Engineering wissensbasierter Navigation und Steuerung autonom-mobiler Systeme

Alexandra Mian Syed

Dissertation

zur Erlangung des Doktorgrades
der Naturwissenschaften
an der Fakultät für Mathematik und Informatik
der Universität Passau

Passau
12. Juni 2001

An dieser Stelle möchte ich vor allem meinem Mann Ahmed für fachliche Diskussion, wie auch für seine seelische Unterstützung während der Arbeit an meiner Dissertation danken. Ein besonderer Dank gilt auch Herrn Prof. W. Hahn, der mir diese Arbeit ermöglicht hat und in schwierigen Situationen jederzeit hinter mir gestanden ist. Ich danke Thamannta und Florian für ihre tatkräftige Unterstützung als Praktikanten. Außerdem danke ich meinen Eltern für ihren Beistand und meinen Freunden, Wolfgang, Florian und Anja, die mir immer wieder Mut gemacht haben. Nicht zuletzt danke ich auch meinen Arbeitskollegen Silvia, Horst, Andreas, Herbert, Rainer und Markus für die gemeinsame Zeit, ihr offenes Ohr und ihr Verständnis.

Zusammenfassung

Die autonome Steuerung mobiler, technischer Systeme in nicht exakt vorherbestimmbaren Situationen erfordert Methoden der autonomen Entscheidungsfindung, um ein planvolles, zielgerichtetes Agieren und Reagieren unter Echtzeitbedingungen realisieren zu können.

Während mittels mathematischer Formeln Basisverhalten, wie beispielsweise in einer Geradeausbewegung, einer Drehung, bei einem Abbremsen, und in Gefahrenmomenten schnelle Reaktionen, berechnet werden, benötigt man auf der anderen Seite ein Regelsystem, um darüber hinaus "intelligentes", d.h. situationsangepaßtes Verhalten zu produzieren und gleichzeitig im Hinblick auf ein Missionsziel planvoll agieren zu können.

Derartige Regelsysteme müssen sich auf einer abstrakten Ebene formulieren lassen, sollen sie vom Menschen problemlos entwickelbar, leicht modifizierbar und gut verifizierbar bleiben. Eine aufgrund ihres Konzeptes geeignete Programmierwelt ist die Logikprogrammierung. Ziel der Logikprogrammierung ist es weniger, Arbeitsabläufe zu beschreiben, als vielmehr Wissen in Form von Fakten zu spezifizieren und mit Hilfe von Regeln Schlußfolgerungen aus diesen Fakten ziehen zu können.

Die klassische Logikprogrammierung ist jedoch aufgrund ihres Auswertungsmechanismus der SLD-Resolution (*linear resolution with selected function for definite clauses*) zu langsam für die Anwendung bei Echtzeitsystemen. Auch parallele Sprachformen, die ebenfalls mit SLD-Resolution arbeiten, erreichen beim Einsatz auf (von Neumann-) Mehrprozessorsystemen bislang nicht die notwendige Effizienz.

Das Anwendungsgebiet der deduktiven Datenbanken hat im Vergleich dazu durch Bottom-Up Techniken einen anderen Auswertungsansatz mit deutlich höherer Effizienz geliefert. Viele dort auftretenden Probleme können jedoch nur durch die Integration anforderungsgetriebener Abarbeitung gelöst werden.

Auf der anderen Seite stellen Datenflußrechnerarchitekturen aufgrund der automatisierten Ausbeutung feinkörniger Parallelität einen hervorragenden Ansatz der Parallelverarbeitung dar. Bei Datenflußrechnerarchitekturen handelt es sich um (Mehrprozessor-) Systeme, deren datengetriebener Abarbeitungsmechanismus sich grundlegend vom weit verbreiteten kontrollflußgesteuerten von Neumann-Prinzip unterscheidet.

Überlegungen zur Struktur von Steuerungssystemen werden ergeben, daß es mittels Ansätzen aus dem Gebiet der deduktiven Datenbanken möglich ist, ein für diese Aufgabenstellung neuartiges, ausschließlich datengetriebenes Auswertungsmodell zu generieren. Dabei vermeidet es Probleme, die bei Bottom-Up Verfahren auftreten können, wie z.B. das Auftreten unendlicher Wertemengen und die späte Einschränkung auf relevante Werte, ohne gleichzeitig die Stratifizierung von Programmen zu gefährden.

Ergebnis der Arbeit ist eine anwendungsbezogene, problemorientierte Entwicklungsumgebung, die einerseits die Entwicklung und Verifikation der Spezifikation mit existierenden Werkzeugen erlaubt und andererseits die effiziente, parallele Ausführung auf geeigneten Rechensystemen ermöglicht. Zusätzlich wird die Voraussetzung geschaffen, verschiedene weitere, für die Steuerung autonomer Systeme unverzichtbare Komponenten in das Abarbeitungsmodell zu integrieren.

Simulationsergebnisse werden belegen, daß das vorgestellte Berechnungsmodell bezüglich realer Anwendungsbeispiele bereits in einer Monoprozessorversion Echtzeitbedingungen genügt. Damit ist die Voraussetzung für die Ausführung zukünftiger, weitaus komplexerer Steuerungsprobleme, ggf. auf Mehrprozessorsystemen, in Echtzeit geschaffen.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Autonom-mobile Systeme 1

1.1.1 Struktur 3

1.1.2 Eigenschaften 5

1.1.3 Projekte 8

1.2 Ziel der Arbeit 9

1.3 Aufbau der Arbeit 9

2 Grundlagen **13**

2.1 Deduktive Datenbanken 13

2.1.1 Logikprogrammierung 13

2.1.2 Semantik von Logikprogrammen 19

2.1.3 Berechnungsmodelle 22

2.1.4 Verfahren 25

2.1.5 Sprachen 28

2.2 Datenflußrechner 30

2.2.1 Datenflußgraphen 30

2.2.2 Datenflußrechnerarchitekturen 31

2.2.3 Projekte 36

2.3 Problemstellung 38

2.4 State of the Art 39

2.4.1 Arbeiten aus Logikprogrammierung und deduktiven Datenbanken 39

2.4.2 Verwendete Methoden 45

2.4.3 Abgrenzung des Verfahrens 47

2.5 Detailziele der Arbeit 50

3 Das Datenflußmodell **51**

3.1 Die Spezifikationssprache 51

3.1.1 Anforderungen und Randbedingungen 51

3.1.2 Typen von Fakten 53

3.1.3 Normalform von $DATALOG_f^{\neg}$-Programmen 55

3.2 Der Strukturgraph 64

3.2.1 Definition des Strukturgraphen 64
3.2.2 Zyklen 71
3.3 Analyse der Datenabhängigkeiten 73
3.3.1 Parameter von Knoten 73
3.3.2 Vergabe eindeutiger Parameternamen 78
3.3.3 Vervielfältigung von Teilgraphen 85
3.3.4 Analyse der Parametermodi 89
3.4 Der Datenflußgraph 109
3.4.1 Konstruktionsprinizipien 110
3.4.2 Zyklenfreie Datenflußgraphen 112
3.4.3 Negation im Datenflußgraph 119
3.4.4 Datenflußgraphen mit Rekursionen 120
3.4.5 Optimierungen 139
3.4.6 Zusammenfassung 140
3.5 Ein Beispiel 142
3.5.1 Ein Expertensystem zur Steuerung eines Überholvorgangs 142
3.5.2 Transformationsschritte 144
3.5.3 Der Datenflußgraph 146
3.6 Funktionalität der Knoten 149
3.7 Korrektheit des Verfahrens 155
3.7.1 Semantische Äquivalenz von Basisprogrammen 155
3.7.2 Korrektheit der Transformation 157

4 Das Berechnungsmodell 173

4.1 Datengetriebene Ausführung 174
4.1.1 Transport von Daten 174
4.1.2 Knoten- und Kantenrepräsentation 193
4.1.3 Terminierung der Rekursion 195
4.1.4 Ausführungsvorschrift 197
4.2 Ereignisflußgetriebene Ausführung 203
4.2.1 Allgemeine Verwendung des Ereignisflußprinzips 204
4.2.2 Anwendbarkeit des Ereignisflußprinzips auf das Modell 204
4.2.3 Ereignisflußfunktionalität des Datenflußgraphen 205

4.2.4 Vergleich mit inkrementeller Bottom-Up Ausführung 206
4.3 Mehrschichten-Ausführung 208
4.3.1 Multiple Berechnungsfronten 208
4.3.2 Priorisierung von Alarmbehandlungen 211
4.4 Integration weiterer Komponenten 214
4.4.1 Mathematische Formeln 214
4.4.2 Neuronale Netze 215
4.4.3 Fuzzy Logic 216
4.4.4 Simulationssystem 218

5 Implementierung 221

5.1 Der Compiler 221
5.2 Der Simulator 222
5.2.1 Abschätzung der Leistung 223
5.2.2 Testergebnisse 226
5.2.3 Rangordnung 226
5.2.4 Ergebnisse 227
5.3 Ein Datenfluß-/Ereignisfluß-Maschinenmodell 231
5.4 Weitere Implementierungsplattformen 232

6 Ergebnis der Arbeit 237

A Formale Grundlagen 241

A.1 Prädikatenlogik 1. Stufe 241
A.2 Eigenschaften von Logikprogrammen 241
A.3 Substitution und Unifikation 243
A.4 Begriffe der Relationale Algebra 246
A.5 Graphen 247

B Das Steuerungssystem VaMoRs 249

B.1 Funktionsweise von VaMoRs 249
B.2 Systemstruktur von VaMoRs 251
B.3 Das Modul Behaviour Decision 253
B.3.1 Schnittstellen zu den anderen Modulen 253
B.3.2 Ablauf des Programms 257

B.4 C-Sourcecode des Moduls Behaviour Decision 264
B.5 Spezifikation des Regelteils in PROLOG 271
B.6 Datenflußgraph-Darstellung des Regelteils 277

1 Einleitung

Im Fokus der Arbeit stehen technische Systeme, die menschliche Fähigkeiten substituieren können. Aufgaben für diese Systeme, kurz Roboter genannt, reichen dabei von einem Einsatz als Fließbandmechaniker bis zum lebensrettenden "Feuerwehrmann". In Deutschland sind nach neuesten Schätzungen gegenwärtig insgesamt ca. 68.000 (Industrie-)Roboter im Einsatz [Buck 99].

Ungeachtet eines hohen Forschungsaufwands steckt die Robotertechnologie jedoch noch in den Anfängen. Roboter können zwar klar definierte Aufgaben vielfach schneller und präziser ausführen als der Mensch, für Arbeiten unter ständig wechselnden Bedingungen und in Kooperation mit Menschen oder anderen technischen Systemen werden die in sie gesetzten Erwartungen jedoch noch bei weitem nicht erfüllt.

Durch die Verwendung immer leistungsfähigerer Hardware, wie z.B. schnelleren Prozessoren und einer zunehmend ausgefeilteren Sensorik für Bild- und Tonverarbeitung, werden wesentliche Voraussetzungen für eine größere Flexibilität und eine höhere Autonomie zukünftiger Robotergenerationen geschaffen.

Eine zentrale Rolle spielen darüber hinaus im verstärkten Maß innovative Softwarekonzepte. Diese zielen darauf ab, daß Roboter autonom entscheiden, und das auch dann, wenn nur ungenaue oder unvollständige Informationen vorliegen. Sie sollen dabei sowohl

- schnell auf unerwartete Veränderungen in der Umgebung reagieren,
- Aktionen über längere Zeit strategisch planen,
- miteinander kommunizieren und ihr Verhalten aufeinander abstimmen, sowie
- aus der Erfahrung in der Umwelt lernen und sich an langfristige Umweltveränderungen anpassen.

1.1 Autonom-mobile Systeme

Autonomie bedeutet allgemein die Fähigkeit zur Selbstverwaltung. *Autonom-mobile Systeme* sind daher technische Systeme, die sich eigenständig, d.h. ohne Unterstützung durch den Menschen, zielgerichtet bewegen können. Der Begriff *Roboter* bezeichnet übergeordnet dazu autonome Systeme, die zielgerichtet allgemeine, vorgegebene Aufgaben durchführen können. Eine Fortbewegung muß jedoch nicht zwingend Bestandteil dieser Aufgabe sein [Aström 85].

Das im Zuge der technischen Entwicklung entstandene Spektrum der Robotertechnologie umfaßt im wesentlichen:

- *Industrieroboter für Fließbandproduktion*

  Fließbandarbeit ist durch einfache, sich monoton wiederholende Aktionen gekennzeichnet. Arbeitsabläufe können detailiert in Form von Algorithmen vorgegeben werden. Die Kontrolle bei Störfällen übernimmt der Mensch.

- *Frei bewegliche Industrieroboter*

  Frei bewegliche Industrieroboter werden z.B. in der Lagerverwaltung eingesetzt. Sowohl Umgebung, als auch Arbeitsabläufe sind im Prinzip klar definierbar. Befinden sich jedoch mehrere Roboter gleichzeitig im Einsatz, ist nicht mehr jede mögliche Situation vorhersehbar. Für die Verhaltensentscheidung sind daher komplexe Algorithmen, in die Techniken der künstlichen Intelligenz integriert sind, notwendig.

- *Assistenzsysteme*

  Assistenzsysteme haben zum Ziel, den Menschen bei komplexen (Steuerungs-)Aufgaben zu unterstützen. Ihr Funktionsumfang umfaßt Umgebungsbeobachtung, Sensorkontrolle, Steuerung in einfachen, standardisierten Situationen und Warnung in Gefahrenmomenten. Tritt ein unvorhergesehenes Ereignis ein, übernimmt der Mensch die Steuerung. Diese Systeme benötigen daher ein Expertensystem für die Planung, unterliegen bei der Entscheidungsfindung jedoch nicht notwendigerweise Echtzeitbedingungen.

- *Vollständig autonome Roboter*

  Vollständig autonome Roboter benötigen ein komplexes Expertensystem zur Verhaltensplanung, da sie in jeder Situation Entscheidungen selbständig treffen und diese in entsprechende Aktionen umsetzen müssen. Sie unterliegen dabei außerdem Echtzeitbedingungen. Als Richtlinie sollte die Reaktionszeit auf spontan eintretende Situationen mindestens der menschlichen Reaktionszeit entsprechen.

  Denkbar ist beispielsweise ein Roboter, der als "Feuerwehrmann" agiert und bei Brandkatastrophen auch an Stellen löschen oder retten kann, die für den Menschen aufgrund der Gefährlichkeit nicht mehr erreichbar sind. Ein weiteres Beispiel ist ein "Mondfahrzeug", wie z.B. der *MARS Pathfinder*, das sich zu Forschungszwecken ohne die Anwesenheit von Menschen auf der Oberfläche eines Mondes oder Planeten sicher fortbewegen und dabei verschiedenste Untersuchungen durchführen kann.

Die letzteren Beispiele machen deutlich, daß wichtige, zukunftsorientierte Einsatzmöglichkeiten für Roboter dort bestehen, wo eine vollständig autonome Entscheidungs- und Handlungsfähigkeit Grundvoraussetzung für den Einsatz ist. Da dazu im allgemeinen die Eigenschaft gehört, sich frei, d.h. in nicht eingeschränkte und vorgegebene Richtungen bewegen zu können, steht der Begriff *autonom-mobiles System* für die Menge der in dieser Arbeit betrachteten technischen Systeme.

Ein autonom-mobiles System besteht aus:

- der motorischen Funktionseinheit, sowie
- einem Navigations- und Steuerungssystem.

Nach DIN 19226 (Teil 1) versteht man unter *Steuerung* einen Vorgang, bei dem eine oder mehrere Größen als Eingangsgrößen andere Größen als Ausgangsgrößen aufgrund der dem System eigentümlichen Gesetzmäßigkeiten beeinflussen. Kennzeichen für das Steuern ist ein offener Wirkungsweg [Mann 96]. (Abgrenzend dazu ist *Regelung* ein Vorgang, bei dem eine zu regelnde Größe fortlaufend erfaßt, mit einer vorgegebenen Führungsgröße verglichen wird und sich selbst abhängig vom Ergebnis dieses Vergleichs im Sinne einer Angleichung an die Führungsgröße beeinflußt. Kennzeichen für das Regeln ist der geschlossene Wirkungsweg.)

Die selbständige *Navigation* eines autonom-mobilen Systems wird durch Integration von Techniken aus dem Forschungsgebiet der künstlichen Intelligenz in das Steuerungssystem erreicht. Man spricht dann von einem *autonomen Navigations- und Steuerungssystem* [Antsalkis 91, Albus 91].

Techniken aus dem Gebiet der künstlichen Intelligenz sind z.B:

- die Repräsentation von Informationen einschließlich darauf basierender Strategien zur Problemlösung (z.B. durch ein wissensbasiertes System),
- erweiterte Techniken zur Regelung und Steuerung (z.B. durch Fuzzy Sets),
- selbständiges Lernen (z.B. mittels neuronaler Netze).

Das Prinzip eines *wissensbasierten Systems* hat wesentliche Bezüge zum Konzept deduktiver Datenbanken. Eine *deduktive Datenbank* besteht sowohl aus einer Menge von Fakten zur expliziten Repräsentation von Informationen, als auch aus einer Menge von Regeln, die es ermöglichen, durch Schlußfolgerungen neue Informationen zu generieren und damit Anfragen an die Datenbank zu beantworten. Die Menge der Fakten wird als extensionaler Teil der Datenbank bezeichnet, die Menge der Regeln als intensionaler Teil [Altenkrüger 87, Kreutzer 91].

Ein wissensbasiertes System bezeichnet eine deduktive Datenbank, deren Daten und zugehörige Regeln dem Fachwissen von Experten entsprechen. Die dem System zugrundeliegende extensionale Datenbank wird dabei als *Wissensbasis* bezeichnet. Die Verwendung von Expertenwissen in einem wissensbasierten System führt außerdem zu dem Begriff des *Expertensystems*.

1.1.1 Struktur

Die grundlegenden Funktionen eines autonomen Navigations- und Steuerungssystems (im folgenden nur noch Steuerungssystem genannt) sind Situationswahrnehmung, Verhaltensplanung und Steuerungsausführung.

Für die Verhaltensberechnung sind mindestens folgende Elemente notwendig:

- ein Basissystem zur Realisierung der Grundfunktionalität und

- ein Expertensystem für die autonome Entscheidungsfindung.

Gegebenenfalls gliedern sich weitere Module an das Steuerungssystem an, wie z.B. ein Simulationssystem oder ein Kommunikationssystem.

Das konventionelle Konzept einer Architektur für ein Steuerungssystem ist eine Modularisierung nach den genannten Aufgaben Situationswahrnehmung, Verhaltensplanung und Steuerungsausführung. Für Echtzeitsysteme ist jedoch zusätzlich eine hierarchische Gliederung der Verhaltensplanung nach dem Abstraktionsgrad der Aufgaben notwendig [Olin 91]: auf unterster Ebene werden Grundfunktionen und kurzfristige Reaktionen berechnet, während auf den höher liegenden Ebenen mittel- und langfristige Verhaltensentscheidungen durch Regeln getroffen werden. Diese hierarchische Aufteilung ist Voraussetzung für die Realisierung von reflexartigem Agieren in Echtzeit. Nur so kann das Erkennen unvorhergesehener Situationen und die Berechnung entsprechender Reaktionen auf unterster Ebene ohne Berücksichtigung der auf höheren Ebenen stattfindenden Missionsplanung erfolgen. Ist kein reflexartiges Agieren notwendig, können dagegen obere Ebenen die Steuerung entsprechend der Missionsplanung vornehmen.

Abbildung 1 zeigt die Struktur eines Steuerungssystems. Die hierarchische Gliederung wird durch die Schichten des Steuerungssystems realisiert:

1. Das Modul *Mission* übernimmt die langfristige Planung des Verhaltens hinsichtlich eines Missionsziels und die entsprechende globale Kontrolle. Es erhält hierfür Informationen über die Mission und ihren möglichen Verlauf, sowie Daten über die Umgebung, in der die Mission auszuführen ist.

2. Das Modul *Aktueller Missionsteil* wertet die gegenwärtige Situation aus und trifft mittelfristige Verhaltensentscheidungen, die der Situation gerecht werden, ohne dabei wesentlich vom aktuellen Missionsteilziel abzuweichen. Notwendige Informationen sind z.B. der erreichte Missionsfortschritt, oder die Ergebnisse einer Analyse des Nahbereichs der Umgebung.

3. Das Modul *Elementares Verhalten* enthält die Berechnungen für kurzfristig im Vordergrund stehendes Basisverhalten. Dazu gehören für ein Straßenfahrzeug z.B. Spurhalten, Beschleunigen und Abbremsen, ebenso wie Berechnungen für das Verhalten in Notfallsituationen.

4. Auf unterster Ebene befinden sich häufig weitere Module, wie z.B. das Modul *Welt-Simulation*. Modellierungselemente für die Simulation der realen Welt sind u.a.:
   - Bewegungseigenschaften des autonomen Systems,
   - die Umgebung,
   - Objekte in der Umgebung.
   Durch die Simulation der realen Welt in vereinfachender Form können viele Steuerungsprobleme gelöst werden, während die Menge der Regeln und die Menge der notwendigen Regelauswertungen gering gehalten wird.

5. Aufgaben anderer Module auf unterster Ebene können sein:
   - eine Steuerung mittels Fuzzy Logic,
   - selbständiges Lernen, basierend auf neuronalen Netzen.

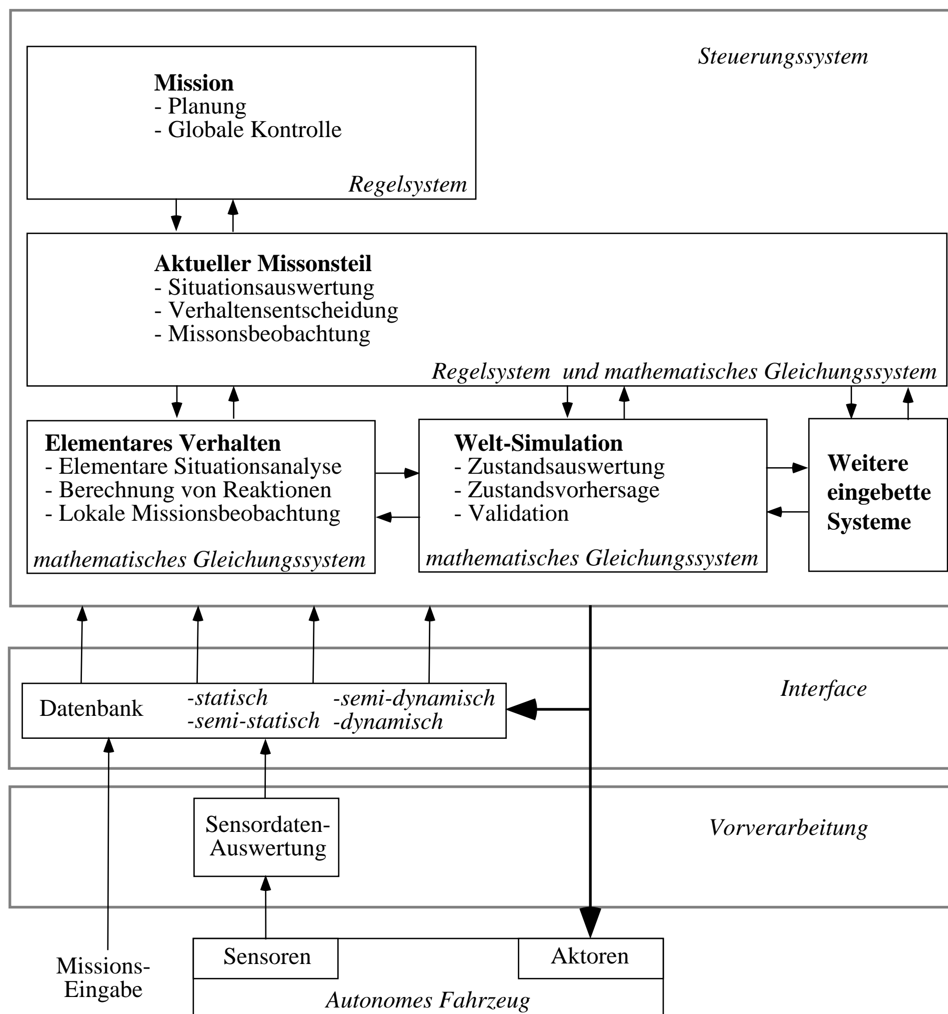


Abbildung 1: Ein hierarchisches Steuerungssystem

Die von den verschiedenen Sensoren des Fahrzeugs erfaßten Informationen werden von der *Sensordaten-Auswertung* vorverarbeitet. Da die z.B. mit einer Kamera aufgenommenen visuellen Umgebungsdaten einen wesentlichen Anteil der Sensordaten darstellen, ist Bilderkennung ein integraler Bestandteil der Sensordaten-Auswertung. Aufgrund der Komplexität der Bildanalyse kommen Forschungen in diesem Bereich eine große Bedeutung hinsichtlich der Steuerung autonom-mobiler Systeme zu. Leistungsfähige Algorithmen finden sich z.B. in [Dickmanns 97, Haas 99].

Die verschiedenen Abstraktionsstufen der Module werden in der Form ihrer Implementierung deutlich:

- Elementares Verhalten muß exakt bestimmbar sein: es wird durch konkrete Zahlenwerte für die Steuerungsausgaben an die technischen Steuerungselemente des autonomen Fahrzeugs ausgedrückt. Diese lassen sich wiederum durch z.B. Differentialgleichungen aus konkreten Zahlenwerten, die die aktuelle Situation und Umgebung beschreiben, berechnen.
- Die Modelle im Simulationssystem werden ebenfalls durch Gleichungssysteme beschrieben. Die Simulation wird durch Eingabe von konkreten Zahlenwerten, die der aktuellen Situation entsprechen, angestoßen. Der Vergleich mit der Situation des nächsten Zeitschrittes und, falls nötig, eine Korrektur der Modelle, erfolgt ebenfalls durch den Vergleich von Zahlenwerten.
- Das Modul *Aktueller Missionsteil* trifft dagegen Verhaltensentscheidungen, die sich in ihrer Komplexität nicht mehr nur durch mathematische Beziehungen ausdrücken lassen. Vielmehr muß auf dieser Ebene zusätzlich bereits ein Regelsystem zur Entscheidungsfindung zur Verfügung stehen.
- Auf der obersten Ebene trifft das Modul *Mission* Verhaltensentscheidungen ausschließlich mittels eines Regelsystems. Arbeitsabläufe sind aufgrund der Situationsvielfalt und -komplexität nicht mehr algorithmisch beschreibbar. Statt dessen tritt das Arbeiten mit symbolischen Daten unter Nutzung von Expertenwissen in den Vordergrund.

1.1.2 Eigenschaften

Wissensbasierte Systeme, die die Steuerung autonom-mobiler Systeme leisten sollen, unterscheiden sich in ihrer Struktur und Auswertungsform von allgemeinen Expertensystemen. Im folgenden werden diese Unterschiede diskutiert.

1.1.2.1 Änderungsrate der Umgebungsdaten

Bei einem Expertensystem ist der Inhalt der extensionalen Datenbank im allgemeinen statisch. Daten können nur durch das Expertensystem selbst hinzugefügt oder entfernt werden.

Bei einem wissensbasierten Steuerungssystem enthält die extensionale Datenbank Wissen über die Mission, über die Umgebung und vor allem über die aktuelle Situation, z.B. als Ergebnis der periodischen Auswertung von Sensordaten. Daher ändert sich der Inhalt der extensionalen Datenbank, respektive ein Teil des Inhalts, kontinuierlich mit dem Fortschreiten der Mission.

Die Daten lassen sich abgestuft nach der Änderungsfrequenz klassifizieren:

- *statisch*
  keine Veränderung während einer Mission des autonom-mobilen Systems.
- *semi-statisch*
  geringe Veränderung, d.h. im Stunden- oder Minutenbereich.
- *semi-dynamisch*
  häufige Veränderung, d.h. im Sekunden- oder Zehntelsekundenbereich.
- *dynamisch*
  kontinuierliche Veränderung, d.h. bei jeder Auswertung des wissensbasierten Systems.

Fazit: Die Daten der extensionalen Datenbank können sich fortlaufend durch externe, vom Expertensystem unabhängige Eingaben ändern.

1.1.2.2 Qualität der Umgebungsdaten

Ein wesentlicher Teil der Daten der extensionalen Datenbank ist das Ergebnis einer visuellen Umgebungserkennung, ergänzt durch eine Bildvorverarbeitung.

Die Daten, die z.B. durch eine Kamera aufgenommen werden, können jedoch ungenau sein, d.h. sich auch dann von Aufnahme zu Aufnahme unterscheiden, wenn das autonome System nahezu bewegungslos ist. Da die Änderungen jeweils minimal sind und sich um einen Mittelwert bewegen, kann man auch von einem "Flackern" der Daten sprechen.

Das "Flackern" der Daten läßt sich jedoch durch den Einsatz von Filtern in der Bildvorverarbeitung ausblenden, so daß davon auszugehen ist, daß Änderungen der Daten nur dann auftreten, wenn tatsächliche, deutliche Veränderungen in der Umgebung zugrunde liegen.

Fazit: Die Änderungsrate vieler Daten wird nicht von der Rate der Bildauswertung bestimmt, sondern von der tatsächlichen Änderungsrate der Umgebung.

1.1.2.3 Festgelegte Steuerungsausgaben

An ein Expertensystem können im Prinzip beliebige Abfragen gestellt werden, die das System mit Hilfe seiner Regeln beantwortet. Außerdem ist es möglich, in die Abfrage Daten zu integrieren, um die Menge der möglichen Lösungen einzuschränken.

Bei einem wissensbasierten Steuerungssystem handelt es sich jedoch immer um dieselbe Menge von Abfragen, da das Ergebnis der Abfragen als Steuerungsausgaben dienen soll:

- *Soll die Geschwindigkeit V geändert werden?*
- *Soll der Lenkwinkel W geändert werden?*

Das Ergebnis der Ausführung des Expertensystems variiert daher nicht durch unterschiedliche Abfragen, sondern nur durch die sich verändernden Daten der extensionalen Datenbank.

Die Parameter in den Abfragen stellen die Schnittstelle dar, über die Daten an das entsprechende technische Steuerungselement des Fahrzeugs übertragen werden. Daher handelt es sich grundsätzlich um Variablen, die bei Ausführungsbeginn nicht an Daten gebunden sind.

Fazit: Da sich die Menge der Einzelabfragen nicht ändert, können sie zu einem festen Bestandteil des Expertensystems werden.

1.1.2.4 Harte Echtzeitbedingungen

Ein wissensbasiertes Steuerungssystem muß harten Echtzeitbedingungen genügen. Dabei versteht man unter dem Begriff *harte Echtzeitbedingung*, daß ein Ausführungsergebnis vorliegen muß, bevor eine bestimmte, vorgegebene Zeitschranke überschritten ist und das Ergebnis irrelevant geworden ist. Z.B. darf bei Auftreten eines Hindernisses das Steuerungssystem eines Roboters den Befehl stehenzubleiben nicht erst dann an die technischen Steuerungselemente ausgeben, wenn das Hindernis bereits überfahren wurde.

Für ein wissensbasiertes Steuerungssystem ist die Zeitschranke beispielsweise durch eine Entsprechung der menschlichen Reaktionzeit vorgegeben. In konkreten Zeitmaßstäben ausgedrückt bedeutet dies eine Reaktionszeit und damit eine maximale Ausführungsdauer für das Expertensystem im Bereich von Zehntel-, besser Hundertstelsekunden [Kujawski 93].

Fazit: Das Berechnungsmodell für das Expertensystem muß hocheffizient sein.

1.1.2.5 Wiederholungsrate der Auswertung

An ein Expertensystem wird im allgemeinen nur bei Bedarf eine Abfrage gestellt. Bei einem wissensbasierten Steuerungssystem entspricht der Bedarf jedoch einer fortwährenden Situationsanalyse. Eine sich durch das Fortschreiten der Mission ständig verändernde Umgebungs- und Systemsituation, repräsentiert durch sich fortlaufend ändernde Werte im Datenspeicher, muß auch fortlaufend neu ausgewertet werden, damit das System jederzeit entsprechend agieren und korrekt reagieren kann.

Somit muß, trotz gleichbleibender Abfrage, das Programm in sich ständig wiederholender Folge ausgeführt werden, um kontinuierlich die der aktuellen Situation entsprechenden Steuerungsausgaben ermitteln zu können.

Da an die Steuerung autonomer Systeme außerdem harte Echtzeitbedingungen gestellt werden, muß dies darüberhinaus in hinreichend kurzen Zeitabständen geschehen. Eingehalten werden die Echtzeitbedingungen aus der Sicht des Steuerungssystems, wenn die Wiederholrate der maximalen Frequenz, mit der die Sensoren arbeiten und neue Daten in die Datenbank einspeisen, entspricht. Damit kann auf jede Situationsänderung tatsächlich unmittelbar reagiert werden.

Ein großer Anteil der Daten der extensionalen Datenbank bleibt jedoch über mehrere Ausführungen hinweg konstant, da sich insbesondere durch den Einsatz von Filtern nur ein kleiner Teil der von Sensoren erfaßten und durch die Sensordaten-Auswertung vorverarbeiteten Daten tatsächlich innerhalb kurzer, der Frequenz der Sensordaten-Erfassung entsprechenden Zeitabstände ändert.

Fazit: Teilergebnisse einer Ausführung sind häufig in der nächsten Ausführung noch gültig.

1.1.2.6 Konsequenz

Die Unterschiede von Expertensystemen für die autonome Steuerung zu allgemeinen Expertensystemen können im Sinne dieser Arbeit für ein neues Ausführungsmodell genutzt werden.

1.1.3 Projekte

Es gibt bereits eine Vielzahl von Entwicklungen auf dem Gebiet autonomer Fahrzeuge. Die folgende Liste soll, ohne Anspruch auf Vollständigkeit zu erheben, einen Überblick über die wichtigsten derzeit existierenden autonomen Fahrzeuge geben:

1. Unterstützende Fahrzeugführungssysteme

   - *CAMA (Cockpitassistenzsystem)*
     Prof. Dr. Onken, Fakultät für Luft- und Raumfahrttechnik, Universität der Bundeswehr, München,
     Entwicklung: seit 1987,
     Kategorie: Unterstützungssystem für Piloten [Werner 94, Werner 97].

   - *DAISY (Fahrzeugunterstützungssystem)*
     Prof. Dr. Onken, Fakultät für Luft- und Raumfahrttechnik, Universität der Bundeswehr, München,
     Entwicklung: 1984 - 1995,
     Kategorie: Unterstützungssystem für Autofahrer [Schreiner 95].

   - *Pilot's Associate*
     DARPA (Defence Advanced Research Project Agency) und US Air Force, USA,
     Entwicklung: seit 1986,
     Kategorie: Unterstützungssystem für Piloten [Banks 91].

2. Vollständig autonome Fahrzeugführungssysteme

   - *VaMoRs (Versuchsfahrzeug für autonome Mobilität und Rechnersehen)*
     Prof. Dr. Dickmanns, Fakultät für Luft- und Raumfahrttechnik, Universität der Bundeswehr, München,
     Entwicklung: seit 1982, Nachfolgeprojekte VaMoRs II, VaMP,
     Kategorie: Straßenfahrzeug [Zapp 88, Dickmanns 92].

   - *ALV (Autonomous Land Vehicle)*
     DARPA (Defence Advanced Research Project Agency), USA,
     Entwicklung: 1985-1987,
     Kategorie: Überlandfahrzeug [Amarel 91, Olin 91].

   - *Navlab*
     Carnegie Mellon Universität, USA,
     Entwicklung: seit 1986, Nachfolgeprojekte Navlab 2 - 11,
     Kategorie: Straßenfahrzeug, [Thorpe 91a, Thorpe 91b].

   - *AFV (Autonomous Flying Vehicle)*
     University of Southern California, USA,
     Entwicklung: seit 1991, Nachfolgeprojekt AVATAR (Autonomous Vehicle Area Tracking and Retrieval),
     Kategorie: Hubschrauber [Fagg 93].

Die in dieser Arbeit verwendeten Beispielfragmente basieren auf VaMoRs, eine Entwicklung der Universität der Bundeswehr in München unter Leitung von Prof. Dr. Dickmanns, Fakultät für Luft- und Raumfahrttechnik. Eine Beschreibung des Steuerungssystems für VaMoRs ist in Anhang B zu finden.

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist die Entwicklung

- *einer problemorientierten Spezifikationsumgebung*

  Als Spezifikationsumgebung soll eine bereits bekannte Programmierhochsprache benutzt werden können. Sie soll weder in ihrer Mächtigkeit eingeschränkt, noch um besondere Elemente erweitert werden mŭsseb. Damit können bereits existierende Werkzeuge zur Programmentwicklung und -verifikation genutzt werden.

- *einer problemorientierten Programmauswertung*

  Bei der Ausführung des Berechnungsmodells soll im Hinblick auf harte Echtzeitbedingungen Rechenleistung nur dort investiert werden, wo Berechnungen etwas zur Entscheidungsfindung beitragen können. Damit ist eine größtmögliche Effizienz bei der Programmauswertung gewährleistet.

- *eines Compilers und Simulationssystems*

  Die in dieser Arbeit vorgestellten Methoden und Algorithmen sollen in einem Compiler realisiert werden. Das aus der Spezifikation mit Hilfe des Compilers erstellte Graphmodell soll mit Hilfe eines Simulationssystems getestet und evaluiert werden. Für die reale Einsetzbarkeit sollen dabei bereits mit dem Simulator Echtzeitbedingungen eingehalten werden können.

1.3 Aufbau der Arbeit

Autonom-mobile Systeme benötigen für die Planung und Koordinierung ihrer Aktionen ein Expertensystem. Eine gängige Form der Implementierung ist ein wissensbasiertes System in Form einer deduktiven Datenbank.

Die Prinzipien deduktiver Datenbanken werden in Kapitel 2.1 vorgestellt. Logikprogrammierung ist eine gebräuchliche Methode eine deduktive Datenbank zu implementieren, da sich mit den Elementen der Sprache sowohl Fakten, als auch Regeln spezifizieren lassen, mit denen aus den Fakten Schlußfolgerungen gezogen werden. Die theoretischen Grundlagen der Logikprogrammierung werden in Abschnitt 2.1.1 behandelt. Die Schritt-für-Schritt Angabe von Berechnungsabfolgen, wie bei prozeduralen, objektorientierten oder funktionalen Sprachen, entfällt. Die theoretischen Grundlagen der Logik ermöglichen zudem eine weitestgehende Verifikation der Semantik eines Programms. Die verschiedenen Semantikansätze werden in Abschnitt 2.1.2, die daraus entstandenen Berechnungsmodelle in Abschnitt 2.1.3 vorgestellt. In Abschnitt 2.1.4 folgt eine Übersicht von Verfahrenstechniken, die im Bereich der deduktiven Datenbanken zur Optimierung von Ausführungsmethoden angewendet werden. Bekannte Programmierhochsprachen aus diesem Bereich sind PROLOG und DATALOG. Auf sie wird in Abschnitt 2.1.5 eingegangen.

Bei prozeduralen oder objektorientierten Programmen läßt sich Parallelität zumeist nur auf Prozedur- oder Blockebene nutzen. Da in Prozeduren oder Blöcken eine Menge von Anweisungen zusammengefaßt sind, spricht man von grobkörniger Parallelität. Die Disjunktheit von Kontrollfluß und Datenfluß erzwingt zudem die Verwendung effizienzmindernder Synchronisationsmechanismen. Die nutzbare Parallelität in PROLOG, bzw.

Datalog befindet sich dagegen auf der Ebene einzelner Regeln (Oder-Parallelität) und innerhalb von Regeln auf der Ebene einzelner Anweisungen (Und-Parallität). Es liegt also nahe, für die parallele Implementierung von Prolog, bzw. Datalog nicht mehr die von Neumann-Rechnerarchitektur zu wählen, sondern eine Rechnerarchitektur, die aufgrund der Identität von Kontrollfluß und Datenfluß sehr viel besser für die Nutzung feinkörniger Parallelität geeignet ist. Eine solche Rechnerarchitektur stellt die Datenflußrechnerarchitektur dar, die in Kapitel 2.2 vorgestellt wird.

Die Basissprache für Datenflußrechner sind die in Abschnitt 2.2.1 beschrieben Datenflußgraphen. Jede Operation eines Programms wird in einen Knoten des Graphen umgesetzt. Die Kanten des Graphen stellen die Verbindungen der Operationen entsprechend ihrer Datenabhängigkeiten dar. Auf ihnen werden die Daten in Form von Token von Knoten zu Knoten transportiert. Operationen können immer dann ausgeführt werden, wenn alle benötigten Eingaben vorhanden sind. Statt einer kontrollflußgesteuerten Ausführung liegt hier also eine datenflußgesteuerte Ausführungsform vor. Es ist keine zentrale Steuerung mehr notwendig. Parallelität im Datenflußgraphen muß nicht explizit durch spezielle Anweisungen festgelegt werden, sondern ist automatisch immer dann gegeben, wenn zwei Operationen voneinander datenunabhängig sind. Feinkörnig parallele Programme können daher auf einer Datenflußrechnerarchitektur mit mehreren Prozessoren sehr effizient ausgeführt werden.

Die Prinzipien statischer, dynamischer und hybrider Datenflußrechnerarchitekturen werden in Abschnitt 2.2.2 erläutert. Abschnitt 2.2.3 stellt Projekte aus diesem Gebiet vor.

In Abschnitt 2.4 wird die Arbeit in das bestehende Umfeld bisher existierender Ansätze zur datengetriebenen Auswertung im Bereich der Logikprogrammierung eingeordnet.

Implementierungen von Logikprogrammen auf Datenflußrechnern stellen sich häufig die gleichen Probleme, wie parallelen Logikprogramm-Implementierungen auf von Neumann-Rechnern: es wird einmal eine Ausführung aller benötigten Regeln zur Aktivierung durch Anforderungen benötigt, sowie eine weitere Ausführung für die Rückgabe der Ergebnisse. Dabei schränkt die erforderliche Aktivierung, die größtenteils nur sequentiell erfolgen kann, die mögliche Parallelität drastisch ein. Alternativ kann auf eine explizite Aktivierung mit dem Nachteil, daß bei einer vollständigen Parallelisierung zu viele redundante Zwischenlösungen berechnet werden, verzichtet werden. Arbeiten in diesem Bereich und ihre Probleme werden in Abschnitt 2.4.1 diskutiert.

Wie in der Einführung bereits erläutert wurde, arbeitet ein Programm für die Steuerung autonom-mobiler Systeme grundlegend anders, als ein Standard-Expertensystem. Die gewünschten Ausgaben sind zur Ausführungszeit fest, während sich die durch die Fakten ausgegebenen Daten dynamisch ändern. Dies schafft die Voraussetzung für ein neues Datenflußmodell für die zu behandelnde Klasse von Expertensystemen, das, wie sich zeigen wird, auf entsprechenden Datenflußrechnerarchitekturen das Potential für einen deutlichen Leistungsgewinn gegenüber bisher verwendeten Implementierungsformen wissensbasierter Systeme schafft. Als Grundlage für das Datenflußmodell dienen Verfahrenstechniken aus dem Bereich der deduktiven Datenbanken die in Abschnitt 2.4.2 aufgeführt werden.

Demzufolge beschäftigt sich Kapitel 3 mit der Methode zur Transformation eines wissensbasierten Systems in einen Datenflußgraphen. Zu Beginn wird in Abschnitt 3.1 die Klasse der Logikprogramme vorgestellt, die sich für die Spezifikation eines Steuerungssystems und die Umsetzung in einen Datenflußgraphen eignen. Die Konstruktion des Datenflußgraphen geschieht dann in drei Schritten. Zunächst wird in Abschnitt 3.2 aus einem Logikprogramm ein Graph abgeleitet, der vorerst nur die syntaktischen Verbindungen der Regeln und Regelteile repräsentiert. Der Graph wird in Abschnitt

3.3 mit dem Ziel, die Datenabhängigkeiten aller Parameter zu bestimmen, analysiert. In Abschnitt 3.4 wird aus dem Graph mit Hilfe des Ergebnisses dieser Analyse ein Datenflußgraph erzeugt. Es folgt ein Beispiel in Abschnitt 3.5. Die Spezifizikation der Funktionalität der Knoten des Datenflußgraphen wird in 3.6 vorgenommen. In Abschnitt 3.7 wird die Korrektheit des Verfahrens nachgewiesen, indem gezeigt wird, daß die Transformation des Programms in einen Datenflußgraphen semantikerhaltend ist.

Einzelheiten über das Berechnungsmodell des Datenflußgraphen beinhaltet Kapitel 4. Der Schwerpunkt liegt auf der Vorstellung des schrittgesteuerten Datenflußberechnungsmodells in Abschnitt 4.1. Zusätzlich werden in Abschnitt 4.2 die ereignisflußgetriebene Abarbeitung und in Abschnitt 4.3 eine mehrschichtige Ausführung als Optimierungsformen eingeführt.

Eine Übersicht über mögliche Erweiterungsformen des Datenflußmodells für mathematische Algorithmen, Fuzzy Logic und neuronale Netze, sowie die Anbindung an Simulationssysteme liefert Abschnitt 4.4.

Kapitel 5 befaßt sich mit der Implementierung des Berechnungsmodells. Dieses besteht aus dem in Abschnitt 5.1 vorgestellten Compiler, der eine geeignete Spezifikation in einen ausführbaren Datenflußgraphen transformiert. Des weiteren beinhaltet die Implementierung den in Abschnitt 5.2 vorgestellten Simulator, der u.a. für die Erstellung der Testdaten verwendet wurde. In Abschnitt 5.3 wird eine, für die Ausführung des Berechnungsmodells geeignete Datenflußrechnerarchitektur beschrieben, Abschnitt 5.4 gibt darüber hinaus einen Überblick über die Einsetzbarkeit auf verschiedenen von Neumann-Systemen.

Den Abschluß der Arbeit bildet Kapitel 6 mit einer Zusammenfassung und Bewertung der in der Arbeit erzielten Ergebnisse.

2 Grundlagen

Diese Arbeit bezieht ihre Grundlagen aus zwei verschiedenen Fachgebieten:

- deduktive Datenbanken als Grundlage für die Spezifikations- und Ausführungsprinzipien und
- datenflußgetriebene Rechnerarchitekturen als Grundlage für das zu erstellende Berechnungsmodell.

Dementsprechend befaßt sich Abschnitt 2.1 mit den Prinzipien deduktiver Datenbanken, bestehend aus

- der Logikprogrammierung als theoretisches Fundament,
- den verschiedenen Semantikansätzen,
- den davon abgeleiteten sequentiellen und parallelen Berechnungsmodellen, sowie
- Verfahren zur Erzeugung und Optimierung von Berechnungsmodellen und
- verbreiteten Hochsprachen für die Implementierung.

Abschnitt 2.2 beschreibt die Konzepte datenflußgetriebener Rechnerarchitekturen:

- die "Maschinensprache" in Form von Datenflußgraphen, sowie
- die verschiedenen Architekturprinzipien

und gibt einen Überblick über Projekte in diesem Bereich.

Eine Einordnung der Arbeit in das existierende Themenfeld erfolgt in Abschnitt 2.4.

2.1 Deduktive Datenbanken

Der Begriff der deduktiven Datenbanken bezeichnet ein Forschungsgebiet, das sich aus Datenbankforschung und logischer Programmierung entwickelt hat.

Ausgehend von klassischen (relationalen) Datenbankmodellen geht es um die Realisierung effizienter, automatischer Inferenz-, bzw. Schlußfolgerungsmethoden, die es erlauben, große Faktenmengen durch einzelne definierende Ausdrücke statt durch explizite Aufzählung darzustellen. Logische Formeln erlauben hierfür in sehr einheitlicher Weise die Beschreibung von Daten, Regeln, Anfragen, Sichten und Integritätsbedingungen.

Prinzipiell basieren deduktive Datenbanken auf relationalen Datenbanken, die um Konzepte der logischen Programmierung erweitert werden. Da viele Thematiken, die Datenbanksysteme generell betreffen, wie Benutzeranzahl, Organisation großer Datenmengen, oder Transaktionsprotokolle, in der vorliegenenden Anwendung keine Rolle spielen, werden in dieser Arbeit deduktive Datenbanken rein von der Seite der logischen Programmierung betrachtet.

2.1.1 Logikprogrammierung

Logikprogrammierung ist aus dem mathematischen Prinzip der Logik entstanden.

Auf dem Konzept der Logikprogrammierung basierende Sprachen bezeichnet man auch als applikative Programmiersprachen. Sie unterscheiden sich wesentlich von anderen Sprachgruppen, wie z.B. prozeduralen, objektorientierten, oder funktionalen Sprachen.

Sprachen aus den letztgenannten Bereichen ermöglichen es dem Programmentwickler, Arbeitsabläufe Schritt für Schritt in Form von Algorithmen anzugeben. Eine solche Beschreibung ist von Natur aus sequentiell, einzelne Arbeitsblöcke können aber mit speziellen Methoden zueinander parallelisiert werden.

Die Verwendung von Logik als Programmiersprache zielt dagegen auf eine stärker deskriptive und weniger präskriptive Formulierung von Algorithmen ab. Anstatt Arbeitsabläufe anzugeben, geht es darum, Wissen über ein Problemfeld zu spezifizieren. Dies geschieht mit einfachen Regeln der Form "wenn ... dann ...".

Die Lösung für einen speziellen Fall aus dem Problemfeld wird dann ohne zusätzliche, explizite Angabe des Lösungswegs durch Schlußfolgerungen über der Menge der Regeln ermittelt. Dieses Verfahren nennt man Inferenz- oder Schlußfolgerungsmechanismus [Knauss 87]. Die Korrektheit der Ausführung eines Programms und damit des Ergebnisses kann mit Hilfe der zugrunde liegenden Prinzipien der Logik bewiesen werden [Dodd 90].

Die Reihenfolge der Schlußfolgerungen ist dabei nicht festgelegt. Statt dessen können prinzipiell alle Regeln, deren Eingabeargumente zu einem bestimmten Zeitpunkt feststehen, parallel zueinander ausgewertet werden.

Ein Logikprogramm besteht aus drei unterschiedlichen Teilen:

- einer *extensionalen Datenbank*,
  in der Eigenschaften von Objekten in Form von Fakten definiert werden,
- einer *intensionalen Datenbank*, bzw. einem *Regelsystem*,
  in dem Regeln Relationen, d.h. Beziehungen zwischen den Objekten, beschreiben, und
- den *Abfragen*,
  die zur Laufzeit an das Programm gestellt werden und das Berechnungsziel des jeweiligen Programmlaufs darstellen.

2.1.1.1 Formale Grundlagen

Die im folgenden aufgeführten Definitionen und Bezeichnungen lehnen sich weitestgehend an die allgemein gebräuchliche Terminologie an, wie sie auch in [Lloyd 87] zu finden ist.

Definition 2.1 *Grundzeichen*

Eine Prädikatenlogik erster Stufe enthält die folgenden Grundzeichen:

1. Abzählbar viele Variablen $v \in V$. V sei die Menge der Variablen.
2. Abzählbar viele n-stellige Funktionszeichen $f \in Fun$ (mit $n \in \mathbb{N}$). Fun sei die Menge der Funktionszeichen.

   2-stellige Funktionszeichen (z.B. $+, -, *, /$) können auch als Infixoperatoren geschrieben werden.

3. Abzählbar viele n-stellige Prädikatszeichen $p \in Pred$ ($n \in \mathbb{N}$). *Pred* sei die Menge der Prädikatszeichen.

   Unter den Begriff Prädikatszeichen fallen auch:

   i) *Vordefinierte Funktionen* $p_\rho \in MP \subset Pred$, wobei MP die Menge der *Metaprädikate (Systemprädikate)* sei. ρ stehe dabei für das jeweilige Systemprädikat;
   insbesondere sei auch $p_f \in MP \subset Pred$, mit $f \in Fun$

   ii) 2-stellige Vergleichs- und Zuweisungssymbole $\tau \in Rel \subset Pred$, wobei *Rel* die Menge der Relationssymbole und τ eines der folgenden Symbole sei:
   - *is* steht für die Zuweisung,
   - $=_a, \neq_a, <_a, \leq_a, >_a, \geq_a$ für arithmetische Vergleiche und
   - $\equiv_u, \neq_u, <_u, \leq_u, >_u, \geq_u$ für den Vergleich von Termen durch Unifikation.

   p_τ bezeichne hierbei die Präfixschreibweise zu τ (d.h.: $p_\tau(a, b) \equiv a\tau b$).

   Die 0-stelligen Prädikatzeichen (z.B. *true* und *false*) heißen aussagenlogische Konstanten.

4. Die logischen Junktoren $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies) und $\leftrightarrow$ (equivalent).

5. Die Quantoren $\forall$ (forall) und $\exists$ (exists).

6. Die Punktuationszeichen "(", ")", "." und ",". ◇

Obwohl die Syntax von Funktionen einerseits und Literalen mit Prädikaten andererseits identisch ist, sind diese jedoch eindeutig voneinander unterscheidbar. Funktionen und Systemprädikate sind durch eine Compiler-Bibliothek definiert, Prädikate, die keine Systemprädikate darstellen, durch Regeln oder Fakten im Programm. Funktionen nehmen außerdem Werte als Argumente und geben Werte zurück. Prädikate nehmen Werte als Argumente und geben ausschließlich Wahrheitswerte zurück.

Definition 2.2 *Terme*

Ein Term $t \in T$ sei induktiv definiert durch:

i) eine Konstante a ist ein Term,

ii) eine Variable x ist ein Term,

iii) wenn $t_1, ..., t_n$, $n \in \mathbb{N}$, Terme sind und $f \in Fun$ eine n-stellige Operation auf Termen ist, dann ist $f(t_1, ..., t_n)$ wieder ein Term. ◇

Definition 2.3 *Atome und Literale*

Sei p ein n-stelliges Prädikatzeichen, $t_1, ..., t_n$ seien Terme. Dann ist $p(t_1, ..., t_n)$ eine *atomare Formel* oder kurz *Atom.*

Ein *positives Literal* P ist ein Atom $p(t_1, ..., t_n)$ (d.h. $P \equiv p(t_1, ..., t_n)$). Ein *negatives Literal* P ist ein negiertes Atom $\neg p(t_1, ..., t_n)$ (d.h. $P \equiv \neg p(t_1, ..., t_n)$).

$(t_1, ..., t_n)$ wird im folgenden als Parameterliste des Literals P bezeichnet, $t_1, ..., t_n$ als Parameter. ◇

Definition 2.4 *Deduktive Regeln*

Eine *deduktive Regel* hat die Form

$$C_1, ..., C_k \leftarrow A_1, ..., A_m, \neg B_1, ..., \neg B_n$$

mit Literalen $C_1, ..., C_k, A_1, ..., A_m, \neg B_1, ..., \neg B_n$ und $k, m, n \in \mathbb{N}$.

$C_1, ..., C_k$ heißt *Kopf* der Regel, $A_1, ..., A_m, \neg B_1, ..., \neg B_n$ bilden zusammen den *Rumpf*.

Eine Regel mit leerem Rumpf heißt *Fakt*. Eine Regel mit leerem Kopf heißt *Abfrage* oder *Goal*.

Regeln werden auch als Klauseln bezeichnet. ⋄

Die Definition eines Logikprogramms erfolgt im Hinblick auf die später zu beschreibenden Graphen bezüglich der Indizierung geringfügig anders, als allgemein üblich. Zusätzlich verwendete Begriffe wie Formeln, Substitution, Unifikation, etc., können in Anhang A.3 nachgelesen werden.

Definition 2.5 *Logikprogramm*

Ein *(normales) Logikprogramm* beinhaltet eine endliche Menge deduktiver Regeln C, wobei mit $n \in \mathbb{N}$ gelten soll:

$C := \{c \mid c \equiv P^0_{i,k} \leftarrow P^1_{i,k}, ..., P^{l_{i,k}}_{i,k}.$, mit $n \in \mathbb{N}$, $i \in \{1, ..., n\}$, $m_i \in \mathbb{N}$, $k \in \{1, ..., m_i\}$, $l_{i,k} \in \mathbb{N}\}$

Dabei gilt:
- n ist die Anzahl verschiedener Literale, die als Kopf einer Regel vorkommen,
- m_i ist die Anzahl der Regeln mit dem gleichen Literal P_i als Kopf und
- $l_{i,k}$ ist die Anzahl der Literale im Rumpf einer Regel mit dem Kopf $P_{i,k}$. ⋄

Der Index i in der oben verwendeten Schreibweise für Regeln ist ein Regelindex, k ein Index für alternative Regeln und $l_{i,k}$ ein lokaler Regelindex.

