (Überwachungs-)Anfragen getrieben, sondern in erster Linie von den Daten, also den Veränderungen assertionaler Fakten. Hierfür wurde ein inkrementelles, vorwärtsverkettendes Verfahren entwickelt, welches in Kapitel 6 vorgestellt wird.

Der Wissensbasis gegenüber steht als komplementärer Berechnungsformalismus die *Handlungsbasis*. Hier werden in einem Prozeßkalkül die Aktionen des Agenten spezifiziert. Der Kalkül kennt elementare Aktionen, zusammengesetzte Aktionen und zu Prozeduren abstrahierte Aktionsmuster.

Elementare Aktionen erlauben zum einen intern die Modifikation der Wissensbasis, also das Einfügen und Löschen von Basisfakten. Zum anderen erlauben sie die Modifikation der Agentenumgebung in Form von externen Aktionen. Dazu gehören das Versenden von Nachrichten und (soweit vorhanden) die Steuerung physischer Effektoren.

Konstrukte zur Bildung zusammengesetzter Aktionen sind die sequentielle Komposition, die parallele Komposition sowie die überwachte Auswahl. Prozedurdefinitionen schließlich erlauben es, zusammengesetzte Aktionsfolgen zu benennen, um diese wiederum in weiteren Zusammensetzungen zu verwenden. Prozeduren sind damit nicht nur Strukturierungsinstrument sondern auch Voraussetzung zur Bildung rekursiver Aktionsausdrücke.

Die elementaren Aktionen zur Modifikation der Basisfakten stellen die eine Richtung der Schnittstelle zwischen Handlungsbasis und Wissensbasis dar. Die entgegengesetzte Richtung wird durch das Konstrukt der überwachten Auswahl gebildet. Die Wächter einer Auswahl stellen Anfragen an die Wissensbasis dar und prüfen auf das Vorliegen jener Situationen, auf die der Agent mit den jeweiligen Aktionen reagieren soll. Die Ausführung einer überwachten Auswahl wird suspendiert, bis einer ihrer Wächter erfüllt ist, das heißt logische Konsequenz der Wissensbasis ist, und fährt dann mit der zugehörigen Rumpfaktion fort. Die übrigen Alternativen der Auswahl werden verworfen. Dieses Triggern eines Auswahlzweiges ist im Berechnungsmodell das, was oben als Übergang vom Erkennen zum Handeln, als Entscheidung beschrieben worden ist. Es ist der Übergang von deklarativer zu imperativer Berechnung.

Durch die Verwendung der parallelen Komposition entstehen multiple Ausführungskontexte. Jeder dieser *Threads* ist durch einen Aktionsausdruck gekennzeichnet, der die noch durchzuführenden Aktionen beschreibt. Die Menge dieser Ausdrücke stellt den Zustand der Handlungsbasis dar.

Ein interner Scheduler bringt jeweils einen der nicht durch eine überwachte Auswahl

blockierten Threads zur Ausführung. Der Scheduler kann durch die Vergabe von relativen Thread-Prioritäten gesteuert werden.

3.3 Sita als Programmiersprache

In diesem Abschnitt werden einige Aspekte der Sita-Architektur aus programmiersprachlicher Sicht beleuchtet.

Jede Programmiersprache läßt sich durch die Identifizierung folgender drei konzeptueller Mittel näher charakterisieren: Atomare Konstrukte, kompositionelle Konstrukte und abstrahierende Konstrukte. In Sita gilt das für jede der beiden Teilsprachen, für die Logikprogrammierung der Wissensbasis genauso wie für den Prozeßkalkül der Handlungsbasis. Die einzelnen Konstrukte wurden teilweise bereits im vorhergehenden Abschnitt erwähnt, und sind in folgender Tabelle zusammengefaßt.

	Wissensbasis	Handlungsbasis
Atomare Konstrukte	Prädikate (eigentlich Atome im Sinne der Logikprogrammierung)	Primitive Aktionen
Komposition	Konjunktion, Disjunktion, Negation	Sequenz, Parallelität, überwachte Auswahl
Abstraktion	Klauseln	Prozeduren

3.3.1 Logikprogrammierung

Die Sita-Wissensbasis verwendet logik-orientierte Programmierung auf der Grundlage von Hornklauseln (erweitert um nicht-monotone Negation). Im Vergleich zu klassischer Logikprogrammierung und deren prominentestem Vertreter Prolog [CKPR73] unterscheidet sich Sita vor allem in der Auswertungsrichtung: bottom-up statt top-down. Der vorrangige Grund hierfür ist die geforderte Reaktivität: Änderungen im Datenbestand werden inkrementell vorwärts propagiert (um gegebenenfalls Auswahlwächter zu triggern). Eine top-down Auswertung müßte dagegen nach jeder Änderung der Basisfakten erneut Beweisversuche für die Menge der wartenden Wächter durchführen.

Ein weiterer wesentlicher Unterschied zu Prolog sind die sprachlichen Mittel zur Modifikation des Zustandes. In Sita stehen dazu im imperativen Part (also außerhalb des Logikprogrammes) entsprechende Aktionen bereit. Prolog dagegen bietet dazu die

speziellen Funktionen `assert` und `retract` an, die sich syntaktisch zwar als Prädikate in die Hornklauselprogramme einfügen, jedoch eine prozedurale, außerlogische Semantik besitzen. An dieser Stelle wird besonders deutlich, daß sich die Semantik eines Prolog-Programmes nicht ohne operationelle Anteile erklären läßt. Andere semantische Schwierigkeiten ergeben sich durch die Suchstrategie der Prologmaschine. Dadurch erhalten die Reihenfolge der Klauselaufschreibung und die Anordnung der Subziele innerhalb einer Klausel eine prozedurale Bedeutung, die aus logischer Sicht nicht vorhanden sein sollte.

Solche Probleme umgeht Sita dadurch, daß streng getrennt wird zwischen Logikprogramm mit rein deklarativer Semantik auf der einen Seite und prozeduralem, imperativen Programm auf der anderen Seite. So sind insbesondere die Aktionen zur Modifikation der Faktenbasis außerhalb der Logiksprache angesiedelt. Dadurch kann für den Logik-Teil eine „saubere“ (Fixpunkt-)Semantik ohne prozedurale Anteile verwendet werden.

Sita nimmt hier große Anleihen aus dem Bereich der deduktiven Datenbanken (vergleiche etwa [CGT90] oder [BMS96]), sowohl was die Formalisierung der Semantik betrifft, als auch beim Grundsatz der bottom-up Auswertung. Die Auswertungsrichtung begünstigt dabei die Reinheit der Semantik, was aber nicht unbedingt das ursprüngliche Motiv ihrer Anwendung ist. Denn wichtiger noch ermöglicht sie im Datenbankbereich die effiziente Auswertung großer Datenmengen. Und innerhalb von Sita führt die bottom-up Richtung zu reaktiver, datengetriebener Auswertung von Logikprogrammen, eine Eigenschaft, die mit klassischer Logikprogrammierung nur schwer zu erreichen ist.

3.3.2 Koordination

Die Ausgestaltung der prozeduralen Seite von Sita beschränkt sich zunächst auf die essentiellen Konstrukte einer Koordinationssprache. Dahinter steht die Idee der Trennung von Berechnung im Sinne datenverarbeitender Prozesse und Koordination im Sinne der Kommunikation und Steuerung dieser Prozesse, vergleiche [GC92].

Die Ausdrucksstärke der Sita-Wissensbasis läßt die Formulierung auch komplexerer Berechnungen rein als Logikprogramm zu. So kann etwa die Berechnung kürzester Wege in einem Graphen durch eine einzelne Wissensbasis-Anfrage ausgelöst werden. Dementsprechend kann sich die Rolle der prozeduralen Seite auf Koordinationsaspekte konzentrieren. Koordination meint hier Prozeßkontrolle, ermöglicht durch den oben beschriebenen Satz an Kompositionsoperatoren und die Abstraktionsmöglichkeit mittels Prozedurdefinitionen.

Ein weiterer Koordinationsaspekt ist die Kommunikation der prozeduralen Vorgänge untereinander. Hierfür stehen in Sita zwei Mittel zur Verfügung. Zum einen findet Kommunikation in der engen Koppelung sich aufrufender Prozeduren durch den

Gebrauch logischer Variablen als Argumente der Prozeduren statt. Je nach zu vereinbarendem Bindungsmuster können Argumente sowohl als Eingabestellen als auch zur Ergebnisrückgabe verwendet werden. Zum anderen kommunizieren Threads asynchron über die Wissensbasis als gemeinsamen Datenraum. Die Wissensbasis ist damit nicht nur Berechnungsstätte sondern auch Medium von zeitentkoppelter Kommunikation und Synchronisation im Dienste der Prozeßkoordination.

Die erwähnte Beschränkung der prozeduralen Sprache auf Koordinationskonstrukte ist jedoch im Sinne einer Minimalforderung zu sehen. Möglicherweise wird sie als Einschränkung empfunden, die einem höheren Programmierkomfort im Wege stehen. Für diesen Fall ist anzumerken, daß weitere prozedurale Konstrukte ergänzt werden können, ohne daß diese die grundsätzliche Architektur von Sita verletzen würden. Solche Konstrukte können Zuweisungen an lokale Variable und Schleifenkonstrukte (neben Rekursion) sein. Neben der überwachten Auswahl können zudem Wissensbasisanfragen ergänzt werden, die, anstatt auf die Erfülltheit eines Ausdruckes zu warten, direkt ein Anfageergebnis liefern.

3.3.3 Verwandte Programmiermodelle

Ähnlich wie Sita verwenden einige bekannte Programmiermodelle eine Aufspaltung in einen logik-orientierten Datenspeicher und eine prozedural orientierte Ablaufsteuerung. Dieser Abschnitt stellt Sita in Beziehung zu diesen Modellen.

Datenbankanwendungen sind ein Zusammenspiel aus klassischer, imperativer Programmierung (etwa in C++) und Datenbank-Update- und Query-Sprachen. Relationale Datenbanken bieten hier etwa SQL, deduktive Datenbanken ergänzen zumindest die Möglichkeit rekursiver Anfragen und verwenden meist Hornklausellogik als Anfragesprache. In beiden Fällen steht eine deklarative Datenbanksprache einem oder mehreren imperativen Prozessen gegenüber.
Anders als Sita sind Datenbanksysteme auf die persistente Verwaltung großer Datenmengen optimiert. Eine Anfrage an eine Datenbank bezieht sich zudem immer auf den aktuellen Zustand. In Sita dagegen steht die reaktive Auswertung von Anfragen im Vordergrund. Eine Anfrage kann dabei als Wächter über eine längere Zeitspanne aktiv bleiben und muß dem sich verändernden Wissensbasis-Zustand folgen. Dies verlangt nach neuartigen, inkrementellen Auswertungsmechanismen.

Produktionensysteme wie etwa OPS5 [For81] oder CLIPS [Cul89] verwenden Bedingungs-Aktions-Regeln, Produktionen genannt, die über einer veränderlichen Faktenbasis arbeiten. (Vergleiche hierzu auch Abschnitt 2.3.1.) Ein vorwärtsverkettender Auswertungsmechanismus befähigt Produktionensysteme zu reaktiver, ereignisgetriebener Programmierung. Dies stellt eine enge Verbindung zwischen Produktionensysteme und Sita her.

Wie bereits angesprochen bringen klassische Produktionensysteme einige Probleme mit sich:

1. Der Mechanismus zur Regelauswahl verwendet einen sehr einfachen Mustervergleich, der zu Lasten der Ausdrucksmächtigkeit geht. Insbesondere gibt es kein Abstraktionskonzept, mit dem man für zusammengesetzte Bedingungsausdrücke neue Namen vereinbaren könnte. Damit fehlt ein Strukturierungsmittel bei der Formulierung komplexerer Bedingungen. Vor allem aber schließt dies die Formulierung rekursiver Bedingungen aus.
2. Die Aktionsteile der Produktionen sind einfache Aktionssequenzen ohne irgendwelche Kontrollstrukturen. Jede Fallunterscheidung oder Schleifenbildung muß daher auf mehrere Produktionen aufgeteilt werden. Einen Ausführungskontext, der diese einzelnen Produktionen wieder zu einem prozeduralen Konstrukt zusammenbindet, muß der Anwender etwa in Form von Kontextfakten selbst verwalten.

Diesen Problemen entgeht Sita dadurch, daß statt einer flachen, unstrukturierten Regelmenge nun zwei ausdrucksstarke Formalismen verwendet werden, die die vermißten Strukturierungsmöglichkeiten sowohl bei der Bedingungssprache wie bei der Aktionssprache beinhalten.
Produktionensysteme zeichnen sich durch die Fähigkeit aus, das Eintreten einer Vielzahl von Situationen gleichzeitig zu überwachen. Durch das Konstrukt der überwachten Auswahl sowie durch den Einsatz von Nebenläufigkeit erhalten Sita-Programme die selbe Fähigkeit, bei zugleich wesentlich größerer Ausdrucksmächtigkeit und besserer Strukturierungsmittel.

Concurrent Constraint Programming (CCP) [Sar93] kombiniert als Programmierparadigma einen globalen Constraint-Speicher mit einem nebenläufigen, prozeduralen Kalkül, der über `ask`- und `tell`-Operationen mit dem Constraint-Speicher interagiert. Diese Operationen können mit den elementaren Sita-Aktionen zum Abfragen und Modifizieren der Wissenbasis verglichen werden.
Obwohl dem globalen Datenspeicher jeweils logik-basierte Formalismen zugrunde liegen, unterscheiden sich Sita-Wissensbasis und Constraint-Speicher in ihrer Struktur sehr deutlich. Eine wesentliche Differenz ergibt sich daraus, daß die `tell`-Operationen Informationen nur hinzufügen kann. In Sita dagegen können Basisfakten auch wieder gelöscht werden, so daß nicht-monotone Änderungen der Wissensbasis möglich sind.

Truth and Action Osmosis (TAO) [PV96] ist ein abstraktes Berechnungsmodell, das ebenfalls auf der Unterteilung des Zustandes in Datenbasis und Task beruht. Tasks sind in einem Prozeßkalkül formalisiert, der ähnlich zu dem von Sita aufgebaut ist. Bezüglich der Datenbasis gibt TAO jedoch keine konkrete Struktur vor, sondern geht von einem abstrakten Situationenraum aus, der über eine

Entailment-Relation definiert ist, und an den bestimmte Konsistenz- und Kohärenzbedingungen gestellt werden.
Der Formalismus der Sita-Wissensbasis erfüllt diese Bedingungen. Sita kann daher (mit kleinen Modifikationen) als konkrete Instanz des abstrakten TAO-Modells gesehen werden.

3.4 Sita als Agentenmodell

Wurde Sita bisher als Berechnungsmodell und Programmiersprache beschrieben, so ist es mit einem Wechsel der Terminologie auch möglich, Sita als eine Agentenarchitektur aufzufassen. Nach Shoham [Sho93] wird ein Hardware- oder Softwaresystem zu einem Agenten, sobald wir *mentale Konzepte* ([SC94]) benutzen, um den Zustand des Systems zu analysieren oder zu modellieren.

Das laufende Kapitel begann mit einer Betrachtung von Erkennen und Handeln. Ob dies schon mentale Begriffe in Shoham's Sinne sind, mag dahin gestellt sein. Man erhält jedoch eindeutig mentale Konzepte, wenn man die Begriffe der Sita-Architektur in die Sprache aktuell diskutierter Agenten-Architekturen übersetzt.

Programmiermodell	Agentenmodell
Fakten	Beliefs (Überzeugungen)
Threads	Intentionen
Logikprogramm	Situationsbibliothek
prozedurales Programm	Planbibliothek

Kontroll- und Datenfluß innerhalb des Agenten stellen sich dann wie folgt dar. Die Menge der Basisfakten ergibt sich aus der Wahrnehmung des Agenten einerseits und als Folge der bisherigen Handlungen andererseits. Basisfakten sind somit die Grundlage für das aktuelle Welt- und Selbstmodell des Agenten. Das Logikprogramm stellt eine domänenabhängige Situationsbibliothek dar. Mit ihrer Hilfe abstrahiert der Agent sein Wissen ausgehend von den Basisfakten. Dieser Vorgang analysiert die aktuelle Situation, baut ein abstraktes Weltmodell auf, aktiviert Zielsetzungen des Agenten und wählt adäquate Pläne aus. An dieser Stelle legt sich der Agent auf die Verfolgung bestimmter Pläne fest, indem die entsprechenden Prozeduren aus der Planbibliothek gestartet werden. Die Menge der aktuellen Threads repräsentieren die aktuellen Intentionen des Agenten.

Damit stellt sich Sita als eine reaktive und zielgerichtete Agentenarchitektur dar. Der Zustand des Agenten ist gegeben einerseits durch seine Beliefs, die das Wissen des

Agenten um sich und seine Umwelt repräsentieren, und andererseits durch seine Intentionen, also die Absichten, auf die der Agent durch die Verfolgung entsprechender Pläne eingegangen ist. Ähnliche funktionale Instrumentarien finden sich etwa in den Agentensprachen AgentSpeak(L) [Rao96], vergleiche Abschnitt 2.3.3, und 3APL [HdvM98]. Eine Programmiermethodik wie Sita steht immer im Spannungsfeld zwischen allgemeiner Anwendbarkeit und spezieller Designunterstützung. Sita hält hier die Balance, indem es gleichzeitig als Berechnungsmodell und als Agentenmodell erklärt werden kann.

Kapitel 4

Wissensbasis

Die Sprache der Wissensbasis wird syntaktisch und semantisch definiert. Sita bedient sich der aus der Literatur bekannten wohl-fundierten Semantik. Deren hier dargestellte Begrifflichkeit wird in Kapitel 6 Grundlage zur Beschreibung der Auswertungsverfahren sein. Besondere Aufmerksamkeit erfordert als nicht-monotones Konstrukt die Negation.

4.1 Syntax

Im folgenden wird die Syntax der Wissensbasis definiert. Dies umfaßt Schreibweisen für das statische Logikprogramm und den dynamischen Faktenspeicher sowie die Schnittstellen der Wissensbasis mit Update-Anweisungen, Anfragen und Anfrageergebnissen.

Der syntaktische Aufbau der Sita-Wissensbasis folgt im wesentlichen den bewährten Traditionen der Logikprogrammierung (vergleiche etwa [Llo87], [Ull88], [Ull89]), wobei wir die sonst üblichen Kommas weglassen und an einigen Stellen der Reihenfolge der Aufschreibung keine Bedeutung zumessen.

Zunächst werden Bezeichner für Konstanten, für Funktionen und Prädikate, sowie für Variablen benötigt. Dazu sei $\mathcal{C}$ eine vorgegebene Menge von Bezeichnern für einfache *Konstanten*. Als einfache Konstanten sind zumindest ganze Zahlen und Symbole als alphanumerische Zeichenketten beginnend mit einem Kleinbuchstaben vorhanden. Diese Symbole bilden zudem auch die Teilmenge $\mathcal{K} \subset \mathcal{C}$ der *Funktoren*, welche sowohl als Funktionsymbole wie auch als Prädikatssymbole verwendet werden können. Weiterhin gibt es eine Menge $\mathcal{V}$ von Bezeichnern für *Variablen*, die wie üblich alphanumerische Zeichenketten beginnend mit einem Großbuchstaben sind.

Im folgenden verwenden wir die kalligraphisch gesetzten Buchstaben ($\mathcal{C}$, $\mathcal{K}$, $\mathcal{V}$, etc.) sowohl als syntaktische Variablen für die bezeichneten Konstrukte als auch zur Be-

Variable	$\mathcal{V}$:	vorgegebene Menge	
Konstante	$\mathcal{C}$:	vorgegebene Menge	
Funktor	$\mathcal{K}$:	vorgegebene Menge, $\mathcal{K} \subset \mathcal{C}$	
Term	$\mathcal{T}$:	$\mathcal{V} \mid \mathcal{C} \mid \mathcal{K}(\mathcal{T} \ldots \mathcal{T})$	
Atom	$\mathcal{A}$:	$\mathcal{K}(\mathcal{T} \ldots \mathcal{T})$	
Literal	$\mathcal{L}$:	$\mathcal{A} \mid \neg\mathcal{A}$	
Regel	$\mathcal{R}$:	$\mathcal{A} \leftarrow \mathcal{L} \ldots \mathcal{L}.$	(bereichsbeschränkt)
Gerichtete Variable	$\mathcal{W}$:	**in** $\mathcal{V}$ $\mid$ **out** $\mathcal{V}$	
Prädikatsdeklaration	$\mathcal{D}$:	**pred** $\mathcal{K}(\mathcal{W} \ldots \mathcal{W})$	
Logikprogramm	$\mathcal{P}$:	$\mathcal{D} \mid \mathcal{R} \ \ldots \ \mathcal{D} \mid \mathcal{R}$	
Faktum	$\mathcal{F}$:	$\mathcal{A}$	(variablenfrei)
Faktenspeicher	$\mathcal{M}$:	$\{\mathcal{F}, \ldots \mathcal{F}\}$	
Update	$\mathcal{U}$:	**insert** $\mathcal{F}$ $\mid$ **delete** $\mathcal{F}$	
Update-Sequenz	$\mathcal{U}^*$:	$\mathcal{U} \ldots \mathcal{U}$	
Anfrage	$\mathcal{Q}$:	$\mathcal{L} \ldots \mathcal{L}$	
Substitution	Σ:	$[\mathcal{V}/\mathcal{T}, \ldots \mathcal{V}/\mathcal{T}]$	
Anfrageergebnis	$\mathcal{S}$:	$\{\Sigma, \ldots \Sigma\}$	(Grundsubstitutionen)

Abbildung 4.1: Syntaktische Elemente der Wissensbasis

zeichnung der damit konstruierten Mengen. Die syntaktischen Elemente sind in Abbildung 4.1 zusammengefaßt.

Es gibt einfache und zusammengesetzte *Terme* $\mathcal{T}$. Variablen und Konstanten sind einfache Terme. Ein Funktor $f \in \mathcal{K}$ und Terme $t_1, \ldots, t_n \in \mathcal{T}$ konstruieren einen zusammengesetzten Term $f(t_1 \ldots t_n)$. Ähnlich ergeben ein Funktor $p \in \mathcal{K}$ und Terme $t_1, \ldots, t_n \in \mathcal{T}$ eine *atomare Formel* $p(t_1 \ldots t_n)$, oder kurz ein *Atom*. Ein *Literal* $l \in \mathcal{L}$ ist ein Atom a oder dessen Negation $\neg a$. Ausdrücke der Sprache (Atome, Literale, etc.), die keine Variablen enthalten, heißen (naheliegenderweise) *variablenfrei*. Variablenfreie Atome heißen auch Grundatome.

Wenn a ein Atom ist und $l_1, \ldots, l_n$ Literale sind, dann ist $a \leftarrow l_1 \ldots l_n$. eine *Regel* (auch *Klausel* genannt). a ist der *Kopf* der Regel, die Literale sind der *Rumpf* der Regel. Der Rumpf wird als Konjunktion der angeführten Literale gelesen. Der Rumpf einer Regel kann leer sein. In diesem Fall läßt man in der Aufschreibung auch den Linkspfeil weg.

Es ist zu fordern, daß jede Regel *bereichsbeschränkt* ist: Jede Variable, die entweder im Kopf der Regel oder in einem negativen Literal im Rumpf der Regel vorkommt, kommt auch in einem positiven Literal im Rumpf der Regel vor. Diese Bedingung stellt sicher, daß aus variablenfreien Fakten stets wieder variablenfreie Fakten geschlußfolgert wer-

den.

Eine Menge von Regeln ergibt schließlich ein *Logikprogramm*, welches den intensionalen Teil einer Sita-Wissensbasis beschreibt. Zu beachten ist, daß der Rumpf einer Regel nicht als Liste sondern als *Menge* von Literalen aufgefaßt wird. Ähnlich ist ein Programm eine Menge von Regeln. Die Aufschreibung impliziert also keine Ordnung der Literale bzw. Regeln.

Die in einem Logikprogramm verwendeten Prädikate sind zudem in Prädikatsdeklarationen $\mathcal{D}$ erklären, wodurch die einzelnen Stellen eines Prädikates als Eingabe- oder Ausgabestellen markiert werden. Diese Unterscheidung wird erst in Abschnitt 4.3 eine Bedeutung erhalten, so daß wir Prädikatsdeklarationen vorerst ignorieren.

Der extensionale Teil einer Wissensbasis ist durch eine veränderliche Menge von Fakten bestimmt. Syntaktisch sind *Fakten* variablenfreie Atome. Wir bezeichnen diese Fakten-Mengen $m \in \mathcal{M}$ als *Faktenspeicher*. Dieser wird durch *Updates* modifiziert: Ein angegebenes Faktum wird entweder dem Faktenspeicher zugefügt oder aus ihm gelöscht. Formal leistet dies die *Update-Funktion*: Sei m ein Faktenspeicher und u ein Update-Ausdruck, dann ist $m \triangleright u$ der daraus resultierende Zustand. Hierbei ist es durchaus erlaubt, ein im Faktenspeicher nicht vorhandenes Faktum zu löschen; der Zustand bleibt dann unverändert. Die Update-Funktion ist somit eine vollständige und deterministische Funktion.

Definition 4.1

$$\begin{array}{rcl} \triangleright : \mathcal{M} \times \mathcal{U} & \rightarrow & \mathcal{M} \\ m \triangleright u & := & \begin{cases} m \cup \{f\} & \text{falls} \quad u = \textbf{insert} f \\ m \setminus \{f\} & \text{falls} \quad u = \textbf{delete} f \end{cases} \end{array}$$

Kanonische Erweiterung auf Update-Sequenzen:

$$\begin{array}{rcl} \triangleright : \mathcal{M} \times \mathcal{U}^* & \rightarrow & \mathcal{M} \\ m \triangleright v & := & \begin{cases} m & \text{falls} \quad v = \epsilon \\ (m \triangleright u) \triangleright v_1 & \text{falls} \quad v = <u> \circ v_1 \end{cases} \end{array}$$

$\triangleleft$

Die Syntax für den Rumpf einer Regel ist zugleich Syntax für Anfragen an die Wissensbasis. Eine *einfache Anfrage* ist ein Atom, eine *allgemeine Anfrage* ist eine Sequenz von Literalen. Die Unterscheidung von einfachen und allgemeinen Anfragen ist technischer Natur. Jede allgemeine Anfrage kann unter Zuhilfenahme einer entsprechenden neuen Regel als einfache Anfrage umformuliert werden. Andererseits wollen wir in der prozeduralen Sprache von Sita allgemeine Anfragen zulassen, ohne explizite Notation einer entsprechenden Hilfsregel. Deshalb erlauben wir an dieser Schnittstelle auch allgemeine Anfragen, die als abkürzende Schreibweise für eine einfache Anfrage plus Hilfsregel zu verstehen ist.

Da Anfragen im allgemeinen Variablen enthalten, ist ein *Anfrageergebnis* als Menge möglicher Variablenbelegungen zu formulieren. Zur Notation benötigen wir deshalb Variablensubstitutionen.

Eine *Substitution* ist eine endliche Menge von Variablenbindungen in der Form $[v_1/t_1, \ldots, v_n/t_n]$, wobei $v_i \in \mathcal{V}$ unterschiedliche Variablen und $t_i \in \mathcal{T}$ Terme sind, und zusätzlich $v_i \neq t_i$ gilt. Die Anwendung einer Substitution $\sigma \in \Sigma$ auf einen Term t wird mit $t\sigma$ notiert. Mit Σ wird die Menge aller Substitutionen bezeichnet, mit Σ_V die Menge all jener Substitutionen, die genau die Variablen $V = \{v_1, \ldots, v_n\}$ ersetzt. Im Falle, daß alle t_i variablenfrei sind, sprechen wir von einer Grundsubstitution.

Im Ergebnis einfacher Anfragen müssen alle Variablen gebunden werden. Das Ergebnis auf eine einfache Anfrage liefert Substitutionen, die angewandt auf das Anfrageatom diejenigen Fakten ergeben, die Grundinstanzen des Anfrageatoms sind und unter der (im nächsten Abschnitt definierten) Semantik wahr sind. Ein Ergebnis ist also eine Menge von Grundsubstitutionen über den in der Anfrage vorkommenden Variablen.

Die in einem Programm vorkommenden Prädikate lassen sich in drei Klassen einteilen. *Intensionale* Prädikate sind solche, die im Kopf einer Regel vorkommen. *Builtins* sind Prädikate, deren Semantik außerhalb des Programmes vorgegeben ist. Zu einer Implementierung von Sita gehört die Auswahl einer passenden Menge von Builtins. Schließlich gibt es die *extensionalen* Prädikate, die im Faktenspeicher verwaltet werden und per Updates modifiziert werden können.

Zur vereinfachten Darstellung nehmen wir ohne Beschränkung der Allgemeinheit an, daß diese drei Klassen von Prädikaten disjunkt sind. Desweiteren nehmen wir an, daß innerhalb eines Programmes jedes Prädikatssymbol in nur einer Stelligkeit verwendet wird. Damit sind Prädikate durch ihren Namen eindeutig gekennzeichnet.

Für jedes intensionale Prädikat p nennen wir die Menge der Regeln, in denen p im Kopf auftritt, die *Definition* von p.

Beispiel 4.2

Das Programm P berechnet die transitive Hülle `t` einer Relation `r`:

```
t(X Y) ← r(X Y).
t(X Y) ← r(X Z)  t(Z Y).
```

`t` ist ein intensionales Prädikat, `r` sei ein extensionales Prädikat.

Ausgehend von einem leeren Faktenspeicher ($m_0 = \emptyset$) werden durch eine Update-Sequenz u_1 zwei Fakten in die Relation `r` eingefügt:

$u_1 =$ `insert r(a b) insert r(b c)`

$m_1 = m_0 \triangleright u_1 = \{$`r(a b)`, `r(b c)`$\}$

Anschließend wird eine Anfrage q berechnet:

Anfrage $q =$ `t(a X)`
Antwort $s =$ {[X/b], [X/c]}
Antwort als Faktenmenge geschrieben: {`t(a b)`, `t(a c)`}

◁

4.2 Semantik

Zur Definition der Semantik für die Wissensbasis stützen wir uns auf die modelltheoretischen Verfahren, wie sie im Zusammenhang mit Logikprogrammierung und deduktiven Datenbanken entwickelt worden sind, siehe etwa [Dix96] oder [Ros91]. Exakter ausgedrückt verwendet Sita die *wohl-fundierte* Semantik, die dem Logikprogramm keine Einschränkungen hinsichtlich der Kombination von Rekursion und Negation auferlegt. Zusätzlich betrachten wir aber dennoch eingeschränkte Klassen von Logikprogrammen. Zum einen ergeben sich einfachere und intuitivere Semantiken, die innerhalb dieser Klassen mit der wohl-fundierten Semantik äquivalent sind. Zum anderen benötigen wir die hierzu einzuführenden Begriffe später bei der Betrachtung von Auswertungsalgorithmen in Kapitel 6.

In diesem Anschnitt ist mit dem Begriff Programm immer der Gesamtzustand einer Wissensbasis gemeint, also das statische Logikprogramm plus einer fixen Ausprägung des Faktenspeichers.

4.2.1 Programmklassen

Bei der nun folgenden Darstellung lehnen wir uns an die in [Kem92] an.

Schwierigkeiten bei der Findung intuitiver Semantiken macht vor allem die Negation [AB94]. Am einfachsten sind daher Programme ohne Negation.

Definition 4.3 Ein Programm heißt *definit*, wenn es keine negativen Literale enthält.

◁

Bei Programmen mit Negation sind Stratifikationsbedingungen ([ABW88], [Van86], [Naq86]) hilfreich, nach denen Negation nur außerhalb von Rekursionspfaden auftritt.

Definition 4.4 Der *Prädikat-Abhängigkeitsgraph* eines Programmes P ist ein gerichteter, markierter Graph, dessen Knoten die in P vorkommenden Prädikate sind. Für jede Regel $a \leftarrow l_1 \ldots l_n.$ und jedes darin enthaltene Literal l_i gibt es eine Kante vom Prädikat von l_i zum Prädikat von a. Die Kante wird mit *positiv* bzw. *negativ* gekennzeichnet, je nach dem ob l_i ein positives oder negatives Literal ist.