Definition 2.6 *Abfrage an ein Logikprogramm*

Eine *Abfrage* Q an ein (normales) Logikprogramm C ist eine deduktive Regel:

$Q := \leftarrow P^1_{0,1}, ..., P^{l_{0,1}}_{0,1}.$, mit $l_{0,1} \in \mathbb{N}$

(Die unteren Indizes dienen hier ausschließlich der Einheitlichkeit des Formalismus im weiteren.) ⋄

Die Literale im Rumpf einer Regel werden bei Anwendung einer Regel wie Abfragen behandelt. Sie werden daher ebenfalls häufig als Abfrage bezeichnet.

Bemerkung 2.7

Im folgenden sei ein Logikprogramm $\mathcal{L}$ immer eine Menge deduktiver Regeln C zusammen mit einer Abfrage Q. ⋄

Das Belegen einer Variablen mit einem Wert oder einer Menge von Werten wird auch *Binden* einer Variablen an Werte genannt. Dies geschieht im allgemeinen durch Unifikation einer Abfrage oder eines Literals im Rumpf einer Regel mit dem Kopf einer Regel oder einem Fakt.

Ergänzende Bemerkungen:

1. In der Parameterliste eines Regelkopfs oder eines Rumpfliterals können Funktionen auftreten. Dies sind entweder zusammengesetzte Terme, d.h. Terme mit Operationssymbolen (z.B. $x-1$) oder Terme, die Funktionsaufrufe (z.B. $incr(x)$) darstellen.

2. In der Parameterliste eines Regelkopfs oder eines Rumpfliterals können Listen auftreten. Einfache Listen sind von der Form a oder X, wobei a Konstante ist, X Variable. Listen mit Konstruktoren sind von der Form $[l]$ oder $[l_1|...|l_n]$, wobei l und $l_1, ..., l_n$, $n \in \mathbb{N}$ wiederum Listen sind.

3. Da $Rel \subset Pred$ gilt, können als Literale im Rumpf einer Regel Zuweisungen (z.B. X *is* $Y * Z$) und Vergleiche (z.B. $X > Y * Z$) auftreten. Die formale Darstellung ist $p_\tau(X, t)$ mit $\tau \in Rel$ für den Ausdruck $X\tau t$, wobei X Variable ist und t Term.

4. Da $MP \subset Pred$ gilt, können als Literale im Rumpf einer Regel Aufrufe vordefinierter Funktionen auftreten (z.B. $write(x)$). Diese Metaprädikate müssen jedoch der Bedingung genügen, daß sie ausschließlich einen boolschen Funktionswert besitzen.

5. Das Binden einer Variablen an Werte kann außer durch Unifikation entweder durch eine Zuweisung (*is* oder $=_u$) geschehen, oder durch die Anwendung eines Metaprädikates auf die übergebenen Parameter.

Als letztes ist noch der Begriff der Unifizierbarkeit von Literalen zu definieren. Die allgemeine Definition von Unifizierbarkeit ist in Anhang A.3 nachzulesen.

Definition 2.8 *Unifizierbarkeit zweier Literale*

Zwei Literale sind genau dann unifzierbar,

- wenn die Prädikate beider Literale identisch (syntaktisch gleich) sind,
- wenn die Anzahl der Parameter in den Parameterlisten gleich ist und
- wenn eine Substitution für die Variablen (bzw. Terme) in den Parameterlisten derart existiert, so daß die Literale syntaktisch identisch werden. ◇

Folgerung 2.9

Sind zwei Literale unifzierbar, so sind auch die Parameter an der jeweils gleichen Position in den Parameterlisten der Literale unifizierbar. ◇

2.1.1.2 Negation

Intuitiv geht man von der Annahme aus, daß alle Fakten, die nicht explizit als wahr gegeben sind, falsch sind. Negative Literale sind jedoch keine semantische Folgerung aus einem Programm, da sich kein semantisches Modell erstellen läßt, in dem die Negation eines Literals wahr ist [Cremers 94].

Eine Lösung dieses Problems ist die *Closed World Assumption (CWA)*, eine Inferenzregel (Schlußfolgerungsregel), mit der sich ein negatives Literal aus einem Programm ableiten läßt, wenn das Literal selbst keine semantische Folgerung des Programms ist. Sie geht von der Annahme einer vollständigen Datenbank aus. Alles was wahr ist, ist explizit gegeben. Alles was nicht explizit gegeben ist, ist falsch. Da es jedoch aufgrund der Unentscheidbarkeit der Prädikatenlogik erster Stufe keinen Algorithmus gibt, der in endlich vielen Schritten entscheidet, ob eine Formel eine semantische Folgerung aus einer Datenbank ist oder nicht, benötigt man für die Praxis eine andere, davon ableitbare Lösung.

Eine real implementierbare Form der Closed World Assumption ist daher *Negation as Failure* [Cremers 94]: ein negatives Literal ist dann eine semantische Folgerung aus einem Programm, wenn in endlich vielen Schritten gezeigt werden kann, daß das Literal selbst keine semantische Folgerung ist.

Dennoch ist auch die Implementierung einer Negation mit *Negation as Failure* problembehaftet. Erweiterungen sind z.B., daß alle Variablen der zu negierenden Abfrage instanziiert sein müssen. Man verhindert damit u.a. das Auftreten unendlicher Rekursionsdurchläufe. Diese Erweiterung entspricht der Forderung nach (bedingter) Sicherheit für Programme (die Definition bedingter Sicherheit ist im Anhang A.2 nachzulesen) [Cremers 94].

Eine andere Erweiterung ist z.B., daß das Verlaufen der Berechnung in unendlichen Zweigen im Berechnungsbaum explizit abgefangen wird. Insbesondere genügt ein erfolgreicher Lösungszweig zum Scheitern von `not`, es müssen nicht alle möglichen Lösungen gefunden werden.

2.1.1.3 Rekursionen

Schleifen werden in einem Logikprogramm grundsätzlich durch Rekursionen realisiert. Da sich jede Iteration als Rekursion ausdrücken läßt, bedeutet dies keinerlei Einschränkung für die Mächtigkeit der Sprache.

Rekursion bedeutet, daß eine Regel sich selbst, gegebenenfalls über den Weg verschiedener anderer Regeln, wieder aufruft.

Eine korrekt implementierte Rekursion benötigt mindestens zwei alternative Regeln. Eine Regel, die "Terminierungsregel", realisiert den Terminierungsfall. Eine zweite Regel, die "Rekursionsregel" enthält den Rekursionsaufruf. Alternativ kann es auch mehrere Terminierungs- und Rekursionsregeln für eine Rekursion geben.

Es gibt prinzipiell folgende Arten von Rekursionen:

1. *Lineare Rekursion*
   Eine Rekursion heißt linear rekursiv, wenn in jeder alternativen Regel im Rumpf höchstens ein weiterer rekursiver Aufruf auftritt.

2. *Kaskadische Rekursion*
   Treten bei einem Zweig des Rumpfs mindestens zwei rekursive Aufrufe auf, spricht man von kaskadenartiger Rekursion.

3. *Vernestete Rekursion*
   Tritt ein rekursiver Aufruf im Rumpf in der Parameterliste eines rekursiven Aufrufs auf, so ist dies eine vernestete Rekursion. (Dies ist in der Logikprogrammierung i.a. jedoch nicht zulässig.)

4. *Verschränkte Rekursion*
   Durch verschränkte Rekursion wird eine Familie von Funktionen rekursiv deklariert.

Die Terminierung von Rekursionen ist nicht immer gewährleistbar, sondern ist häufig von der vom Programmierer gewählten Aufschreibungsreihenfolge der Regeln, sowie von den, von Benutzern gestellten, Abfragen abhängig [Sterling 86].

Es gibt verschiedene Möglichkeiten für die Terminierung einer Rekursion:

- Wenn ein Literal in der Rekursionsregel scheitert, scheitert die Rekursion und terminiert damit.

- Zu einer im Zyklus enthaltenen Regel gibt es eine alternative Regel (Terminierungsregel), die ihrerseits den Zyklus nicht fortsetzt. Genügt für den Erfolg einer Programmausführung das Finden einer einzigen Lösung, so terminiert die Rekursion, sobald die Terminierungsregel erfolgreich ist.

- Sollen alle möglichen Lösungen gefunden werden, erfolgt die Terminierung über die Berechnung des kleinsten Fixpunktes der Menge in der Rekursion berechneter Werte.

2.1.2 Semantik von Logikprogrammen

Ein Logikprogramm kann unter zwei verschiedenen Gesichtspunkten betrachtet werden. Die *deklarative Bedeutung* eines Programms betrifft die logischen Relationen des Programms. Dagegen legt die *prozedurale Bedeutung* fest, wie diese Relationen vom Logikprogramm bearbeitet werden.

Aus deklarativer Sicht kann die Regel $P_0 \leftarrow P_1, ..., P_n$ gelesen werden als: "Aus $P_1, ..., P_n$ kann P_0 abgeleitet werden", bzw. "P_0 ist wahr, wenn P_1 bis P_n wahr sind".

Dagegen kann die Regel aus operationeller Sicht gelesen werden als: "Damit P_0 erfüllt ist, muß zuerst P_1, dann P_2, bis hin zu P_n erfüllt sein".

2.1.2.1 Operationelle Bedeutung

Die operationelle Semantik wird durch einen abstrakten Logikinterpreter realisiert. Das Kernstück des abstrakten Logikinterpreters ist der Unifikationsalgorithmus. Ein abstrakter Logikinterpreter und ein Unifikationsalgorithmus werden in [Sterling 86] vorgestellt und sind im Anhang A.3 aufgeführt.

Der abstrakte Interpreter agiert in zwei Fällen nicht-deterministisch: bei der Wahl des nächsten zu unifizierenden Literals aus dem Rumpf einer angewendeten Regel und bei der Wahl der nächsten anwendbaren Regel aus der Menge der Regeln, deren Kopf mit der Abfrage oder einem gewählten Literal unifizierbar sind.

In realen Implementierungen eines Logikinterpreters müssen beide Fälle von Nichtdeterminismus des abstrakten Interpreters ersetzt werden:

- Es wird immer die Regel gewählt, die in der Aufschreibung als erste steht. Scheitert diese Regel, wird an den Auswahlpunkt zurückgegangen, alle Variablenbelegungen, wie sie zuletzt an dieser Stelle waren, restauriert und die in der Aufschreibung folgende Regel gewählt. Dieses Verfahren nennt man *Backtracking*. Existiert keine weitere Regel mehr, ist die übergeordnete Regel gescheitert.
- Die Literale im Rumpf einer Regel werden in ihrer Aufschreibungsreihenfolge von links nach rechts behandelt.

Der durch die Regeln des Programms aufgestellte Berechnungsbaum wird folglich mit Tiefensuche von links nach rechts durchlaufen. Dies gilt sowohl für sequentielle Formen der Abarbeitung, als auch für parallele Formen, die nur ein Ergebnis berechnen sollen.

Diese Form der Ausführung nennt sich SLD-Resolution (*linear resolution with selected function for definite clauses*).

Ein wesentliches Element der SLD-Resolution ist das *Backtracking*. Dabei handelt es sich um ein schrittweises Zurückgehen und wiederholtes Berechnen mit dem Ziel einer weiteren Lösung, wenn eine bereits erreichte Lösung im Verlauf eines Berechnungspfads gescheitert ist.

Die Reihenfolge der Regeln und der Literale im Rumpf der Regeln haben aufgrund der verschiedenen Pfade durch den Berechnungsbaum einen großen Einfluß auf die Effizienz, ebenso wie auf die Terminierung der Ausführung. Die Terminierung ist zusätzlich von der Art der Abfrage, d.h. von der Menge der gebundenen Variablen und ihrer Werte in der Abfrage abhängig [Sterling 86].

Die Effizienz eines Programms durch geschickte Anordnung der Regeln oder der Literale im Rumpf von Regeln zu steigern, oder einer Nicht-Terminierung vorzubeugen, bleibt damit dem Geschick des Programmierers, dem die Reihenfolge der Auswertung durch die SLD-Resolution bekannt ist, überlassen.

2.1.2.2 Deklarative Bedeutung

Die Ergebnisse eines Logikprogramms werden (theoretisch) nur vom deklarativen Aspekt bestimmt, d.h. das Logikprogramm ermittelt Lösungen für die gestellte Abfrage ohne explizite Angabe des Lösungswegs und damit ohne Berücksichtigung der Reihenfolge der Regeln [Bratko 87].

Definition 2.10 *Interpretation, Modell*

i) Eine *Interpretation* für ein Logikprogramm ist die Abbildung aller Prädikatsymbole auf Relationen, aller Funktionssymbole und Konstanten auf sich selbst und aller Variablen auf Variablenbelegungen. Jeder Regel und jedem Fakt wird als Bedeutung ein Wahrheitswert zugeordnet.

ii) Ein *Modell* ist eine Teilmenge der Interpretation, für die die Wahrheitswerte *wahr* ergeben. ◇

Definition 2.11 *Semantische Folgerung*

Eine Abfrage heißt semantische Folgerung aus einem Logikprogramm $\mathcal{L}$ genau dann, wenn jede Interpretation, die Modell von $\mathcal{L}$ ist, auch Modell der Abfrage ist. ◇

Um die möglichen Interpretationen eines Logikprogramms zu minimieren, wird der Begriff der Herbrand-Interpretation eingeführt.

Definition 2.12 *Herbrand-Universum, Herbrand-Basis*

i) Ein Herbrand-Universum ist die Menge aller variablenfreien Terme von $\mathcal{L}$.

ii) Eine Herbrand-Basis ist die Menge aller variablenfreien Abfragen von $\mathcal{L}$, die sich aus den Prädikaten und dem Herbrand-Universum formen lassen. Ist das Herbrand Universum unendlich, so ist damit auch die Herbrand Basis unendlich. ◇

Folglich ist eine *Herbrand-Interpretation* eine Interpretation, die den Elementen der Herbrand-Basis Wahrheitswerte zuordnet. Ein Herbrand-Modell ist eine Herbrand-Interpretation von $\mathcal{L}$, die Modell von $\mathcal{L}$ ist [Cremers 94].

Bildet man die Schnittmenge aller Herbrand-Modelle, so erhält man ein *minimales Herbrand-Modell*. Das minimale Herbrand-Modell beschreibt die deklarative Bedeutung des Programms $\mathcal{L}$ [Sterling 86]. Für ein Logikprogramm gibt es immer ein minimales Herbrand-Modell [Lloyd 87]. Dieses muß jedoch nicht eindeutig sein.

Die Bedeutung eines Logikprogramms läßt sich nun hinsichtlich der Korrektheit exakt und hinsichtlich der Vollständigkeit mit Einschränkungen, z.B. wenn das Programm Negationen enthält, bestimmen. Damit wird jedes Logikprogramm (in gewissem Maße) verifizierbar [Ullman 88].

Die korrekte Behandlung der Negation ist durch die Herbrand-Semantik nicht möglich: für Programme mit Negationen ist kein eindeutiges minimales Herbrand-Modell garantierbar. Aufgrund dieses Problems wurden verschiedene Semantik-Modelle entwickelt [Cremers 94]:

- Für die Klasse der definiten Programme, d.h. der Programme ohne Negation, genügt das *kleinste Herbrand-Modell* zur korrekten Semantikbeschreibung.
- Das *perfekte Modell* ist eine hinreichende Semantikbeschreibung für die Klasse der stratifizierten Programme. Das perfekte Modell entspricht einem minimalen Herbrand Modell (aus der Menge der minimalen Herbrand Modelle).
- Die *Stable-model Semantik* ist eine Semantikbeschreibung für normale Programme, für die gilt: wenn negative Literale gemäß dem Modell ausgewertet werden, besitzt das resultierende definite Programm dasselbe Modell. Der Nachteil dieser Semantik ist, daß es nicht für alle Programme ein Modell geben muß, bzw. daß dieses eindeutig ist.
- *Vervollständigungs-Semantiken* beruhen auf der Vervollständigung des Programms. Im wesentlichen werden bei einer Vervollständigung die Implikationen in den Regeln zu Äquivalenz-Bedingungen transformiert, um die *Closed World Assumption*, die für die Negation von Literalen nötig ist, formalisierbar zu machen.
  Bei der Verwendung von zweiwertiger Logik ist allerdings nicht jedes vervollständigte Programm erfüllbar. Die Verwendung von dreiwertiger Logik dagegen birgt Schwächen in der Aussdrucksstärke.
- Die *Well-founded Semantik* basiert auf einer dreiwertigen Logik und kann jedem Programm eine Semantik zuordnen. Für *schwach stratifizierte* Programm genügt für die well-founded Semantik sogar eine zweiwertige Logik. (Die Definition schwach stratifizierter Programme ist in Anhang A.2 nachzulesen.)

Funktionen bei der Betrachtung von Semantiken nur sehr eingeschränkt behandelt, Systemprädikate werden sogar ganz beiseite gelassen.

Es gibt eine Reihe neuerer Arbeiten mit dem Ziel, effektive, vollständige Verfahren für stable-model Semantik [Gelfond 88], aber auch insbesondere zur Berechnung der well-founded Semantik [Kemp 95, Brass 95a, Brass 95b], zu erhalten:

- Der alternierende Fixpunktansatz wurde von [Gelder 88] entwickelt. Er garantiert ein polynominales Wachstum der Tupelmengen, leidet aber unter Effizienzproblemen, da viele Fakten in jeder Iteration neu berechnet werden müssen (siehe auch [Gelder 89, Gelder 91, Gelder 93]).
- Die SLG-Resolution (linear resolution with selection function for general logic programs) [Chen 93, Chen 95, Chen 96] basiert auf einer Programmtransformation und einer anschließenden tabellarischen top-down Auswertung unter Vermeidung mehrfacher Berechnungen. Unendliche Rekursionen können dabei auch im Fall negierter Aufrufe abgefangen werden. Die SLG-Resolution wurde im XSB-System [Sagonas 94,

Pelov 99] erfolgreich implementiert. Durch die Verwendung von Tabellen wird jede Neuberechnung von Fakten vermieden, es kann jedoch zu einem exponentiellen Wachstum der Tupelmengen bei nicht definiten Programmen kommen, Das XSB-System verwendet als Sprache für deduktive Datenbanken HiLog. Das Verfahren bezieht sich speziell auf die well-founded Semantik, es läßt sich jedoch auch für die stable-model Semantik anwenden.

- Das LOLA-Sytstem [Brass 97a, Brass 97b] berechnet die Well-founded Semantik effizient. Das Verfahren stellt eine vereinfachende Bottom-Up Ausführung der SLG-Resolution für bereichsbeschränkte Programme dar. Unter Verwendung von lediglich einer Teilmenge von Grundinstanziierungen kann wie beim alternierenden Fixpunkt Ansatz ein polynominales Wachstum der Tupelmengen garantiert werden. Das Verfahren eignet sich für die well-founded Semantik, läßt sich aber auch für die stable-model Semantik als Vorverarbeitungsschritt anwenden.

2.1.3 Berechnungsmodelle

Die in Abschnitt 2.1.2.2 vorgestellten Modelle dienen als Grundlage für Berechnungsmodelle deduktiver Datenbanken.

Die Auswertungsmechanismen für deduktive Datenbanken und damit auch für Logikprogramme lassen sich zunächst anhand der Richtung der Auswertung klassifizieren.

1. *Top-down Verfahren*

   Logik-Kalküle oder Deduktionssysteme sind einfache syntaktische Systeme, die mit Hilfe von Regeln festlegen, wie aus einer gegebenen Menge von Formeln neue Formeln abgeleitet werden können.

   Das bekannteste Verfahren ist die SLD-Resolution (siehe auch Abschnitt 2.1.2.1). Sie geht von einer Abfrage, d.h. einer zu beweisenden Formel aus. Unter Verwendung der Regeln des Programms werden neue zu beweisende Formeln erzeugt. Gelingt der Beweis für alle Formeln, oder läßt sich keine neue Formel mehr erzeugen, terminiert das Verfahren.

   Diese Auswertungsrichtung heißt *top-down* oder auch *backward-chaining*.

   Bei Top-Down Verfahren erweist sich Breitensuche besser als Tiefensuche, wenn es um die Vollständigkeit der Auswertung geht, da sich bei Tiefensuche der Berechnungsweg in einem unendlichen Zweig verlaufen kann, bevor alle Lösungen gefunden sind. Breitensuche dagegen ergibt alle Lösungen, bevor sich die Berechnung in einem unendlichen Zweig verlaufen kann.

2. *Bottom-Up Verfahren*

   Bottom-Up Verfahren sind Auswertungsverfahren, bei denen aus den Fakten des Programms mit Hilfe der Regeln neue Fakten erzeugt werden [Ullman 88]. Dies entspricht der Erzeugung von Herbrand-Modellen, bzw. der Berechnung des kleinsten Fixpunktes [Lloyd 87] von definiten Logikprogrammen. Der Vorgang terminiert, wenn keine neuen Fakten mehr hergeleitet werden können, oder eine gegebene Abfrage bewiesen ist.

   Die Auswertungsrichtung heißt *bottom-up* oder auch *forward-chaining*.

   Unter Bottom-Up Verfahren fallen *naive* und *semi-naive Auswertung*. Der Vorteil des semi-naiven Ansatzes gegenüber dem naiven Ansatz besteht darin, daß keine Berechnung durch Fakten mehrfach erfolgen muß.

Prinzipiell können Bottom-Up Verfahren sehr schnell sein, wobei der Nachteil dieser Verfahren jedoch darin besteht, daß mit allen Fakten gearbeitet wird und die Einschränkung auf die für eine gestellte Abfrage wichtigen Fakten erst sehr spät erfolgt.

3. *Kombination von Top-Down und Bottom-Up Verfahren*

   Es gibt verschiedene Forschungsarbeiten, die sich mit einer Kombination von Top-Down und Bottom-Up-Verfahren beschäftigen.

   Grundsätzlich kann dies auf zwei Arten geschehen:

   - *Sequentielle Phasen:*

     Vor der Ausführung eines Programms erfolgt eine Top-Down Analyse, um z.B. vor der Ausführung zusätzliche Informationen zum Programm hinzufügen zu können. Zur Ausführungszeit wird das Programm dann ausschließlich bottom-up ausgewertet.

     Das bekannteste und mittlerweile Standard gewordene Verfahren ist die in Abschnitt 2.1.4 näher erläuterte Magic-Sets Methode [Ullman 88], bei der durch Analyse der Abfrage Regeln verändert und zusätzliche Regeln in das Programm eingefügt werden können, bevor dann eine reine Bottom-Up Auswertung gestartet wird. Die Menge der Zwischenergebnisse wird bei der Bottom-Up Auswertung durch die zusätzlichen Regeln schneller eingeschränkt und somit sind insgesamt weniger Berechnungen notwendig.

   - *Nebenläufige Phasen:*

     Diese Verfahren verwenden zur Ausführungszeit sowohl Bottom-Up, als auch Top-Down Methoden.

     · Die Kernidee bei der von [Zukowski 96] entwickelten Methode der *Flexible Query Evaluation* ist, das Programm in Gruppen positiver, bzw. positiv rekursiver Literale aufzuteilen. Diese werden mit (beliebigen) Bottom-Up Verfahren ausgewertet. Negative Abhängigkeiten, die bei Bottom-Up Verfahren normalerweise Probleme bereiten, werden mit Hilfe eines Top-Down Requests zur Ausführung angestoßen. Je nach Art des Programms kann dieses daher rein bottom-up, rein top-down, oder mit einer beliebigen Kombination beider Methoden ausgewertet werden. Dieses Verfahren ist mit Erfolg im System LOLA [Zukowski 97] realisiert worden.

     · [Bader 90] arbeitet mit Graphen die ähnlich zu Dependency-Graphen sind und wertet diese prinzipiell bottom-up aus, kann aber je nach Bedarf Anforderungen (Requests) für Berechnungen top-down an einzelne Knoten schicken. Dieses Verfahren fällt damit unter den Begriff der anforderungsgetriebenen Abarbeitung.

Eine zusätzliche Klassifikation der Verfahren ist anhand der Auswertungsart möglich:

A) *Tupelorientierte Auswertung*

   Tupelorientierte Auswertungsverfahren verarbeiten jeweils eine der erzeugten Formeln oder Fakten weiter. Kann die Formel oder das Fakt nicht mehr weiter berechnet werden, wird mit der nächsten Formel oder dem nächsten Fakt weitergearbeitet.

   - *Tupelorientierte Top-Down Verfahren*

     Der am weitesten verbreitete Inferenzmechanismus der SLD-Resolution [Lloyd 87] ist Grundlage für die Auswertungsverfahren, die in vielen Logikprogrammiersprachen eingesetzt werden.

- *Tupelorientierte Bottom-Up Verfahren*

  Diese Methode eröffnet aufgrund der einfachen Handhabbarkeit der berechneten und zu berechnenden (Zwischen-)Ergebnisse eine Reihe von Erweiterungsmöglichkeiten, wie sie z.B. auch in der Arbeit von [Schütz 93] vorgestellt werden.

B) *Mengenorientierte Auswertung*

Mengenorientierte Auswertungsverfahren verarbeiten die Formeln oder die Fakten gleichzeitig.

- *Mengenorientierte Top-Down Verfahren*

  Mengenorientierte Top-Down Verfahren arbeiten ebenfalls mit SLD-Resolution. Dabei treten folgende Formen von Parallelität auf, die je nach Berechnungsmodell einzeln oder gemeinsam genutzt werden [Kacsuk 90]:

  - *Oder-Parallelität*

    Wenn mehr als eine alternative Regel zu einem Aufruf existiert, können diese parallel bearbeitet werden. In [Delgado 92a] wird das zugrunde liegende Konzept ausführlich erläutert.

    i) *Don't care non-determinism*

       Bei *Don't care non-determinism* wird immer nur eine Lösung gesucht. Eine alternative Regel wird daher nicht mehr ausgeführt, sobald eine Lösung gefunden wurde. Somit werden bei *OR-parallelism* mit *don't care non-determinism* nur alternative Regeln parallel gesucht, sie werden jedoch sequentiell zueinander ausgeführt und die Ausführung stoppt bei der ersten erfolgreichen Lösung.

    ii) *Don't know non-determinism*

       Bei *Don't know non-determinism* werden alle möglichen Lösungen gesucht. Folglich werden bei *OR-parallelism* mit *don't know non-determinism* alternative Regeln nicht nur parallel gesucht, sondern auch parallel ausgeführt.

  - *Und-Parallelität*

    Die Literale im Rumpf einer Regel können parallel zueinander ausgeführt werden. Es gibt verschiedene Formen von Und-Parallelität. Die Konzepte werden in [Delgado 92b, Delgado 92c] ausführlich beschrieben.

    i) *restricted AND-parallelism*

       Verschiedene Literale in der Abfrage oder dem Rumpf einer Regel, die keine gemeinsamen Variablen miteinander teilen, lassen sich parallel ausführen.

    ii) *stream AND-parallelism*

       Werden Werte z.B. in einer Liste von einem Literal an ein anderes Literal übergeben, so kann die Verarbeitung im zweiten Literal beginnen, sobald der erste Wert verfügbar ist. Es muß nicht erst auf die vollständige Liste gewartet werden.

    iii) *all-solutions AND-parallelism*

       Der Unterschied zu den anderen Formen an Parallelität besteht darin, daß nicht an einer Lösung parallel gearbeitet wird, sondern daß die Literale

im Rumpf einer Regel derart mehrfach aufgerufen werden, so daß sie an verschiedenen Lösungen parallel arbeiten. Hierbei ist jedoch die Gefahr zu vieler redundanter Berechnungen gegeben.

- *Mengenorientierte Bottom-Up Verfahren*

  Diese Methode wird sehr häufig in deduktiven Datenbanken angewendet. Hierunter fallen Auswertungsmethoden wie naive Auswertung und semi-naive Auswertung.

2.1.4 Verfahren

Das folgende Kapitel gibt einen Überblick über wichtige Begriffe und Verfahren aus dem Bereich der deduktiven Datenbanken.

Verwendete Begriffe sind:

1. *Dependency Graph, Operatorbaum*

   Dependency Graphen [Ullman 88] sind Graphen, die die Abhängigkeiten eines Logikprogramms darstellen.

   Jedes Prädikat des Programms wird durch einen Knoten dargestellt. Von einem Knoten für das Prädikat p geht eine Kante zu einem Knoten für das Prädikat q, wenn es eine Regel gibt, bei der p im Kopf vorkommt und q im Rumpf. Gibt es Regelköpfe mit gleichen Prädikaten, so werden diese mit einen zusätzlichen Knoten verbunden [Cremers 94].

   Bottom-Up Verfahren können anhand eines Dependency Graphen durchgeführt werden.

   Ähnliche Graphen sind *Rule/Goal-Graphen* [Ullman 85] oder *AND/OR-Graphen* [Emden 76].

   Ein *Operatorbaum* ist im Unterschied zu einem Dependency Graphen eine Darstellung der Abhängigkeiten des Logikprogramms, wobei die Knoten, die die Prädikate repräsentieren, keine semantische Bedeutung besitzen.

2. *(Bedingte) Sicherheit*

   Systemprädikate (z.B. auch $<, \leq, \neq$) können bei Bottom-Up Auswertungen nur eingeschränkt verwendet werden, da bei reinen Bottom-Up Verfahren bei entsprechenden Wertebreichen (Domains) für die jeweiligen Parameter unendliche Wertemengen entstehen können.

   Regeln, die mit Bottom-Up Verfahren ausgewertet werden sollen, müssen daher verschiedenen Sicherheitsbedingungen genügen:

   - Eine Variable, die im Kopf einer Regel vorkommt, muß in einem positiven Atom im Rumpf vorkommen.
   - Eine Variable, die in einem Systemprädikat vorkommt, muß in einem positiven Atom im Rumpf vorkommen.
   - Eine Variable, die in einem negativen Atom im Rumpf einer Regel vorkommt, muß auch in einem positiven Atom im Rumpf vorkommen.

Für die Definition bedingter Sicherheit genügt es, daß eine Variable in einem positiven Atom im Rumpf oder im Kopf der Regel vorkommt.

Programme, die diesen Bedingungen genügen, nennt man sicher, bzw. bedingt sicher. Eine genaue Definition (bedingter) Sicherheit ist in Anhang A.2 nachzulesen.

3. *Stratifizierung*

   Tritt eine Negation in einer Rekursionsschleife auf, so kann dies zu mehrfachen, aber unvereinbaren Ergebnissen führen, d.h. das Programm hat verschiedene, minimale Herbrand-Modelle [Cremers 94].

   Soll eine solche Situation ausgeschlossen werden, darf in einer Rekursionsschleife keine Negation auftreten. Programme bei denen dies gewährleistet ist, nennt man *stratifiziert*. Eine genaue Definition der Stratifizierung ist in Anhang A.2 nachzulesen.

Verfahren, zu denen im Rahmen der Arbeit ein Bezug hergestellt wird, sind:

1. *Sideways Information Passing*

   Bei einer Top-Down Betrachtungsweise lassen sich über die in den Literalen einer Regel vorkommenden Argumente Aussagen treffen, ob sie bei einer Anwendung des Literals an Werte gebunden sind oder ob sie frei sind. Dafür wird z.B. der Dependency Graph im Tiefensuchverfahren durchlaufen und es werden alle Stellen registriert, an denen Werte für Parameter erzeugt werden. Auf dem Pfad vor diesen Stellen sind sie frei, auf dem Pfad danach gebunden.

   Für viele Argumente innerhalb einer Regel lassen sich bereits bei alleiniger Betrachtung der Regel schon Aussagen treffen. Da die Untersuchung innerhalb einer Regel horizontal vorgeht, nennt sich diese Methode *Sideways Information Passing Strategie* (*SIP-Strategie*).

2. *Bestimmung von Bindungsmustern*

   Die Bestimmung von Bindungsmustern, d.h. von Modi (Ein- oder Ausgabemodus) für Parameter kann auf zwei Arten erfolgen: Entweder die Modi werden vom Programmierer vorgegeben, unterstützt durch eine entsprechende Programmiersprache, oder sie werden automatisch mit Hilfe einer gegebenen Abfrage in einer Top-Down Analyse durch die Beobachtung der Entstehung von Bindungen mit Hilfe der SIP-Strategie abgeleitet [Cremers 94].

3. *Rektifizierung von Literalen*

   Ein Literal ist rektifiziert, wenn jedes Argument eine Variable ist, und eine Variable in der Parameterliste höchstens einmal vorkommt.

   Um einen Regelkopf oder die Literale im Rumpf einer Regel zu rektifizieren, werden die Parameter eines Literals durch neue, eindeutige Variablen ersetzt und zusätzliche, zumeist triviale Constraints eingeführt, die die bisherigen Parameter mit den jeweils entsprechenden Variablen unifizieren [Ullman 88].

   Die Rektifizierung von Regeln benötigt man z.B. um eine einfache Verarbeitbarkeit von Regeln in relationaler Algebra zu erhalten.

4. *Duplizierung von Regeln*

   Die z.B. mit der SIP-Strategie erzeugten Bindungsmuster für ein Literal müssen nicht eindeutig sein. Benötigt man Regeln mit eindeutigen Bindungsmustern, so müssen diese zunächst rektifiziert werden. Dann können sie vervielfältigt werden, wobei jede Regelalternative ein eindeutiges Bindungsmuster zugewiesen erhält.

5. *Reihenfolgeoptimierung der Rumpfliterale*

   In Logiksprachen, die mit SLD-Resolution arbeiten, wird als Abarbeitungsreihenfolge die Aufschreibungsreihenfolge der Regeln und innerhalb einer Regel die Aufschreibungsreihenfolge der Rumpfliterale verwendet.

   In deduktiven Datenbanken, die meist mit Bottom-Up Verfahren arbeiten, gibt es verschiedene Optimierungsverfahren, die an Regeln hinsichtlich der Reihenfolge der Rumpfliterale Veränderungen vornehmen, um kostengünstigere Abarbeitungen zu erhalten. Da eine günstige Auswertungsreihenfolge der Rumpfliterale im allgemeinen von den Bindungen des Kopfliterals einer Klausel abhängt, legt z.B. die SIP-Strategie fest, in welcher Reihenfolge die Rumpfliterale einer Regel ausgewertet werden sollen. Nachdem die Bestimmung der Parametermodi erfolgt ist, können die Literale im Rumpf umgestellt werden und zwar derart, daß die Literale, die Variablen binden, nach vorne gestellt werden, Literale, die keine Variablen binden, dagegen nach hinten.

6. *Unfolding und Folding von Regeln*

   Unfolding bedeutet das Einsetzen eines Regelrumpfs anstelle eines Literals, wenn das Literal sich mit dem Kopf der Regel unifizieren läßt, *Folding* das Zusammenfassen einiger Literale im Rumpf einer Regel zu einer neuen Regel und das Einsetzen des Regelkopfs an ihrer Stelle [Tamaki 84, Kanamori 87, Gardner 91].

   Die Arbeiten von [Seki 93, Aravindan 95] beschäftigen sich mit der Korrektheit der Unfold/Fold-Transformation.

7. *Magic Sets Methode*

   Die *Magic Sets Methode* ist eine Methode bei der das Bottom-Up Verfahren dahingehend optimiert wird, daß die Einschränkung auf die für die gestellte Abfrage wichtigen Fakten bereits zu einen sehr frühen Zeitpunkt erfolgt. Dafür ist es nötig, daß das Programm zur Übersetzungszeit mit einer Top-Down Analyse untersucht wird und neue, zusätzliche Regeln generiert, sowie vorhandene Regeln ergänzt werden [Ramakrishnan 88, Ramakrishnan 90, Ullman 88]. Diese Methode fällt also in den Bereich der *Rewriting Techniken*.

   Die Transformationen müssen für jede Abfrage, bzw. jedes Abfragemuster neu generiert werden. Ferner wächst die Größe des Programms stark: die Zahl der zusätzlich erzeugten Regeln kann im schlechtesten Fall quadratisch zur Anzahl der in Regelrümpfen auftretenden Literale sein. Zusätzlich gilt, daß ein stratifiziertes Programm nach der Transformation nicht mehr stratifiziert sein muß [Cremers 94], es können nun also Rekursionen auftreten, die Negationen enthalten. Damit ist die Semantik des Programms nicht mehr einfach bestimmbar.

   Die Magic Sets Methode ist außerdem ein Verfahren, mit der nicht sichere Regeln zu sicheren Regeln transformiert werden. In jeder Regel werden Literale derart eingefügt, so daß alle benötigten Bindungen an Werte lokal im Regelrumpf geschehen können und dadurch eine reine Bottom-Up Auswertung auch mit Systemprädikaten möglich wird.

8. *Extraktion von Funktionen und Operationen in Parameterlisten*

   Um Funktionen in Parameterlisten behandeln zu können, wird in [Bader 90] ein Verfahren vorgestellt, mit dem Funktionen aus der Parameterliste extrahiert und in einem eigenen Rumpfliteral behandelt werden. Es kann, je nach Bindungsmuster, auch der Fall eintreten, daß nicht die Funktion selbst, sondern die dazugehörige Umkehrfunktion berechnet werden muß. Dabei muß jedoch berücksichtigt werden, daß dies nur auf einer eingeschränkten Menge von Funktionen (d.h. Funktionen, für die jeweils eine Umkehrfunktion existiert) und einer eingeschränkten Menge von Werten (d.h. Werte, für die die Umkehrfunktion ein eindeutiges Ergebnis liefert) für die Parameter gelingen kann.

9. *Selection Pushing*

   Die Magic Sets Methode versucht so früh, d.h. so weit unten wie möglich, die Mengen an Fakten auf die für die Abfrage relevanten Fakten einzuschränken. Dabei ist es jedoch nicht möglich, Constraints in der Abfrage oder sehr weit "oben" im Programm, ebenfalls so bald wie möglich zu berücksichtigen. Mit Hilfe von *Selection Pushing* können Constraints von "oben" nach "unten" durchgereicht werden, um früher die Faktenmengen einzuschränken. Die Methoden, Magic Sets und Selection Pushing, können gut miteinander kombiniert werden [Kemp 89, Srivastava 93].

2.1.5 Sprachen

Als Hochsprachen, basierend auf dem Konzept der Logikprogrammierung, sind vor allem Prolog und Datalog bekannt. Jede der beiden Sprachen hat ihr eigenes Anwendungsgebiet, das auch durch die Ausführungsmethode gekennzeichnet ist:

- Mit Prolog werden wissensbasierte Systeme implementiert. Die Ausführung basiert auf SLD-Resolution und ist damit top-down.
- Mit Datalog werden hauptsächlich datenbankorientierte Anwendungen realisiert. Die Ausführung ist grundsätzlich bottom-up.

2.1.5.1 PROLOG

Prolog ist eine weit verbreitete Hochsprache aus dem Bereich der Logikprogrammierung [Sterling 86].

Zusätzlich zu den aus der Logikprogrammierung vorgegebenen Elementen ist sie um eine umfangreiche Bibliothek, auf deren Funktionen durch Systemprädikate zugegriffen werden kann, und das Konzept von Kontrollstrukturen erweitert.

Die bekannteste Kontrollstruktur in Prolog ist der Cut. Mit seiner Hilfe kann ein Backtracking unterbunden werden, wenn der Programmierer von vornherein weiß, daß ohnehin keine weitere Lösung möglich ist.

Die semantische Bedeutung eines Progamms kann sich durch den Cut ändern, da sich mit dem Cut Regeln der Form:

Wenn Bedingung dann Regel_1 ansonsten Regel_2

(anstelle einer Regel: *Wenn Bedingung_1 dann Regel_1, wenn Bedingung_2 dann Regel_2* realisieren lassen.

Man unterscheidet daher zwei Formen des Cuts [Sterling 86]:

- *grüner Cut*
  Ein Cut wird als grüner Cut bezeichnet, wenn das Programm nach Weglassen des Cuts semantisch äquivalent zu dem Programm mit Cut ist.
- *roter Cut*
  Ein Cut wird als roter Cut bezeichnet, wenn das Programm nach Weglassen des Cuts nicht semantisch äquivalent zu dem Programm mit Cut ist.

Es gibt außerdem Kontrollstrukturen, die durch Metaprädikate realisiert werden, wie beispielsweise assert, um neue Fakten an die Datenbank anzuhängen, oder retract, zum Entfernen von Fakten aus der Datenbank.

Die Hauptziele paralleler PROLOG-Implementierungen sind einerseits eine Effizienzsteigerung und andererseits die Vollständigkeit der Berechnung, d.h. das Auffinden jeder möglichen Lösung. Diese Ziele sind widersprüchlich und haben zu einer Vielzahl von verschiedenen parallelen Implementierungen von Logikinterpretern geführt [Yang 87], z.B. PARLOG [Clark 84, Gregory 87], Concurrent PROLOG [Shapiro 83, Shapiro 86] und Guarded Horn Clauses [Ueda 86]. Sie basieren wie PROLOG auf dem Auswertungsprinzip der SLD-Resolution, nutzen jedoch, je nach Implementierung, Und- und Oder-Parallelität.

2.1.5.2 DATALOG

Grundsätzlich wird als Sprache für die Erstellung deduktiver Datenbanken nicht PROLOG sondern DATALOG, bzw. DATALOG$^{\neg}$ verwendet. Dabei handelt es sich um eine logische Datenbanksprache, die in ihrer Ausdrucksform und Mächtigkeit gegenüber der Logikprogrammierung eingeschränkt ist [Ullman 88]:

- Funktionen als Argumente von Literalen sind nicht zugelassen.
- Systemprädikate beschränken sich weitestgehend auf Relationszeichen (wie z.B. $=, <, \leq, ...$).
- Metaprädikate wie der Cut oder fail kommen nicht vor.
- Die Negation ist nur in DATALOG$^{\neg}$ zugelassen.

Diese Einschränkungen beruhen auf der Tatsache, daß Bottom-Up Auswertungswertungsverfahren bei deduktiven Datenbanken aus Effizienzgründen der in PROLOG üblichen SLD-Resolution vorgezogen werden. Bei Verwendung von Systemprädikaten oder Funktionen und Operationen können bei Bottom-Up Verfahren jedoch Probleme aufgrund der Möglichkeit der Erzeugung unendlicher Datenmengen enstehen [Ullman 88].

Bei Datenbankanwendungen geht es jedoch im wesentlichen um die Verwaltung großer Datenbestände. Hierfür sind Funktionen und Systemprädikate zur Manipulation von Daten nicht zwingend erforderlich. Damit ist durch die Verwendung von DATALOG bezogen auf das Anwendungsgebiet kaum eine Einschränkung gegeben.

2.2 Datenflußrechner

Mehrprozessorsysteme erschließen aufgrund ihrer hohen Leistungsfähigkeit immer neue Anwendungsgebiete. Eine Voraussetzung für den Einsatz von Mehrprozessorsystemen ist jedoch die Parallelisierbarkeit eines Problems, d.h. die Aufteilung in Teilprobleme, die verschiedene Prozessoren gleichzeitig bearbeiten können. Da entsprechend dem Kontrollfluß die Datenverfügbarkeit gewährleistet sein muß, ist eine Synchronisation von Kontroll- und Datenfluß zwingend notwendig. Dies bedeutet einen hohen Verwaltungsaufwand. Daher lassen sich (von Neumann-)Mehrprozessorsysteme im allgemeinen nur dann effizient einsetzen, wenn jedes der Teilprobleme eine relativ lange Ausführungszeit benötigt.

Datenflußrechner lösen das Problem der Synchronisation durch die Vereinheitlichung von Kontroll- und Datenfluß: beim Datenflußprinzip wird die Befehlsausführung allein durch die Verfügbarkeit der Operanden des Befehls ausgelöst. Die "Maschinensprache" ist dementsprechend ein Graph, dessen Knoten Befehle, bzw. Operationen darstellen, während Kanten Datenabhängigkeiten repräsentieren.

2.2.1 Datenflußgraphen

Ein *Datenflußgraph* ist ein Graph, dessen Knoten über gerichtete Kanten miteinander verbunden sind. Die Knoten repräsentieren dabei die Operationen, die Kanten die Datenabhängigkeiten zwischen den Operationen. Von Operation zu Operation werden die Operanden entlang der Kanten in Form von *Token* weitergegeben. Abbildung 2 veranschaulicht das Prinzip.

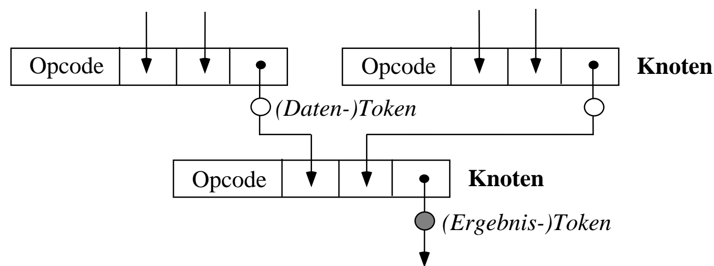


Abbildung 2: Datenflußprinzip

Die Ausführung eines Datenflußmodells basiert auf:

- dem Kontrollmechanismus und

- dem Datenmechanismus.

Der Kontrollmechanismus des Datenflußmodells definiert einen Knoten des Datenflußgraphen genau dann als ausführbar, wenn alle Operanden verfügbar sind. Das bedeutet, daß ein Knoten zur Ausführung seiner Operation bereit ist, sobald auf allen Eingangskanten des Knotens (Daten-)Token vorhanden sind. Die Befehlsausführung geschieht daher datengesteuert. Es ist keine zentrale Kontrollinstanz mehr für die Befehlsausführung notwendig [Veen 86].

Der Datenmechanismus ist dadurch definiert, daß nach Ausführung eines Befehls alle Operanden von den Eingangskanten entfernt werden und das Ergebnis der Operationsausführung auf die Ausgangskanten gelegt wird. Damit wird es den nachfolgenden Knoten automatisch übermittelt. Durch Vervielfältigung zu unabhängigen Kopien kann ein Datum auch mehreren Knoten gleichzeitig zur Verfügung gestellt werden [Müller 96].

Knoten, die nicht direkt über eine Kante miteinander verbunden sind, sind voneinander datenunabhängig und können, vorausgesetzt, alle Operanden sind verfügbar, parallel zueinander oder in beliebiger Reihenfolge ausgeführt werden. Dies ermöglicht eine feinkörnige und dadurch stark ausgeprägte Ausführungsparallelität, die auf Ebene einzelner Operationen möglich ist. Der Graph kann somit leicht auf einem Rechensystem implementiert werden, das aus einer Vielzahl einzelner Prozessoreinheiten aufgebaut ist [Ungerer 93]. Ferner müssen parallele Operationen nicht vom Programmierer als solche spezifiziert werden, oder mit Hilfe aufwendiger Analyseverfahren gefunden werden.

Es gibt jedoch bestimmte Programmkonstrukte, wie Verzweigungen oder Schleifen, bei denen eine Ablaufsteuerung, die nur über den Fluß von Daten stattfindet, nicht ausreichend ist. Hier muß zusätzlich ein Triggermechanismus eingeführt werden [Veen 86]. Da im gleichen Teilgraphen auf unterschiedlichen Datentokensätzen gearbeitet wird, ist die Realisierung von Schleifen, wie auch von Prozeduren aufwendig [Treleaven 82b].

In jeder Rechnerarchitektur geschieht die Programmausführung durch die repetierende Anwendung der Folge der drei Phasen [Treleaven 82a]:

1. *Auswahlphase*
   In der Auswahlphase bestimmt die *Computation Rule* welche Befehle im nächsten Schritt zur Berechnung bereitgestellt werden. Im Datenflußprinzip betrifft dies im allgemeinen alle Befehle.

2. *Überprüfungsphase*
   In der Überprüfungsphase wird durch die *Firing Rule* überprüft, welche der bereitgestellten Befehle tatsächlich ausführbar sind. Im Datenflußprinzip sind dies alle Befehle, deren Operanden vollständig verfügbar sind.

3. *Aufführungsphase*
   Die *Result Distribution Rule* übernimmt in der Ausführungsphase die tatsächliche Ausführung des Befehls und die Weitergabe der Ergebnisse.

2.2.2 Datenflußrechnerarchitekturen

Die Realisierung von Schleifeniterationen und Prozedurinstanzen führte zur Entwicklung verschiedener Datenflußrechnerarchitekturen [Arvind 91].

Vereinfacht ausgedrückt läßt ein *statischer Datenflußrechner* innerhalb einer Schleife oder einer Prozedur nur Token ein und derselben Iterationstiefe zu. Eine Zuordnung von Token als Eingaben für Operationen geschieht über direkte Adressierung.

Bei der *dynamischen Datenflußrechnerarchitektur* können sich dagegen innerhalb einer Schleife oder einer Prozedur Token aus beliebigen Iterationstiefen aufhalten. Sie müssen jedoch mit einer eindeutigen Markierung versehen sein, einem sogenannten *Tag*, um sicherzustellen, daß Operationen nur auf Token der gleichen Iterationstiefe angewendet werden. Nachteilig wirkt sich hier jedoch das aufwendige, gegenseitige Zuordnen (Matching) von Token anhand ihrer Tags aus.

Bei der *hybriden Datenflußrechnerarchitektur* wird das Problem des Tokenmatchings gelöst, indem Tags nicht mehr Kennzeichnungen darstellen, sondern Adressen. Token

mit gleichem Tag gelangen so in einen strukturierten Adreßbereich und können einander leicht zugeordnet werden.

2.2.2.1 Statische Datenflußrechnerarchitekturen

Jeder Knoten in einem Datenflußgraphen stellt einen Befehl dar. Da ein Knoten außer der Bezeichnung seiner Operation Speicherplatz für die Operanden enthalten muß und zur Ausführung bereitsteht, sobald seine Operandenplätze mit gültigen Werten belegt sind, wird er als *Template* oder *Activity Template* bezeichnet.

In einem Template muß Platz sein für den Operationscode, die Operandenwerte und die Zieladressen. Eine Zieladresse besteht aus Zieltemplate und Operandenplatz. Ein Token, das Ergebnisse von einem Template zu anderen Zieltemplates transportiert, enthält einen Wert und die Zieladresse.

Eine statische Rechnerarchitektur besteht, in ihren Grundzügen dargestellt, aus einem *Activity Store*, der *Instruction Queue*, der *Fetch Unit*, der *Operation Unit* und der *Update Unit* [Dennis 80].

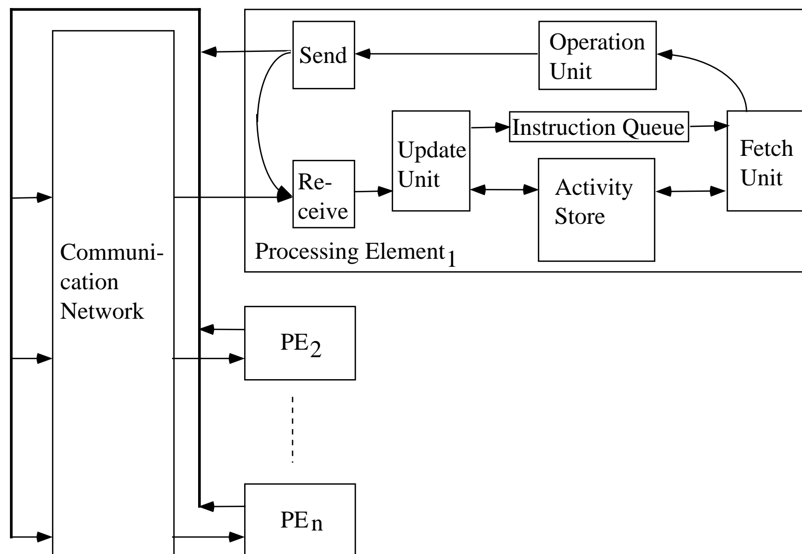


Abbildung 3: Architektur eines statischen Datenflußrechners

Der *Activity Store* enthält alle zur Berechnung bereitstehenden Befehle, also alle *Activity Templates*. Jedes Activity Template besitzt eine eindeutige Adresse, die in die *Instruction Queue* eingetragen wird, wenn das Activity Template aufgrund verfügbarer Operandenwerte durch die Firing Rule zur Berechnung freigegeben wird. Die *Fetch Unit* nimmt sich aus der Instruction Queue eine Adresse, liest das zugehörige Activity Template aus dem Activity Store aus, bildet daraus ein *Operation Packet* und gibt dieses an die *Operation Unit* weiter. Die Operation Unit berechnet mit Hilfe des Operationscodes aus den Operanden das Ergebnis und generiert ein *Result Packet*, also ein Token mit den Resultaten, für jede Zieladresse. Die *Update Unit* schließlich erhält diese Result Packets und trägt die enthaltenen Werte in die durch die Zieladresse spezifizierten Felder der Templates ein. Außerdem überprüft die Update Unit bei dieser Gelegenheit, ob

die Operanden des Zieltemplates nun vollständig sind. Ist dies der Fall, nimmt sie die Adresse des Templates und trägt diese in die Instruction Queue ein.

Diese Grundstruktur läßt sich ohne Probleme auf ein Mehrprozessorsystem übertragen. Der Datenflußgraph wird aufgeteilt und auf die verschiedenen Prozessorelemente verteilt. Die einzelnen Activity Stores der Prozessorelemente werden zusammen wie ein großer Speicher linear addressiert, so daß ein Activity Template im ganzen System mit einer eindeutigen Adresse gefunden werden kann. Jedes Prozessorelement sendet seine Result Packets durch ein sogenanntes *Store & Forward Network*, wenn die Zieladresse des Result Packets ein nicht lokales Template spezifiziert. Andernfalls wird es zu seiner eigener Update Unit gesendet.

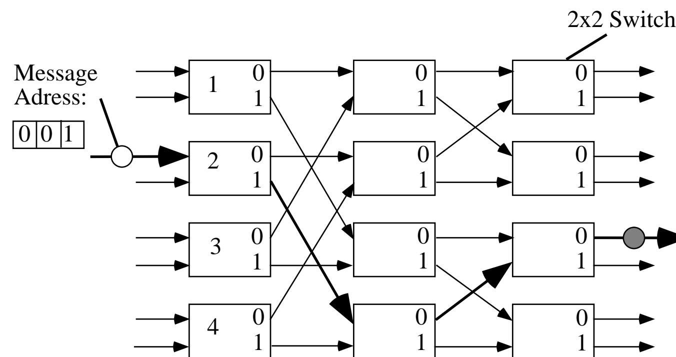


Abbildung 4: 8x8 Packet Switching Network

Das Communication Network muß aufgrund der Struktur der Rechnerarchitektur die Result Packets nur in einer einzigen Richtung durchleiten. Außerdem können Verzögerungen beim Verteilen in Kauf genommen werden, da jedes Prozessorelement ohnehin mehrere zur Ausführung bereite Templates enthält. Aus diesem Grund verwendet man meist ein *Packet Switching Network*, also ein Netz, das aus Routern aufgebaut ist, die nur in einer Richtung durchleiten.

Der Vorteil statischer Datenflußrechnerarchitekturen ist, daß ihre Struktur einfach ist. Token können ohne Zeitverlust miteinander gematcht werden. Der Nachteil statischer Datenflußrechnerarchitekturen ist jedoch, daß die Parallelisierung von Schleifeniterationen oder Prozedurinstanzen nicht möglich ist, da sich immer nur der Tokensatz einer Schleifeniteration in der Schleife befinden darf.

Fazit: Statische Datenflußrechnerarchitekturen sind schnell, jedoch nicht für alle Arten von Anwendungen optimal geeignet.

2.2.2.2 Dynamische Datenflußrechnerarchitekturen

Dynamische Datenflußrechner lassen Token aus verschiedenen Schleifeniterationen zu. Dabei wird die Zusammengehörigkeit von Token durch ein Tag als Färbung sichtbar gemacht.

Das grundlegende Prinzip von *Tagged Tokens* wurde von [Arvind 82] entwickelt und mit dem *U-Interpreter (Unfolding Interpreter)* realisiert. Es ermöglicht die parallele Ausführung von Prozeduren und Schleifeniterationen, ohne den Code der Prozedur

oder Schleife kopieren zu müssen. Statt dessen werden mehrere Token, die zu verschiedenen Prozeduraufrufen oder Schleifeniterationen gehören, gleichzeitig auf den Kanten des entsprechenden Teilgraphen zugelassen. Um die Token der gleichen Iterationstiefe miteinander verarbeiten zu können, müssen sie jedoch mit einer eindeutigen Markierung versehen sein, einem sogenannten *Tag*, um sicherzustellen, daß Operationen nur auf Token der gleichen Iterationstiefe angewendet werden. Ein Tag ist systemweit eindeutig.

Die Struktur eines Verarbeitungselements eines dynamischen Datenflußrechners, der *Manchester Dataflow Machine*, ist in Abbildung 5 dargestellt.

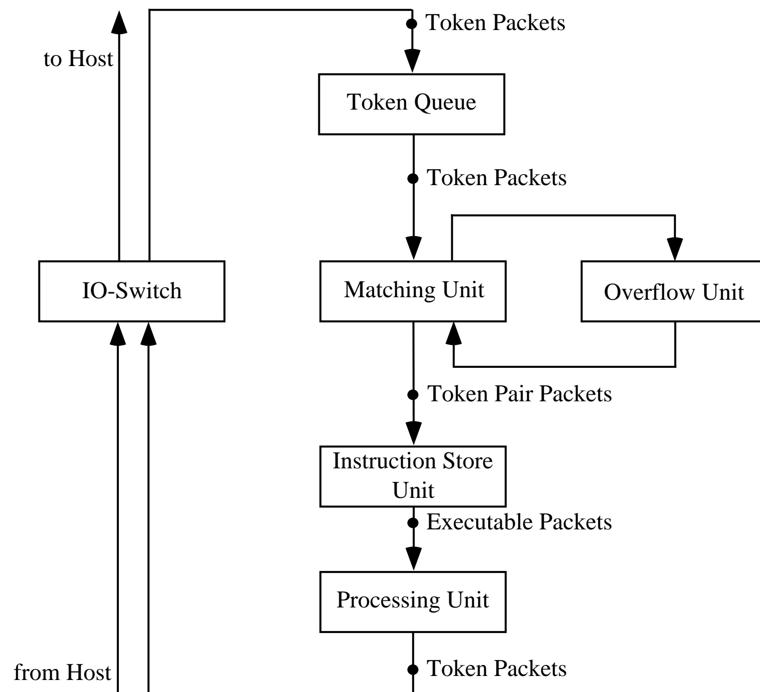


Abbildung 5: Organisation eines dynamischen Datenflußrechners (Manchester)

Das Verarbeitungselement besteht aus einer Pipline aus folgenden Elementen:

- einer *Token Queue* für ankommende Token,
- einer *Matching Unit*, die Token miteinander kombiniert, so daß *Token Pair Packets* entstehen (dabei wird davon ausgegangen, daß eine Operation maximal zwei Operanden benötigt),
- einer *Instruction Store Unit*, die den durch die Token referenzierten Befehl aus dem Programmspeicher holt und zusammen mit den Token zu einem *Executable Packet* zusammenfügt, und
- einer *Processing Unit*, der Verarbeitungseinheit, die als Ergebnis wiederum *Token Packets* generiert.

Der *Matching Unit* ist eine *Overflow Unit* zur Seite gestellt, die Token aufnehmen kann, wenn für diese in der *Matching Unit* kein Platz mehr ist.

Über den *IO-Switch* werden *Token Packets* an andere Verarbeitungseinheiten gesendet oder von ihnen empfangen.

Ein Token besteht aus

- einem *Activation Name* für den aktuellen Kontext, z.B. eine aktuelle Prozedurinstanz,
- einem *Iteration Level* für die Angabe einer Interationstiefe bei Schleifen und
- einem *Index* für den Zugriff auf Elemente komplexer Datenstrukturen.

Da die Anzahl von Token jedoch sehr groß werden kann, ist das *Token-Matching*, d.h. das Finden zusammengehörender Token, ein aufwendiger Suchvorgang und weitaus zeitintensiver als bei statischen Datenflußrechnern. Das Problem dynamischer Datenflußrechner besteht daher in der Implementierung einer effizienten Vergleichseinheit für die Tags ankommender Token mit den Tags der im Speicher bereits vorhandenen Token zur Bildung neuer Operandensätze.

Fazit: Dynamische Datenflußrechnerarchitekturen ermöglichen eine hohe Ausbeutung von Parallelität, nehmen jedoch einen deutlichen Effizienzverlust in Kauf.

2.2.2.3 Hybride Datenflußrechnerarchitekturen

Um das Problem dynamischer Datenflußrechner bezüglich des Token-Matchings über Adreßbezüge zu lösen, wurde das Prinzip des *Explicite Token Stores* entwickelt.

Der Speicher ist zweigeteilt: der Befehlspeicher enthält die Codeblöcke, der Rahmenspeicher speichert die Aktivierungsrahmen (*Activation Frames*) und die Konstantenbereiche. Zu jedem Codeblock, d.h. zu jeder Schleifeniteration und zu jeder Prozedurinstanz gibt es einen eigenen Aktivierungsrahmen im Rahmenspeicher. Um ein übermäßiges Anwachsen der Menge der Aktivierungsrahmen zu verhindern wurde dabei das k-begrenzte Schleifenschema eingeführt. Dabei werden nur für k Iterationen Aktivierungsrahmen erzeugt, so daß nur k Knoten parallel ausgeführt werden können. Die Tags der Token stellen nun keine Färbung der Token mehr da, sondern Adressen. Die Adressierung geschieht mittels einer Offset-Adresse bezüglich der Basisadresse des jeweiligen Aktivierungsrahmens [Culler 90, Papadopoulos 90].

Diese Konzepte wurden im *Monsoon* Rechner realisiert. Die grundlegende Struktur eines Verarbeitungselements des Monsoon Rechners ist in Abbildung 6 dargestellt.

Der Explicite Token Store enthält drei Bereiche:

- einen Bereich für Befehle (Knoten),
- Activation Frames für die durch Token übermittelten Werte und
- einen Bereich für Konstanten.

Jedem Codeblock sind zur Laufzeit eine Anzahl von Aktivierungsrahmen und ein Konstantenbereich zugeordnet. Ein Befehl besteht aus dem Operationscode, einem Offset bezüglich eines Aktivierungsrahmens oder eines Konstantenbereichs und einer Liste von Zielverweisen.

Jeder Aktivierungsrahmen besteht aus Speicherzellen zur Aufnahme von Operanden-Tokens, die auf die Ankunft des Partnertokens waren müssen.

Die grundlegende Struktur der Verarbeitungselemente besteht aus [Ungerer 93]:

- Einer Befehlsbereitstellungeinheit (*Instruction Fetch Unit*), die jeweils einen Befehl zur Verarbeitung bereitstellt.
- Einer Vergleichseinheit (*Operand Matching*), die die für eine Operation nötigen Daten im Speicher ablegt, wenn noch nicht alle benötigten Operanden vorhanden sind, oder sie aus dem Speicher holt, wenn sie vollständig sind.

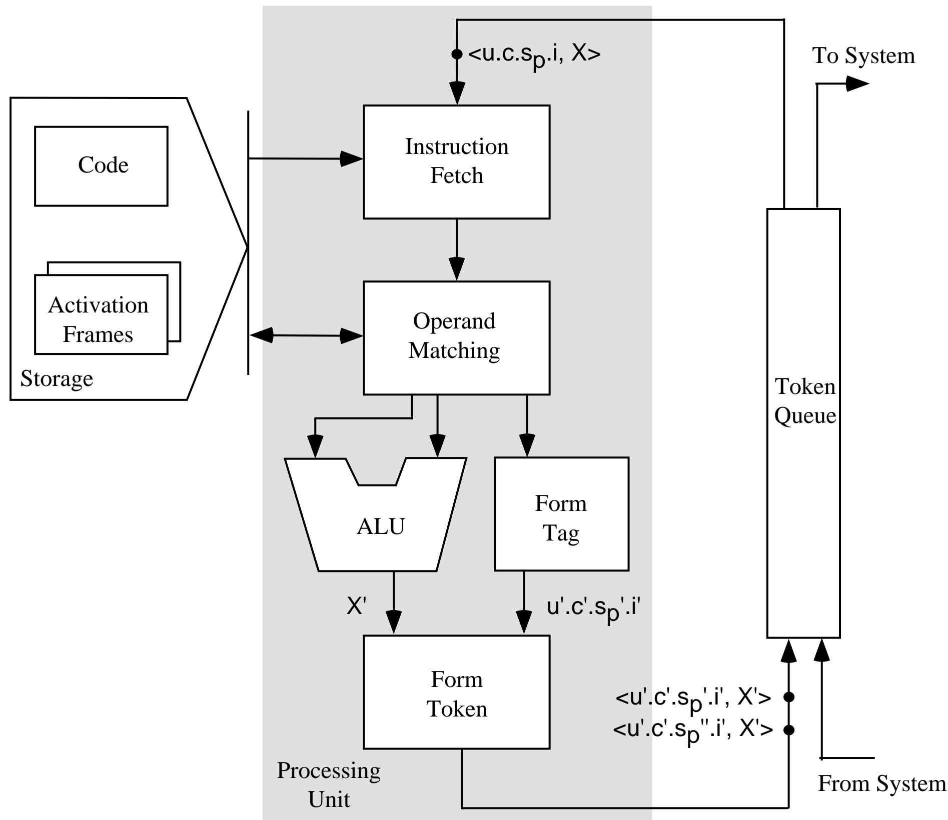


Abbildung 6: Rechnerorganisation von Monsoon

- Einer Verarbeitungseinheit (ALU), die den Befehl mit den dazugehörenden Operanden ausführt.
- einer *Form-Tag* Einheit, die die korrekten Zieladressen erzeugt.
- einer *Form-Token* Einheit, die für die Resultattoken die korrekten Tags erzeugt und verschickt.

Die Speicherelemente ermöglichen die Behandlung komplexer, strukturierter Datenobjekte durch das Prinzip der *I-Structures* [Gaudiot 86]. Mit dem *I-Structure Memory*, bestehend aus einem Speichermodul und einem Speicher-Controller, läßt sich das Problem der Speicherung und des Zugriffs auf große, strukturierte Datenmengen lösen.

Fazit: Hybride Datenflußrechnerarchitekturen vereinen die Vorteile statischer und dynamischer Datenflußrechnerarchitekturen.

2.2.3 Projekte

Datenflußrechner und ihre Implementierungen blicken auf eine lange Reihe von Forschungsarbeiten zurück.

Im Massachusetts Institute of Technology leitete J. Dennis [Dennis 75, Dennis 80] ein Projekt, das den *MIT Static Dataflow Computer* als Ergebnis hatte.

An der University of California in Irvine wurde sowohl die Datenflußsprache *Id*, als auch die erste dynamische Datenflußrechnerarchitektur, die *Irvine Dataflow Machine*,

von Arvind und Gostelow entwickelt [Arvind 78]. Arvind und Gostelow [Arvind 78, Arvind 81, Arvind 83] entwickelten später am MIT den *MIT Tagged-Token Dataflow Computer*, mit dem zusammen der U-Interpreter und das k-begrenzte Schleifenschema eingeführt wurden. Das Projekt wurde von Culler und Papadopulus unter dem Namen *Monsoon* weitergeführt, einer Datenflußrechnerarchitektur, bei dem der Explicite Token Store implementiert wurde [Papadopoulos 90].

An der University of Manchester wurde der *Manchester Dataflow Computer* gebaut [Gurd 80]. In Japan entwickelte man im Elektroniklabor von Tsukaba die SIGMA-1 [Hiraki 91, Shimada 86].

Am hiesigen Lehrstuhl von Prof. Dr. W. Hahn erfolgte der Entwurf des *Munich Simulation Computers (MuSiC)*, einem Ereignisflußrechner [Hahn 85, Hahn 92].

Kommerziell führten die Ideen dieser Systemarchitekturen zur Entwicklung von Digitalen Signal-Prozessoren (DSPs).

In [Papadopoulos 91] kann ein ausführlicher Überblick über die existierenden Datenflußrechnerarchitekturen gefunden werden.

2.3 Problemstellung

Die Grundlage für diese Arbeit bilden wissensbasierte Systeme, die mittels Logikprogrammierung spezifiziert sind.

Klassische Anwendungen auf diesem Gebiet dienen jedoch fast ausschließlich der Beschreibung von statischen Zusammenhängen. Im Gegensatz dazu beschäftigt sich die vorliegende Arbeit jedoch mit einer dynamischen Situationen die charakterisiert wird durch die Veränderung von Informationen, d.h. von Werten, die über die Basisfakten in das System eingegeben werden. Die Abfrage, bzw. die Abfragen an das Programm, sowie die Menge der Fakten und ihre Bedeutung, d.h. die durch die Fakten spezifizierte Relationen, bleiben während der gesamten Lebenszeit des Systems konstant.

Logikprogrammiersprachen mit einem Top-Down Verfahren wie PROLOG, die eine Anfrage z.B. mittels SLD-Resolution unter Anwendung der Regeln des Programms und dadurch ausgelöster Teilanfragen auf die Fakten zurückführen, bieten nur wenig Möglichkeiten für eine effiziente Aktualisierung der abgeleiteten Informationen [Sterling 86].

Für (deduktive) Datenbanksysteme wird dagegen mit dem Bottom-Up Verfahren allgemein eine datengetrieben Auswertung vorgeschlagen, die ausgehend von den Fakten sukzessive alle Regeln des Programms solange ausfuehrt, bis keine neue Information mehr abgeleitet werden kann [Ullman 88].

Besonders unter Echtzeitbedingungen ist die Effizienz der Auswertung und damit der Anfragebearbeitung das entscheidende Optimierungsziel. Eine Optimierung ist hier jedoch nur unter Ausnutzung aller verfügbaren Informationen aus der Abfrage selbst oder durch frühzeitige Einbeziehung von Ergebnissen aus anderen Teilanfragen möglich. Programmoptimierungen, die dies gewährleisten sind z.B. Selection Pushing [Kemp 89, Srivastava 93] oder Magic Sets [Ramakrishnan 88, Ramakrishnan 90, Ullman 88].

Es hat sich jedoch herausgestellt, daß hybride Systeme, die eine Mischung aus zielgetriebener (Top-Down) und datengetriebener (Bottom-Up) Auswertung darstellen, die besten Ergebnisse bezüglich der Zeiteffizienz liefern. Dies konnte vor allem im LOLA-System von [Brass 97a, Brass 97b] unter Beweis gestellt werden. Durch den hybriden Ansatz einer generell bottom-up erfolgenden Auswertung die, immer dann mit dem top-down erfolgenden Aufruf von Teilanfragen kombiniert wird, wenn die datengetriebene Auswertung zu Problemen führen würde, wie bei Auftreten von Rekursionen oder Negationen, kann man zudem darauf verzichten, für diese Bereiche komplexe, gesonderte Behandlungen einzuführen.

Ein Verfahren, das ähnlich leistungsfähig ist, ist z.B. die SLG Resolution [Chen 93, Chen 95, Chen 96], die mit Erfolg im XSB-System implementiert wurde [Sagonas 94, Pelov 99]. Außerdem gibt es einer Reihe differentieller Verfahren zur Rekursionsbehandlung [Balbin 86, Balbin 87, Güntzer 87] mit dem Ziel der effizienten und korrekten Ausführung von Rekursionen.

Zielsetzung der vorliegenden Arbeit ist jedoch eine rein datengetriebene Ausführung anhand einer Graphdarstellung, die sich als Ausführungsvorschrift für einen Datenflußrechner eignet.

Es sind daher unter anderem folgende Probleme zu lösen:

- Eine frühzeitige Verwendung von Teilergebnissen muß gewährleistet werden, um die Redundanz der Berechnungen gering zu halten.
- Es wird ein spezielles Verfahren für den Berechnungsablauf bei Rekursionen benötigt.
- Es wird außerdem ein spezielles Verfahren für die korrekte Behandlung von Negationen benötigt.

2.4 State of the Art

Logikprogrammiersprachen sind grundsätzlich einer datenflußartigen Auswertung zugänglich, wie bereits [Kifer 86] geziegt haben.

In diesem Kapitel werden zunächst Arbeiten aus dem Bereich der Logikprogrammierung vorgestellt, deren Zielsetzung analog zur Zielsetzung dieser Arbeit ist. Da sich jedoch zeigen wird, daß Bottom-Up Abarbeitungsprinzipien aus dem Gebiet der deduktiven Datenbanken eine größere Bedeutung zukommt, werden im weiteren Arbeiten untersucht, die diesem Ansatz folgen.

2.4.1 Arbeiten aus Logikprogrammierung und deduktiven Datenbanken

Arbeiten mit dem Ziel eines datengetriebenen Abarbeitungsmodells für deduktive Datenbanken gehen hauptsächlich vom Auswertungsmodell der Top-Down Verfahren mit SLD-Resolution für Logikprogramme, speziell PROLOG-Programme, oder Programme in PROLOG-ähnlichen Sprachen aus.

Obwohl in einem PROLOG-Programm inherent Parallelität enthalten ist und es mittlerweile eine breitgefächerte Forschung auf dem Gebiet der Implementierung von parallelen PROLOG-Sprachen auf Mehrprozessorsystemen gibt, existieren insgesamt nur wenige Arbeiten über PROLOG als Hochsprache für Datenflußrechner.

- *L. Bic*
  - Datenflußmodell für Logikprogramme

    Spezifikation: Logikprogrammierung.

    Prinzip: Nutzung von Oder-Parallelität und eingeschränkter Und-Parallelität (restricted AND-parallelism).

    Berechnungsmodell: Die Daten aus den Fakten bilden die Knoten des Datenflußgraphen, die Kanten zwischen den Knoten entstehen aus den durch die Fakten spezifizierten Beziehungen der Daten zueinander. Eine Abfrage wird als spezielle Datenstruktur generiert, als Token eingegeben und durch Anwendung der PROLOG-Regeln zu einer zweiten Graphstruktur erweitert, bis diese mit einem Teil des Ausgangsgraphen identisch wird [Bic, 84a, Bic, 84a].

    Architektur: Simulation eines Datenflußrechners.

    Wertung: Dieses Datenflußmodell ist aufgrund der auf den Fakten aufbauenden Graphdarstellung hauptsächlich für relationale Datenbanken von Interesse.

  - Datenflußmodell für Entity-Relationship-Modelle

    Spezifikation: Entity-Relationship-Modelle.

    Berechnungsmodell: Das Entity-Relationship-Modell wird direkt als Datenflußgraph verwendet. Die Auswertung startet bei einer Menge von Werten aus einer Entity und geht im nächsten Schritt zu Entities, die über Relationen mit dieser Entity verbunden sind [Bic 87, Bic 89].

    Architektur:Active Graph Model (AGM)

    Wertung: Obwohl ein Zusammenhang zwischen der algebraischen Darstellung relationaler Datenbanken und Entity-Relationship Modellen besteht, unterscheiden sich die aus Entity-Relationship Modellen erzeugten Graphen sowohl in ihrer Struktur, als auch in der Funktionalität der Knoten von den in dieser Arbeit verwendeten Graphen.

- *G. Blützingsloewen*

  Spezifikation: Relationale Datenbanksysteme.

  Prinzip: Aus der relationalen Darstellung der Query wird ein Datenflußgraph erzeugt. Die Knoten realisieren die relationalen Operationen und Methoden. Die Auswertung erfolgt im Sinne eines Datenflußgraphen, d.h. datenunabhängige Knoten können parallel ausgewertet werden [Bültzingsloewen 88, Bültzingsloewen 89].

  Architektur: KARDAMOM, eine Dataflow Database Machine.

  Wertung: KARDAMOM ist ein Mehrprozessorsystem, basierend auf von Neumann-Prozessoren und daher keine Datenflußrechnerarchitektur. Die Operationalität der Knoten und damit die Kommunikation zwischen den Knoten ist grundlegend anders organisiert, als bei dem in dieser Arbeit vorgestellten Datenflußmodell.

- *M. Bruynooghe*

  Spezifikation: PROLOG mit einer Erweiterung um Kontrollkonstrukte, sowie Modus- und Typdefinitionen.

  Prinzip: Bei diesem Verfahren handelt es sich um eine Top-Down Auswertung von Logikprogrammen unter Nutzung von Oder-Parallelität. Besonderer Aspekt ist die Bearbeitung von Listen. Das Konzept von Funktionen wird für die Behandlung von Rekursionen implementiert.

  Berechnungsmodell: Ein Makroknoten des Datenflußgraphen besteht aus Knoten, die die Elemente jeweils einer Regel repräsentieren. Er wird als Prozedur verstanden. Die Makroknoten sind eingebettet in eine Graphstruktur mit Knoten für die Verwaltung des Prozeduraufrufs und die Rückkehr aus der Prozedur [Bruynooghe 88].

  Architektur: Manchester Dataflow Computer.

  Wertung: Die Ergebnisse dieser Arbeit waren aufgrund des hohen Verwaltungsaufwands für das "prozedurale" Konzept nicht zufriedenstellend.

- *D. DeWitt*

  Spezifikation: Relationale Datenbanksysteme.

  Prinzip: Die datenflußbasierte Query-Auswertung geschieht nach folgendem Prinzip: die relationale Darstellung der Query wird in einzelne Prozesse aufgeteilt. Die Prozesse werden bis auf Aktivierung und Beendigung nur durch den Fluß von Daten synchronisiert [DeWitt 82, DeWitt 82].

  Architektur: GAMMA, eine Dataflow Database Machine.

  Wertung: Der Zusammenhang mit Datenflußgraphen ist bei der Query-Auswertung nur oberflächlich gegeben. Darüberhinaus handelt es sich bei GAMMA um ein Mehrprozessorsystem, basierend auf von Neumann-Prozessoren.

- *Z. Halim*

  Spezifikation: Logikprogrammiersprachen.

  Prinzip: Bei diesem Verfahren handelt es sich um eine Top-Down Auswertung von Logikprogrammen unter Nutzung von Oder-Parallelität.

  Berechnungsmodell: Der Datenflußgraph basiert auf Teilgraphen für jede Regel, wobei die Teilgraphen jeweils aus Knoten für die Unifikation des Regelaufrufs mit der Regel, die Aktivierung, dem sequentiellen "Aufruf" weiterer Teilgraphen und der Generierung einer Ausgabe durch das Zusammensetzen von Teilergebnissen besteht. Die Berechnung der Teilgraphen selbst kann parallel angestoßen werden [Halim, 84, Halim 86].

  Architektur: Manchester Dataflow Computer.

Wertung: Offen bleibt in der Arbeit die Verwaltung verschiedener Belegungen mit Werten in parallelen Zweigen des Graphen für die gleiche Variable. Hier ist im Prinzip die Integration eines aufwendigen Speichermanagements notwendig.

- *R. Hasegawa*

  Spezifikation: Logikprogrammiersprachen.

  Prinzip: Bei diesem Verfahren handelt es sich um eine Top-Down Auswertung von Logikprogrammen unter Nutzung von Oder-Parallelität. Wesentlicher Aspekt ist die effiziente, parallele Verarbeitung von Listen [Hasegawa 84].

  Berechnungsmodell: Die Knoten des Graphen realisieren im wesentlichen Unifikation, Aufruf weiterer Regeln, sowie Sammeln und Weitergabe von Ergebnissen.

  Architektur: Die Implementierung erfolgte auf einer real entwickelten Rechnerarchitektur, der *List-Processing Oriented Machine* (LPO) [Hasegawa 85].

  Wertung: Es wird nur Oder-Parallelität realisiert. Die Maschine ist jedoch gut geeignet für die Behandlung strukturierter Datenobjekte, insbesondere für Listen.

- *N. Ito*

  Spezifikation: Parallel PROLOG und *Concurrent PROLOG*.

  Prinzip: Mit diesem Verfahren werden Oder- und Und-Parallelität, sowie Parallelität beim Finden unifizierbarer Regeln vollständig genutzt.

  Berechnungsmodell: Der Datenflußgraph basiert auf Knoten für die Substitution und Unifikation von Variablen und Strukturen, ähnlich dem Unifikationsalgorithmus. Die Bindung von Variablen, sowie der lesende Zugriff auf Variablen in parallel ausgeführten Pfaden wird durch ein Semaphorkonzept geregelt [Ito 85, Ito 86].

  Architektur: Die real entwickelte Datenflußrechenrarchitektur ist die *Parallel Inference Engine* (PIM-D).

  Wertung: Die Graphen sind aufgrund des Synchronisationskonzepts mit Semaphoren sehr komplex. Das Verfahren ist jedoch gut geeignet für die Auswertung verschiedener paralleler PROLOG-Varianten.

- *P. Kacsuk*

  Spezifikation: PROLOG.

  Berechnungsmodell: Aus einem PROLOG-Programm wird ein Rule/Goal Graph erzeugt. Abfragen werden als Token an den entsprechenden Startknoten injeziert und wandern mit Tiefensuche durch den Graphen, wobei eine Kante in beide Richtungen durchlaufen wird: von oben nach unten als Abfrage und von unten nach oben als Ergebnis [Kacsuk 90, Kacsuk 91, Kacsuk 92].

  Architektur: Die Implementierung erfolgte schließlich nicht auf einem Datenflußrechner, sondern einem speziellen Multiprozessorsystem, *Homogenous Processor Space* genannt.

  Wertung: Dieses Form der Abarbeitung fällt unter den Begriff anforderungsgetriebene Abarbeitung. Es handelt sich dabei also nicht um eine rein datengetriebene Auswertung.

- *A.V.S. Sastry*

  Spezifikation: Logikprogrammiersprachen.

  Prinzip: Bei diesem Verfahren handelt es sich um eine Top-Down Auswertung von Logikprogrammen unter Nutzung von Oder-Parallelität. Zusätzlich kann das Auffinden anwendbarer Regeln parallel geschehen.

Berechnungsmodell: Die Knoten des Datenflußgraphen repräsentieren komplexe, zusammengesetzte Funktionen, die außer Unifikation und dem Sammeln von Ergebnissen die Verwaltung von Kontextwechseln übernehmen [Sastry 88, Sastry 91].

Architektur: Die vorgestellte Rechnerarchitektur ist die *Multi-Ring Dataflow Architecture*. Sie basiert auf den Prinzipien der Manchester Dataflow Machine.

Wertung: Die auf der Manchester Dataflow Machine notwendige Kontextverwaltung ist aufwendig. Sie löst jedoch das Problem verschiedener Bindungen in parallelen Zweigen des Graphen für die gleiche Variable.

- *C.C. Tseng*

  Spezifikation: Logikprogrammierung.

  Prinzip: Top-Down Auswertung von Logikprogrammen unter Nutzung von Oder- und Und-Parallelität (restricted AND-parallelism, stream AND-parallelism).

  Berechnungsmodell: Die Knoten des Graphen realisieren im wesentlichen Unifikation, Aufruf weiterer Regeln, sowie Sammeln und Weitergabe von Ergebnissen. Dabei ermöglicht eine spezielle Datenstruktur, der S-Stream, die Verwaltung unterschiedlicher Bindungen in parallelen Zweigen des Graphen [Tseng 88a, Tseng 88b].

  Architektur: LogDf, basierend auf der Manchester Dataflow Machine.

  Wertung: Das vorgestellte Modell der S-Streams erlaubt eine gute Parallelisierung unter geringem Verwaltungsaufwand.

- *S. Umeyama*

  Spezifikation: Logikprogramme.

  Prinzip: Bei diesem Verfahren handelt es sich um eine Top-Down Auswertung von Logikprogrammen unter Nutzung von Oder-Parallelität.

  Berechnungsmodell: Genaugenommen handelt es sich um einen Prozeßgraphen anstelle eines Datenflußgraphen, da er außer Knoten für Unifikation und Sammeln zurückgegebener Ergebnisse Knoten für das Anstoßen von Kindprozessen enthält [Umeyama 83].

  Architektur: Mehrprozessorsysteme.

  Wertung: Dieses Berechnungsmodell ist in der vorliegenden Form nicht auf einem Datenflußrechner implementierbar.

- *M. J. Wise*

  Spezifikation: Epilog, eine Logikprogrammierungs-Sprache mit Modusdefinitionen.

  Prinzip: Auch hier geht es um die Top-Down Auswertung von Logikprogrammen unter Nutzung von Oder-Parallelität. Negationen werden durch Negation as Failure implementiert.

  Berechnungsmodell: Der Datenflußgraph repräsentiert im wesentlichen die Elemente des Logikprogramms. Es gibt spezielle Knoten für die Behandlung des Cuts [Wise 86].

  Wertung: Die Arbeit ist über das Stadium einer Sprachentwicklung nicht weiter fortgeschritten.

Alle bekannten Ansätze aus dem Bereich deduktiver Datenbanken führen ganz offensichtlich nicht zu Berechnungsmodellen, die direkt für die Anwendung auf einem Datenflußrechner geeignet sind.

Die genannten Ansätze zur datenflußgetriebenen Abarbeitung eines Top-Down Berechnungsmodells für Logikprogramme müssen dagegen meist eine "Code-Duplizierung" in

Kauf nehmen. Für jede Regel werden zunächst Knoten, die für die Unifikation und den Aufruf weiterer Regeln zuständig sind, erzeugt. Dies entspricht einem Fluß von Anforderungen. Zusätzlich werden die Knoten durch weitere Knoten ergänzt, die dem Zurückerhalten, Zusammenfassen und dem weiteren Zurückschicken von Ergebnissen dienen. Dieser zweite Teil stellt im wesentlichen eine "Spiegelung" des ersten Teils dar.

Das Prinzip soll an den folgenden Beispielen veranschaulicht werden.

Abbildung 7 stellt zunächst eine Top-Down Auswertung anhand eines Dependency-Graphen zu dem folgendem Beispiel dar.

Beispiel:

$f(X,Z) \leftarrow g(X,Y),\ \ h(Y,Z).$
$h(Y,Z) \leftarrow k(W,Z),\ \ Y > W.$
$g(1,6).\ \ g(1,7).\ \ g(2,2).\ \ g(2,4).$
$k(2,4).\ \ k(4,8).\ \ k(8,6).$
$\leftarrow f(2,Z).$

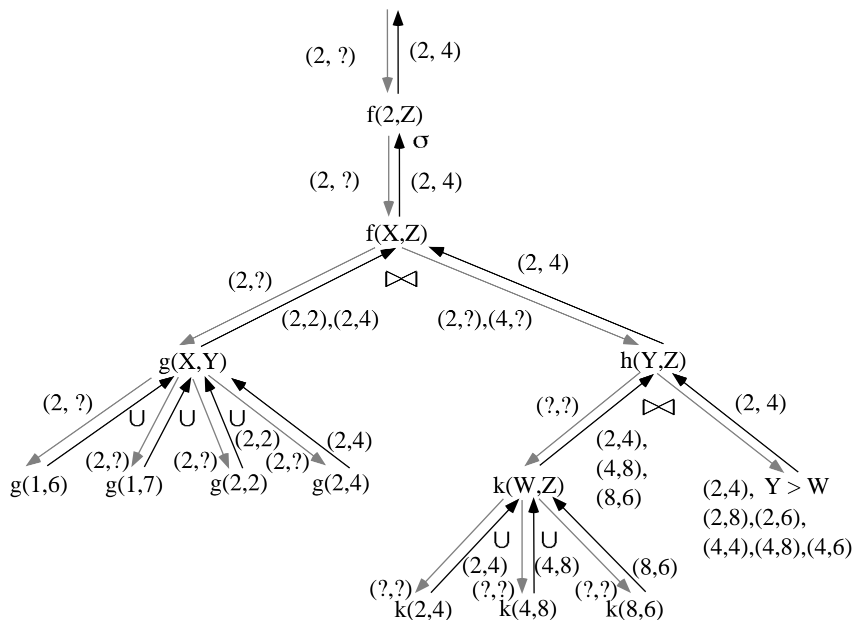


Abbildung 7: Top-Down Auswertung

Abbildung 8 zeigt den Datenflußgraphen zu obigem Beispiel, wie er durch diejenigen Ansätze aus dem Bereich der Logikprogrammierung entsteht, die mit einer Duplizierung der Knoten für die Rückgabe der Ergebnisse arbeiten.

Obwohl dieser Graph unidirektional ausgewertet werden kann, hat er nichts mit dem in der Arbeit vorgestellten Graphen gemeinsam, der so erzeugt wird, daß die unidirektionale Auswertung ohne eine solche Spiegelung erreicht werden kann.

Allgemein stoßen Arbeiten mit dem Ziel der Auswertung von Logikprogrammen auf Datenflußrechnern an ähnliche Grenzen wie parallele Prolog-Implementierungen auf von Neumann-Rechnerarchitekturen: aufgrund der Variablität der Abfragen und damit

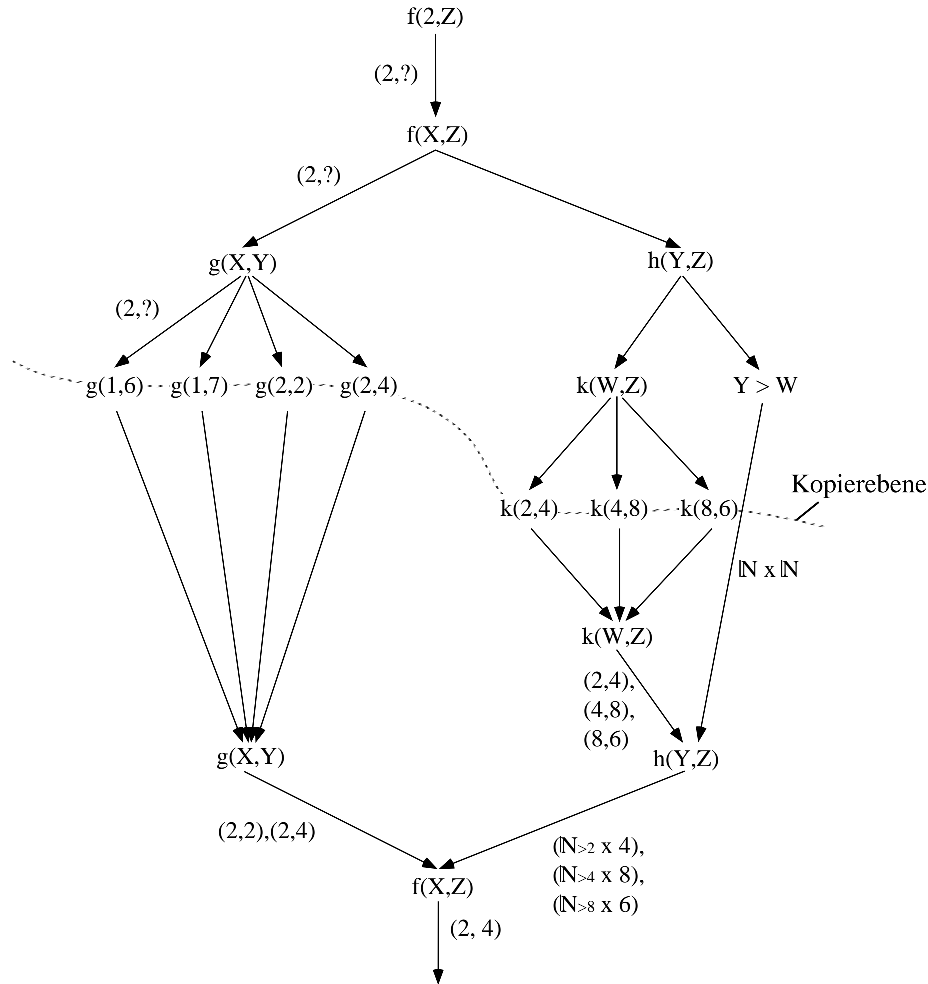


Abbildung 8: Datenflußgraph mit Code-Duplizierung

der Eingabewerte läßt sich die notwendige und gleichzeitig hinreichende Menge an Parallelität nicht statisch bestimmen. Wird soviel wie möglich parallel berechnet, so werden dabei auch Zweige verfolgt, die überflüssig sind und deren Ergebnisse später ungültig werden. Berechnet man jedoch nur sicher benötigte Regeln parallel, erhält man nicht die vollständige Ausnutzung der möglichen Parallelität.

Ferner ist als weiteres Problem bei parallelen Logikprogramm-Implementierungen das bei jedem Berechnungsschritt übermäßige Ansteigen der Menge zu berechnender Regeln zu beobachten, weshalb die Implementierung einer vollständigen Und-Parallelität zusammen mit Oder-Parallelität bislang kaum realisiert wurde. Dieser Nachteil überträgt sich analog auf einen, aus einem Logikprogramm erzeugten Datenflußgraphen.

Zusammenfassend läßt sich folgendes feststellen:

1. Alle bestehenden Arbeiten, die im Bereich relationaler Datenbanken anzusiedeln sind, arbeiten mit Datenflußgraphen, die nicht aus einer Darstellung der Datenbank mittels Logikprogrammierung resultieren.
   Da es in dieser Arbeit um die Transformation eines Logikprogramms in Datenflußgraphen geht, haben diese Arbeiten keine Bedeutung bezüglich der Ziele dieser Arbeit.

2. Alle bestehenden Arbeiten, die zum Ziel haben, aus Logikprogrammen Datenflußgraphen zu erzeugen, verwenden ein Top-Down Berechnungsmodell für die Erstellung und Ausführung des Datenflußgraphen.
   Da das spezielle Anwendungsgebiet, das hier vorliegt, andere Strukturbedingungen für Logikprogramme vorgibt (siehe Kapitel 1.1.2), als bei allgemeinen Anwendungen vorliegen, haben diese Arbeiten ebenfalls keine Bedeutung bezüglich der Ziele dieser Arbeit.

Fazit: Es gibt bislang keine veröffentlichen Arbeiten, die zum Ziel haben, Logikprogramme auf Datenflußrechnern mit einem Berechnungsmodell, das der Bottom-Up Ausführung entspricht, zu implementieren.

2.4.2 Verwendete Methoden

Das in dieser Arbeit entwickelte Berechnungsmodell verwendet daher einige der Verfahrenstechniken aus dem Bereich deduktiver Datenbanken. Dazu gehören:

- *Erzeugung eines Operatorbaums*

  Die Erzeugung des Datenflußgraphen, der in der Arbeit vorgestellt wird, startet mit einem Graphen, der konzeptuell mit dem Operatorbaum vergleichbar ist (siehe Kapitel 3.2.1).

- *Codeduplizierung zur Erzeugung eindeutiger Bindungsmuster*

  Da im Datenflußgraph wie bei der Magic Sets Methode mit eindeutigen Bindungsmustern gearbeitet werden muß, bedeutet dies die Vervielfältigung von Knoten mit mehreren Bindungsmustern, um Knoten mit eindeutigen Bindungsmustern zu erhalten (siehe Kapitel 3.3.3).

- *Extraktion von Funktionen aus der Parameterliste*

  Um den Knoten des Datenflußgraphen eine einfach zu beschreibende Funktionalität zuordnen zu können, müssen Funktionen aus der Parameterliste von Literalen entfernt werden und daraus eigenständige Literale erzeugt werden. Dies geschieht mit Hilfe einer Transformation, die mit der Φ-Transformation, die in der Arbeit von [Bader 90] für Funktionen vorgestellt wird, vergleichbar ist. Dies ist für Funktionen möglich, bei denen das Ergebnis der Funktion die Ausgabe sein soll, und für Funktionen, für die eine Umkehrfunktion spezifiziert ist (siehe Kapitel 3.1.3).

Zusätzlich werden folgende Methoden eingesetzt, die sich aus bisher bekannten Verfahrenstechniken ableiten, diese aber erweitern:

- *Extraktion von Listen mit Operatoren aus der Parameterliste*

  In der Arbeit von [Bader 90] wird das Verfahren, Funktionen aus der Parameterliste zu entfernen, nicht auf Listen mit Operatoren angewendet. Der Vorteil, Listen mit Operatoren als Funktionsaufrufe aufzufassen, ermöglicht es, im Datenflußgraph später an diesen Stellen nicht mit Unifikation operieren zu müssen, sondern funktional.

  Bei Listen mit Konstruktoren lassen sich sowohl Funktion (Zusammensetzen einer Liste aus einzelnen Elementen und Rest) als auch Umkehrfunktion (Auslesen der Elemente am Anfang der Liste, Ausgeben des Restes) einfach bilden (siehe Kapitel 3.1.3).

- *Analyse der Parametermodi*

  Für die Erzeugung des Datenflußgraphen ist eine Analyse der Parametermodi notwendig. Diese soll jedoch ohne Berücksichtigung der Reihenfolge von Regeln und Literalen im Rumpf einer Regel vorgehen (siehe Kapitel 3.1.3.7). Gleichzeitig kann die Wahl der Modi nicht beliebig frei stattfinden, sondern muß aufgrund der Integration von Systemprädikaten festen Randbedingungen genügen.

  Daher geht die Analyse nicht wie bereits bekannten Verfahren, z.B. SIP-Verfahren, vor. Die Analogie ist nur insoweit gegeben, daß aus der Existenz einer Erzeugungsstelle für Daten folgt, daß andere Stellen als Verwendungsstellen dienen können.

  Für die Analyse der Modi ist eine graphweite Betrachtung der Verwendung von Parametern notwendig. Somit ist es zunächst erforderlich, den Parametern aller Literale graphweit eindeutige Namen zuzuweisen (siehe Kapitel 3.3.2). Es kann vorkommen, daß ein Parameter mehr als einen eindeutigen Namen erhält. Die Anzahl eindeutiger Namen eines Parameters entspricht der Menge der möglichen Modi, mit denen dieser Parameter verwendet wird.

  Da bei der Analyse an vielen Stellen Wahlmöglichkeiten auftreten, von denen sich jedoch in der weiteren Analyse im allgemeinen nur ein Teil als korrekt herausstellt, ist zusätzlich die Integration eines Algorithmus zur Vermeidung modus-zyklischer Datenabhängigkeiten in die Analyse zwingend als Erweiterung notwendig (siehe Kapitel 3.3.4).

- *Aufspaltung von Rekursionen in eine absteigende und eine aufsteigende Schleife*

  Rekursionen in Logikprogrammen entsprechen Zyklen im Dependency-Graphen. Sollen sie auf einen Datenflußgraphen übertragen werden, muß dafür gesorgt werden, daß sich Zyklen unidirektional ausführen lassen. Dazu eignet sich eine Aufteilung der Rekursion in zwei getrennte Zyklen, einen "absteigenden" Zyklus für die Eingabeparameter der Rekursion, einen "aufsteigenden" Zyklus für die Ausgabeparameter (siehe Kapitel 3.4.4.2). Auf diese Art lassen sich auch alle komplexeren Formen von Rekursionen (kaskadische, verschachtelte, mehrfach lineare Rekursion) behandeln (siehe Kapitel 3.4.4.4).

  Bei Anwendung der Magic Set Methode entsteht eine ähnliche Aufteilung der Rekursion in Schleifen, die Anzahl der Schleifen ist jedoch im allgemeinen durch die bei Magic Sets zusätzlich erzeugten Regeln deutlich größer als die in dieser Arbeit entstehende Anzahl von zwei Schleifen pro rekursiver Regel. Das hier vorgestellte Verfahren stellt daher eine Verfeinerung der Magic Set Methode bezüglich der Behandlung von Rekursionen dar. Damit ist auch der Aufbau der Gesamtstruktur deutlich komplexer. Das Problem, das für zunächst stratifizierte Programme nach Anwendung der Magic Sets Methode die Stratifizierung nicht mehr garantiert ist, tritt bei dem in dieser Arbeit vorgestellten Verfahren nicht auf (siehe Kapitel 3.4.4).

- *Integration von Elementen einer Top-Down Ausführung*

  Die Form der Abarbeitung des Datenflußgraphen entspricht rein prinzipiell dem Bottom-Up Verfahren. In der vorliegenden Arbeit wird jedoch der Dependency Graph, der in klassischen Datenbanksystemen dieser Auswertungsform zugrunde liegt, nicht 1:1 auf einen Datenflußgraphen übertragen. Vielmehr werden aus Top-Down Verfahren bekannte Request-Ansätze in den Graphen integriert. Dies geschieht in einer bislang noch nicht existierenden Form derart, daß der resultierende Graph nach wie vor rein unidirektional auswertbar ist. Dies ist Voraussetzung für eine spätere Ausführung auf einem Datenflußrechner.

Die Integration von Top-Down Elementen geschieht durch eine entsprechende Analyse der Modi der Parameter und einer adäquaten Erzeugung des Datenflußgraphen. Das Verfahren entspricht damit einer Kombination von Top-Down und Bottom-Up Verfahren mit sequentiellen Phasen, wobei jedoch keine Rewriting Techniken verwendet werden.

2.4.3 Abgrenzung des Verfahrens

Das aus dieser Arbeit resultierende Berechnungsmodell ist in seiner Struktur grundlegend anders, als die im Bereich deduktiver Datenbanken bekannten Formen von Graphen.

In der Begriffswelt der deduktiven Datenbanken fallen unter den Begriff Datenfluß alle Betrachtungen zur Werteübergabe zwischen Parametern. Dies gilt z.B. für die Bestimmung von Parametermodi mit Hilfe der *Sideways Information Passing (SIP)-Strategie* [Cremers 94]. Die Repräsentation dieser Werteübergaben wird (fälschlicherweise) als Datenflußgraph bezeichnet. In der Begriffswelt der Datenflußrechnerarchitekturen und damit auch in der vorliegenden Arbeit ist demgegenüber aber ein Datenflußgraph ein Graph, der ein Berechnungsmodell für ein Programm darstellt. Eine explizite Verwendung von Datenflußgraphen in dieser Art gibt es im Bereich der deduktiven Datenbanken bislang nicht.

Dies soll der in Abbildung 9 gezeigte Datenflußgraph zu folgendem, bereits im letzten Abschnitt verwendeten Beispiel, demonstrieren:

Beispiel:

$f(X,Z) \leftarrow g(X,Y),\;\; h(Y,Z).$
$h(Y,Z) \leftarrow k(W,Z),\;\; Y > W.$
$g(1,6).\;\; g(1,7).\;\; g(2,2).\;\; g(2,4).$
$k(2,4).\;\; k(4,8).\;\; k(8,6).$
$\leftarrow f(2,Z).$

Ein reines Bottom-Up Verfahren müßte in diesem Beispiel Annahmen über die Wertemengen für Y in der Regel $h(Y,Z)$ treffen, die sich dann je nach der Wertemenge der extensionalen Datenbank als sehr groß oder sogar unendlich erweisen kann, bis der Wert für Y durch die Regel $f(X,Z)$ auf den einzigen gültigen Wert eingeschränkt wird. Figur 10 veranschaulicht dies.

Magic Sets verhindern dies durch entsprechende Umformung der Regeln, wenn für alle beteiligten Variablen Bindungslieferanten gefunden werden können. Somit würden im Regelrumpf der Regel $h(Y,Z)$ beispielsweise auch die Bindungen für Y erzeugt werden und zwar vor ihrer Verwendung in $Y > W$. Das Verfahren zeigt Figur 11.

Da keine identischen Teile des Programms bei einer semi-naiven Auswertung mehrfach berechnet werden, ist diese zu erwartende Aufblähung des Programms ein zu vertretender Nachteil. Für die direkte Umsetzung in einen Datenflußgraphen eignet sich ein mit Magic Sets transformiertes Programm jedoch nicht, da selbst bei Structure Sharing im Graphen durch die Menge an hinzugekommen Kanten die Menge an erforderlicher Kommunikation beträchtlich steigt. Für Datenflußrechner gilt jedoch, wie auch für alle anderen Arten von Parallelrechnern, daß Programme um so effizienter abgearbeitet werden können, je geringer die Kommunikationslast ist.

Der in der Arbeit vorgestellte Datenflußgraph kommt dagegen ohne die Erzeugung zusätzlicher Knoten aus, um dieses Problem zu lösen: in dem Graphen wird eine Kante

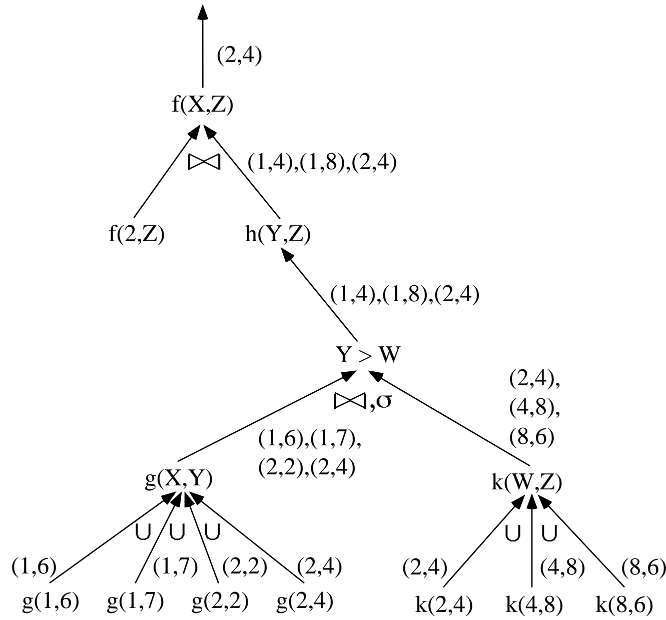


Abbildung 9: In dieser Arbeit vorgestelltes Auswertungsverfahren

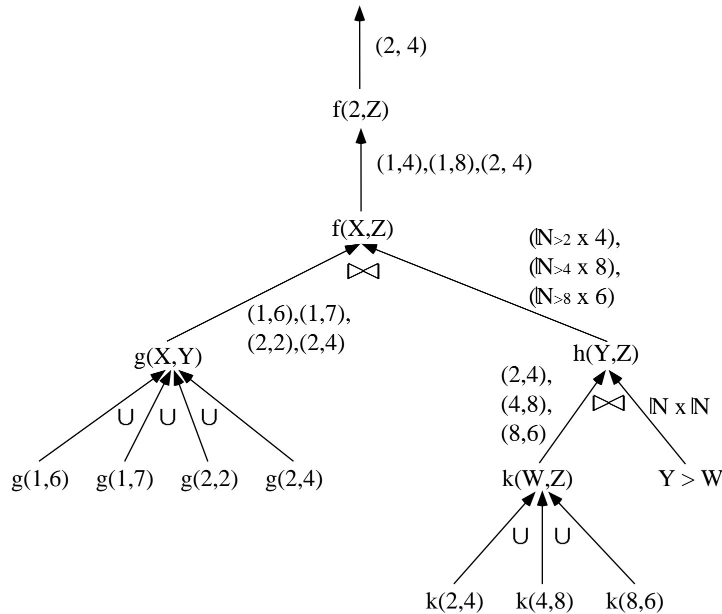


Abbildung 10: Bottom-Up Auswertung

erzeugt, die von dem Fakt $g(6)$ direkt zu dem Literal $Y > W$ führt und ebenso Kanten von den Fakten $k(1,2)$, $k(2,4)$, $k(3,6)$ zu einem Oder-Knoten und von dort direkt zu dem Literal $Y > W$. Damit wird eine sinnvolle Integration von Funktionen und Systemprädikaten ermöglicht und gleichzeitig verhindert, daß diese unendliche Wertemengen als Ergebnis zurückliefern. Auf Funktionen und Systemprädikate kann bei der Implementierung von Steuerungssystemen nicht verzichtet werden.

Die Vorteile der Auswertungsmethode des in dieser Arbeit erzeugten Datenflußgraphen

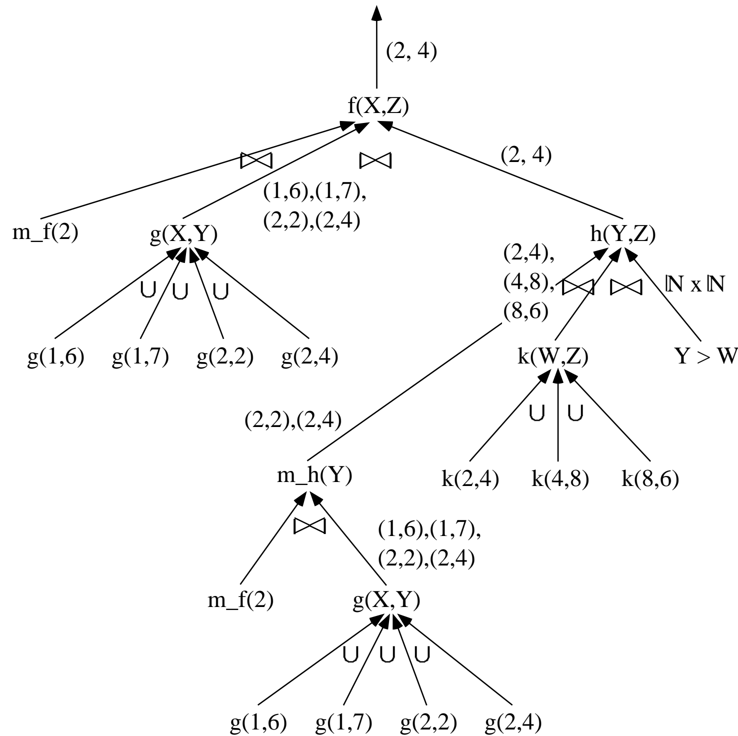


Abbildung 11: Bottom-Up Auswertung nach der Magic Set-Transformation

gegenüber Bottom-Up Verfahren sind daher:

- Die Verwendung von Systemprädikaten, Funktionen und Operationen führt zu keinen Problemen durch unendlich große Tupelmengen.
- Tupelmengen werden wie bei Magic Sets frühzeitig eingeschränkt, ohne daß Rewriting Techniken verwendet werden müssen und sich die die Größe des Graphen gegenüber der Größe des Orginalprogramms überproportional verändert.
- Bei Rekusionen bleibt die Stratifizierung von Programmen erhalten.

Die Ausführung des Datenflußgraphen gemäß des Prinzips der datengetriebenen Abarbeitung bietet darüberhinaus folgende Vorteile:

- Horizontale Parallelsierung auf feinkörniger Ebene durch das Datenflußprinzip.
- Vertikale Parallelisierung durch mehrstufige Ausführung.
- Reduzierung der Anzahl auszuführender Knoten durch das Ereignisflußprinzip.
- Einfaches Scheduling in Datenflußrechnerarchitekturen.
- Einfache Kommunikation in Datenflußrechnerarchitekturen.

2.5 Detailziele der Arbeit

Ziel der Arbeit ist die Entwicklung einer Systemarchitektur für die effiziente, parallele Ausführung der wissensbasierten Komponente von Steuerungsmodulen für autonom-mobile Systeme.

Die in dieser Arbeit vorgestellte Systemarchitektur besteht aus folgenden Komponenten:

- einem Berechnungsmodell in Form eines Datenflußgraphen,
- einer Ausführungsvorschrift für die Elemente des Datenflußgraphen,
- einer Transformationsmethode, mit der das in einer Sprache aus dem Bereich der Logikprogrammierung spezifizierte Expertensystem in einen Datenflußgraphen übersetzt wird, und
- einer Rechnerarchitektur, die das Berechnungmodell mit dem Ziel einer effizienten, parallelen Ausführung implementiert.

Weitere Teilziele sind darüber hinaus:

- Eine weitestgehende Unabhängigkeit von der von einem Programmierer gewählten Reihenfolge der Aufschreibung der Spezifikation.
- Die Optimierbarkeit des Berechnungsmodells hinsichtlich der Menge an notwendigen Berechnungen.
- Die Behandelbarkeit von Systemprädikaten, Funktionen und Operationen.
- Die korrekte Behandlung von Rekursionen.
- Die korrekte Behandlung von Negationen.
- Die Integrierbarkeit weiterer Komponenten in das Berechnungsmodell, die auf anderen Spezifikationsmethoden beruhen.
- Die interpretative Ausführung auf einem von Neumann-Rechner zu Simulations- und Testzwecken, sowie als reale Implementierungsform.

3 Das Datenflußmodell

Als Ausgang soll die Spezifikation eines wissensbasierten Systems in $\text{DATALOG}_f^{\neg}$ verwendet werden. $\text{DATALOG}_f^{\neg}$ steht dabei für eine Erweiterung von $\text{DATALOG}^{\neg}$ um eine Reihe von Funktionssymbolen und Systemprädikaten. In Abschnitt 3.1 werden sowohl die Bedingungen für $\text{DATALOG}_f^{\neg}$ beschrieben, als auch die für die Transformation notwendige Vorarbeiten.

Die Transformation der Spezifikation in einen Datenflußgraphen erfolgt dann in drei Schritten:

- Die Erzeugung eines Strukturgraphen wird in Abschnitt 3.2 beschrieben.
- Die Analyse der Modi von Parametern und damit der Datenabhängigkeiten im Graphen erfolgt in Abschnitt 3.3.
- Die Konstruktion des Datenflußgraphen wird in Abschnitt 3.4 zur Erleichterung des Verständnisses zunächst für zyklenfreie Datenflußgraphen und anschließend für Datenflußgraphen mit Zyklen beschrieben.

Abschnitt 3.5 beinhaltet ein Beispiel.