◁

Der Abhängigkeitsgraph kann nun auf Zyklen untersucht werden, die rekursiv definierte Prädikate kennzeichnen. Dazu wird er in *starke Zusammenhangskomponenten* (*SCC*[1]) zerlegt, also maximale Knotenmengen, innerhalb derer Pfade zwischen jedem Paar von Knoten existieren.

Definition 4.5 Eine SCC des Prädikat-Abhängigkeitsgraphen ist *positiv*, wenn es keine negativen Kanten innerhalb dieser SCC gibt. Ansonsten ist die SCC negativ. Eine SCC s_1 heißt tiefer als s_2, wenn es einen Pfad von einem Knoten in s_1 zu einem Knoten in s_2 gibt. $\triangleleft$

Definition 4.6 Eine *Stratifikation* eines Prädikat-Abhängigkeitsgraphen eines Programmes ist eine Partitionierung der Prädikate in eine geordnete Folge disjunkter Mengen $S_1, S_2, \ldots$ (genannt *Strata*) mit folgender Eigenschaft:
Für jedes Paar a_i, a_j von Prädikaten mit $a_i \in S_u$ und $a_j \in S_v$ gilt:
Falls eine positive Kante von a_i nach a_j führt, so ist $u \leq v$.
Falls eine negative Kante von a_i nach a_j führt, so ist $u < v$.

Ein Programm heißt *stratifizierbar*, wenn sein Prädikat-Abhängigkeitsgraph eine Stratifikation besitzt. $\triangleleft$

Lemma 4.7 [ABW88] Ein Programm ist genau dann stratifizierbar, wenn alle SCCs seines Prädikat-Abhängigkeitsgraphen positiv sind. $\triangleleft$

Stratifikationen eines stratifizierbaren Programmes erhält man zum Beispiel, indem man die partielle Ordnung der SCCs nach Definition 4.5 beliebig linearisiert.

Stratifizierbarkeit bedeutet Verzicht auf Negation innerhalb der zyklischen Abhängigkeit rekursiv definierter Prädikate, aber gleichzeitig Gewinn einer intuitiver zu definierenden Semantik und einfacherer Auswertungsalgorithmen. Es gibt jedoch weichere Bedingungen mit ähnlichen Vorteilen. Hier werden nicht die gegenseitigen Abhängigkeiten der Prädikate betrachtet, sondern die gegenseitigen Abhängigkeiten der Fakten. Wir definieren hier die lokal stratifizierbaren Programme, die von Przymusinski [Prz88] eingeführt wurden.

Definition 4.8 Das *Herbrand-Universum* $\mathcal{W}$ ist die Menge aller variablenfreien Terme.
Die *Herbrand-Basis* $\mathcal{B}$ ist die Menge aller Grundatome, die aus den Prädikatssymbolen und Argumenten aus dem Herbrand-Universum gebildet werden können.
Die Menge der *Grundinstanzen* einer Regel ergibt sich dadurch, daß für jede in der Regel vorkommende Variable alle Terme des Herbrand-Universums eingesetzt werden. $\triangleleft$

[1] nach dem englischen *strongly connected component*

Diese Begriffe beziehen sich auf die Signatur einer Sprache, also einer Menge von Konstanten-, Funktions- und Prädikatssymbolen. Im allgemeinen gehen wir dabei von der in Abschnitt 4.1 eingeführten Sprache aus. Manchmal will man die Sprache auf die in einem Programm P tatsächlich vorkommenden Symbole beschränken. Die entsprechende Herbrand-Basis wird mit $\mathcal{B}_P$ notiert.

Definition 4.9 Der *Atom-Abhängigkeitsgraph* eines Programmes P ist ein gerichteter, markierter Graph, dessen Knoten die Elemente der Herbrand-Basis sind. Für jede Grundinstanz $a \leftarrow l_1 \ldots l_n.$ aller Regeln und jedes darin enthaltene Literal l_i gibt es eine Kante von der positiven Variante des Literals l_i zum Atom a. Die Kante wird mit *positiv* bzw. *negativ* gekennzeichnet, je nach dem ob l_i ein positives oder negatives Literal ist. ◁

Definition 4.10 Eine *lokale Stratifikation* eines Atom-Abhängigkeitsgraphen eines Programmes ist eine Partitionierung der Atome der Herbrand-Basis in eine geordnete (und im allgemeinen unendliche) Folge disjunkter Mengen $S_1, S_2, \ldots$ mit folgender Eigenschaft:
Für jedes Paar a_i, a_j von Atomen mit $a_i \in S_u$ und $a_j \in S_v$ gilt:
Falls eine positive Kante von a_i nach a_j führt, so ist $u \leq v$.
Falls eine negative Kante von a_i nach a_j führt, so ist $u < v$.

Ein Programm heißt *lokal stratifizierbar*, wenn sein Atom-Abhängigkeitsgraph eine lokale Stratifikation besitzt. ◁

Lemma 4.11 [Prz88], [Kem92] Ein Programm ist genau dann lokal stratifizierbar, wenn alle SCCs seines Atom-Abhängigkeitsgraphen positiv sind, und weiterhin jeder Pfad, der negative Kanten enthält, einen initialen Knoten enthält (also nicht „von links her“ unendlich ist). ◁

Beide eingeführten Stratifizierbarkeitsbegriffe, global und lokal, stützen sich lediglich auf die Form der Regeln des untersuchten Programmes. Sie sind damit unabhängig von den Fakten des Programmes, also insbesondere auch unabhängig vom konkreten Zustand des Faktenspeichers.

In Abschnitt 6.8.1 werden wir eine noch schwächere Stratifikationsbedingung angegeben, die nur noch fordert, daß sich in den tatsächlich durchgeführten Ableitungen niemals ein Faktum auf seine eigene Negation abstützt. Allerdings kann diese Bedingung im allgemeinen nicht a priori sichergestellt werden.

Definition 4.12 Programme ohne Stratifikationsbedingung heißen *normale Programme*. ◁

Der Begriff der normalen Programme wurde eingeführt, um „herkömmliche“ Logikprogramme von disjunktiven Logikprogrammen zu unterscheiden, wo der Kopf von

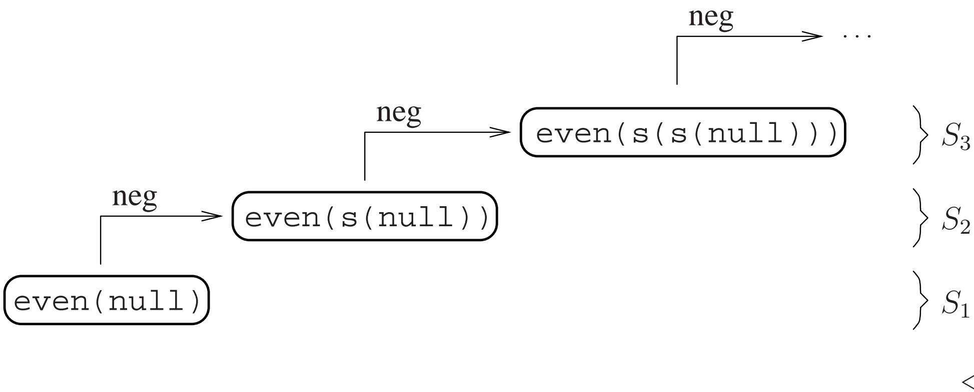
Regeln nicht aus einem einzigen Atom bestehen muß, sondern auch eine Disjunktion von Atomen sein kann. Vergleiche [LMR92].[2]

Normale Programme unterliegen lediglich der in Abschnitt 4.1 definierten Bereichsbeschränktheit.

Beispiel 4.13 Folgendes Programm ist nicht global stratifizierbar.

```
even(null).
even(s(X)) ← ¬ even(X).
```

Sein Atom-Abhängigkeitsgraph kann aber in Strata $S_1, S_2, \ldots$ zerlegt werden. Das Programm ist daher lokal stratifizierbar.



◁

Beispiel 4.14 Nicht lokal stratifizierbar ist folgendes Programm:

```
p(Y) ← ¬ p(X) q(X Y).
```

Zur Darstellung des Atom-Abhängigkeitsgraphen nehmen wir an, daß das Herbrand-Universum aus nur einem Element besteht, $\mathcal{W} = \{u\}$.

pos
p(u)
neg
q(u u)

Der Graph besitzt keine lokale Stratifikation. (Dies gilt für jedes nicht-leere Herbrand-Universum.)

Falls `q` aber eine nicht-zyklische Relation modelliert, so entstehen auch keine zyklischen Abhängigkeiten innerhalb der `p`-Fakten. Somit ist der bei der Auswertung tatsächlich realisierte Fakten-Abhängigkeitsgraph stratifizierbar. ◁

[2] Viele Verfahren für normale Programme lassen sich auch auf disjunktive Programme übertragen. So wurden etwa bottom-up Auswertungsalgorithmen für disjunktive deduktive Datenbanken vorgeschlagen [BL92].

4.2.2 Semantiken für stratifizierbare Programme

Modelltheoretische Semantiken, wie wir sie verwenden, liefern zu einem gegebenen Logikprogramm die Aussage, welche aller möglichen Fakten als wahr und welche als falsch zu interpretieren sind. Für definite Programme liefert das kleinste Herbrand-Modell[3] eine intuitive Semantik.

Definition 4.15 Eine Menge von Literalen I heißt *konsistent*, wenn es kein Atom a gibt, mit $a \in I$ und $\neg a \in I$.

Eine *partielle Interpretation* ist eine konsistente Menge von variablenfreien Literalen.

Eine *Interpretation* enthält jedes Atom der Herbrand-Basis, entweder positiv oder negiert. ◁

In den folgenden zwei Definition erhalten Regeln die beabsichtigte Bedeutung, daß sie von der Gültigkeit der Konjunktion der Rumpf-Literale auf die Gültigkeit des Kopf-Atoms schließen, und daß Variablen über die gesamte Implikation universell quantifiziert sind.

Definition 4.16 Eine (partielle) Interpretation *erfüllt* den Rumpf $l_1 \ldots l_n$ einer Regel, wenn es eine Belegung der im Rumpf vorkommenden Variablen gibt, so daß alle Literale l_i Elemente der Interpretation sind.

Eine (partielle) Interpretation *erfüllt* eine Regel, wenn entweder der Kopf der Regel (nach geeigneter Variablenbelegung) Element der Interpretation ist, oder wenn die Interpretation den Rumpf der Regel nicht erfüllt. ◁

Definition 4.17 Eine Interpretation heißt *Modell* (oder *totales Modell*) eines Programmes, wenn sie alle Regeln des Programmes erfüllt.

Eine partielle Interpretation heißt *partielles Modell*, wenn sie zu einem Modell erweitert werden kann. ◁

Ein totales Modell liefert eine *zwei-wertige* Semantik, in der jedes Grundatom entweder „wahr“ oder „falsch“ ist. Ein partielles Modell läßt einen dritten Wahrheitswert „undefiniert“ zu. In Abschnitt 4.2.3 wird das wohl-fundierte Modell beschrieben, das eine solche *drei-wertige* Semantik liefert.

Die Semantik von definiten Programmen wird im allgemeinen durch das kleinste Herbrand-Modell definiert. Hierzu fassen wir, im Gegensatz zu obiger Definition, Modelle als Mengen der wahren Fakten auf, also ohne explizite Angabe der im Modell falschen Atome.

[3]Wir sprechen im folgenden kurz von Modellen und Interpretationen, wenn Herbrand-Modelle und Herbrand-Interpretationen gemeint sind.

Lemma 4.18 Der Durchschnitt aller Modelle eines definiten Programmes ist wieder Modell des Programmes (und heißt daher kleinstes Modell). ◁

Alle Atome, die nicht im kleinsten Modell sind, werden als falsch interpretiert. Dies entspricht der Anwendung der *closed world assumption*, die von Reiter in [Rei78] präsentiert wird.

Die Menge der wahren Fakten kann alternativ durch eine *Fixpunktiteration* angegeben (und berechnet) werden. Iteriert wird der Operator T_p, der die *unmittelbaren Konsequenzen* eines Programmes P formalisiert (und der von van Emden und Kowalski [VK76] in der Frühzeit der Logikprogrammierung eingeführt wurde).

Definition 4.19 Für jedes definite Logikprogramm P definiere T_P folgende Abbildung:

$$\begin{array}{rcl} T_P : 2^{\mathcal{B}_P} & \rightarrow & 2^{\mathcal{B}_P} \\ T_P(I) & := & \{a \mid a \leftarrow l_1 \ldots l_n \text{ ist Grundinstanz einer Regel in } P \\ & & \quad \text{und es gilt } l_i \in I \text{ für alle } i = 1..n\} \end{array}$$

◁

Für monotone Operatoren, die Teilmengen der Herbrand-Basis $\mathcal{B}$ auf Teilmengen der Herbrand-Basis abbilden, definieren wir folgende Schreibweisen (vergleiche [Kem92]):

Definition 4.20 Sei $G : 2^{\mathcal{B}} \rightarrow 2^{\mathcal{B}}$ monoton und $\alpha \in \mathbb{N}$.

$$\begin{array}{rcl} G \uparrow 0 & := & \emptyset \\ G \uparrow \alpha & := & G(G \uparrow (\alpha - 1)) \\ G \uparrow \infty & := & \sup\{G \uparrow \beta \mid \beta \in \mathbb{N}\} \\ G \downarrow 0 & := & \mathcal{B} \\ G \downarrow \alpha & := & G(G \downarrow (\alpha - 1)) \\ G \downarrow \infty & := & \inf\{G \downarrow \beta \mid \beta \in \mathbb{N}\} \end{array}$$

◁

Der Operator T_P ist bezüglich Mengeninklusion monoton und stetig. Daraus folgt, daß ein kleinster Fixpunkt $\text{lfp}(T_P)$ existiert und per Iteration konstruiert werden kann. Er stimmt mit dem kleinsten Herbrand-Modell überein (Beweis z.B. in [Llo87]):

Lemma 4.21 Kleinstes Herbrand-Modell von $P = \text{lfp}(T_P) = T_P \uparrow \infty$ ◁

Varianten dieser Fixpunktiteration bilden den Kern aller bottom-up Auswertungsverfahren, so auch die inkrementellen Algorithmen der Sita-Wissensbasis, die in Kapitel 6 entwickelt werden.

Nicht-definite Programme enthalten mit der Negation ein nicht-monotones Konstrukt. Ein entsprechender unmittelbarer Folgerungsoperator verliert mit der Monotonie im allgemeinen auch seinen kleinsten Fixpunkt, und entsprechend gibt es kein eindeutiges kleinstes Herbrand-Modell.

In stratifizierbaren Programmen kann man sich leicht behelfen, indem man das Programm Stratum für Stratum auswertet. Innerhalb eines Stratums stört keine Negation, hier kann man bis zum Fixpunkt iterieren. Die dabei auftretenden negativen Literale gehören zu bereits ausgewerteten, tieferen Strata, und werden dann entsprechend der closed world assumption interpretiert.

Die Intuition hinter der Stratifikation ist es, Regeln als *gerichtete* logische Aussagen zu sehen, und nicht etwa als allgemeine prädikatenlogische Ausdrücke, wo $p \leftarrow \neg q$ äquivalent zu $p \vee q$ und zu $q \leftarrow \neg p$ wäre. Dieser Gedanke wurde formal in der *perfect-model* Semantik gefaßt (siehe [Prz88]), die ein mathematisches Modell für die erwähnte Strata-weise Berechnung darstellt.

4.2.3 Wohl-fundierte Semantik

Für normale Programme wird in [vRS91] die *well-founded semantics* vorgeschlagen (in dieser Arbeit als *wohl-fundierte Semantik* bezeichnet), die in der Literatur großes Echo findet.

Wir wählen die wohl-fundierte Semantik für die Sita-Wissensbasis aus folgenden Gründen:

- Sie benötigt keine Stratifikationsbedingungen.
- Sie liefert eindeutige Modelle (im Gegensatz etwa zur *stable-model Semantik* [GL88]).
- Sie ist in polynomieller Zeit zu berechnen (ebenfalls im Gegensatz zur stable-model Semantik).
- Sie ist verträglich mit den im Abschnitt 4.2.2 genannten Semantiken. Das heißt, auf lokal stratifizierbaren Programmen liefert die perfect-model Semantik und die wohl-fundierte Semantik das gleiche Modell.
- Sie ist im Bereich der deduktiven Datenbanken allgemein als Standard anerkannt und findet in letzter Zeit auch innerhalb der klassischen Logikprogrammierung Beachtung [CW93], [RSS$^+$97].

Der Nachteil, ein im allgemeinen dreiwertiges Modell zu liefern, scheint sich kaum vermeiden zu lassen. Zumindest ist keine allgemein anerkannte Semantik bekannt, die ein eindeutiges zweiwertiges Modell liefern würde.

Wir definieren das wohl-fundierte Modell gemäß [Van93] mit Hilfe alternierender Fixpunkte. Die Idee ist, die Interpretation negativer Literale zunächst festzuhalten, um über das dann definite Programm einen ersten Fixpunkt zu berechnen. Dieser Fixpunkt liefert für den nächsten Schritt wiederum eine fixe Interpretation der negativen Literale des ursprünglichen Programms. Dies wird solange wiederholt, bis die jeweiligen Fixpunkte selbst gegen einen äußeren Fixpunkt konvergieren.

Zur formalen Definition erweitern wir zunächst den unmittelbare-Konsequenzen-Operator:

Definition 4.22 Für jedes normale Logikprogramm P und jede Menge A von Grundatomen aus $\mathcal{B}_P$ definiere T_P^A folgende Abbildung:

$$\begin{array}{rcl} T_P^A : 2^{\mathcal{B}_P} & \rightarrow & 2^{\mathcal{B}_P} \\ T_P^A(I) & := & \{a \mid a \leftarrow l_1 \dots l_n \text{ ist Grundinstanz einer Regel in } P \\ & & \quad \text{und für alle positiven } l_i \text{ gilt: } l_i \in I \\ & & \quad \text{und für alle negativen } l_i = \neg b_i \text{ gilt: } b_i \notin A \\ & & \quad (i = 1..n)\} \end{array}$$

◁

Der Parameter A des Operators bestimmt den Wahrheitswert der negierten Literale: $\mathcal{B}_P - A$ sind die im vorhergehenden Schritt der äußeren Iteration berechneten falschen Atomen.

Die inneren Iterationen werden nun als neuer Operator F_P geschrieben:

Definition 4.23

$$\begin{array}{rcl} F_P(A) & := & T_P^A \uparrow \infty \\ F_P^2(A) & := & F_P(F_P(A)) \end{array}$$

◁

F_P ist anti-monoton[4]. F_P^2 ist daher monoton und besitzt kleinsten und größten Fixpunkt. Der kleinste Fixpunkt gibt die wahren Atome des wohl-fundierten Modells an, der größte enthält zusätzlich die Atome mit dem Wahrheitswert „undefiniert“. Das Komplement des größten Fixpunkts sind also die Atome mit dem Wahrheitswert „falsch“.

Definition 4.24

$$\begin{array}{rcl} A_P^+ & := & lfp(F_P^2) \\ A_P^- & := & \mathcal{B} - gfp(F_P^2) \end{array}$$

◁

[4] Das heißt: $A_1 \subset A_2 \;\Rightarrow\; F_P(A_1) \supset F_P(A_2)$

Lemma 4.25 [Van93] Für jedes normale Programm P ist die Interpretation

$$A_P^* := A_P^+ \cup \{\neg a \mid a \in A_P^-\}$$

ein partielles Modell. Es heißt wohl-fundiertes Modell. ◁

In Kapitel 6 werden die Eigenschaften dieser verschachtelten Fixpunktiteration genauer behandelt.

Mit dieser Semantik ausgestattet können wir nun das Ergebnis von Anfragen an eine Sita-Wissensbasis formal definieren:

Definition 4.26 In einer Sita-Wissensbasis mit Logik-Programm P sei m ein Faktenspeicher und q eine Anfrage. Zudem sei V die Menge der in q vorkommenden Variablen. Dann ist das Ergebnis $\mathrm{ans}(P, m, q)$ wie folgt definiert:

1. Falls q eine einfache Anfrage ist (ein einzelnes Atom):

$$\mathrm{ans}(P, m, q) := \{\sigma \in \Sigma_V \mid q\sigma \in A^+_{P \cup m}\}$$

2. Falls q als allgemeine Anfrage eine Menge von Literale $l_1, \ldots, l_n$ ist:
   Es sei q' ein Atom bestehend aus einem frischen Prädikatssymbol und den Variablen V als Argumenten.

$$\begin{aligned} \mathrm{ans}(P, m, q) &:= \{\sigma \in \Sigma_V \mid q'\sigma \in A^+_{P' \cup m}\} \\ \text{mit} \quad P' &= P \cup \{q' \leftarrow l_1 \ldots l_n.\} \end{aligned}$$

◁

Alternativ zur Darstellung von Ergebnissen als Mengen von Substitutionen werden wir Ergebnisse auch mit den entsprechenden Faktenmengen repräsentieren.

Diese Definition läßt auch bei einem echt drei-wertigen Modell nur die Fakten mit dem Wahrheitswert „wahr" in das Ergebnis einfließen. Atome mit dem Wahrheitswert „undefiniert" werden im Ergebnis nicht berücksichtigt. Wir begründen dieses Verhalten mit folgenden zwei Überlegungen:

- Das wohl-fundierte Modell liefert den Wahrheitswert „undefiniert" nur für solche Fakten, die sich auf ihre eigene Negation begründen. Es erscheint nicht intuitiv, solche Fakten als Folgerungen der durch Programm und Faktenspeicher gegebenen Informationen anzusehen.

- Durch die Gleichbehandlung von undefinierten und falschen Fakten entsteht kein echter Informationsverlust. Zwischen diesen Wahrheitswerten kann unterschieden werden, indem die Negation der interessierenden Fakten betrachtet wird. Dort wird ein falsches Faktum wahr, während ein undefiniertes Faktum undefiniert bleibt. Die Ausdrucksstärke ist somit nicht eingeschränkt.

4.3 Magic-Set Transformation

Eine bottom-up Auswertung berechnet grundsätzlich das gesamte Modell des Logikprogrammes bezüglich eines Faktenspeichers, unabhängig davon, welche Berechnungen zur Beantwortung der aktuellen Anfragen wirklich notwendig sind. Im Vergleich dazu geht eine top-down Auswertung in Form der SLD-Resolution[5] [Llo87] zielgerichtet vor, da nur der Teil des Modells betrachtet wird, der zur Beantwortung einer Anfrage benötigt wird. Die naive Anwendung der bottom-up Auswertung kann daher einen großen Effizienzverlust mit sich bringen. Darüber hinaus kann eine effektive Auswertung unmöglich werden, wenn die Extensionen mancher Prädikate unendlich sind, obwohl der zur Beantwortung der Anfragen benötigte Teil endlich und durchaus effektiv zu berechnen wäre.

An dieser Stelle setzt die *magic-set Transformation* an, die in [BMSU86] zum ersten mal beschrieben wurde und seitdem viel Aufmerksamkeit im Bereich der deduktiven Datenbanken erhalten hat, vergleiche etwa [BR87b] und [Ull89]. Wegen ihrer großen Bedeutung auch für Sita-Programme erläutern wir diese Technik im folgenden kurz.

Die Grundidee ist die Unterscheidung von Eingabe- und Ausgabestellen eines Prädikats, das damit einen gewissen prozeduralen Charakter erhält: Es werden nicht mehr alle Elemente der entsprechenden Relation berechnet, sondern nur noch diejenigen, die einer bestimmten Menge von Eingabewerten entsprechen. Diese Sichtweise ist etwa auch bei der Prolog-Programmierung üblich und ergibt sich zumeist intuitiv aus der intendierten Semantik eines Prädikats, bei der man nicht nur an eine bloße Relation sondern zumeist auch an eine (nicht-deterministische) Funktion denkt.

In einem Sita-Programm wird daher jedes Prädikat in einer Deklaration mit einem Bindungsmuster versehen, das die einzelnen Stellen als Eingabe- oder Ausgabestellen ausweist. Die Syntax dieser Deklarationen wurde bereits in der Abbildung 4.1 auf Seite 32 dargestellt.

Auf der Basis der Bindungsmuster muß der Compiler nun für jede Regel des Programmes eine *SIP-Strategie*[6] finden, die den Datenfluß innerhalb der Regel beschreibt[7]: Jede Eingabestelle eines Literals im Regelrumpf sowie jede Ausgabestelle im Regelkopf muß mit einem Wert versorgt werden. Dieser Wert kann von einer Eingabestelle im Kopf oder einer Ausgabestelle eines Literals im Rumpf geliefert werden. Nur wenn sich ein Datenflußgraph finden läßt, bei dem die Rumpf-Literale nicht zyklisch voneinander abhängen, wird die Regel vom Compiler als korrekt akzeptiert.

Des weiteren wird nun jedem Prädikat ein zusätzliches *magic*-Prädikat zur Seite gestellt, das soviele Stellen besitzt, wie das Hauptprädikat Eingabestellen hat. In diesen

[5] „Selected Linear Resolution for Definite Clauses“
[6] „sideway-information-passing“, [BR87b]
[7] Wir beschränken uns hier auf definite Regeln.

neuen Prädikaten werden die Eingabewerte vermerkt, für die die Hauptprädikate berechnet werden sollen. Anschließend wird jede Regel dergestalt umgeschrieben, daß sich ihre SIP-Strategie in einer entsprechenden Verwendung der magic-Prädikate widerspiegelt. Wir demonstrieren dies an einem Beispiel:

Beispiel 4.27

```
t(X Y) ← r(X Y).
t(X Y) ← r(X Z)  t(Z Y).
```

Dieses bereits in in Beispiel 4.2 auf Seite 34 verwendete Programm berechnet die transitive Hülle `t` einer Relation `r`. Wir nehmen nun an, daß `t` nur für bestimmte Startpunkte berechnet werden soll, daß also die erste Stelle von `t` eine Eingangsstelle ist. Auch die erste Stelle von `r` kann dann als Eingangsstelle deklariert werden:

```
pred t(in X out Y)
pred r(in X out Y)
```

Für die beiden Regeln ergeben sich einfache SIP-Strategien:

```
t(X Y) ← r(X Y).

t(X Y) ← r(X Z)  t(Z Y).
```

Nun kann das Programm transformiert werden. Dazu werden zunächst die Rümpfe der Regeln um die dem Kopf entsprechenden magic-Prädikate erweitert. Dies liefert die gewünschte Einschränkung der Relationen:

```
t(X Y) ← magic_t(X)  r(X Y).
t(X Y) ← magic_t(X)  r(X Z)  t(Z Y).
```

Zusätzlich muß in die magic-Prädikate eingetragen werden, für welche Werte ihre Hauptprädikate zu berechnen sind. Für jede Verwendung eines Hauptprädikates im Rumpf einer Regel ist dazu eine weitere Regel zu ergänzen:

```
magic_r(X) ← magic_t(X).
magic_r(X) ← magic_t(X).
magic_t(Z) ← magic_t(X)  r(X Z).
```

(Das erhaltene transformierte Programm kann zumeist noch optimiert werden. So ist hier von den beiden identischen Regen natürlich nur eine notwendig.)

Zuletzt muß noch (durch den Anfragesteller) in `magic_t` vermerkt werden, für welche Startpunkte die transitive Hülle berechnet werden soll.

Das Modell des transformierten Programmes beschreibt dann nur noch die von den angegebenen Startpunkten ausgehenden Teile des durch `r` aufgespannten

Graphen. Die anschließende bottom-up Auswertung berechnet somit keine für das Anfrageergebnis irrelevanten Tupel. ◁

Dem transformierten Programm muß mitgeteilt werden, welche Anfragen zu berechnen sind. Diese ergeben sich aus den überwachten Auswahlen der Handlungsbasis (vergleiche Abschnitt 5.3.1). Für jeden Zweig einer überwachten Auswahl in der Handlungsbasis ergibt sich eine Anfragerelation in der Wissensbasis. Die magic-Varianten dieser Anfrageprädikate werden im Programmlauf mit den Werten versorgt, die sich aus den Ausführungskontexten derjenigen Threads ergeben, welche in einer überwachten Auswahl auf ein Anfrage-Ergebnis warten.

Die Technik der magic-set Transformation sowie verwandter Verfahren wird in der Literatur ausführlich beschrieben, siehe etwa [BR87b], [Ull89]. Die Anwendbarkeit der Transformation auf Programme mit Negation wird in [Kem92] und [KSS95] untersucht. Dort werden Bedingungen an die SIP-Strategien aufgezeigt, unter denen die Transformation die wohl-fundierte Semantik des Programmes (bezogen auf die Anfrage) erhält. Die technische Durchführung im Rahmen eines Sita-Compilers wird in [Per97] aufgezeigt.

Die bottom-up Auswertung eines magic-set-transformierten Programmes simuliert in gewisser Weise die SLD-Resolution einer top-down Auswertung. Insbesondere werden dabei nicht mehr Fakten berechnet als diejenigen, die während einer SLD-Resolution im Ableitungsbaum vorkommen. Aus dieser Sicht ist die bottom-up Auswertung höchstens so aufwendig wie die top-down Auswertung. Eine genaue Darstellung des Zusammenhangs wird in [Ull89] gegeben.

Zusammenfassend läßt sich sagen, daß die magic-set Transformation die Vorteile der bottom-up Auswertung (Reaktivität, inkrementelle Auswertung, deklarative Semantik) mit der Zielgerichtetheit der top-down Auswertung verbindet.

Kapitel 5

Handlungsbasis

Die Handlungsbasis verwendet einen Prozeßkalkül, dessen Syntax und Semantik im folgenden beschrieben wird. Wir geben zudem einige mögliche Erweiterungen des Kalküls an.

5.1 Syntax

Wir erweitern nun die in Abschnitt 4.1 eingeführte Syntax um die prozeduralen Anteile des Berechnungsmodells. Hierbei übernehmen wir einige der bereits in Abbildung 4.1 definierte Konstrukte: Konstante, Atom, Funktor, Term, gerichtete Variable und Regel. Die neuen Konstrukte sind in Abbildung 5.1 dargestellt.

Anweisungen werden in Form von Task-Ausdrücken, oder kurz *Tasks*, notiert. Es gibt primitive Tasks und zusammengesetzte Tasks. Zu den primitiven Tasks zählen die Anweisungen zum *Update* der Wissensbasis, die Anweisung zum *Prozeduraufruf* sowie die Anweisung, dem ausführenden Thread eine bestimmte *Priorität zuzuweisen*.

Für zusammengesetzte Tasks stehen drei Kompositionskonstrukte zur Verfügung. Die *sequentielle Komposition* wird durch hintereinanderschreiben der auszuführenden Tasks notiert. Die *überwachte Auswahl* wird durch die **when**-Anweisung geschrieben, die aus einem oder mehreren alternativen Zweigen besteht. Jeder Zweig besitzt eine Wissensbasis-Anfrage als Wächter und einen Task-Ausdruck als Rumpf. Zuletzt gibt es die *parallele Komposition* in Form der **par**-Anweisung, die eine oder mehrere Rumpf-Tasks enthält.

Beliebige Task-Ausdrücke können als *Prozeduren* definiert werden, die mit einem Namen und einer Liste von Variablen als formale Parameter ausgewiesen werden. Ähnlich wie Prädikate erhalten Prozeduren *Bindungsmuster*, indem jede Parameterstelle als Eingabe- oder Ausgabeparameter markiert wird.