Anschließend wird die Funktionalität der Knoten des Datenflußgraphen in Abschnitt 3.6 spezifiziert. Den Abschluß dieses Kapitels bildet der Beweis der Korrektheit der Transformation in Abschnitt 3.7.

3.1 Die Spezifikationssprache

Ein wesentliches Anliegen der vorliegenden Arbeit ist es, die Spezifikation eines wissensbasierten Steuerungsmoduls für autonom-mobile Systeme in einer standardisierten Programmiersprache zu erlauben. Eine solche, für die Erstellung von Expertensystemen häufig verwendete Programmiersprache aus dem Bereich der Logikprogrammierung ist PROLOG [Altenkrüger 87]. Da die Semantik von PROLOG sehr komplex ist, bietet sich für diese Arbeit die aus dem Bereich der deduktiven Datenbanken bekannte Sprache DATALOG (bzw. $\text{DATALOG}^{\neg}$) an, die alle wesentlichen Sprachkonzepte der Logikprogrammierung beinhaltet. Sie muß für diese Arbeit lediglich um Systemprädikate erweitert werden und wird daher $\text{DATALOG}_f^{\neg}$ genannt.

3.1.1 Anforderungen und Randbedingungen

Die Elemente eines Logikprogramms lassen sich in einfacher Weise auf die Elemente eines Graphen abbilden, wie man anhand bekannter Darstellungsformen wie z.B. dem Rule/Goal Graphen in [Ullman 85] sehen kann.

Im Unterschied zu Datenbankanwendungen werden bei Steuerungssystemen vielfach arithmetische Ausdrücke, wie Zuweisungen und Vergleiche, benötigt. Diese können durch Systemprädikate implementiert werden. Ebenso kann der Aufruf komplexerer Bibliotheksfunktionen durch Systemprädikate realisiert werden.

Zusätzlich können Operationen und Funktionsaufrufe in beliebigen Literalen auftreten, so auch in Systemprädikaten z.B. für Vergleiche oder Zuweisungen. Insbesondere können Operationen und Funktionsaufrufe auch in einem Regelkopf auftreten.

Eine Negation von Literalen mit not muß möglich sein. Einschränkungen dürfen dabei nur dort geschehen, wo die Verwendung zu fehlerhaften, bzw. unsicheren Ergebnissen führen würde, wie das Auftreten von not in einer Rekursion. Es werden daher stratifizierte Programme als Eingabe gefordert.

Die Voraussetzung für die Verwendung von Systemprädikaten oder negierten Literalen ist auch in dieser Arbeit, daß die Programme bedingt sicher sein müssen. Das bedeutet, daß jeder Parameter, der in einer Regel in einem Systemprädikat oder in einem negierten Literal vorkommt, zusätzlich mindestens im Kopf der Regel oder in einem Rumpfliteral, das weder negiert ist, noch ein Systemprädikat darstellt, auftreten muß. Diese Bedingung kann im Rahmen der Arbeit jedoch soweit gelockert werden, daß bedingte Sicherheit nur für die Parameter von Systemprädikaten oder negierten Literalen gefordert wird, die Werte als Eingabe benötigen, um korrekt ausgeführt werden zu können. Darüberhinaus sollte gewährleistet sein, daß für diese Parameter an mindestens einer Stelle im Programm eine Ausgabe erzeugt wird. Verzichtet man auf diese Voraussetzung, läßt sich die Endlichkeit der Ergebnisse bei Systemprädikaten nicht garantieren. Im hier vorgestellten Modell kann der Datenflußgraph zudem nicht korrekt erzeugt werden.

Der Cut, als besonderes Sprachkonstrukt von Prolog zur Steuerung des Kontrollflusses, wird dagegen nicht implementiert. Ein korrektes Programm enthält, wenn überhaupt, nur *green cuts* [Shapiro 83]. Diese verändern die (Herbrand-)Semantik jedoch nicht, sondern schneiden lediglich überflüssige Auswertungspfade ab. Das ist jedoch nur bei Top-Down Verfahren möglich. Das in dieser Arbeit vorgestellte Auswertungsmodell agiert jedoch weitestgehend bottom-up und berechnet dabei alle möglichen Lösungen.

Schleifen werden in Logikprogrammen durch Rekursionen realisiert. Im Rahmen dieser Arbeit soll daher auf die Implementierung der linearen Rekursion, als einfachster und damit grundlegender Rekursion, besonderer Wert gelegt werden. Gleichzeitig sollen die Voraussetzungen zur Implementierung jeder beliebigen Rekursion geschaffen werden. In einem korrekten Programm terminiert jede Rekursion nach endlicher Zeit auch dann, wenn alle Lösungen gefunden werden sollen.

Da es sich bei den Eingabedaten für die dieser Arbeit zugrunde liegenden Anwendungen in den meisten Fällen um Sensordaten und damit z.B. um beliebige, reelle Zahlen handelt, sind diese im Prinzip (außer durch Hardwarevoraussetzungen) nicht bereichsbeschränkt.

Zusammenfassend läßt sich über die zu verarbeitenden Programme folgendes aussagen:

- Spezifikationssprache ist $\text{DATALOG}_f^\neg$.
- Die Negation mit not ist prinzipiell zulässig.
- Systemprädikate dürfen verwendet werden.
- Zuweisungen und Vergleiche dürfen als eigene Literale vorkommen.
- Mathematische Ausdrücke und Funktionen dürfen in der Parameterliste eines Literals auftreten.
- Mathematische Ausdrücke und Funktionen dürfen auf der rechten Seite einer Zuweisung, bzw. auf beiden Seiten eines Vergleichs auftreten.
- Die Eingabedaten sind nicht bereichsbeschränkt.

Definition 3.1 *Konsistenzbedingungen an ein $DATALOG_f^\neg$-Programm*

Ein $\text{DATALOG}_f^\neg$-Programm sei konsistent, wenn gilt:

- Das Programms ist bedingt sicher, d.h. für jeden im Programm auftretenden Parameter wird an mindestens einer Stelle, die nicht unter einer Negation auftritt, ein Wert erzeugt, definiert oder aus einem Datenspeicher eingelesen (siehe auch Anhang A.2 für die Definition bedingter Sicherheit).
- Es gibt keine zyklischen Datenabhängigkeiten außerhalb von Rekursionsschleifen.
- Das Programm ist stratifiziert, d.h. der Aufruf von Rekursionen erfolgt nicht negiert.
- Alle verwendeten mathematischen Ausdrücke und Funktionen müssen semi-deterministisch (nach [Somogyim 96]) sein, d.h. jede Anfrage kann entweder scheitern oder erfolgreich sein, wobei im Erfolgsfall genau ein eindeutiges Ergebnis ausgegeben wird (siehe Abschnitt 3.1.3.3).
- Das Ergebnis des Programms enthält keine alternativen Lösungen. Dies ist z.B. sichergestellt, wenn keine alternativen Fakten (direkte Oder-Verknüpfung von Fakten) zugelassen sind und sich alle alternativen Regeln gegenseitig ausschliessen, d.h. es ist immer nur eine Regel erfolgreich. Will man diese Restriktion lockern und alternative Zwischenergebnisse zulassen, so muß gewährleistet sein, daß sie eindeutige Schlüsselkandidaten enthalten und durch eine entsprechende Auswahl über die Schlüsselkandidaten für das Endergebnis entsprechend eingeschänkt werden. Dies könnte gegebenenfalls durch entsprechende Zusatzregeln erfolgen.
- Zu jeder Rekursion existiert mindestens ein Terminierungszweig.
- Das Programm terminiert in endlicher Zeit auch dann, wenn alle Lösungen gefunden werden sollen. Dies erreicht man, wenn die Wertemenge bei Rekursionen nach endlich vielen Durchläufen garantiert stationär wird. Dafür dürfen nur nicht-aufschichtende Programme zugelassen werden, d.h. Rekursionen mit Ausdrücken wie $p(f(x)) \leftarrow ..., p(x), ...$ dürfen nicht auftreten [Apt 89, Apt 91]. ◇

3.1.2 Typen von Fakten

Üblicherweise realisieren in $\text{D}\small{\text{ATALOG}}_f^{\neg}$ Fakten die extensionale Datenbank. Im vorliegenden Anwendungsfall ist die extensionale Datenbank durch eine Schnittstelle zu anderen Modulen des Steuerungssystems gegeben. Auf diese Daten muß das wissensbasierte Modul über eine Schnittstellendefinition zugreifen können.

Für eine solche Schnittstellendefinitionen sind verschiedene Formen denkbar. Um jedoch bei einer in $\text{D}\small{\text{ATALOG}}_f^{\neg}$ realisierbaren Form der Spezifikation zu bleiben, soll die Schnittstelle über Fakten realisiert werden. Das Prädikat des Fakts ist dabei das Attribut (im Sinne einer Adreßangabe), über das die einzulesenden Werte gefunden werden können.

Dabei existiert zu jedem Parameter eines Fakts genau ein Wert in einem dafür reservierten Speicherplatz. Aus der Sichtweise relationaler Datenbanken spezifiziert das Fakt dabei eine Relation über die durch die Parameter des Fakts ausgegebenen Werte. Die Werte bilden zusammen ein (Werte-)Tupel.

Beispiel:

Zulässig als Schnittstelle zum Datenspeicher sind also Fakten wie:

own_velocity (*Vel*).
curve (*Angle, Length*). ◇

Dies schränkt die Mächtigkeit der Sprache nicht ein, da sich, falls notwendig, leicht eine Regel angeben läßt, die die Werte von Fakten zur mehrfachen Weitergabe vervielfältigt.

Beispiel:

velocities (*Vel, LeftVel, FrontVel, LeftVel, FrontVel*).

läßt sich ersetzen durch:

velocities (*Vel, LeftVel, FrontVel, LeftVel, FrontVel*) ←
  own_velocity (*Vel*),
  front_velocities (*LeftVel, FrontVel*).
own_velocity (*Vel*).
front_velocities (*LeftVel, FrontVel*). ◇

Zusätzlich zu den Fakten, die das Interface realisieren, kann es auch Fakten geben, die explizit Werte (durch die Angabe von Konstanten) definieren oder die Eingaben vom Programm erhalten und Ausgaben an das Programm zurückgeben:

Beispiel:

Es können durchaus Fakten vorkommen wie z.B.:

state_transition (*2,4*).
state_transition (*3,4*).
state_transition (*4,5*).
state_transition (*5,7*).,
usw...

In diesem Zusammenhang kann folgendes, ebenso zulässiges Fakt vorkommen:

state_transition (*X,X*).

Das Fakt erhält für den ersten Parameter eine Eingabe vom Programm und gibt diese über den zweiten Parameter an das Programm zurück.

Wenig Sinn macht dagegen das Fakt:

change_state_transition (*X*).,

da in $\text{DATALOG}_f^{\neg}$ aufgrund des Einmalzuweisungsprinzips eine Ausgabe nicht über den gleichen Parameter erfolgen kann wie die Eingabe und ein Fakt, das nur eine Eingabe erhält, überflüssig ist. ◇

Fakten, die keine Schnittstelle zum Datenspeicher darstellen, sind folglich entweder daran zu erkennen, daß sie Konstanten enthalten, oder ein Parameter gleichen Namens mehr als einmal in der Parameterliste vorkommt. Im folgenden werden sie als Programmfakten bezeichnet.

Mischformen, also Fakten, die sowohl eine Schnittstelle zum Datenspeicher darstellen, als auch Konstanten enthalten, bzw. Parameter, die Eingaben vom Programm erhalten, sind aufgrund der Ununterscheidbarkeit der Fälle ausgeschlossen.

Definition 3.2 *Konsistenzbedingungen für Fakten eines $DATALOG_f^{\neg}$-Programms*

Die Fakten eines $\text{DATALOG}_f^{\neg}$-Programms müssen folgende Bedingungen erfüllen, um konsistent zu sein:

- Für ein Fakt, das eine Schnittstelle zum Datenspeicher darstellt, muß gelten: alle Parameter sind Variablen und es gibt keine zwei Parameter mit dem gleichen Namen.
- Für ein Programmfakt muß gelten: wenn ein Parameter eine Variable ist, dann besitzt das Fakt mindestens einen weiteren Parameter mit gleichem Namen. ◇

3.1.3 Normalform von DATALOG$_f^{\neg}$-Programmen

Während sich die letzten beiden Abschnitte mit Konsistenzbedingungen für DATALOG$_f^{\neg}$-Programme befaßt haben, um Voraussetzungen für die Berechenbarkeit der Ausgangsspezifikation zu schaffen, geht es in diesem Abschnitt um eine Normalisierung von DATALOG$_f^{\neg}$-Programmen als Grundlage für die weitere Übersetzung in einen Datenflußgraphen.

3.1.3.1 Mathematische Ausdrücke als Literale

DATALOG$_f^{\neg}$ erlaubt mathematische Ausdrücke im Rumpf von Regeln, wie z.B.

$$X \ is \ (Y * Z) / - (Y + Z).$$

Diese Ausdrücke können entweder durch Unifikation über Termstrukturen behandelt werden [Ullman 88], oder durch die in realen DATALOG$_f^{\neg}$-Interpretern üblichere Implementierung als Bibliotheksfunktionen. Bottom-Up Verfahren berücksichtigen darüberhinaus solche Konstruktionen zumeist nicht oder lediglich sehr eingeschränkt. Da derartige Ausdrücke im gegebenen Anwendungsfall häufig auftreten, müssen sie behandelt werden können.

Mathematische Ausdrücke lassen sich auch in Präfixschreibweise darstellen, z.B. als:

$$is \ (X, /(*(Y, Z), -(+(Y, Z)))).$$

In DATALOG$_f^{\neg}$ kann dies als Literal verstanden werden, dessen Prädikat als Systemprädikat implementiert ist, und das Funktionsaufrufe als Parameter besitzt (wobei in diesem Beispiel als Parameter eines Funktionsaufrufs wieder Funktionsaufrufe aufreten).

Jedes Systemprädikat liefert als "Funktionswert" *success (true)* oder *failure (false)*, je nachdem, ob die Operation erfolgreich ausgeführt werden konnte, oder ob beispielsweise eine Typverletzung, ein Bereichsüberlauf oder eine Division durch 0 aufgetreten ist.

Generell ist in DATALOG$_f^{\neg}$ eine Infixschreibweise üblich und wird im folgenden bei der (informellen) Darstellung arithmetischer Ausdrücke auch beibehalten. Für die Umsetzung in einen Datenflußgraphen müssen die Ausdrücke jedoch als Literale mit Systemprädikaten (in Präfxschreibweise) interpretiert werden.

3.1.3.2 Constraints als Literale

DATALOG$_f^{\neg}$ erlaubt Constraints:

- arithmetische Vergleiche und
- Unifikationen (d.h. Vergleiche auf Termstrukturen, sowie Zuweisungen von Werten oder Termstrukturen an Variablen).

Es gilt wie bei arithmetischen Ausdrücken, daß zusätzlich zum Vergleichs- oder Unifikationssymbol Operationen oder Funktionsaufrufe auftreten dürfen. Bei Constraints dürfen diese im Unterschied zu Zuweisungen jedoch auch auf beiden Seiten stehen.

Sind Funktionen über den Termaufbau implementiert, kann die Auswertung von Constraints sehr aufwendig werden. Daher wird die Behandlung in dieser Arbeit darauf beschränkt, daß implizit davon ausgegangen wird, daß Constraints wie mathematische Operationen als Systemprädikate implementiert sind.

Beispiele:

i) $X < Y - 1$.
ii) $X + 1 < Y$.
iii) $X + 1 \leq Y - 1$. ◇

Auch Constraints können in Präfixschreibweise dargestellt werden. Die Relationszeichen (Vergleichssymbole) werden in $\text{DATALOG}_f^{\neg}$ ebenfalls durch Systemprädikate implementiert.

Beispiele:

i) $X < Y - 1$ wird zu: $<(X, -(Y, 1))$.
ii) $X + 1 < Y$ wird zu: $<(+(X, 1), Y)$.
iii) $X + 1 \leq Y - 1$ wird zu: $\leq(+(X, 1), -(Y, 1))$. ◇

Jedes Systemprädikat für ein Vergleichsymbol liefert als "Funktionswert" *success (true)* oder *failure (false)* für das Ergebnis des Vergleichs.

3.1.3.3 Extraktion von Funktionen aus der Parameterliste

Als Parameter eines Regelkopfs oder von Literalen im Rumpf einer Regel dienen für gewöhnlich Konstanten und Variablen. Ein $\text{DATALOG}_f^{\neg}$-Programm läßt jedoch auch Operationen und Funktionsaufrufe als Parameter zu.

Beispiel: Operation

$f(X, Y, X + Y)$ berechnet $X + Y$ aus X und Y, bzw. X und Y aus $X + Y$. ◇

Beispiel: Funktionsaufrufe

$f(X, exp(X, 4))$ berechnet X^4, bzw. X aus X^4. ◇

Da Operationen, mathematisch betrachtet, ebenfalls Funktionen sind, wird im folgenden nur noch der Begriff Funktion verwendet.

In einer Implementierungsform der Logikprogrammierung, deren Auswertungsverfahren rein auf Unifikation basiert, kann eine Funktion nur über den im Programm vollständig definierten Aufbau einer Termstruktur "ausgewertet" werden [Ullman 88].

Beispiel:

$int(0)$.
$int(succ(X)) \leftarrow int(X)$.
$sum(X, 0, X) \leftarrow int(X)$.
$sum(X, succ(Y), succ(Z)) \leftarrow sum(X, Y, Z)$.

Dabei ist auch *succ* nicht arithmetisch, sondern über eine Termstruktur definiert:

Der Wert 1 wird dargestellt als $succ(0)$
Der Wert n wird dargestellt als $\underbrace{succ(succ(...succ}_{n-mal}(0)...))$ ◇

Dies führt offensichtlich zu komplexen Darstellungsformen und damit zu einer Ausführungsszeit, die für Echtzeitanwendungen indiskutabel ist. Die Ausführung des Datenflußgraphen wird daher auf Berechnungen anstelle von Unifikationen basieren.

Daher müssen Funktionen direkt ausgewertet werden können, beispielsweise über die Implementierung einer Funktionsbibliothek, oder bei der Ausführung auf einem Datenflußrechner direkt durch die Hardware.

Die Verwendung von Funktionen als Parameter von Literalen bedeutet darüberhinaus eine Erweiterung des Abarbeitungsprinzips dieser Literale: statt Werte als Übergabe an die Variablen zu erwarten oder den Wert einer Konstanten direkt zu verwenden, müssen zusätzlich Operationen durchgeführt werden können, bevor die eigentliche Abarbeitung (z.B. Und-Verknüpfung der Eingabewerte) beginnt.

Da die Literale zu den Knoten eines Datenflußgraphen werden sollen und eine mehrstufige Funktionalität den Prinzipien von Datenflußrechnerarchitekturen widerspricht, sollte eine strukturellen Aufteilung dieser Funktionalität geschehen, bevor der Datenflußgraph erzeugt wird.

Beispiel:

> $f(X+1) \leftarrow g(X).$
>
> Dient X als Ausgabe von g, so wird X bei Weitergabe durch f um 1 erhöht. Dient X jedoch als Eingabe an f und dadurch auch als Eingabe von g, so muß X um 1 erniedrigt werden können. ◇

Eine Funktion, die als Parameter eines Literals verwendet wird, kann jedoch auch in einem eigenen Literal stehen, ohne die deklarative Bedeutung der Regel zu verändern, wenn bekannt ist, ob an ihre Parameter eine Eingabe erfolgt, oder ob über sie eine Ausgabe erzeugen soll.

Beispiel:

> Obige Regel läßt sich, wenn X Ausgabeparameter von f und g ist, mit
>
> $f(Y) \leftarrow g(X),\ Y\ \mathit{is}\ X{+}1.$
>
> und, wenn X Eingabeparameter ist, mit
>
> $f(Y) \leftarrow X\ \mathit{is}\ Y{-}1,\ g(Y).$
>
> schreiben. ◇

Beispiel:

> Für folgenden Fall gilt entsprechendes:
>
> $f(X) \leftarrow g(X{+}1).$.
>
> Die Regel läßt sich, wenn X Ausgabeparameter von f und g ist, schreiben als:
>
> $f(X) \leftarrow g(Y),\ X\ \mathit{is}\ Y{-}1.$
>
> und, wenn X Eingabeparameter ist, als:
>
> $f(X) \leftarrow Y\ \mathit{is}\ X{+}1,\ g(Y).$ ◇

Vor einer Analyse der Parametermodi ist jedoch im allgemeinen nicht bekannt, ob die Funktion an der Stelle eines Eingabe- oder eines Ausgabeparameters steht. Daher muß

eine allgemeine Schreibweise gefunden werden, die beide Fälle zusammenfaßt. Dafür ließe sich die Einführung eines Systemprädikats mit zwei verschiedenen Bindungsmustern denken (analog zu bereits existierenden Bibliotheksfunktionen in DATALOG$^{\neg}_{f}$), das in beiden Bedeutungen existieren kann.

Beispiel:

$f(X+1) \leftarrow g(X).$

wird dabei zu

$f(Y) \leftarrow p_{+}(Y, X, 1),\ g(X).$

mit $p_{+} \in MP \subset Pred.$ ◇

Da in diesem Fall nicht entscheidbar ist, ob die Eingabe an Y erfolgt und die Ausgabe durch X, oder umgekehrt, dürfen aus der Reihenfolge der Literale in diesem Fall keine Annahmen zur Bestimmung der Parametermodi abgeleitet werden (wie dies beispielsweise bei Verwendung von SIP-Verfahren geschieht).

Die Beispiele zeigen, daß es möglich ist, eine Funktion aus der Parameterliste eines Literals zu entfernen, indem sie durch eine neue, bisher noch nicht verwendete Variable ersetzt wird und zusätzlich im Rumpf der Regel ein neues Literal eingefügt wird, das ein Systemprädikat darstellt und sowohl den Funktionsaufruf als auch die Zuweisung des Funktionswertes an die neu erzeugte Variable realisiert. Die deklarative Bedeutung der Regel bleibt dabei erhalten.

Die Realisierung der benötigten Systemprädikate ist jedoch nicht in jedem Fall trivial: Während eine Funktion, die in der Parameterliste des Regelkopfs an der Stelle eines Eingabeparameters, oder in der Parameterliste eines Literals im Rumpf der Regel an der Stelle eines Ausgabeparameters steht, sich trivial in ein gesondertes Literal umsetzen läßt, benötigt die Umsetzung im jeweils umgekehrten Fall (im Kopf Ausgabemodus bzw. im Rumpf Eingabemodus), die Umkehrfunktion der angegebenem Funktion.

Für die Funktion "*incr*" ist z.B. die Umkehrfunktion "*decr*" und umgekehrt.

Jedoch treten bei der Bildung von Umkehrfunktionen folgende Probleme auf:

- Wenn die Funktion nicht surjektiv ist, ist die Umkehrfunktion nur für einen eingeschränkten Wertebereich der Funktion sinnvoll definiert.
- Wenn die Funktion nicht injektiv ist, liefert die Umkehrfunktion nicht in jedem Fall ein eindeutiges Ergebnis, d.h. sie ist keine Funktion mehr im eigentlichen Sinne. Konkret bedeutet dies, daß nicht wie bei einer (mathematischen) Funktion nur eine Ausgabe, sondern mehrere, möglicherweise sogar unendlich viele Ausgaben aus einer Eingabe entstehen können. Bei solchen "Umkehrfunktionen" handelt es sich im mathematischen Sinne allenfalls um Umkehrrelationen und nicht mehr um Funktionen.

Beispiel:

Das Literal

komplexe_zahl $(X, Y, X+Y)$

mit dem Systemprädikat komplexe_zahl kann entweder bedeuten, daß die komplexe Zahl $X+iY$ aus den reellen Zahlen X und Y erzeugt wird, oder daß X und Y, die zusammengenommen einen gegebenen Wert aus dem Bereich der komplexen Zahlen ergeben, aus dieser errechnet und einzeln ausgegeben werden.

Das Systemprädikat liefert sowohl als Funktion, wie auch als Umkehrfunktion ein eindeutiges Ergebnis.

In den meisten Fällen sind jedoch Umkehrfunktionen nicht eindeutig, d.h. zu einem Eingabwert kann es verschiedene, alternative Ausgabewerte geben:

addiere $(X, Y, X+Y)$

kann entweder bedeuten, daß $X+Y$ aus X und Y berechnet wird, oder daß alle Möglichkeiten für X und Y, die addiert den an den Parameter $X+Y$ eingegebenen Wert ergeben, berechnet werden sollen. ◇

Aus nicht eindeutigen Umkehrrelationen können gegebenenfalls beliebig große Datenmengen entstehen, die im Anwendungsfall, der Steuerung autonomer Systeme, zu unsicheren Ausgaben aufgrund nicht eindeutiger Ergebnisse führen können. Daher werden im Rahmen dieser Arbeit Umkehrrelationen, die keine Funktionen im mathematischen Sinne sind, ausgeschlossen.

Die Problematik der Behandlung solcher Umkehrrelationen tritt bei reinen Bottom-Up Verfahren in der gleichen Form auf, wenn Operationen, bzw. Funktionen in der Parameterliste von Literalen zugelassen werden. Modifizierte Bottom-Up Verfahren arbeiten an diesen Stellen lokal mit Top-Down Prinzipien, wie z.B. die in [Bader 90] vorgestellte Ausführungsmethode.

Als Voraussetzung für die Entfernung von Funktionen aus der Parameterliste eines Literals und Umsetzung in gesonderte Literale, ist es notwendig, daß es für alle Funktionssymbole $f \in \mathit{Fun}$ ein entsprechendes Systemprädikat $p_f \in \mathit{MP}$ gibt, wobei gelten muß:

ist $f(t_1, ..., t_n) \in \mathit{Fun}$, $n \in \mathbb{N}$, so ist $p_f(t_0, t_1, ..., t_n) \in \mathit{MP}$,
und: $t_0 = f(t_1, ..., t_n)$.

Es hat also entweder t_0 den Modus Ausgabe und $t_1, ..., t_n$ den Modus Eingabe (Verwendung als Funktion), oder t_0 den Modus Eingabe und $t_1, ..., t_n$ den Modus Ausgabe (Verwendung als Umkehrfunktion).

Bei dem in dieser Arbeit vorgestellten Ausführungsmodell handelt es sich jedoch um kein reines Bottom-Up Verfahren, bei dem alle Parameter eines Regelkopfs der Ausgabe dienen. Es ist also nicht bekannt, ob ein Systemprädikat nun als Funktion oder als Umkehrfunktion fungieren muß. Dies bestimmt sich erst durch die Analyse der Parametermodi (siehe Kapitel 3.3.4.

Daher muß die Implementierung von Funktionen als Systemprädikate in einer Bibliothek folgende Bedingungen erfüllen: je nach Bindungsmuster der Parameter muß das Systemprädikat entweder die Funktion oder die Umkehrfunktion berechnen können.

Jedes Systemprädikat liefert als “Funktionswert” *success (true)* wenn die Funktion oder Umkehrfunktion erfolgreich ausgeführt werden konnte, *failure (false)* wenn nicht. Wenn die Ausführung einer Umkehrfunktion nicht implementiert ist, oder nicht sinnvoll über dem gegebenen Wertebereich berechenbar ist, muß sie ebenfalls *failure (false)* zurückliefern, um das Scheitern der Ausführung zu melden.

Definition 3.3 *Konsistenzbedingungen für Funktionen eines $DATALOG_f^{\neg}$-Programms*

Die Funktionen eines $\text{DATALOG}_f^{\neg}$-Programms müssen folgende Bedingungen erfüllen, um konsistent zu sein:

- die Umkehrfunktion muß existieren,

- die Eingaben müssen im gültigen Wertebereich liegen,

- bei eindeutiger Eingabe muß ein eindeutiges Ergebnis berechenbar sein.

Bemerkung:

In der Logikprogrammierung gilt das Prinzip der Einmalzuweisung.

Es ist zwar möglich durch Oder-Verknüpfung einer Variablen mehr als einen Wert zuzuweisen. Werte können auch durch das Scheitern einer Regel ungültig werden. Es ist jedoch nicht möglich einen Wert direkt zu verändern, d.h. durch einen anderen zu ersetzen.

Das heißt folglich, daß eine Funktion, die für bestimmte Parameter Werte erhält, für diese keine Ausgabewerte erzeugen kann. Es kann also kein Parameter zur gleichen Zeit sowohl Eingabe- als auch Ausgabemodus besitzen. ◇

Anstelle einfacher Funktionen können auch zusammengesetzte mathematische Ausdrücke als Parameter von Literalen auftreten.

Eine wiederholte Hintereinanderausführung der vorgestellten Methode ermöglicht die Behandlung dieser Fälle (unter der Bedingung, daß die entsprechenden Systemprädikate existieren müssen).

Beispiel:

$f(exp((X+1)*Y,2)) \leftarrow g(X,Y).$

wird dabei zunächst zu

$f(Z) \leftarrow g(X,Y),\ p_{exp}((X+1)*Y,2).$

mit $p_{exp} \in MP \subset Pred$

und anschließend zu

$f(Z) \leftarrow g(X,Y),\ p_{+}(Z2,X,1),\ p_{*}(Z1,Z2,Y),\ p_{exp}(Z1,2).$ ◇

Bemerkung:

Es gilt jedoch für die meisten Operationen, also auch schon für so einfache Operationen wie + und *, daß die Umkehrung keine Funktion mehr darstellt. In umso größerem Maße trifft das für Ausdrücke zu, die mehr als ein Operationszeichen enthalten. ◇

3.1.3.4 Extraktion von Listen aus der Parameterliste

Listen (z.B. $[X|Xs]$) als Parameter eines Literals werden in der Logikprogrammierung durch Unifikation ausgewertet.

Die Unifikation ist bei Listen im Unterschied zu Funktionen vergleichsweise einfach. Da die spätere Ausführung des Datenflußgraphen jedoch auf Berechnungen anstelle von Unifikationen basiert, werden Listen mit Operatoren ("[", "|", "]") als Funktionen aufgefaßt. Damit liegt nahe, diese ebenfalls in eigene Literale zu transformieren.

Beispiel: Einsortieren eines neuen Listenelements in die Liste nach der Größe

$f([X|Xs],Y,[Y|X|Xs]) \leftarrow Y < X.$
$f([X|Xs],Y,[X|Zs]) \leftarrow f(Xs,Y,Zs).$

Die beiden Regeln lassen sich ohne Probleme auch schreiben als:

$f(Xl,Y,Zl) \leftarrow X\ is\ \mathsf{head}(Xl),\ Xs\ is\ \mathsf{rest}(Xl),\ Y < X,\ Zl\ is\ [Y|X|Xs].$
$f(Xl,Y,Zl) \leftarrow X\ is\ \mathsf{head}(Xl),\ Xs\ is\ \mathsf{rest}(Xl),\ f(Xs,Y,Zs),\ Zl\ is\ [X|Zs].$

◇

Bei Listen ist die Bildung einer “Umkehrfunktion” mit den Funktionen *head* (erstes Element der Liste ausgeben) und *rest* (das erste Element der Liste abschneiden) ohne weiteres möglich.

Als allgemeine Form eignet sich das Systemprädikat $p_{[]}$(*Xlist,X,Xrest*) (wobei *Xlist, Xrest* Listen sind und X eine Variable ist). $p_{[]}$ übernimmt als Funktion die Konstruktion einer Liste aus einer Variable und einer Liste, sowie als Umkehrfunktion das Zerteilen einer Liste in Kopfelement und Restliste.

3.1.3.5 Elimination von Konstanten aus der Parameterliste

Konstanten lassen sich als nullstellige Funktionen auffassen. Da hier jedoch keine Operationen oder Funktionsaufrufe durchgeführt werden müssen, um einen Ergebniswert zu erhalten, ist die Notwendigkeit einer Extraktion von Konstanten zunächst nicht gegeben.

Konstanten im Kopf einer Regel dienen als Ausgabe der Regel: da sie keinem in einem Rumpfliteral verwendeten Parameter über einen identischen Namen zugeordnet werden können, können sie keine Eingabe darstellen. Konstanten gleichen Wertes als Parameter von Rumpfliteralen sind von ihnen unabhängige Parameter, da (trotz der Identität von Werten) keine Werteübergabe zwischen Konstanten erfolgen kann.

Analog kann eine Konstante im Rumpf nicht als Ausgabe an einen Parameter des Regelkopfs oder an einen anderen im Rumpf vorkommenden Parameter verwendet werden, da eine Zuordnung hier ebenso nicht möglich ist. Es bleibt also nur die Verwendung als Eingabe an das Rumpfliteral selbst.

Im Kopf können Konstanten stehen bleiben: der Kopf wird im zu erzeugenden Graphen in einen Knoten umgesetzt, in dem Konstanten als feste Werte eingetragen werden können. Ein Rumpfliteral, das ein Systemprädikat darstellt, wird ebenfalls durch einen eigenen Knoten verkörpert, auch hier kann die Konstante direkt aufgenommen werden. Nicht so dagegen, wenn es sich bei dem Rumpfliteral um den (evtl. negierten) Aufruf einer weiteren Regel handelt. Diese Aufrufe werden im Graphen lediglich durch Kanten zu den Knoten, die die aufgerufenen Regelköpfe repräsentieren, dargestellt. Es muß daher ein eigener Knoten geschaffen werden, der die Konstante enthält und diese als Eingabe an das Rumpfliteral, in dem die Konstante zuvor aufgetreten ist, weitergibt.

Als Vorbereitung zur Erzeugung des Graphen ist daher eine Elimination von Konstanten aus der Parameterliste von Rumpfliteralen, die keine Systemprädikate sind, notwendig. Die Elimination geschieht mittels Ersetzung der Konstante durch eine neue, eindeutige Variable und die Erzeugung eines eigenen Literals, in dem die Zuweisung der Konstante an diese Variable vorgenommen wird. Das Literal wird im Rumpf der Regel eingefügt.

Beispiel:

$f(X) \leftarrow g(23, X)$

wird also zu

$f(X) \leftarrow Y = 23,\ g(Y, X).$ ◇

3.1.3.6 Elimination doppelter Variablen aus der Parameterliste

In Parameterlisten von Literalen können Parameter mehr als einmal den gleichen Namen haben. Während dies bei Rumpfliteralen keine weiteren Konsequenzen hat, würde

es im Kopf von Regeln bei der Vergabe von Identifikatoren (Kapitel 3.3.2) zu Schwierigkeiten führen. Daher werden, wenn mehrere Parameter in der Parameterliste eines Regelkopfs den gleichen Namen besitzen, alle Parameter, bis auf einen, durch einen neuen, noch nicht in der Regel auftretenden Parameter ersetzt. Zusätzlich werden in der Regel Literale eingefügt, die den jeweils neuen Parameter mit dem alten unifizieren.

Beispiel:

$\mathsf{max}(X, Y, X) \leftarrow X > Y$

wird zu:

$\mathsf{max}(X, Y, X1) \leftarrow X1 = X, X > Y.$ ◇

Bemerkung: Dieses Verfahren entspricht im wesentlichen einer Rektifizierung von Regelköpfen, wie sie in [Ullman 88] vorgestellt wird, mit dem Unterschied, daß diese hier nur für mehrfach vorkommende Variablen angewendet wird.

3.1.3.7 Normalisierung eines DATALOG$_f^{\neg}$-Programms

Definition 3.4 *Normalform von Regeln*

Eine Regel ist in Normalform, wenn gilt:

- es treten keine Operationen oder Funktionsaufrufe in der Parameterliste eines Literals auf,
- es treten keine Listenkonstruktionen in der Parameterliste eines Literals auf,
- es treten keine Konstanten in der Parameterliste eines Literals im Rumpf einer Regel (bzw. in der Abfrage), das kein Systemprädikat ist, auf,
- es treten keine Parameter gleichen Namens in der Parameterliste eines Regelkopfs auf.

Ein Programm ist in Normalform, wenn jede Regel einschließlich der Abfrage in Normalform ist. ◇

Der Algorithmus zur Normalisierung eines DATALOG$_f^{\neg}$-Programms lautet wie folgt:

Definition 3.5 *Algorithmus für die Normalisierung eines DATALOG$_f^{\neg}$-Programms*

Für eine Regel $P_0 \leftarrow P_1, ..., P_n.$, $n \in \mathbb{N}$, mit $P_i \equiv p(t_1, ..., t_{m_i})$ für $i \in \{0, ..., n\}$, $m_i \in \mathbb{N}$, gelte, wenn (für $j \in \{1, ..., m_i\}$):

i) $t_j \equiv f(l_1, ..., l_u) \in T,\ l_1, ..., l_u \in T,\ u \in \mathbb{N},\ f \in \mathit{Fun}$

dann werde t_j ersetzt durch eine neue, eindeutige Variable X_{ij} und ein neues Literal P werde erzeugt, wobei $p_{[]} \in MP$ mit geeigneter Anzahl von Parametern existieren muß und

$P_{ij} \equiv p_f(X_{ij}, l_1, ..., l_u).$

ii) $t_j \equiv [l_1|...|l_u] \in Z_{\mathcal{P}},\ l_1, ..., l_u \in Z_{\mathcal{P}},\ u \in \mathbb{N}$

dann werde t_j ersetzt durch eine neue, eindeutige Variable X_{ij} und ein neues Literal P werde erzeugt, wobei $p_{[]} \in MP$ mit geeigneter Anzahl von Parametern existieren muß und

$P_{ij} \equiv p_{[]}(X_{ij}, t_1, ..., t_u).$

iii) $t_j \equiv a$, a Konstante für $i \geq 1$ und $p \notin Pred$:

dann werde t_j ersetzt durch eine neue, eindeutige Variable X_{ij} und ein neues Literal P erzeugt, wobei

$P_{ij} \equiv X_{ij} = a$.

iv) $t_{j'} = t_j$ für ein $j' \in \{1, ..., m_i\}$, $j \neq j'$:

dann werde t_j ersetzt durch eine neue, eindeutige Variable X_{ij} und ein neues Literal P erzeugt, wobei

$P_{ij} \equiv X_{ij} = t_j$.

P_i werde dann zu $p(t_1, ..., t_{j-1}, X_{ij}, t_{j+1}, ..., t_{m_i})$.

Die Regel werde zu $P_0 \leftarrow P_{ij}, P_1, ..., P_l$.

Der Algorithmus ist anzuwenden, bis keine Ersetzung mehr möglich ist. ◇

Satz 3.6

Der Algorithmus überführt ein $\text{DATALOG}_f^{\neg}$-Programm in Normalform. ◇

Beweis:

Der Algorithmus führt, gegebenenfalls unter Wiederholung der Einzelschritte, offensichtlich sämtliche Regeln, die die Bedingungen der Normalform verletzen, in äquivalente Regeln über, die der Normalform nicht widersprechen. ◇

Folgerung 3.7

Nach Anwendung des Verfahrens sind alle in Literalen vorkommenden Parameter entweder Konstanten oder Variablen. ◇

3.2 Der Strukturgraph

Der im folgenden vorgestellte Strukturgraph entspricht im Kern dem aus der Logikprogrammierung bekannten Operatorbaum [Ullman 85].

Obwohl es sich bei den verwendeten Graphen nicht um Bäume handelt, werden einige Sprachkonventionen, die für Bäume benutzt werden, in analoger Bedeutung auch hier verwendet. Sie werden im Anhang A.5 formal definiert.

Da die Abfrage im vorliegenden Anwendungsfall statisch und vor der Ausführung bekannt ist, kann sie in das Programm integriert und gemeinsam mit dem Programm in einen Graphen transformiert werden.

Zu einem Programm gebe es genau eine Abfrage. Der Rumpf der Abfrage beinhalte die Menge der Einzelabfragen. Im Anwendungsfall stellen diese die Anforderung der benötigten Steuerungsausgaben für das autonom-mobile System dar.

3.2.1 Definition des Strukturgraphen

Aus jedem normalisierten $\text{DATALOG}_f^{\neg}$-Programm, das den in Abschnitt 3.1 geforderten Konsistenzbedingungen genügt, läßt sich durch einfache Syntaxanalyse der Strukturgraph ableiten.

Es werden dafür folgende Knotentypen eingeführt, die die verschiedenen $\text{DATALOG}_f^{\neg}$-Elemente repräsentieren:

IFACT: Ein IFACT-Knoten steht für ein Fakt, das einen Teil der Schnittstelle zum Datenspeicher realisiert (I steht für Interface).

FACT: Ein FACT-Knoten steht für ein Fakt, das Konstanten enthält, oder Eingaben vom Programm selbst erhält.

FUNC: Bei einem FUNC-Knoten handelt es sich um einen Knoten, der den Aufruf eines Metaprädikates, also einer vordefinierten (Bibliotheks-)Funktion realisiert.

AND: Ein AND-Knoten repräsentiert den Kopf einer Regel und realisiert damit die konjunktive Verknüpfung, entsprechend einem Join, der Ergebnisse mehrerer Zweige des Graphen (für die Definition eines Join siehe Anhang A.4).

OR: Ein OR-Knoten sammelt die Ergebnisse alternativer Zweige im Graphen und realisiert damit eine Vereinigung.

QUERY: Der QUERY-Knoten stellt die Abfrage dar, bzw. den Knoten im Graphen, an dem die Ergebnisse ausgegeben werden.

NOT: Für negierte Literale im Rumpf einer Regel wird ein spezieller NOT-Knoten eingeführt.

Definition 3.8 *Strukturgraph*

Ein Strukturgraph $SG_{\mathcal{L}}$ zu einem $\text{DATALOG}_f^{\neg}$-Programm $\mathcal{L}$ mit Regeln $P_{i,k}^0 \leftarrow P_{i,k}^1, ..., P_{i,k}^{l_{i,k}}$. und einer Abfrage $P_{0,1}^1, ..., P_{0,1}^{l_{0,1}}$. (wobei $n \in \mathbb{N}$, $i, i' \in \{0, ..., n\}$, $m_i, m_{i'} \in \mathbb{N}$, $k \in \{1, ..., m_i\}$, $k' \in \{1, ..., m_{i'}\}$, $l_{i,k} \in \mathbb{N}$, $r \in \{1, ..., l_{i,k}\}$, siehe Def. 2.5 und Def. 2.6) sei definiert als $SG_{\mathcal{L}} = (V, E)$, mit

i) V sei eine endliche, nicht leere Menge von Knoten, definiert als

$V := V^\wedge \cup V^\vee \cup V^\neg \cup V^\mathcal{R}$, wobei

$$- V^\wedge := \{v^\wedge_{i,k} \mid \exists P^0_{i,k}\} \cup \{v^\wedge_{0,1} \text{ für den leeren Kopf der Abfrage}\}, \quad (1)$$

$$- V^\vee := \{v^\vee_i \mid \text{es existiert außer } P^0_{i,1} \text{ mindestens noch } P^0_{i,2}\}, \quad (2)$$

$$- V^\mathcal{R} := \{v^r_{i,k} \mid \exists P^r_{i,k}, P^r_{i,k} \text{ hat Prädikat } p \in \mathit{Pred}\}, \quad (3)$$

$$- V^\neg := \{v^{r\,\neg}_{i,k} \mid \exists P^r_{i,k} \text{ als negatives Atom und } \exists P^0_{i',1} \text{ mit } P^0_{i',1} \simeq P^r_{i,k}\} \cup$$
$$\{v^{r\,\neg}_{i,k} \mid \exists P^r_{i,k} \text{ als negatives Atom und } P^r_{i,k} \text{ hat Prädikat } p \in \mathit{Pred}\}, \quad (4)$$

mit $i, i' \in \{0, ..., n\}$, $k \in \{1, ..., m_i\}$, $r \in \{1, ..., l_{i,k}\}$.

ii) $E \subset V \times V$ sei eine endliche nicht leere Menge gerichteter Kanten, definiert als

$E := E^\wedge \cup E^\vee \cup E^\neg$, wobei

$$- E^\wedge := \{e_{v^\wedge_{i,k}, v^\vee_{i'}} \mid \exists v^\wedge_{i,k} \in V^\wedge \wedge \exists v^\vee_{i'} \in V^\vee \wedge \nexists v^{r\,\neg}_{i,k} \in V^\neg \text{ und}$$
$$\exists P^r_{i,k}, P^0_{i',1} \text{ mit } P^r_{i,k} \simeq P^0_{i',1}\} \cup \quad (1)$$
$$\{e_{v^\wedge_{i,k}, v^\wedge_{i',1}} \mid \exists v^\wedge_{i,k} \in V^\wedge \wedge \exists v^\wedge_{i',1} \in V^\wedge \wedge \nexists v^\vee_{i'} \in V^\vee \wedge \nexists v^{r\,\neg}_{i,k} \in V^\neg \text{ und}$$
$$\exists P^r_{i,k}, P^0_{i',1} \text{ mit } P^r_{i,k} \simeq P^0_{i',1}\} \cup \quad (2)$$
$$\{e_{v^\wedge_{i,k}, v^r_{i,k}} \mid \exists v^\wedge_{i,k} \in V^\wedge \wedge \exists v^r_{i,k} \in V^\mathcal{R} \wedge \nexists v^{r\,\neg}_{i,k} \in V^\neg\} \cup \quad (3)$$
$$\{e_{v^\wedge_{i,k}, v^{r\,\neg}_{i,k}} \mid \exists v^\wedge_{i,k} \in V^\wedge \wedge \exists v^{r\,\neg}_{i,k} \in V^\neg\}, \quad (4)$$

$$- E^\vee := \{e_{v^\vee_i, v^\wedge_{i,k}} \mid \exists v^\vee_i \in V^\vee \wedge \exists v^\wedge_{i,k} \in V^\wedge\ \forall k \in \{1, ..., m_i\}\}, \quad (5)$$

$$- E^\neg := \{e_{v^{r\,\neg}_{i,k}, v^\vee_{i'}} \mid \exists v^{r\,\neg}_{i,k} \in V^\neg \wedge \exists v^\vee_{i'} \in V^\vee \text{ und}$$
$$\exists P^r_{i,k}, P^0_{i',1} \text{ mit } P^r_{i,k} \simeq P^0_{i',1}\} \cup \quad (6)$$
$$\{e_{v^{r\,\neg}_{i,k}, v^\wedge_{i',1}} \mid \exists v^{r\,\neg}_{i,k} \in V^\neg \wedge \exists v^\wedge_{i',1} \in V^\wedge \wedge \nexists v^\vee_{i'} \in V^\vee \text{ und}$$
$$\exists P^r_{i,k}, P^0_{i',1} \text{ mit } P^r_{i,k} \simeq P^0_{i',1}\} \cup \quad (7)$$
$$\{e_{v^{r\,\neg}_{i,k}, v^r_{i,k}} \mid \exists v^{r\,\neg}_{i,k} \in V^\neg \wedge \exists v^r_{i,k} \in V^\mathcal{R}\}, \quad (8)$$

mit $i, i' \in \{0, ..., n\}$, $k \in \{1, ..., m_i\}$, $r \in \{1, ..., l_{i,k}\}$.

$P \simeq Q$ stehe dabei für: die Literale P und Q sind unifizierbar. ◇

Bemerkung: $V^\wedge$ enthält alle AND-, QUERY-, FACT- und IFACT-Knoten. $V^\vee$ enthält alle OR-Knoten, $V^\mathcal{R}$ alle Knoten für Systemprädikate und $V^\neg$ alle NOT-Knoten. $E^\wedge$ enthält alle Kanten von AND-Knoten zu ihren jeweiligen Unterknoten (die verschiedenen Typs sein können), $E^\vee$ alle Kanten von OR-Knoten zu ihren Unterknoten (die nur AND- und FACT-Knoten sein können und damit aus $V^\wedge$ sind), und $E^\neg$ alle Kanten von NOT-Knoten zu ihren Unterknoten (die wieder verschiedenen Typs sein können).

Das Schema der Erzeugung des Strukturgraphen ist dabei folgendes:

- Jeder Kopf einer Regel wird zu einem AND-Knoten (siehe Def. 3.8 i.1).
- Fakten werden durch FACT/IFACT-Knoten dargestellt (i.2).
- Regeln und Fakten mit gleichem Kopf, d.h. mit gleichem Prädikat und gleicher Parameteranzahl, stellen disjunktive Berechnungswege dar. Um diese Berechnungswege verknüpfen zu können, wird ein OR-Knoten erzeugt (i.2). Von diesem OR-Knoten gehen dann Kanten zu den AND-, FACT und IFACT-Knoten (ii.5).

- Metaprädikate und Prädikate, die Vergleiche oder Zuweisungen repräsentieren, werden durch FUNC-Knoten repräsentiert (i.3). Für jedes Prädikat wird eine Kante vom AND-Knoten für den Kopf der Regel zum FUNC-Knoten erzeugt (ii.3).
- Alle anderen Literale im Rumpf einer Regel sind "Aufrufe" von weiteren Regeln oder Fakten, d.h. sie lassen sich mit dem Kopf mindestens einer Regel oder mit mindestens einem Fakt des Programms unifizieren. Folglich gehen für alle diese Literale Kanten vom AND-Knoten für den Kopf der Regel zu den Knoten, die die unifizierbaren Regeln oder Fakten repräsentieren (i.1, i.2). Dies ist bei Anwesenheit von mehreren, alternativen Regeln der OR-Knoten, ansonsten der jeweilige AND- oder FACT-Knoten.
- Für Literale im Rumpf einer Regel, die negierte Atome sind, wird zusätzlich ein NOT-Knoten eingefügt (i.4). Die Kantenverbindung geht nun von dem AND-Knoten für den Kopf der Regel zu dem NOT-Knoten (ii.4) und von dort zu dem Knoten, der das positive Atom repräsentiert (ii.6-8).
- Der Graph besitzt genau eine Wurzel, nämlich die Abfrage, die an das Programm gestellt wird und die mit dem Programm in den Graphen übersetzt wird. Sie wird durch den QUERY-Knoten repräsentiert. Dabei wird nicht die Abfrage an sich in den QUERY-Knoten umgesetzt, sondern der (leere) Regelkopf der Abfrage (i.1). Der Rumpf der Abfrage, bestehend aus Einzelabfragen, wird wie der Rumpf jeder anderen Regel behandelt.

Bemerkung:

- Der QUERY-Knoten ist die Wurzel des Graphen.
- Der NOT-Knoten zählt wie AND und OR zu den inneren Knoten des Graphen.
- Die Knoten IFACT, FACT und FUNC stellen Blätter des Graphen dar, da sie keine weiteren Unterknoten mehr besitzen.

Definition 3.9 *Nachfolgeknoten, Vorgängerknoten*

Sei $SG_{\mathcal{L}} = (V, E)$ ein Strukturgraph zu einem Logikprogramm $\mathcal{L}$.

Ein Knoten $w \in V$ heißt Nachfolgeknoten von $v \in V$, wenn $e_{v,w} \in E$ existiert. v heißt dann Vorgängerknoten von w. $\diamond$

Lemma 3.10

Es seien die Bezeichnungen wie in Definition 3.8. Die allgemeine Definition eines Pfades als Kombination mehrerer, zusammenhängender Kanten, ist in Anhang A.5 nachzulesen.

1. Wenn es Regeln
   $P^0_{i,k} \leftarrow ..., P^r_{i,k}, ...$
   $P^0_{i',k'} \leftarrow ... .$
   gibt mit
   $P^r_{i,k} \simeq P^0_{i',k'}$,
   dann gibt es einen Pfad in $SG_{\mathcal{L}}$ von $v^{\wedge}_{i,k}$ nach $v^{\wedge}_{i',k'}$, wobei $v^{\wedge}_{i,k}$ den Kopf der Regel $P^0_{i,k} \leftarrow ..., P^r_{i,k}, ...$ und $v^{\wedge}_{i',k'}$ den Kopf der Regel $P^0_{i',k'} \leftarrow ... .$ repräsentiert.
2. Wenn es eine Regel
   $P^0_{i,k} \leftarrow ..., P^r_{i,k}, ...,$
   mit $P^r_{i,k}$ hat Prädikat $p \in Pred$, gibt,
   dann gibt es einen Pfad in $SG_{\mathcal{L}}$ von $v^{\wedge}_{i,k}$ nach $v^r_{i,k}$, wobei $v^{\wedge}_{i,k}$ den Kopf der Regel $P^0_{i,k} \leftarrow ..., P^r_{i,k}, ...$ und $v^r_{i,k}$ das Literal $P^r_{i,k}$ repräsentiert. $\diamond$

Beweis:

Allgemein gilt: für $P^0_{i,k}$ existiert $v^{\wedge}_{i,k}$. Gibt es $P^0_{i',k'}$, so existiert $v^{\wedge}_{i',k'}$ und gibt es $P^r_{i',k'}$, so existiert $v^r_{i',k'}$ (Def. 3.8 i, (1)).

1. für alle $P^r_{i,k}$, mit $P^r_{i,k}$ hat nicht Prädikat $p \in \mathit{Pred}$, gilt:

   i) wenn gilt: $P^r_{i,k}$ ist kein negatives Atom und $\exists P^0_{i',k'}$ mit $P^r_{i,k} \simeq P^0_{i',k'}$, $k' = 1$ und $\nexists P^0_{i',2}$, dann existiert $e_{v^{\wedge}_{i,k}, v^{\wedge}_{i',k'}} \in E_{DG}$ (Def. 3.8 ii, (2)); d.h. es existiert eine Kante von $v^{\wedge}_{i,k}$ nach $v^{\wedge}_{i',k'}$.

   ii) wenn gilt: $P^r_{i,k}$ ist kein negatives Atom, $\exists P^0_{i',k'}$ mit $P^r_{i,k} \simeq P^0_{i',k'}$ und $\exists P^0_{i',k''}$ mit $k'' \neq k'$ (d.h. es gibt eine alternative Regel), dann existiert $v^{\vee}_{i'}$ (Def. 3.8 i, (2)) und damit existiert $e_{v^{\wedge}_{i,k}, v^{\vee}_{i'}} \in E_{DG}$ (Def. 3.8 ii, (1)), sowie $e_{v^{\vee}_{i'}, v^{\wedge}_{i',k'}} \in E_{DG}$ (Def. 3.8 ii, (5)); d.h. es existiert ein Pfad (über genau zwei Kanten) von $v^{\wedge}_{i,k}$ nach $v^{\wedge}_{i',k'}$.

   iii) wenn gilt: $P^r_{i,k}$ ist ein negatives Atom, $\exists P^0_{i',k'}$ mit $P^r_{i,k} \simeq P^0_{i',k'}$, $k' = 1$ und $\nexists P^0_{i',2}$, dann existiert $v^{r\neg}_{i,k}$ (Def. 3.8 i, (4)). Damit existiert $e_{v^{\wedge}_{i,k}, v^{r\neg}_{i,k}} \in E_{DG}$ (Def. 3.8 ii, (4)), sowie $e_{v^{r\neg}_{i,k}, v^{\wedge}_{i',k'}} \in E_{DG}$ (Def. 3.8 ii, (7)); d.h. es existiert ein Pfad (über genau zwei Kanten) von $v^{\wedge}_{i,k}$ nach $v^{\wedge}_{i',k'}$.

   iv) wenn gilt: $P^r_{i,k}$ ist ein negatives Atom, $\exists P^0_{i',k'}$ mit $P^r_{i,k} \simeq P^0_{i',k'}$ und $\exists P^0_{i',k''}$ mit $k'' \neq k'$, dann existiert $v^{r\neg}_{i,k}$ (Def. 3.8 i, (4)) und es existiert $v^{\vee}_{i'}$ (Def. 3.8 i, (2)) und damit existiert $e_{v^{\wedge}_{i,k}, v^{r\neg}_{i,k}} \in E_{DG}$ (Def. 3.8 ii, (4)), sowie $e_{v^{r\neg}_{i,k}, v^{\vee}_{i'}} \in E_{DG}$ (Def. 3.8 ii, (6)) und $e_{v^{\vee}_{i'}, v^{\wedge}_{i',k'}} \in E_{DG}$ (Def. 3.8 ii, (5)); d.h. es existiert ein Pfad (über genau drei Kanten) von $v^{\wedge}_{i,k}$ nach $v^{\wedge}_{i',k'}$.

2. für $P^r_{i,k}$ mit Prädikat $p \in \mathit{Pred}$ gilt:

   i) wenn gilt: $P^r_{i,k}$ ist kein negatives Atom, dann existiert $v^r_{i,k}$ (Def. 3.8 i, (3)) und damit existiert $e_{v^{\wedge}_{i,k}, v^r_{i,k}} \in E_{DG}$ (Def. 3.8 ii, (3)); d.h. es existiert eine Kante von $v^{\wedge}_{i,k}$ nach $v^r_{i,k}$.

   ii) wenn gilt: $P^r_{i,k}$ ist negatives Atom, dann existiert $v^{r\neg}_{i,k}$ (Def. 3.8 i, (4)) und damit existiert $e_{v^{\wedge}_{i,k}, v^{r\neg}_{i,k}} \in E_{DG}$ (Def. 3.8 ii, (4)), sowie $e_{v^{r\neg}_{i,k}, v^r_{i,k}} \in E_{DG}$ (Def. 3.8 ii, (8)); d.h. es existiert ein Pfad (über genau zwei Kanten) von $v^{\wedge}_{i,k}$ nach $v^r_{i,k}$.