Der prozedurale Anteil eines Sita-Programmes besteht dann aus einer Menge von Prozedurdefinitionen. Jedes Programm sollte eine Prozedur mit Namen `main` ohne Para-

Bereits eingeführte syntaktische Elemente (siehe Abbildung 4.1):

Atom	$\mathcal{A}$	
Funktor	$\mathcal{K}$	
Term	$\mathcal{T}$	
Gerichtete Variable	$\mathcal{W}$	
Anfrage	$\mathcal{Q}$	
Prädikatsdeklaration	$\mathcal{D}$	
Regel	$\mathcal{R}$	
Primitiver Task	$\mathcal{J}$:	
(Update-Task)		**insert** $\mathcal{A}$
	$\mid$	**delete** $\mathcal{A}$
(Prozeduraufruf)	$\mid$	$\mathcal{K}(\mathcal{T} \dots \mathcal{T})$
(Prioritätsanweisung)	$\mid$	**priority** $\mathcal{T}$
Task	$\mathcal{I}$:	
(primitiver Task)		$\mathcal{J}$
(Sequenz)	$\mid$	$\mathcal{I} \dots \mathcal{I}$
(überwachte Auswahl)	$\mid$	**when** $\mathcal{Q} \Rightarrow \mathcal{I}$
		[] $\mathcal{Q} \Rightarrow \mathcal{I}$
		$\vdots$
		end
(parallele Komposition)	$\mid$	**par** $\mathcal{I}$
		[] $\mathcal{I}$
		$\vdots$
		end
Prozedurdefinition	$\mathcal{N}$:	**proc** $\mathcal{K}(\mathcal{W} \dots \mathcal{W})$
		$\mathcal{I}$
		end
Sita-Programm	$\mathcal{Z}$:	$\mathcal{D}\vert\mathcal{R}\vert\mathcal{N} \; \dots \; \mathcal{D}\vert\mathcal{R}\vert\mathcal{N}$

Abbildung 5.1: Syntaktische Elemente der Handlungsbasis

meter haben. Mit ihr beginnt bei Programmstart die Abarbeitung.

Deklarative und prozedurale Anteile können gemischt notiert werden, so daß ein Sita-Programm insgesamt eine Menge von Prädikatsdeklarationen, Prädikatsdefinitionen (in Form von Regeln) und Prozedurdefinitionen ist.

5.2 Verwendung von Variablen

Zugunsten einer konzeptionellen Kontinuität wird die Semantik von Variablen in Prozeduren möglichst ähnlich zu der von Variablen in Klauseln gehalten. Insbesondere sind Variablen als Platzhalter aufzufassen, die beim ersten Auftreten an einen Wert gebunden werden. Es gibt also keine explizite Zuweisungsoperation, wie sonst in imperativen Sprachen üblich.[1]

Wie in Klauseln auch sind Variablen in Prozeduren nicht zu deklarieren. Der Sichtbarkeitsbereich einer Variable ist immer die gesamte Prozedur.

Der Datenfluß innerhalb einer Prozedur wird durch Bindungsmuster gesteuert, die für Prädikate und Prozeduren gleichermaßen deklariert werden. Durch die Bindungsmuster ergibt sich für jedes Auftreten einer Variable eine festgelegte Rolle: Entweder die Variable wird *beschrieben*, nimmt also ein Datum auf, oder sie wird *gelesen*, liefert also ein Datum.

Hier ergeben sich zwei Bedingungen, die für jeden möglichen sequentialisierten Ablauf einer Prozedur erfüllt sein müssen: Bevor eine Variable gelesen wird, muß sie beschrieben sein; und eine Variable darf nur einmal beschrieben werden. Die erste Bedingung garantiert die Ausführbarkeit einer Sequenz in der gegebenen Reihenfolge, da der Datenfluß immer vorwärts gerichtet ist. Die zweite Bedingung vermeidet implizite Gleichheitstests, die neben der überwachten Auswahl ein weiteres Kontrollkonstrukt zur Fallunterscheidung notwendig machen würden.

Daraus ergibt sich für eine Variable an jeder Stelle der Prozedur einer von drei möglichen Bindungszuständen: Vor einem beschreibenden Auftreten ist sie *ungebunden*; dort, wo sie bereits sicher beschrieben ist, ist sie *gebunden*; falls sie in Abhängigkeit einer überwachten Auswahl möglicherweise beschrieben ist, nennen wir sie *gesperrt*.

Mit diesen Begriffen ausgestattet lassen sich die oben geforderten Bedingungen als Regeln für die einzelnen syntaktischen Konstrukte formulieren:

Prozedurkopf: Variablen auf Eingabestellen sind mit Beginn der Prozedur gebunden. Variablen auf Ausgabestellen sind zunächst ungebunden und müssen im Rumpf der Prozedur gebunden werden.

[1] In imperativen Sprachen werden Variablen zumeist als Container für veränderliche Werte gesehen. Diese Rolle übernimmt in Sita die Wissensbasis. Aus dieser Sicht können die Update-Tasks als Zuweisungsoperationen aufgefaßt werden.

Primitive Tasks: Variablen in Update-Tasks und Prioritätsanweisungen müssen bereits gebunden sein. In Prozeduraufrufen müssen Variablen, die in Eingabeparametern vorkommen, bereits gebunden sein. Als Ausgabeparameter müssen Variablen eingesetzt werden, die zunächst ungebunden sein müssen. Sie werden durch den Aufruf gebunden. Diese Regeln für Prozeduraufrufe gelten in gleicher Weise für die Wissensbasisabfragen der Wächter.

Sequenz: Eine durch einen Task einer Sequenz gebundene Variable ist auch im Rest der Sequenz gebunden und darf zudem im Rest nicht nochmal gebunden werden. Die gleiche Regel gilt auch für die „Sequenz“ aus Abfrage und Rumpf-Task in den Zweigen einer überwachten Auswahl.

Überwachte Auswahl: Eine bereits gebundene Variable ist auch in allen Zweigen der Auswahl gebunden. Eine Variable, die in allen Zweigen der Auswahl gebunden wird, wird auch durch die gesamte Auswahl gebunden. Wird dagegen eine Variable in einigen Zweigen gebunden, in anderen aber nicht, so gilt die Variable bezogen auf die gesamte Auswahl als gesperrt. Eine gesperrte Variable darf weder vor der Auswahl bereits gebunden sein, noch darf sie nach der Auswahl gebunden oder gelesen werden.

Parallele Komposition: Eine bereits gebundene Variable ist auch in allen parallelen Zweigen gebunden. Wird eine bisher ungebundene Variable in einem Zweig gebunden, so darf sie in den übrigen Zweigen weder gebunden noch gelesen werden.

Durch die letztgenannte Bedingung wird zum einen ein Wettlauf nebenläufiger Prozesse um das Binden einer Variable vermieden. Zum anderen werden Variablen als Synchronisations- und Kommunikationsmittel zwischen nebenläufigen Prozessen ausgeschlossen. Diese Funktionalitäten sind der Kommunikation über die Wissenbasis vorbehalten.

5.3 Semantik

Zur Klärung der Semantik der Handlungsbasis werden die folgenden Fragestellungen behandelt: Die Ausführung der überwachten Auswahl, ein Mechanismus zur Koordination nebenläufiger Zugriffe auf die Wissensbasis, das Scheduling der Threads, sowie die Integration von Builtins.

5.3.1 Überwachte Auswahl

Wenn ein Thread eine überwachte Auswahl ausführt, werden die als Wächter gegebenen Anfragen an die Wissensbasis gestellt. Die Ausführung des Threads wird minde-

stens solange suspendiert, bis wenigstens eine der Anfragen ein nicht-leeres Ergebnis liefert. Die Ausführung des Threads wird mit dem obersten Zweig der Auswahl wieder aufgenommen, dessen Anfrageergebnis nicht leer ist; der Auswahlzweig wird *getriggert*. Aus dem Anfrageergebnis (vergleiche Definition 4.26, Seite 43) wird ein beliebiges Tupel ausgewählt und zur Bindung der Ausgangsvariablen des Anfrageausdruckes verwendet. Die Ausführung wird sodann mit dem Rumpf des Zweiges fortgesetzt. Sobald der Rumpf-Task abgearbeitet ist, ist auch die Ausführung der überwachten Auswahl beendet.

Die Festlegung, bei Wiederaufnahme den obersten der möglichen Zweige zu wählen, ermöglicht es, auch auf leere Anfrageergebnisse zu reagieren, wie in folgendem Beispiel:

```
when q(X)    ⇒ insert r(have_q(X))
  [] true() ⇒ insert r(empty_q)
end
```

Die Bedingung im zweiten Zweig ist immer erfüllt. Dennoch wird dieser nur gewählt, wenn das Prädikat q leer ist.

5.3.2 Koordinierter Wissensbasis-Zugriff

In Gegenwart nebenläufiger Prozesse wird immer ein Konzept zum geschützten Zugriff auf gemeinname Ressourcen benötigt. In Sita ist die Wissensbasis eine gemeinsame Ressource, für die domänenabhängige Konsistenzregeln einzuhalten sind. Aus diesem Grunde soll es einem Thread möglich sein, eine Wissensbasisanfrage mit anschließenden Updates zu einer atomaren Aktion zu koppeln.

Wir führen hierzu kein neues syntaktisches Konstrukt ein, sondern erweitern die Semantik der überwachten Auswahl um folgende Regel: Nach Triggern eines Auswahlzweiges werden alle direkt darauf folgenden Update-Anweisungen auf jenen Wissensbasiszustand angewendet, der zum Triggern des Zweiges geführt hat, also bevor ein anderer Thread Zugriff auf die Wissensbasis erhält.

Diese Regel wird dadurch realisiert, daß ein Thread ausschließlich an folgenden Stellen unterbrochen werden kann: Bei Eintritt in eine überwachte Auswahl, zu Beginn oder Ende einer parallelen Komposition, und nach Ausführung einer Prioritätsanweisung. Die mit dem Triggern einer Auswahl beginnende Ausführung einer Sequenz von Update-Tasks und Builtin-Aufrufen (vergleiche 5.3.4) ist dagegen nicht unterbrechbar, selbst über Prozedurgrenzen hinweg.

Diese ununterbrechbaren Folgen von Anweisungen sind so gewählt, daß ihre Ausführung nicht blockieren kann. Es werden ja nur die Wissensbasis modifiziert bzw. externe Aktionen angestoßen. Soll ein Thread die Kontrolle dennoch vorzeitig abgeben, so

kann dazu eine „leere“ Auswahl eingesetzt werden[2]:

```
insert r(now)
when true() ⇒ end
insert r(later)
```

Das erläuterte Konzept zum koordinierten Wissensbasiszugriff ist ein einfaches, aber mächtiges Basiskonstrukt, auf dem sich komplexere Synchronisationsmechanismen aufbauen lassen. Als Beispiel sei hier die Realisierung binärer Semaphore angeführt:

```
pred sema(out Name out State)

proc create(in Name)
    insert sema(Name free)
end
proc enter(in Name)
    when sema(Name free)
      ⇒ delete sema(Name free)
         insert sema(Name occupied)
    end
end
proc leave(in Name)
    when sema(Name occupied)
      ⇒ delete sema(Name ocucpied)
         insert sema(Name free)
    end
end
```

5.3.3 Thread-Scheduling

Jeder Thread befindet sich in einem von drei Zuständen, ausführend, ausführbereit oder wartend. Im Wartezustand befinden sich genau die Threads, die entweder in einer überwachten Auswahl stehen, bei der kein Wächter erfüllt ist, oder die am Ende eines Zweiges einer parallelen Komposition stehen und es mindestens einen noch nicht beendeten Parallelzweig gibt.

Zu jedem Zeitpunkt ist höchstens ein Thread im Zustand ausführend. Wie im letzten Abschnitt bereits angedeutet, verläßt der Thread diesen Zustand, sobald eines der folgenden Ereignisse auftritt:

[2] Der Inferenzalgorithmus für die Wissenbasis kann so gestaltet werden, daß (trotz inkrementeller Auswertung) der Berechnungsaufwand nicht bereits bei den Update-Anweisungen entsteht, sondern erst bei der Beantwortung der aktuellen Anfragen (siehe Abschnitt 6.6.2). Somit scheint eine Unterbrechung innerhalb einer Update-Sequenz aus Effizienzgründen im allgemeinen unnötig zu sein.

- Eintritt in eine überwachte Auswahl
- Eintritt in eine parallele Komposition
- Beendigung eines Zweiges einer parallelen Komposition
- Ausführung einer Prioritätsanweisung

Daraufhin wird ein neuer Thread zur Ausführung ausgewählt: Von allen nicht wartenden Threads wähle die höchst-priorisierten, und von diesen wähle den Thread, der die längste Zeit nicht zur Ausführung gekommen ist.

Sollte kein Thread ausführungsbereit sein, kann nur ein externes Ereignis (Nachrichtenempfang) die Ausführung fortsetzen. Wenn es keinen Thread mehr gibt, terminiert das Programm.

5.3.4 Builtin-Prozeduren

Genauso wie es in der Wissensbasis Builtin-Prädikate gibt, werden Builtin-Prozeduren für die Handlungsbasis benötigt, also Prozeduren, deren Funktionen durch eine Implementierung von Sita festgelegt sind.

Ihre wichtigste Anwendung ist das Auslösen externer Aktionen, etwa das Versenden von Nachrichten an andere Prozesse bzw. Agenten, oder, falls vorhanden, die Steuerung von Effektoren. Zusätzlich können jene Builtins angeboten werden, die auch in der Wissensbasis zur Verfügung stehen, vorausgesetzt es handelt sich um deterministische Funktionen, die nicht scheitern können, etwa arithmetische Funktionen.

Um die Orthogonalität der Konzepte nicht zu verletzen, sollten Builtins nicht die Möglichkeit haben, den ausführenden Thread zu blockieren, wie es etwa durch Warten auf das Ende einer zeitlich ausgedehnten Aktion der Fall wäre. Stattdessen sollten Rückmeldungen, die sich nicht unmittelbar beim Aufruf sondern erst mit zeitlicher Verzögerung ergeben, als asynchrone Nachrichten realisiert sein, die dann als Fakten entsprechender Prädikate in die Wissensbasis eingetragen werden. Als mögliche Beispiele seien hier das Versenden einer Nachricht mit Empfangsquittierung oder das Setzen eines Timers genannt.

5.4 Erweiterungen

Ein Ziel des Designs von Sita ist es, das in Kapitel 3 geschilderte Grundkonzept in Form einer möglichst minimalen Menge orthogonaler Konstrukte umzusetzen. Auf Seiten der Handlungsbasis bedeutet dies die Konzentration auf die essentiellen Konstrukte einer Koordinationssprache, wie bereits in Abschnitt 3.3.2 ausgeführt. Dank

dieser Konzentration zeigt sich der Sita-Kern offen für Kombinationen mit anderen Programmierkonzepten. Einige mögliche Erweiterungen der Handlungsbasis werden im folgenden kurz angedeutet.

- Eine parallele Komposition terminiert erst, wenn alle Sub-Tasks terminieren. Als Alternative kann ein Konstrukt zur Abspaltung eines Threads nützlich sein, etwa in Form einer Operation, die unmittelbar terminiert, aber als Seiteneffekt einen neuen Thread erzeugt (*spawning*). Die beteiligten Threads wären damit in ihrer Terminierung entkoppelt. Bei Bedarf kann eine Synchronisation immer über die Wissenbasis hergestellt werden.

- Die überwachte Auswahl stellt immer asynchron Anfragen an die Wissensbasis. Ihre Funktion ist somit neben der Fallunterscheidung auch das Warten auf einen zukünftigen Zustand. Als Ergänzung dazu können nicht-wartende Wissensbasisanfragen zugelassen werden, wie sie der Normalfall in (deduktiven) Datenbanken sind.

  Aufgrund der relationalen Struktur der Wissensbasis würde eine solche Anfrage eine Tupelmenge zum Ergebnis haben, die nun auf unterschiedliche Weise verwertet werden kann. Entweder wird sie in Gänze als Ergebniswert aufgefaßt, eventuell unter Zwischenschaltung von Aggregatsfunktionen[3]. Oder die Ergebnismenge dient als Grundlage einer neuen Kontrollstruktur, in der jedes Element (sequentiell oder parallel) in einem Sub-Task verarbeitet wird, etwa in der Form „**foreach** *Query* **do** *Task*“. Solche Konstrukte werden auch im Rahmen von Update-Sprachen für Datenbanken diskutiert, vergleiche etwa den Übersichtsartikel [Abi88].

- Die prozedurale Sprache kann in Richtung klassischer imperativer Programmierung ausgebaut werden. Hierzu ist zum einen ein Konzept überschreibarer Variablen einzuführen, zusammen mit einer entsprechenden Zuweisungsoperation. Zum anderen sind als Kontrollstrukturen Fallunterscheidungen und Schleifen hinzuzufügen, die im Gegensatz zur überwachten Auswahl nicht auf die Erfülltheit eines Wissensbasisanfrage warten, sondern unmittelbar auszuwertende, logische Ausdrücke über Variablenwerte testen. Beispiel einer Sprache, die sowohl logische Variablen als auch zuweisbare Variablen (sogenannte Zellen) unterstützt ist Oz [SHW95].

- Aus Agentensicht ist das prozedurale Programm eine Planbibliothek, und ein sich in Ausführung befindlicher Task somit ein zur Intention erklärter Plan. Konzeptionell ergeben sich daraus zusätzliche Aspekte der Ausführung: Pläne können aufgegeben werden, Pläne können scheitern, beim Scheitern eines Planes

[3] Ein alternatives Vorgehen wäre die Einführung von Aggregatsfunktionen in der Wissensbasis selbst, siehe Abschnitt 6.7.2.

stehen möglicherweise alternative Pläne zur Verfügung. Um diese Aspekte auf programmiersprachlicher Ebene zu unterstützen, bieten sich zwei alternative Erweiterungen an. Zum einen können Konstrukte zur Ausnahmebehandlung angeboten werden, so wie das etwa in C++ in Form von **try-catch-throw**-Klauseln möglich ist. Zum anderen können Threads etwa durch entsprechende Builtin-Prozeduren zu „first-class citizens“ gemacht werden, dem Programmierer also in Form manipulierbarer Datenstrukturen zugänglich gemacht werden.

KAPITEL 6

Auswertungsalgorithmen

*Der zentrale Punkt für die Ausführung von Sita-Programmen ist der Inferenzprozeß, der fortlaufend von den assertierten Fakten auf die deduzierten Fakten schließt. Hierzu entwickeln wir ein inkrementelles, vorwärtsverkettendes Verfahren, das wir IDC-Algorithmus (*Incremental Deductive Closure*) nennen. Dieser stellt eine wesentliche Weiterentwicklung des Rete-Algorithmus [For82] dar. Das Logikprogramm einer Wissensbasis wird in ein Netzwerk von Relationen und relationen-algebraischer Operationen transformiert, welches die operationellen Beziehungen zwischen extensionalen und intensionalen Relationen repräsentiert. Wir führen einen Konsistenzbegriff ein, der — gemäß der in Kapitel 4 definierten Semantik — den Modellbegriff auf Zustände des Relationen-Netzes überträgt.*
Update-Anweisungen führen zu Veränderungen an den extensionalen Relationen. Der Algorithmus propagiert diese Veränderungen durch das Relationen-Netz, bis ein konsistenter Zustand erreicht wird. Der hierbei bestehenden Gefahr von Zirkelschlüssen begegnen wir mit zwei alternativen Techniken. Im experimentellen Vergleich wird sich eine der beiden Techniken als im Laufzeitverhalten wesentlich stabiler erweisen. Anschließend untersuchen wir den Einfluß verschiedener Propagierungsstrategien auf die Effizienz des Algorithmus.
Die Anwendbarkeit des Algorithmus setzt eine gewisse, sehr schwache Stratifikationsbedingung voraus, die in der Praxis kaum als Einschränkung empfunden werden dürfte.

6.1 Differentielle Fixpunktiteration

Erster Ansatzpunkt zur bottom-up Auswertung ist die konstruktive Berechnung des kleinsten Herbrand-Modells mit Hilfe einer Fixpunktiteration gemäß Lemma 4.21. Dazu beschränken wir uns zunächst auf definite Programme.

Wir verwenden den Operator T_P, der die unmittelbaren Konsequenzen eines Programmes P berechnet, vergleiche Definition 4.19, und iterieren ausgehend von der leeren

Faktenmenge:

$$\begin{aligned} I_0 &= \emptyset \\ I_1 &= T_P(I_0) \\ &\vdots \\ I_{n+1} &= T_P(I_n) \end{aligned}$$

Nachteilig hierbei ist, daß bei jedem Iterationsschritt alle bereits berechneten Fakten nochmals berechnet werden. Dies wird durch die *differentielle* (auch *semi-naive*) Iteration vermieden [BR87a]. Hier werden in jedem Schritt nur die im vorherigen Schritt neu gefundenen Fakten zur Berechnung weiterer Konsequenzen herangezogen. Zur Formalisierung erweitern wir den T_P Operator, um zwischen schon behandelten Fakten I und neuen Fakten I' zu differenzieren:

Definition 6.1 Für ein definites Logikprogramm P definiere T_P^Δ folgende Abbildung:

$$\begin{aligned} T_P^\Delta : 2^{\mathcal{B}_P} \times 2^{\mathcal{B}_P} &\rightarrow 2^{\mathcal{B}_P} \\ T_P^\Delta(I, I') &:= \{a \mid a \leftarrow l_1 \dots l_n \text{ist Grundinstanz einer Regel in } P \\ &\qquad \text{mit } l_j \in I' \text{ für ein } j = 1..n \\ &\qquad \text{und } l_i \in I \text{ für alle } i = 1..n, i \neq j\} \end{aligned}$$

◁

Dieser differentielle Operator „joint“ also die neuen Fakten mit den alten Fakten. Da er nur noch Regeln mit wenigstens einem Literal im Rumpf betrachtet, müssen die Fakten des Programmes bzw. des Faktenspeichers zu Iterationsbeginn eigens eingerechnet werden.

$$\begin{aligned} I_0 &= \emptyset \\ I_1 &= T_P(I_0) \\ I_2 &= I_1 \cup T_P^\Delta(I_0,\ I_1 - I_0) \\ &\vdots \\ I_{n+1} &= I_n \cup T_P^\Delta(I_{n-1},\ I_n - I_{n-1}) \end{aligned}$$

Sobald $I_{n+1} = I_n$ gilt, ist das kleinste Herbrand-Modell gefunden.

In der dargestellten Form erlauben diese Iterationsverfahren die Auswertung eines Logikprogrammes gegen eine feste extensionale Datenbasis, so wie es in deduktiven Datenbanken gefordert ist. Das differentielle Vorgehen, bisher nur Effizienzvorteil, liefert aber auch den Schlüssel zur inkrementellen Auswertung, wie sie für das

Sita-Berechnungsmodell benötigt wird. Dazu werden Modifikationen des Faktenspeichers konzeptionell wie die „neuen“ Fakten im T_P^Δ Operator behandelt. Im anschließenden Iterieren wird dafür gesorgt, daß die Änderungen durch alle Regeln hindurch propagiert werden. Zur Realisierung dieser Idee sind einige weitere Gesichtspunkte wesentlich:

- Der T_P^Δ Operator bildet Mengen neuer Fakten auf Mengen neuer Fakten ab. Er kann aber genauso zur Behandlung nur eines neuen Faktums verwendet werden (*tupel-at-a-time* statt *set-at-a-time*).
- Die Reihenfolge, in der Neuerungen propagiert werden, muß nicht obigem Iterationsschema folgen. Sie kann vielmehr flexibel gewählt werden.
- Das Entfernen von Fakten kann analog zum Einfügen gestaltet werden.
- Das Verfahren kann um die Behandlung der Negation erweitert werden. Dabei sind gewisse Stratifikationsbedingungen an das Programm zu stellen.

Im Rest dieses Kapitels werden diese Aspekte verwendet, um den IDC-Algorithmus zur effizienten inkrementellen bottom-up Auswertung von Logikprogrammen zu entwickeln.

6.2 Relationen-Netze

Zur Auswertung werden die Logikprogramme übersetzt in Gleichungssysteme der relationalen Algebra. Statt von Prädikaten sprechen wir nun von Relationen. Die Klauseln, die ein Prädikat definieren, werden zu der definierenden Gleichung der entsprechenden Relation. Zudem bezeichnen wir Fakten nun meist als Tupel.

Dieser Wechsel des Formalismus ergibt zunächst nur eine andere Repräsentation des Logikprogrammes. Für stratifizierbare Programme liefert dieser Wechsel aber auch eine operationelle Semantik, die mit der modelltheoretischen Semantik übereinstimmt.

Analog zu den Prädikaten werden auch die Relationen in extensionale und intensionale unterschieden. Die *extensionale Datenbank* (*EDB*) ist die Menge aller Relationen, die extensionale Prädikate repräsentieren, die *intensionale Datenbank* (*IDB*) ist entsprechend die Menge aller Relationen, die intensionale Prädikate repräsentieren.

Die Technik des Übersetzungsvorganges kann als Standardverfahren bezeichnet werden, das etwa in [Ull88] und [Ull89] erläutert ist. In [Per97] sind Formalia und Realisierung eines solchen Übersetzers speziell für Sita detailiert beschrieben.

Wir stellen die resultierenden Gleichungssysteme bildlich als *Relationen-Netze* dar. Dies sind bipartite Graphen, deren Knoten zum einen die Relationen sind und zum

anderen Operationsknoten, die die zu berechnenden Beziehungen zwischen den Relationen darstellen. Es gibt drei Arten von Operationsknoten:

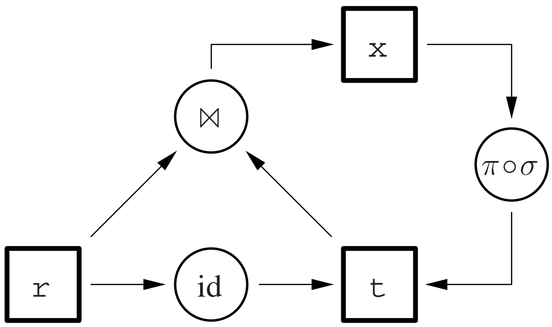
- *Join-Knoten* berechnen das Kreuzprodukt zweier Relationen.
- *Differenz-Knoten* berechnen die Differenz zweier (gleichstelliger) Relationen.
- *Funktions-Knoten* haben nur eine Eingangsrelation. Sie berechnen die einstelligen Operationen Selektion und Projektion und beherbergen zudem die als Builtins zur Verfügung stehenden Funktionen.

Die Operationsknoten umfassen damit alle Operatoren der relationalen Algebra bis auf die Vereinigung. Letztere kommt implizit dadurch zustande, daß mehrere Operationsknoten am Ausgang in die gleiche Relation münden. Diese Relation ist dann die Vereinigung der von den Operationsknoten berechneten Relationen.

Beispiel 6.2 Folgendes Programm berechnet die transitive Hülle `t` einer Relation `r` (vergleiche Beispiel 4.2):

```
t(X Y) ← r(X Y).
t(X Y) ← r(X Z)  t(Z Y).
```

Das zugehörige Relationen-Netz benötigt einen Join-Knoten, zwei Funktions-Knoten (einmal Identität und einmal Selektion mit anschließender Projektion), sowie eine Hilfsrelation `x`.



◁

Wie im Beispiel ersichtlich sind Hilfsrelationen unter anderem dort einzusetzen, wo das Ergebnis eines Kreuzproduktes oder einer Differenz noch selektiert und projiziert werden muß. Die Trennung von Join einerseits und Selektion bzw. Projektion andererseits in verschiedene Typen von Operationsknoten geschieht zum Zwecke der einfacheren formalen Darstellung. In einer Implementierung kann die Darstellung der Relationen-Netze auf einer maschinennäheren Ebene erfolgen. Durch eine größere Übersetzungstiefe kann etwa ein Kreuzprodukt direkt weiteren Operationen unterworfen

werden, bzw. als Index-Join durchgeführt werden, ohne daß eine Hilfsrelation nötig wäre. Deswegen nennen wir den entsprechenden Knoten auch Join-Knoten statt Kreuzprodukt-Knoten. Selektion und Projektion hinter einem Kreuzprodukt werden wir gelegentlich nicht eigens aufführen, sondern als Funktionalität des Join-Knoten betrachten.

Zur Darstellung von Funktionsknoten benötigen wir den Begriff der Tupel- bzw. Relationenfunktion.

Definition 6.3 f heißt *Tupelfunktion* von Relation r_1 nach Relation r_2, wenn f eine Abbildung

$$f : \mathrm{dom}(r_1) \quad \rightarrow \quad 2^{\mathrm{dom}(r_2)}$$

ist, und jeder Bildpunkt als Menge endlich ist.

Jede Tupelfunktion wird zu einer Relationenfunktion erweitert:

$$\begin{aligned} f : 2^{\mathrm{dom}(r_1)} \quad &\rightarrow \quad 2^{\mathrm{dom}(r_2)} \\ f(\{a_1, \ldots a_n\}) \quad &:= \quad \bigcup_{i=1..n} f(a_i) \end{aligned}$$

Gemäß der Konstruktion ist jede Relationenfunktion monoton. ◁

Tupelfunktionen sind aus zweierlei Gründen mengenwertig: Zum einen können dadurch Selektionen ausgedrückt werden, indem sie null- bzw. ein-elementige Mengen zurückliefern. Zum anderen können so auch nicht-deterministische Builtins verpackt werden, die zu einem Eingangstupel eine Menge von Ergebnistupeln liefern.

Bevor wir nun Relationen-Netze formal darstellen können, ist noch eine Vorbemerkung über die Ausgänge einer Relation notwendig. Im allgemeinen besitzt eine Relation mehrere Operationsknoten als Nachfolger. So besitzt im vorangegangenen Beispiel 6.2 etwa die Relation `r` zwei ausgehende Pfeile. Im Abschnitt 6.4 wird die Notwendigkeit deutlich werden, zwischen den verschiedenen Ausgängen einer Relation unterscheiden zu können. Wir versehen deshalb jeden Relationenausgang mit einem eindeutigen Bezeichner. Die Verbindungen im Relationen-Netz sind dann gegeben zum einen durch Pfeile, die jeweils von einem Relationenausgang zu einem Operationsknoten führen, und zum anderen durch Pfeile, die jeweils von einem Operationsknoten zu einer Relation führen. In der folgenden Definition sind diese Verknüpfungen dadurch modelliert, daß in jedem Operationsknoten der oder die vorgelagerten Relationenausgänge sowie die nachfolgende Relation genannt werden.

Definition 6.4 Ein *Relationen-Netz* $N = (R, A, O, K)$ ist gekennzeichnet durch:

R	Menge der beteiligten Relationennamen
A	Menge von Bezeichnern für Relationenausgänge
O	Zuordnung von Relationenausgängen zu Relationen, $O \subset R \times A$ $r \rightsquigarrow a \quad :\Leftrightarrow \quad (r, a) \in O \quad :\Leftrightarrow$ „a ist ein Ausgang der Relation r“
K	Menge von Operationsknoten

Operationsknoten sind:

$\text{join}(a_1, a_2, r)$	auch geschrieben als	$a_1 \bowtie a_2 \rightarrow r$
$\text{diff}(a_1, a_2, r)$		$a_1 \setminus a_2 \rightarrow r$
$\text{func}(f, a, r)$		$f(a) \rightarrow r$

wobei f eine Tupelfunktion ist.

Zusätzlich muß das Relationen-Netz „wohlgeformt“ sein:
Jeder Ausgang $a \in A$ ist genau einer Relation $r \in R$ zugeordnet und wird von genau einem Operationsknoten $k \in K$ als Eingang verwendet. ◁

Beispiel 6.5 Das im vorangegangenen Beispiel 6.2 graphisch dargestellte Relationen-Netz $N = (R, A, O, K)$ setzt sich aus folgenden Komponenten zusammen:

$$\begin{array}{rcl} R & = & \{\texttt{r}, \texttt{t}, \texttt{x}\} \\ A & = & \{\texttt{r_a1}, \texttt{r_a2}, \texttt{t_a1}, \texttt{x_a1}\} \\ O & : & \texttt{r} \rightsquigarrow \texttt{r_a1},\ \texttt{r} \rightsquigarrow \texttt{r_a2},\ \texttt{t} \rightsquigarrow \texttt{t_a1},\ \texttt{x} \rightsquigarrow \texttt{x_a1} \\ K & = & \{\ \text{func}(\text{id}, \texttt{r_a1}, \texttt{t}), \\ & & \quad \text{join}(\texttt{r_a2}, \texttt{t_a1}, \texttt{x}), \\ & & \quad \text{func}(\pi_{\texttt{X,Y}} \circ \sigma_{\texttt{r.Z=t.Z}}, \texttt{x_a1}, \texttt{t})\ \} \end{array}$$

◁

Durch die Verwendung rekursiv definierter Prädikate entstehen im allgemeinen Relationen-Netze mit Zyklen. Wir sprechen von einem *negativen Zyklus*, wenn am Zyklus wenigstens einmal der negative Eingang eines Differenzknotens beteiligt ist. Ansonsten sprechen wir von einem *positiven Zyklus*. Die globale Stratifizierbarkeit auf Programm-Ebene entspricht damit der Freiheit von negativen Zyklen auf Netz-Ebene.