Es gibt keine andere Möglichkeit die Knoten miteinander zu verbinden, da die Teilmengen (aus Def 3.8 ii, (1) - (8)) der Kantenmengen $E^{\wedge}, E^{\vee}$ und $E^{\neg}$ aufgrund ihrer Vorbedingungen offensichtlich disjunkt sind. ◇

Definition 3.11 *Pfad*

Es seien die Bezeichnungen wie in Definition 3.8.

Ein einfacher Pfad ist ein Pfad von $v^{\wedge}_{i,k}$ nach $v^{\wedge}_{i',k'}$, wenn $v^{\wedge}_{i,k}$ den Kopf der Regel $P^0_{i,k} \leftarrow$ ..., $P^r_{i,k}$, ... und $v^{\wedge}_{i',k'}$ den Kopf der Regel $P^0_{i',k'} \leftarrow$ .... repräsentiert und $P^r_{i,k} \simeq P^0_{i',k'}$ gilt.

Ein zusammengesetzter Pfad ist ein einfacher Pfad oder eine Aneinanderreihung einfacher Pfade. ◇

Definition 3.12 *Berechnung*

(siehe auch Anhang A.3)

Die *Berechnung* einer Menge von Abfragen $\mathcal{Q} = \mathcal{Q}_0$ durch ein Programm $\mathcal{P}$ ist eine Folge von Tripeln $(\mathcal{Q}_s, \mathcal{G}_s, \mathcal{C}_s)$, $s \in \mathbb{N}$. $\mathcal{Q}_s$ ist eine Menge von Abfragen, $\mathcal{G}_s$ ist eine Abfrage aus $\mathcal{Q}_s$ und $\mathcal{C}_s$ ist eine Klausel, die derart umbenannt wurde, daß sie nur Variablennamen enthält, die nicht in $\mathcal{Q}_t$, $0 \leq t \leq s$ vorkommen. Für alle $s > 0$ entsteht $\mathcal{Q}_{s+1}$ aus der Ersetzung von $\mathcal{G}_s$ durch den Rumpf von $\mathcal{C}_s$ in $\mathcal{Q}_s$ und der Anwendung des allgemeinsten Unifikators Φ_s von $\mathcal{G}_s$ und dem Kopf von $\mathcal{C}_s$ auf $\mathcal{Q}_s$ [Sterling 86].

$(\mathcal{Q}_s, \mathcal{G}_s, \mathcal{C}_s)$ wird als Berechnungsschritt bezeichnet. ◇

Bemerkung: Als Menge von Abfragen werden die Einzelabfragen im Rumpf der eigentlichen Abfrage verstanden.

Satz 3.13

Der Strukturgraph repräsentiert die strukturellen Abhängigkeiten des zugrunde liegenden $\text{DATALOG}_f^{\neg}$-Programms (d.h. jeder möglichen Berechnung in einem $\text{DATALOG}_f^{\neg}$-Programm entspricht ein Pfad im Strukturgraphen). ◇

Beweis:

Es seien die Bezeichnungen wie in Definition 3.8.

Sei $Bs_{\mathcal{L}}$ die Menge alle möglichen Berechnungsschritte eines $\text{DATALOG}_f^{\neg}$-Programms $\mathcal{L}$, $Pf_{\mathcal{L}}$ die Menge aller von der Wurzel des Graphen $SG_{\mathcal{L}}$ erreichbaren einfachen Pfade. (Dabei heißt erreichbar, daß ein zusammengesetzter Pfad von der Wurzel bis zum ersten Knoten des Pfades existiert.)

Zu zeigen ist: $Bs_{\mathcal{L}} \doteq Pf_{\mathcal{L}}$ (d.h. jedem Element aus $Bs_{\mathcal{L}}$ entspricht ein Element aus $Pf_{\mathcal{L}}$), wobei "$\sqsubset$", "$\sqsupset$" für die Existenz einer Entsprechung in der jeweiligen Richtung stehen soll.

"$\sqsubset$" Wenn es einen Berechnungsschritt $(\mathcal{Q}_s, \mathcal{G}_s, \mathcal{C}_s) \in Bs_{\mathcal{L}}$, $s \in \mathbb{N}$ gibt (wobei gilt: $\mathcal{G}_s \in \mathcal{Q}_s$), dann muß:

- $\mathcal{G}_s$ entweder Teil der Abfrage $(\leftarrow ..., \mathcal{G}_s, ...)$ gewesen sein, also $\mathcal{G}_s \in \mathcal{Q}_0$, oder
- es muß einen Berechnungsschritt $(\mathcal{Q}_t, \mathcal{G}_t, \mathcal{C}_t)$, $t \in \mathbb{N}$, mit $t < s$ und einer Regel $\mathcal{C}_t$ gegeben haben, wobei $\mathcal{G}_s$ im Rumpf von $\mathcal{C}_t$ $(\mathcal{C}_t \leftarrow ..., \mathcal{G}_s, ...)$ vorkommt.

Für den (leeren) Kopf der Abfrage stehe $v^{\wedge}_{i,k}$ mit $i = 0, k = 1$, für den Kopf der Regel $\mathcal{C}_t$ stehe $v^{\wedge}_{i,k}$ mit $i > 0, k \geq 1$. Für den Kopf der Regel $\mathcal{C}_s$ stehe $v^{\wedge}_{i',k'}$ (nach Def 3.8 i, (1)). Nach Voraussetzung existiert eine Unifikation Φ_s, für $\mathcal{G}_s, \mathcal{C}_s$ (Definition eines Berechnungschrittes), d.h. $\mathcal{G}_s \simeq \mathcal{C}_s$. Nach Lemma 3.4 existiert dann ein einfacher Pfad von $v^{\wedge}_{i,k}$ nach $v^{\wedge}_{i',k'}$, d.h. $e_{v^{\wedge}_{i,k}, v^{\wedge}_{i',k'}} \in Pf_{\mathcal{L}}$

Durch wiederholte Anwendung dieser Argumentation, ausgehend von $(\mathcal{Q}_0, \mathcal{G}_0, \mathcal{C}_0)$, ergibt sich die Behauptung auch für (von der Wurzel ausgehende) zusammengesetzte Pfade.

"$\sqsupset$" Ist $e_{v^{\wedge}_{i,k}, v^{\wedge}_{i',k'}} \in Pf_{\mathcal{L}}$ und steht $v^{\wedge}_{i,k}$ für den Kopf einer Regel $\mathcal{C}_s$ und $v^{\wedge}_{i',k'}$ für den Kopf einer Regel $\mathcal{C}_t \leftarrow ..., \mathcal{G}_s, ...$ mit $\mathcal{G}_s \simeq \mathcal{C}_s$, so gibt es nach Definition 3.8 einen allgemeinsten Unifikator Φ_s für $\mathcal{G}_s, \mathcal{C}_s$. Aufgrund der Erreichbarkeit des einfachen Pfades mit erstem Knoten $v^{\wedge}_{i',k'}$, gibt es Berechnungsschritte $(\mathcal{Q}_{t'}, \mathcal{G}_{t'}, \mathcal{C}_{t'})$ für alle $t' \in \{t_0, ..., t_n, t\}$, so daß $\mathcal{G}_s \in \mathcal{Q}_s$ für $s > t$. Mit Φ_s gibt es dann auch einen Berechnungsschritt $(\mathcal{Q}_s, \mathcal{G}_s, \mathcal{C}_s) \in Bs_{\mathcal{L}}$.

Eine Folge von Berechnungschritten entspricht dann einem zusammengesetzten Pfad. ◇

Beispiel:

Figur 12 veranschaulicht Definition 3.8 an folgendem Beispiel:

$$f(X,Y) \leftarrow g(X,Z),\ \mathsf{not}\ h(Z), l(Z,Y), h(Y),\ \mathsf{not}\ k(Z).$$

wobei es für $l(..)$ und $h(..)$ jeweils mehrere Regeln, für $g(..)$ und $k(..)$ genau eine Regel geben soll.

Dabei entspricht:

$$\begin{array}{llllll} f(X,Y) \leftarrow & g(X,Z), & \mathsf{not}\ h(Z), & l(Z,Y), & h(Y), & \mathsf{not}\ k(Z). \\ P^0_{1,1} & P^1_{1,1} & P^2_{1,1} & P^3_{1,1} & P^4_{1,1} & P^5_{1,1} \end{array}$$

$$\begin{array}{lll} g(X,Y) \leftarrow ... & & \\ P^0_{2,1} & & \\ h(X) \leftarrow ... & h(X) \leftarrow ... & \\ P^0_{3,1} & P^0_{3,2} & \\ l(X,Y) \leftarrow ... & l(X,Y) \leftarrow ... & l(X,Y) \leftarrow ... \\ P^0_{4,1} & P^0_{4,2} & P^0_{4,3} \\ k(X) \leftarrow ... & & \\ P^0_{5,1} & & \end{array}$$

◇

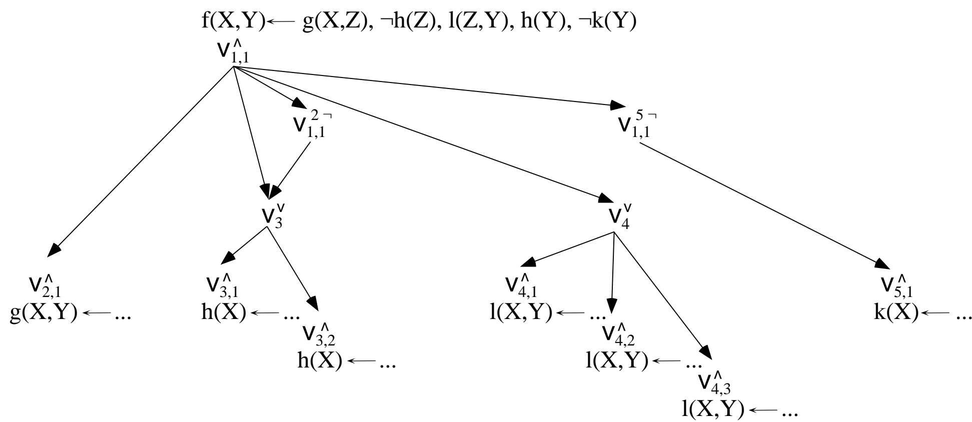


Abbildung 12: Beispiel des Aufbaus eines Strukturgraphen

Ergänzend erhalten die Knoten des Strukturgraphen die Attribute "Name" und "Typ" zur Identifikation.

Definition 3.14 *Attributierung der Knoten*

Zur weiteren Kennzeichnung erhalten alle Knoten $v \in V$ in $SG_{\mathcal{L}}$ zwei Attribute (es gelten die Bezeichnungen aus Definition 3.8):

i) *v.type* bezeichne den Typ, der dem Knoten v zugeordnet wird:

$$- v^{\wedge}_{i,k}.type = \begin{cases} \textsc{And} & \text{wenn } P^0_{i,k} \text{ Kopf einer Regel ist,} \\ \textsc{Fact/IFact} & \text{wenn } P^0_{i,k} \text{ ein Fakt ist, bzw. eine Schnittstelle} \\ & \text{zum Datenspeicher darstellt,} \\ \textsc{Query} & \text{wenn die Regel von der Form } \leftarrow P^1_{i,k}, ..., P^{l_0}_{i,k}. \\ & \text{ist, d.h. } i = 0, k = 1, \end{cases}$$

- $v^{\vee}_i.type =$ Or,
- $v^{r\,\neg}_{i,k}.type =$ Not,
- $v^r_{i,k}.type =$ Func, wenn gilt: $P^r_{i,k}$ hat Prädikat $p \in Pred$ mit $r \in \{1, ..., l_{i,k}\}$.

mit $i \in \{0, ..., n\}$, $k \in \{1, ..., m_i\}$, $r \in \{1, ..., l_{i,k}\}$.

ii) $v.name$ bezeichne das Literal, das dem Knoten v zugeordnet wird:

- $v^{\vee}_i.name = P^0_{i,1}$,

$$- v^{\wedge}_{i,k}.name = \begin{cases} P^0_{i,k} & \text{wenn } \exists P^0_{i,k} \text{ (gilt } \forall i > 0,\ k \in \mathbb{N}\text{),} \\ \bot & \text{sonst (Query-Knoten erhalten keinen Namen),} \end{cases}$$

- $v^{r\,\neg}_{i,k}.name = P^r_{i,k}$,
- $v^r_{i,k}.name = P^r_{i,k}$ mit $r \in \{1, ..., l_{i,k}\}$,

mit $i \in \{0, ..., n\}$, $k \in \{1, ..., m_i\}$, $r \in \{1, ..., l_{i,k}\}$.

Bemerkung: $v.name$ ist (durch die Zurodnung von $P^r_{i,k}$) eindeutig. ◇

Definition 3.15 *Attributierung der Kanten*

Alle Kanten $e \in E$ in $SG_{\mathcal{L}}$ erhalten zur weiteren Kennzeichnung ein Attribut (unter Verwendung der Bezeichnungen aus Definition 3.8):

$e.call$ bezeichne das Literal in der Regel, über das der Aufruf erfolgt, den die Kante repräsentiert:

- $e_{v^{\wedge}_{i,k}, v^{\vee}_{i'}}.call = P^r_{i,k}$ wenn $\exists P^r_{i,k}, P^0_{i',1}$ mit $P^r_{i,k} \simeq P^0_{i',1}$
- $e_{v^{\wedge}_{i,k}, v^{\wedge}_{i',1}}.call = P^r_{i,k}$ wenn $\exists P^r_{i,k}, P^0_{i',1}$ mit $P^r_{i,k} \simeq P^0_{i',1}$
- $e_{v^{\wedge}_{i,k}, v^r_{i,k}}.call = v^r_{i,k}.name$
- $e_{v^{\wedge}_{i,k}, v^{r\,\neg}_{i,k}}.call = v^{r\,\neg}_{i,k}.name$
- $e_{v^{\vee}_i, v^{\wedge}_{i,k}}.call = v^{\wedge}_{i,k}.name$
- $e_{v^{r\,\neg}_{i,k}, v^{\vee}_{i'}}.call = P^r_{i,k}$ wenn $\exists P^r_{i,k}, P^0_{i',1}$ mit $P^r_{i,k} \simeq P^0_{i',1}$
- $e_{v^{r\,\neg}_{i,k}, v^{\wedge}_{i',1}}.call = P^r_{i,k}$ wenn $\exists P^r_{i,k}, P^0_{i',1}$ mit $P^r_{i,k} \simeq P^0_{i',1}$
- $e_{v^{r\,\neg}_{i,k}, v^r_{i,k}}.call = v^r_{i,k}.name$

mit $i \in \{0, ..., n\}$, $k \in \{1, ..., m_i\}$, $r \in \{1, ..., l_{i,k}\}$. ◇

Die Syntax eines Logikprogramms kann also schematisch auf einen Graphen übertragen werden. Dieser Graph wird, da er die syntaktische Struktur des Logikprogramms repräsentiert, in dieser Arbeit Strukturgraph genannt. Er entspricht bezüglich seiner Struktur aus dem Bereich der deduktiven Datenbanken bekannten Graphformen, wie z.B. dem Operatorbaum.

3.2.2 Zyklen

Ein DATALOG$_f^{\neg}$-Programm kann Rekursionen enthalten. Im Strukturgraphen verursachen diese Rekursionen Zyklen.

Definition 3.16 *Zyklen im Strukturgraphen*

$SG_{\mathcal{L}} = (V, E)$ sei ein Strukturgraph zu einem Logikprogramm $\mathcal{L}$.

Existiert ein Pfad $(e^1, ..., e^n)$ in $SG_{\mathcal{L}}$, $n \in \mathbb{N}$ mit $e^i \equiv e^i_{v_i, v_{i+1}} \in E,\ v_i, v_{i+1} \in V,\ \forall i \in \{1, ..., n-1\}$ und ist $v_1 = v_n$, so besitzt der Strukturgraph einen Zyklus.

Die Kanten $e^1, ..., e^{n-1}$ sind dann vorwärtsgerichtete Kanten, die Kante e^n wird o.B.d.A. als rückführende Kante bezeichnet. ◇

Bemerkung: Ein Zyklus von mehreren, durch einen Pfad verbundenen Knoten kann also aufgeteilt werden in normale, d.h. vorwärtsgerichtete Kanten und genau eine rückführende Kante.

Der folgende Algorithmus aus [Thulasirman 92] weist jedem Knoten genau zwei Zahlen zu: *dfsnr* steht für die Reihenfolge des Knotens beim Durchlauf durch den Graphen mit Tiefensuche. Das Attribut *visited* zeigt an, ob der Knoten schon besucht wurde.

Definition 3.17 *Ein Algorithmus zum Auffinden von Zyklen in Graphen*

$SG_{\mathcal{L}} = (V, E)$ sei ein Strukturgraph zu einem Logikprogramm $\mathcal{L}$.

Jeder Knoten $v \in V$ in $SG_{\mathcal{L}}$ besitze zwei zusätzliche Attribute *dfsnr* und *visited*. E teile sich auf in $\mathcal{E}^v \cup \mathcal{E}^m \cup \mathcal{E}^r$.

1. Für alle Knoten $v \in SG_{\mathcal{L}}$ werden $v.dfsnr := 0$ und $v.visited := 0$ gesetzt. Die Zähler `dfs` und `vis` werden mit 1 initialisiert: `dfs`:= 1, `vis`:= 1.

2. Beginne mit dem QUERY-Knoten: $v := v_q$, wobei $v_q.type =$ QUERY. Setze $v.dfsnr$:= `dfs`, `dfs` := `dfs`+1.

3. Für den aktuellen Knoten $v \in SG_{\mathcal{L}}$ und für alle Knoten $w \in SG_{\mathcal{L}}$ mit $e_{v,w} \in \mathcal{E}$ tue:
   - wenn $w.dfsnr > 0$ und $w.visited = 0$, dann ist $e_{v,w} \in \mathcal{E}^r$,
   - wenn $w.dfsnr > 0$ und $w.visited \neq 0$, dann ist $e_{v,w} \in \mathcal{E}^m$,
   - sonst ist $e_{v,w} \in \mathcal{E}^v$
     · $w.dfsnr$:= `dfs`, `dfs` := `dfs`+1,
     · Rufe 3 mit w als dem aktuellen Knoten rekursiv auf: $v := w$,
     · $w.visited$:= `vis`, `vis` := `vis`+1. ◇

Bemerkung 3.18

Die Menge der gerichteten Kanten wird durch diesen Algorithmus aufgeteilt:

- $\mathcal{E}^v$ ist die Menge der vorwärtsgerichteten Kanten; bei Knoten, die von mehreren Knoten aus erreicht werden, gehört in diese Menge nur die erste vom Algorithmus gefundene Kante, die zu diesem Knoten führt.
- $\mathcal{E}^m$ ist die Menge der vorwärtsgerichteten Kanten, die zu einen Knoten führen, der schon besucht wurde, d.h. wenn von mehreren Knoten aus Kanten zu diesem Knoten führen, gehören alle Kanten ohne die erste gefundene Kante in diese Menge.

- $\mathcal{E}^r$ ist die Menge der rückführenden Kanten.

Satz 3.19

Es gelten die Bezeichnungen wie in Def. 3.17.

Mit $E = \mathcal{E}^v \cup \mathcal{E}^m \cup \mathcal{E}^r$ gilt: $(\mathcal{E}^v \cap \mathcal{E}^m = \emptyset) \wedge (\mathcal{E}^v \cap \mathcal{E}^r = \emptyset) \wedge (\mathcal{E}^m \cap \mathcal{E}^r = \emptyset)$. $\diamond$

Beweis:

Zu zeigen ist: Jede Kante wird genau einer Menge $\mathcal{E}^v, \mathcal{E}^m$ oder $\mathcal{E}^r$ zugeordnet.

- Jede Kante wird mindestens einmal erreicht (Erreichbarkeit von der Wurzel vorausgesetzt):
  Für einen Knoten v werden alle von ihm wegführenden Kanten betrachtet. Der Algorithmus fährt mit den Knoten w fort, die Endknoten dieser Kanten sind (Tiefensuche).
- Jede Kante wird maximal einmal erreicht:
  Die von einem Knoten v ausgehenden Kanten werden nur dann betrachtet, wenn $v.dfsnr = 0$ ist. Bei der ersten Betrachtung wird $v.dfsnr \neq 0$ gesetzt, daher kann keine Kante ein zweites Mal betrachtet werden.

Zusätzlich gilt, daß jede erreichte Kante durch den Algorithmus in genau eine Menge eingeordnet wird. $\diamond$

3.3 Analyse der Datenabhängigkeiten

Der Strukturgraph ist selbst noch kein Datenflußgraph, d.h. nicht auf einem Datenflußrechner ausführbar. Es gibt z.B. zwischen zwei Unterknoten eines Knotens, von denen der eine Werte produziert, die der andere konsumiert, keine direkte Verbindung, sondern nur den Weg über den Vorgängerknoten. Dieser müßte sowohl Werte nach "unten" zu einem Unterknoten weitergeben, d.h. top-down agieren können, als auch Werte von "unten" bekommen, d.h bottom-up agieren können.

Hierbei handelt es sich also um eine anforderungsgesteuerte Ausführung. Für eine rein datengetriebene Ausführung dürfen bei einem Datenflußgraphen Kanten jedoch nur unidirektional benutzt werden.

Um nun aus dem erweiterten Strukturgraphen einen rein bottom-up auswertbaren Datenflußgraphen zu erstellen, muß von der existierenden Graphform eine neue Graphform mit veränderter Kantenstruktur abgeleitet werden.

3.3.1 Parameter von Knoten

Knoten benötigen Stellen, an denen Werte ein- oder ausgegeben werden können. Der Einfachheit wegen werden diese Stellen wie bei Literalen als Parameter bezeichnet.

Ein Knoten hat zunächst folgende Parameter:

- Parameter, die den Parametern des Literals entsprechen, das der Knoten repräsentiert. Diese werden im folgenden als *Übergabeparameter* bezeichnet.
- Einen Parameter für den Ergebniswert (jedes Literal kann erfolgreich sein oder scheitern, der Ergebniswert kann also *true* oder *false* sein). Dieser Parameter wird als *Ergebnisparameter* bezeichnet.

Zusätzlich kann ein (AND-)Knoten weitere Parameter erhalten:

- Treten im Rumpf einer Regel neue Parameter auf, so erhält der Knoten, der den Kopf der Regel repräsentiert, zusätzlich Parameter, die diesen entsprechen. Diese Parameter werden als *lokale Parameter* bezeichnet.

Definition 3.20 *Parameter von Knoten*

Sei $SG_{\mathcal{L}} = (V, E)$ ein Strukturgraph zu einem Logikprogramm $\mathcal{L}$.

Ein Knoten $v \in V$ mit $v.name = P \equiv p(t_1, ..., t_u)$, $u \in \mathbb{N}$, $p \in Pred$, $t_1, ..., t_u$ Terme, habe Parameter δ_i, $i \in \{1, ..., u' + 1\}$, $u, u' \in \mathbb{N}, u \leq u'$. Sie seien wie folgt definiert:

i) Übergabeparameter

$\delta_i.name$ mit $i \in \{1, ..., u\}$ sei die Bezeichnung des Parameters:

· $\delta_i.name = a$, wenn $t_i \equiv a$, a Konstante,

· $\delta_i.name = X$, wenn $t_i \equiv X$, X Variable.

(Terme treten aufgrund der Extraktion von Funktionen und Operationen aus der Parameterliste eines Literals nicht mehr auf).

ii) Lokale Parameter

$\delta_i.name$ mit $i \in \{u+1, ..., u'\}$ sei gegeben durch:

· $\delta_i.name = a$, wenn $\exists e_{v,w} \in E$, $e_{v,w}.call = q(l_1, ..., l_m)$, $m \in \mathbb{N}$ mit $l_j \equiv a$ für ein $j \in \{1, ..., m\}$ und a Konstante und l_j wurde keinem $\delta_{i'}$ für $i' \neq i$ zugeordnet,

· $\delta_i.name = X$, wenn $\exists e_{v,w} \in E$, $e_{v,w}.call = q(l_1, ..., l_m)$, $m \in \mathbb{N}$, $q \in Pred$, $l_1, ..., l_m$ Terme, mit $l_j \equiv X$ für ein $j \in \{1, ..., m\}$ und X Variable und $\nexists \delta_{i'}$ mit $\delta_{i'}.name = X$ für $i' \neq i$.

iii) Ergebnisparameter

$\delta_{u'+1}.name = \perp$ (der Ergebnisparameter erhält keinen Namen).

Außerdem erhalten die Parameter folgende Attribute, die später im Detail spezifiziert werden:

- $\delta.ident$ bezeichne eine Liste von eindeutigen "Namen", die dem Parameter δ zugeordnet werden.
- $\delta.loop_ident$ bezeichne eine weitere Liste von eindeutigen "Namen", die dem Parameter δ zugeordnet werden.
- $\delta.history$ bezeichne die "Vorgeschichte" des Parameters.
- $\delta.mode$ bezeichne den Ein- oder Ausgabetyp (Modus) des Parameters. ◇

Bemerkung 3.21

Prinzipiell soll nun für Konstanten wie für Variablen eine Unterscheidung zwischen ihrem Bezeichner a und ihrem Wert gemacht werden. Konstanten mit unterschiedlichem Bezeichner $a_1, a_2, ...$ können den gleichen Wert besitzen. ◇

Bei einer Ausführung der Eingabespezifikation als DATALOG$^{\neg}_{f}$-Programm werden Werte für bestimmte Parameter, z.B. von Fakten, erzeugt und über Parameter von Regelköpfen an die Parameter der Abfrage weitergegeben. Es besteht daher eine enge Verbindung zwischen den Parametern verschiedener Literale. Diese soll im folgenden näher definiert werden.

Definition 3.22 *Korrespondenz von Parametern in Literalen*

Die Parameter zweier Literale eines Logikprogramms $\mathcal{L}$ heißen korrespondierend,

i) wenn die Literale in derselben Regel eines Programms $\mathcal{L}$ vorkommen und die Parameter den gleichen Namen haben (d.h. eine beliebige, durch einen Berechnungsschritt ausgelöste Unifikation würde beide Parameter grundsätzlich durch den gleichen Ausdruck oder Wert ersetzen).

ii) wenn die Literale unifizierbar sind und die Parameter die gleiche Position in der jeweiligen Parameterliste besitzen (so daß bei einer Unifikation der Literale die Parameter miteinander unifiziert werden). ◇

Da Knoten Literale repräsentieren und damit die Parameter von Knoten den Parametern von Literalen entsprechen, kann nun auch die Korrespondenz von Parametern in Knoten definiert werden.

Definition 3.23 *Korrespondenz von Parametern in Knoten*

Zu einem Strukturgraphen $SG_{\mathcal{L}} = (V, E)$ heißen die Parameter δ, θ zweier Knoten $v, w \in V$ (wobei δ Parameter von v ist und θ Parameter von w) korrespondierend (i. Z. $\delta \simeq_k \theta$), wenn es eine Kante $e_{v,w} \in E$ gibt und

i) die Knoten v, w Literale repräsentieren, die innerhalb einer Regel des Programms $\mathcal{L}$ vorkommen und die Parameter δ, θ den gleichen Namen haben, oder

ii) wenn die Kante $e_{v,w}$ eine Unifikation der Knoten v, w und damit der Parameter δ, θ repräsentiert.

Zusätzlich gilt trivialerweise, daß die Parameter zweier Knoten $v, w \in V$ korrespondierend sind, wenn es eine Kante $e_{v,w} \in E$ gibt und

iii) die Knoten v, w das gleiche Literal aus $\mathcal{L}$ repräsentieren und die Parameter δ, θ an der gleichen Position in der Parameterliste stehen, oder

iv) es sich bei den Parametern δ, θ um die Ergebnisparameter der Knoten handelt. ◇

Folgerung 3.24

Die Parameter δ, θ zweier Knoten $v, w \in V$ (wobei δ Parameter von v ist und θ Parameter von w) sind dann korrespondierend, wenn $e_{v,w} \in E$ (w ist dann Nachfolgeknoten von v) und

a) wenn v ein OR-Knoten ist und δ und θ die gleiche Position in der Parameterliste der zugehörigen Literale besitzen,

b) wenn v ein NOT-Knoten ist und δ und θ die gleiche Position in der Parameterliste der zugehörigen Literale besitzen,

c) wenn v ein AND- oder QUERY-Knoten und w ein FUNC-Knoten ist und δ und θ den gleichen Namen haben,

d) wenn v ein AND- oder QUERY-Knoten und w ein NOT-Knoten ist und δ und θ den gleichen Namen haben,

e) für δ an einem AND- oder QUERY-Knoten und θ an einem Nachfolgeknoten (FUNC- und NOT-Knoten ausgenommen) gilt:
   - der Parameter δ hat den gleichen Namen wie ein Parameter θ' desjenigen Literals, das der Verbindungskante zugeordnet ist und
   - θ' hat die gleiche Position in der Parameterliste wie θ. ◇

Beweis:

Es seien die Bezeichnungen wie in Definition 3.8.

a) Ein OR-Knoten $v_i^{\vee} \in V$ wird genau dann erzeugt, wenn es mindestens die AND-Knoten $v_{i,1}^{\wedge}, v_{i,2}^{\wedge}, ... \in V$ mit gleichem Kopf (gleiches Prädikat und gleiche Anzahl von Parametern) gibt (Def. 3.8 i, (2)).

   Für $v_i^{\vee}$ gilt dann: $v_i^{\vee}.name = v_{i,1}^{\wedge}.name = P_{i,1}^0$ (Def. 3.14 i). $v_i^{\vee}$ repräsentiert daher das gleiche Literal wie $v_{i,1}^{\wedge}$. Folglich korrespondieren jeweils die Parameter, die an den gleichen Positionen in der Parameterliste stehen (Def. 3.23, iii).

   Für die Knoten $v_{i,k}^{\wedge} \in V$, mit $k > 1$, gilt, daß die ihnen zugeordneten Literale $P_{i,k}^0$ bis auf gegebenenfalls eine Umbenennung der Parameter identisch zu $P_{i,1}^0$ sind (siehe Def. 2.5). Damit ist aber das Literal $P_{i,1}^0$ von $v_i^{\vee}$ unifizierbar mit dem Literal $P_{i,k}^0$ des Knotens $v_{i,k}^{\wedge}$ für $k > 1$. Folglich korrespondieren jeweils die Parameter, die an den gleichen Positionen in der Parameterliste stehen (Def. 3.23, ii, und Def. 3.22 ii).

b) Ein NOT-Knoten $v^{r\,\neg}_{i,k} \in V$ mit Literal $P^r_{i,k} \equiv p(...)$, $p \in \mathit{Pred}$ zu einem Knoten $v^r_{i,k} \in V$ repräsentiert das gleiche Literal, wie sein (einziger) Nachfolgeknoten (Def. 3.14 i). Folglich korrespondieren jeweils die Parameter, die an den gleichen Positionen in der Parameterliste stehen (Def. 3.23, iii).

   Dagegen führt von einem NOT-Knoten $v^{r\,\neg}_{i,k} \in V$ mit Literal $P^r_{i,k} \equiv p(...)$, $p \notin \mathit{Pred}$ eine Kante zu einem Knoten $v^{\vee}_{i'} \in V$ mit Literal $P^0_{i',1}$ oder, falls dieser nicht existiert, zu $v^{\wedge}_{i',1} \in V$ mit Literal $P^0_{i',1}$, wenn $P^r_{i,k}$ mit dem Kopf $P^0_{i',1}$ einer Regel unifizierbar ist (Def. 3.8 ii, (6) und (7)). Aufgrund der Unfizierbarkeit der Literale korrespondieren ebenfalls jeweils die Parameter, die an den gleichen Positionen in der Parameterliste stehen (Def. 3.23, ii, und Def. 3.22 ii).

Ein AND- (QUERY-) Knoten $v^{\wedge}_{i,k} \in V$ mit Literal $P^0_{i,k}$ repräsentiert den (leeren) Kopf einer Regel. Jeder Parameter, der im Rumpf der Regel vorkommt, aber nicht in der Parameterliste des Kopfs, ist ein lokaler Parameter des Knotens (Def. 3.20 ii). Es gibt also zu jedem Parameter eines Rumpfliterals einen Parameter gleichen Namens an dem Knoten, der für das Kopfliteral steht.

c) Von einem AND- (QUERY-) Knoten $v^{\wedge}_{i,k} \in V$ zu einem FUNC-Knoten $v^r_{i,k} \in V$ mit Literal $P^r_{i,k} \equiv p(...)$, $p \in \mathit{Pred}$ führt genau dann eine Kante, wenn der FUNC-Knoten ein Literal im Rumpf der Regel repräsentiert (Def. 3.8 ii, (3)). Zwei Parameter korrespondieren also, wenn sie am AND- (QUERY-) und am FUNC-Knoten den gleichen Namen haben (Def. 3.23 i).

d) Von einem AND- (QUERY-) Knoten $v^{\wedge}_{i,k} \in V$ zu einem NOT-Knoten $v^{\neg,r}_{i,k} \in V$ mit Literal $P^r_{i,k}$ führt genau dann eine Kante, wenn dieser Knoten für ein negiertes Literal im Rumpf der Regel steht (Def. 3.8 ii, (4)). Zwei Parameter korrespondieren also, wenn sie am AND- (QUERY-) und am NOT-Knoten den gleichen Namen haben (Def. 3.23 i).

e) Eine Kante e mit zugeordnetem Literal $P^r_{i,k}$ (d.h. $e.\mathit{call} = P^r_{i,k}$) führt vom AND- (bzw. QUERY-) Knoten zu einem Nachfolgeknoten $v^{\vee}_{i'} \in V$, oder, falls es diesen Knoten nicht gibt, zu $v^{\wedge}_{i',1} \in V$, wenn ein Literal $P^r_{i,k}$ im Rumpf der Regel existiert, das mit einem Literal $P^0_{i',1}$ des Knotens $v^{\vee}_{i'}$ ($v^{\wedge}_{i',1}$) unifizierbar ist (Def. 3.8 ii, (1) und (2)).

   Mit $P^r_{i,k} \simeq P^0_{i',1}$ sind auch die Parameter an der jeweils gleichen Position unifizierbar. Sei nun θ ein Parameter von $P^r_{i,k}$, der unifizierbar ist mit einem Parameter θ' von $P^0_{i',1}$. Sei außerdem δ ein Parameter von $P^0_{i,k}$ mit $\delta.\mathit{name} = \theta.\mathit{name}$. Wird nun θ mit θ' unifiziert, so wird damit auch δ mit θ' unifiziert, da aufgrund der Definition der Substitution für zwei Parameter mit gleichem Namen die Ersetzung für diese Parameter im Fall einer Unifikation identisch ist.

Bemerkung: Fall c) und d) lassen sich mit e) zusammenfassen, da für Fall c) und d) gilt, daß das Literal, das der Kante e zugeordnet ist, ebenfalls $P^r_{i,k}$ ist:

- Sei δ ein Parameter von $P^0_{i,k}$, θ ein Parameter des Literals $P^r_{i,k}$ das der Kante e zugeordnet ist, θ' ein Parameter des Rumpfliterals $P^r_{i,k}$, und gilt, daß $\delta.\mathit{name} = \theta.\mathit{name}$ und θ und θ' haben die gleiche Position in der Parameterliste, so sind nach Aussage e) δ und θ' korrespondierend. Aufgrund der Identität der Literale $P^r_{i,k}$ der Kante und des Rumpfliterals $P^r_{i,k}$ gilt auch $\delta.\mathit{name} = \theta'.\mathit{name}$ und damit gilt Aussage c) und d).
- Umgekehrt gilt: ist δ ein Parameter von $P^0_{i,k}$, θ' ein Parameter des Rumpfliterals $P^r_{i,k}$, und gilt $\delta.\mathit{name} = \theta'.\mathit{name}$, so sind die Parameter nach Aussage c) und d) korrespondierend. Damit existiert ein Parameter θ des Literals $P^r_{i,k}$ das der Kante e zugeordnet ist, so daß $\delta.\mathit{name} = \theta.\mathit{name}$ und θ' und θ haben die gleiche Position in der Parameterliste. Damit gilt Aussage e). ◇

Satz 3.25

$SG_{\mathcal{L}} = (V, E)$ sei ein Strukturgraph zu einem Logikprogramm $\mathcal{L}$.

Für zwei Knoten $v, w \in V$ mit $e_{v,w} \in E$ gilt: zu jedem Parameter von w gibt es genau einen korrespondierenden Parameter am Vorgängerknoten v. ◇

Beweis:

Für v kann es keine anderen Knotentypen als Or, Not, Query und And geben, da dies die einzigen Knotentypen sind, die nach Definition 3.8 Nachfolgeknoten haben können.

i) Ein Or-Knoten v hat, bis auf eine Umbenennung der Parameter, genau die gleiche Parameterliste wie ein Nachfolgeknoten w (Def. 3.8 i). Aufgrund der Eindeutigkeit der Positionen gilt daher die Aussage.

   Für einen Not-Knoten ist der Beweis analog.

ii) Bei einem And-Knoten v ist der dem Parameter eines Nachfolgeknotens w entsprechende Parameter entweder in der Parameterliste oder aufgrund der Definition lokaler Parameter in der Liste der lokalen Parameter (Def. 3.20). Er kommt also mindestens einmal in der Parameterliste eines And-Knotens vor.

   Aufgrund der Elimination doppelter Variablen aus Regelköpfen (Def. 3.4) kann es zu einem Parameter von v keinen weiteren mit gleichem Namen geben. Lokale Parameter sind aufgrund ihrer Definition eindeutig, d.h. es gibt keine zwei lokalen Parameter mit gleichem Namen. Es gibt außerdem per Definition keinen lokalen Parameter mit gleichem Namen wie ein Übergabeparameter (Def. 3.20). Damit gilt, daß es maximal einen entsprechenden Parameter zu einem Parameter eines Nachfolgeknotens w geben kann.

   Für einen Query-Knoten ist der Beweis analog. ◇

Satz 3.26

Die Umkehrung von Satz 3.25 gilt nicht: für zwei Knoten $v, w \in V$ mit $e_{v,w} \in E$ gibt es nicht zu jedem Parameter von v genau einen korrespondierenden Parameter am Nachfolgeknoten w. ◇

Beweis:

Mehrere Nachfolgeknoten eines And-Knotens können Parameter gleichen Namens haben. Andererseits muß nicht jeder Parameter in jedem Nachfolgeknoten auftreten. ◇

Definition 3.27 *Transitive Erweiterung der Korrespondenz von Parametern*

Zwei Parameter δ und θ sollen auch dann als korrespondierend bezeichnet werden, wenn es einen Parameter η gibt, der sowohl zu δ, als auch zu θ korrespondierend ist.

Die durch $\simeq_k$ spezifizierte Relation ist also transitiv: $\delta \simeq_k \eta \wedge \eta \simeq_k \theta \Rightarrow \delta \simeq_k \theta$. ◇

3.3.2 Vergabe eindeutiger Parameternamen

Um über die Verwendung eines Parameters eines bestimmten Knotens mehr Informationen zu erhalten, muß bekannt sein, an welchen Knoten dieser Parameter im Graphen noch vorkommt.

Nun können Parameter beim "Aufruf" einer Regel durch ein Literal umbenannt werden, treten also nicht im ganzen Graphen unter dem gleichen Namen auf. Deshalb muß für jeden Parameter ein *eindeutiger Name* (im folgenden *Identifikator* genannt) vergeben werden.

Die Vergabe geschieht anhand des Strukturgraphen mittels Breitensuche. Jedem neu auftretenden Parameter, d.h. einem Parameter in einer Regel, der noch keinen Identifikator besitzt, wird ein neuer Identifikator zugewiesen. Die Identifikatoren der Parameter eines Knoten übertragen sich auf die korrespondierenden Parameter der Unterknoten. Welche Parameter korrespondieren, läßt sich unter Zuhilfenahme der ursprünglichen Regeln und durch die Untersuchung der Aufrufstellen feststellen: innerhalb einer Regel sind das alle Parameter gleichen Namens (d.h. gleicher Variablenbezeichnung), über den Weg eines Aufrufs zählt dagegen die Reihenfolge der Parameter in der Parameterliste der Aufrufstelle und des referenzierten Regelkopfs.

Bemerkung: in den folgenden Beispielen wird die Vergabe von Identifikatoren aufgrund der leichteren Verständlichkeit anhand von Regeln veranschaulicht. Außerdem stehen die für dieses Kapitel wichtigen Sätze und Beweise des besseren Verständnisses wegen nach den einführenden Erläuterungen. Aus diesem Grund treten bei den Erläuterungen Vorwärtsverweise auf verwendete Sätze auf.

Beispiel: Vergabe von Identifikatoren

$f(X,Y) \leftarrow g(X,Z), h(Z,Y).$
$g(X,Y) \leftarrow ...$
$h(U,V) \leftarrow ...$

X soll zunächst den Identifikator "1" und Y den Identifikator "2" haben. In der ersten Regel ist nun Z ein neu auftretender Parameter. Er erhält also einen neuen Identifikator, "3".

Die Identifikatoren werden nun über die Literale im Rumpf der ersten Regel an die Regelköpfe der anderen Regeln weitergegeben.

Beim Aufruf der zweiten Regel durch das Literal $g(X,Z)$ entspricht der Parameter X im Kopf der zweiten Regel dem Parameter X der Aufrufstelle. Er erhält daher vom Parameter der Aufrufstelle den Identifikator "1". Der Parameter Y im Kopf der zweiten Regel entspricht dagegen dem Parameter Z der Aufrufstelle und erhält den Identifikator "3".

Im Regelkopf der dritten Regel erhält der Parameter U auf die gleiche Weise den Identifikator "3", der Parameter V den Identifikator "2".

Das Ergebnis ist:

$f(X_{[1]}, Y_{[2]}) \leftarrow g(X_{[1]}, Z_{[3]}), g(Z_{[3]}, Y_{[2]}).$
$g(X_{[1]}, Y_{[3]}) \leftarrow ...$
$h(U_{[3]}, V_{[2]}) \leftarrow ...$ ◇

Das folgende Beispiel zeigt, daß es Situationen gibt, in denen Parameter mehr als einen Identifikator erhalten können:

Beispiel: Listen von Identifikatoren (1)

$f(X,Y) \leftarrow g(X,Z), g(Z,Y).$
$g(X,Y) \leftarrow h(X,Y).$

Für die erste Regel gilt nun zunächst das gleiche, wie für die erste Regel im vorherigen Beispiel.

Beim ersten Aufruf der zweiten Regel durch das Literal $g(X,Z)$ erhält der Parameter X im Kopf der zweiten Regel wieder den Identifikator "1", der Parameter Y den Identifikator "3".

Beim zweiten Aufruf der zweiten Regel mit $g(Z,Y)$ korrespondiert jedoch der Parameter X im Kopf der zweiten Regel mit dem Parameter Z der Aufrufstelle und erhält diesmal den Identifikator "3". Der Parameter Y im Kopf der zweiten Regel korrespondiert mit dem Parameter Y der Aufrufstelle und erhält den zusätzlichen Identifikator "2".

Es entstehen also Listen von Identifikatoren: $[1,3],[3,2]$.

Das Ergebnis ist also:

$f(X_{[1]}, Y_{[2]}) \leftarrow g(X_{[1]}, Z_{[3]}), g(Z_{[3]}, Y_{[2]}).$
$g(X_{[1,3]}, Y_{[2,3]}) \leftarrow h(X_{[1,3]}, Y_{[2,3]}).$,

wobei z.B. "1" einen Identifikator und "$[1,3]$" eine Liste von Identifikatoren repräsentiert. ◇

Auch der gleiche eindeutige Name kann mehrfach in der Identifikator-Liste vorkommen:

Beispiel: Listen von Identifikatoren (2)

Regeln	Eindeutige Namen
$f(X,Y,Z) \leftarrow g(X,Y), g(X,Z).$	$f(X_{[1]}, Y_{[2]}, Z_{[3]}) \leftarrow g(X_{[1]}, Y_{[2]}), g(X_{[1]}, Z_{[3]}).$
$g(X,Y) \leftarrow h(X,Y).$	$g(X_{[1,1]}, Y_{[2,3]}) \leftarrow h(X_{[1,1]}, Y_{[2,3]}).$

◇

Als Tupel faßt man nun die jeweils ersten, jeweils zweiten, usw., Elemente der Identifikator-Listen (im folgenden als ID-Listen von Knoten bezeichnet) eines Knotens auf, also die Elemente, die aufgrund der Tatsache, daß sie von ein und demselben Vorgängerknoten erzeugt wurden, zusammengehören. Die Reihenfolge der Identifikatoren in den ID-Listen ist wichtig, um über sie eine Zusammengehörigkeit der Identifikatoren festzuhalten.

Für die im weiteren Verlauf der Arbeit notwendige vollständige Analyse der Modi der Parameter sind alle Tupel von Identifikatoren, die an einem Knoten entstehen, relevant. Listen von Identifikatoren müssen daher möglich sein.

Wie nach Satz 3.31 gilt, erhalten bei dieser Methode alle Parameter eines Knotens, die ihre Identifikatoren von den Parametern von Vorgängerknoten ableiten, ID-Listen gleicher Länge. Aus Konsistenzgründen sollten dabei auch die im Knoten neu vorkommenden Parameter (lokale Parameter) nicht nur einen Identifikator, sondern eine ID-Liste mit gleiche Länge erhalten. Dabei ist nach Satz 3.33 jedoch die wiederholte Vergabe des gleichen Identifikators ausreichend.

Beispiel: Mehrfachreferenzierung von Knoten

Regeln	Eindeutige Namen
$f(X,Y) \leftarrow fa(X,Z),\ fb(Z,Y).$	$f(X_{[1]},Y_{[2]}) \leftarrow fa(X_{[1]},Z_{[3]}),\ fb(Z_{[3]},Y_{[2]}).$
$fa(X,Y) \leftarrow g(X,Z), h(Z,Y).$	$fa(X_{[1]},Y_{[3]}) \leftarrow g(X_{[1]},Z_{[4]}), h(Z_{[4]},Y_{[3]}).$
$fb(X,Y) \leftarrow fba(X,Y),\ fbb(X,Y).$	$fb(X_{[3]},Y_{[2]}) \leftarrow$ $fba(X_{[3]},Y_{[2]}),\ fbb(X_{[3]},Y_{[2]}).$
$fba(X,Y) \leftarrow h(X,Y).$	$fba(X_{[3]},Y_{[2]}) \leftarrow h(X_{[3]},Y_{[2]}).$
$h(X,Y) \leftarrow ha(X,Z),\ hb(Z,Y).$	$h(X_{[4,3]},Y_{[3,2]}) \leftarrow$ $ha(X_{[4,3]},Z_{[5,5]}),\ hb(Z_{[5,5]},Y_{[3,2]}).$

Figur 13 zeigt den Strukturgraphen zu diesem Beispiel (alle Knoten sind hier AND-Knoten).

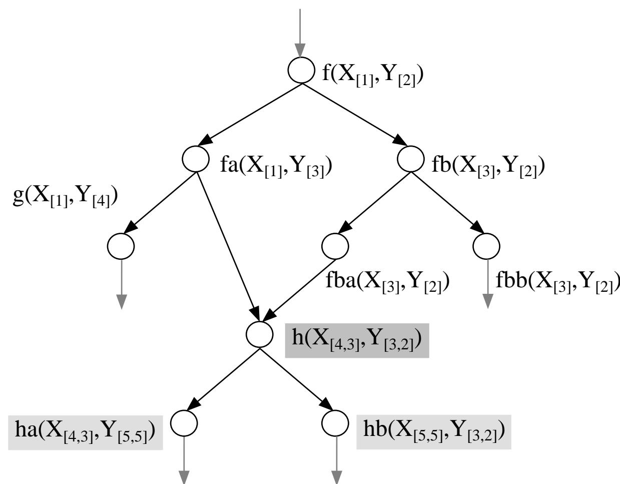


Abbildung 13: Entstehung mehrfacher Identifikatoren für Parameter

In diesem Beispiel sind bei $h([4,3],[3,2])$ die Identifikatoren an der jeweils ersten Stelle in der ID-Liste, 4 für X und 3 für Y, durch eine Referenz entstanden, die Identifikatoren an der jeweils zweiten Stelle in der ID-Liste, 3 für X und 2 für Y, durch eine andere. Die derart zusammengehörigen Identifikatoren bilden Tupel. Alle anderen Kombinationen von Identifikatoren haben keine Bedeutung. ◇

Die ID-Liste eines Parameters im Kopf einer Regel wird identisch an den Parameter eines Literals im Rumpf der Regel mit gleichem Namen weitergegeben. Damit wird sie in identischer Form an den Parameter im Kopf der von diesem Literal aufgerufenen Regel weitergegeben, der an der gleichen Position in der Parameterliste steht. Daher entstehen längere ID-Listen an Nachfolgeknoten immer nur dann, wenn der Knoten einen Regelkopf oder ein Fakt repräsentiert, das im DATALOG$^{\neg}_{f}$-Programm von verschiedenen Stellen aus aufgerufen wird. Die Behauptung gilt nach Satz 3.32. Im Graphen handelt es sich dabei entsprechend um die Knoten, die von mehreren Knoten des Graphen aus entweder durch Mehrfachreferenzierung oder aufgrund eines Zyklus (Schleifenbeginn) erreicht werden können.

Das nachfolgende Beispiel veranschaulicht die Situation bei einer Rekursion.

Beispiel: Schleifen

Regel: $f(X, Y, Z) \leftarrow g(X, W), f(W, Z, Y)$.

im ersten Schleifendurchlauf:
$f(X_{[1]}, Y_{[2]}, Z_{[3]}) \leftarrow g(X_{[1]}, W_{[4]}), f(W_{[4]}, Z_{[3]}, Y_{[2]})$.
im zweiten Schleifendurchlauf:
$f(X_{[1,4]}, Y_{[2,3]}, Z_{[3,2]}) \leftarrow g(X_{[1,4]}, W_{[4,4]}), f(W_{[4,4]}, Z_{[3,2]}, Y_{[2,3]})$.
im dritten Schleifendurchlauf:
$f(X_{[1,4,4]}, Y_{[2,3,2]}, Z_{[3,2,3]}) \leftarrow g(X_{[1,4,4]}, W_{[4,4,4]}), f(W_{[4,4,4]}, Z_{[3,2,3]}, Y_{[2,3,2]})$.
im vierten Schleifendurchlauf:
$f(X_{[1,\mathbf{4},4,\mathbf{4}]}, Y_{[2,\mathbf{3},2,\mathbf{3}]}, Z_{[3,\mathbf{2},3,\mathbf{2}]}) \leftarrow$
$\quad g(X_{[1,\mathbf{4},4,\mathbf{4}]}, W_{[4,\mathbf{4},4,\mathbf{4}]}), f(W_{[4,\mathbf{4},4,\mathbf{4}]}, Z_{[3,\mathbf{2},3,\mathbf{2}]}, Y_{[2,\mathbf{3},2,\mathbf{3}]})$.
im fünften Schleifendurchlauf:
$f(X_{[1,4,\mathbf{4},4,\mathbf{4}]}, Y_{[2,3,\mathbf{2},3,\mathbf{2}]}, Z_{[3,2,\mathbf{3},2,\mathbf{3}]}) \leftarrow$
$\quad g(X_{[1,4,\mathbf{4},4,\mathbf{4}]}, W_{[4,4,\mathbf{4},4,\mathbf{4}]}), f(W_{[4,4,\mathbf{4},4,\mathbf{4}]}, Z_{[3,2,\mathbf{3},2,\mathbf{3}]}, Y_{[2,3,\mathbf{2},3,\mathbf{2}]})$.
usw...

Beim ersten Schleifeneintritt erhält man das Tupel von Identifikatoren (1,2, 3) für die Parameter des Regelkopfs. Beim zweiten die Tupel (1,2,3) und (4,3,2). Beim dritten Schleifeneintritt kommt das Tupel (4,2,3) hinzu. Erst beim vierten Schleifeneintritt ist das neu erzeugte Tupel (4,3,2) eine Wiederholung eines bereits existierenden Tupels (fett gedruckt). Aus diesem Tupel wird nun wieder das gleiche Tupel erzeugt, wie bereits aus dem Wiederholten: (4,2,3). Jede weitere Erzeugung von Tupel wird dann eine Wiederholung bereits erzeugter Tupel sein. Dies gilt nach Satz 3.34.

Es genügt also die Vergabe bis zum dritten Schleifendurchlauf, d.h. bis die erste Wiederholung auftreten würde:

$f(X_{[1,4,4]}, Y_{[2,3,2]}, Z_{[3,2,3]}) \leftarrow g(X_{[1,4,4]}, W_{[4,4,4]}), f(W_{[4,4,4]}, Z_{[3,2,3]}, Y_{[2,3,2]})$. ◇

Man sieht an diesem Beispiel, daß die Vergabe von Identifikatoren bei Schleifen aufgrund der endlichen Anzahl verschiedener Parameter in einem Programm und der daraus resultierenden endlichen Anzahl möglicher Permutationen beim Aufruf einer Regel periodisch wird. Sobald eine Periode festgestellt wird, kann die Vergabe aufhören, da keine neue Abhängigkeiten mehr entstehen. Dies gilt nach Satz 3.33.

Für die Konstruktion eines Datenflußgraphen genügt jedoch im Fall einer Rekursion die Information, welche Parameter der Regel über die Aufrufstelle mit den Parametern der aufgerufenen Stelle korrespondieren (siehe auch Abschnitt 3.4.4.2). Die neuen Tupel von Identifikatoren müssen nicht nocheinmal innerhalb der Schleife weitergegeben werden. Um dies im Rahmen des Algorithmus zu realisieren und eine Unterscheidung zu anderen Identifikatoren zu ermöglichen, erhält jeder Parameter in diesem Fall eine zweite ID-Liste.

Informell lassen sich die Vergaberegeln wie folgt formulieren:

1. Die ID-Liste eines Parameters wird in identischer Form an korrespondierende Nachfolgeparameter weitergegeben.

2. Tritt ein lokaler Parameter an einem Knoten auf, so wird für ihn eine ID-Liste erzeugt, die der Länge der ID-Listen der weiteren an diesem Knoten vorkommenden Parameter entspricht. Gibt es dabei für keinen Parameter des Knotens eine von einem Vorgängerknoten übergebene ID-Liste, ist die Länge der neu erzeugten ID-Liste definitionsgemäß eins. In die ID-Liste wird der an den Parameter vergebene eindeutige Name entsprechend oft eingetragen.

3. Erhalten die Parameter eines Knotens erneut Identifikatoren (z.B. bei Mehrfachreferenzierung oder Schleifen), so wird die ID-Liste eines lokalen Parameters dieses Knotens entsprechend verlängert, wobei derjenige Identifikator eingetragen wird, den der Parameter schon erhalten hat.

4. Eine Schleife muß nur einmal durchlaufen werden. Nach der Vergabe von Identifkatoren von der Aufrufstelle an die aufgerufene Stelle kann diese enden. Die Identifikatoren werden jedoch in eine zweite ID-Liste der jeweiligen Parameter des Knotens, der die aufgerufenen Stelle repräsentiert, eingetragen.

5. An OR- und NOT-Knoten werden die ID-Listen mit Identifikatoren lediglich an die Unterknoten durchgereicht. (Da dies ein trivialer Fall ist, wurde bei obigen Betrachtungen nicht auf diese Knotentypen eingegangen.)

Der Algorithmus lautet nun:

Definition 3.28 *Algorithmus zur Vergabe von Identifikatoren für Parameter*

Sei $SG_{\mathcal{L}} = (V, E)$ ein Strukturgraph zu einem Logikprogramm $\mathcal{L}$. Es gelten außerdem die Bezeichnungen aus Definition 3.14.

Initialisiere für alle Knoten $v \in V$ mit $v.name = p(t_1, ..., t_u)$, $u \in \mathbb{N}$ alle Parameter δ_i, $i \in \{1, ..., u'+1\}$, $u' \in \mathbb{N}$, $u' \geq u$ (wobei $\delta_i.name = t_i$) mit $\delta_i.ident := [\,]$, d.h. der leeren Liste.

1. Beginne mit dem QUERY-Knoten $v_q \in V$ ($v_q.attr =$ QUERY), $v := v_q$ und $n := 1$: für alle $i \in \{1, ..., u\}$ vergebe einen neuen Identifikator $\delta_i.ident = [n]$, $n := n + 1$ für alle i.

2. Für den Knoten $v \in V$ gelte:

   a) ist $v.type =$ OR oder $v.type =$ NOT

   mit $v.name = p(t_1, ..., t_u)$, $u \in \mathbb{N}$, Übergabeparametern δ_i, $i \in \{1, ..., u\}$ (wobei $\delta_i.name = t_i$), sowie Resultatparameter δ_{u+1}, dann setze für alle $w \in V$ mit $e_{v,w} \in E$ mit $w.name = p(t'_1, ..., t'_u)$ (eine Umbenennung von Parametern ist möglich) Übergabeparametern δ'_i, $i \in \{1, ..., u\}$ (wobei $\delta'_i.name = t'_i$), sowie Resultatparameter $\delta'_{u'+1}$, $u' \in \mathbb{N}$, $u' \geq u$:

   - für alle $i \leq u$: $\delta'_i.ident := \delta_i.ident$

   - $\delta'_{u'+1}.ident := \delta_{u+1}.ident.$

   b) ist $v.type =$ AND

   mit $v.name = p(t_1, ..., t_u)$, $u \in \mathbb{N}$, Übergabeparametern δ_i, $i \in \{1, ..., u\}$ (wobei $\delta_i.name = t_i$), lokalen Parametern $\delta_{i'}$, $i' \in \{u+1, ..., u'\}$, $u' \in \mathbb{N}$, $u' \geq u$, sowie Resultatparameter $\delta_{u'+1}$, dann setze

   i) für alle $\delta_{i'}$, $i' \in \{u+1, ..., u'\}$:

   - wenn $\delta_{i'}.ident = [\,]$: $\delta_{i'}.ident = [\underbrace{n|...|n}_{k \text{ mal}}]$; $n := n + 1$,

   d.h. vergebe für alle lokalen Parameter einen neuen Identifikator und hänge ihn k-mal an die ID-Liste an, wobei k der Länge der ID-Listen der anderen Parameter entspricht. k kann insbesondere auch den Wert 1 haben.

- wenn $\delta_{i'}.ident \neq [\,]$: $\delta_{i'}.ident := [\delta_{i'}.ident|\underbrace{m|...|m}_{k' \text{ mal}}]$,

  d.h. hänge den an θ_j bereits vergebenen Identifikator m k'-mal an die ID-Liste an, wobei k' so gewählt ist, daß die ID-Liste die gleiche Länge wie die ID-Listen der anderen Parameter erhält.

ii) für alle $w \in V$ mit $e_{v,w} \in E$ mit $w.name = q(l_1, ..., l_m)$, $m \in \mathbb{N}$, Übergabeparametern θ_j, $j \in \{1, ..., m\}$ (wobei $\theta_j.name = l_j$), sowie Resultatparameter $\theta_{m'+1}$, $m' \geq m$ und $e_{v,w}.attr = q(l'_1, ..., l'_m)$:

- wenn $e_{v,w} \in \mathcal{E}^v \cup \mathcal{E}^m$:
  - · für alle $j \in \{1, ..., m\}$ so daß $\delta_i.name = t_i = l'_j$:
    $\theta_j.ident := [\theta_j.ident|\delta_i.ident]$, mit $i \in \{1, ..., u'\}$,
    d.h. übernehme die ID-Liste vom korrespondierenden Vorgängerparameter.
  - · $\theta_{m'+1}.ident := [\theta_{m'+1}.ident|\delta_{u'+1}.ident]$.
- wenn $e_{v,w} \in \mathcal{E}^r$:
  - · für alle $j \in \{1, ..., m\}$ so daß $\delta_i.name = t_i = l'_j$:
    $\theta_j.loop_ident := [\theta_j.loop_ident|\delta_i.ident]$, mit $i \in \{1, ..., u'\}$.
  - · $\theta_{m'+1}.loop_ident := [\theta_{m'+1}.loop_ident|\delta_{u'+1}.ident]$.

3. Für den Knoten v in $SG_{\mathcal{L}}$ und für alle Knoten $w \in SG_{\mathcal{L}}$ mit $e_{v,w} \in \mathcal{E}^v \cup \mathcal{E}^m$ rufe 2. mit $v := w$ auf, wenn in w ein neues Tupel von Identifikatoren eingetragen wurde. ◇

Die Blätter des Graphen müssen im Algorithmus nicht berücksichtigt werden, da die Identifikatoren für Parameter grundsätzlich vom Vorgängerknoten aus vergeben werden. Dies kann nur der QUERY-, ein OR-, NOT- oder AND-Knoten sein. Sonderfälle, wie lokale Parameter treten nur an AND-Knoten auf.

Der Algorithmus terminiert, wenn alle Blätter des Graphen erreicht worden sind, oder wenn beim Duchlauf von Schleifen im Graphen keinen neuen Identifikatoren mehr an einen Knoten der Schleifen vergeben werden können (bzw. falls in 3. anstelle von E nur die Menge $\mathcal{E}^v \cup \mathcal{E}^m$ verwendet wird, werden Knoten in Schleifen nur genau einmal passiert).

Satz 3.29

Der Algorithmus gibt die ID-Liste von dem Parameter eines Knotens an genau die korrespondierenden Parameter von Nachfolgeknoten weiter. ◇

Beweis:

i) Der Parameter δ eines OR- oder NOT-Knoten gibt seine Identifikatoren an genau den Parameter θ eines Nachfolgeknoten weiter, wenn dieser die gleiche Position in der Parameterliste hat. Es handelt sich dabei also genau um die korrespondierenden Parameter (siehe Folgerung 3.24 a und b).

ii) Der Parameter δ eines AND- und QUERY-Knoten gibt seine Identifikatoren an genau den Parameter eines Nachfolgeknotens weiter, für den gilt:
- der Parameter δ hat den gleichen Namen wie ein Parameter θ' desjenigen Literals, das der Verbindungskante zugeordnet ist und
- θ' hat die gleiche Position in der Parameterliste wie θ.

Dies sind ebenfalls genau die korrespondierenden Parameter (siehe Folgerung 3.24 c bis e und Bemerkung).

iii) Resultatparameter sind allgemein korrespondierende Parameter (Satz 3.23 iv). ⋄

Folgerung 3.30

Zwei Parameter von beliebigen Knoten haben genau dann den gleichen Identifikator in ihrer ID-Liste, wenn sie korrespondierend sind. ⋄

Beweis:

Folgt aus Satz 3.29 und der Transitivität der Korrespondenz (Def. 3.27). ⋄

Folgende Aussagen lassen sich nun mit Hilfe des Algorithmus beweisen:

Satz 3.31

Für jeden Parameter eines Knotens hat die ID-Liste die gleiche Länge $n \in \mathbb{N}$. ⋄

Beweis:

Der Algorithmus beginnt mit der Wurzel des Graphen, d.h. mit dem QUERY-Knoten.

Wenn kein Parameter eines Knotens einen Identifkator erhalten hat, so erhalten alle Parameter genau einen neuen Identifikator (Def. 3.28 b, i, Teil 2). Damit erhalten auch alle lokalen Parameter genau einen Identifikator (Def. 3.28 b, i, Teil 1).

Die Parameter von Nachfolgeknoten erhalten ihre Identifikatoren durch Weitergabe der Identifikatoren von korrespondierenden Parameter am betrachteten Knoten. Da es für jeden Parameter genau einen korrespondierenden Parameter am Knoten gibt (Satz 3.25), erhält jeder Parameter genau die gleiche Anzahl von Identifikatoren, die der korrespondierende Parameter hat. Hatten am Knoten alle Parameter ID-Listen gleicher Länge, haben dies folglich auch alle Parameter am Nachfolgeknoten. Für lokale Parameter wird zusätzlich eine ID-Liste gleicher Länge erzeugt (Def. 3.28 b, i, Teil 1).

Gibt es aufgrund von Mehrfachreferenzen mehrere Vorgängerknoten, so erhalten alle Parameter die Identifikatoren von den korrespondierenden Parametern dieser Knoten durch einfaches Aneinanderhängen der ID-Listen. Für jeden Parameter gibt es wiederum an jedem Vorgängerknoten genau einen korrespondierenden Parameter. Hatten am Knoten alle Parameter ID-Listen gleicher Länge, so haben dies folglich auch alle Parameter am Nachfolgeknoten. Für lokale Parameter wird zusätzlich eine ID-Liste gleicher Länge erzeugt (Def. 3.28 b, i, Teil 2). ⋄

Folgerung 3.32

Die Parameter eines Nachfolgeknoten haben genau dann längere ID-Listen, als die Parameter des Knotens selbst, wenn der Nachfolgeknoten von mindestens einem weiteren Knoten aus erreichbar ist. ⋄

Beweis:

Hat ein Knoten genau einen Vorgängerknoten, haben die ID-Listen aufgrund der einfachen Weitergabe die gleiche Länge wie die ID-Listen der Parameter am Vorgängerknoten.

Hat ein Knoten mehrere Vorgängerknoten, erhalten die Parameter die Identifikatoren von jedem Vorgängerknoten durch einfaches Aneinanderhängen der ID-Listen. ⋄

Satz 3.33

Es genügt die (wiederholte) Vergabe eines einzigen Identifikators für einen lokalen Parameter. ◇

Beweis:

Die Weitergabe der Identifikatoren an die korrespondierenden Parameter der Nachfolgeknoten ist eindeutig. Daher erfolgt die Weitergabe zweier verschiedener Identifikatoren eines Paramters an genau die gleichen Parameter. Ist die Erzeugungsstelle beider Identifikatoren dieselbe (z.B. der gleiche lokale Parameter), gibt es folglich keinen Parameter an dem nur einer der beiden Identifikatoren auftritt. Einer der beiden Identifikatoren ist also redundant. ◇

Satz 3.34

Die tupelweise Vergabe von Identifikatoren wird bei Schleifen nach endlich vielen Durchläufen periodisch. ◇

Beweis:

Die Anzahl der Parameter und damit die Anzahl der benötigten Identifikatoren im Graphen ist endlich. Damit ist auch die Anzahl der möglichen Permutationen von Identifikatoren endlich. Folglich muß bei Schleifen nach endlich vielen Durchläufen eine Wiederholung eines bereits vorhandenen Tupels von Identifikatoren für die Parameter eines Knotens auftreten, nachdem für alle lokalen Parameter Identifikatoren vergeben sind.

Aufgrund der Eindeutigkeit der Weitergabe an korrespondierende Parameter erhalten die korrespondierenden Parameter an Nachfolgeknoten bei gleichem Ausgangsidentifikator (bzw. bei gleicher ID-Liste) in unterschiedlichen Durchläufen den gleichen Identifikator (bzw. die gleiche ID-Liste).

Damit können nach dem ersten Auftreten einer Wiederholung keine Tupel von Identifikatoren mehr entstehen, die nicht bereits erzeugt wurden.

(Dies garantiert die Terminierung des Algorithmus, wenn in Definition 3.28 $e \in E$ statt $e \in \mathcal{E}^v \cup \mathcal{E}^m$ gilt). ◇

3.3.3 Vervielfältigung von Teilgraphen

Die Parameter eines Knoten besitzen genau dann mehrere Identifikatoren, wenn der Knoten von verschiedenen Knoten aus erreicht werden kann. Alle Parameter eines Knotens haben darüberhinaus gleich lange ID-Listen gewährleistet.

Da bei der Mehrfachreferenzierung eines Knotens von verschiedenen Knoten aus die Parameter unterschiedlich verwendet werden können, z.B. kann einmal eine Ausgabe erwartet werden, ein anderes Mal kann eine Eingabe erfolgen, ist es für eine einfache Weiterverarbeitung des Graphen sinnvoll, die Knoten, bzw. die Teilgraphen, die diese Knoten enthalten, entsprechend der Anzahl der eindeutigen Namen der Parameter zu duplizieren.

Stellt sich später heraus, daß duplizierte Knoten die gleichen Bindungsmuster erhalten, so können sie in einer späteren Optimierung des Datenflußgraphen wieder zusammengefaßt werden (siehe Abschnitt 3.4.5).

Jeder Knoten hat dann Parameter mit genau einem eindeutigen Namen. Die eindeutigen Namen der Knoten in den duplizierten Teilgraphen müssen dabei natürlich derart aufgeteilt werden, daß eine Neuvergabe der eindeutigen Namen für den gesamten Graphen das gleiche Ergebnis liefern würde.

Figur 14 zeigt ein Beispiel als Fortsetzung zum Beispiel “Mehrfachreferenzierung von Knoten” (Abbildung 13) in Abschnitt 3.3.2.

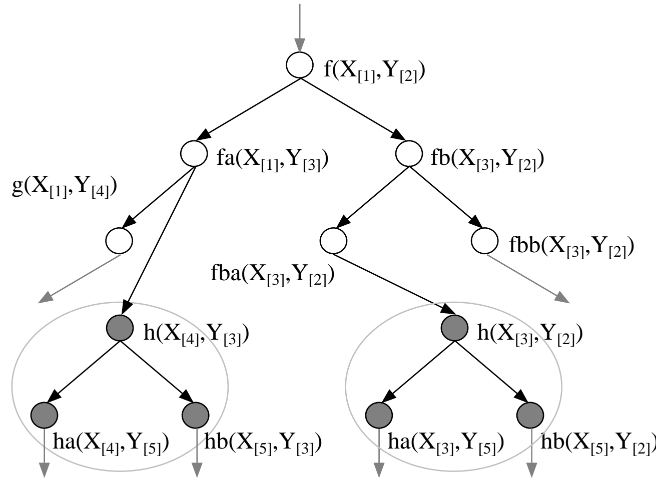


Abbildung 14: Vervielfältigung von Teilgraphen bei Mehrfachreferenzierung

Der oberste Knoten einer Schleife wird von einer Vervielfältigung bezüglich externem Schleifenaufruf und Aufruf aus der Schleife selbst heraus, ausgenommen. Hier steht die Mehrfachreferenzierung nicht für eine mehrfache Verwendung des gleichen Teilgraphen in möglicherweise unterschiedlichen Bedeutungen. Damit können die durch die Mehrfachreferenzierung entstandenen verschiedenen Identifikatoren nicht für eine unterschiedliche Verwendung im Sinne von Ein- oder Ausgabe stehen, sondern nur für die Korrespondenz von Parametern zwischen Aufrufstelle und aufgerufener Stelle. Dieser Sachverhalt wird durch die Aufteilung der Identifikatoren in *idents* und *loop_idents* realisiert.

Bei einer Schleife, für die es mehrere externe Aufrufe gibt, handelt es sich um eine Mischform. Sie muß entsprechend vervielfältigt werden. Dabei erhält jeder Parameter des Knotens am Schleifenbeginn genau zwei Identifikatoren: den Identifikator, der sich aus der korrekten Aufteilung der Mehrfachreferenzierung für die externen Aufrufe ergibt *ident*, und den Identifikator, den der Aufruf der Schleife erzeugt hat *loop_ident*.

Für das korrekte Duplizieren und Aufteilen der eindeutigen Namen sind folgende Voraussetzungen notwendig:

1. Die eindeutigen Namen in den ID-Listen der Parametern eines Knoten müssen einander korrekt zuordenbar sein. Dies ist über die Reihenfolge gewährleistet.

2. Alle ID-Listen eindeutiger Namen von Parametern eines Knoten müssen gleiche Länge haben. Dies ist nach Satz 3.31 garantiert.

Definition 3.35 *Erweiterter Strukturgraph*

Sei $SG_{\mathcal{L}} = (V, E)$ ein Strukturgraph zu einem Logikprogramm $\mathcal{L}$. Es gelten außerdem die Bezeichnungen aus Definition 3.14.

Ein erweiterter Strukturgraph $SG^{E}_{\mathcal{L}} = (V_{SG}, E_{SG})$ sei definiert durch

i) $\forall v \in V$ mit $v.name = p(t_1, ..., t_u)$, $u \in \mathbb{N}$, Parametern δ_i, $i \in \{1, ..., u'+1\}$, $u' \in \mathbb{N}$, $u' \geq u$:
wenn $\delta_i.ident = [\iota_1|...|\iota_k]$, $k \in \mathbb{N}$, dann ist $v_1, ..., v_k \in V_{SG}$, wobei für v_s mit $s \in \{1, ..., k\}$ gilt:

- $v_s.name = v.name$,
- $v_s.type = v.type$,
- v_s hat Parameter $\delta_{s,i}$ mit
  · $\delta_{s,i}.name = \delta_i.name$,
  · $\delta_{s,i}.ident = [\iota_s]$,
  · $\delta_{s,i}.loop_ident = \delta_i.loop_ident$.

v_s heiße zu v äquivalent und v heiße zu v_s äquivalent (i.Z. $v_s \sim v$).

ii) $E_{SG} = E^v \cup E^m \cup E^r$, wobei

- $\forall e_{v,w} \in \mathcal{E}^v \cup \mathcal{E}^m$ mit $v \in V$ mit Parametern δ_i, $i \in \{1, ..., u'+1\}$, $u' \in \mathbb{N}$ und $w \in V$ mit Übergabeparametern θ_j, $j \in \{1, ..., m\}$, sowie Resultatparameter $\theta_{m'+1}$, $m, m' \in \mathbb{N}$, $m' \geq m$:
  wenn $\exists s, s' \in \mathbb{N} : v_s \sim v$, $w_{s'} \sim w$ und $\forall i \in \{1, ..., u'+1\}$ $\exists j \in \{1, ..., m, m'+1\}$ mit $\delta_i \simeq \theta_j$ und $\theta_j.ident$ ist aus $\delta_i.ident$ nach i) entstanden, dann ist $e_{v_s, w_{s'}} \in E^v \cup E^m$.

  Es gelte: wenn $e_{v,w} \in \mathcal{E}^v$, dann ist $e_{v_s, w_{s'}} \in E^v$, wenn $e_{v,w} \in \mathcal{E}^m$, dann ist $e_{v_s, w_{s'}} \in E^m$,

- $\forall e_{v,w} \in \mathcal{E}^r$ mit $w \in V$ mit $v \in V$ mit Parametern δ_i, $i \in \{1, ..., u'+1\}$, $u' \in \mathbb{N}$ und $w \in V$ mit Übergabeparametern θ_j, $j \in \{1, ..., m\}$, sowie Resultatparameter $\theta_{m'+1}$, $m, m' \in \mathbb{N}$, $m' \geq m$:
  wenn $\exists s, s' \in \mathbb{N} : v_s \sim v$, $w_{s'} \sim w$ und es existiert ein Pfad von $w_{s'}$ nach v_s in E_{SG}, dann ist $e_{v_s, w_{s'}} \in E^r$.

  Für $w_{s'}$ wird dann die ID-Liste für die Schleifenkennzeichnung neu gesetzt:
  wenn $w_{s'}.name = q(l_1, ..., l_m)$, $m \in \mathbb{N}$ und $e_{v,w}.attr = q(l'_1, ..., l'_m)$:

  - für alle $j \in \{1, ..., m\}$ so daß $\delta_i.name = l'_j$:
    $\theta_j.loop_ident := [\theta_j.loop_ident|\delta_i.loop_ident]$, mit $i \in \{1, ..., u'\}$.
  - $\theta_{m'+1}.loop_ident := [\theta_{m'+1}.loop_ident|\delta_{u'+1}.loop_ident]$. ◇

Satz 3.36

i) Zu jeder Kante im erweiterten Strukturgraphen gibt es eine Kante im Strukturgraphen zwischen den jeweils äquivalenten Knoten.

ii) Zu jeder Kante im Strukturgraphen gibt es eine Kante im erweiterten Strukturgraphen zwischen den jeweils äquivalenten Knoten.

iii) Kein Knoten im erweiterten Strukturgraphen wird durch vorwärtsgerichtete Kanten mehrfachreferenziert. ◇

Beweis:

i) Zu zeigen ist: zwischen zwei Knoten $v_s, w_{s'} \in V_{SG}$, $s, s' \in \mathbb{N}$ existiert eine Kante $e_{v_s, w_{s'}} \in E_{SG}$, wenn eine Kante $e_{v,w} \in E$ existiert mit $v, w \in V$, $v_s \sim v$, $w_{s'} \sim w$.

   Dies folgt direkt aus Definition 3.35 ii): die Existenz einer Kante in E ist Voraussetzung für die Erzeugung einer Kante in E_{SG}.

ii) Es gibt mindestens eine Kante im erweiterten Strukturgraphen:

   Wenn es $v, w \in V$ mit $e_{v,w} \in E$ gibt, so ist nach Definition 3.28 die ID-Liste von v eine Teilmenge der ID-Liste von w. Es müssen also $w', v' \in V_{SG}$ existieren, wobei die Parameter von w' jeweils genau einen Identifikator aus den ID-Listen der korrespondierenden Parameter von w und die Parameter von v' jeweils genau einen Identifikator aus den ID-Listen der korrespondierenden Parameter von v enthalten. Folglich müssen die Parameter von w' jeweils einen Identifikatoren haben, der aus dem Identifikator eines korrespondierenden Parameter von v' entstanden ist. Damit wird nach Definition 3.35 2. eine Kante $e_{v',w'}$ erzeugt.

iii) Es gibt höchstens eine Kante im erweitereten Strukturgraphen:

   Die Parameter von w' haben jeweils genau einen Identifikator. Da einerseits die Identifikatoren der Parameter von w eindeutig den Identifikatoren der Parameter von v über die Entstehung zuordenbar sind, andererseits die Identifikatoren von v disjunkt auf Knoten $v', v'', ...$ mit $v' \sim v$, $v'' \sim v, ...$ aufgeteilt werden, gibt es zu jedem w' nur ein v', so daß die Identifikatoren von w' aus denen von v' entstanden sind. Es kann daher nur höchstens eine Kante erzeugt werden. ◇

Satz 3.37

Für jeden Pfad im Strukturgraph gibt es einen Pfad im erweiterten Strukturgraph, der über äquivalente Knoten geht. ◇

Beweis:

Bemerkung: Unterschiedliche Pfade im Strukturgraphen entstehen durch

a) Pfade über verschiedene Knoten

b) Pfade über dieselben Knoten, die sich aber aus verschiedenen Kanten zusammensetzen (zwei Knoten können mehrere Kanten verbinden).

zu a) Sei $e_1, ..., e_{n-1}$, $n \in \mathbb{N}$ ein Pfad in $SG_{\mathcal{L}}$, wobei $e_i \equiv e_{v^i, v^{i+1}}$ mit $v^i, v^{i+1} \in V$, $i \in \{1, ..., n-1\}$. Dann existiert zu jedem v^i ein äquivalenter Knoten $v^i_{s_i}$ nach 3.35 i) (analog gibt es $v^{i+1}_{s_{i+1}}$ zu v^{i+1}). Nach Satz 3.36 gibt es zu jeder Kante $e_{v^i, v^{i+1}}$ eine Kante $e_{v^i_{s_i}, v^{i+1}_{s_{i+1}}}$ Damit existiert ein Pfad $e'_1, ..., e'_{n-1}$ in $SG^E_{\mathcal{L}}$ mit $e_i \equiv e_{v^i_{s_i}, v^{i+1}_{s_{i+1}}}$. Daraus folgt: zu jedem Pfad über verschiedene Knoten im Strukturgraphen gibt es einen Pfad im erweiterten Strukturgraphen über äquivalente Knoten.

zu b) Existieren Kanten $(e^1_{v,w}, ..., e^n_{v,w}) \in SG_{\mathcal{L}}$, $n \in \mathbb{N}$ für $v, w \in V$, so ist der Knoten w mehrfachreferenziert. Haben die Parameter von v ID-Listen der Länge l, so gibt es die Knoten $v_1, ..., v_l \in V_{SG}$. Außerdem haben dann die Parameter des Knotens w ID-Listen mindestens der Länge $l * n$ nach Folgerung 3.32. Damit existieren mindestens $w_1, ..., w_{l*n} \in V_{SG}$. Nach Definition 3.28 sind die Identifikatoren der Parameter von w aus den Identifikatoren der Parameter von v entstanden. Daher gilt (gegebenenfalls unter Umordnung der Knoten $w_1, ..., w_{l*n} \in V_{SG}$): $\forall j \in \{1, ..., l\} : e_{v_j, w_{n*(j-1)+1}}, ...,$ $e_{v_j, w_{n*j}} \in E$. Damit existieren für n Kanten von v nach w für jeden zu v äquivalenten Knoten $v_j \in V_{SG}$, $j \in \{1, ..., l\}$ insgesamt n Kanten von v_j zu $w_{n*(j-1)+1}$, ..., von v_j zu w_{n*j}. Daraus folgt: zu jedem Pfad über verschiedene Kanten im Strukturgraphen gibt es einen Pfad im erweiterten Strukturgraphen über äquivalente Knoten. ◇

Satz 3.38

Wird kein Knoten im Strukturgraph $SG_{\mathcal{L}}$ mehrfachreferenziert, dann ist der erweiterte Strukturgraph $SG^{E}_{\mathcal{L}}$ identisch zu $SG_{\mathcal{L}}$. $\diamond$

Beweis:

Behauptung: Wird kein Knoten im Strukturgraph $SG_{\mathcal{L}}$ mehrfachreferenziert, so hat jeder Parameter genau einen Identifikator in der IDListe.

Beweis: Die Parameter des QUERY-Knotens erhalten laut Algorithmus 3.28 genau einen Identifikator. Der Rest folgt aus Folgerung 3.32: da keine Knoten mehrfachreferenziert werden, hat die ID-Liste eines Parameters eines Knotens die gleiche Länge wie die ID-Liste des korrespondierenden Parameters am Vorgängerknoten.

Behauptung: $V = V_{SG}$

Beweis: Da jeder Parameter genau einen Identifikator hat, wird nach Definition 3.35 i) zu $v \in V$ genau $v_1 \in V_{SG}$ erzeugt.

Behauptung: $E = E_{SG}$

Beweis:

i) für $e_{v,w} \in E^v \cup E^m$ mit $v, w \in V$ gibt es genau $v', w' \in V_{SG}$ mit $v' \sim v$, $w' \sim w$ und damit wird nach Definition 3.35 ii) genau eine Kante $e_{v',w'} \in E_{SG}$ erzeugt.

ii) für $e_{v,w} \in E^r$ mit $v, w \in V$ gibt es ebenfalls genau $v', w' \in V_{SG}$ mit $v' \sim v$, $w' \sim w$. Außerdem existiert ein Pfad von w nach v in E (nach Konstruktion der Menge $E^v \cup E^m$) und damit existiert nach i) ein Pfad von w' nach v' in E_{SG}. Damit wird nach Definition 3.35 ii) genau eine Kante $e_{v',w'} \in E_{SG}$ erzeugt. $\diamond$

Bemerkung: In [Cremers 94] werden Knoten des Dependency Graphen vervielfältigt, um zu vermeiden, daß für ein Literal mehrfache Bindungsmuster erzeugt werden. Dieses Verfahren behandelt jedoch nicht vollständig alle bei Rekursionen oder bei mehrfacher Referenzierung von Regeln auftretenden Situationen, sondern nur diejenigen, die bei nicht rektifizierten Regelrümpfen entstehen können.

3.3.4 Analyse der Parametermodi

Das folgende Kapitel befaßt sich mit der Bestimmung der (Ein-/Ausgabe-)Modi für die Parameter der Knoten. In den meisten Fällen kann der Modus eines Parameters lokal bestimmt werden, d.h. durch die Betrachtung der Verwendung dieses Parameters in einer Regel, bzw. an von Literalen dieser Regel referenzierten Regeln oder Fakten. Auf den Graphen bezogen bedeutet das die Bestimmung entweder am Knoten selbst oder durch die Betrachtung von Nachfolgeknoten.

Bei einer Zuweisung müssen z.B. alle Parameter auf der rechten Seite des Zuweisungssymbols den Modus Eingabe haben, der Parameter auf der linken Seite hat dagegen den Modus Ausgabe. Bei einem Vergleich müssen alle Parameter sowohl auf der linken, als auch auf der rechten Seite des Vergleichssymbols den Modus Eingabe haben. Die Parameter eines IFACTS haben aufgrund ihrer Schnittstellenfunktion immer den Modus Ausgabe.

Es gibt jedoch Situationen, in denen der Modus eines Parameters nicht lokal bestimmt werden kann. Z.B. ist bei einem FACT nicht lokal bestimmbar, welches Vorkommen eines

Parameters eine Eingabe erwartet und über welches diese Eingabe wieder ausgegeben wird. Um den Modus eines Parameters dennoch bestimmen zu können, müssen andere Vorkommen des Parameters im Graphen betrachtet werden.

In Abschnitt 3.3.4.5 wird gezeigt, daß der vorgeschlagene Algorithmus zur Modusbestimmung korrekt ist und, sofern die DATALOG$_f^{\neg}$-Programme, die als Ausgangsspezifikation dienen, den in Definition 3.1 genannten Forderungen genügen, sie garantiert auswertbar sind. Dies gilt insbesondere auch dann, wenn alle alle Parameter der Anfrage den Modus Ausgabe haben, wie es im Fall der dieser Arbeit zugrunde liegenden Anwendungen generell zutrifft (siehe Kapitel 1.1.2.6).

3.3.4.1 Einordnung

Bekannte Strategien zur Bestimmung von Parametermodi auf Dependency Graphen für Top-Down Verfahren gehen von der Wurzel des Graphen mit Tiefensuche von links nach rechts durch den Graphen, und stellen dabei fest, an welchen Stellen Bindungen für einen Parameter erzeugt werden. Bei dieser Strategie wird die Reihenfolge der Literale im Rumpf von Regeln in ihrer Aufschreibung von "links nach rechts" betrachtet. Bottom-Up Verfahren, die eine Bestimmung von Parametermodi benötigen, lehnen sich bei der Bestimmung der Parametermodi zumeist an Top-Down Verfahren an, es gilt also auch hier die Reihenfolge "links nach rechts". Dies geschieht im allgemeinen mit Hilfe der SIP-Strategie unter Bezug auf eine bestimmte Abfrage, bzw. ein bestimmtes Bindungsmuster für die Abfrage [Debray 88, Debray 89].

Für die Bestimmung der Parametermodi bedeutet dies (für Parameter, die nicht durch eine Werteübergabe in der Abfrage bereits gebunden sind): Das erste Auftreten eines Parameters (von links gesehen) im (Dependency-)Graphen muß eine Ausgabe nach "oben", also an die anderen Knoten des Graphen, sein. Alle weiteren Auftreten sind dann gebunden, d.h. erhalten eine Eingabe.

Aus folgenden Gründen wird in dieser Arbeit jedoch darauf verzichtet, die Reihenfolge der Regeln untereinander und die Reihenfolge der Literale im Rumpf von Regeln in die Analyse der Parametermodi miteinzubeziehen:

- Bei Verfahren, die alle Lösungen finden sollen, darf die Reihenfolge von Regeln keine Rolle spielen.
- Die Ableitung der Parametermodi aus der Reihenfolge von Literalen im Rumpf von Regeln übergibt dem Programmierer eine große Verantwortung für ein hinsichtlich dieser Reihenfolge korrektes Programm. Es ist daher von Vorteil Programme mit beliebiger Reihenfolge von Regeln und Literalen in Regeln zuzulassen.
- Bei einer Extraktion von Funktionen und Operationen aus Parameterlisten in eigene Literale kann, wie in Abschnitt 3.1.3 erläutert, keine einer korrekten Reihenfolge entsprechenden Einordnung der neuen Literale erfolgen. Auf eine gleichzeitige Durchführung beider Verfahren sollte jedoch hinsichtlich der bei einer Kombination entstehenden Komplexität der Algorithmen verzichtet werden.

Anderersetits können die Modi für die Parameter nicht beliebig vergeben werden. Da z.B. zumindest bei Systemprädikaten Bindungsmuster vorgegeben sind, entstehen Bedingungen für die Modi anderer Parameter, denen nur mit einem Analyseverfahren, das graphweite Zusammenhänge berücksichtigt, Rechnung getragen werden kann.

Das Verfahren ist daher allgemeiner und stellt eine Erweiterung der aus der Literatur bekannten SIP-Verfahren dar.

3.3.4.2 Lokale Analyse

Im vorliegenden Anwendungsfall sind Abfragen fixiert und enthalten außerdem nur Variablen (siehe Kapitel 1.1.2). Bindungen für Parameter können nur in Blättern und in wenigen Ausnahmefällen (Konstanten in AND-Knoten) an inneren Knoten erzeugt werden.

Bemerkung: Da aufgrund der Vergabe von Identifikatoren für alle Parameter eindeutig und vollständig erkannt wird, welche anderen Vorkommen ein Parameter hat, ist eine vollständige Rektifizierung der Regelrümpfe wie in [Ullman 88] zur Bestimmung vollständiger und korrekter Bindungsmuster unnötig.

Die Analyse basiert auf dem erweiterten Strukturgraphen. Zunächst soll der Algorithmus informell angegeben werden:

1. **IFACT-Knoten:**

   IFACT-Knoten dienen als Schnittstelle zu einer extensionalen Datenbank. Die Parameterliste eines solchen Knotens enthält mindestens eine Variable, jedoch keine Konstanten. Da über die Variablen Daten in den Graphen eingegeben werden, haben diese den Modus Ausgabe.

2. **FACT-Knoten:**

   FACT-Knoten sind Fakten des ursprünglichen DATALOG$^{\neg}_{f}$-Programms, die keine Ausgabe der extensionalen Datenbank darstellen (siehe Abschnitt 3.1.2). Für sie gilt, daß die Parameterliste leer sein, oder Konstanten und Variablen enthalten kann. Gelten die Konsistenzbedingungen laut Definition 3.2, so kommt jede Variable genau zweimal in der Parameterliste vor.

   Mindestens ein Vorkommen einer Variablen muß eine Eingabe erhalten. Diese Eingabe kann über das andere Vorkommen ausgegeben werden, oder wird, falls das zweite Vorkommen ebenfalls eine Eingabe erhält, mit dieser verglichen. Es können jedoch nicht beide Vorkommen eine Ausgabe liefern, da es in diesem Fall keine Entstehungsmöglichkeit für die Ausgabe gibt. Welches Vorkommen welchen Modus hat, läßt sich jedoch nicht lokal, sondern nur unter Einbeziehung anderer Verwendungsstellen dieser Parameter bestimmen.

   Konstanten dienen zunächst generell als Ausgabe.

3. **FUNC-Knoten:**

   Der Modus des Parameters wird aus der Funktionsspezifikation, d.h. der Deklaration des entsprechenden Systemprädikates (z.B. in Header-Dateien zu Bibliotheken) übernommen. Es spielt keine Rolle, ob der Parameter eine Variable oder eine Konstante ist.

   Einige Systemprädikate erlauben verschiedene Bindungsmuster. Welches Bindungsmuster gültig ist, entscheidet sich durch die Art der an sie übergebenen Parameter: gebunden oder nicht gebunden. Dafür müssen ebenfalls andere Verwendungsstellen dieser Parameter einbezogen werden. Die Randbedingung dabei ist, daß das Ergebnis dieser Betrachtung ein zulässiges Bindungsmuster ergeben muß.

4. **AND-Knoten:**

Eine Konstante hat den Modus Ausgabe, da sich bei einer Konstante im Kopf einer Regel kein Bezug zu einem Parameter im Rumpf der Regel herstellen läßt. Dagegen kann ihr Wert jedoch an Vorgängerknoten weitergegeben werden.

Für eine Variable bestimmt sich zunächst ihr Modus aus dem Vorkommen der Variablen in den Nachfolgeknoten (gegebenenfalls unter Verwendung der Modi aus der globalen Bestimmung der Parametermodi, siehe nächster Abschnitt):

i) Wenn alle Vorkommen an Unterknoten den Modus Ausgabe haben, so bekommt die Variable ebenfalls den Modus Ausgabe zugewiesen, da über sie die Werte nach oben weitergegeben werden müssen.

ii) Wenn alle Vorkommen an Unterknoten den Modus Eingabe haben, so bekommt die Variable ebenfalls den Modus Eingabe zugewiesen, da über sie die Werte nach unten weitergegeben werden müssen.

iii) Hat die Variable an den Nachfolgeknoten sowohl den Modus Ausgabe, als auch den Modus Eingabe, so erhält sie den Modus Ausgabe, da die Ausgabe eines Nachfolgeknotens als Eingabe für einen anderen Nachfolgeknoten verwendet werden kann.

iv) In einem rekursiven Aufruf ist der Modus einer Variablen nicht bestimmbar, wenn sie sich über die Rekursion selbst referenziert. Hier müssen zusätzlich andere Vorkommen des Parameters betrachtet werden. Dies wird in Abschnitt 3.4.4.2 erläutert.

v) Kommt eine Variable nicht in Nachfolgeknoten vor, kann sie keine Ausgabe von dort erzeugten Werten sein. Sie erhält daher den Modus Eingabe.

Bemerkung: Nach der Elimination doppelter Variablen aus Parameterlisten (Def. 3.4) kann eine Variable nur jeweils genau einmal in der Parameterliste vorkommen.

5. **OR-Knoten:**

Hat ein Parameter an allen Nachfolgeknoten den gleichen Modus, so erhält er am OR-Knoten ebenfalls diesen Modus, um Werte in die entsprechende Richtung weitergeben zu können.

Tritt der gleiche Parameter sowohl mit dem Modus Ausgabe als auch mit dem Modus Eingabe auf, so ist der resultierende Modus ebenfalls Eingabe, da die Funktionalität eines OR-Knotens nur die gemeinsame Weitergabe von Werten in entweder die eine oder die andere Richtung bewirken kann.

Der Modus des betreffenden Parameters der Nachfolgeknoten (nur AND-, FACT- und IFACT-Knoten) an denen er den Modus Ausgabe hatte, wird dann explizit auf Eingabe gesetzt. Die Nachfolgeknoten erhalten dann an dieser Stelle von ihren Nachfolgeknoten Ausgaben und von anderen Teilgraphen Eingaben und übernehmen an dieser Stelle den Join. Wie in Abschnitt 3.7 gezeigt wird, ändert sich dadurch nichts an dem durch den Graphen berechneten Ergebnis.

6. **QUERY-Knoten:**

Für einen QUERY-Knoten, der den (leeren) Kopf der Abfrage repräsentiert, gilt das gleiche wie für AND-Knoten (die den Kopf normaler Regeln darstellen).

7. **NOT-Knoten:**

Ein NOT-Knoten darf ausschließlich Eingaben erhalten. Damit dies möglich ist, benötigt man die Voraussetzung bedingt sicherer Programme. Die Modi der Parameter des Nachfolgeknotens bleiben von dieser Modusfestlegung unbeeinflußt.

Sinnvollerweise startet die Analyse für die inneren Knoten an den Blättern des Graphen und geht jeweils von den Knoten zu den Vorgängerknoten.

Im folgenden soll der Algorithmus formal angegeben werden.

Definition 3.39 *Ein- und Ausgabemodi von Knotenparametern*

Sei $SG^{E}_{\mathcal{L}} = (V_{SG}, E_{SG})$ ein erweiterter Strukturgraph.

Die Modi, die an die Parameter vergeben werden, seien definiert als: $Modes = \{$in, out, undef, unknown$\}$. Dabei stellt undef einen Fehlerfall dar, unknown einen zunächst noch nicht bestimmbaren Modus.

Für $v \in V_{SG}$ mit $v.name = p(t_1, ..., t_u)$ (bzw. $p_f(t_1, ..., t_u)$), mit Parametern $\delta_1, ...,$ $\delta_{u'+1}$ und $u, u' \in \mathbb{N}$, $u' \geq u$. bezeichne $\delta_i.mode$ den Typ des Parameters δ_i von v für $i \in \{1, ...u'+1\}$.

Dann bestimme sich $\delta_i.mode$ für alle $i \in \{1, ...u'\}$ wie folgt:

1. $v.type$ = IFACT (Schnittstelle zum Datenspeicher):
   $\delta_i.mode :=$ out.

2. $v.type$ = FACT (normales Fakt)
   - wenn $\delta_i.name = a$, a Konstante,
     dann ist $\delta_i.mode :=$ out,
   - wenn $\delta_i.name = X$, X Variable und $\exists j \in \{1, ..., u\} : \delta_j.name = \delta_i.name$,
     dann ist $\delta_i.mode :=$ unknown,
   - sonst ist $\delta_i.mode :=$ undef.

3. $v.type$ = FUNC
   wenn p ein Systemprädikat ist und den Aufruf von f darstellt,
   wobei f definiert sei durch $f(l_1, ..., l_u)$ mit Übergabeparametern $\theta_1, ..., \theta_u$ und Bindungsmustern $(\theta_1.modes_s, ..., \theta_u.modes_s)$ für $s \in \{1, ..., t\}$, $t \in \mathbb{N}$.
   - ist $t > 1$, d.h. gibt es für $f(l_1, ..., l_u)$ mehrere Bindungsmuster,
     und $\exists s, s' \in \{1, ..., t\}$, $s \neq s'$ mit $\theta_i.modes_s \neq \theta_i.modes_{s'}$
     dann ist $\delta_i.mode :=$ unknown,
   - sonst ist $\delta_i.mode := \theta_i.mode_1$,

   Die Menge der im weiteren verwendbaren Bindungsmuster ist dann:
   $\{(\theta_1.modes_s, ..., \theta_u.modes_s) \mid s \in \{1, ..., t\}, \forall i \in \{1, ..., u\} : \delta_i.mode = \text{in} \Rightarrow$
   $\theta_i.modes_s = \text{in}\}$.

4. $v.type$ = AND und $v.type$ = QUERY
   (hier gilt im allgmeinen: $u' > u$ mit $\delta_{u+1}, ..., \delta_{u'}$ lokale Parameter)
   - wenn $\delta_i.name = a$, a Konstante,
     dann ist $\delta_i.mode :=$ out,
   - wenn $\delta_i.name = X$, X Variable und $\exists w$ mit $e_{v,w} \in E_{SG}$, $w.name = q(l_1, ..., l_m)$, $m \in \mathbb{N}$ und Übergabeparametern $\theta_1, ..., \theta_m$ mit $\exists j \in \{1, ..., m\}$ $\delta_i.ident = \theta_j.ident$
     i) mit $\theta_j.mode =$ undef,
        dann ist $\delta_i.mode :=$ undef,
     ii) mit $\theta_j.mode =$ unknown und Bedingung i) ist nicht erfüllbar,
        dann ist $\delta_i.mode :=$ unknown,
     iii) mit $\theta_j.mode =$ in und Bedingungen i) und ii) sind nicht erfüllbar,
        dann ist $\delta_i.mode :=$ in,

iv) mit $\theta_i.mode = \mathsf{out}$ und Bedingungen i), ii) und iii) sind nicht erfüllbar, dann ist $\delta_i.mode := \mathsf{out}$,

- wenn $\delta_i.name = X$, X Variable und $\nexists w$ mit $e_{v,w} \in E_{SG}$, $w.name = q(l_1, ..., l_m)$, $m \in \mathbb{N}$ und Übergabeparametern $\theta_1, ..., \theta_m$ mit $\exists j \in \{1, ..., m\}$ $\delta_i.name = \theta_j.name$ dann ist $\delta_i.mode := \mathsf{in}$,
- $\delta_i.mode := \mathsf{out}$, für $i \in \{u+1, ..., u'\}$,

5. $v.type = $ OR
   wenn $\exists w$ mit $e_{v,w} \in E_{SG}$, $w.name = p(l_1, ..., l_u)$ und Übergabeparametern $\theta_1, ..., \theta_u$
   i) mit $\theta_i.mode = \mathsf{undef}$,
   dann ist $\delta_i.mode := \mathsf{undef}$,
   ii) mit $\theta_i.mode = \mathsf{unknown}$ und Bedingung i) ist nicht erfüllbar,
   dann ist $\delta_i.mode := \mathsf{unknown}$,
   iii) mit $\theta_i.mode = \mathsf{in}$ und Bedingungen i) und ii) sind nicht erfüllbar,
   dann ist $\delta_i.mode := \mathsf{in}$;
   iv) mit $\theta_i.mode = \mathsf{out}$ und Bedingungen i), ii) und iii) sind nicht erfüllbar,
   dann ist $\delta_i.mode := \mathsf{out}$;

   Wenn $\delta_i.mode = \mathsf{in}$ und $\exists w$ mit $e_{v,w} \in E_{SG}$, $w.name = p(l_1, ..., l_u)$ und Übergabeparametern $\theta_1, ..., \theta_u$ mit $\theta_i.mode = \mathsf{out}$, dann wird $\theta_i.mode := \mathsf{in}$,

6. $v.type = $ NOT
   $\delta_i.mode := \mathsf{in}$.

Für den Ergebnisparameter $\delta_{u'+1}$ gilt:
$\delta_{u'+1}.mode := \mathsf{out}$. ◇

Damit ist nun für jeden Parameter, dessen Modus lokal eindeutig bestimmbar ist, dieser festgelegt. Da es bei der Festlegung keine Alternativen gibt, ist das Ergebnis des Algorithmus korrekt (siehe auch Abschnitt 3.3.4.5).

3.3.4.3 Globale Analyse

Es gibt Situationen, in denen der Modus eines Parameters nicht lokal bestimmt werden kann. Z.B. ist bei einem FACT nicht lokal bestimmbar, welches Vorkommen eines Parameters eine Eingabe erwartet und über welches diese Eingabe wieder ausgegeben wird.

Bemerkung: Dies trifft nach Algorithmus 3.39 jedoch grundsätzlich nicht für lokale Parameter und für Ergebnisparameter zu.

Um den Modus eines Übergabeparameters auch dann bestimmen zu können, wenn er nicht lokal ableitbar ist, müssen andere Vorkommen des Parameters im Graphen hinsichtlich ihrer Verwendung betrachtet werden. Aufgrund der Vergabe von Identifikatoren sind alle zu einem Parameter korrespondierenden Vorkommen nun einfach zu finden.

Da sich die Modi der Parameter der inneren Knoten bis auf wenige Ausnahmen von den Modi der Parameter an den Nachfolgeknoten ableiten, darf die Bestimmung der Parametermodi im globalen Zusammenhang zunächst nur die Blätter des Graphen berücksichtigten. Für die Ausnahmen gilt, daß der Modus bereits bekannt ist und daher eine Bestimmung des Modus im globalen Zusammenhang nicht nötig ist, wie im Beweis zu Satz 3.47 näher erläutert wird.

Sind die Modi an den Blättern vollständig bestimmt, können mit Definition 3.39 auch die Modi an den inneren Knoten bestimmt werden.

In diesem Kapitel werden die in deduktiven Datenbanken üblichen Bezeichnungen b (*bound*) und f (*free*) für Modi verwendet, um sie von den im Strukturgraphen vergebenen Modi für Parameter in und out zu unterscheiden.

Informell motiviert und bestimmt sich die Vorgeschichte eines Parameters folgendermaßen: Wird an einer beliebigen Stelle im Graphen, z.B. an einem Blatt des Graphen, eine Belegung für den Parameter erzeugt, so kann diese Belegung an allen Stellen verwendet werden, an denen der Parameter sonst vorkommt. Der Parameter könnte also, falls nötig, an diesen Stellen den Modus Eingabe erhalten. Dies gilt jedoch nur, wenn sich die Vorkommen nicht in alternativen Zweigen des Graphen befinden: der unterste Knoten aller gemeinsamen Vorgängerknoten darf kein OR-Knoten sein, da die Ausgabe des einen Zweiges grundsätzlich nicht in einem alternativen Zweig verwendet werden kann. Gibt es hingegen nur ein Vorkommen des Parameters im Graphen, oder keine Stelle an der eine Belegungen erzeugt wurde, die nicht mit dem aktuellen Vorkommen über einen OR-Knoten verbunden ist, muß der Parameter den Modus Ausgabe erhalten.

In reinen Bottom-Up Verfahren wird jeder Parameter an einem Blatt des Graphen als Ausgabe angesehen. Dies erzeugt insbesondere bei der Anwendung von den Systemprädikaten Probleme, die möglicherweise ohne eine Beschränkung durch Eingaben unendlich große Ausgabemengen erzeugen können. Im vorliegenden Modell sollen die Systemprädikate, sowie die FACT-Knoten bei denen Eingaben benötigt werden, tatsächlich Eingaben erhalten.

Darüberhinaus ist, da die Aufschreibungsreihenfolge der Literale und Regeln für dieses Verfahren keine Rolle spielen soll, für Parameter, die bei mehreren Vorkommen einen unbekanntem Modus haben (und keinerlei weiteren Randbedingungen unterliegen), die Auswahl einer Stelle für die Ausgabe beliebig. In Anlehnung an das SIP-Verfahren würde es sogar genügen, für jeden Parameter an einer Stelle eine Ausgabe zu erzeugen und diese an allen anderen Stellen als Eingabe zu verwenden.

Der Vorteil dabei ist, daß die Knoten, an denen der Parameter den Modus Eingabe besitzt, eine Einschränkung auf ihre Ausgabemenge (Anzahl der Tupel) erfahren. Dadurch sind theoretisch weniger Berechnungen an Nachfolgeknoten nötig. Der Preis dafür ist jedoch, daß man damit gleichzeitig die mögliche Parallelität der Abarbeitung im Graphen einschränkt.

Bei der Steuerung autonomer Systeme sind Eingaben grundsätzlich konkreter Natur, d.h. pro Eingabemöglichkeit existiert pro Zeiteinheit genau ein zuordenbarer Wert anstelle einer Menge alternativer Werte. Da sich das Modell, das in dieser Arbeit vorgestellt wird, in erster Linie auf solche Systeme bezieht, ist der Aspekt einer hohen Parallelität gegenüber einer Einschränkung von Belegungsmengen vorzuziehen. Es sollen daher nur an den Knoten Eingaben verwendet werden, an denen sie unbedingt notwendig sind.

Definition 3.40 *Modus-zyklische Datenabhängigkeiten*

Sei $SG^{E}_{\mathcal{L}} = (V_{SG}, E_{SG})$ ein erweiterter Strukturgraph zu einem Logikprogramm $\mathcal{L}$.

Eine modus-zyklische Datenabhängigkeit liegt vor, wenn $\exists v_1, ..., v_n \in V_{SG}$ mit $n \in \mathbb{N}$ und für alle $i \in \{1, .., n-1\}$ gilt: v_i hat Parameter δ_i, v_{i+1} Parameter θ_{i+1}, so daß $\delta_i.mode = \mathsf{out}$, $\theta_{i+1}.mode = \mathsf{in}$ und $\delta_i.ident = \theta_{i+1}.ident$, sowie v_n hat Parameter δ_n, v_1 Parameter θ_1, so daß $\delta_n.mode = \mathsf{out}$, $\theta_1.mode = \mathsf{in}$ und $\delta_n.ident = \theta_1.ident$. $\diamond$

Bemerkung 3.41

1. Für die Überprüfung, bzw. Vermeidung von Modus-Zyklen ist die Betrachtung der Modi der Parameter an den Blättern der Graphen ausreichend, da sich alle Modi (bis auf wenige Sonderfälle) von Parametern an inneren Knoten von diesen ableiten und damit für die Parameter an inneren Knoten äquivalente Abhängigkeiten bestehen. Dies gilt nach Satz 3.47.
2. Die Datenabhängigkeiten in einer Schleife des Strukturgraphen sind nicht modus-zyklisch, da sich die Parametermodi des Knotens, an dem die rückführende Kante beginnt, von den Modi der Parameter des Knotens, an dem die Kante endet, ableiten. Die Behauptung gilt aufgrund von Satz 3.48.

Die Begriffsprägung "modus-zyklische Datenabhängigkeiten" dient dazu, um von zyklischen Datenabhängigkeiten in Rekursionen zu unterscheiden. Im Unterschied zu Rekursionen würden sie bei einer Berechnung eine Verklemmung verursachen: es wird unendlich lange auf eine Eingabe gewartet, die nie kommen kann, da zur Erzeugung der Eingabe das Ergebnis der Berechnung benötigt wird. Diese Art der Verklemmung kann bei Rekursionen nicht auftreten. Für eine Rekursion kann es nur genau zwei Fälle geben: es gibt nur Ausgaben oder es gibt mindestens eine Eingabe. Für den Fall, daß ein Knoten der Rekursion einen Eingabe benötigt, muß es jedoch mindestens einen Knoten außerhalb der Schleife geben, von dem die Eingabe kommt, da diese die Rekursion überhaupt erst anstößt.

Die fehlenden Modi bei Parametern an Blättern des Graphen sind unter folgenden Randbedingungen beliebig wählbar:

Definition 3.42 *Konsistenzbedingungen für die Vergabe der Parametermodi*

i) Für jeden Parameter muß es ein Vorkommen mit Modus Ausgabe geben. (Da im vorliegenden Anwendungsfall über die Abfrage keine Werte eingegeben werden, müssen Werte für Parameter an Knoten des Graphen, insbesondere an Blättern des Graphen erzeugt werden.)

ii) Die Bindungsmuster an einem Knoten müssen einem dort zulässigen Bindungsmuster entsprechen.

iii) Es dürfen keine modus-zyklischen Datenabhängigkeiten existieren. ◇

Bedingung i) entspricht der allgemeinen Konsistenzbedingung für $\text{DATALOG}_f^{\neg}$-Programme (Def. 3.1).

Würde die Wahl der Modi nur die grundlegenden Randbedingungen i) und ii) berücksichtigen, so können auch in einem Graphen, der auf einem korrekten $\text{DATALOG}_f^{\neg}$-Programm basiert, modus-zyklische Datenabhängigkeiten entstehen: wenn ein Parameter mit unbekanntem Modus an einem Knoten den Modus Ausgabe erhält und dadurch am gleichen Knoten den Modus Eingabe für einen anderen Parameter erzwungen wird, so kann dann möglicherweise ein Zyklus von Ausgaben und Eingaben entstehen.

Zusätzlich könnten noch folgende Kriterien eine Rolle spielen:

iv) eine Analyse der Größe der Belegungsmenge an einem FACT-Knoten: je kleiner, desto besser.

v) Das "bessere Bindungsmuster" an FUNC-Knoten: z.B. ist bei $p_+(Xres, X1, X2)$ das Bindungsmuster $p_+(Xres^f, X1^b, X2^b)$ sinnvoller als $p_+(Xres^b, X1^f, X2^f)$.

Da im Anwendungsfall die Belegungsmenge bei Fakten grundsätzlich die Größe eins hat (genau ein Wert je Parameter), soll im Rahmen dieser Arbeit, soweit es möglich ist, die Wahl eines Bindungsmusters anhand des zweiten Kriteriums getroffen werden, ansonsten aber eine "zufällige", d.h. nichtdeterministische Auswahl genügen.

Das folgende Beispiel veranschaulicht die Situation an einem Programm und in Form eines Graphen, in dem die durch die Modi vorgegebenen Abhängigkeiten mit Hilfe von Kanten eingezeichnet sind.

Beispiel: Modus-Zyklen in einem (nicht ganz sinnvollen) Programm

$f(X, Y, Z) \leftarrow g(X, Y), h(X, Y, Z).$
$h(X, Y, Z) \leftarrow X = Y * Z, Z = Y + X.$
$g(0, 0).\ g(1, -1).\ g(2, 2).\ g(3, -3).$

Die Bindungsmuster der Literale $X = Y * Z$ und $Z = Y + X$ können jeweils sein: fbb, bff, sowie bbb. Die Modi von X, Y und Z in der zweiten Regel sind daher nicht lokal bestimmbar.

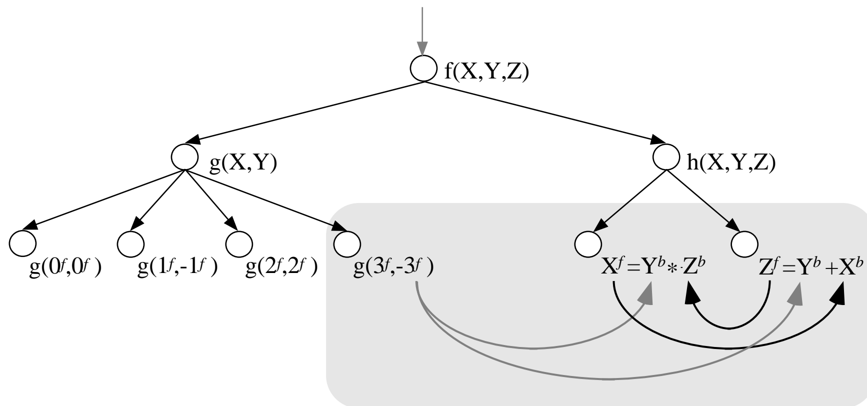


Abbildung 15: Beispiel für modus-zyklische Datenabhängigkeiten

Setzt man den Modus für X im Literal $X = Y * Z$ auf Ausgabe (f), so sind damit die Modi für Y und Z auf Eingabe (b) festgelegt. Nun muß es aber für Z eine Ausgabestelle geben. Die einzige Möglichkeit dafür ist das Literal $Z = Y + X$, also muß dort Z den Modus Ausgabe erhalten. Damit sind in diesem Literal X und Y auf Eingabe festgelegt. X im zweiten Literal erhält dann, zusätzlich zur Eingabe von außerhalb, eine Eingabe vom ersten Literal. Man erhält die in Fig. 15 gezeigten modus-zyklischen Datenabhängigkeiten (wobei nur grau unterlegte Knoten betrachtet werden).