6.3 Netz-Konsistenz

Wir betrachten nun den Zustand eines Relationen-Netzes, der sich daraus ergibt, daß hinter den Relationennamen nun konkrete Instanzen dieser Relationen stehen. Ein neuer Formalismus ist dazu nicht notwendig, da sich jeweils aus dem Zusammenhang ergibt, ob der statische Aspekt eines Netzes gemeint ist, oder aber sein konkreter, dynamischer Zustand.

Da der zu entwickelnde Auswertungsalgorithmus letztendlich Modelle nach den in Kapitel 4 angesprochenen Semantiken berechnen soll, müssen wir die Modell-Eigenschaft auf die Ebene von Netzzuständen übertragen. Dort nennen wir diese Eigenschaft *Konsistenz*. Dabei beschränken wir uns vorerst auf stratifizierbare Programme. Im Abschnitt 6.8 betrachten wir dann allgemeinere Programme.

Die Semantik definiter Programme ist durch das kleinste Herbandmodell gegeben, vergleiche Lemma 4.18. Bei der Übertragung auf Relationen-Netze erhalten wir mehrere zu fordernde Eigenschaften, die zusammen Konsistenz bedeuten.

Zum einen muß ein konsistenter Netzzustand ein Modell repräsentieren, also nach Definition 4.17 alle Regeln des Programmes erfüllen. Existieren Fakten, die den Rumpf einer Regel erfüllen, so müssen auch Fakten existieren, die den Kopf erfüllen. Wir bezeichnen diese Eigenschaft mit *Vollständigkeit*.

Um nun den Netzzustand mit dem kleinsten aller Herbrandmodelle zu vergleichen, benötigen wir ein Minimalitätskriterium. Dazu ist erforderlich, daß sich die Existenz von Tupeln entweder auf ihre Extensionalität stützt, oder aber aufgrund von Regeln notwendig folgt. In gewisser Weise werden hier die Operationsknoten rückwärts betrachtet: Jedes nicht-extensionale Tupel muß durch eine entsprechende Operation begründet sein. Wir bezeichnen dies mit *lokaler Minimalität*. Ein Zustand, der zugleich vollständig und lokal minimal ist, heißt auch *lokal konsistent*.

Lokale Eigenschaften allein reichen aber nicht aus, um die Minimalität insgesamt zu sichern. Dieser entgegen steht die Möglichkeit von „Zirkelschlüssen“, also Tupel, die sich nur rein untereinander stützen, so daß sie ohne Verlust der Modelleigenschaft komplett gestrichen werden können. Solche Zyklen sind nicht lokal an den einzelnen Operationsknoten zu erkennen, sondern nur durch globale Betrachtung des Netzzustandes. Wir führen deshalb als weitere Eigenschaft die *(globale) Minimalität* ein, die die lokale Minimalität verschärft. Entsprechend nennen wir einen Zustand *global konsistent* (oder kurz *konsistent*), wenn er vollständig und global minimal ist.

Wir formalisieren im folgenden diese Begriffe.

Definition 6.6 Ein Netzzustand heißt *vollständig*, wenn für alle Operationsknoten die folgenden, dem Operationstyp entsprechenden Eigenschaften erfüllt sind:

- Für jeden Join-Knoten $\mathrm{join}(a_1, a_2, r_3)$ mit $r_1 \rightsquigarrow a_1$ und $r_2 \rightsquigarrow a_2$ gilt:

  $$\forall t_1 \in r_1 \;\; \forall t_2 \in r_2 \;:\;\; (t_1, t_2) \in r_3$$

- Für jeden Differenz-Knoten $\mathrm{diff}(a_1, a_2, r_3)$ mit $r_1 \rightsquigarrow a_1$ und $r_2 \rightsquigarrow a_2$ gilt:

  $$\forall t \in r_1 \;:\;\; t \notin r_2 \;\Rightarrow\; t \in r_3$$

- Für jeden Funktions-Knoten $\mathrm{func}(f, a_1, r_2)$ mit $r_1 \rightsquigarrow a_1$ gilt:

  $$\forall t \in r_1 \;:\;\; f(t) \subset r_2$$

◁

Definition 6.7 Ein Tupel $t \in r$ heißt *lokal begründet*, wenn wenigstens eine der folgenden Bedingungen erfüllt ist:

- Die Relation r ist extensional.

- Es gibt einen Join-Knoten $\text{join}(a_1, a_2, r)$ mit $r_1 \rightsquigarrow a_1$ und $r_2 \rightsquigarrow a_2$ und:

$$\exists t_1 \in r_1 \;\; \exists t_2 \in r_2 \;:\;\; (t_1, t_2) = t$$

- Es gibt einen Differenz-Knoten $\text{diff}(a_1, a_2, r)$ mit $r_1 \rightsquigarrow a_1$ und $r_2 \rightsquigarrow a_2$ und:

$$t \in r_1 \;\; \wedge \;\; t \notin r_2$$

- Es gibt einen Funktions-Knoten $\text{diff}(f, a_1, r)$ mit $r_1 \rightsquigarrow a_1$ und:

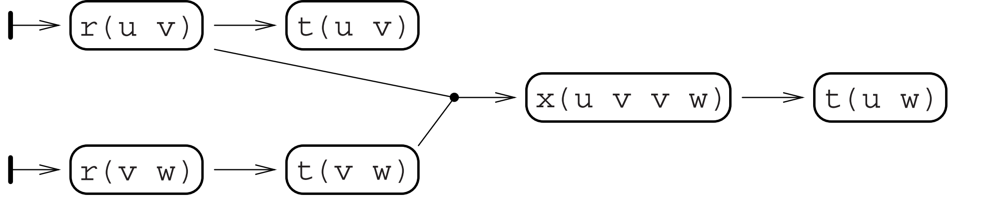
$$t \in f(r_1)$$

Ein Netzzustand heißt *lokal minimal*, wenn alle Tupel aller Relationen lokal begründet sind.

Ein Netzzustand heißt *lokal konsistent*, wenn er vollständig und lokal minimal ist. ◁

Zur Definition der globalen Minimalität benötigen wir einen Blick auf den gesamten Zustand des Netzes. Hierzu stellen wir lokal konsistente Zustände als *Tupelgraphen* dar. Die in den Relationen gespeicherten Tupel werden als Knoten dargestellt, ihre Ableitungsbeziehungen als Kanten. Aufgrund der lokalen Konsistenz sind diese Ableitungsbeziehungen bei gegebenem Relationen-Netz und Netzzustand eindeutig.

Beispiel 6.8 Wir setzen das Beispiel 6.2 zur Berechnung der transitiven Hülle fort. `r` sei nun eine extensionale Relation, in der die zwei Tupel `r(u v)` und `r(v w)` vorhanden seien. Damit erhalten wir folgenden Tupelgraphen, der einen konsistenten Zustand visualisiert:



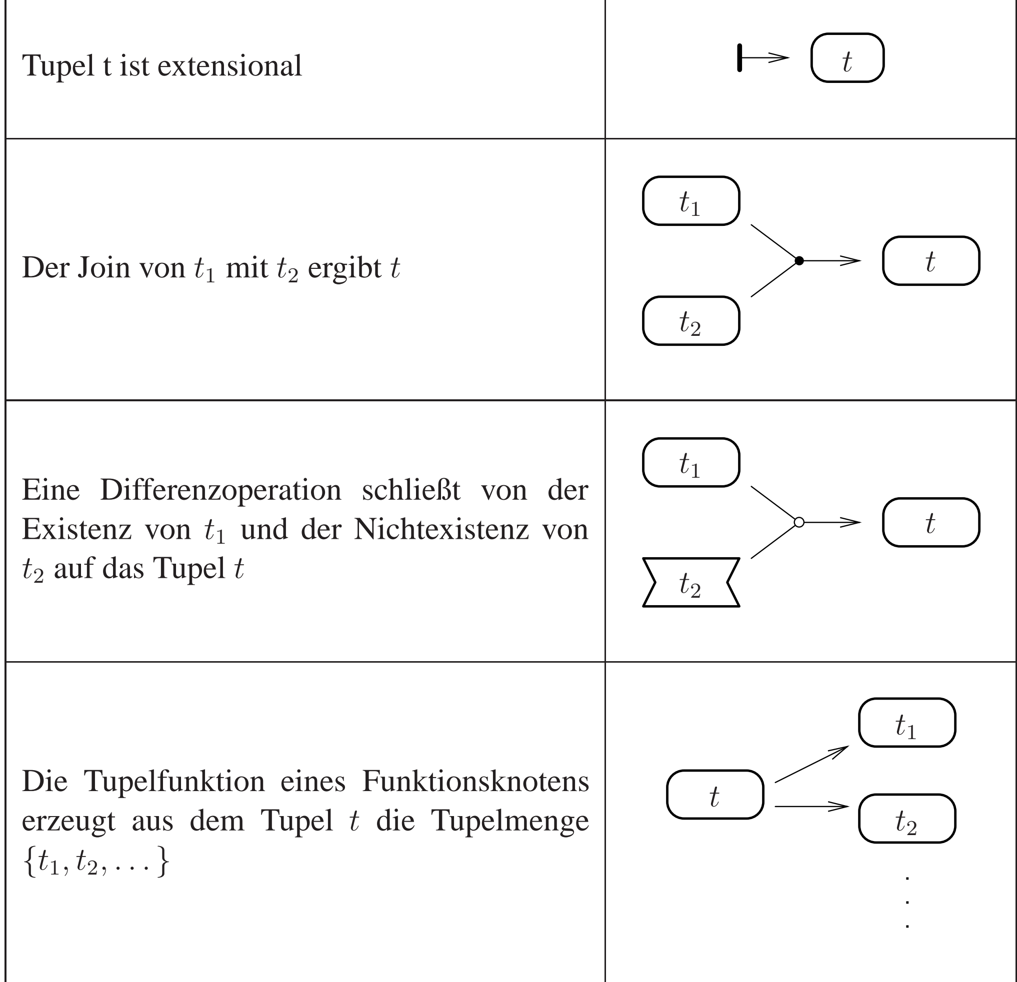
◁

Definition 6.9 Jedem lokal konsistenten Netzzustand wird ein *Tupelgraph* zugeordnet, der nach dem in Tabelle 6.1 gezeigten Schema aufgebaut ist.

Jedes Tupel einer Relation wird als Knoten dargestellt. Neben diesen (normalen) Knoten gibt es noch speziell gekennzeichnete Knoten (siehe nachfolgende Illustration), welche auf die Nichtexistenz eines Tupels hinweisen. Diese Knoten werden benötigt, um in der Begründung einer Differenzoperation auch den negativen Anteil darstellen zu können.

Nach den Definitionen 6.6 und 6.7 ergeben sich eindeutige Ableitungsbeziehungen, die gemäß der Tabelle 6.1 als (Hyper-) Kanten eingefügt werden.

◁



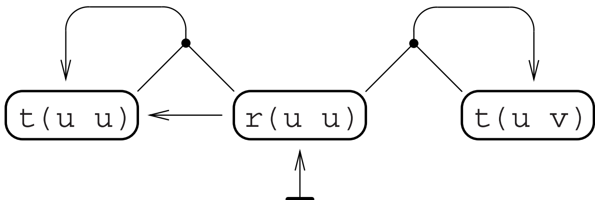
Tupel t ist extensional	t
Der Join von t_1 mit t_2 ergibt t	t_1, t_2, t
Eine Differenzoperation schließt von der Existenz von t_1 und der Nichtexistenz von t_2 auf das Tupel t	t_1, t_2, t
Die Tupelfunktion eines Funktionsknotens erzeugt aus dem Tupel t die Tupelmenge $\{t_1, t_2, \ldots\}$	t, t_1, t_2, $\vdots$

Tabelle 6.1: Darstellung der Operationen im Tupelgraph

An dieser Stelle ist noch eine Bemerkung notwendig. Im Hornklauselprogramm dürfen Regeln mit leerem Rumpf notiert werden, die man auch als intensionale Fakten bezeichnen könnte. Es stellt sich die Frage, wie das Begründetsein dieser Fakten im Tupelgraph notiert werden soll. Eine Möglichkeit besteht darin, den leeren Rumpf durch `true()` zu ersetzen. Zusätzlich wird dann eine extensionale, null-stellige Relation `true` eingeführt und dort das `true()`-Tupel standardmäßig eingefügt. Damit ist dieser Fall in die Notation des Tupelgraphen integriert. Alternativ kann als Vereinfachung das „Erdungssymbol" im Tupelgraphen direkt an das intensionale Faktum gehängt werden.

Tupelgraphen erlauben nun, unberechtigte Zirkelschlüsse zu erkennen, wie folgendes einfaches Beispiel zeigt.

Beispiel 6.10 Im Beispielprogramm 6.2 für die transitive Hülle sei `r` wieder eine extensionale Relation, in der nun das eine Tupel `r(u u)` vorhanden sei. Dann ist folgender Tupelgraph lokal konsistent. Dir Begründung des Tupels `t(u v)` baut jedoch auf die Existenz eben dieses Tupels. Der Zustand ist somit nicht global minimal.



◁

Definition 6.11 Ein Tupel t einer Relation r heißt *begründet*, wenn wenigstens eine der folgenden Bedingungen erfüllt ist:

- Die Relation r ist extensional.
- Für einen Join-Knoten $\text{join}(a_1, a_2, r)$ mit $r_1 \rightsquigarrow a_1$ und $r_2 \rightsquigarrow a_2$ gilt:
  Es gibt begründete Tupel $t_1 \in r_1$ und $t_2 \in r_2$ mit $(t_1, t_2) = t$
- Für einen Differenz-Knoten $\text{diff}(a_1, a_2, r)$ mit $r_1 \rightsquigarrow a_1$ und $r_2 \rightsquigarrow a_2$ gilt:
  t ist als Tupel von r_1 begründet und $t \notin r_2$
- Für einen Funktions-Knoten $\text{diff}(f, a_1, r)$ mit $r_1 \rightsquigarrow a_1$ gilt:
  Es gibt ein begründetes Tupel $t_1 \in r_1$ mit $t \in f(r_1)$

Ein Netzzustand heißt *minimal*, wenn alle Tupel aller Relationen begründet sind.

Ein Netzzustand heißt *konsistent*, wenn er vollständig und minimal ist. ◁

Aus den Definitionen 6.7 und 6.11 folgt sofort, daß ein minimaler Netzzustand auch lokal minimal ist. Entsprechend ist ein konsistenter Zustand auch lokal konsistent.

6.4 Einfaches Ausgleichen

Ziel des zu entwickelnden Algorithmus ist es, zu einer gegebenen Menge extensionaler Tupel einen konsistenten Zustand des Netzes zu berechnen. Diese Berechnung erfolgt inkrementell: Nachdem extensionale Fakten ergänzt oder gestrichen wurden, propagiert das Verfahren diese Veränderungen sukzessive durch das Netz, bis wieder ein konsistenter Zustand erreicht wird. Dieser Prozeß wird im folgenden auch *Ausgleichen* des Netzes genannt.

Um den Vorgang zeitlich weiter aufzugliedern, wird das Konzept des Netzzustandes dahingehend erweitert, daß die noch zu bearbeitenden Veränderungen mit dargestellt

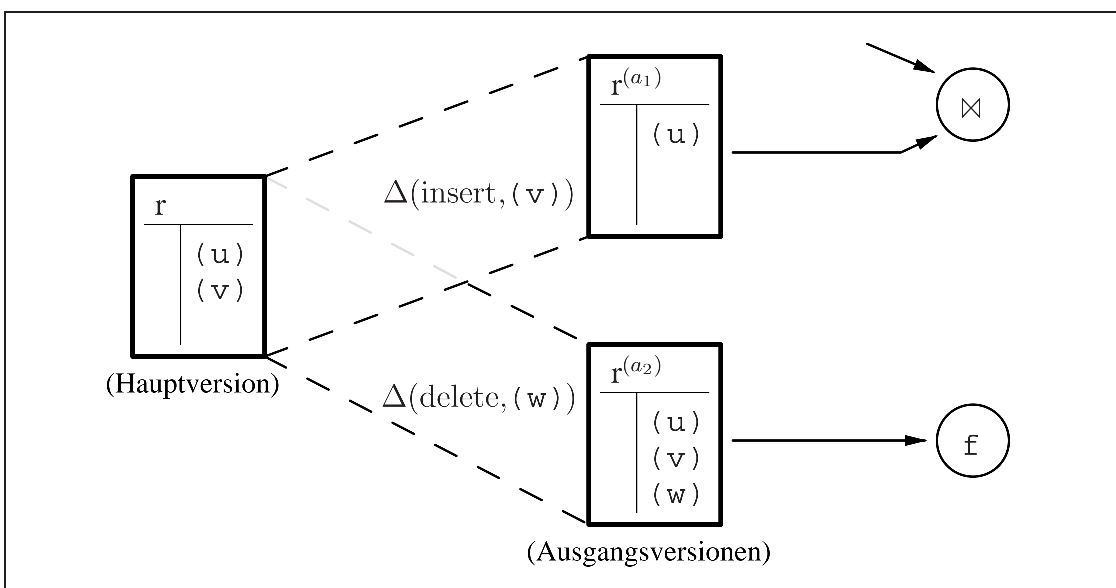


Abbildung 6.1: Deltas zwischen Haupt- und Ausgangsversionen, siehe Beispiel 6.12

werden. Dazu fächern wir eine Relation in mehrere Versionen auf. Neben der Hauptversion, in der alle bisher eingegangenen Veränderungen eingetragen sind, gibt es pro Relationenausgang eine weitere Version, die den bisher an den direkt nachfolgenden Operationsknoten mitgeteilten Stand darstellt. Die Differenz zwischen einer Ausgangsversion und der Hauptversion einer Relation entspricht damit den auf diesem Ausgang noch weiterzuleitenden Veränderungen. Die Differenz in einem Tupel bezeichnen wir auch als *Delta-Tupel*, oder kurz als *Delta*. Dies ist ein Tupel zusammen mit einer Markierung, ob das Tupel eingefügt oder gelöscht werden soll.

Deltas manifestieren sich an den Tupeln der Ausgangsversionen. Wir nennen ein Paar bestehend aus einem Tupel und einem Ausgang einer Relation eine *Stelle*. Weicht eine Stelle bezüglich ihrem Tupel von der Hauptversion ab, so ist sie als *unausgeglichen* zu markieren.

Wird die Hauptversion einer Relation geändert, so werden also die betroffenen Tupel, für jeden Ausgang getrennt, als unausgeglichen markiert. Die Gesamtheit dieser Stellen-Markierungen aller Relationen bezeichnen wir als *Delta-Menge*. Wenn die Delta-Menge leer ist, sind in jeder Relation alle Ausgangsversionen identisch mit der Hauptversion. Ziel der Berechnung ist es also, das Netz in einem konsistenten Zustand mit leerer Delta-Menge zu hinterlassen.

Beispiel 6.12 Wir betrachten eine Relation `r`, die zwei Ausgänge a_1 und a_2 besitzt. In Abbildung 6.1 ist nun folgende Situation dargestellt: Die Hauptversion von `r` enthält zwei Tupel, von denen der Ausgangsversion $\mathrm{r}^{(a_1)}$ erst eines bekannt ist. Hieraus ergibt sich ein insert-Delta. In der anderen Ausgangsversion $\mathrm{r}^{(a_2)}$ hingegen ist ein Tupel überzählig, woraus sich ein delete-Delta ergibt. ◁

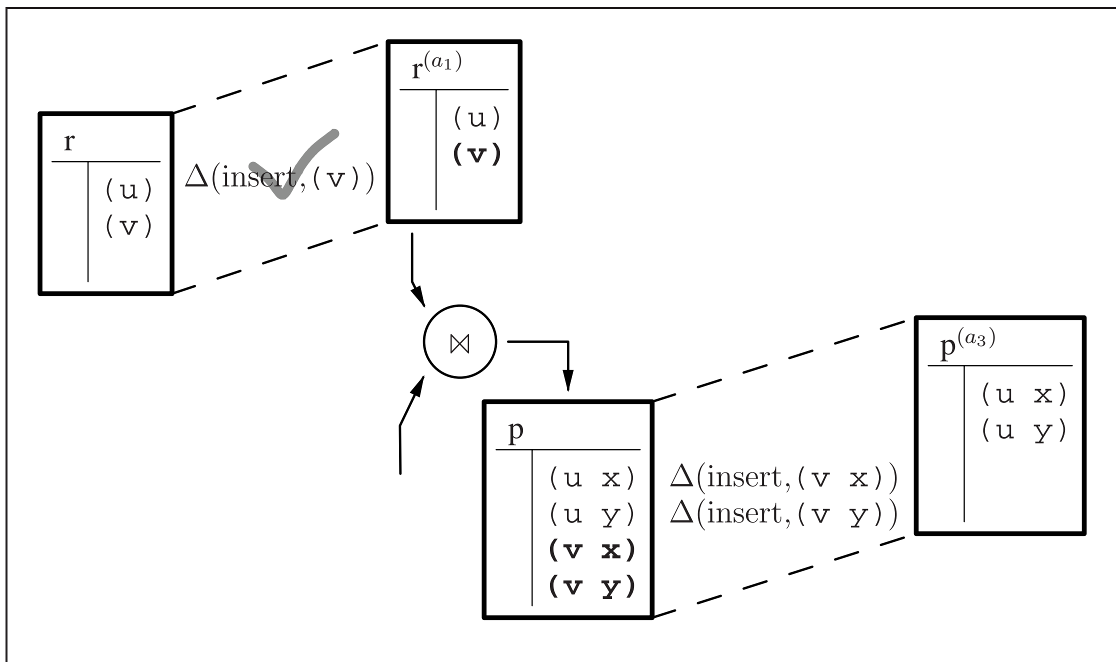


Abbildung 6.2: Reduktion eines Deltas, siehe Beispiel 6.13. Die neu entstehenden Tupel sind fett gedruckt.

Der Ausgleichsprozeß besteht aus einer Sequenz von *Reduktionen*. Pro Schritt wird eine unausgeglichene Stelle zur Reduktion ausgewählt. Nun wird das entsprechende Delta durch den nachfolgenden Operationsknoten propagiert, der daraus eine Menge neuer Deltas berechnet. Diese Deltas werden der hinter dem Operationsknoten liegenden Relation aufgeschlagen, also in die Hauptversion dieser Relation eingerechnet. Ergeben sich hierdurch Veränderungen, so werden entsprechende neue Markierungen in die Delta-Menge eingetragen. Dieser Prozeß wird fortgeführt, bis die Delta-Menge leer ist.

Beispiel 6.13 In Fortsetzung des Beispiels 6.12 ist in Abbildung 6.2 die Reduktion eines Deltas dargestellt. Zur Reduktion ausgewählt wird die unausgeglichene Stelle `(v)` am Ausgang $\mathrm{r}^{(a_1)}$. Das bisher fehlende Tupel wird in die Ausgangsversion eingetragen und als insert-Delta an den nachfolgenden Operationsknoten gesendet. Im Beispiel ist dies eine Join-Operation, bei der in der Folge zwei neue Tupel entstehen. Diese werden in die Hauptversion der nachfolgenden Relation `p` eingetragen. Da sie dort neu sind, werden hier die entsprechenden Stellen als unausgeglichen markiert, woraus wiederum neue Deltas resultieren. ◁

Zur weiteren Formalisierung werden wir den Begriff der lokalen Konsistenz auch auf unausgeglichene Zustände ausweiten, deren Delta-Menge also nicht leer ist. Ein Reduktionsschritt muß dann einen konsistenten Zustand in einen konsistenten Zustand überführen. Die erweiterte Definition der Konsistenz ist mit der ursprünglichen verträglich. Führt also der Reduktionsprozeß zu einem Zustand mit leerer Delta-Menge,

so ist dieser Zustand konsistent im ursprünglichen Sinne der Definition 6.11.

Wir erweitern zunächst den Zustands- und Konsistenzbegriff und beschreiben das Verfahren zur Aufrechterhaltung lokaler Konsistenz. Techniken zur Vermeidung unberechtigter Zirkelschlüsse führen wir im Abschnitt 6.5 ein.

6.4.1 Unausgeglichene Zustände

Um den oben angedeuteten Ausgleichsprozeß formalisieren zu können, genügt es nicht mehr, Relationen im Sinne der relationalen Programmierung auf einfache mathematische Relationen abzubilden. Der Zustand einer Relation wird daher um Versionen für die Relationenausgänge sowie um Faktenzähler für die Hauptversion erweitert.

Die Bedeutung der Ausgangsversionen wurde oben bereits erwähnt. Der Faktenzähler hat folgenden Hintergrund: Im Relationen-Netz münden im allgemeinen mehrere Operationsknoten in eine Relation. Relational-algebraisch stellt die Relation die Vereinigung der entsprechenden Operationsergebnisse dar. Ein Tupel der Relation kann damit mehrere unmittelbare Begründungen haben. Ändert sich der Netzzustand, so darf das Tupel erst gestrichen werden, wenn alle seine Begründungen wegfallen. Zu diesem Zwecke wird jedem in der Relation vorhandenen Tupel ein Zähler zur Seite gestellt, der angibt, auf wievielen Eingangswegen das Tupel gestützt wird. Formal modellieren wir diese Zähler mit Hilfe von Multimengen:

Definition 6.14 Der dynamische Zustand einer Relation r ist durch folgende Strukturen charakterisiert:

1. *Hauptversion*: r ist Multimenge über dem entsprechenden Tupelraum. (Sei $\mathcal{W}$ das zugrundeliegende Universum und n die Stelligkeit von r. Dann ist r eine Abbildung $r : \mathcal{W}^n \to \mathbb{N}$)
2. *Ausgangsversionen*: Für jeden Ausgang a von r bezeichnet $r^{(a)}$ die Version, die dem nachfolgenden Operationsknoten bekannt ist. ($r^{(a)} \subset \mathcal{W}^n$)

$\triangleleft$

Definition 6.15 (Ausweitung der Definitionen 6.6 und 6.7 auf unausgeglichene Zustände)

Ein Netzzustand heißt *vollständig*, wenn für alle Operationsknoten die folgenden, entsprechenden Eigenschaften erfüllt sind:

- Für jeden Join-Knoten $\mathrm{join}(a_1, a_2, r_3)$ mit $r_1 \rightsquigarrow a_1$ und $r_2 \rightsquigarrow a_2$ gilt:

$$\forall t_1 \in r_1^{(a_1)} \ \ \forall t_2 \in r_2^{(a_2)} \ : \ (t_1, t_2) \in r_3$$

- Für jeden Differenz-Knoten $\mathrm{diff}(a_1, a_2, r_3)$ mit $r_1 \rightsquigarrow a_1$ und $r_2 \rightsquigarrow a_2$ gilt:

$$\forall t \in r_1^{(a_1)} \ : \ t \notin r_2^{(a_2)} \ \Rightarrow \ t \in r_3$$

- Für jeden Funktions-Knoten $\mathrm{func}(f, a_1, r_2)$ mit $r_1 \rightsquigarrow a_1$ gilt:

$$\forall t \in r_1^{(a_1)} \ : \ f(t) \subset r_2$$

Ein Tupel $t \in r$ heißt *lokal begründet*, wenn wenigstens eine der folgenden Bedingungen erfüllt ist:

- Die Relation r ist extensional.
- Es gibt einen Join-Knoten $\mathrm{join}(a_1, a_2, r)$ mit $r_1 \rightsquigarrow a_1$ und $r_2 \rightsquigarrow a_2$ und:

$$\exists t_1 \in r_1^{(a_1)} \ \exists t_2 \in r_2^{(a_2)} \ : \ (t_1, t_2) = t$$

- Es gibt einen Differenz-Knoten $\mathrm{diff}(a_1, a_2, r)$ mit $r_1 \rightsquigarrow a_1$ und $r_2 \rightsquigarrow a_2$ und:

$$t \in r_1^{(a_1)} \ \wedge \ t \notin r_2^{(a_2)}$$

- Es gibt einen Funktions-Knoten $\mathrm{diff}(f, a_1, r)$ mit $r_1 \rightsquigarrow a_1$ und:

$$t \in f(r_1^{(a_1)})$$

Ein Netzzustand heißt *lokal minimal*, wenn alle Tupel aller Relationen lokal begründet sind.

Ein Netzzustand heißt *lokal konsistent*, wenn er vollständig und lokal minimal ist. $\triangleleft$

Globale Konsistenz, wie sie für statische Zustände definiert ist, kann für unausgeglichene Zustände nicht definiert werden. Bei ungeschicktem Vorgehen besteht immer die Gefahr, einen unausgeglichenen Zustand in einen ausgeglichenen, aber global nicht-minimalen Zustand zu überführen. Globale Minimalität kann aber durch eine umsichtige Steuerung des Reduktionsprozesses sichergestellt werden, wie in Abschnitt 6.5 diskutiert werden wird.

6.4.2 Reduktion

Im Reduktionsprozeß fließen Deltas von Relationsknoten durch Operationsknoten in die dahinterliegenden Relationsknoten. Wir geben erst eine Schreibweise für Deltas an und definieren, wie Operationsknoten auf Deltas arbeiten. Anschließend können wir den Reduktionsprozeß zusammenhängend beschreiben.

Definition 6.16

- Ein *Delta* d beschreibt das Einfügen bzw. Löschen eines Tupels t:

$$d = \begin{cases} \Delta(\mathrm{insert}, t) & \text{(Einfügen)} \\ \Delta(\mathrm{delete}, t) & \text{(Löschen)} \end{cases}$$

- Die *Anwendung* $r \blacktriangleright d$ eines Deltas d auf eine Relation r liefert die Relation mit entsprechend eingefügtem bzw. gelöschten Tupel: Tupels t:

$$r \blacktriangleright d = \begin{cases} r \cup \{t\} & \text{falls} \quad d = \Delta(\mathrm{insert}, t) \\ r \setminus \{t\} & \text{falls} \quad d = \Delta(\mathrm{delete}, t) \end{cases}$$

- Die Anwendung $r \blacktriangleright D$ einer Menge D von Deltas ist definiert, wenn sich alle Deltas der Menge auf unterschiedliche Tupel beziehen, und berechnet sich durch die beliebig sequentialisierte Anwendung der Einzel-Deltas.

$\triangleleft$

Ein an einem Join-Knoten ankommendes Delta bewirkt ein Iterieren der Relation am jeweils anderen Join-Eingang. Zu beachten ist, daß die zugehörige Ausgangsvariante der iterierten Relation verwendet wird. Oder anders ausgedrückt, beim Iterieren müssen genau die Tupel geliefert werden, die sich auch aus dem Aufsummieren der bisherigen Deltas an diesem Ausgang ergeben würden. Die erwähnte Atomarität stellt sicher, daß die iterierte Relation während der Operation konstant bleibt.

Auch beim Differenz-Knoten muß auf die Ausgangsvariante der Relation am jeweilig anderen Eingang zugegriffen werden. Naturgemäß unterscheidet sich die Behandlung von Deltas am positiven und am negativen Eingang des Differenz-Knotens. Insbesondere wird ein insert-Delta am negativen Eingang zu einem delete-Delta am Ausgang, soweit es dort überhaupt ankommt, und entsprechend ein delete-Delta zu einem insert-Delta.

Folgende Definition faßt die Behandlung der Deltas in den verschiedenen Operationsknoten zusammen.