Setzt man dagegen den Modus für X im Literal $X = Y * Z$ auf Eingabe, da an einer anderen Stelle im Programm bereits eine Ausgabe für X erzeugt wird, hat man für das erste Literal die Bindungsmuster bff und bbb zur Auswahl. Für das zweite Literal stehen nun die Bindungsmuster fbb und bbb zur Auswahl, da für Y und Z in beiden Fällen eine Ausgabe erzeugt wird. Figur 16 zeigt ein Beispiel für korrekte Bindungsmuster.

Die folgende Tabelle zeigt die Situation für verschiedene Bindungsmuster. Andere Kombinationen sind nicht möglich, da für Z an keiner anderen Stelle außer in einem der beiden Literale eine Ausgabe erzeugt wird.

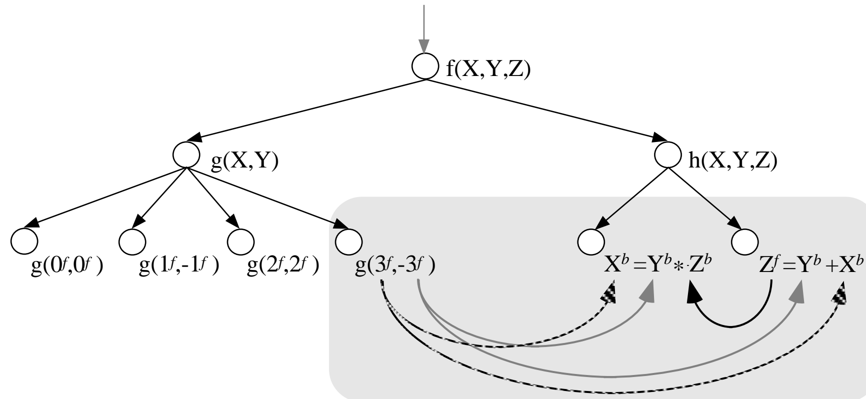


Abbildung 16: Beispiel ohne modus-zyklische Datenabhängigkeiten

$X = Y * Z$	$Z = Y + X$	Abhängigkeit
fbb	fbb	modus-zyklisch
bff	bff	modus-zyklisch
bff	fbb	korrekt
bff	bbb	korrekt
bbb	fbb	korrekt

◇

Der folgende Algorithmus behandelt die globale Vergabe von Modi für Parameter unter Vermeidung modus-zyklischer Datenabhängigkeiten:

Schritt 1: Vergabe ohne Wahlmöglichkeit

Zunächst sollen die Fälle betrachtet werden, in denen es für das Setzen eines noch unbestimmtem Modus für einen Parameter aufgrund der Situation an anderen Knoten des Graphen keine Alternativen gibt:

i) Gibt es keine weiteren Vorkommen des Parameters, so muß der Parameter den Modus Ausgabe erhalten.

ii) Gibt nur Vorkommen an Knoten, die nicht über Or-Knoten mit dem aktuellen Knoten verbunden sind, so muß der Parameter den Modus Ausgabe erhalten.

iii) Haben alle nicht durch Or-Knoten verbundene Vorkommen den Modus Eingabe, so muß der Parameter ebenfalls den Modus Ausgabe erhalten.

iv) Ergibt sich aus einer Bestimmung einzelner Modi an Knoten mit Bindungsmuster-Alternativen ein festgelegtes Bindungsmuster, so müssen daraus die restlichen Modi für die anderen Parameter des Knotens abgeleitet werden.

Bemerkung: In diesen Fällen gibt es offensichtlich keine Wahlmöglichkeit.

Übrig bleiben folgende Fälle, bei denen eine Wahlmöglichkeit besteht:

v) Zu einem Parameter mit unbekanntem Modus existiert mindestens eine Ausgabestelle. Er kann damit sowohl den Modus Ausgabe, als auch den Modus Eingabe erhalten.

vi) Zu einem Parameter mit unbekanntem Modus existiert keine Ausgabestelle, aber mindestens eine weitere Stelle mit unbekanntem Modus. Einer der beiden Parameter kann also sowohl den Modus Ausgabe, als auch den Modus Eingabe erhalten.

Bei der Wahl der Modi muß nun darauf geachtet werden, daß keine modus-zyklischen Datenabhängigkeiten erzeugt werden. Konsequenterweise muß dies mit einer Vermeidungsstrategie geschehen, statt mit einem einfachen, aber bezüglich der Ausführungszeit teurem "Trial-and-Error"-Verfahren, bei dem die Modi solange versuchsweise gesetzt werden, bis bei einer Überprüfung keine Zyklen mehr festgestellt werden können.

Schritt 2: Vergabe mit Wahlmöglichkeit

Um festzustellen, ob durch die Festlegung des Modus eines weiteren Parameters am Knoten auf Eingabe ein Zyklus entstehen kann, muß zunächst ein Graph erstellt werden, der alle potentiellen Zyklen enthält, d.h. alle tatsächlichen und alle möglichen Verbindungen zwischen den verschiedenen Vorkommen der Parameter müssen in diesem Graphen eingetragen sein.

Es genügt hierfür die Betrachtung der Blätter des Graphen (siehe auch Satz 3.47).

Dabei dürfen alternative Blätter aus alternativen Teilgraphen, d.h. Teilgraphen, die durch einen OR-Knoten miteinander verbunden sind, nicht gemeinsam betrachtet werden, da sich alternative Zweige nicht gegenseitig beeinflussen können. Das folgende Verfahren muß daher für alle möglichen Kombinationen über Teilgraphen, die nicht alternativ sind, angewendet werden.

Dies geschieht mit Hilfe folgender Kantenverbindungen:

- Eine *sichere, gerichtete* Kante geht von einem Parameter mit Modus out zu allen Vorkommen des Parameters mit Modus in.
- Eine *unsichere, gerichtete* Kante geht von einem Parameter mit Modus out zu allen Vorkommen des Parameters mit Modus unknown.
- Eine *unsichere, gerichtete* Kante geht von einem Parameter mit Modus unknown zu allen Vorkommen des Parameters mit Modus in.
- Eine *unsichere, ungerichtete* Kante geht von einem Parameter mit Modus unknown zu allen Vorkommen des Parameters mit Modus unknown.

Bemerkung: Die Richtung der Kante zeigt von einer (möglichen) Ausgabestelle zu einer (möglichen) Eingabestelle. Ist die Kante ungerichtet, heißt dies, daß beide Richtungen möglich sind. Sicher heißt dabei, daß die Kante bereits festgelegt ist, unsicher, daß an mindestens einem ihrer beiden Enden ein Parameter unbekannten Modus hat und die Kante durch entsprechendes Setzen des Modus entweder sicher werden oder wegfallen kann.

Anhand dieses Graphen lassen sich nun echte und mögliche Zyklen erkennen. Echte Zyklen sind Zyklen, die nur über Parameter mit Modus in und Modus out gehen. Sie dürfen nur dann auftreten, wenn das dem Graphen zugrunde liegende DATALOG$_f^\neg$-Programm einen solchen Zyklus vorgibt (und damit die Ausgangsspezifikation schon fehlerhaft ist). Mögliche Zyklen sind Zyklen, die unsichere Kanten enthalten und sich daher möglicherweise auflösen lassen.

Beispiel:

Figur 17 zeigt ein Beispiel für einen solchen Graphen. Parameter, die sowohl b, als auch f eingetragen haben, sind Parameter mit unbekanntem

Modus. Sichere Kanten sind schwarz eingetragen, unsichere grau. Ungerichtete Kanten haben Pfeile in beide Richtungen.

In Figur 18 und Figur 19 werden jeweils die grau unterlegten Bindungsmuster ausgewählt.

Aufgabe ist nun, die noch unbekannten Modi der Parameter so zu setzen, daß Kanten aus dem Graphen derart entfernt werden, daß keine Zyklen mehr existieren.

Wird ein Bindungsmuster falsch gewählt, z.B. bff bei $fb(U, V, W)$ (Figur 18), ist das Ergebnis ein Graph mit echten Zyklen. Werden die Bindungsmuster richtig gewählt (Figur 19) ist das Ergebnis dagegen ein Graph ohne Zyklen. ◇

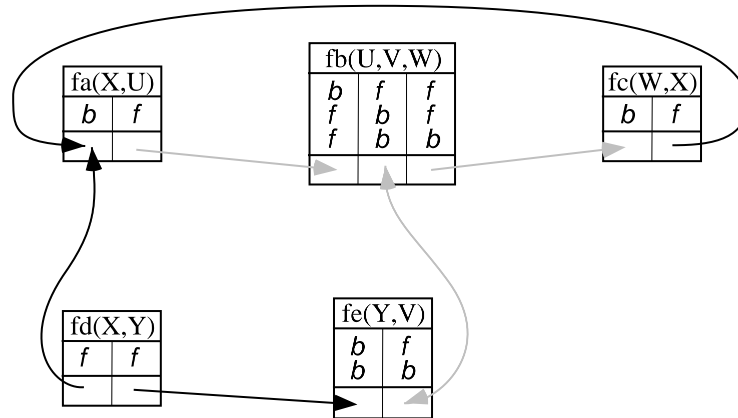


Abbildung 17: Graph für modus-zyklische Datenabhängigkeiten

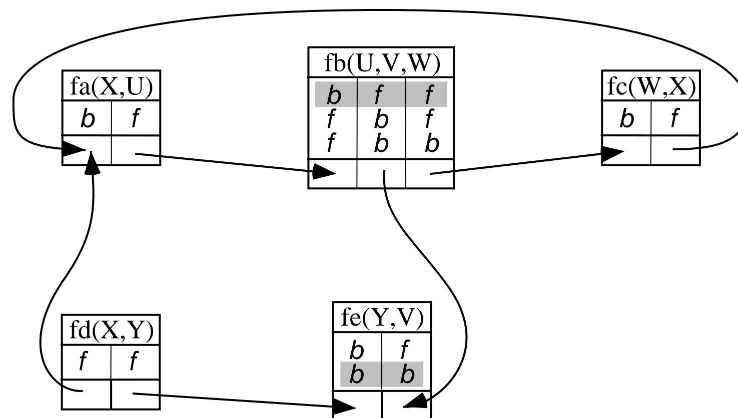


Abbildung 18: Graph mit echten Modus-Zyklen

Kanten können folgendermaßen entfernt werden:

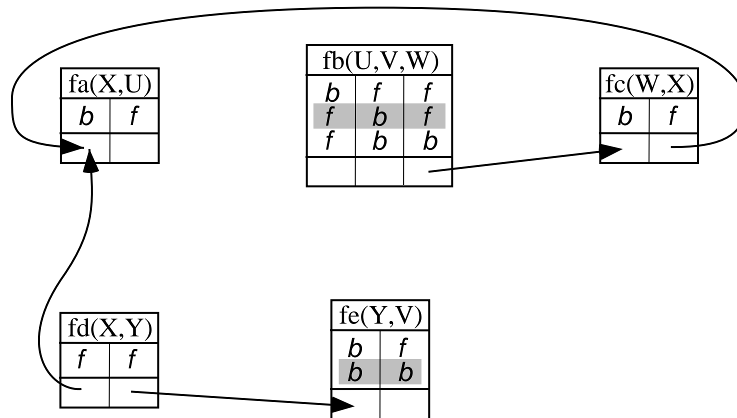


Abbildung 19: Graph ohne Modus-Zyklen

- Eine Kante, die von einem Parameter mit Modus out zu einem Vorkommen des Parameters mit Modus unknown zeigt, entfällt, wenn dieser auf out gesetzt wird.
- Eine Kante, die von einem Parameter mit Modus unknown zu einem Vorkommen des Parameters mit Modus in zeigt, entfällt, wenn der Modus unknown auf in gesetzt wird.

Ungerichtete Kanten können durch Setzen eines der beiden Modi unknown zunächst nur zu gerichteten Kanten werden.

Die Schwierigkeit dabei ist, daß durch das Setzen eines Modus nicht nur eine Kante entfernt werden kann, sondern möglicherweise weitere, unsichere Kanten zu sicheren Kanten und damit auch ein möglicher Zyklus zu einem echten werden könnte.

Folgende Fälle können auftreten:

1. Existiert ein Zyklus, bei dem kein Knoten einen Parameter mit unbekanntem Modus hat, so ist dies ein Fehlerfall.
2. Existiert nur ein Zyklus, dessen Knoten Parameter mit unbekannten Modi haben, kann in diesem Zyklus eine beliebige Stelle ausgesucht werden, an der ein Modus so gesetzt wird, daß eine Kante entfällt, oder eine ungerichtete Kante in Gegenrichtung zum Zyklus ausgerichtet wird.
   - Da es laut Voraussetzung (siehe Schritt 1) zu der aktuellen Stelle mindestens ein anderes Vorkommen mit Modus out gibt, oder eines, das noch auf out gesetzt werden kann, kann der Modus, falls notwendig, in werden.
   - Erzwingt das Setzen des Modus auf out für einen anderen Parameter am Knoten den Modus in, kann kein echter Zyklus entstehen, da kein weiterer Zyklenkandidat existiert.
3. Existieren mehrere Zyklen, die nicht oder nur über sichere Kanten miteinander verbunden sind, können diese als einzelne Zyklen behandelt werden, da das Setzen eines Modus offensichtlich die Modi der anderen Zyklen nicht beeinflussen kann. Ansonsten müßte eine unsichere Kante die Zyklen miteinander verbinden.
4. Existieren mehrere Zyklen die über einen Knoten mit unbekannten Modi verbunden sind, so gilt:
   i) Gibt es in einem der beteiligten Zyklen nur eine Möglichkeit ein Bindungsmuster so zu wählen, daß der Zyklus aufgelöst wird, besteht keine echte Wahlmöglichkeit und das Bindungsmuster muß so festgelegt werden.

ii) Existiert bei allen beteiligten Zyklen mehr als eine Möglichkeit den Zyklus aufzulösen, so ist die Art der Auflösung so zu wählen, daß unsicheren Kanten in anderen Zyklen wegfallen, oder zumindest nicht verändert werden, d.h. nicht zu sicheren Kanten werden. Das bedeutet, daß keine Modi in anderen möglichen Zyklen automatisch mit festgelegt werden.

iii) Ist ii) nicht möglich, so muß die Art der Auflösung derart erfolgen, daß eine ungerichtete Kante im anderen Zyklus gerichtet wird, so daß sie in Gegenrichtung zum Zyklus zeigt.

Ist diese Form der Auflösung nicht möglich, bedeutet dies, daß alle Zyklen sich (symmetrisch) auf exakt die gleiche Art und Weise beeinflussen und es in keinem Zyklus eine zusätzliche Art der Auflösung gibt. Durch jede beliebige Wahl eines Zyklus, der aufgelöst wird, ein anderer Zyklus wird dadurch zu einem echten Zyklus. Dies ist also ein Fehlerfall, der nicht durch eine falsche Wahl entstanden ist, sondern in der Ausgangspezifikation zu suchen ist.

5. Existieren mehrere Zyklen die miteinander über unsichere Kanten verbunden sind, so gilt:

i) Gibt es in einem der beteiligten Zyklen nur eine Möglichkeit ein Bindungsmuster so zu wählen, daß der Zyklus aufgelöst wird, besteht keine echte Wahlmöglichkeit und das Bindungsmuster muß so festgelegt werden.

ii) Existiert bei allen beteiligten Zyklen mehr als eine Möglichkeit den Zyklus aufzulösen, so ist die Art der Auflösung so zu wählen, daß die unsicheren Kanten zu anderen Zyklen entweder wegfallen, oder zumindest nicht zu sicheren Kanten werden. Das bedeutet, daß keine Modi im anderen Zyklus automatisch mit festgelegt werden.

iii) Ist ii) nicht möglich, so muß die Art der Auflösung derart erfolgen, daß bei Zyklen,
- die über eine ungerichtete, unsichere Kante miteinander verbunden sind, die Kante derart gerichtet wird, daß sie vom aktuellen Zyklus wegzeigt,
- die über eine unsichere Kante miteinander verbunden sind, die vom aktuellen Zyklus wegzeigt, zu einer sicheren Kante wird.

In beiden Fällen bedeutet dies, daß der zu setzende Modus des aktuellen Parameters Ausgabe wird. Dadurch werden aber im anderen Zyklus die Modi der anderen Vorkommen des Parameters nicht beeinflußt: da es nun mindestens eine Ausgabestelle gibt, können sie weiterhin Eingabe oder Ausgabe werden.

iv) Sind ii) und iii) nicht möglich, bedeutet dies, daß beide Zyklen für den Parameter an dieser Stelle eine Eingabe haben möchten. Gibt es nun eine andere Ausgabestelle, so können beide Modi auf Eingabe gesetzt werden und dadurch auch beide Zyklen aufgelöst werden. Gibt es keine andere Ausgabestelle, liegt ein Fehlerfall vor.

Schritt 3

Nach jeder Entscheidung in Schritt 2 müssen die verbleibenden unbekannten Modi mit Schritt 1 erneut überprüft werden, damit keine ungültigen Bindungsmuster entstehen können.

Formalisiert lautet der Algorithmus für die Bestimmung der Parametermodi im globalen Zusammenhang (unter Weglassung der Details des Algorithmus zur Vermeidung moduszyklischer Datenabhängigkeiten):

Definition 3.43 *Ein- und Ausgabemodi von Knotenparametern, Fortsetzung*

Es sei $v \in V_{SG}$ in $SG^E_{\mathcal{L}}$ mit $v.name = p(t_1, ..., t_u)$, $u \in \mathbb{N}$ mit Übergabeparametern $\delta_1, ..., \delta_u$ ein Blatt des Graphen, d.h. $v.type$ = FACT, $v.type$ = IFACT, $v.type$ = FUNC. $Modes = \{\mathsf{in}, \mathsf{out}, \mathsf{undef}, \mathsf{unknown}\}$.

Für alle δ_i, $i \in \{1..., u\}$ mit $\delta_i.mode = \mathsf{unknown}$ bestimmt sich $\delta_i.mode$ wie folgt:

1. i) wenn $\nexists w \in V_{SG}$ in $SG^E_{\mathcal{L}}$ mit $w.name = q(l_1, ..., l_m)$, $m \in \mathbb{N}$ und Übergabeparametern $\theta_1, ..., \theta_m$ mit $\theta_j.ident = \delta_i.ident$ für ein $j \in \{1, ..., m\}$, dann ist $\delta_i.mode := \mathsf{out}$.

   ii) wenn $\forall w \in V_{SG}$ in $SG^E_{\mathcal{L}}$ mit $w.name = q(l_1, ..., l_m)$, $m \in \mathbb{N}$ und Übergabeparametern $\theta_1, ..., \theta_m$ mit $\theta_j.ident = \delta_i.ident$ für ein $j \in \{1, ..., m\}$, gilt: v und w haben als untersten gemeinsamen Vorgängerknoten einen OR-Knoten, dann ist $\delta_i.mode := \mathsf{out}$.

   iii) wenn $\forall w \in V_{SG}$ in $SG^E_{\mathcal{L}}$ mit $w.name = q(l_1, ..., l_m)$, $m \in \mathbb{N}$, Übergabeparametern $\theta_1, ..., \theta_m$, $\theta_j.ident = \delta_i.ident$ für ein $j \in \{1, ..., m\}$, und v und w haben als gemeinsamen Vorgängerknoten keinen OR-Knoten gilt: $\theta_j.ident = \mathsf{in}$, dann ist $\delta_i.mode := \mathsf{out}$.

   iv) wenn p ein Systemprädikat ist, das den Aufruf von f darstellt, f definiert ist durch $f(l_1, ..., l_u)$ mit Übergabeparametern $\theta_1, ..., \theta_u$, und Bindungsmustern $(\theta_1.modes_s, ..., \theta_u.modes_s)$ für $s \in \{1, ..., t\}$, $t \in \mathbb{N}$ und $\{(\theta_1.modes_s, ..., \theta_u.modes_s) \mid s \in \{1, ..., t\}, \forall i' \in \{1, ..., u\} : \delta_{i'}.mode \neq \mathsf{unknown} \Rightarrow \theta_i.modes_s = \delta_{i'}.mode\} = \{(\theta_1.modes_{s'}, ..., \theta_u.modes_{s'})\}$ für ein $s' \in \{1, ..., t\}$ (d.h. durch $\{\delta_{i'} \mid i' \in \{1, ..., u\}$ und $\delta_{i'}.mode \neq \mathsf{unknown}\}$ ist ein eindeutiges Bindungsmuster festgelegt), dann ist $\delta_i.mode := \theta_i.mode'_s$.

2. v) wenn keine der Bedingungen i) - iv) erfüllbar sind und $\nexists w \in V_{SG}$ in $SG^E_{\mathcal{L}}$ mit $w.name = q(l_1, ..., l_m)$, $m \in \mathbb{N}$ und Übergabeparametern $\theta_1, ..., \theta_m$ mit $j \in \{1, ..., m\}$, so daß $\theta_j.ident = \delta_i.ident$ und $\theta_j.mode = \mathsf{out}$, dann setze für ein (nach den Konsistenzkriterien für Modi) geeignetes $w' \in V_{SG}$ in $SG^E_{\mathcal{L}}$ mit $w'.name = q(l_1, ..., l_{m'})$, $m' \in \mathbb{N}$ und Übergabeparametern $\theta'_1, ..., \theta'_{m'}$ mit $\theta'_{j'}.ident = \delta_i.ident$ für ein $j' \in \{1, ..., m'\}$: $\theta'_{j'}.mode := \mathsf{out}$,

   vi) wenn keine der Bedingungen i) - v) erfüllbar sind und $w \in V_{SG}$ in $SG^E_{\mathcal{L}}$, mit $w.name = q(l_1, ..., l_m)$, $m \in \mathbb{N}$ und Übergabeparametern $\theta_1, ..., \theta_m$ existiert, so daß gilt $\theta_j.ident = \delta_i.ident$ und $\theta_j.mode = \mathsf{out}$ für ein $j \in \{1, ..., m\}$, dann setze für ein geeignetes $w' \in V_{SG}$ in $SG^E_{\mathcal{L}}$ mit $w'.name = q(l_1, ..., l_{m'})$, $m' \in \mathbb{N}$ und Übergabeparametern $\theta'_1, ..., \theta'_{m'}$ mit $\theta'_{j'}.ident = \delta_i.ident$ für ein $j' \in \{1, ..., m'\}$: $\theta'_{j'}.mode :=$ (nach den Konsistenzkriterien für Modi) geeignet.

3. Wiederhole ab 1. bis keine Parameter mit unbekannten Modi mehr existieren. ◇

Der Algorithmus terminiert, da in jedem Durchlauf für mindestens einen Parameter mit unbekanntem Modus der Modus gesetzt wird.

Da nun an allen Blättern die Modi vollständig bekannt sind, können für die inneren Knoten des Graphen (AND-, NOT- OR- und QUERY-Knoten) zur Vervollständigung der Parametermodi die Regeln zur lokalen Bestimmung (Def. 3.39) erneut angewendet werden.

3.3.4.4 Analyse bei Rekursionen

Bei Rekursionen, d.h. Schleifen im Graphen kann es vorkommen, daß sich ein Parameter unter anderem selbst referenziert. An dieser Stelle ist der Modus folglich nicht mit Hilfe der obigen Regeln bestimmbar, da diese darauf basieren, daß der Modus des Parameters sich entweder lokal bestimmen läßt oder aber an Nachfolgeknoten bereits bestimmt ist.

Zu jeder korrekten Rekursion in $\text{DATALOG}_f^{\neg}$ gibt es einen nicht rekursiven Zweig, in dem die Rekursion terminiert (siehe Abschnitt 3.4.4). Würde dieser Zweig fehlen, gäbe es für die Ausführung der Rekursion nur zwei Möglichkeiten: entweder sie scheitert immer (die Rekursion stoppt, wenn sie im Rekursionszweig vor dem Rekursionsaufruf scheitert), oder sie terminiert nie (weil es keinen erfolgreichen Terminierungszweig gibt).

Es muß also mindestens einen zur Rekursion alternativen Zweig geben. In diesem Zweig muß der Parameter ebenfalls vorkommen, da die Aufrufstelle der Rekursion und des Terminierungszweigs identisch bezüglich der Parameterliste sein müssen.

Hat der Parameter im Terminierungszweig den Modus Ausgabe, so bedeutet dies, daß er in der Schleife ebenfalls "nach oben" weitergegeben wird, er kann also auch dort den Modus Ausgabe haben. Benötigt der Parameter dagegen eine Eingabe im Terminierungszweig, so muß diese Eingabe durch die Schleife "nach unten" gereicht werden. Er muß also dort den Modus Eingabe haben.

Es gilt also für die Knoten, die in einer Schleife sind, und die einen Parameter mit unbekanntem Modus haben:

Definition 3.44 *Ein- und Ausgabemodi von Knotenparametern, Fortsetzung*

Es sei $v \in V_{SG}$ in $SG_{\mathcal{L}}^{E}$ mit $v.name = p(t_1, ..., t_u)$, $u \in \mathbb{N}$ mit Übergabeparametern $\delta_1, ..., \delta_u$ der oberste Knoten eines Zyklus des Graphen,
$Modes = \{\mathsf{in}, \mathsf{out}, \mathsf{undef}, \mathsf{unknown}\}$.

Es gilt aufgrund der Existenz von Terminierungs- und Rekursionszweig: $v.type =$ OR.

Für alle δ_i, $i \in \{1, ..., u\}$ mit $\delta_i.mode = \mathsf{unknown}$ bestimmt sich $\delta_i.mode$ wie folgt:

wenn $\exists w$ mit $e_{v,w} \in E_{SG}$, $w.name = p(l_1, ..., l_u)$ und Übergabeparametern $\theta_1, ..., \theta_u$

i) mit $\theta_i.mode = \mathsf{undef}$,
dann $\delta_i.mode := \mathsf{undef}$,

ii) mit $\theta_i.mode = \mathsf{in}$ und Bedingung i) ist nicht erfüllbar,
dann $\delta_i.mode := \mathsf{in}$,

iii) mit $\theta_i.mode = \mathsf{out}$ und Bedingungen i) und ii) sind nicht erfüllbar,
dann $\delta_i.mode := \mathsf{out}$.

Es gilt auch hier (wie beim OR-Knoten):
wenn $\delta_i.mode = \mathsf{in}$ und $\exists w$ mit $e_{v,w} \in E$, $w.name = p(l_1, ..., l_u)$ und Übergabeparametern $\theta_1, ..., \theta_u$ mit $\theta_i.mode = \mathsf{out}$, dann wird $\theta_i.mode := \mathsf{in}$. ◇

Da nun an einem in der Rekursion enthaltenen Knoten der Modus für alle Parameter vollständig bekannt ist, können auch die Parametermodi für die anderen inneren Knoten, die im Zyklus enthalten sind, zur Vervollständigung anhand der Regeln der lokalen Analyse bestimmt werden.

Sinnvollerweise werden dabei die Knoten des Rekursionszyklus in der Reihenfolge untersucht, in der sie, von unten nach oben gesehen, Vorgängerknoten des OR-Knotens sind, der der oberste Knoten des Zyklus ist.

3.3.4.5 Korrektheit der Analyse

Um die Berechenbarkeit des Graphen zu gewährleisten, ist dafür zu sorgen, daß der Graph während und insbesondere nach der Vergabe der Parametermodi in einem konsistenten Zustand ist.

Satz 3.45

Der Graph ist während der Vergabe der Parametermodi (Abschnitt 3.3.4) in einem konsistenten Zustand (siehe Def. 3.42). ◇

Beweis:

Die Konsistenzbedingungen für die Vergabe der Parametermodi sind (siehe auch Def. 3.42):

a) Es gibt für jeden Parameter mindestens eine Ausgabestelle oder eine Stelle mit noch unbekanntem Modus, die nicht über einen OR-Knoten mit dem aktuellem Vorkommen verbunden ist.

b) Alle bereits bestimmten Modi entsprechen zulässigen Bindungsmustern.

c) Es gibt keine modus-zyklischen Datenabhängigkeiten.

Der Beweis erfolgt mit Induktion über die schrittweise Vergabe der Parametermodi.

Induktionsanfang:

Wenn das zugrunde liegende DATALOG$^{\neg}_{f}$-Programm korrekt ist, so ist der Graph nach Anwendung der Regeln für die lokale Bestimmung der Parametermodi in einem konsistenten Zustand:

a) Ein inkonsistenter Zustand kann dann auftreten, wenn es zu einer Eingabestelle keine entsprechende Ausgabestelle oder eine Verwendungsstelle mit noch unbekanntem Modus gibt.
   - Bei IFACT-Knoten treten nur Ausgabestellen auf.
   - Bei FACT-Knoten haben nur Konstanten den Modus Ausgabe, alle anderen unbekannten Modus.
   - An FUNC-Knoten haben die Parameter entweder unbekannten Modus oder, wenn es nur eine Möglichkeit für eine Bindung gibt, den vorgegebenen Modus.
   - An allen inneren Knoten des Graphen ist der Modus entweder Ausgabe oder wird von den Modi der Parameter der Nachfolgeknoten abgeleitet, d.h. ein Parameter mit Modus Eingabe hat auch an seiner Verwendungsstelle im Nachfolgeknoten den Modus Eingabe; es genügt also die dortige Betrachtung.

Eingabestellen treten bislang folglich nur bei FUNC-Knoten auf. Ein inkonsister Graphen kann nur entstehen, wenn es zu diesen Eingabestellen keine Vorkommen mit Ausgabemodus oder unbekanntem Modus gibt. Das bedeutet aber, daß auch im zugrunde liegenden DATALOG$_f^{\neg}$-Programm keine Ausgabestelle existiert. Es muß also dort schon ein Fehler vorliegen.

b) Da bei der lokalen Vergabe der Modi diese entsprechend der Zulässigkeit vergeben werden, kann ein inkonsistenter Zustand aufgrund eines ungültiges Bindungsmusters nicht auftreten.

c) Ist aufgrund der Vergabe der Modi ein Zyklus von Datenabhängigkeiten entstanden, d.h. es gibt in diesem Zyklus auch keine unbekannten Modi mehr, so ist dieser Zyklus auch in der Ausgangsspezifikation vorhanden, da es bei der lokalen Vergabe keine Wahlmöglichkeit gibt.

Induktionsschritt:

Bei der globalen Vergabe der Parametermodi können folgende Situation auftreten:

Schritt 1:

i) bis iii): In diesen Fällen wird für jeden Parameter, für den es noch keine Ausgabestelle gibt, aber auch keine Auswahl an verschiedenen noch unbekannten Stellen, eine Ausgabestelle erzeugt.

vi): Wird an einem FUNC-Knoten mit alternativen Bindungsmustern aus der Festlegung eines oder mehrerer Modi ein eindeutiges Bindungsmuster ausgewählt und ergibt sich dadurch für einen Parameter mit noch unbestimmten Modus der Modus Eingabe, so führt dies zu keinen Problemen, da es nach Ausführung von i) und ii) zu jedem Parameter mit noch unbestimmten Modus entweder eine Ausgabestelle oder eine weitere Stelle mit unbekanntem Modus geben muß.

Schritt 2: v) und vi)

In diesem Schritt werden die Modi mit Hilfe des Algorithmus zur Vermeidung modus-zyklischer Datenabhängigkeiten vergeben. Der Algorithmus ist per Konstruktion korrekt: es wird grundsätzlich diejenige Stelle ausgewählt, bei der das Setzen eines entsprechenden Modus andere noch unbekannte Modi nicht oder nur so beeinflußt, daß ausschließlich dann Eingabestellen entstehen, wenn es zu diesen Ausgabestellen oder Stellen mit noch unbekanntem Modus gibt.

Schritt 3:

Aufgrund der Tatsache, daß im Schritt 2 nicht alle Parameter betrachtet werden, sondern die Vergabe jeweils nur für eine Stelle durchgeführt wird, wird verhindert, daß die Wahl von Ausgabestellen zunächst für mehr als einen Parameter beliebig getroffen wird. Andernfalls könnte dies leicht zu nicht gültigen Bindungsmustern an FACT- oder FUNC-Knoten führen.

Die Vergabe der Parametermodi bei Rekursionen ändert nur die Modi von Parametern an den inneren Knoten, die an Blättern des Graphen vorkommen. Die vergebenen Modi entsprechen zudem den Modi an den Blättern , d.h. es genügt die dortige Betrachtung. ◇

Der Algorithmus terminiert, wenn kein Parameter mehr einen unbekannten Modus hat. Da in jedem Schritt mindestens ein Modus auf `in` oder `out` gesetzt wird ist dies nach endlich vielen Schritten der Fall.

Folgerung 3.46

Da der Graph zu jedem Zeitpunkt während der Vergabe bei jedem Schritt in einem konsistenten Zustand bleibt, ist er dies auch nach Ende des Verfahrens. ◇

Satz 3.47

Wenn es keinen Modus-Zyklus über den Blättern des Graphen gibt, dann gibt es auch keinen Modus-Zyklus über den Parametern an inneren Knoten des Graphen. ◇

Beweis:

Für die Parameter an inneren Knoten gilt, daß sie sich aufteilen in:

i) Parameter, die korrespondierende Parameter an den Blättern des Graphen haben.

   Für sie gilt, daß sich ihr Modus von mindestens einem Modus eines korrespondierenden Parameters an einem Blatt ableitet. Außerdem gilt: wenn an den Blättern des Graphen kein Modus-Zyklus existiert, wird er auch nicht von einer Teilmenge dieser Parameter erzeugt, da der Algorithmus zur Vermeidung von Modus-Zyklen alle möglichen Verbindungen berücksichtigt. Damit kann auch von den korrespondierenden Parametern an inneren Knoten kein Modus-Zyklus erzeugt werden.

ii) Parameter, die keine korrespondierende Parameter an den Blättern des Graphen haben.

   Es gibt folgende Fälle in denen sich der Modus eines Parameters eines inneren Knotens nicht vom Modus eines korrespondierenden Parameters an einem Blatt ableitet:

   - Konstanten an AND-Knoten haben generell den Modus Ausgabe, es sei denn, der AND-Knoten ist Nachfolgeknoten eines OR-Knotens, der den Modus explizit auf Eingabe gesetzt ist. Dies ist aber nur dann möglich, wenn ein anderer Nachfolgeknoten des OR-Knotens, bzw. ein Blatt im von diesem Nachfolgeknoten aufgespannten Teilgraphen eine Eingabe für einen korrespondierenden Parameter benötigt. Damit leitet sich in diesem Fall der Modus des Parameter wiederum vom Modus eines Parameters an einem Blatt ab. Für den Modus Ausgabe gibt es keine Wahlmöglichkeit, daher kann ein Modus-Zyklus nur entstehen, wenn die Ausgangsspezifikation fehlerhaft war.
   - Variablen, die an einem AND-Knoten aber nicht an einem Nachfolgeknoten vorkommen, haben den Modus Eingabe. Für ihren Modus gibt es jedoch ebenfalls keine Wahlmöglichkeit. Nach den Konsistenzbedingungen für Programme (Def. 3.1) erzeugen sie daher keinen Modus-Zyklus.
   - Alle Parameter von NOT-Knoten haben generell den Modus Eingabe, unabhängig vom Modus korrespondierender Parameter am (einzigen) Nachfolgeknoten. Ein Knoten, der nur Eingaben hat, kann nicht in einem Modus-Zyklus enthalten sein. ◇

Satz 3.48

Die Datenabhängigkeiten in einer Schleife des Strukturgraphen sind nicht modus-zyklisch. ◇

Beweis:

Der Modus der Parameter desjenigen Knotens, an dem die rückführende Kante beginnt, leitet sich per Definition von den Modi der Parameter des Knotens, an dem die

Kante endet, ab (siehe Def. 3.44). Zwei Knoten mit denselben Modi für ihre Parameter können jedoch nicht ursächlich für einen Modus-Zyklus sein, wenn dies nicht schon einer der beiden Knoten ist. Damit sind die Datenabhängigkeiten in einer Schleife des Struktugraphen (die beide Knoten enthalten muß) nicht modus-zyklisch. ◇

3.4 Der Datenflußgraph

Nach der Vergabe eindeutiger Identifikatoren für Parameter an Knoten und der Analyse der Modi der Parameter kann nun aus den Knoten des Strukturgraphen ein neuer Graph, der Datenflußgraph erzeugt werden.

Definition 3.49 *Datenflußgraph*

Ein Datenflußgraph $\mathcal{D}$ sei definiert durch $\mathcal{D} = (V_\mathcal{D}, E_\mathcal{D}, Fun_\mathcal{D})$, wobei $(V_\mathcal{D}, E_\mathcal{D})$ ein gerichteter Graph ist und

i) $V_\mathcal{D}$ ist eine endliche, nicht leere Menge von Knoten.

Für die Knoten $v \in V_\mathcal{D}$ gelte:
- $v.in$ mit $v.in \geq 0$ bezeichne die Anzahl der Eingabeparameter von v. Knoten mit $v.in = 0$ heißen Eingabe- oder Startknoten (bezüglich des Graphen).
- $v.out$ mit $v.out \geq 0$ bezeichne die Anzahl der Ausgabeparameter von v. Knoten mit $v.out = 0$ heißen Ausgabeknoten (bezüglich des Graphen).
- $v.const$ mit $v.const \geq 0$ bezeichne die Anzahl der Konstanten von v.
- $v.ip_1, ..., v.ip_{v.in}$, $v.in > 0$, seien die Eingabeparameter des Knotens.
- $v.op_1, ..., v.op_{v.out}$, $v.out > 0$, seien die Ausgabeparameter des Knotens.
- $v.cp_1, ..., v.cp_{v.const}$, $v.const > 0$, seien die Konstanten des Knotens.
- $v.IP := \{v.ip_1, ..., v.ip_{v.in}\}$ bezeichne die Menge der Eingabeparameter.
- $v.OP := \{v.op_1, ..., v.op_{v.out}\}$ bezeichne die Menge der Ausgabeparameter.
- $v.CP := \{v.cp_1, ..., v.cp_{v.const}\}$ bezeichne die Menge der Konstanten.
- $v.ip_i.in$ bezeichne die Anzahl der Eingabekanten für $v.ip_i$ für $i = 1, ..., v.in$.
- $v.op_i.out$ bezeichne die Anzahl der Ausgabekanten für $v.op_i$ für $i = 1, ..., v.out$.

ii) $E_\mathcal{D}$ ist eine endliche, nicht leere Menge von gerichteten Kanten.

Für die Kanten $e \in E_\mathcal{D}$ gelte:
- $e_{v.ip_i,j}$ bezeichne die Eingangskante j von Knoten $v \in V_\mathcal{D}$ am Parameter $v.ip_i$ für $i = 1, ..., v.in$, $j \in \{1, ..., v.ip_i.in\}$.
- $e_{v.op_i,j}$ bezeichne eine Ausgangskante j von Knoten $v \in V_\mathcal{D}$ am Parameter $v.op_i$ für $i = 1, ..., v.out$, $j \in \{1, ..., v.op_i.out\}$.
- $e_{v.op_i,w.ip_j}$ bezeichne eine Kante von Knoten $v \in V_\mathcal{D}$ am Parameter $v.op_i$ für $i = 1, ..., v.out$ zum Knoten $v \in V_\mathcal{D}$ am Parameter $w.ip_j$ für $j = 1, ..., w.in$.

Damit ist
$v.vorg = |\{w \mid w \in V_\mathcal{D}, \exists i \in \{1, ..., v.in\}, \exists i' \in \{1, ..., w.out\}, e_{w.op_{i'},v.ip_i} \in E_\mathcal{D}\}|$
$v.nach = |\{w \mid w \in V_\mathcal{D}, \exists i \in \{1, ..., v.out\}, \exists i' \in \{1, ..., w.in\}, e_{v.op_{i'},w.ip_i} \in E_\mathcal{D}\}|$

iii) $Fun_\mathcal{D}$ ist eine endliche, nicht leere Menge von Knotenfunktionen für alle $v \in V_\mathcal{D}$.

◇

Ein Datenflußgraph muß zusätzlich folgenden Randbedingungen genügen:

Definition 3.50 *Randbedingungen eines Datenflußgraphen*

i) Für die Menge $V_\mathcal{D}$ der Knoten muß gelten:
- $EK \subset V_\mathcal{D}$, mit $EK := \{v \mid v.in = 0\}$; $|EK| \geq 1$.
- $AK \subset V_\mathcal{D}$, mit $AK := \{v \mid v.out = 0\}$; $|AK| \geq 1$, (insbesondere aber auch $|AK| = 1$).
- $EK \cap AK = \emptyset$.

ii) Für alle Knoten $v \in V_{\mathcal{D}}$ muß gelten:
- $v.ip_i.in > 0$ für $i = 1, ..., v.in$, wenn $v.in > 0$.
- $v.op_i.out > 0$ für $i = 1, ..., v.out$, wenn $v.out > 0$.

iii) Für die Menge der Kanten $E_{\mathcal{D}}$ muß außerdem gelten:
$e_{v.op_i} = e_{w.ip_i} \Rightarrow v \neq w, \; \forall v, w \in V_{\mathcal{D}}$,
d.h. es geht keine Kante direkt von einem Ausgang zu einem Eingang desselben Knotens. ◇

3.4.1 Konstruktionsprinizipien

nDie Bezeichnungen Vorgängerknoten und Nachfolgeknoten sollen sich, falls nicht anders angegeben, auch im weiteren auf die Ordnung der Knoten im Strukturgraphen und nicht auf die Ordnung im Datenflußgraphen beziehen.

Ein Wertetupel sei im folgenden definiert als eine Menge von bezüglich einer Knotenoperation zusammengefaßter Werte.

Es gibt folgende alternative Prinzipien, um den Datenflußgraphen zu konstruieren:

1. *Erzeugung der Kanten*

   Die Konstruktion des Datenflußgraphen erfordert die Verbindung der Knoten des Strukturgraphen in einer Art und Weise, so daß der resultierende Graph gemäß des Datenflußprinzips abgearbeitet werden kann, d.h. die Knoten müssen so miteinander verbunden werden, daß Datenabhängigkeiten durch Kanten repräsentiert werden.

   A) *Kanten zwischen den Knoten*

   Diese Methode ist unrealisierbar, da folgende Probleme auftreten:

   i) Jeder Knoten, der einen Wert eines Wertetupels als Eingabe benötigt, muß bei paralleler Ausführung eine eigene Kopie des Wertetupels bekommen. Zusätzlich muß der Zusammenhang zwischen den Ergebnissen und den Eingaben aus denen sie entstanden sind, erhalten bleiben, um die Korrektheit der Berechnungen zu gewährleisten (siehe Kapitel 4.1.1.3). Ergebnisse können daher nur zusammen mit den Eingaben weitergegeben werden. Dies führt zu einem unverhältnismäßig großen Anwachsen der zu transportierenden Datenmengen.

   ii) Wenn ein Wertetupel an jeweils nur einen Knoten weitergegeben wird, erhält man eine unnötige Sequentialisierung der Graphausführung. Darüber hinaus müssen die Ergebnisse zum Wertetupel hinzugefügt werden, da die Werte des Wertetupels an nachfolgenden Knoten eventuell noch benötigt werden. Um zu große Wertetupel zu vermeiden, wäre zusätzlich eine Analyse notwendig, die besagt, wann Werte nicht mehr benötigt werden und deshalb aus dem Wertetupel entfernt werden können.

   B) *Kanten zwischen den Parametern der Knoten*

   Da ein Knoten mehrere Ausgabeparameter haben kann, von denen zusätzlich Werte in unterschiedlicher Weise an Eingabeparameter verschiedener Knoten gehen können, können die Parameter der Knoten einzeln durch Kanten miteinander verbunden werden. Es kann vorkommen, daß es zwischen zwei Knoten mehrere, parallele Kanten gibt.

Die Zusammengehörigkeit von Werten, die als ein Wertetupel ausgegeben werden, darf dabei jedoch nicht verloren gehen, wenn diese an verschiedene Knoten weitergegeben werden, da diese Zusammengehörigkeit für die Korrektheit der Berechnungen notwendig ist. Gehört ein Wert zu einem Tupel und wird er durch eine Knotenoperation ungültig, so werden damit auch alle anderen Werte des Tupels ungültig. Darüber hinaus müssen auch Werte gelöscht werden, die bereits durch Knotenoperationen über Eingaben von Werten dieses Tupels entstanden sind. Man benötigt also eine eindeutige Kennzeichnung von Werten als zueinander gehörig. Zusätzlich muß in die Kennzeichnung der durch eine Knotenoperation auf Eingaben erzeugten Werte die Kennzeichnung der Eingaben miteingehen. Ein solches Verfahren wird in Kapitel 4.1.1.3 vorgestellt.

2. *Struktur des Graphen*

   Das Ziel des in dieser Arbeit vorgestellten Modells ist ein Graph, der bottom-up ausgewertet werden kann, wobei jedoch alle Literale, die Eingaben benötigen, wie Fakten, Systemprädikate und Negationen, diese auch erhalten sollen.

   Es gibt verschiedene Möglichkeiten, um mit Hilfe der Information, welche Parameter Erzeugungs- und welche Verwendungsstellen für Werte sind, den Graphen zu konstruieren:

   A) *Kanten direkt von jeder Erzeugungsstelle zur jeder Verwendungsstelle*

      Ist der Graph so aufgebaut, daß alle Erzeugungsstellen direkt mit allen Verwendungsstellen verbunden sind, bleibt der Graph soweit wie möglich parallel abarbeitbar. Der Nachteil ist jedoch, daß die Kennzeichnung der Tupel-Zusammengehörigkeit für Werte sehr groß sein muß, da sie über die gesamte Ausführungszeit des Graphen erhalten bleiben muß. Außerdem werden vielfach überflüssige Knotenoperationen durchgeführt, wenn Eingaben verwendet werden, die nachträglich gelöscht werden.

   B) *Kanten von der obersten Ausgabestelle zu allen Verwendungsstellen*

      Werden alle möglichen Joins über Wertetupel ausgeführt, bevor ein Wert aus einem Tupel an einem anderen Knoten als Eingabe verwendet wird, benötigt man weder eine Kennzeichnung, noch werden überflüssige Knotenoperationen ausgeführt. (Eine Definition des Joins ist in Anhang A.1 nachzulesen.)

      Diese Methode ist jedoch nicht für jedes Programm realisierbar: im stark vereinfachten Beispiel

      $$f(X^b) \leftarrow g(X^b, Y^b), h(Y^f, Z^b), k(Z^f, X^b)$$

      kann der Join über X aus $g(X,Y)$ und X aus $k(Z,X)$ erst berechnet werden, wenn die Berechnung von $h(Y,Z)$ zuvor erfolgt ist. D.h. der Wert von Y wird benötigt, bevor der entscheidende Join für X ausgeführt werden kann.

   C) *Zwischenform*

      Aus der Problematik der ersten Methode und der Unrealisierbarkeit der zweiten empfiehlt sich eine sinnvolle Kombination beider Methoden. Die Ausgaben gehen soweit wie möglich im Graphen nach oben, bevor sie als Eingaben verwendet werden. Der Vorteil ist, daß bis auf wenige Sonderfälle alle notwendigen Joins für die Ausgabe bereits durchgeführt wurden.

Für die Definition einer geeigneten obersten Stelle zur Realisierung der Zwischenform (3.) benötigt man zunächst die Definition disjunkter, benachbarter Teilgraphen.

Definition 3.51 *Benachbarte, disjunkte Teilgraphen*

- Zwei Teilgraphen sollen disjunkt heißen, wenn sie keine gemeinsamen Knoten haben.
- Zwei Teilgraphen sollen benachbart heißen, wenn die Wurzel jedes Teilgraphen den gleichen Vorgängerknoten besitzt.

Da der Strukturgraph außer im Fall von Zyklen keine Mehrfachreferenzierung von Knoten mehr enthält, gilt allgemein, daß benachbarte Teilgraphen disjunkt sind. (Ein Zyklus lßt sich aufgrund seiner eindeutigen Wurzel nicht in verschiedene Teilgraphen aufteilen). ◇

3.4.2 Zyklenfreie Datenflußgraphen

Die folgende Definition eines zyklenfreien Datenflußgraphen basiert auf einem Strukturgraphen, bei dem die rückführenden Kanten (und damit Rekursionen) vorerst nicht berücksichtigt werden.

Definition 3.52 *Konstruktion eines zyklenfreien Datenflußgraphen*

Ein zyklenfreier Datenflußgraph $DG_{\mathcal{L}} = (V_{DG}, E_{DG}, Fun_{DG})$ entsteht aus einem erweiterten Strukturgraphen $SG^{E}_{\mathcal{L}} = (V_{SG}, E_{SG})$ (mit Bezeichnungen für Parameter aus Def. 3.20 und $u, u' \in \mathbb{N}$, $u \leq u'$ und $E^v, E^m \subset E_{SG}$ definiert wie in Definition 3.35):

1. Für V_{DG} gilt:

   a) wenn $v \in V_{SG}$ mit $v.type$ = IFACT, Parametern δ_1, ..., δ_u und Ergebnisparameter δ_{u+1}, dann ist $v \in V_{DG}$ mit
   - $v.in = 0$
   - Ausgabeparametern $v.OP = \{\delta_i \mid i \in \{1, ..., u\}\} \cup \{\delta_{u+1}\}$
   - $v.const = 0$

   b) wenn $v \in V_{SG}$ mit $v.type$ = FACT, Parametern δ_1, ..., δ_u und Ergebnisparameter δ_{u+1}, dann ist $v \in V_{DG}$ mit
   - Eingabeparametern $v.IP = \{\delta_i \mid i \in \{1, ..., u\}, \delta_i.mode = \mathsf{in}\} \cup \{\delta_{u+1}\}$
   - Ausgabeparametern $v.OP = \{\delta_i \mid i \in \{1, ..., u\}, \delta_i.mode = \mathsf{out}\} \cup \{\delta_{u+1}\}$
   - Konstanten $v.CP = \{\delta_i \mid i \in \{1, ..., u\}, \delta_i.name = a,\ a \text{ Konstante}\}$

   c) wenn $v \in V_{SG}$ mit $v.type$ = FUNC, Parametern δ_1, ..., δ_u und Ergebnisparameter δ_{u+1}, dann ist $v \in V_{DG}$ mit
   - Eingabeparametern $v.IP = \{\delta_i \mid i \in \{1, ..., u\}, \delta_i.name = X,\ X \text{ Variable}, \delta_i.mode = \mathsf{in}\}$
   - Ausgabeparametern $v.OP = \{\delta_i \mid i \in \{1, ..., u\}, \delta_i.mode = \mathsf{out}\} \cup \{\delta_{u+1}\}$
   - Konstanten $v.CP = \{\delta_i \mid i \in \{1, ..., u\}, \delta_i.name = a,\ a \text{ Konstante}\}$

   d) wenn $v \in V_{SG}$ mit $v.type$ = AND, Parametern δ_1, ..., δ_u, lokalen Parametern δ_{u+1}, ..., $\delta_{u'}$ und Ergebnisparameter δ_{u+1}, dann ist $v \in V_{DG}$ mit
   - Eingabeparametern $v.IP = \{\delta_i \mid i \in \{1, ..., u\}, \delta_i.mode = \mathsf{in}\} \cup \{\delta_i \mid i \in \{1, ..., u'\}, \delta_i.name = X,\ X \text{ Variable}, \delta_i.mode = \mathsf{out}\} \cup \{\delta_{u'+1}\}$
   - Ausgabeparametern $v.OP = \{\delta_i \mid i \in \{1, ..., u\}, \delta_i.mode = \mathsf{out}\} \cup \{\delta_{u'+1}\}$
   - Konstanten $v.CP = \{\delta_i \mid i \in \{1, ..., u\}, \delta_i.name = a,\ a \text{ Konstante}\}$

   e) wenn $v \in V_{SG}$ mit $v.type$ = QUERY, Parametern δ_1, ..., δ_u, lokalen Parametern δ_{u+1}, ..., $\delta_{u'}$ und Ergebnisparameter δ_{u+1}, dann ist $v \in V_{DG}$ mit
   - Eingabeparametern $v.IP = \{\delta_i \mid i \in \{1, ..., u\}, \delta_i.mode = \mathsf{in}\} \cup \{\delta_i \mid i \in \{1, ..., u'\}, \delta_i.name = X,\ X \text{ Variable}, \delta_i.mode = \mathsf{out}\} \cup \{\delta_{u'+1}\}$

- $v.out = 0$
- $v.const = 0$

f) wenn $v \in V_{SG}$ mit $v.type$ = OR, Parametern δ_1, ..., δ_u und Ergebnisparameter δ_{u+1}, dann ist $v \in V_{DG}$ mit
- Eingabeparametern $v.IP = \{\delta_i \mid i \in \{1, ..., u\}, \delta_i.mode = \mathsf{in}\} \cup \{\delta_i \mid i \in \{1, ..., u\}, \delta_i.mode = \mathsf{out}\} \cup \{\delta_{u+1}\}$
- Ausgabeparametern $v.OP = \{\delta_i \mid i \in \{1, ..., u\}, \delta_i.mode = \mathsf{out}\} \cup \{\delta_{u+1}\}$
- $v.const = 0$

g) wenn $v \in V_{SG}$ mit $v.type$ = NOT, Parametern δ_1, ..., δ_u und Ergebnisparameter δ_{u+1}, dann ist $v \in V_{DG}$ mit
- Eingabeparametern $v.IP = \{\delta_i \mid i \in \{1, ..., u\}, \delta_i.name = X$, X Variable, $\delta_i.mode = \mathsf{in}\} \cup \{\delta_{u+1}\}$
- Ausgabeparameter $\{\delta_{u+1}\}$
- $v.const = 0$

2. Für E_{DG} gilt:

i) $e_{w.op_j, v.ip_i} \in E_{DG}$
wenn $\exists v, w \in V_{DG}$ mit $e_{v,w} \in E^v \cup E^m$ und
$\exists i, j \in \mathbb{N}$, $1 \leq i \leq v.in$, $1 \leq j \leq w.out$,
mit $v.ip_i.ident = w.op_j.ident$.

ii) $e_{v'.op_{i'}, w.ip_j} \in E_{DG}$
wenn $\exists v, w \in V_{DG}$ mit $e_{v,w} \in E^v \cup E^m$ und
$\exists i, j \in \mathbb{N}$, $1 \leq i \leq v.in$, $1 \leq j \leq w.in$, mit $v.ip_i.ident = w.ip_j.ident$.
und $\exists v' \in V_{SG}$, $i' \in \mathbb{N}, 1 \leq i' \leq v'.out$ mit $e_{v,v'} \notin E^v \cup E^m$ und $e_{v'.op_{i'}, v.ip_i} \in E_{DG}$

iii) $e_{w'.op_{j'}, w.ip_j} \in E_{DG}$
wenn $\exists v, w \in V_{DG}$ mit $e_{v,w} \in E^v \cup E^m$ mit $v.type$ = AND,
$\exists i, j \in \mathbb{N}$, $1 \leq i \leq v.in$, $1 \leq j \leq w.in$
mit $v.ip_i.ident = w.ip_j.ident$
und $\exists w' \in V_{SG}$, $j' \in \mathbb{N}, 1 \leq j' \leq w'.out$ mit $v.ip_i.ident = w'.op'_j.ident$.

3. Fun_{DG} sei eine Funktion, die den Knoten $v \in V_{DG}$ eine Knotenfunktionalität zuordnet (Fun_{DG} wird in Abschnitt 3.6 näher spezifiziert). $\diamond$

Für eine algorithmische Formulierung der obigen formalen Definition sei auf Abschnitt 3.4.4.3 verwiesen.

Für jeden Knoten des Strukturgraphen gibt es einen Knoten im Datenflußgraphen. Es gibt nun jedoch drei Kathegorien von Parametern: Eingabeparameter, Ausgabeparameter und Konstanten.