Definition 6.17 Die Anwendung eines Deltas auf einen Operationsknoten ist gegeben durch:

- Für eine Join-Operation $\mathrm{join}(a_1, a_2, r_3)$ mit $r_1 \leadsto a_1$ und $r_2 \leadsto a_2$:

$$\begin{aligned} \Delta(\frac{\mathrm{insert}}{\mathrm{delete}},\ t_1) \quad \bowtie \quad r_2^{(a_2)} \quad &:= \quad \{\Delta(\frac{\mathrm{insert}}{\mathrm{delete}},\ (t_1, t_2)) \quad | \quad t_2 \in r_2^{(a_2)}\} \\ r_1^{(a_1)} \quad \bowtie \quad \Delta(\frac{\mathrm{insert}}{\mathrm{delete}},\ t_2) \quad &:= \quad \{\Delta(\frac{\mathrm{insert}}{\mathrm{delete}},\ (t_1, t_2)) \quad | \quad t_1 \in r_1^{(a_1)}\} \end{aligned}$$

- Für eine Differenz-Operation $\mathrm{diff}(a_1, a_2, r_3)$ mit $r_1 \rightsquigarrow a_1$ und $r_2 \rightsquigarrow a_2$:

$$\Delta(\frac{\text{insert}}{\text{delete}}, t) \quad \setminus \quad r_2^{(a_2)} \quad := \quad \begin{cases} \{\Delta(\frac{\text{insert}}{\text{delete}}, t)\} & \text{falls} \quad t \notin r_2^{(a_2)} \\ \emptyset & \text{falls} \quad t \in r_2^{(a_2)} \end{cases}$$

$$r_2^{(a_2)} \quad \setminus \quad \Delta(\frac{\text{insert}}{\text{delete}}, t) \quad := \quad \begin{cases} \{\Delta(\frac{\text{delete}}{\text{insert}}, t)\} & \text{falls} \quad t \in r_1^{(a_1)} \\ \emptyset & \text{falls} \quad t \notin r_1^{(a_1)} \end{cases}$$

- Für eine Funktions-Operation $\mathrm{func}(f, a_1, r_2)$:

$$f(\ \Delta(\frac{\text{insert}}{\text{delete}}, t_1)\ ) \quad := \quad \{\Delta(\frac{\text{insert}}{\text{delete}}, t_2) \quad | \quad t_2 \in f(t_1)\}$$

◁

Diese Operationen auf Deltas sind im Ergebnis mengenwertig definiert. Join- und Funktions-Operationen liefern tatsächlich Mengen beliebiger Mächtigkeit zurück, bei Differenz-Operationen ist die Ergebnismenge entweder null- oder eins-elementig. Die Bearbeitung der gesamten Ergebnismenge ist dabei als atomarer Schritt im Reduktionsprozeß aufzufassen.

Nun sind wir in der Lage, den Ablauf eines Reduktionsschrittes vollständig zu beschreiben. Er besteht aus folgenden Aktionen:

1. Auswahl einer als unausgeglichen markierten Stelle.

2. Bildung eines entsprechenden Deltas gemäß dem Zustand von Hauptversion und Ausgangversion; Entfernung der Marke.

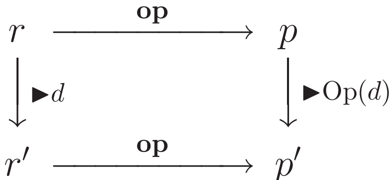
3. Anwendung des dahinterliegenden Operationsknotens auf das Delta gemäß obiger Definition.

4. Aufnahme der berechneten Deltamenge in die Zielrelation. Dazu werden die Zähler der betroffenen Tupel der Hauptversion entsprechend modifiziert. Alle Stellen derjenigen Tupel, bei denen der Zählerstand die Grenze zwischen Null und Eins nach oben oder unten überschreitet, werden als unausgeglichen markiert.

Der Reduktionsprozeß beginnt mit dem Eintrag von Deltas an extensionalen Relationen aufgrund imperativer **insert**- und **delete**-Aktionen, und endet, sobald alle Unausgeglichenheiten reduziert worden sind.

Die Korrektheit eines Reduktionsschrittes ergibt sich aus der Eigenschaft, einen lokal konsistenten Zustand in einen lokal konsistenten Zustand zu überführen. Dies läßt sich

an folgendem Diagramm verdeutlichen:



Hierbei bezeichnet $\mathbf{op}$ einen beliebigen Operationsknoten, also Join, Differenz oder Funktion. Die entsprechende, waagrecht dargestellte Beziehung ist die der lokalen Konsistenz. p ist Ergebnis einer Operation auf r. Mit d sei ein Delta bezeichnet, das die Relation r in den neuen Zustand r' überführt, zu dem wiederum ein konsistenter Zustand p' gehört. Die Aussage ist nun, daß dieser Zustand p' inkrementell berechnet werden kann, indem das Delta den Operationsknoten durchläuft und das Ergebnis auf p angewandt wird. Wir formulieren dies als Satz:

Satz 6.18 Sei $\mathbf{op}$ eine Operation auf einer Relation (Join oder Differenz mit konstantem zweiten Operanden oder eine Tupelfunktion). Sei r eine Relation und d ein Delta. Dann gilt:

$$\mathbf{op}(r) \blacktriangleright \mathbf{op}(d) \quad = \quad \mathbf{op}(r \blacktriangleright d)$$

Beweis:
Die Behauptung folgt leicht aus den bisherigen Definitionen. Wir zeigen das zunächst für die Join-Operation und ein insert-Delta. Sei $\mathbf{op}$ der Join von links mit einer konstanten Relation q und sei das Delta $d = \Delta(\mathrm{insert}, t)$. Hierbei ist q als die zur Join-Operation führende Ausgangsversion der entsprechenden Relation aufzufassen.

$$\begin{aligned}
\mathbf{op}(r) \blacktriangleright \mathbf{op}(d) &= (r \times q) \blacktriangleright \{\, \Delta(\mathrm{insert}, (t, t_q) \mid t_q \in q \,\} \\
&= (r \times q) \cup \{(t, t_q) | t_q \in q\} \\
&= (r \times q) \cup (\{t\} \times q) \\
&\overset{(*)}{=} (r \cup \{t\}) \times q) \\
&= (r \blacktriangleright d) \times q = \mathbf{op}(r \blacktriangleright d)
\end{aligned}$$

Wesentlich hierbei ist die mit $(*)$ gekennzeichnete algebraische Äquivalenz. Ähnliche Äquivalenzen gelten für delete-Deltas sowie für Differenz- und Funktions-Operationen. Es sei f eine Tupelfunktion. Für „$\circledast$“ kann sowohl „$\cup$“ für

insert-Deltas als auch „\“ für delete-Deltas eingesetzt werden:

Join-Operationen:

$$(r \circledast \{t\}) \times q = (r \times q) \circledast (r \times \{t\})$$

Differenz-Operationen:

$$(r \circledast \{t\}) \setminus q = \begin{cases} r \setminus q & \text{falls} \quad t \in q \\ (r \setminus q) \circledast \{t\} & \text{falls} \quad t \notin q \end{cases}$$

$$q \setminus (r \cup \{t\}) = \begin{cases} (q \setminus r) \setminus \{t\} & \text{falls} \quad t \in q \\ q \setminus r & \text{falls} \quad t \notin q \end{cases}$$

$$q \setminus (r \setminus \{t\}) = \begin{cases} (q \setminus r) \cup \{t\} & \text{falls} \quad t \in q \\ q \setminus r & \text{falls} \quad t \notin q \end{cases}$$

Funktions-Operationen:

$$f(\, r \circledast \{t\}\,) = f(r) \circledast f(t)$$

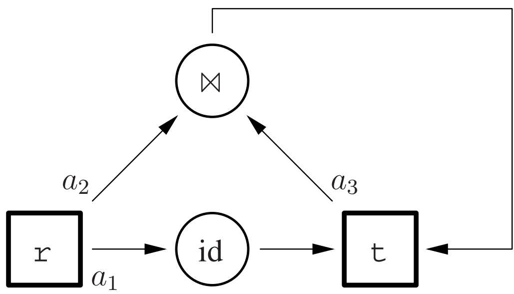
◁

Beispiel 6.19

Zur Verdeutlichung des Reduktionsprozesses verwenden wir wieder ein Netz zur Berechnung der transitiven Hülle, ähnlich wie im Beispiel 6.2.

```
t(X Y) ← r(X Y).
t(X Y) ← r(X Z)  t(Z Y).
```

Zur Vereinfachung werden Selektion und Projektion in den Join-Knoten gezogen, eine Hilfsrelation ist damit nicht mehr erforderlich. Die Ausgänge der Relation `r` sind mit a_1 und a_2 benannt, der Ausgang von `t` mit a_3.

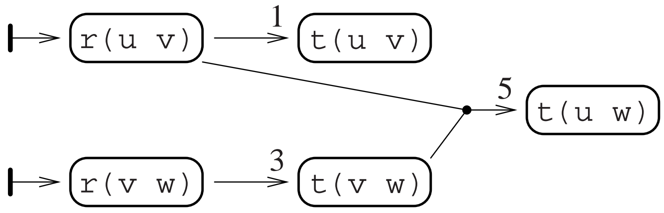


Ausgehend von leeren Relationen sollen nun die zwei extensionalen Fakten `r(u v)` und `r(v w)` eingetragen werden. Dadurch entstehen an den beiden Ausgängen von `r` jeweils zwei unausgeglichene Stellen, also insgesamt vier Marken. Eine mögliche Reduktionssequenz ist in Tabelle 6.2 dargestellt. Die hierbei verwendete Strategie arbeitet ältere Marken vor jüngeren Marken und bei Gleichaltrigkeit den Ausgang a_1 vor dem Ausgang a_2 ab.

Schritt	Aktion bzw. reduziertes Delta	unausgeglichene Stellen							Relationen und Ausgänge				
		$r^{(a_1)}(u\ \ v)$	$r^{(a_2)}(u\ \ v)$	$r^{(a_1)}(v\ \ w)$	$r^{(a_2)}(v\ \ w)$	$t^{(a_3)}(u\ \ v)$	$t^{(a_3)}(v\ \ w)$	$t^{(a_3)}(u\ \ w)$	r	$r^{(a_1)}$	$r^{(a_2)}$	t	$t^{(a_3)}$
									$r = \{\}$	$r^{(a_1)} = \{\}$	$r^{(a_2)} = \{\}$	$t = \{\}$	$t^{(a_3)} = \{\}$
	insert(u v)	•	•						$r = \{(u\ \ v)\}$				
	insert(v w)	•	•	•	•				$r = \{(u\ \ v), (v\ \ w)\}$				
1	$r^{(a_1)} : \Delta(\text{insert}, (u\ \ v))$		•	•	•	•				$r^{(a_1)} = \{(u\ \ v)\}$		$t = \{(u\ \ v)\}$	
2	$r^{(a_2)} : \Delta(\text{insert}, (u\ \ v))$			•	•	•					$r^{(a_2)} = \{(u\ \ v)\}$		
3	$r^{(a_1)} : \Delta(\text{insert}, (v\ \ w))$				•	•	•			$r^{(a_1)} = \{(u\ \ v), (v\ \ w)\}$		$t = \{(u\ \ v), (v\ \ w)\}$	
4	$r^{(a_2)} : \Delta(\text{insert}, (v\ \ w))$					•	•				$r^{(a_2)} = \{(u\ \ v), (v\ \ w)\}$		
5	$t^{(a_3)} : \Delta(\text{insert}, (u\ \ v))$						•						$t^{(a_3)} = \{(u\ \ v)\}$
6	$t^{(a_3)} : \Delta(\text{insert}, (v\ \ w))$							•				$t = \{(u\ \ v), (v\ \ w), (u\ \ w)\}$	$t^{(a_3)} = \{(u\ \ v), (v\ \ w)\}$
7	$t^{(a_3)} : \Delta(\text{insert}, (u\ \ w))$												$t^{(a_3)} = \{(u\ \ v), (v\ \ w), (u\ \ w)\}$

Tabelle 6.2: Verlauf der Reduktionsschritte in Beispiel 6.19

Das Ausgleichen benötigt sieben Reduktionsschritte und endet mit folgendem, als Tupelgraph dargestellten Zustand. Die Kanten sind mit der Nummer des Reduktionsschrittes gekennzeichnet, in dem sie entstanden sind.



◁

6.5 Vermeidung von Zirkelschlüssen

Hier wird zunächst exemplarisch gezeigt, wie Zirkelschlüsse entstehen können. Wir untersuchen, wie Zyklen erkannt werden können, und stellen dann zwei konkrete Vorgehensweisen vor, die anschließend verglichen werden.

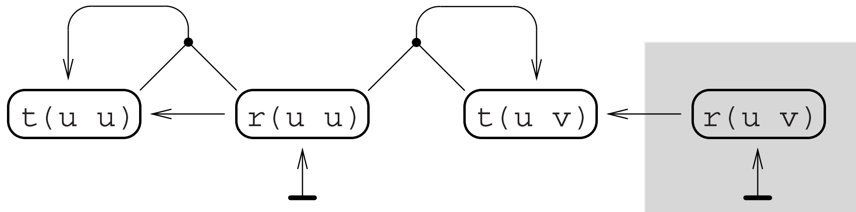
6.5.1 Entstehen und Auflösen von Zyklen

Selbsttragende Zyklen entstehen potentiell dann, wenn in einem positiven Zyklus des Relationen-Netzes ein Tupel seine eigene Begründung liefert. Wir verdeutlichen das an folgendem Beispiel.

Beispiel 6.20 Im Beispiel 6.10 ist ein lokal konsistenter Netzzustand angegeben, der nicht global minimal ist. Dieser Zustand kann, ausgehend von leeren Relationen, etwa durch folgende Aktionssequenz erreicht werden:

```
insert r(u u)
insert r(u v)
 --- erstes Ausgleichen ---
delete r(u v)
 --- zweites Ausgleichen ---
```

Nach dem Einfügen der zwei Tupel soll ein erstes Ausgleichen stattfinden. Dabei entsteht folgender Tupelgraph. Der grau hinterlegte Teil wird beim zweiten Ausgleichen gestrichen.



Das Tupel `t(u v)` wird im Reduktionsprozeß nicht gestrichen. Zwar wird sein Zähler von zwei auf eins gesetzt. Das Verfahren kann aber nicht erkennen, daß die verbleibende Unterstützung auf einem Zirkelschluß beruht.

◁

In diesem Beispiel hat der Zyklus eine Länge von eins. Im allgemeinen können die Zyklen aber beliebig lang werden. Die Zykluslänge im Tupelgraphen ist auch nicht durch die Länge der Zyklen im Relationen-Netz begrenzt. Denn letztere können unter Umständen mehrfach durchlaufen werden, bis sich der Kreis im Tupelgraphen schließt.

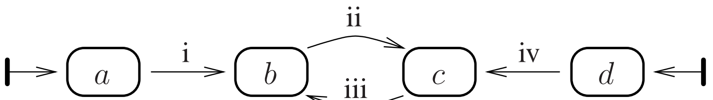
6.5.2 Zwei-Phasen-Löschen

Da mit den bisher eingeführten Mitteln aufgrund ihrer Beschränkung auf lokale Betrachtungen Zyklen nicht erkennbar sind, ist eine globales Vorgehen notwendig. Kritisch ist der Punkt, an dem durch eine delete-Operation dem Zyklus die letzte externe Begründung entzogen wird. Denn dann müssen auch die Fakten, die den Zyklus aufspannen, entfernt werden. Dazu muß aber der Zyklus als solcher erkannt sein.

Zum Erkennen von Zyklen gehen wir von folgender Idee aus: Wenn die Gefahr besteht, daß durch das Löschen eines Tupels andere Tupel in ungestützten Zyklen stehen bleiben, muß der Löschprozeß auf Verdacht weiterbetrieben werden und so die Chance erhalten, Zyklen aufzulösen. Hierbei können allerdings Tupel zunächst gelöscht werden, von denen sich danach herausstellt, daß sie doch noch eine echte Begründung besitzen. Deshalb werden diese Tupel markiert und dann in einer zweiten Phase erneut besucht. Wenn sie dann noch eine Begründung besitzen, werden sie wieder in Stand gesetzt.

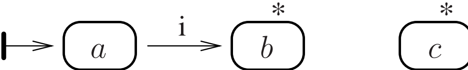
Insgesamt ist das Verfahren zur Vermeidung von Zyklen also dadurch gekennzeichnet, daß in einer ersten Phase Tupel gelöscht und in einer zweiten Phase Tupel eingefügt werden. Wir nennen das Verfahren deshalb *Zwei-Phasen-Löschen*.

Beispiel 6.21 Der Zustand eines Netzes sei durch folgenden Tupelgraphen gegeben, dessen Kanten zur einfacheren Bezugnahme numeriert sind:

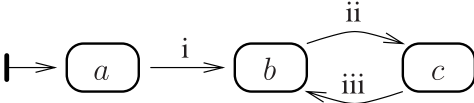


Es werde das Tupel d gelöscht. Der darauffolgende Reduktionsschritt entfernt die Kante iv. Aus lokaler Sicht auf das Tupel c ist nun unklar, ob die verbleibende Eingangskante ii aus einem ungestützten Zyklus stammt. Daher wird c vorsorglich als zu löschen angenommen und gleichzeitig als wieder zu besuchen markiert (in der Zeichnung mit einem Stern). Der nächste Reduktionsschritt

löscht Kante iii. Nun wird b als zu löschen angenommen und markiert. Im nächsten Schritt wird Kante ii gelöscht. Hier stoppt der Löschprozeß, da c bereits als gelöscht gilt.



Nun beginnt die zweite Phase, in der die markierten Tupel wieder besucht werden. Die Reihenfolge kann wieder beliebig gewählt werden. Wir beginnen mit Tupel c, das nun keine eingehende Kanten mehr besitzt. Also löschen wir es tatsächlich. Das Tupel b dagegen wird noch von Kante i gestützt und wird daher wieder eingefügt. Dies zieht zwei weitere Reduktionsschritte nach sich, durch die die Kante ii, das Tupel c und die Kante iii wieder eingefügt werden. Alle Markierungen sind abgearbeitet, die zweite Phase ist beendet.



Nun werde durch eine weitere Aktion das Tupel a gelöscht. Die erste Phase verläuft analog zu oben: Die Kanten i, ii und iii werden gelöscht, die Tupel b und c werden als wieder zu besuchen markiert.

In der zweiten Phase haben nun aber beide dieser Tupel keine Unterstützung mehr, sie werden also beide endgültig gestrichen. Der Zyklus ist damit korrekt aufgelöst. ◁

Der weiteren Darstellung des Zwei-Phasen-Löschen muß eine Bemerkung zu Reduktionsstrategien vorausgeschickt werden. Bislang wurde keine Aussage darüber getroffen, in welcher Reihenfolge die Einträge der Delta-Menge reduziert werden. Das bisher beschriebene Verfahren behält seine Eigenschaften unabhängig von der Auswahl des jeweils als nächstes zu reduzierenden Deltas. Andererseits wird sich zeigen, daß die Auswahlstrategie einen großen Einfluß auf die Effizienz des Verfahrens hat. Wir nennen das Modul, das die Delta-Menge verwaltet und die Auswahl des nächsten Deltas trifft, im folgenden *Delta-Scheduler* oder kurz Scheduler. Abschnitt 6.6 wird ausführlich auf verschiedene Scheduling-Strategien eingehen.

Die Idee des Zwei-Phasen-Löschens ist jedoch gerade eine Einschränkung der Delta-Auswahl: Zirkelschlüsse sollen aufgelöst werden, *bevor* die möglicheweise zu viel gelöschten Tupel wieder eingefügt werden. Ziel ist also ein Verfahren, das zum einen diese Beschränkung strikt einhält, zum anderen aber die Freiheit, die Scheduling-Strategie nach Effizienzaspekten zu wählen, nicht mehr als notwendig beschneidet.

Hierfür werden zwei Ideen benützt: Die Beschränkung auf starke Zusammenhangskomponenten, und die Unterscheidung zweier Delta-Mengen im Scheduler, entsprechend der zwei Löschphasen.

Starke Zusammenhangskomponenten (SCCs) wurden bereits in Abschnitt 4.2.1 zur Charakterisierung stratifizierbarer Programme eingeführt und können nun auf Relationen-Netze übertragen werden. Solange wir uns auf stratifizierbare Programme beschränken, sind die SCCs stets positiv (im Sinne der Definition 4.5).

Da sich aber herausstellen wird, daß das Verfahren auch für bestimmte nicht-stratifizierbare Programme geeignet ist, solange im Tupelgraph keine Zyklen über negative Kanten hinweg entstehen, berücksichtigen wir hier gleich den allgemeineren Fall. Von Interesse sind also nur positive Zyklen. Deswegen betrachten wir nicht die SCCs des gesamten Relationen-Netzes, sondern nur die der Restriktion des Netzes auf positive Kanten. Zur Unterscheidung nennen wir sie RSCC (restricted strongly connected component).

Definition 6.22 Sei N ein Relationen-Netz und G der Graph, der aus N dadurch entsteht, daß die Kanten zu den negativen Eingängen der Differenz-Knoten gestrichen werden. Eine $RSCC$ ist dann eine maximale Relationenmenge, innerhalb derer von jeder Relation zu jeder anderen ein gerichteter Pfad in G existiert. $\triangleleft$

Man beachte den Zusammenhang mit der wohl-fundierten Semantik: Die Restriktion auf positive Kanten ist der Teil des Netzes, in dem sich die innere Fixpunktiteration im Sinne der Definition 4.23 abspielt.

Zirkelschlüsse sind nur jeweils innerhalb einer RSCC möglich. Das spezielle Propagieren von delete-Deltas in der ersten Lösch-Phase kann daher auf die jeweils aktuelle RSCC beschränkt werden. Dies kann den Aufwand des Verfahrens erheblich reduzieren, da „zuviel-löschen“ nur noch innerhalb einer RSCC vorkommen kann.

Zur Realisierung des Verfahrens unterscheiden wir zwei Zustände, in denen sich die Auswertungsmaschine befinden kann. Die *Rotphase* liegt genau dann vor, wenn die erste Phase eines Zwei-Phasen-Löschen abläuft. Ansonsten befindet sich die Maschine in der *Grünphase*. Letztere ist also sowohl für die zweite Löschphase zuständig, als auch für z.B. das Propagieren von insert-Deltas.

Der Scheduler verwaltet nun zwei Delta-Mengen, eine rote und eine grüne. Rote Deltas sind immer delete-Deltas und werden vorrangig behandelt. Sobald ein delete-Delta in die rote Menge eingestellt wird, beginnt eine Rotphase, die solange andauert, bis alle roten Deltas reduziert sind.

Pro Rotphase braucht ein Tupel nur einmal speziell behandelt zu werden. Um bereits behandelte Tupel zu erkennen, werden zeitlich aufeinander folgende Rotphasen durch Numerierung unterschieden. Zudem erhält jedes Tupel ein Attribut, in dem die Nummer derjenigen Rotphase vermerkt ist, in der das Tupel zuletzt involviert war. Somit ist

leicht festzustellen, ob ein Tupel in der aktuellen Rotphase bereits behandelt worden ist.

Die Reduktion eines roten Deltas besteht aus folgenden Aktionen:

1. Die Auswahl eines roten Deltas ergibt ein bestimmtes Tupel an einem Relationenausgang.

2. Falls das Tupel nicht Element der entsprechenden Ausgangsversion ist, ist die Reduktion beendet.

3. Andernfalls wird das Tupel aus der Ausgangsversion entfernt und als delete-Delta propagiert.

4. Anwendung des dahinterliegenden Operationsknotens auf das Delta gemäß der Definition 6.17.

5. Aufnahme der berechneten delta-Menge in die Zielrelation. Dies geschieht pro Ergebnis-Delta in zwei Schritten:

   (a) Rotes Schedulieren: Falls das Tupel in der aktuellen Rotphase noch nicht behandelt wurde, wird es für jeden Ausgang, der zu einer Relation innerhalb der gleichen RSCC führt, als rotes Delta scheduliert.

   (b) Grünes Schedulieren: Zusätzlich wird das Delta für jeden Ausgang grün scheduliert, unabhängig davon, ob das Tupel Element der Ausgangsversionen ist.

Das im letzten Schritt genannte grüne Schedulieren realisiert die Kennzeichnung zum Wiederbesuch. Jedes von der Rotphase betroffene Tupel wird in einer späteren Grünphase nochmals besucht.

Die Reduktion eines grünen Deltas besteht nun aus folgenden Schritten:

1. Die Auswahl eines grünen Deltas ergibt ein bestimmtes Tupel an einem Relationenausgang.

2. Falls sich bei dem Tupel Hauptversion und Ausgangsversion nicht unterscheiden, ist die Reduktion beendet.

3. Falls die Differenz zwischen Hauptversion und Ausgangsversion ein delete-Delta ergibt und zusätzlich der Ausgang zu einer Relation in der gleichen RSCC führt, wird eine neue Rotphase eingeleitet. Dazu wird das Delta neu scheduliert, einmal als rotes delete-Delta und zudem nochmal als grünes Delta. Die ursprüngliche Reduktion ist damit beendet.

4. Andernfalls wird die Ausgangsversion der Hauptversion angepaßt und das entsprechende Delta propagiert.

5. Anwendung des dahinterliegenden Operationsknotens auf das Delta gemäß der Definition 6.17.

6. Aufnahme der berechneten delta-Menge in die Zielrelation. Dabei Schedulieren der entsprechenden Tupel an allen Ausgängen, die eine Differenz zur Hauptversion aufweisen. (Falls diese Stelle bereits scheduliert ist, wird sie nicht nocheinmal in den Scheduler eingestellt.)

6.5.3 Selektives Löschen mit Höheninformationen

Das dargestellte Verfahren verbleibt in der ersten Löschphase, solange die gelöschten Tupel eine weitere Stütze aus der gleichen RSCC besitzen. Um die Menge der zu viel gelöschten Fakten zu verringern, ist ein selektiveres Abbruchkriterium wünschenswert. In diesem Abschnitt schlagen wir die Verwendung von Höheninformationen vor, die mit der Länge von Ableitungspfaden arbeiten. Hierdurch kann der entstehende Löschüberhang verkleinert werden. Der dabei gewonnene Vorteil muß dann mit dem Aufwand zur Verwaltung dieser Informationen verglichen werden.

Jedes Tupel wird hierzu mit einer Höhe markiert. Eine Höhe ist eine natürliche Zahl, die die Länge der kürzesten Herleitung des Tupels angibt. Die Anzahl der Herleitungsschritte entspricht dabei der Anzahl der Anwendungen des unmittelbare-Konsequenzen-Operators (vergleiche Definition 4.19), die notwendig sind, um das Faktum zu erzeugen. Wir definieren die Höhe formal anhand des Tupelgraphen eines konsistenten Netzzustandes, indem wir die Konstruktion begründeter Tupel in Definition 6.11 um Höhen erweitern.

Definition 6.23 In einem konsistenten Netzzustand wird jedem Knoten des Tupelgraphen eine natürliche Zahl als Höhe zugeordnet:

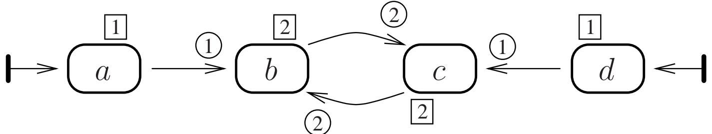
- Jedes extensionale Tupel erhält die Höhe eins.
- Jedes intensionale Tupel erhält die um eins inkrementierte Höhe des Minimums aller stützenden Kanten. Diese Kanten werden je nach Typ des erzeugenden Operationsknotens gemäß der folgenden Punkte bewertet.
- Der Join zweier Tupel wird mit dem Maximum der beiden Höhen bewertet.
- Das Ergebnis einer Differenz wird mit der gleichen Höhe wie das positiv stützende Tupel bewertet.
- Die von einem Funktionsknoten berechneten Tupel werden mit der gleichen Höhe wie das Ausgangstupel bewertet.

$\triangleleft$

Man beachte, daß Höhen nur für global minimale Zustände wohldefiniert sind. Innerhalb von Zirkelschlüssen kann den Tupeln keine eindeutige Höhe zugewiesen werden. Dies ist der Schlüssel dazu, Höhen als strengeres Abbruchkriterium der Löschphase zu nutzen.

Im Zwei-Phasen-Verfahren ist nach dem Wegfall einer Unterstützungskante ein vorsorgliches Weiterlöschen nur noch dann notwendig, wenn alle verbleibenden Kanten größere Höhen „mitbringen" als die wegfallende Kante. Existiert dagegen noch eine Unterstützung mit einer kleineren Höhe, so kann diese nicht Folge des betrachteten Faktums sein, also auch keinen Zirkelschluß manifestieren.

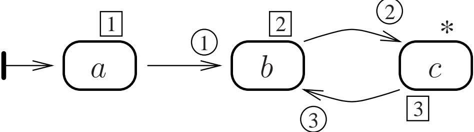
Beispiel 6.24 Wir erweitern die in Beispiel 6.21 dargestellte Situation um Höheninformationen. Zunächst sei der Zustand durch folgenden Tupelgraph gegeben:



Die Höhen der Fakten sind mit Kästchen umrandet eingezeichnet, die Höhen der eingehenden Kanten sind mit Kreisen umrandet. Gemäß obiger Definition ergeben sich die Höhen der Fakten aus dem Minimum der eingehenden Kanten plus eins.

Nun werde das Tupel d gelöscht. Damit entfällt am Tupel c die eingehende Höhe 1. Da diese die minimale, aber nicht einzige Höhe gewesen ist, wird weitergelöscht und c als wieder zu besuchen markiert. Als nächstes wird dem Tupel b die eingehende Höhe 2 entzogen. Dieses Tupel besitzt noch die kleinere Höhe 1, wird also nicht gelöscht. Die erste Phase ist abgeschlossen.

Beim Wiederbesuch von Tupeln in der zweiten Phase 2 wird deren Höhe aus den verbleibenden eingehenden Kanten neu bestimmt und propagiert. Hier erhält Tupel c die Höhe 3. Das Minimum an Tupel b wird dadurch nicht unterboten, somit bleibt dieses Tupel auf Höhe 2, und die zweite Phase ist beendet.



Nun werde als weitere Aktion auch Tupel a gelöscht. Dadurch wird dem Tupel b die minimale eingehende Höhe 1 gestrichen. Da es aber eine weitere Stütze besitzt, wird es als wieder zu besuchen markiert. Das Löschen wird fortgesetzt mit dem Entfernen von Tupel c und dem Streichen der zweiten Stütze von Tupel b.

Der anschließende Wiederbesuch von Tupel b findet keine eingehenden Kanten mehr vor. Folglich wird auch b endgültig gelöscht. ◁

Wie bereits am vorangegangen Beispiel ersichtlich muß das in Abschnitt 6.5.2 dargestellte Verfahren an einigen Stellen erweitert werden, um Höheninformationen zu verwalten und zu nutzen. Zunächst genügt es nicht mehr, in der Hauptversion einer Relation pro Tupel einen Zähler für die Anzahl der Stützen zu unterhalten. Statt dessen muß für jedes Tupel eine Multimenge der stützenden Höhen gespeichert werden. In den Ausgangsversionen wird zu jedem Tupel die zuletzt propagierte Höhe vermerkt. Deltas tragen nun statt dem $\mathrm{insert}/\mathrm{delete}$-Flag ein Höhenpaar (h_1/h_2) für bisherige Höhe und neue Höhe. Hierbei wird der spezielle Höhenwert ∞ mit der Bedeutung „nicht-vorhanden" eingesetzt.

In einem Reduktionsschritt können nun verschiedene Delta-Typen entstehen: Deltas, deren bisherige und neue Höhe identisch sind, werden wie bisher ignoriert. Deltas, die ein Tupel neu einfügen oder die Höhe eines vorhandenen Deltas verringern, werden normal propagiert. Deltas, die ein Tupel löschen oder die Höhe eines vorhandenen Tupels vergrößern, leiten ein Zwei-Phasen-Löschen ein, das analog zur Darstellung im vorherigen Abschnitt abläuft.

6.5.4 Vergleich der beiden Löschverfahren

Es stellt sich die Frage, ob der Mehraufwand der Höhen-Variante durch eine bessere Performanz gerechtfertigt wird. Da ein analytischer Vergleich der Verfahren sehr schwierig sein dürfte, haben wir das Laufzeitverhalten in einigen Experimenten untersucht.

Als Problemstellung wurde wieder die Berechnung der transitiven Hülle gewählt. In die zunächst leere Basisrelation werden sukzessive Tupel eingetragen, bis die Basisrelation eine totale Ordnung erzeugt. Nach jedem Eintrag wird die Anpassung der transitiven Hülle berechnet. In einem zweiten Schritt wird die Basisrelation in gleicher Weise wieder abgebaut. Die für Auf- und Abbau benötigten Berechnungszeiten werden getrennt gemessen.

Das Programm wurde in folgenden Punkten variiert:

- Art der Rekursion in der Hüllenberechnung:
  Lineare Rekursion:

```
t(X Y) ← r(X Y).
t(X Y) ← r(X Z)  t(Z Y).
```

Nichtlineare Rekursion:

```
t(X Y) ← r(X Y).
t(X Y) ← t(X Z)  t(Z Y).
```

- Bindungsmuster der Relationen `r` und `t`: Es wird entweder `(in out)` oder `(out out)` verwendet.
- Reihenfolge des Tupeleintrags in die Basisrelation. Eine Variante trägt die Tupel gemäß ihrer Sequenz ein, nach dem Muster (1 2), (2 3), (3 4), ...
  In der anderen Variante werden die Tupel in Schichten nach einem einfachen Schema eingetragen. Bei einer Problemgröße von acht lautet die Sequenz: (1 2), (3 4), (5 6), (7 8), (2 3), (6 7), (4 5).

Es werden alle Kombinationen dieser drei Variablen getestet, so daß sich insgesamt acht Versuche ergeben.

Als Scheduling-Strategie (siehe Abschnitt 6.6) wird eine Warteschlange gewählt. Tests ergaben, daß in diesem Beispiel die in Abschnitt 6.6.1 beschriebene horizontale Priorisierung kaum Einfluß auf die Effizienz hat.

Die Problemgröße wird so gewählt, daß sich eine Laufzeit im Sekundenbereich ergab. Es ergeben sich Größen zwischen 64 und 256 Punkten, die von den Relationen zu verbinden sind. Man beachte, daß die dem Problem inhärente Komplexität in der linearen Version quadratisch und in der nicht-linearen Version kubisch ist.

Die detailierten Ergebnisse sind in einer Tabelle im Anhang A.1 angegeben. Wir fassen hier die wesentlichen Punkte zusammen:

- Erwartungsgemäß ist beim Aufbau der transitiven Hülle die Zähler-Variante stets besser. Beim Abbau hingegen ist in sechs der acht Versuche die Höhen-Variante schneller.
- In der Summe von Auf- und Abbau sind in fünf der acht Versuche beide Verfahren vergleichbar performant. Hierbei gewinnt dreimal die Zähler-Variante mit bis zu 23%, zweimal die Höhen-Variante mit bis zu 51%.
- Für diese fünf Versuche wurde durch Variieren der Problemgröße die Komplexität der Berechnung gemessen. Bei der Verwendung von `(in out)`-Bindungsmuster bewegen sich die Ergebnisse beider Varianten in engen Grenzen um die inhärente Problemkomplexität $O(n^2)$ bzw. $O(n^3)$. Bei `(out out)`-Bindungsmuster sind die Ergebnisse teilweise deutlich schlechter.

- Bei den drei verbleibenden Versuchen kommt es in der Zähler-Variante allerdings zu auffälligen Ausreißern. Die ermittelten Zeiten liegen hier bei dem ca. 20- bis 100-fachen gegenüber der Höhen-Variante. Diese Werte kommen jeweils bei der Verwendung der `(out out)`-Bindungsmuster vor.

Eine genauere Analyse der letztgenannten Fälle ergibt, daß in der Abbauphase beim Löschen eines Tupel aus der Basisrelation viel zu große Teile der Hüllenrelation gelöscht werden, die dann wieder aufgebaut werden müssen. Die Ursache hierfür wird erst deutlich, wenn das magic-set-transformierte Programm betrachtet wird. Aufgrund der `(out out)`-Bindungsmuster sind die beiden magic-Relationen jeweils null-stellig. In gewissen Konstellationen kann es nun vorkommen, daß infolge des Löschens eines Basistupels auch das Tupel der `magic_p` Relation vorübergehend gelöscht wird. Währenddessen wird ungünstigerweise mit dem Abbau der gesamten Hüllenrelation begonnen. Der Fehler wird erst korrigiert, wenn in `magic_p` wieder das (null-stellige) Tupel eingetragen wird.

Insgesamt zeigt dieses Beispiel, daß im Normalfall beide Löschverfahren ähnlich performant sind, alldings nur solange sich die Zähler-Variante gutmütig verhält. Dies ist jedoch nicht immer der Fall, und die Gründe hierfür sind bereits in diesem einfachen Beispiel erst durch eine genauere Untersuchung zu erkennen. Ähnliche Ausreißer konnten wir bei der Höhen-Variante bis jetzt nicht beobachten.

Diese Ergebnisse werden auch durch unsere Erfahrungen mit größeren Beispielprogrammen bestätigt. Diese zeigten in der Zähler-Variante ein zumeist indiskutabel schlechtes Laufzeitverhalten, bzw. terminierten gar nicht innerhalb vernünftiger Zeiten. Beim Einsatz der Höhen-Variante ergaben sich dagegen keine unerwartet langen Laufzeiten.

Die Höhen-Variante des Löschverfahrens scheint also wesentlich robuster zu sein. Wir geben ihr daher für eine IDC-Implementierung eindeutig den Vorzug.

6.6 Scheduling-Strategien

Die Effizienz des Algorithmus ist in hohem Maße von der Strategie zur Auswahl des als nächstes zu reduzierenden Deltas abhängig. In diesem Abschnitt werden verschiedene Scheduling-Strategien miteinander verglichen.

Wir betrachten zunächst zwei einfache Strategien, die ohne Wissen um die Struktur des Relationen-Netzes arbeiten. In einer trivialen Strategie verhält sich der Scheduler als Keller: Zuletzt eingestellte Deltas werden als erstes reduziert. Dies entspricht der Vorgehensweise des originalen Rete-Algorithmus von Forgy [For82]. Dort gibt es allerdings kein explizites Scheduling. Die Abarbeitungsreihenfolge ergibt sich vielmehr implizit aus dem Algorithmus.

Eine alternative Strategie verwendet eine Warteschlange als Scheduler: Zuerst eingestellte Deltas werden auch als erste reduziert. Experimentelle Messungen (siehe Abschnitt 6.6.3) zeigen, daß dieser Ansatz gegenüber dem Keller-Ansatz im allgemeinen weit im Vorteil ist. Hieraus ergeben sich verallgemeinerbare Hinweise für effizientes Scheduling, denen im folgenden Abschnitt nachgegangen wird.

6.6.1 Horizontale Priorisierung

Scheduling mit einem Keller benötigt wesentlich mehr Reduktionsschritte um zum gleichen Endzustand zu gelangen als Scheduling mit einer Warteschlange. Da jeder Reduktionsschritt Folge eines Zustandswechsels einer Stelle ist, bedeutet dies, daß bei ungünstigem Scheduling Stellen ihren Zustand unnötig oft wechseln.

Mehrfache Zustandswechsel können verschiedene Auslöser haben, so etwa den folgenden Vorgang an einem Differenzknoten: Wird dort ein Tupel zunächst am positiven Eingang eingefügt und später zusätzlich am negativen Eingang, so wird es im Resultat des Differenzoperation zunächst eingefügt und später wieder gelöscht. Ähnlich kann sich die Höhe einer Stelle mehrfach ändern, wenn sich im Laufe des Reduktionsprozesses mehrere alternative, unterschiedlich lange Ableitungen des entsprechenden Tupels ergeben.

Diese Auslöser mehrfacher Zustandswechsel lassen sich kaum vermeiden. Wesentlich ist, wieviel Mehrarbeit sie in ihrer Folge nach sich ziehen. Diese entsteht dann, wenn der transiente Zustand im Netz weiter propagiert wird. Der dabei entstehende Berechnungsaufwand kann beliebig groß werden, so daß er in ungünstigen Fällen in keinem Verhältnis mehr zum eigentlich notwendigen Berechnungsaufwand steht.

Das Ziel eines effizienten Schedulings muß es somit sein, möglichst wenig Reduktionsschritte auf das Propagieren von instabilen Zwischenzuständen zu verwenden. Der Zustand einer Stelle sollte möglichst erst dann propagiert werden, wenn er sich im aktuellen Reduktionsprozeß nicht mehr ändert.

Mit diesen Überlegungen wird klar, daß die Verwendung eines Kellers in gewisser Weise die ungünstigste Strategie darstellt. Denn dort wird jede Veränderung eines Zustandes sofort bis zur letzten Konsequenz durchgerechnet, bevor der Zustand die Möglichkeit erhält, sich eventuell wieder zu ändern.

Bei der Verwendung einer Warteschlange hingegen werden erst alle gegenwärtig vorhandenen Deltas reduziert, bevor die dadurch neu entstehenden Deltas abgearbeitet werden. Dadurch werden Zwischenzustände, die sich innerhalb einer Delta-Warteschlange (etwa durch insert/delete-Paare) auflösen, nicht unnötig propagiert.

Die Strategie kann verfeinert werden, indem nun auch die Struktur des Relationen-Netzes ausgenützt wird (vergleiche Abbildung 6.3). Hierzu wird das Netz in RSCCs zerlegt, also in starke Zusammenhangskomponenten der Restriktion des Netzes auf

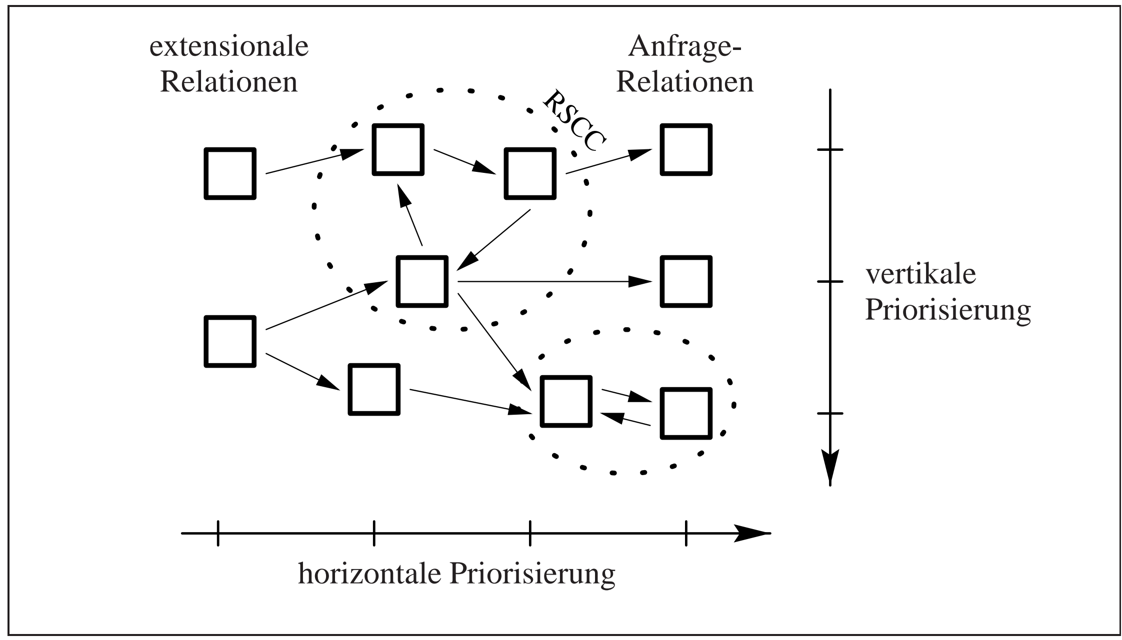


Abbildung 6.3: Scheduling-Strategien durch Priorisierung der Relationenausgängen

positive Kanten (vergleiche Abschnitt 6.5.2). Der Reduktionsprozeß wird nun RSCC-weise durchgeführt, beginnend bei den extensionalen Relationen und dann fortschreitend in Richtung abgeleitete Relationen. Auf diese Weise wird vermieden, daß transiente Zustände über RSCC-Grenzen hinweg propagiert werden.

Da in typischen Programmen RSCCs recht groß werden können, ist eine weitere Verfeinerung wünschenswert. Hier hat es sich als hilfreich erwiesen, zwischen Magic-Relation und „normalen“ Relationen zu unterscheiden und erstgenannte vorrangig zu behandeln. Die Idee hinter dieser Heuristik ist, zunächst die Magic-Relationen zu stabilisieren, da sie ja in gewisser Weise die Eingabedaten für die Berechnung der normalen Relationen liefern. Zur Durchführung wird jede RSCC in zwei Teilnetze zerlegt, eines für die Magic-Relationen und eines für die normalen Relationen. Diese Teilnetze werden dann wiederum in RSCCs zerlegt. Insgesamt ergibt sich damit eine relativ feine Aufgliederung der Relationen (bzw. eigentlich der Relationenausgänge) in Prioritätsklassen. Innerhalb einer Klasse wird dann unter Verwendung einer Warteschlange scheduliert.

6.6.2 Vertikale Priorisierung

Bei den bisher genannten Scheduling-Strategien wird immer das gesamte Netz ausgeglichen. Dies ist aber im allgemeinen nicht notwendig. Der Reduktionsprozeß dient dazu, die von der Handlungsbasis an die Wissensbasis gestellten Anfragen zu beantworten. Die Anfragen werden von Threads gestellt, die eine überwachte Auswahl bearbeiten. Da aber Threads untereinander priorisiert sind, ergeben sich auch Prioritäten

für die Anfragen. Die Idee der vertikalen Priorisierung ist es, das Netz nur soweit auszugleichen, bis bekannt ist, welcher Thread mit welchem Anfrageergebnis fortgesetzt werden muß.

Zur Realisierung ist zunächst zu bemerken, daß Threads, obwohl sie ihre Priorität dynamisch ändern können, ihre Anfragen mit einer festen Priorität stellen, die also unabhängig vom Anfrageergebnis ist. Somit besitzt jede Anfragerelation während eines Reduktionsprozesses eine feste Menge von Prioritätsangaben. Indem nun jeweils das Maximum dieser Menge im Relationen-Netz zurückpropagiert wird, erhält jeder Relationenausgang einen aktuellen Prioritätswert. Die dadurch in Prioritätsklassen eingeteilten Relationenausgänge werden nun von oben her reduziert, solange bis sich eine nicht-leere Anfragerelation ergibt. Damit ist der höchst-priorisierte, rechenbereite Thread gefunden.

Die vertikale Priorisierung ist nicht nur als eine Optimierung zur allgemeinen Effizienzsteigerung zu sehen. Sie ist vielmehr auch ein Mittel, die Reaktivität der Berechnungsmodells sicherzustellen. Mit ihr werden die auf der Seite der Handlungsbasis vergebenen Prioritäten in die Rechenzeitvergabe der Wissensbasis übertragen. Aufwendige, möglicherweise lang andauernde Ableitungsprozesse in der Wissensbasis werden damit unterbrechbar gemacht für vorrangig zu erkennende Situationen. Eine schnelle Reaktion auf solche Situationen wird erst durch die vertikale Priorisierung ermöglicht.

6.6.3 Experimentelle Bewertung

Um den Einfluß der Scheduling-Strategie auch quantitativ abzuschätzen, wurden einige Experimente zur Laufzeitbestimmung durchgeführt. Dabei werden drei Strategien verglichen: Scheduling per Keller, Scheduling per Warteschlange und die beschriebene horizontale Priorisierung. Da in der Implementierung, an der die Messungen durchgeführt wurden, die vertikale Priorisierung noch nicht realisiert war, können wir hier keine Zahlen vorlegen.

Es werden drei Anwendungsbeispiele getestet:

1. Berechnung der transitiven Hülle (vergleiche 6.5.4)
2. n-Damen Problem (formuliert als deklaratives Logikprogramm zur Berechnung aller Lösungen)
3. Ein Systemmodul zur SCC-Bestimmung

Das dritte Beispiel bedarf einer Erläuterung: Die horizontale Priorisierung wird in der verwendeten Implementierung nicht nur durch C++ Code realisiert, sondern auch durch ein in Sita geschriebenes Modul, das die Berechnung der SCCs sowie deren

Stratifizierung übernimmt. Dieses Vorgehen hat die Implementierung wesentlich vereinfacht. Zum einen ist die Kodierung in der Hochsprache Sita wesentlich einfacher zu bewerkstelligen, zum anderen verhält sich die Berechnung der horizontalen Prioritäten durch ein Wissensbasisprogramm automatisch inkrementell: Sobald dem Relationennetz neue Programmteile als Module hinzugefügt werden, wird die Stratifizierungsberechnung der neuen Situation angepaßt. Hier werden die Reaktivität und die Inkrementalität von Sita ausgenutzt. Zur Laufzeitbestimmung wurde nun gemessen, welche Zeit dieses Systemmodul zur Analyse seiner selbst und einiger weiterer Module benötigt.

Alle Messungen wurden mit der Höhen-Variante als Löschverfahren durchgeführt.

Bereits im ersten Anwendungsbeispiel zeigt sich der Vorteil einer Warteschlange gegenüber einem Keller, siehe Anhang A.2. In der nichtlinearen Version kann dadurch die Zeit für den Aufbau der transitiven Hülle wesentlich beschleunigt werden. Die ist wohl darauf zurückzuführen, daß beim Scheduling per Warteschlange die kürzesten Ableitungen der Wege schneller gefunden werden und somit eine Anpassung der entsprechenden Höhen seltener erforderlich ist.

Ansonsten ist in diesem Beispiel keine wesentlicher Einfluß der Scheduling-Strategie auf das Laufzeitverhalten zu erkennen. Insbesondere zeigt die horizontale Priorisierung gegenüber der Warteschlange praktisch keinerlei Veränderung (in der Tabelle nicht dargestellt). Das Beispiel scheint hierfür zu klein zu sein; das Relationennetz ist zu flach.

Im Unterschied dazu liegen bei den beiden komplexeren Anwendungen Größenordnungen zwischen den verschiedenen Strategien. Die Ergebnisse finden sich wiederum in Anhang A.2. Der gemessene Speedup von Warteschlange gegenüber Keller beträgt im zweiten Beispiel Faktor 20, im dritten Beispiel mindestens[1] Faktor 30. Durch die Anwendung der horizontalen Priorisierung ergibt sich nochmals eine Beschleunigung um den Faktor 5 bzw. 2.4.

Insgesamt zeigt sich der große Einfluß der Scheduling-Strategie auf die Performanz des IDC-Algorithmus. Das beste bisher gefundene Verfahren ist die beschriebene horizontale Priorisierung. Gegenüber der einfachen Keller-Strategie lassen sich so Beschleunigungen von mehreren Größenordnungen erzielen.

6.7 Weitere Optimierungen

Zwei weitere Optimierungsaspekte sollen hier kurz angesprochen werden, die einen entscheidenden Einfluß auf die Effizienz haben können: Wahl der Join-Order und Verwendung von aggregierenden Operatoren.

[1]Der Lauf mit Keller-Strategie wurde nach 15 Minuten abgebrochen.

6.7.1 Join-Order-Optimierung

Die Join-Order-Optimierung ist eine Standardtechnik in relationalen Datenbanken (vergleiche etwa [Smi85], [SG88]). Sind bei einer Anfrage drei oder mehr zu joinende Relationen beteiligt, so kann für die Berechnung zwischen unterschiedlichen Join-Strukturen gewählt werden, die sich im wesentlichen in der Reihenfolge der Relationen im Operatorbaum unterscheiden. Optimierungskriterien sind hier zum einen die Anwendbarkeit effizienter Join-Techniken wie z.B. Index-Joins und zum anderen die Erzielung möglichst kleiner Zwischenergebnisse. Hierzu gehört auch die Heuristik, Selektionen und Projektionen möglichst frühzeitig im Operatorbaum anzuwenden.

Diese Techniken sind genauso bei der Übersetzung eines Logikprogrammes in ein effizientes Relationen-Netz anwendbar. Durch die inkrementelle Auswertung ergibt sich jedoch als zusätzlicher Aspekt die erwartete Häufigkeit von Veränderungen an den Relationen. Dadurch kommt als Heuristik dazu, Relationen mit einer hohen Modifikationsfrequenz möglichst weit hinten im Operatorbaum „einzujoinen".

Informationen über erwartete Größen und Modifikationsfrequenzen der Relationen können entweder durch den Programmierer in Form von Hinweisen an den Compiler vorgegeben werden, oder sie werden durch Messungen in Testläufen des Programmes abgeschätzt. Letztere Methode wird von Ishida in [Ish94] im Zusammenhang mit OPS5 Produktionensystemen untersucht.

In [Per97] wird die Realisierung eines Sita-Compilers beschrieben, in der etliche Heuristiken zur Gestaltung effizienter Netzstrukturen zur Anwendung kommen. Der Einsatz dynamischer Aspekte, wie Relationengröße und Modifikationsfrequenz, ist im Zusammenhang mit Sitas IDC-Algorithmus bisher nicht untersucht worden.

6.7.2 Aggregate

Aggregatsfunktionen, wie *min*, *sum*, *count*, sind z.B. aus SQL bekannt. Ihre Funktion ist es, aus einer *Menge* von Tupeln einen Wert zu berechnen. Ähnliche Funktionen werden in deduktiven Datenbanken angeboten (vergleiche [BNR+87], [NT89], [RRS+93], [KS91]). Durch ihre Verfügbarkeit wird zum einen die Ausdrucksstärke erhöht, zum anderen dienen sie teilweise auch der Effizienzsteigerung. Dieser zweite Aspekt soll hier kurz dargestellt werden.

Eine häufig anzutreffende Situation ist die Notwendigkeit, aus einer Relation das nach einer gewissen Ordnung minimale Faktum zu ermitteln. Dies kann wie im folgenden Beispiel bereits ohne Aggregatsfunktionen durch den Einsatz der Negation erzielt werden.

Beispiel 6.25 Für die Relation `element(Id Val)` soll der kleinste vorkommende Wert `Val` berechnet werden. Als Ordnung wird ein Builtin „<" zugrundegelegt.

```
minimal(Val) ← element(Id Val)   ¬ smaller(Val).
smaller(Val) ← element(Id1 Val1)  Val1 < Val.
```

◁

Nachteilig bei dieser Lösung ist das schlechte Effizienzverhalten. Die Auswertung des Beispielprogrammes erfordert im wesentlichen einen Self-Join der `element` Relation und ist deswegen von quadratischer Komplexität. Das Minimumsproblem ist aber, etwa durch den Einsatz einer Heap-Datenstruktur, durch Algorithmen der günstigeren Komplexitätsklasse $O(n \log(n))$ zu lösen.

Es bietet sich daher an, die Sprache um entsprechende Aggregatsfunktionen zu ergänzen — gleichsam als Builtins über Relationen, die dann in der Maschine durch entsprechende effiziente Algorithmen implementiert werden können.

Beispiel 6.26 In der Notation von LDL [NT89] kann obiges Beispiel unter Verwendung der Aggregatsfunktion `min` geschrieben werden:

```
minimal(min<Val>) ← element(Id Val).
```

◁

Aggregatsfunktionen sind genauso wie die Negation nicht-monotone Konstrukte und bedürfen entsprechender Behandlung bezüglich ihrer Semantik ([KS91], [SSRB93]) und Auswertung ([RRSS94]).

6.8 Nicht-stratifizierbare Programme

6.8.1 Modulare Stratifikation

Der IDC-Algorithmus wurde bisher nur im Hinblick auf global stratifizierbare Programme betrachtet. Beim Verlassen dieser Klasse ist die Konvergenz des Algorithmus in Gefahr, insbesondere dann, wenn im realisierten Tupelgraph ein negativer Zyklus entsteht. Zur Demonstration nehmen wir das Beispiel 4.14 wieder auf:

Beispiel 6.27

```
p(Y) ← ¬ p(X) q(X Y).
```

Im Faktenspeicher sei zusätzlich das Tupel `q(a a)` eingetragen. Bei der Auswertung dieses nicht stratifizierbaren Programmes verfängt sich der Algorithmus in einem Zyklus: Aus der Ungültigkeit von `p(a)` wird auf die Gültigkeit von `p(a)` geschlossen und umgekehrt. Die Berechnung terminiert nicht. (Im wohlfundierten Modell besitzt `p(a)` den Wahrheitswert „undefiniert“.) ◁

Wir sind nun an weicheren Bedingungen als der globalen Stratifikation interessiert, unter denen der IDC-Algorithmus anwendbar ist.

Da ist zunächst die lokale Stratifizierbarkeit zu nennen, vergleiche Definition 4.10. Die Konvergenz der Auswertung ist nicht durch negative Zyklen im Prädikat-Abhängigkeitsgraph gefährdet, sondern durch solche im Fakten-Abhängigkeitsgraph. Der IDC-Algorithmus ist somit auch auf lokal stratifizierbare Programme anwendbar.

Dies ist jedoch eine noch immer unnötig restriktive Bedingung, da sie fordert, daß im gesamten Atom-Abhängigkeitsgraph keine negativen Zyklen vorkommen. Sie betrachtet also alle möglichen Interpretationen der Prädikate, unabhängig davon, welche Teile des Graphen in Abhängigkeit des Faktenspeichers tatsächlich aufgespannt werden.

So kann etwa obiges Beispielprogramm, obwohl nicht lokal stratifizierbar, problemlos ausgewertet werden, solange q eine nicht-zyklische Relation darstellt. Wie kann also die Grenze der Anwendbarkeit des IDC-Algorithmus formalisiert werden?

Es zeigt sich, daß eine schwächere Bedingung, nämlich die von Ross eingeführte *modulare Stratifikation* [Ros90] ausreicht. Sie beruht allerdings nicht nur auf dem statischen Programm sondern auch auf dem Inhalt des Faktenspeichers und kann deswegen nicht vorab, etwa durch den Compiler entschieden werden. Vielmehr geht es um die Zusage, daß ein inklusive Faktenspeicher modular stratifizierbares Programm vom IDC-Algorithmus korrekt ausgewertet wird.

Die Idee der modularen Stratifikation ist es, den Atom-Abhängigkeitsgraphen auf jene Teile zu beschneiden, die gemäß dem aktuellen Faktenspeicher bei der Vorwärtsverkettung tatsächlich besucht werden. Lassen sich dadurch alle negativen Zyklen vermeiden, ist das Programm modular stratifizierbar. Die Beschneidung geschieht SCC-weise mit Hilfe des folgenden Reduktionsbegriffs.

Definition 6.28 [Ros90] Sei S eine SCC des Prädikat-Abhängigkeitsgraphen eines Programmes P und sei L die Menge aller Prädikate aller SCCs tiefer als S (vergleiche Definition 4.5). Sei M eine Interpretation der Prädikate in L.

Die *Reduktion von S modulo M* wird folgendermaßen gebildet:

- Nimm alle Regeln, deren Köpfe Prädikate in S sind.
- Bilde alle Grundinstanzen dieser Regeln.
- Entferne alle Regeln, deren Rümpfe aufgrund von M nicht erfüllt sind.
- Streiche aus den Rümpfen der Regeln alle Literale, deren Prädikate in L sind.

$\triangleleft$

Im Anschluß an die Reduktion enthalten die Regeln nur noch Prädikate aus S. In gewisser Weise sind die Regeln gemäß M partiell evaluiert.

Beispiel 6.29 Wir betrachten wieder das Programm aus obigem Beispiel, nun mit einer azyklischen Relation q:

```
p(Y) ← ¬ p(X) q(X Y).
q(a b).
q(b c).
q(c d).
```

Das Programm besteht aus zwei SCCs. Betrachte $S = \{\texttt{p}\}$ und somit $L = \{\texttt{q}\}$. Die Menge $M = \{\texttt{q(a b)}, \texttt{q(b c)}, \texttt{q(c d)}\}$ ist eine Interpretation zu L. Die Reduktion von S modulo M ist:

```
p(a) ← ¬ p(b).
p(b) ← ¬ p(c).
p(c) ← ¬ p(d).
```

◁

Definition 6.30 [Ros90] Ein Programm P heißt modular stratifizierbar, wenn für jede SCC S seines Prädikat-Abhängigkeitsgraphs gilt:

- Die Vereinigung aller SCCs tiefer als S besitzt ein totales, wohl-fundiertes Modell.
- Die Reduktion von S modulo M ist lokal stratifizierbar.

◁

Lemma 6.31 [Ros90] Jedes modular stratifizierbare Programm besitzt ein totales, wohl-fundiertes Modell. ◁

Beispiel 6.32 In obigem Beispiel besitzt die tiefste SCC, also das Prädikat q, ein totales Modell, nämlich das angegebene M. Zudem ist die Reduktion von S modulo M lokal stratifizierbar. Also ist das Programm modular stratifizierbar. Sein totales Modell ist $\{\texttt{q(a b)}, \texttt{q(b c)}, \texttt{q(c d)}, \texttt{p(a)}, \texttt{p(c)}\}$. ◁

Die Reduktion einer fraglichen SCC S modulo der bereits ausgewerteten tieferen SCCs liefert also einen formalen Begriff für die Menge der bei der Auswertung von S tatsächlich vorkommenden Fakten. Sind diese Reduktionen stets lokal stratifizierbar, so konvergiert auch ihre Auswertung per IDC-Algorithmus.

Voraussetzung ist allerdings, daß die Auswertung der SCCs in einer mit ihrer Ordnung verträglichen Reihenfolge geschieht. Dies wird aber gerade durch die horizontale Priorisierung (siehe Abschnitt 6.6.1) gewährleistet.[2] Auch die inkrementelle Ausführung behindert die Konvergenz nicht, da sich die (inkrementelle) Anpassung einer SCC immer auf die bereits vollständig ausgewerteten, tieferen SCCs abstützt.

[2]Man beachte hierbei, daß die dort verwendete Ordnung auf den RSCCs eine Verfeinerung der Ordnung auf den SCCs ist.

Zusammenfassend kann also folgende Aussage gegeben werden: Ist während eines Sita-Programmlaufes das Wissensbasisprogramm zusammen mit dem Faktenspeicher zu jedem Zeitpunkt (das heißt nach jeder Update-Operation) modular stratifizierbar, so ist bei Anwendung der horizontalen Priorisierung ein nicht-Terminieren des IDC-Algorithmus aufgrund von sich selbst verneinenden Fakten ausgeschlossen.

6.8.2 Normale Programme

Ist ein Programm nicht modular stratifizierbar, so besitzt es im allgemeinen kein totales sondern nur ein partielles wohl-fundiertes Modell. Hier ist der IDC-Algorithmus in der dargestellten Form nicht geeignet.

Andererseits liefert die konstruktive Definition des wohl-fundierten Modells (siehe Lemma 4.25) auch das Prinzip zur Auswertung normaler Programme, wie in [Van93] dargestellt.

Ähnlich wie in Abschnitt 6.1 bei der einfachen Fixpunktiteration kann auch hier die Berechnung weitgehend differentiell durchgeführt werden [KSS95]. In diesem Blickwinkel scheint es möglich zu sein, eine inkrementelle Version der alternierenden Fixpunktiteration abzuleiten, wie sie Voraussetzung für den Einsatz im Sita-Berechnungsmodell wäre. Erste Schritte hierzu wurden in [Ung97] unternommen. Die Arbeiten hierzu sind noch nicht abgeschlossen.

Es stellt sich allerdings die Frage, ob der Zuwachs an Ausdrucksstärke den erhöhten Aufwand rechtfertigt. Während für stratifizierbare Programme eine „einfache“ Fixpunktiteration ausreicht, ist hier nun eine verschachtelte Iteration durchzuführen: Für jeden äußeren Iterationsschritt ist ein innerer Fixpunkt zu berechnen. Obgleich sich aufgrund von Monotonieeigenschaften der Fixpunkte die meisten Berechnungswiederholungen vermeiden lassen [KSS95], muß dennoch mit einem erheblichen Mehraufwand gerechnet werden.[3]

Andererseits ist unklar, ob sich die erhöhte Ausdrucksstärke in Sita-Anwendungen tatsächlich vorteilhaft nutzen ließe. Modellierungen, bei denen sich die logische Herleitung von Fakten auf die Negation jener selben Fakten stützt, erscheinen im allgemeinen wenig intuitiv. Vorteilhaft wäre es andererseits, daß ein nicht-Terminieren aufgrund solcher negativen Zyklen ausgeschlossen wäre.

6.9 Verwandte Arbeiten

Die bottom-up Auswertung von Logikprogrammen findet in der Literatur zwei Hauptanwendungsfelder, Produktionensysteme und deduktive Datenbanken. Wir vergleichen kurz einige der dort entwickelten Ansätze mit dem IDC-Algorithmus.

[3] Mit dieser Problematik beschäftigen sich etliche neuere Arbeiten, etwa [BSF95], [ZF96], [ZBF97],

Produktionensysteme verlangen aus Effizienzgründen nach einer inkrementellen Auswertung, deren Grundprinzip durch den bereits mehrfach erwähnten Rete-Algorithmus [For82] eingeführt wurde.

Eine Variante davon stellt der Treat-Algorithmus [Mir87] dar, der statt binären Join-Knoten mit festgelegter Join-Order (vergleiche Abschnitt 6.7.1) Mehrwege-Joins einsetzt, um dann die Join-Order zur Laufzeit zu optimieren. In [Han93] wurde unter dem Namen Gator-Algorithmus eine Vereinheitlichung der beiden Verfahren vorgenommen.

Keines dieser Verfahren ist zur korrekten Auswertung rekursiver Logikprogramme geeignet. Das Produktionensystem CLIPS[4] [Cul89] erlaubt neben Produktionen auch die Angabe hornklauselähnlicher Logikregeln. Dadurch wird zwar auf Sprachebene die Formulierung rekursiver Prädikate möglich, der eingesetzte Algorithmus trifft jedoch keine Vorkehrungen, entstehende Zirkelschlüsse aufzulösen.

Arbeiten, die den Einfluß verschiedener Scheduling-Strategien des Reduktionsprozesses auf die Effizienz untersuchen, sind uns nicht bekannt.

Im Bereich der deduktiven Datenbanken erfolgt die Auswertung im allgemeinen zwar bottom-up, aber nicht inkrementell. Ausnahmen laufen unter den Stichworten View-Update und aktive Datenbanken (vergleiche [Wid93]). Eine Übersicht wird in [GM95] gegeben. In [GMS93] wird der DRed[5]-Algorithmus vorgestellt, der im wesentlichen der Zähler-Variante des Zwei-Phasen-Löschens entspricht. Das gleiche trifft auf den PF[6]-Algorithmus [HD92] zu. Keine der uns bekannten Arbeiten verwendet einen Ansatz ähnlich der Höhen-Variante, um den Löschüberhang des Zwei-Phasen-Löschens zu begrenzen.

[4] „C-language integrated production system“
[5] „Deletion and Rederivation“
[6] „Propagation / Filtration“

KAPITEL 7

Anwendungsbeispiel

Wir zeigen in diesem Kapitel ein Anwendungsbeispiel für Sita auf. Ziele sind dabei, einen Einblick in die Praxis der Sita-Programmierung zu geben, die Funktionstüchtigkeit der Konzepte zu verdeutlichen und ihre Vorteile gegenüber alternativen Sprachkonzepten aufzuzeigen.
Das Beispiel, eine reaktive Wegeplanung, erfordert eine Kombination von reaktivem und zielgerichteten Verhalten. Wir stellen Lösungen in Magsy, AgentSpeak(L) und Sita gegenüber.

7.1 Szenario: Reaktive Wegeplanung

Wir übernehmen aus [Rao96] leicht abgewandelt folgendes Beispiel, das trotz seiner Einfachheit wichtige Aspekte verdeutlicht:

In einer Welt aus miteinander verbundenen Räumen bewegt sich ein Roboter mit der Aufgabe, Abfall aufzusammeln und in einen Papierkorb zu befördern, siehe Abbildung 7.1. In den gleichen Räumen bewegen sich zudem Personen, denen der Roboter ausweichen muß. Sobald eine Person den Raum betritt, in dem sich der Roboter aufhält, muß dieser den Raum verlassen. Außerdem soll er keinen Raum betreten, in dem sich eine Person aufhält.

Der Roboter, dessen Steuerung in diesem Beispiel realisiert werden soll, kennt neben seiner eigenen Position auch die von Abfall, Papierkorb und Personen. Zudem sei ihm bekannt, welche Räume benachbart sind. Wir nehmen also an, daß der Roboter allwissend ist. Später werden wir diese Voraussetzung einschränken.

Als Aktionen stehen dem Roboter neben dem Aufnehmen und Ablegen des Abfalls die Fahrt zwischen benachbarten Räumen zur Verfügung.

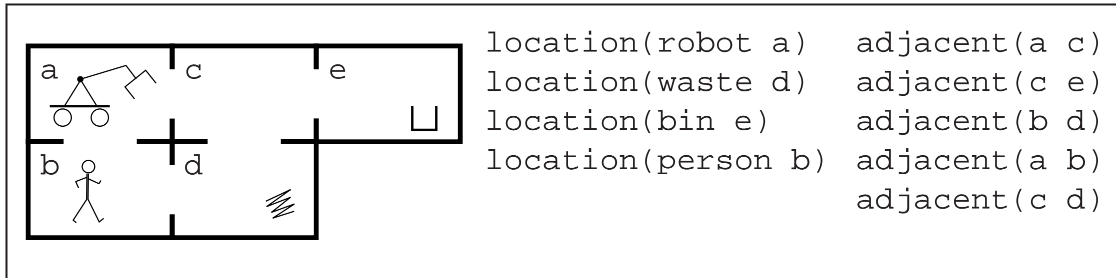


Abbildung 7.1: Szenario zur Wegeplanung

Zur Formalisierung verwenden wir folgende Prädikate bzw. Aktionen (die Variablen `X` und `Y` stehen für Raumbezeichnungen):

Wahrnehmung:	Aktionen:

`location(robot X)`
`location(waste X)`
`location(bin X)`
`location(person X)`
`adjacent(X Y)`
	`pickup()`
	`drop()`
	`move(X)`

Die Aufgabenstellung läßt sich in drei Verhaltensmuster oder Pläne zerlegen:

(VM1)	Abfall beseitigen:	Wenn der Roboter über die Position eines Abfalls informiert ist, soll er eine Sequenz von Handlungsschritten einleiten: Zum Abfall fahren, diesen aufnehmen, zum Papierkorb fahren, Abfall ablegen.
(VM2)	Zielort anfahren:	Das Fahren zu einer Zielposition, das als „Unterverhalten“ von VM1 verwendet wird, muß als Folge von Einzelbewegungen ausgeführt werden.
(VM3)	Person ausweichen:	Sobald der Roboter an seinem aktuellen Aufenthaltsort eine Person sichtet, muß er mit der Fahrt in einen benachbarten Ort reagieren.

Bevor wir näher auf die Realisierung dieses Verhaltens in Sita eingehen, skizzieren wir entsprechende Lösungen unter Verwendung der in Kapitel 2 vorgestellten Sprachen Magsy (Abschnitt 2.3.1, Seite 13) und AgentSpeak(L) (Abschnitt 2.3.3, Seite 16). Dies erlaubt uns, die Unterschiede dieser drei Sprachkonzepte zu verdeutlichen.

7.2 Realisierung in Magsy

Abbildung 7.2 zeigt eine Kodierung als Magsy-Programm.[1] Jedes der drei oben genannten Verhaltensmuster wird durch eine Menge von Produktionen realisiert. Die er-

[1] Zur leichteren Lesbarkeit haben wir die Syntax dem Stil von Sita angepaßt.

```
(   state(free)
    location(waste X)
==>
    delete state(free)
    insert state(pickup X)
    insert goto(X)
)

(   state(pickup X)
    location(robot X)
    location(bin Y)
==>
    call   pickup()
    delete state(pickup X)
    insert state(drop Y)
    insert goto(Y)
)

(   state(drop Y)
    location(robot Y)
==>
    call   drop()
    delete state(drop Y)
    insert state(free)
)
```

```
(   goto(X)
    location(robot X)
==>
    delete goto(X)
)

(   goto(X)
    location(robot Y)
    X <> Y
    adjacent(Y Z)
    ¬ location(person Z)
==>
    call move(Y Z)
)

"high priority:"
(   location(robot X)
    location(person X)
    adjacent(X Z)
==>
    call move(X Z)
)
```

Abbildung 7.2: Magsy-Version der Wegeplanung

sten drei Regeln (in der Abbildung links) kodieren das sequentielle Durchlaufen der Schritte von VM1. Da hierbei an drei Stellen auf ein Ereignis zu warten ist (das Wahrnehmen von Abfall, das Erreichen des Abfalls und das Erreichen des Papierkorbes), muß die Sequenz in drei Regeln aufgespalten werden. Diese Aufspaltung macht wiederum die explizite Verwaltung eines Ausführungskontextes notwendig, was hier mit dem Hilfsprädikat `state` durchgeführt wird.

Den Auftrag, einen Zielort `X` anzufahren, wird von VM1 nach VM2 durch Fakten der Form `goto(X)` kommuniziert. Die Ausführung muß zwei Fälle unterscheiden: Der Roboter ist bereits am Zielort, oder er ist es nicht. Diese Fallunterscheidung spiegelt sich in der Verwendung zweier Regeln wieder.

Das Anfahren des Zielortes erfolgt zunächst blind, indem der Roboter nicht-deterministisch einen der benachbarten Räume anfährt, ohne Rücksicht darauf, ob er so dem Ziel näher kommt. Eine intelligentere Lösung werden wir bei der Darstellung der Sita-

```
location(waste X)
: location(bin Y)
<-      !location(robot X)
        pickup()
        !location(robot Y)
        drop()
.

!location(robot X)
: location(robot X)
<-      true()
.
```

```
!location(robot X)
: location(robot Y)
  X <> Y
  adjacent(Y Z)
  ¬ location(person Z)
<-      move(Y Z)
        !location(robot X)
.

location(person X)
: location(robot X)
  adjacent(X Z)
<-      move(X Z)
.
```

Abbildung 7.3: AgentSpeak(L)-Version der Wegeplanung

Version diskutieren.

VM3 schließlich wird durch die letzte Produktion realisiert, die relativ zu den übrigen Produktionen höher priorisiert wird. Da das Verhaltensmuster eine einfache Sensor-Aktor-Reaktion darstellt, kann es problemlos als Magsy-Regel repräsentiert werden.

Bereits dieses kleine Beispiel verdeutlicht die in der Einleitung geäußerte Kritik, nach der sich Produktionenprogramme als flache, unstrukturierte Regelmenge präsentieren. Die innere Struktur des entstandenen Programmes findet in der syntaktischen Erscheinungsform keine Entsprechung. Vielmehr werden drei unterschiedliche Strukturierungskonzepte, als da sind Verhaltensmuster, Sequenz und Fallunterscheidung, jeweils auf das gleiche Ausdrucksmittel projiziert: das Ansammeln von Regeln. Die Intentionenstruktur des Agenten (vergleiche Abschnitt 2.4) muß der Programmierer in Form geeigneter Kontextfakten (Prädikate `state` und `goto`) selbst verwalten.

7.3 Realisierung in AgentSpeak(L)

Die in Abbildung 7.3 gezeigte AgentSpeak(L)-Version des Beispiels besteht aus vier Plänen. Pläne werden gemäß folgendem Muster notiert:

TriggerEreignis `:` *Plankontext* `<-` *Planschritte* `.`

Ein AgentSpeak(L)-Plan besitzt Ähnlichkeiten zu einer Magsy-Produktion, mit zwei Unterschieden: Zum einen ist der Bedingungsteil aufgeteilt in eine atomare Triggerbedingung und einen Plankontext. Zum anderen können im Planrumpf neben den in Magsy möglichen Konstrukten auch Zustands- und Abfrageziele verwendet werden.

Zustandsziele, im Beispiel `!location(robot X)`, stellen dabei den Aufruf eines entsprechenden Plans als Unterprogramm dar. Im Gegensatz zu Magsy besitzt AgentSpeak(L) ein Konzept zur Ausführung der Pläne, in dem die Verwaltung des aktuellen Ausführungsstandes und der Aufrufbeziehungen zwischen Plänen eingeschlossen ist. Dadurch wird es im Beispiel möglich, die für VM1 notwendige Sequenz durch einen einzigen Plan darzustellen, statt wie in Magsy in drei Regeln.

Die angesprochene Aufteilung des Bedingungsteils in AgentSpeak(L) muß, im Vergleich zu Magsy, als Einschränkung der Reaktivität bewertet werden. Im Beispiel kann der erste Plan nur durch das Auftauchen des Abfalls getriggert werden. Falls nun aber zuerst die Position des Abfalls bekannt wird und danach die Position des Papierkorbes, kann der Plan nicht mehr darauf reagieren. Dagegen hat in Magsy, ebenso wie in Sita, die Reihenfolge, in der die konjunktiv verknüpften Einzelbedingungen wahr werden, keine Bedeutung.

Die Entscheidung, in AgentSpeak(L) nur atomare Triggerbedingungen zuzulassen, wurde aus Effizienzgründen getroffen. Denn damit entfällt die Notwendigkeit, komplexere Bedingungen reaktiv auszuwerten. In Magsy wird diese Auswertung durch den Rete-Algorithmus realisiert. Dieser kann insofern ineffizient sein, als er stets die Bedingungsteile aller Produktionen auswertet, ohne die Ausführungsstände und Prioritäten der Handlungsstränge zu berücksichtigen. Dieses Problem wird in Sita durch die Anwendung der magic-set Transformation (Abschnitt 4.3) sowie die Technik der vertikalen Priorisierung (Abschnitt 6.6.2) vermieden. Dadurch wird die Auswertung der Wissensbasis auf die Prädikate beschränkt, die zur Entscheidung über die Fortsetzung des Programmlaufes tatsächlich benötigt werden. Eine Einschränkung der Bedingungssprache aus Effizienzgründen ist damit nicht notwendig.

7.4 Realisierung in Sita

In Abbildung 7.4 ist eine erste Version der Sita-Lösung dargestellt. Diese Version entspricht in ihrer Funktionalität den bisher vorgestellten Lösungen, wir werden sie später noch erweitern.

Jedes der drei Verhaltensmuster wird durch eine Sita-Prozedur realisiert. Wie bereits in AgentSpeak(L) kann die in VM1 enthaltene Sequenz durch eine sequentielle Prozedur ausgedrückt werden. Darüberhinaus kann nun die Fallunterscheidung in VM2 auf eine überwachte Auswahl mit zwei Zweigen abgebildet werden, und somit innerhalb einer Prozedur abgewickelt werden.

Da in Sita Threads nicht spontan entstehen, müssen VM1 und VM3 bei Programmstart durch die `main`-Prozedur als nebenläufige Handlungen gestartet werden. Die erzeugten Threads werden durch die rekursiven Aufrufe von `gatherWaste` und `watchPerson` am laufen gehalten.

```
proc main()
   par priority 1
       gatherWaste()
    [] priority 2
       watchPerson()
   end
end

proc gatherWaste()
  when location(waste X)
       location(bin   Y)
    ⇒ goto(X)
       pickup()
       goto(Y)
       drop()
       gatherWaste()
   end
end
```

```
proc goto(in X)
  when location(robot X)
     ⇒ /* nothing to do */
    [] location(robot Y)
       X <> Y
       adjacent(Y Z)
       ¬location(person Z)
     ⇒ move(Y Z)
       goto(X)
   end
end

proc watchPerson()
  when location(robot  X)
       location(person X)
       adjacent(X Z)
     ⇒ move(X Z)
       watchPerson()
   end
end
```

Abbildung 7.4: Sita-Version der Wegeplanung (erste Version)

Das an AgentSpeak(L) kritisierte Problem atomarer Trigger besteht in Sita nicht.

Im Vergleich mit der Magsy-Version gewinnt die Sita-Version deutlich an Struktur und Übersichtlichkeit. Denn nun stehen für die verschiedenen Modellierungskonzepte jeweils entsprechende Sprachkonstrukte zur Verfügung: Verhaltensmuster werden auf Prozeduren abgebildet, Handlungssequenzen auf Tasksequenzen, Wartezustände auf überwachte Auswahlen und Fallunterscheidungen auf Zweige der überwachten Auswahl.

Gleichzeitig bleibt aber die Reaktivität erhalten. Der Programmierer kann beliebig viele Threads erzeugen, die in überwachten Auswahlen jeweils auf das Eintreffen beliebiger Bedingungen reagieren können. In gewisser Weise wird die Reaktivität noch erhöht, da die Sprache zur Situationsbeschreibung im Vergleich zu Magsy wesentlich ausdrucksstärker ist.

Bisher verwendet das Sita-Programm keine Klauseln zur Definition abgeleiteter Situationsprädikate. Wir werden jetzt ein Klauselprogramm ergänzen, das die bisher blinde Suche nach dem Bestimmungsort durch ein zielgerichtetes Anfahren ersetzt. In Abbildung 7.5 werden dazu zwei abgeleitete Prädikate eingeführt, die zusammen auf Grundlage der Adjazenz-Relation den kürzesten Weg zwischen zwei Orten angeben.

```
pred step(in  Start
          in  Target
          out Next
          out Len )
pred best(in  Start
          in  Target
          out Next
          out Len )

step(Start Target Next Len)
← adjacent(Start Next)
   Target = Next
   Len = 1
.

step(Start Target Next Len)
← adjacent(Start Next)
   best(Next Target _ Len1)
   Len = Len1 + 1
.

best(Start Target Next Len)
← step(Start Target
             Next Len)
   ¬ ( step(Start Target
                  _ Len1)
       Len1 < Len )
.

proc goto(in X)
  when location(robot X)
     ⇒ /* nothing to do */
     [] location(robot Y)
        X <> Y
        best(Y X Z _)
        ¬ location(person Z)
     ⇒ move(Y Z)
        goto(X)
   end
end
```

Abbildung 7.5: Zielgerichtete Anfahrt

Die Relation `step` verlängert die bisher gefundenen minimalen Wege um einen Schritt, die Relation `best` wählt unter alternativen Wegen zwischen zwei Orten den oder die kürzesten aus und stützt sich dabei verschränkt rekursiv auf `step` ab. Mit diesem kleinen Wissensbasisprogramm kann nun die `goto`-Prozedur zielgerichtet auf den Bestimmungsort zufahren.

Diese Definition kürzester Wege geschieht rein deklarativ. Das hat neben der eleganten Formulierung auch den Vorteil, daß sich die Berechnung dynamisch an Veränderungen der Adjazenz-Relation anpaßt.

Zur Verdeutlichung schränken wir die Allwissenheit des Roboters ein und nehmen an, daß das Wissen um benachbarte Räume nicht vorgegeben ist. Der Roboter muß somit seine Umgebung erst erkunden. Betritt er einen Raum, so nimmt er die dort vorhandenen Übergänge wahr ebenso wie eventuell vorhandenen Abfall oder den Papierkorb. Des weiteren lassen wir zu, daß bereits als offen erkannte Durchgänge später wieder geschlossen sein können. In dieser neuen Situation verändert sich die Adjazenz-Relation fortlaufend und nicht-monoton. Trotzdem kann die angegebene Definition der kürzesten Wege unverändert beibehalten werden, ihre Berechnung paßt sich inkrementell an eine veränderte Situation an.

Sollen bei der Wegeplanung auch die aktuell bekannten Aufenthaltsorte von Personen

```
pred busy()
pred seen(out X)

proc explore()
  when ¬ busy()
       location(robot X)
    ⇒ when adjacent(X Y)
            ¬ seen(Y)
         ⇒ move(X Y)
            insert seen(Y)
            explore()
         [] adjacent(X Y)
         ⇒ move(X Y)
            explore()
         end
   end
end
```

```
proc main()
   par priority 0
       explore()
     [] priority 1
       gatherWaste()
     [] priority 2
       watchPerson()
   end
end

proc gatherWaste()
  when location(waste X)
       location(bin   Y)
     ⇒ insert busy()
       goto(X)
       pickup()
       goto(Y)
       drop()
       delete busy()
       gatherWaste()
   end
end
```

Abbildung 7.6: Erkunden der Umgebung

berücksichtigt werden, so müssen lediglich die beiden Klauseln der `step`-Relation um eine entsprechende Bedingung erweitert werden, Räume mit Personen nicht mit einzuplanen. Wieder sorgt die datengetriebene Auswertung dafür, daß eine Änderung des Wissens um Personen zu einer dynamische Anpassung des geplanten Weges führt.

In Magsy oder AgentSpeak(L) wäre das gleiche Verhalten ungleich aufwendiger zu realisieren. Insbesondere die Berücksichtigung nicht-monotoner Änderungen der Adjazenz-Relation würde hier zu einem wesentlich komplexeren, imperativen Algorithmus führen.

Das Beispiel abschließend geben wir in Abbildung 7.6 eine Prozedur `explore` an, die den Roboter zu Erkundungsfahrten veranlaßt. Dazu führen wir auch zwei extensionale Prädikate ein: In `seen` werden die bereits besuchten Räume vermerkt, damit der Roboter noch unbesuchte Räume während der Erkundung bevorzugen kann. Das Prädikat `busy` dient der Synchronisation der beiden nebenläufigen Prozeduren `explore` und `gatherWaste`. Die Prozeduren `main` und `gatherWaste` werden entsprechend ergänzt.

KAPITEL 8

Schlußbemerkungen

Nach einer Zusammenfassung der wesentlichen Ergebnisse der Arbeit skizzieren wir kurz einige mögliche Erweiterungen der Sprache: zusätzliche prozedurale Konstrukte, Modularisierung und Objektorientierung sowie Updates abgeleiteter Prädikate.

8.1 Zusammenfassung

Situatives Agieren modelliert Agentenverhalten, indem die Situationen, in denen sich der Agent vorfinden kann, verknüpft werden mit Handlungen, die zu den jeweiligen Situationen geeignete Reaktionen darstellen. Agentensprachen, die diesem Modell folgen, verwenden zumeist eine Form von Regeln, um diese Verknüpfung darzustellen.

Die exemplarische Untersuchung einiger Programmiersprachen zeigte auf, daß das Regelkonstrukt als oberstes Strukturierungsmittel nicht unproblematisch ist. Unser Ansatz, situatives Agieren im Rahmen eines Programmiermodells zu formalisieren, war es daher, das Situative und das Agieren jeweils mit einem eigenen Berechnungsformalismus auszustatten. Als Gewinn ergeben sich zum einen reichhaltigere Strukturierungsmittel und zum anderen eine erhöhte Ausdrucksstärke, insbesondere auf Seiten der Situationserkennung.

Mit ihrer Form der Logik-Programmierung verbindet die Wissensbasis drei wesentliche Eigenschaften: Deklarativität, Ausdrucksstärke und Ereignisgetriebenheit. Damit findet die für die Formulierung von Agentenverhalten wichtige Situationserkennung bereits auf programmiersprachlicher Ebene weitreichende Unterstützung.

Aus dem Bereich der deduktiven Datenbanken konnten zwei wichtige Konzepte übernommen werden: die wohl-fundierte Semantik und die magic-set Transformation.

Im Zentrum der Sita-Maschine steht der IDC-Algorithmus, der die inkrementelle bottom-up Auswertung des Hornklauselprogrammes leistet. Dieses Verfahren ist der Schlüssel dazu, Logikprogramme reaktiv auszuwerten und somit Deklarativität mit Ereignisgetriebenheit zu verbinden.

Entscheidend für die Praxistauglichkeit ist eine akzeptable Performanz des Deduktionsverfahrens. Neben dem Einsatz der magic-set Transformation konnten wir in unseren Experimenten vor allem zwei kritische Punkte ausmachen, die die Effizienz wesentlich beeinflussen: die Technik zur Vermeidung von Zirkelschlüssen sowie die Steuerung des Reduktionsprozesses durch Scheduling-Strategien. Durch die Verwendung von Höheninformationen im Zwei-Phasen-Löschverfahren werden die beobachteten Instabilitäten des einfacheren Zählerverfahrens vermieden. Die beschriebene horizontale Priorisierung konnte in unseren Versuchen die Laufzeit um Größenordnungen verbessern. Die vertikale Priorisierung schließlich schränkt die Auswertung der Wissensbasis auf jene Aspekte ein, die im Sinne der Thread-Prioritäten in der jeweiligen Situation vordringlich sind. Dadurch werden zeitaufwendige Berechnungen unterbrechbar gemacht und somit die Reaktivität des Systems sichergestellt.

Zum Testen und Experimentieren wurde eine umfangreiche Implementierung von Sita angefertigt. Ein in Prolog realisierter Compiler übersetzt Hornklauselprogramme in Relationennetze, die als Code einer virtuellen Register-Maschine repräsentiert werden. Dieser Code wird von der Sita-Maschine interpretiert, die in C++ realisiert ist. Diese Implementierung diente als Grundlage zur Beurteilung der untersuchten Optimierungstechniken.

Das im vorangegangenen Kapitel dargestellte Anwendungsbeispiel demonstriert die im Vergleich zu Magsy und AgentSpeak(L) erzielten Verbesserungen bei der Modellierung situativen Verhaltens. Zum einen konnte das Programm wesentlich besser strukturiert werden. Zum anderen konnte die Suche der kürzesten Wege deklarativ formuliert werden, was im Vergleich mit einer imperativen Lösung nicht nur den Vorzug der einfacheren Programmierung hat. Auch paßt sich die Berechnung automatisch und inkrementell einer veränderten Situation an.

Über das angegebene Anwendungsbeispiel hinaus haben wir vor allem die Anwendbarkeit vorwärtsverkettender Logikprogrammierung untersucht. So war eine Formulierung des n-Damen Problems als rein deklaratives Logikprogramm ein wichtiges Beispiel für die durchgeführten Effizienzmessungen. Erfahrungen mit größeren Anwendungen, die auch die Reaktivität der Sprache ausnutzen, stehen bis jetzt noch aus.

8.2 Ausblick

Einige mögliche Erweiterungen wurden bereits in dieser Arbeit angesprochen. Wir fassen sie hier nochmals zusammen, bevor wir auf weitergehende zukünftige Entwicklungsmöglichkeiten eingehen.

In Abschnitt 5.4 wurden Erweiterungen der prozeduralen Sprache diskutiert. Dazu zählen nicht-wartende Wissensbasis-Anfragen, zusätzliche Kontrollstrukturen zur

Steuerung nebenläufiger Threads und klassische imperative Konstrukte wie zustandstragende Variablen. Eine andere Entwicklung in ähnliche Richtung wäre die Einbettung der Sita-Wissensbasis in eine vorhandene nebenläufige prozedurale Sprache wie etwa Oz [SHW95].

Neben den ausführlich untersuchten Optimierungstechniken wie den Scheduling-Strategien wurden in Abschnitt 6.7 zwei weitere Techniken vorgeschlagen, von denen anzunehmen ist, daß sie in vielen Fällen deutliche Effizienzsteigerungen zulassen dürften. Da ist zum einen die Join-Order-Optimierung, oder allgemeiner die Optimierung von Operator-Bäumen durch algebraische Umformungen. Entsprechende Techniken sind im Datenbank-Bereich, aber auch im Zusammenhang mit Produktionensystemen entwickelt worden und sollten auch innerhalb von Sita mit Erfolg anwendbar sein. Zum anderen wurde die Einführung von Aggregatsfunktionen vorgeschlagen. Hierdurch ließen sich manche Algorithmen in einer Version mit günstigerer Berechnungskomplexität formulieren. Zur Effizienzsteigerung erscheint die Realisierung von Aggregatsfunktionen daher sehr aussichtsreich.

Im folgenden skizzieren wir abschließend einige Ideen, die Architektur von Sita über die bisherigen Konzepte hinaus zu erweitern: zusätzliche Strukturierungsmittel und Updates intensionaler Prädikate.

8.2.1 Weitergehende Strukturierungsmittel

Einerseits bietet Sita im Vergleich zu reinen Regelsprachen wie OPS5 deutlich verbesserte Strukturierungsmöglichkeiten. Andererseits sind in der Entwicklungsgeschichte der Programmiersprachen weitergehende Strukturierungkonzepte eingeführt worden, die teilweise auch auf Sita übertragbar sind. Wir betrachten hier kurz die Modularisierung und die Objektorientierung.

Ein Modulkonzept, wie es in gängigen prozeduralen Sprachen wie etwa C/C++ vorhanden ist, kann verhältnismäßig einfach auf Sita übertragen werden. Ein Modul besteht aus einer öffentlichen Schnittstelle und einer privaten Implementierung. Die Module kommunizieren untereinander sowohl durch den Aufruf von Prozeduren als auch über gemeinsam genutzte Prädikate, die den globalen Variablen anderer Sprachen entsprechen. In der Schnittstelle eines Moduls sind damit jeweils der Export und der Import von Prozeduren und Prädikaten zu erklären.

Während Module meist relativ große Programmeinheiten darstellen, strebt die Objektorientierung eine feinere Einteilung an. Hier werden Daten und dazugehörige Operationen zu Objekten bzw. Objektklassen zusammengefaßt und so ein gemeinsames Strukturierungskonzept geschaffen. Eine Übertragung dieser Ideen auf Sita scheint recht schwierig zu sein, sind hier doch Daten und Prozeduren getrennt in Wissensbasis und Handlungsbasis repräsentiert. Eine bloße Kapselung von Prädikaten und Prozeduren zu Klassen geht dann aber über das angedeutete Modulkonzept nicht hinaus. Hier

scheint also die Aufspaltung eines Systems in Module — oder aber im größeren in Agenten — das naheliegendere Konzept zu sein.

Innerhalb der Wissensbasis ist jedoch durchaus ein objektorientiertes Schema zur Wissensrepräsentation denkbar, ähnlich wie in Frame-basierten Systemen [Min75]. Informationen wie die Existenz von Objekten, ihre Klassenzugehörigkeit sowie ihre Attributwerte könnten auf entsprechende Einzelfakten abgebildet werden. Klauseln können ergänzend zur semantischen Anreicherung der repräsentierten Informationen dienen, etwa um Spezialisierungsbeziehungen zwischen Klassen auszudrücken oder Default-Werte für Attribute zu vereinbaren. Ein entsprechendes Schema ist in [Mül96] angedeutet.

8.2.2 Updates abgeleiteter Prädikate

Die Sita-Wissensbasis unterscheidet zwischen assertierten Fakten und abgeleiteten Fakten (bzw. zwischen extensionalen und intensionalen Prädikaten). Der Deduktionsprozeß kennt nur eine Richtung: Veränderungen der Assertionen bewirken Veränderungen der Ableitungen.

Hier kann nun der Wunsch entstehen, bestimmte Veränderungen in den abgeleiteten Prädikaten zu erzielen. Dazu ist es notwendig, von der logischen Folgerung auf die Ursache zu schließen, also neben Deduktion auch Abduktion ([KKT93], [Men96]) zu betreiben. Hierfür sind vor allem zwei Anwendungen denkbar:

- Der Agent baut sich mit Hilfe von Deduktionsregeln ein abstraktes Weltbild auf. Werden ihm neue Informationen bekannt, muß er sein Wissen entsprechend revidieren ([Kow94], [AD94]). Nun kann es vorkommen, daß das neue Wissen dem alten Weltbild auf abstrakter Ebene widerspricht, so daß der Agent gefordert ist, konkrete Erklärungen für abstrakte Aussagen zu finden. Wenn aber die Abstrahierung durch Deduktionsregeln geleistet wird, verlangt diese Erklärungssuche nach dem umgekehrten Vorgang, also dem Schluß von den Ableitungen hin zu den Basisfakten.

- Dort, wo Deduktionsregeln von Ereignissen auf Effekte schließen, kann die Forderung nach einem bestimmten abgeleiteten Faktum das Ziel repräsentieren, den entsprechenden Effekt zu erreichen ([Sha89]). Gesucht sind dann die Basisfakten, die als auszuführende Aktionen den gewünschten Effekt nach sich ziehen würden. Aufbauend auf diesem Prinzip lassen sich flexible Planungsverfahren ableiten ([Esh88], [JFB96]).

Es stellt sich die Frage, ob es möglich und sinnvoll ist, diese Art des Schlußfolgerns auf programmiersprachlicher Ebene im Rahmen einer Architektur wie Sita zu unterstützen. Ansätze dazu können die Arbeiten zu abduktiver Logikprogrammierung [KKT93]

sein, die zumeist auf einer Erweiterung der SLDNF-Resolution beruhen ([DS98]). Aber auch die Verfahren, die für Truth-Maintenance-Systeme [Doy79], [dK85] entwickelt wurden, könnten hierfür Ansätze liefern.

Beide genannten Anwendungen müssen mit dem Problem umgehen, daß abduktive Schlüsse nicht eindeutig sind. Im allgemeinen lassen sich viele mögliche Veränderungen der Basisfakten finden, um ein gewünschtes abgeleitetes Faktum entstehen oder ein ungewünschtes verschwinden zu lassen. Dabei werden meist nur einige der möglichen Lösungen den Anforderungen der Problemstellung gerecht. Hier ist ein Formalismus notwendig, der zur Bewertung der unterschiedlichen Lösungen geeignet ist und den Abduktionsprozeß entsprechend steuern kann.

Es ist bisher recht unklar, ob sich diese Art des Schlußfolgerns genügend effizient realisieren und in die Architektur von Sita integrieren läßt. Das Ergebnis aber wäre ein ausdrucksstarker, deklarativ zu programmierender Suchmechanismus, der das berechnungstechnische Instrumentarium von Sita um einen weiteren Aspekt bereichert: zum situativen Agieren kommt das situative Planen.

ANHANG A

Ergebnisse der Performanz-Messungen

Die Messungen wurden auf einem Intel-basierten PC mit 600 MHz und 256 MB Hauptspeicher durchgeführt, der unter Linux betrieben wird. Es wird eine auf Geschwindigkeit optimierte Sita-Implementierung eingesetzt. Als C++ Übersetzer kommt der GNU-Compiler zum Einsatz.

Alle Zeiten sind in $1/100$ Sekunden angegeben.

A.1 Löschverfahren

Es werden die beiden Varianten des Zwei-Phasen-Verfahren zum Vermeiden von Zirkelschlüssen gegenübergestellt, das Zählerverfahren und das Höhenverfahren. Beschreibung und Auswertung des Experiments finden sich in 6.5.4.

Rekursionsart		lin.	lin.	lin.	lin.	n.-l.	n.-l.	n.-l.	n.-l.
Bindungsmuster		(i o)	(i o)	(o o)	(o o)	(i o)	(i o)	(o o)	(o o)
Tupelreihenfolge		1	2	1	2	1	2	1	2
Problemgröße		256	256	256	256	128	128	64	64
Aufbau	Zähler	24	26	28	203	69	63	109	79
	Höhen	28	30	34	224	129	75	146	93
Abbau	Zähler	75	30	7557	51872	114	67	415	4180
	Höhen	32	30	58	252	109	79	119	97
Gesamt	Zähler	99	56	7585	52075	183	130	524	4259
	Höhen	60	60	92	476	238	154	256	190

A.2 Scheduling-Strategie

Dieses zweite Experiment mißt den Einfluß verschiedener Scheduling-Strategien (Keller, Warteschlange, horizontale Priorisierung) auf die Laufzeit, siehe Abschnitt 6.6.3.

Anwendung				Scheduling-Strategie		
				Keller	Warteschlange	horizont. Prior.
trans. Hülle	lin. (i o)	n = 512	Aufbau	121	101	105
			Abbau	148	114	122
	lin. (o o)	n = 256	Aufbau	31	30	32
			Abbau	58	56	58
	n.-l. (i o)	n = 128	Aufbau	311	112	113
			Abbau	100	114	116
	n.-l. (o o)	n = 64	Aufbau	482	165	162
			Abbau	124	132	131
n-Damen Problem		n = 7		40992	1815	115
		n = 8		>100000	22670	1449
SCC-Berechnung				>100000	2800	1167

LITERATURVERZEICHNIS

[AB94] APT, KRZYSZTOF R. und ROLAND N. BOL: *Logic programming and Negation: A Survey*. The Journal of Logic Programming, 19 & 20:9–72, Mai 1994.

[Abi88] ABITEBOUL, SERGE: *Updates, A New Frontier*. In: GYSSENS, M., J. PAREDAENS und D. VAN GUCHT (Herausgeber): *ICDT'88, 2nd International Conference on Database Theory*, Band 326 der Reihe *Lecture Notes in Computer Science*, Seiten 1–18, Berlin, 1988. Springer.

[ABW88] APT, K., H. A. BLAIR und A. WALKER: *Towards a theory of declarative knowledge*. In: MINKER, J. (Herausgeber): *Foundations of deductive databases and logic programming*, Seiten 89–142, Los Altos, CA, USA, 1988. Morgan Kaufmann.

[AD94] ARAVINDAN, C. und P. M. DUNG: *Belief Dynamics, Abduction, and Databases*. In: MACNISH, CRAIG, DAVID PEARCE und LUÍS MONIZ PEREIRA (Herausgeber): *Proceedings of the European Workshop on Logics in Artificial Intelligence*, Band 838 der Reihe *LNAI*, Seiten 66–85, Berlin, September 1994. Springer.

[BFKM85] BROWNSTON, LEE, R. FARRELL, ELAINE KANT und N. MARTIN: *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming*. Addison-Wesley, Reading, MA, USA, 1985.

[BIP88] BRATMAN, MICHAEL, DAVID ISRAEL und MARTHA POLLACK: *Plans and resource bounded practical reasoning*. Computational Intelligence, 4:349–355, 1988.

[BL92] BRASS, STEFAN und UDO W. LIPECK: *Generalized Bottom-Up Query Evaluation*. In: PIROTTE, ALAIN, CLAUDE DELOBEL und GEORG GOTTLOB (Herausgeber): *Advances in Database Technology — EDBT'92, 3rd Int. Conf.*, Nummer 580 in *LNCS*, Seiten 88–103, Berlin, 1992. Springer.

[BMS96] BRY, FRANÇOIS, RAINER MANTHEY und HERIBERT SCHÜTZ: *Deduktive Datenbanken*. KI — Künstliche Intelligenz, 10(3):17–23, 1996.

[BMSU86] BANCILHON, FRANÇOIS, DAVID MAIER, YEHOSHUA SAGIV und JEFFREY D. ULLMAN: *Magic sets and other strange ways to implement logic programs*. In: ACM (Herausgeber): *PODS '86. Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24–26, 1986, Cambridge, MA*, Seiten 1–15, New York, USA, 1986. ACM Press.

[BNR$^+$87] BEERI, C., S. NAQVI, R. RAMAKRISHNAN, O. SHMUELI und S. TSUR: *Sets and negation in a logic data base language (LDL1)*. In: ACM (Herausgeber): *PODS '87. Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 23–25, 1987, San Diego, California*, Seiten 21–37, New York, USA, 1987. ACM Press.

[Boc89] BOCIONEK, SIEGFRIED: *Modularisierung als Grundkonzept zur Entwicklung systemunterstützer Programmierumgebungen für parallele Regelprogramme*. Dissertation, TU München, Institut für Informatik, Mai 1989.

[BR87a] BALBIN, ISAAC und KOTAGIRI RAMAMOHANARAO: *A Generalization of the Differential Approach to Recursive Query Evaluation*. Journal of Logic Programming, 4(3):259–262, September 1987.

[BR87b] BEERI, C. und R. RAMAKRISHNAN: *On the power of magic*. In: ACM (Herausgeber): *PODS '87. Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 23–25, 1987, San Diego, California*, Seiten 269–284, New York, USA, März 1987. ACM Press.

[Bro86] BROOKS, RODNEY: *A Layered Intelligent Control System for a Mobile Robot*. IEEE Journal of Robotics and Automation, RA-2:14–23, April 1986.

[BS84] BUCHANAN, BRUCE und EDWARD H. SHORTLIFFE: *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison Wesley, Reading, MA, USA, 1984.

[BSF95] BERMAN, K. A., J. S. SCHLIPF und J. V. FRANCO: *Computing the well-founded Semantics faster*. In: MAREK, V. W., A. NERODE und M. TRUSZCZYŃSKI (Herausgeber): *Proceedings of the 3rd International Conference on Logic Programming and Nonmonotonic Reasoning*, Band 928 der Reihe *LNAI*, Seiten 113–126, Berlin, Juni 1995. Springer.

[CGT90] CERI, S., G. GOTTLOB und L. TANCA: *Logic Programming and Databases*. Springer, Berlin, 1990.

[CKPR73] COLMERAUER, A., H. KANOUI, P. PASERO und P. ROUSSEL: *Un systeme de communication homme-machine en Francais*. Rapport, Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille II, Marseille, France, November 1973.

[Cul89] CULBERT, CHRIS: *CLIPS Reference Manual*. A. I. Section, L. B. J. Space Center, Juli 1989.

[CW93] CHEN, WEIDONG und DAVID S. WARREN: *Query Evaluation Under the Well-Founded Semantics*. In: ACM (Herausgeber): *PODS '93. Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems: May 25–28, 1993, Washington, DC*, Band 12 der Reihe *Proceedings of the ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems*, Seiten 168–179, New York, USA, 1993. ACM Press.

[d'I98] D'INVERNO, MARK P.: *Agents, Agency and Autonomy: A Formal Computational Model*. PhD thesis, Department of Computer Science, University College London, London, UK, 1998.

[Dix96] DIX, JÜRGEN: *Semantics of Logic Programs: Their Intuitions and Formal Properties, An Overview*. In: FUHRMANN, ANDRÉ und HANS ROTT (Herausgeber): *Logic, Action, and Information: Essays on Logic in Philosophy and Artificial Intelligence*, Seiten 241–327. Walter de Gruyter, Berlin, 1996.

[dK85] KLEER, JOHAN DE: *An Assumption-Based TMS*. Artificial Intelligence, 26(1):127–162, 1985.

[dKLW98] D'INVERNO, MARK, DAVID KINNY, MICHAEL LUCK und MICHAEL WOOLDRIDGE: *A Formal Specification of dMARS*. In: SINGH, MUNINDAR P., ANAND RAO und MICHAEL J. WOOLDRIDGE (Herausgeber): *Proceedings of the 4th International Workshop on Agent Theories, Architectures, and Languages (ATAL-97)*, Band 1365 der Reihe *LNAI*, Seiten 155–176, Berlin, Juli 24–26 1998. Springer.

[dL98] D'INVERNO, M. und M. LUCK: *A Formal Specification of AgentSpeak(L)*. Journal of Logic and Computation, 8(3), 1998.

[Doy79] DOYLE, JON: *A Truth Maintenance System*. Artificial Intelligence, 12(3):231–272, 1979.

[DS98] DENECKER, MARC und DANNY DE SCHREYE: *SLDNFA: An Abductive Procedure for Abductive Logic Programs*. Journal of Logic Programming, 34(2):111–167, Februar 1998.

[Esh88] ESHGHI, KAVE: *Abductive Planning with Event Calculus*. In: KOWALSKI, ROBERT A. und KENNETH A. BOWEN (Herausgeber): *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 1*, Seiten 562–579, Cambridge, MA, USA, 1988. The MIT Press.

[FFO93] FINGER, MARCELO, MICHAEL FISHER und RICHARD OWENS: *METATEM at Work: Modelling Reactive Systems Using Executable Temporal Logic*. In: *Proceedings of Sixth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA-AIE)*, Edinburgh, U.K., Juni 1993. Gordon and Breach.

[Fis92] FISHER, MICHAEL: *A Normal Form for First-Order Temporal Formulae*. In: KAPUR, DEEPAK (Herausgeber): *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, Band 607 der Reihe *LNAI*, Seiten 370–384, Berlin, Juni 1992. Springer.

[Fis93a] FISCHER, KLAUS: *The Rule-based Multi-Agent System MAGSY*. In: *Proceedings of the CKBS'92 Workshop*. DAKE Centre, University of Keele, UK, 1993.

[Fis93b] FISCHER, KLAUS: *Verteiltes und kooperatives Planen in einer flexiblen Fertigungsumgebung*, Band 26 der Reihe *DISKI, Dissertationen zur Künstlichen Intelligenz*. Infix, St. Augustin, Deutschland, 1993.

[Fis93c] FISHER, MICHAEL: *Concurrent METATEM — A Language for Modelling Reactive Systems*. In: BODE, ARNDT, MIKE REEVE und GOTTFRIED WOLF (Herausgeber): *Proceedings of PARLE '93 – Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science, Seiten 185–196, Berlin, Juni 14–17, 1993. Springer.

[Fis94] FISHER, MICHAEL: *A Survey of Concurrent METATEM: The Language and its Applications*. In: GABBAY, DOV M. und HANS JÜRGEN OHLBACH (Herausgeber): *Proceedings of the 1st International Conference on Temporal Logic*, Band 827 der Reihe *LNAI*, Seiten 480–505, Berlin, Juli 1994. Springer.

[Fis95] FISHER, MICHAEL: *Representing and Executing Agent-Based Systems*. In: WOOLDRIDGE, MICHAEL J. und NICHOLAS R. JENNINGS (Herausgeber): *Proceedings of the ECAI-94 Workshop on Agent Theories, archi-*

tectures and languages: Intelligent Agents I, Band 890 der Reihe *LNAI*, Seiten 307–323, Berlin, August 1995. Springer.

[For81] FORGY, CHARLES L.: *OPS5 Reference Manual*. Technischer BerichtCMU-CS-81-135, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, USA, 1981.

[For82] FORGY, CHARLES L.: *Rete: A fast Algorithm for the Many Patterns/Many Objects Pattern Match Problem*. Artificial Intelligence, 19(1):17–37, September 1982.

[FW94] FISHER, MICHAEL und MICHAEL WOOLDRIDGE: *Specifying and Executing Protocols for Cooperative Action*. In: DEEN, S. M. (Herausgeber): *Proceedings of the Second International Working Conference on Cooperating Knowledge-Based Systems (CKBS-94)*, Berlin, 1994. Springer.

[Gab89] GABBAY, DOV M.: *The Declarative Past and Imperative Future: Executable Temporal Logic for Interactive Systems*. In: BANIEQBAL, B., H. BARRINGER und A. PNUELI (Herausgeber): *Proceedings of the Conference on Temporal Logic in Specification*, Band 398 der Reihe *LNCS*, Seiten 409–448, Berlin, April 1989. Springer.

[GC92] GELERNTER, DAVID und NICHOLAS CARRIERO: *Coordination languages and their significance*. Communications of the ACM, 35(2):97–107, Februar 1992.

[GJ98] GERBER, CHRISTIAN und CHRISTOPH G. JUNG: *Resource management for boundedly optimal agent societies*. In: *Proceedings of the ECAI98 Workshop on Monitoring and Control of Real-Time Intelligent Systems*, Seiten 23–28, Brighton, U.K., 1998.

[GL87] GEORGEFF, M. P. und A. L. LANSKY: *Reactive Reasoning and Planning*. In: *The Proceedings of AAAI-87*, Seiten 677–682, Seattle, WA, USA, 1987.

[GL88] GELFOND, MICHAEL und VLADIMIR LIFSCHITZ: *The Stable Model Semantics for Logic Programming*. In: KOWALSKI, ROBERT A. und KENNETH BOWEN (Herausgeber): *Proceedings of the Fifth International Conference on Logic Programming*, Seiten 1070–1080, Cambridge, MA, USA, 1988. The MIT Press.

[GM95] GUPTA, ASHISH und INDERPAL SINGH MUMICK: *Maintenance of Materialized Views: Problems, Techniques and Applications*. IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing, 18(2):3–18, 1995.

[GMS93] GUPTA, ASHISH, INDERPAL SINGH MUMICK und V. S. SUBRAHMANIAN: *Maintaining Views Incrementally*. In: BUNEMAN, PETER und SUSHIL JAJODIA (Herausgeber): *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Seiten 157–166, Washington, D.C., USA, 26–28 Mai 1993.

[GN87] GENESERETH, MICHAEL R. und NILS J. NILSSON: *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, Los Altos, USA, 1987.

[Han93] HANSON, ERIC N.: *Gator: A Discrimination Network Structure for Active Database Rule Condition Matching*. Technischer BerichtUF-CIS-TR-93-009, CIS Department, University of Florida, Gainesville, Florida, USA, 1993.

[HD92] HARRISON, J. und S. DIETRICH: *Maintenance of Materialized Views in Deductive Databases: An Update Propagation Approach.* In: *Workshop on Deductive Databases, Washington DC*, November 1992.

[HdvM98] HINDRIKS, KOEN V., FRANK S. DE BOER, WIEBE VAN DER HOEK und JOHN-JULES CH. MEYER: *Formal Semantics for an Abstract Agent Programming Language*. In: SINGH, MUNINDAR P., ANAND RAO und MICHAEL J. WOOLDRIDGE (Herausgeber): *Proceedings of the 4th International Workshop on Agent Theories, Architectures, and Languages (ATAL-97*, Band 1365 der Reihe *LNAI*, Seiten 215–230, Berlin, Juli24–26 1998. Springer.

[Ish94] ISHIDA, TORU: *Parallel, Distributed and Multi–Agent Production Systems*, Band 878 der Reihe *LNAI*. Springer, Berlin, 1994.

[JF98] JUNG, CHRISTOPH G. und KLAUS FISCHER: *A Layered Agent Calculus with Concurrent, Continuous Processes*. In: SINGH, MUNINDAR P., ANAND RAO und MICHAEL J. WOOLDRIDGE (Herausgeber): *Proceedings of the 4th International Workshop on Agent Theories, Architectures, and Languages (ATAL-97*, Band 1365 der Reihe *LNAI*, Seiten 245–258, Berlin, Juli 24–26 1998. Springer.

[JFB96] JUNG, CHRISTOPH G., KLAUS FISCHER und ALASTAIR BURT: *Multi-Agent Planning Using an Abductive Event Calculus*. Research Report RR-96-04, Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Deutschland, 1996.

[Kem92] KEMP, DAVID B.: *On the Foundations of Query Evaluation in Deductive Databases*. PhD thesis, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1992.

[KKT93] KAKAS, ANTONIS C., ROBERT A. KOWALSKI und FRANCESCA TONI: *Abductive Logic Programming*. Journal of Logic and Computation, 6?(2?):719–770, 1993.

[Kow94] KOWALSKI, ROBERT A.: *Logic without Model Theory*. In: GABBAY, D. M. (Herausgeber): *What is a Logical System?*, Seiten 35–71. Oxford University Press, 1994.

[KS86] KOWALSKI, ROBERT A. und MAREK J. SERGOT: *A Logic-Based Calculus of Events*. New Generation Computing, 4:67–95, 1986.

[KS91] KEMP, DAVID B. und PETER J. STUCKEY: *Semantics of Logic Programs with Aggregates*. In: SARASWAT, VIJAY; UEDA, KAZUNORI (Herausgeber): *Proceedings of the 1991 International Symposium on Logic Programming (ISLP'91)*, Seiten 387–404, San Diego, CA, USA, Oktober 1991. MIT Press.

[KSS95] KEMP, DAVID B., DIVESH SRIVASTAVA und PETER J. STUCKEY: *Bottom-up evaluation and query optimization of well-founded models*. Theoretical Computer Science, 146(1–2):145–184, Juli 1995.

[LLL+96] LESPÉRANCE, YVES, HECTOR J. LEVESQUE, FANGZHEN LIN, DANIEL MARCU, RAYMOND REITER und RICHARD B. SCHERL: *Foundations of a Logical Approach to Agent Programming*. In: WOOLDRIDGE, MICHAEL, JÖRG P. MÜLLER und MILIND TAMBE (Herausgeber): *Proceedings on the IJCAI Workshop on Intelligent Agents II : Agent Theories, Architectures, and Languages*, Band 1037 der Reihe *LNAI*, Seiten 331–346, Berlin, 1996. Springer.

[Llo87] LLOYD, J. W.: *Foundations of Logic Programming*. Springer, Berlin, 2 Auflage, 1987.

[LMR92] LOBO, J., J. MINKER und A. RAJASEKAR: *Foundations of Disjunctive Logic Programming*. MIT press, Cambridge, MA, USA, 1992.

[Men96] MENZIES, TIM: *Applications of Abduction: Knowledge-Level Modelling*. International Journal of Human-Computer Studies, 45(3):305–335, 1996.

[MH69] MCCARTHY, J. und P. HAYES: *Some Philosophical Problems from the Standpoint of Artificial Intelligence*. In: MELTZER, B. und D. MICHIE (Herausgeber): *Machine Intelligence 4*, Seiten 463–502. Edinburgh University Press, 1969. Also appears in N. Nilsson and B. Webber (editors), Readings in Artificial Intelligence, Morgan-Kaufmann.

[Min75] MINSKY, MARVIN: *A framework for representing knowledge*. The Psychology of Computer Vision, ed. Patrick Henry Winston, McGraw Hill, Seiten 211–277, 1975.

[Mir87] MIRANKER, DANIEL P.: *TREAT: A Better Match Algorithm for AI Production Systems*. In: FORBUS, KENNETH; SHROBE, HOWARD (Herausgeber): *Proceedings of the 6th National Conference on Artificial Intelligence*, Seiten 42–47, Seattle, WA, USA, Juli 1987. Morgan Kaufmann.

[MP93] MÜLLER, JÖRG P. und MARKUS PISCHEL: *The Agent Architecture InteRRaP: Concept and Application*. Research Report RR-93-26, Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Deutschland, 1993.

[Mül96] MÜLLER, JÖRG P.: *An Architecture for Dynamically Interacting Agents*. Dissertation, Universität des Saarlandes, Saarbrücken, 1996.

[Naq86] NAQVI, S. A.: *A Logic for Negation in Database Systems*. In: MINKER, J. (Herausgeber): *Proc. Workshop on Foundations of Deductive Databases and Logic Programming*, Seiten 378–387, Washington, DC, USA, August 18-22, 1986.

[NT89] NAQVI, S. und S. TSUR: *A Logical Language for Data and Knowledge Bases*. Computer Science Press, New York, USA, 1989.

[Per97] PEREZ, PETER: *Optimierende Transformation von Hornklauselprogrammen und Übersetzung in Gleichungssyteme der relationalen Algebra*. Diplomarbeit, Technische Universität München, 1997.

[Prz88] PRZYMUSINSKI, T. C.: *On the Declarative Semantics of Deductive Databases and Logic Programming*. In: MINKER, J. (Herausgeber): *Foundations of Deductive Databases and Logic Programming*, Seiten 193–216. Morgan Kaufmann Pub., Washington, D.C., USA, 1988.

[PV96] PORTO, ANTÓNIO und VASCO T. VASCONCELOS: *Truth and Action Osmosis: the TAO computation model*. In: ANDREOLI, J.-M., C. HANKIN und D. LE MÉTAYER (Herausgeber): *Coordination Programming: Mechanisms, Models, and Semantics*, Seiten 65–97. Imperial College Press, 1996.

[Rao96] RAO, ANAND S.: *AgentSpeak(L) : BDI Agents Speak Out in a Logical Computable Language*. In: VELDE, WALTER VAN DE und JOHN W. PERRAM (Herausgeber): *Proceedings of the 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Band 1038 der Reihe *LNAI*, Seiten 42–55, Berlin, Januar 22–25 1996. Springer.

[Rei78] REITER, RAYMOND: *On Close World Data Bases*. In: GALLAIRE, H. und J. MINKER (Herausgeber): *Logic and Databases*, Seiten 55–76. Plenum Press, New York, USA, 1978.

[RG91] RAO, A. S. und M. P. GEORGEFF: *Modeling Agents Within a BDI-Architecture*. In: FIKES, R. und E. SANDEWALL (Herausgeber): *Proc. of the 2rd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, Seiten 473–484, Cambridge, MA, USA, April 1991. Morgan Kaufmann.

[Ros90] ROSS, KENNETH A.: *Modular stratification and magic sets for Datalog programs with negation*. In: ACM (Herausgeber): *PODS '90. Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems: April 2–4, 1990, Nashville, Tennessee*, Band 51(1) der Reihe *Journal of Computer and Systems Sciences*, Seiten 161–171, New York, USA, 1990. ACM Press.

[Ros91] ROSS, KENNETH A.: *The Semantics of Deductive Databases*. PhD thesis, Department of Computer Science, Stanford University, Stanford, CA, USA, 1991.

[RRS+93] RAMAKRISHNAN, RAGHU, WILLIAM G. ROTH, PRAVEEN SESHADRI, DIVESH SRIVASTAVA und S. SUDARSHAN: *The CORAL Deductive Database System*. In: BUNEMAN, PETER und SUSHIL JAJODIA (Herausgeber): *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Seiten 544–545, Washington, D.C., USA, 26–28 Mai 1993.

[RRSS94] RAMAKRISHNAN, RAGHU, KENNETH A. ROSS, DIVESH SRIVASTAVA und S. SUDARSHAN: *Efficient incremental evaluation of Queries with aggregation*. In: BRUYNOOGHE, MAURICE (Herausgeber): *Logic Programming - Proceedings of the 1994 International Symposium*, Seiten 204–218, Massachusetts Institute of Technology, USA, 1994. The MIT Press.

[RSS+97] RAO, P., K. SAGONAS, T. SWIFT, D. S. WARREN und J. FREIRE: *XSB: A system for efficiently computing WFS*. In: DIX, JÜRGEN, ULRICH FURBACH und ANIL NERODE (Herausgeber): *Proceedings of the 4th International Conference on Logic Programing and Nonmonotonic Reasoning*, Band 1265 der Reihe *LNAI*, Seiten 430–440, Berlin, 1997. Springer.

[Sar93] SARASWAT, VIJAY A.: *Concurrent Constraint Programming*. ACM Doctoral Dissertation Awards: Logic Programming. The MIT Press, Cambridge, MA, USA, 1993.

[SBJ87] SOLOWAY, ELLIOT, JUDY BACHANT und KEITH JENSEN: *Assessing the Maintainability of Xcon-in-Rime: Coping with the Problems of a Very Large Rule Base*. In: FORBUS, KENNETH und HOWARD SHROBE (Herausgeber): *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seiten 824–829, Cambridge, MA, USA, 1987. American Association for Artificial Intelligence, The MIT Press.

[SC94] SHOHAM, YOAV und STEVE B. COUSINS: *Logics of Mental Attitudes in AI*. In: LAKEMEYER, GERHARD und BERNHARD NEBEL (Herausgeber): *Foundations of knowledge representation and reasoning*, Band 810 der Reihe *LNAI*, Seiten 296–309. Springer, Berlin, 1994.

[SG88] SWAMI, ARUN N. und ANOOP GUPTA: *Optimization of Large Join Queries*. In: BORAL, HARAN und PER ÅKE LARSON (Herausgeber): *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, Seiten 8–17, Chicago, Illinois, USA, Juni 1988.

[Sha89] SHANAHAN, MURRAY: *Prediction is Deduction but Explanation is Abduction*. In: SRIDHARAN, N. S. (Herausgeber): *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, Seiten 1055–1060, Detroit, MI, USA, August 1989. Morgan Kaufmann.

[Sho93] SHOHAM, YOAV: *Agent-oriented programming*. Artificial Intelligence, 60:51–92, März 1993.

[SHW95] SMOLKA, GERT, MARTIN HENZ und JÖRG WÜRTZ: *Object-oriented concurrent constraint programming in Oz*. In: SARASWAT, V. und P. VAN HENTENRYCK (Herausgeber): *Principles and Practice of Constraint Programming.*, Seiten 27–48. MIT Press, 1995.

[Smi85] SMITH, D.: *Ordering Conjunctive Queries*. Artificial Intelligence, ; ACM CR 8512-1170, 26, 1985.

[SSRB93] SUDARSHAN, S., DIVESH SRIVASTAVA, RAGHU RAMAKRISHNAN und CATRIEL BEERI: *Extending the Well-Founded and Valid Semantics for Aggregation*. In: MILLER, DALE (Herausgeber): *Logic Programming - Proceedings of the 1993 International Symposium*, Seiten 590–608, Vancouver, Canada, 1993. The MIT Press.

[Sta90] STARKE, PETER H.: *Analyse von Petri-Netz-Modellen*. B.G. Teubner, Stuttgart, 1990.

[Ste90] STEELS, L.: *Cooperation Between Distributed Agents Through Self Organization*. In: DEMAZEAU, Y. und J.-P. MÜLLER (Herausgeber): *Decentralized AI — Proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-89)*, Seiten

175–196. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands, 1990.

[Suc87] SUCHMAN, LUCY A.: *Plans and Situated Actions: The Problem of Human-Computer Communication*. Cambridge University Press, New York, USA, 1987.

[Tho95] THOMAS, S. REBECCA: *The PLACA Agent Programming Language*. In: WOOLDRIDGE, MICHAEL J. und NICHOLAS R. JENNINGS (Herausgeber): *Proceedings of the ECAI-94 Workshop on Agent Theories, architectures and languages: Intelligent Agents I*, Band 890 der Reihe *LNAI*, Seiten 355–370, Berlin, August 1995. Springer.

[Tic87] TICHY, W.: *What can software engineers learn from AI?* IEEE Computer, 20(11):43–54, November 1987.

[Ull88] ULLMAN, JEFFREY D.: *Principles of Database and Knowledge-Base Systems. Volume I: Classical Database Systems*. Computer Science Press, New York, USA, 1988.

[Ull89] ULLMAN, JEFFREY D.: *Principles of Database and Knowledge-Base Systems. Volume II: The New Technologies*. Computer Science Press, New York, USA, 1989.

[Ung97] UNGER, ANKE: *Ein Verfahren zur inkrementellen bottom-up Auswertung allgemeiner Gleichungssyteme der relationalen Algebra*. Diplomarbeit, Technische Universität München, 1997.

[Van86] VAN GELDER, ALLEN: *Negation as Failure Using Tight Derivations for General Logic Programs*. In: *Proceedings of the International Symposium on Logic Programming*, Seiten 127–139. IEEE Computer Society, The Computer Society Press, September 1986.

[Van93] VAN GELDER, ALLEN: *The Alternating Fixpoint of Logic Programs with Negation*. Journal of Computer and System Sciences, 47(1):185–221, August 1993.

[VK76] VAN EMDEN, MAARTEN H. und ROBERT A. KOWALSKI: *The Semantics of Predicate Logic as a Programming Language*. Journal of the ACM, 23(4):733–742, Oktober 1976.

[vRS91] VAN GELDER, ALLEN, KENNETH ROSS und JOHN S. SCHLIPF: *The Well-Founded Semantics for General Logic Programs*. Journal of the ACM, 38(3):620–650, Juli 1991.

[Wid93] WIDOM, J.: *Deductive and Active Databases: Two Paradigms or Ends of a Spectrum?* In: PATON, N. W. und M. H. WILLIAMS (Herausgeber): *Proceedings of the 1st International Workshop on Rules in Database Systems*, Workshops in Computing, Seiten 306–315, Berlin, 1993. Springer.

[WJ95] WOOLDRIDGE, MICHAEL und NICOLAS R. JENNINGS: *Intelligent Agents: Theory and Practice*. Knowledge Engineering Review, 10(2):115–152, 1995.

[Woo99] WOOLDRIDGE, MICHAEL: *Intelligent Agents*. In: WEISS, GERHARD (Herausgeber): *Multiagent Systems*, Seiten 27–78. The MIT Press, Cambridge, MA, USA, 1999.

[ZBF97] ZUKOWSKI, ULRICH, STEFAN BRASS und BURKHARD FREITAG: *Improving the Alternating Fixpoint: The Transformation Approach*. In: DIX, JÜRGEN, ULRICH FURBACH und ANIL NERODE (Herausgeber): *Logic Programming and Nonmonotonic Reasoning*, LNAI 1265, Seiten 40–59, Berlin, 1997. Springer.

[ZF96] ZUKOWSKI, ULRICH und BURKHARD FREITAG: *The Differential Fixpoint of General Logic Programs*. In: BOULANGER, DIMITRI, ULRICH GESKE, FOSCA GIANOTTI und DIETMAR SEIPEL (Herausgeber): *Proc. of the Workshop DDLP'96 on Deductive Databases and Logic Programming. 4th Workshop in conjunction with JICSLP'96. Bonn, Deutschland, September 2 – 6, 1996*, Band 295 der Reihe *GMD-Studien*, Seiten 45–56, St. Augustin, Deutschland, 1996. GMD.