Bemerkung: Es gilt nicht: $v.CP \cup v.IP = \emptyset$, $v.CP \cup v.OP = \emptyset$. Ausgabeparameter haben lediglich symbolischen Charakter, in der späteren maschinennahen Implementierung wird für sie, im Unterschied zu Eingabeparametern, kein Speicherplatz reserviert. Daher muß Konstanten, die den Modus Ausgabe haben, über die Definition der Menge der Konstanten, zusätzlich explizit (Speicher-)Platz im Knoten zugewiesen werden. Eine Konstante mit Modus Eingabe benötigt ebenfalls einen eigenen Platz, da der Vergleich ankommender Werte mit einer Konstanten erst durch die Knotenoperation möglich ist und daher zusätzlich Platz für Eingaben vorhanden sein muß. Ausnahme ist der FUNC-Knoten. Hier erfolgt auf eine Konstante mit Modus Eingabe keine Eingabe, da der Modus Eingabe die Eingabe der Konstante an das Systemprädikat bedeutet. Daher

kann sie nicht in die Menge der Eingabeparameter aufgenommen werden. Auch hier ist jedoch die Deklaration eines expliziten Speicherplatzes für Konstanten notwendig.

Eingabeparameter dienen der Aufnahme von Werten für den Knoten, Ausgabeparameter der Weitergabe von Ergebnissen der Knotenoperation an andere Knoten. Da der Modus eines Parameters eines inneren Knotens, z.B. eines AND-Knotens, im Strukturgraphen nicht für eine Aktion (ein Wert wird erzeugt und ausgegeben), sondern für eine Richtungsangabe der Weitergabe von Werten steht, wird in diesen Fällen sowohl ein Eingabe- als auch ein Ausgabeparameter erzeugt. Der Ergebnisparameter übernimmt die Verarbeitung und Weitergabe des boolschen Funktionswertes für den Ergebnisstatus von Berechnungen: erfolgreich oder nicht erfolgreich.

Die Verbindung der Knoten über Kanten geschieht nach folgendem Schema (die Bezeichnung Vorgänger- und Nachfolgeknoten beziehen sich dabei immer noch auf die Anordnung der Knoten im Strukturgraphen):

- Kanten für Ausgaben:
  i) Die Ausgabeparameter eines Nachfolgeknotens werden mit den korrespondierenden Eingabeparametern (mit gleichem *ident*) des betrachteten Knotens verbunden. Dadurch werden alle Ausgaben von Knoten im Graphen nach oben an Vorgängerknoten weitergegeben.

- Kanten für Eingaben:
  i) Ist der Parameter eines Knotens, der kein Nachfolgeknoten des betrachteten Knotens ist, mit einem korrespondierenden Eingabeparameter am betrachteten Knoten verbunden, wird er auch mit dem korrespondierenden Eingabeparameter an dessem Nachfolgeknoten verbunden.
  ii) Der Ausgabeparameter eines Nachfolgeknotens eines AND-Knotens wird mit dem korrespondierenden Eingabeparameter eines anderen Nachfolgeknotens verbunden.

  Durch eine Kombination der Fälle wird erreicht, daß Ausgaben soweit wie möglich nach oben gehen (i), dann unter einem AND-Knoten von der Wurzel eines Teilgraphen an einen benachbarten, disjunkten Teilgraphen (ii) und in diesem direkt an die Eingabestelle (iii) gehen.

Bemerkung: Ist der Vorgängerknoten der Teilgraphen ein OR-Knoten, ist eine Eingabe von einem Teilgraphen an den anderen nicht zulässig, da die Ergebnisse von alternativen Zweigen im Graphen voneinander unabhängig sind und somit das Ergebnis eines alternativen Zweiges nicht als Eingabe für einen anderen alternativen Zweig verwendet werden kann.

Die Kantenverbindung dieses Modells ist ähnlich zu einem Berechnungsweg in (sequentiellen) Top-Down Verfahren, bei denen die Berechnungen mit einem Tiefensuchverfahren über den vom Programm aufgespannten Berechnungsbaum angestoßen und weitergegeben werden. Auch hier wandern die an einem Fakt oder von einem Systemprädikat erzeugten Werte bis zu demjenigen Rumpfliteral nach oben, das in einer Regel steht, in der ein weiteres Rumpfliteral diese Werte als Eingabe benötigt.

Der wesentliche Unterschied ist jedoch, daß Eingaben im hier vorgestellten Modell nicht über Regelköpfe an ihre Verwendungsstellen gehen, sondern direkt. Dadurch läßt sich ein Graph aufspannen, bei dem Daten nur noch in eine Richtung gehen.

Knoten, die keine Eingaben bekommen, also z.B. Fakten, sind nun an "unterster" Stelle im Graphen. Knoten, die Eingaben von anderen Knoten erwarten, stehen in der Abarbeitungsreihenfolge nach diesen Knoten, also auf einer "höheren" Ebene im Graphen.

Der “oberste” Knoten, die Wurzel des Graphen, ist die Ausgabestelle des Gesamtergebnisses. Dies ist der QUERY-Knoten, der seinerseits keine Ausgaben mehr an andere Knoten weitergibt. Der Fluß der Daten kann dann in genau einer Richtung gehen, von “unten”, der Eingabe der Werte aus dem Datenspeicher, über Knoten mit Berechnungen, nach “oben” zur Ausgabe. Das Ergebnis ist also ein Datenflußgraph im Sinne der Datenflußrechnerarchitekturen.

Satz 3.53

$DG_{\mathcal{L}}$ ist ein Datenflußgraph entsprechend Definition 3.49. ◇

Beweis:

$DG_{\mathcal{L}}$ ist ein Graph mit einer endlichen, nicht leeren Knotenmenge und mit einer endlichen, nicht leeren Menge gerichteter Kanten. Daher sind lediglich die geforderten Randbedingungen zu überprüfen.

i) zu zeigen: $EK \cup AK = \emptyset$
$EK = \{v \mid v.type = \text{IFACT}\} \cup \{v \mid v.type = \text{FACT},\ v.in = 0\} \cup$
$\{v \mid v.type = \text{FUNC},\ v.in = 0\}$ und
$AK = \{v \mid v.type = \text{QUERY}\}$ (nach Def. 3.52, 1);
es gilt daher also offensichtlich: $EK \cap AK = \emptyset$.

ii) zu zeigen: für alle Knoten $v \in V_{\mathcal{D}}$ gilt:
a) $v.op_i.out > 0$ für $i = 1, ..., v.out$, $v.out > 0$.
b) $v.ip_i.in > 0$ für $i = 1, ..., v.in$, $v.in > 0$.

zu a) Jeder Ausgabeparameter wird mindestens einmal betrachtet (d.h. von ihm ausgehend wird eine Kante erzeugt):

Zu jedem Ausgabeparameter eines Nachfolgeknotens gibt es einen korrespondierenden Parameter am Knoten:

· ist der Knoten ein AND- oder QUERY-Knoten, und gibt es keinen normalen korrespondierenden Parameter, so gibt es einen lokalen korrespondierenden Parameter (Def 3.20 ii);

· ist der Knoten ein OR- oder QUERY-Knoten, so sind die Parameterlisten und damit die Identifikatoren identisch.

In die Liste der Eingabeparameter der inneren Knoten (AND-, OR-, NOT- und QUERY-Knoten) werden alle Parameter aufgenommen. Es gibt also zu jedem Ausgabeparameter eines Knotens, der Nachfolgeknoten eines inneren Knotens ist, einen korrespondierenden Eingabeparameter an seinem Vorgängerknoten. Folglich wird für jeden Ausgabeparameter mindestens eine Kante erzeugt (Def. 3.52 2. i). Der QUERY-Knoten hat keine Ausgabeparameter.

zu b) Jeder Eingabeparameter wird mindestens einmal betrachtet (d.h. zu ihm hinführend wird eine Kante erzeugt):

Zu jedem Eingabeparameter eines Knotens gibt es laut Konsistenzbedingung für die Vergabe der Parametermodi (Def. 3.42) einen korrespondierenden Parameter, der eine Ausgabe erzeugt.

· Gibt es zu einem Eingabeparameter an einem Knoten einen korrespondierenden Ausgabeparameter an einem Nachfolgeknoten, so wird nach 3.52 2. i eine Kante erzeugt.

· Gibt es zu einem Eingabeparameter an einem Nachfolgeknoten eines Knotens einen korrespondierenden Ausgabeparameter an einem Nachfolgeknoten, so wird nach 3.52 2. ii eine Kante erzeugt.

· Gibt es zu einem Eingabeparameter an einem Nachfolgeknoten eines Knotens einen korrespondierenden Eingabeparameter am Knoten, so wird nach 3.52 2. iii eine Kante erzeugt.

iii) zu zeigen: $e_{v.op_i} = e_{w.ip_j}$ für ein $i, j \in \mathbb{N} \Rightarrow v \neq w \; \forall v, w \in V_{DG}$
Es gilt: $E = E^v \cup E^m \cup E^r$ (Def. 3.11). Für $e_{v,v} \in E$ gilt: $e_{v,v} \in E^r$ (Def. 3.11, Teil 3) und damit, nach Satz 3.19: $e_{v,v} \notin E^v \cup E^m$. Somit folgt für alle $e_{v,w} \in E^v \cup E^m$: $v \neq w$.
Damit gilt $\forall v, w \in V_{DG}$ in $DG_{\mathcal{L}}$: wenn $e_{v.op_i,w.ip_j} \in E_{DG}$ für $i, j \in \mathbb{N}$, dann ist $v \neq w$, da nur Knoten v und w mit $e_{v,w} \in E^v \cup E^m$ betrachtet werden. ⋄

Im folgenden sollen einzelne Situationen, die an OR- und AND-Knoten vorkommen können, veranschaulicht werden. Es genügt dabei die Betrachtung dieser beiden Knotentypen ohne auf Vollständigkeit zu verzichten, da der Algorithmus die Kanten zwischen Knoten und Nachfolgeknoten behandelt. Nachfolgeknoten haben aber nur OR-, NOT-, AND- und QUERY-Knoten. Der QUERY-Knoten ist identisch zum AND-Knoten. Der NOT-Knoten kann hinsichtlich der Kantenverbindung als Sonderfall des AND-Knotens angesehen werden und wird im nächsten Abschnitt explizit behandelt.

Es werden folgende Vereinbarungen getroffen:
- Die grauen und gestrichelten Kanten zeigen die Graphstruktur. Gestrichelte Kanten bedeuten dabei, daß der Weg über mehrere Zwischenknoten führen kann.
- Die schwarzen Pfeile in den Graphen gelten jeweils nur für einen speziellen Parameter, d.h. sie bezeichnen nur für die Belegungen dieses einen Parameters den Transportweg.
- f bezeichnet den Knoten für die Regel f. Darüber hinaus sagt z.B. OR f aus, daß es sich um einen OR-Knoten handelt, AND f, daß es sich um einen AND-Knoten handelt.
- Ein "i" im Knoten bedeutet, daß der behandelte Parameter an diesem Knoten vom Typ Eingabe ist. Ein "o" bedeutet, daß er vom Typ Ausgabe ist.
- Die Bezeichnung Nachfolgeknoten bezieht sich auf den Strukturgraphen: ein Knoten $w \in V_{DG}$ ist Nachfolgeknoten eines Knotens $v \in V_{DG}$, wenn $e_{v,w} \in E$.

1. Ausgabe eines Parameters durch einen OR-Knoten (Fig. 20, Def. 3.52 2. i):

   Jeder der AND- oder FACT-Knoten, die Nachfolgeknoten eines OR-Knoten sind, hat die gleichen Parameter wie der OR-Knoten. Alle korrespondierenden Parameter der Nachfolgeknoten haben laut Analyse der Parametermodi den Modus Ausgabe, da nur dann der Parameter des OR-Knotens den Modus Ausgabe erhält.

   Die Kantenverbindung geht in diesem Fall von jedem korrespondierenden Parameter eines Nachfolgeknotens zum Parameter des OR-Knotens.

2. Eingabe eines Parameters durch einen OR-Knoten (Fig. 21, Def. 3.52 2. ii):

   Alle korrespondierenden Parameter der Nachfolgeknoten haben laut Algorithmus den gleichen Modus (insbesondere durch das expliziten Setzen des Modus in, falls die Modi an den Nachfolgeknoten vorher sowohl Eingabe als auch Ausgabe waren.

   Die Kantenverbindung geht in diesem Fall von jedem korrespondierenden Parameter mit Modus Ausgabe von der Wurzel eines benachbarten, disjunkten Teilgraphen sowohl zum Parameter des OR-Knotens, als auch zu den korrespondierenden Parametern der Nachfolgeknotens des OR-Knotens.

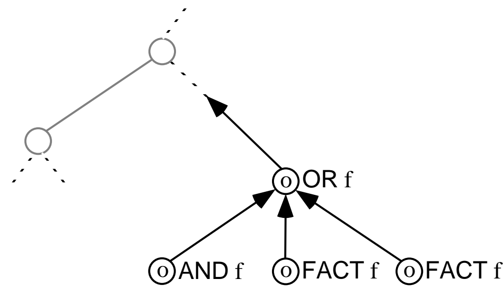


Abbildung 20: Ausgabe eines Parameters durch einen OR-Knoten

Durch die Eingabekante von einem vorherliegenden Knoten erfolgt die Weitergabe von Werten direkt ohne den Umweg über den OR-Knoten. (Die Eingabekante zum OR-Knoten ist dadurch im Prinzip überflüssig.)

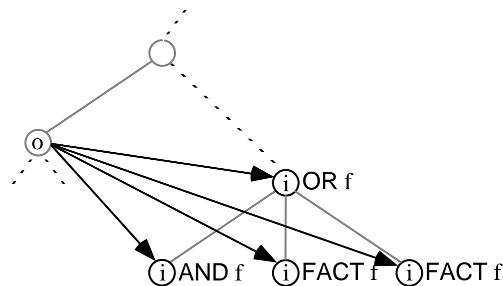


Abbildung 21: Eingabe eines Parameters durch einen OR-Knoten

3. Ausgabe eines Parameters durch einen AND-Knoten, 1. Fall, wenn alle korrespondierenden Parameter an Nachfolgeknoten den Modus Ausgabe haben (Fig. 22, Def. 3.52 2. i):

   Der einzige Unterschied zur Ausgabe bei OR-Knoten ist, daß der Parameter nicht an allen Nachfolgeknoten auftreten muß. Tritt er nicht auf, wird keine Kante erzeugt.

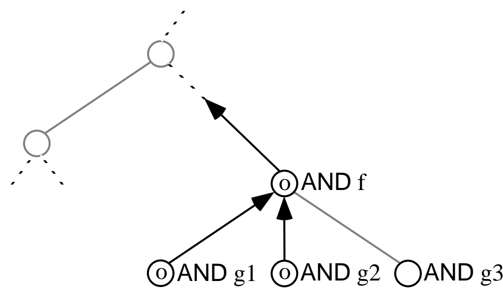


Abbildung 22: Ausgabe eines Parameters durch einen AND-Knoten, 1.Fall

4. Ausgabe eines Parameters durch einen AND-Knoten, 2. Fall, wenn mindestens ein Auftreten in einem Nachfolgeknoten den Modus Ausgabe hat und weitere den Modus Eingabe haben (Fig. 23, Def. 3.52 2. i und iii):

   Für jeden Ausgabeparameter wird eine Kante erzeugt, die den Parameter des Nachfolgeknotens mit dem korrespondierenden Parameter des AND-Knotens verbindet. Diese Kante geht von unten nach oben.

   Ist das Vorkommen eines Parameter durch einen Nachfolgeknoten vom Typ Ausgabe und ein anderes vom Typ Eingabe, so wird eine Kante erzeugt, die die Ausgabe mit der Eingabe verbindet. Gibt es mehrere solcher “Quer-”Kanten muß der Ziel-Nachfolgeknoten die korrekte Verknüpfung der mehrfachen Eingaben übernehmen.

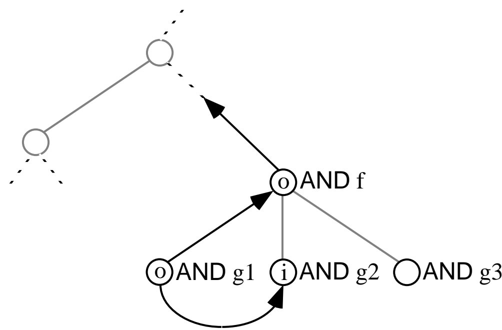


Abbildung 23: Ausgabe eines Parameters durch einen AND-Knoten, 2. Fall

5. Eingabe eines Parameters durch einen AND-Knoten, 1. Fall, wenn alle Auftreten ein und desselben Parameters an Nachfolgeknoten den Modus Eingabe haben (Fig. 24, Def. 3.52 2. ii):

   Dieser Fall ist analog zur Eingabe durch einen OR-Knoten: die Kantenverbindung geht von jedem korrespondierenden Parameter mit Modus Ausgabe an der Wurzel eines benachbarten, disjunkten Teilgraphen zum Parameter des AND-Knotens.

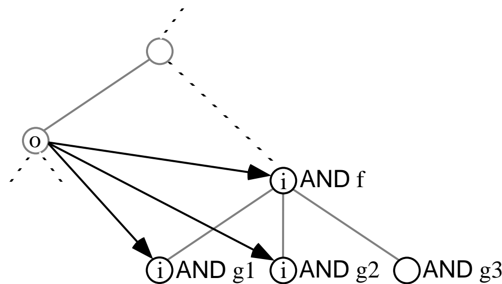


Abbildung 24: Eingabe eines Parameters durch einen AND-Knoten, 1. Fall

6. Eingabe eines Parameters durch einen AND-Knoten, 2. Fall, wenn alle Auftreten des Parameters an Nachfolgeknoten den Modus Ausgabe haben (Fig. 25, Def. 3.52 2. i und ii):

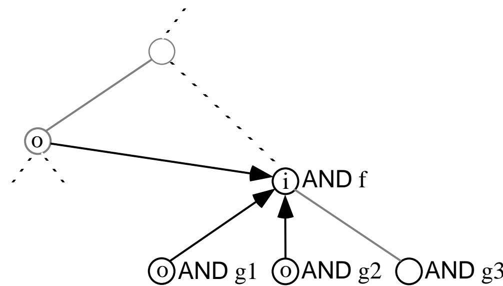


Abbildung 25: Eingabe eines Parameters durch einen AND-Knoten, 2. Fall

Für den Parameter des AND-Knotens wird eine Kante erzeugt, wie sie auch bei der allgemeinen Eingabe durch einen AND-Knoten (1. Fall) erzeugt wird.

Es werden jedoch keine Kanten zu den korrespondierenden Parametern der Nachfolgeknoten erzeugt. Statt dessen gehen von den korrespondierenden Parametern der Nachfolgeknoten Kanten zum Parameter des AND-Knotens.

Dieser Fall entsteht, wenn die Analyse der Ein-Ausgabetypen für einen OR-Knoten, dessen Nachfolgeknoten der AND-Knoten ist, für die Auftreten des Parameters in anderen FACT- oder AND-Nachfolgeknoten den Modus Eingabe festgestellt hat. Der AND-Knoten muß dann die konjunktive Verknüpfung der Eingabewerte von einem äußeren Knoten mit den Ausgabewerten der Nachfolgeknoten übernehmen.

Die Resultatwerte eines Knotens haben immer den Modus Ausgabe. Für sie gelten daher nur die Fälle 1 bis 3 der obigen Beispiele.

Lokale Parameter treten nur an AND-Knoten auf. Sie haben ebenfalls immer den Modus Ausgabe. Auch hier gelten daher nur die Fälle 1 bis 3. Da es sich um lokale Parameter handelt, gibt es außerdem keine Kanten vom betrachteten Knoten nach "oben".

3.4.3 Negation im Datenflußgraph

Für eine ausführbare Negation sind sowohl für Top-Down Verfahren, als auch für Bottom-Up Verfahren, sichere, bzw. zumindest bedingt sichere Programme Voraussetzung (siehe Def. 3.1). Dies bedeutet unter anderem, daß für die in einer Negation auftretenden Parameter an mindestens einer anderen Stelle im Programm Ausgaben erzeugt werden, d.h. alle Parameter der Negation können Eingaben erhalten, wenn dies notwendig ist. Nach den Konsistenzbedingungen für $\mathrm{DATALOG}_f^{\neg}$-Programme in Definition 3.1 ist dies gewährleistet.

Um das Problem einer nicht-korrekten Auswertung der Negation sowohl bei Bottom-Up als auch bei Top-Down Verfahren durch eine (möglicherweise falsche) Bindung von Werten durch negierte Literale zu vermeiden, garantiert zum einen die Vorausetzung bedingt sicherer Programme, daß es für jeden Parameter der Negation eine (weitere) Ausgabestelle gibt, zum anderen bezieht das hier angewendete Auswertungsverfahren wesentliche Elemente des Top-Down Ansatzes mit ein. Die Funktionalität des NOT-Knotens ist so zu wählen, daß keine Bindung von Werten geschieht, sondern das Ergebnis ein Wahrheitswert in Bezug auf an anderen Stellen berechneten Werten ist. (Die entsprechende Funktionalität des NOT-Knotens wird in Abschnitt 3.6 beschrieben.)

Eingaben, die über negierte Literale an Blätter des Graphen gehen, werden von der Negation nicht beeinflußt. Sie können daher, wie in Abschnitt 3.4.2 beschrieben, direkt

an die Eingabestellen gegegeben werden. Die Negation betrifft (in einer noch genauer zu beschreibenden Form) vielmehr die im Teilgraphen unter dem NOT-Knoten erzeugten Ausgaben. Sie müssen daher über den NOT-Knoten gehen. Genau diese Form der Verbindung gewährleistet das in Definition 3.45 vorgestellte Verfahren. Der Beweis für die Korrektheit dieses Verfahrens ist in Kapitel 3.7 nachzulesen.

Figur 27 und Figur 26 zeigen graphisch die Situation für NOT-Knoten, wobei dieselbe Symbolik wie im vorherigen Abschnitt verwendet wird (nicht dargestellt ist dabei die Ausgabe des boolschen Ergebniswertes des NOT-Knotens, die nach oben an den Vorgängerknoten weitergereicht wird). Im ersten Fall erhält der negierte Knoten, d.h. der Nachfolgeknoten des NOT-Knotens eine Eingabe. Im zweiten Fall liefert der negierte Knoten eine Ausgabe, die vom NOT-Knoten mit einer zusätzlichen Eingabe verarbeitet werden muß.

Auf eine Lösung des Problems der in Top-Down Verfahren vorhandene Gefahr der Nicht-Terminierung von nach, bzw. unter der Negation folgenden Rekursionen im Fall unvollständiger Programme wird in Abschnitt 4.1.3 näher eingegangen.

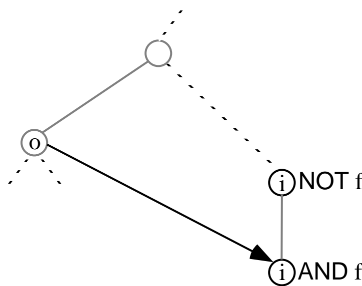


Abbildung 26: Eingabe eines Parameters an einem NOT-Knoten, 1. Fall

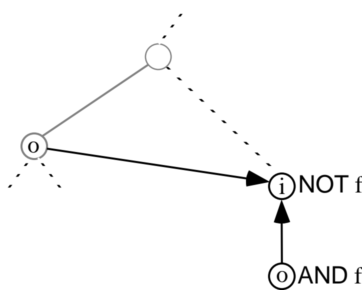


Abbildung 27: Eingabe eines Parameters an einem NOT-Knoten, 2. Fall

3.4.4 Datenflußgraphen mit Rekursionen

Ein Strukturgraph enthält Zyklen, wenn in dem zugrunde liegenden DATALOG$_f^{\neg}$-Programm Rekursionen auftreten. Diese Zyklen müssen nun derart auf den Datenflußgraphen übertragen werden, daß sie einerseits eine korrekte Rekursion implementieren

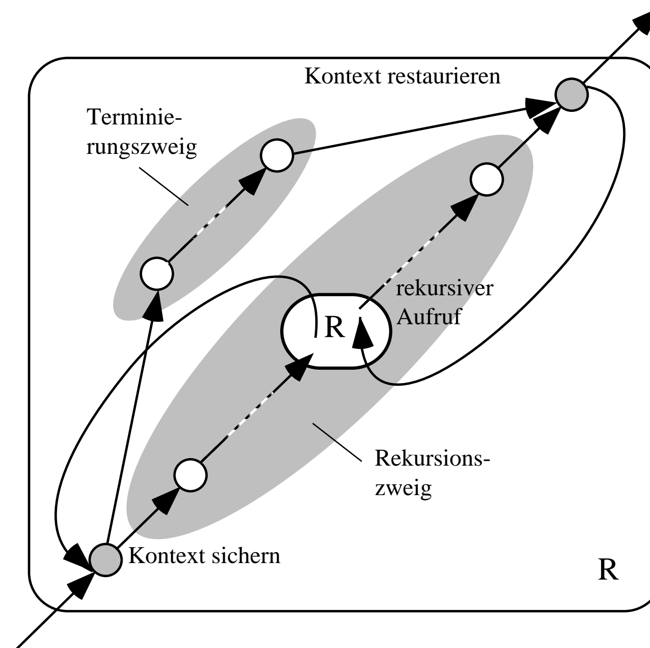


Abbildung 28: Allgemeines Schema für Rekursionen

und andererseits die Möglichkeiten der datenflußgetriebenen Abarbeitung des Graphen nicht einschränken.

Prinzipiell lassen sich Rekursionen ebenso wie Prozeduren problemlos in Datenflußgraphen implementieren [Arvind 83]. Das allgemeine Schema zeigt Figur 28.

Die Idee, die diesem Schema zugrunde liegt, ist, daß eine Art Aufruf des rekursiven Komplexes stattfindet und anschließend, nach der Abarbeitung des Terminierungsfalles, eine schrittweise Rückkehr erfolgt. Die jeweilige Sicherung und Restaurierung des Kontextes stellt dabei sicher, daß für alle Token, die sich in diesem Teilgraphen befinden, bekannt ist, zu welcher Rekursionstiefe sie gehören und damit auch, welche Token zusammen gehören [Ungerer 93]. Ein Verfahren für die Realisierung wird in 4.1.3 vorgestellt.

Um die Prinzipien zu erläutern, soll es zunächst genügen, einfache, lineare Rekursionen zu betrachten.

3.4.4.1 Struktur einer Rekursion

Eine Rekursion erzeugt einen Zyklus im Strukturgraph. Jeder Zyklus enthält genau eine rückführende Kante. Diese führt zum "obersten" Knoten des Zyklus, d.h. zu dem Knoten des Zyklus, auf den man zuerst trifft, wenn man den Strukturgraphen mit Tiefensuche durchwandert.

Der oberste Knoten ist, aufgrund der Bedingung der Konsistenz des Programms (siehe Def. 3.1) immer ein Or-Knoten, da es bei jeder Rekursion zusätzlich zu den Zweigen mit rekursiven Aufrufen immer mindestens einen alternativen Zweig geben muß, in dem kein rekursiver Aufruf stattfindet, d.h. in dem, bei erfolgreicher Ausführung, die Rekursion terminieren kann.

Der Zyklus geht über Knoten, die im allgemeinen sowohl Ein-, als auch Ausgabeparameter besitzen, da bei Rekursionsaufrufen in $\text{DATALOG}_f^{\neg}$ meistens beide Formen von Parametern auftreten. Ein solcher Zyklus, in dem Eingaben in die eine Richtung wandern und Ausgaben logischerweise in die umgekehrte Richtung, läßt sich nicht einfach auf einen Datenflußgraphen übertragen, da sonst das Datenflußprinzip verletzt werden würde, bei dem Daten nur unidirektional transportiert werden dürfen.

Der Zyklus des Strukturgraphen muß daher in zwei verschiedene Schleifen aufgeteilt werden, eine Schleife für die Eingabeparameter und eine Schleife für die Ausgabeparameter. Der Begriff Schleife steht hier für die Menge an Kanten für alle Parameter deren Werte in diesem Zyklus transportiert werden sollen.

In $\text{DATALOG}_f^{\neg}$ hat eine (einfache, lineare) Rekursion (mit einem Terminierungszweig) prinzipiell folgendes Aussehen:

$$R_t \leftarrow T.$$
$$R_r \leftarrow M, D, R, U, N. \;,$$

R_t und R_r sind alternative Regeln R, die durch $_r$ und $_t$ klassifiziert werden:
- R_t wird als Terminierungsregel bezeichnet. Sie kann auch ein Fakt sein: R_t.
- R_r wird als Rekursionsregel bezeichnet.

M, D, U, N und T sind Mengen von Literalen, R ist der (rekursive) Aufruf der Regel R. Optional können die Mengen auch leer sein.

Die Einordnung von Literalen in die Mengen M, D, U und N dient einer groben Klassifizierung und bezieht sich nicht direkt auf die tatsächliche Reihenfolge der Literale im Regelrumpf. Sie motiviert sich dadurch, daß sich die im Rumpf eines Rekursionzweigs vorkommenden Literale in folgende Gruppen unterteilen lassen:

- Literale, die Parameter mit dem rekursiven Aufruf gemeinsam haben:
  D: Literale, die Eingaben an den rekursiven Aufruf liefern.
  U: Literale, die Eingaben vom rekursiven Aufruf erhalten.
  Ein Literal, in dem beide Fälle kombiniert auftreten, ist offensichtlich nicht berechenbar und kommt daher sinnvollerweise nicht vor. Ebenso dürfen die Literale aus D zwar Eingaben an die Literale aus U weitergeben, aber nicht umgekehrt.
- Literale, die keine Parameter mit dem rekursiven Aufruf gemeinsam haben:
  M: Literale, die Eingaben für D erzeugen können. (In diese Menge sollen auch Literale eingeordnet werden, die mit keinen anderen Literalen Parameter gemeinsam haben.)
  N: Literale, die Eingaben von U bekommen können.
  Literale aus M und D dürfen außerdem Eingaben an Literale aus U und N liefern, jedoch nicht umgekehrt, da sonst zyklische (von der Rekursion unabhängige) Datenabhängigkeiten existieren würden. Zusätzlich dürfen die Literale aus M auch Eingaben von D erhalten und die Literale aus N Eingaben an U liefern.

Damit läßt sich, da offensichtlich nur nicht sinnvolle Kombinationen ausgeschlossen wurden, jede einfache, lineare Rekursion in $\text{DATALOG}_f^{\neg}$ in eine solche Form bringen.

Um zu wissen, wie Schleifen für eine Rekursion erzeugt werden sollen, müssen zunächst die Literale in der Terminierungsregel und der rekursiven Regel näher betrachtet werden, die Eingaben erhalten können (M, N, T, aber auch D und U), die Eingaben an den rekursiven Aufruf weitergeben (D), die Ausgaben für die Regeln erzeugen (M, N, T und auch D und U), und die Ausgaben vom rekursiven Aufruf bekommen (U).

Da für jeden Knoten ein Resultatwert existiert, der immer vom Typ Ausgabe ist, gibt es bei jeder Form der Rekursion mindestens eine Ausgabe. Es kann jedoch der Fall auftreten, daß es keine Eingabeparameter gibt.

Die folgenden Beispielgraphen dienen lediglich zur Veranschaulichung der verschiedenen Datenabhängigkeiten, sie sind jedoch keine Datenflußgraphen.

Es werden folgende Vereinbarungen getroffen:
- R_v steht für den OR-Knoten, der den Terminierungszweig und den Rekursionszweig vereinigt.
- d (down) und u (up) sind Kennzeichnungen für die Art der Schleife und dienen nur zur Veranschaulichung. d bezieht sich darauf, daß es sich um eine Schleife für die Eingaben handelt, u bedeutet, daß es sich um eine Schleife für die Ausgaben handelt.
- Die Pfeilrichtung entspricht der Richtung von Ausgabe an einem Knoten zu Eingabe an einem anderen Knoten.

Beispiele:

1. Gibt es nur Ausgabeparameter in R, so erfolgt in jeder Rekursionstiefe eine Ausgabe. Änderungen können dabei nur auftreten, wenn die Parameter von R im Rumpf der Rekursionsregel zu R_r vertauscht sind. Die Rekursion terminiert, wenn sich die von R_r erzeugte Menge an Tupel nicht mehr verändert.
   Figur 29 a) zeigt eine Situation bei der M, D, U und N leer sind. In Figur 29 b) existieren dagegen M und U.

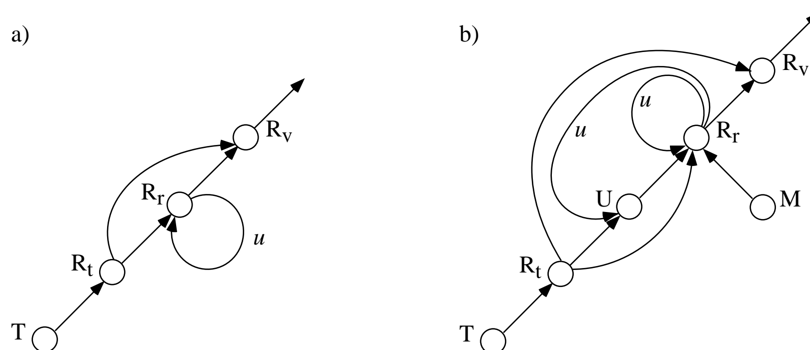


Abbildung 29: Rekursion mit Ausgabeparametern

   Unendliche Rekursionen können entstehen, wenn in der Rekursionsregel oder einer von einem Literal aus U aufgerufenen Regel Funktionen und Operationen zugelassen sind. Auf eine Behandlung dieser Problematik wird in Abschnitt 4.1.3 näher eingegangen.
   Treten dagegen nur FACT- oder IFACT-Knoten auf, läßt sich das Abbruchskriterium durch das Erreichen des kleinsten Fixpunktes der Menge der Ausgaben bestimmen, wie dies bei Bottom-Up Verfahren üblich ist [Cremers 94].

2. Hat R Eingabeparameter, so bedeutet das gleichzeitig, daß auch Literale in T in jeder (absteigenden) Rekursionsschleife Eingaben erhalten. M kann außerdem Eingaben vom Regelkopf R_r erhalten.

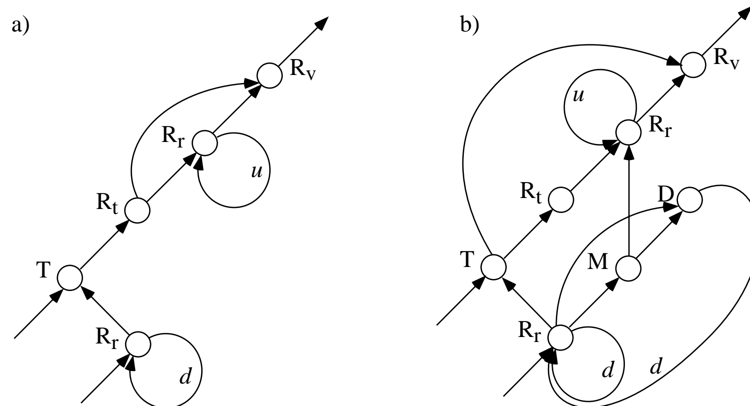


Abbildung 30: Rekursion mit Ein- und Ausgabeparametern

Figur 30 a) zeigt ein Beispiel bei dem M, D, U und N leer sind. In Figur 30 b) existieren D und M. M kann in jeder Rekursionstiefe Zwischenergebnisse für die Ausgabe in R_r liefern.
Gibt es keine Literale in D oder M, so entsteht die Schleife nur durch das mögliche Verändern der Eingabeparameter von R zu R_r.
Die absteigende Rekursionschleife ist beendet, wenn bei einem Schleifendurchlauf mindestens ein Literal in D scheitert. Alternativ ist sie beendet, wenn der kleinste Fixpunkt über die Menge der Eingaben erreicht ist. Ist D leer, oder treten Systemprädikate im Rekursionszweig oder in vom Rekursionszweig aufgerufenen Regeln auf, kann dies zu unendlichen Rekursionen führen.
Die Ausgabeschleife wird nur dann durchlaufen, wenn die Literale in T für mindestens eine Eingabe erfolgreich waren. Die Anzahl der Durchläufe muß gleich der Anzahl der Durchläufe durch die Eingabeschleife sein.
Berechnet M während der Eingabeschleife für jede Rekursionstiefe Ausgaben, müssen diese "zwischengespeichert" werden. In jeder Ausgabeschleife muß diejenige Ausgabe von M verwendet werden, die durch eine Eingabe aus einer Eingabeschleife gleicher Rekursionstiefe erzeugt wurde.

Zusätzlich zu diesem Schema muß das Datenflußmodell folgende Randbedingungen erfüllen:

- Es muß Knoten mit einer Funktionalität geben, die die Verwaltungsarbeit für die Ausführung der Rekursion leistet, z.B. die Werte hinsichtlich der aktuellen Rekursionstiefe markieren, oder zur Ausführung in einer bestimmten Rekursionstiefe aussuchen.
- Zwischenergebnisse, die bei jeder Rekursionstiefe bei Ausführung der Eingabeschleife anfallen können und die bei Berechnungen der Ausgabeschleife benötigt werden, müssen zwischengespeichert werden.
- Es muß Kriterien zur Terminierung von Rekursionen geben, da nicht bei jeder Programmstruktur automatisch eine Terminierung der Rekursion gewährleistet ist.

Der Strukturgraph hat bei einer Rekursion mit diesem Schema die in Figur 31 gezeigte Form, wobei zur Vereinfachung M, D, U, N und T nicht als Mengen von Literalen sonderen als Literale selbst angenommen wurden.

Im Datenflußgraph müssen anstelle der rückführenden Kante nun die im Sinne der Datenflußrechnerarchitekturen korrekten Datenwege wiedergespiegelt werden, d.h. es

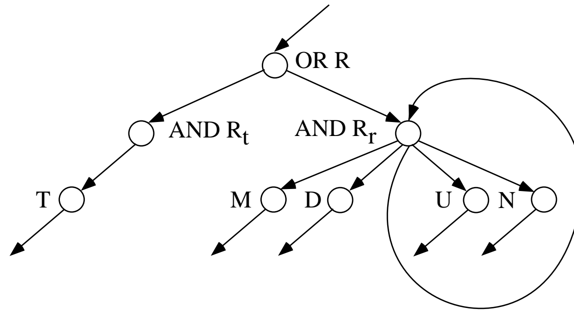


Abbildung 31: Rekursion im Strukturgraphen

muß Kanten geben, die die Eingabeschleife realisieren, wenn es Eingaben gibt, und Kanten, die die Ausgabeschleife realisieren.

Im Unterschied zum bisher konstruierten Datenflußgraphen dürfen außerdem die Eingaben an die Rekursion nicht mehr direkt an die unterste Verwendungsstelle im Graphen gehen, sondern müssen über einen Rekursions-Eintrittsknoten gehen. Analog müssen die Ausgaben über einen Rekursions-Austrittsknoten gehen. Das gleiche gilt für den Terminierungszweig. Ist dieser beim ersten Aufruf der Rekursion erfolgreich, müssen die Ergebnisse direkt nach außen weitergegeben werden können. Ist er jedoch erst nach Ausführung einiger absteigender Rekursionsschleifen erfolgreich, müssen die Ergebnisse an die aufsteigende Rekursionsschleife gegeben werden.

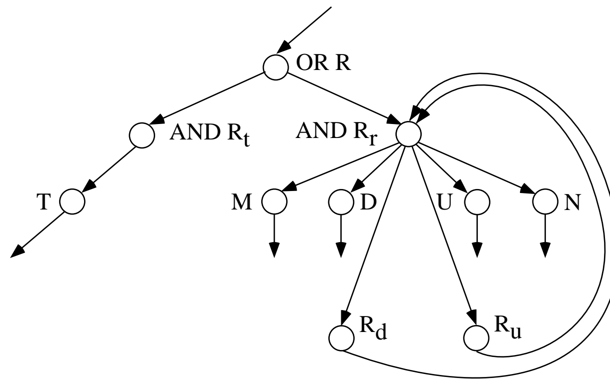


Abbildung 32: Einfügen der Knoten v_d und v_u (im Strukturgraphen dargestellt)

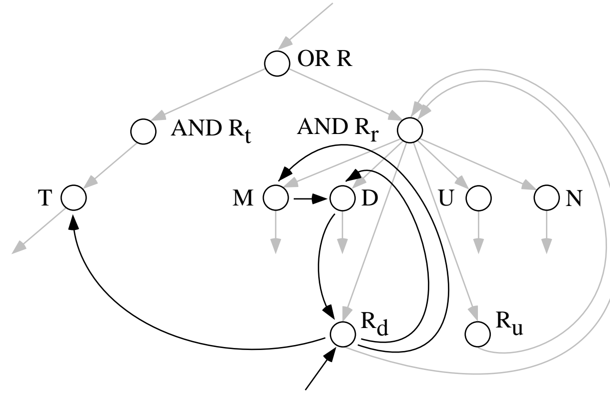


Abbildung 33: Eingabewege über den Knoten v_d

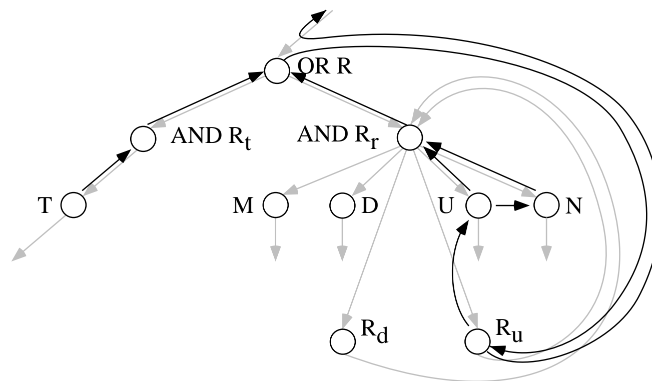


Abbildung 34: Ausgabewege über den Knoten v_u

Es müssen also zusätzlich zwei neue Knoten, "Sichern des Kontextes" und "Restaurieren des Kontextes", eingeführt werden, die die Verwaltungsarbeit übernehmen: v_d und v_u.

Figur 32 zeigt den Strukturgraphen mit den eingefügten Knoten. Die Bezeichnungen R_d und R_u stehen für die Namen der Knoten v_d und v_u. Diese Grafik dient jedoch nur zur Veranschaulichung der Einfügestellen, die Knoten werden im Rahmen dieses Modells erst direkt im Datenflußgraphen eingeführt.

Alle Eingaben für den rekursiven Aufruf werden über den Knoten v_d an die Knoten der Rekursionsschleife und des Terminierungszweigs geschickt, die diese Eingaben bekommen sollen (siehe Figur 33). Alle von der Rekursionsschleife oder dem Terminierungszweig produzierten Ausgaben werden über den Knoten v_d an die Knoten geschickt, die diese als Eingaben benötigen (siehe Figur 34). Dabei entstehen automatisch mindestens zwei getrennte Zyklen im Graphen.

Figur 33 und Figur 34 dienen zur Veranschaulichung der im Datenflußgraph zu generierenden Kanten. R_d und R_u stehen für die Namen der Knoten v_d und v_u.

3.4.4.2 Konstruktionsprinzip

Die Knoten "Sichern des Kontextes" v_d und "Restaurieren des Kontextes" v_u müssen nun so in den Graphen eingefügt werden, daß sich einerseits das in Abschnitt 3.4.4, Figur 28 (siehe auch Figur 35a)) gezeigte Schema ergibt und andererseits eine korrekte Rekursion im Sinne von $\text{DATALOG}_f^{\neg}$ implementiert wird. Figur 35 b) zeigt den entsprechenden Datenflußgraphen.

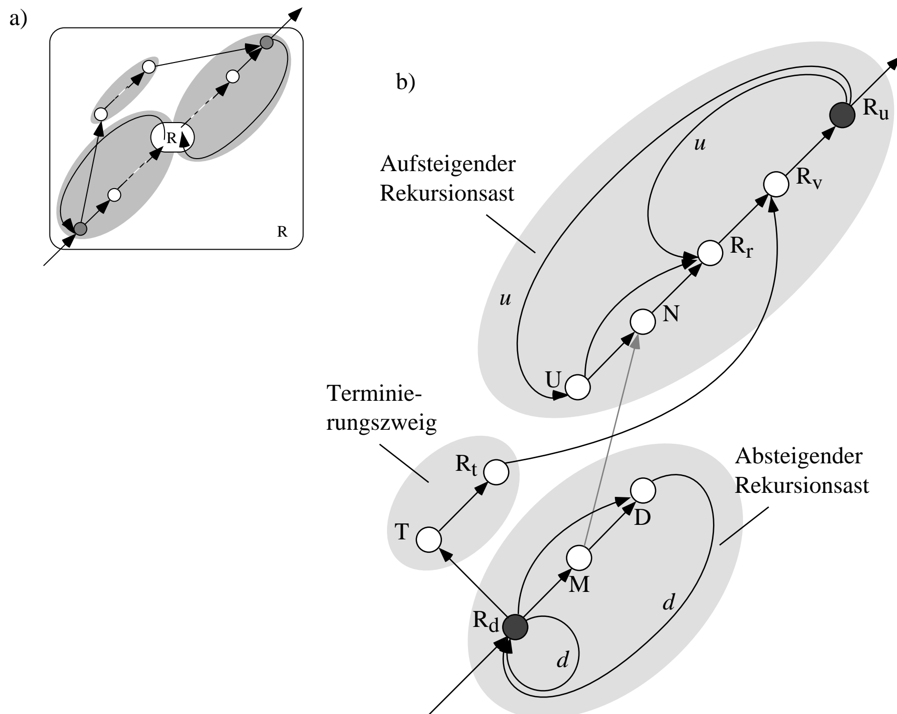


Abbildung 35: Rekursion als Datenflußgraph

R_u steht für den Knoten v_u, R_d für den Knoten v_d. Der Knoten v_d hat die Aufgabe einen neuen Kontext zu generieren und den alten zu sichern, v_u die Aufgabe, den jeweils letzten Kontext aus dem aktuellen Kontext zu restaurieren. R_r steht für den AND-Knoten, der den Kopf der Rekursionsregel repräsentiert, R_t steht für den AND-Knoten, der den Kopf der Terminierungsregel repräsentiert, R_v für den OR-Knoten, der im Strukturgraph beide Zweige verbindet. M, D, U, N und T stehen für Rumpfliterale und sind wie in Abschnitt 4.1.3 definiert.

Wie sich anhand von Figur 33 und Figur 34 sehen läßt, gelten für die Erzeugung der Kanten eines Datenflußgraphen bei einer Rekursion analoge Regeln wie für die Erzeugung eines Datenflußgraphen ohne Zyklen, bis auf folgende Unterschiede:

- Die Eingabekanten gehen im Rekursionszweig nicht direkt an die untersten Verwendungsstellen, sondern über den Knoten v_d.
- Ein- und Ausgaben werden nun auch zwischen den Zweigen unter einem OR-Knoten ausgetauscht, d.h. zwischen Terminierungs- und Rekursionszweig. Insbesondere auch zwischen verschiedenen Terminierungs- und Rekursionszweigen im Fall allgemeiner, linearer Rekursionen. v_d übernimmt dabei die Weitergabe von Werten an die verschiedenen Zweige für den nächsttieferen Schleifendurchlauf. Das Zusammenfassen der Ergebnisse von Terminierungs- und Rekursionszweig übernimmt der OR-Knoten der schon im Strukturgraphen die verschiedenen Zweige vereinigt. Der Knoten v_u übernimmt anschließend wiederum das Versenden der Ergebnisse an den richtigen Rekursionszweig für den nächsthöheren Schleifendurchlauf.
- Die Knoten v_d und v_u müssen außer Verwaltungstätigkeit für die Rekursion keine weitere Funktionalität besitzen: Werte müssen nur weitergegeben werden. Gibt es mehrere Rekursions- und Terminierungszweige, müssen die Knoten auch die Weitergabe von Werten übernehmen können, die an verschiedene Zweige gehen.

Definition 3.54 *Strukturgraph mit Verwaltungsknoten für Rekursionen*

Ein Strukturgraph mit Verwaltungsknoten für Rekursionen $SG_{\mathcal{L}}^{R} = (V^{R}, E^{R})$ sei definiert durch einen erweiterten Strukturgraphen $SG_{\mathcal{L}}^{E} = (V_{SG}, E_{SG})$ (mit $E^v, E^m, E^r \subset E_{SG}$ definiert wie in Definition 3.35) und:

1. $V^{R} = V \cup \{v^u \mid \exists v, w \in V : e_{w,v} \in E^r\} \cup$
$\{v^d \mid \exists v, w \in V : e_{w,v} \in E^r, v$ hat Parameter $\delta_1, ..., \delta_u,$
$u \in \mathbb{N}$ und $\delta_i.mode = \mathtt{in}$ für ein $i \in \{0, ..., u\}\}$.

   Es gelte außerdem für alle $v^d, v^u \in V^{R}$:

   - $v^u.type = $ RECUP,
     $v^d.type = $ RECDOWN,
   - $v_u.name = v.name$,
     $v_d.name = v.name$.
   - v_d und v_u haben die gleichen Parameter wie v (einschließlich ihrer Identifikatoren).

2. $E^{vm} = ((E^v \cup E^m) \setminus$
$\{e_{v,w} \mid \exists e_{v,w} \in E^v \cup E^m$ und $\exists w_u \in V^{R}$ mit $w_u.name = w.name\}) \cup$
$\{e_{v,w_u} \mid \exists e_{v,w} \in E^v \cup E^m$ und $\exists w_u \in V^{R}$ mit $w_u.name = w.name\} \cup$
$\{e_{w_u,w} \mid \exists e_{v,w} \in E^v \cup E^m$ und $\exists w_u \in V^{R}$ mit $w_u.name = w.name\} \cup$
$\{e_{v,w_d} \mid \exists e_{v,w} \in E^v$ und $\exists w_d \in V^{R}$ mit $w_d.name = w.name\} \cup$
$\{e_{w_d,w} \mid \exists e_{v,w} \in E^v$ und $\exists w_d \in V^{R}$ mit $w_d.name = w.name\}$

   $E^{R} = E^{vm} \cup E^r$.

Bemerkung: Zu jedem Zyklus im Strukturgraphen mit oberstem Knoten v gibt es mindestens v_u mit $v_u.type$ = RECUP. ◇

Satz 3.55

Die Knoten $v_u, v_d \in V^R$ mit $v_u.type$ = RECUP und $v_d.type$ = RECDOWN werden derart in den Strukturgraphen integriert, daß Satz 3.25 (zu jedem Parameter eines Nachfolgeknotens gibt es einen korrespondierenden Parameter am Vorgängerknoten) nicht verletzt wird. ◇

Beweis:

Existiert für $v \in V$: $v_u, v_d \in V^R$ mit $v.name = v_u.name = v_d.name$, sowie $w \in V$ mit $e_{w,v} \in E^v \cup E^r$, so werden die Knoten v_u und v_d zwischen w und v eingefügt, d.h. v_u und v_d sind Nachfolgeknoten von w in E^R. v_d und v_u sind insbesondere die Parameter betreffend Kopien von v. Da für v als Nachfolgeknoten von w in E Satz 3.25 gilt, gilt nun Satz 3.25 auch für v_u und v_w als Nachfolgeknoten von w in E^R.

Für v als Nachfolgeknoten von v_u und v_w in E^R gilt Satz 3.25 offensichtlich aufgrund der Identität der Parameter. ◇

Definition 3.56 *Datenflußgraph mit Zyklen*

Sei $SG^R_{\mathcal{L}} = (V^R, E^R)$ ein Strukturgraph mit Verwaltungsknoten für Rekursionen (mit E^v, E^m, E^r definiert wie in Definition 3.35, E^{vm} wie in Definition 3.56), und $DG_{\mathcal{L}} = (V_{DG}, E_{DG}, Fun_{DG})$ ein Datenflußgraph.

Ein Datenflußgraph mit Rekursionen $DG^R_{\mathcal{L}} = (V^R_{DG}, E^R_{DG}, Fun^R_{DG})$ sei ein Graph gegeben durch den zyklenfreien Datenflußgraphen $DG_{\mathcal{L}}$ (mit Bezeichnungen für Parameter aus Def. 3.20 und $u \in \mathbb{N}$):

1. V_{DG} wird durch folgende Erweiterung zu V^R_{DG}:

   a) $\forall v \in V_{DG} : \ v \in V^R_{DG}$

   b) wenn $v_d \in V^R$ mit $v.type$ = RECDOWN, Parametern δ_1, ..., δ_u und Ergebnisparameter δ_{u+1}, dann ist $v_d \in V_{DG}$ mit
   - Eingabeparametern $v_d.IP = \{\delta_i \mid i \in \{1,...,u\}, \delta_i.mode = \mathsf{in}\} \cup \{\delta_i \mid i \in \{1,...,u\}, \delta_i.mode = \mathsf{out}\} \cup \{\delta_{u+1}\}$
   - Ausgabeparametern $v_d.OP = v_d.IP$
   - $v_d.const = 0$

   c) wenn $v \in V^{R_\circ}$ mit $v.type$ = RECUP, Parametern δ_1, ..., δ_u und Ergebnisparameter δ_{u+1}, dann ist $v_u \in V_{DG}$ mit
   - Eingabeparametern $v_u.IP = \{\delta_i \mid i \in \{1,...,u\}, \delta_i.mode = \mathsf{out}\} \cup \{\delta_{u+1}\}$
   - Ausgabeparametern $v_u.OP = v_u.IP$
   - $v_u.const = 0$

2. E^R_{DG} sei definiert durch $E^R_{DG} = E^{vm}_{DG} \cup E^r_{DG}$
   Für die Erzeugung der Kanten in E^{vm}_{DG} gilt zunächst nur wenig modifiziert das Gleiche wie für E_{DG}:

   i) $e_{w.op_j, v.ip_i} \in E^{vm}_{DG}$
   wenn $\exists v, w \in V_{DG}$ mit $e_{v,w} \in E^{vm}$ und
   $\exists i, j \in \mathbb{N}$, $1 \leq i \leq v.in$, $1 \leq j \leq w.out$,
   mit $v.ip_i.ident = w.op_j.ident$.

ii) $e_{v'.op_{i'},w.ip_j} \in E^{vm}_{DG}$
wenn $\exists v, w \in V_{DG}$ mit $e_{v,w} \in E^{vm}$ und $v.type \neq$ RECDOWN
$\exists i, j \in \mathbb{N}$, $1 \leq i \leq v.in$, $1 \leq j \leq w.in$, mit $v.ip_i.ident = w.ip_j.ident$.
und $\exists v' \in V$, $i' \in \mathbb{N}, 1 \leq i' \leq v'.out$ mit $e_{v,v'} \notin E^v \cup E^m$ und $e_{v'.op_{i'},v.ip_i} \in E_{DG}$

iii) $e_{w'.op_{j'},w.ip_j} \in E^{vm}_{DG}$
wenn $\exists v, w \in V_{DG}$ mit $e_{v,w} \in E^{vm}$ mit $v.type =$ AND,
$\exists i, j \in \mathbb{N}$, $1 \leq i \leq v.in$, $1 \leq j \leq w.in$
mit $v.ip_i.ident = w.ip_j.ident$
und $\exists w' \in V$, $j' \in \mathbb{N}, 1 \leq j' \leq w'.out$ mit $v.ip_i.ident = w'.op'_j.ident$.

iv) $e_{v.op_i,w.ip_j} \in E^{vm}_{DG}$
wenn $\exists v, w \in V_{DG}$ mit $e_{v,w} \in E^{vm}$ und $v.type =$ RECDOWN
$\exists i, j \in \mathbb{N}$, $1 \leq i \leq v.in$, $1 \leq j \leq w.in$, mit $v.ip_i.ident = w.ip_j.ident$.

Die rückführenden Kanten in E^r_{DG} werden wie folgt erzeugt, wobei *loop_ident* den Identifikator in der ID-Liste für die rückführenden Kanten bezeichnet. Diesen Identifikator haben daher die Parameter des obersten Schleifenknotens im Strukturgraphen und damit auch die Parameter der Knoten v_d und v_u (siehe Abschnitt 3.3.3):

v) $e_{w_u.op_j,w'.ip_{j'}} \in E^r_{DG}$
wenn $\exists w, w' \in V_{DG}$ mit $e_{w',w} \in E^r$ und
$\exists w_u \in V_{DG}$ mit $w.name = w_u.name$ und
$\exists j, j' \in \mathbb{N}$, $1 \leq j \leq w_u.out$, $1 \leq j' \leq w'.in$,
mit $w_u.op_j.loop_ident = w'.ip_{j'}.ident$

vi) $e_{w'.op_{j'},w_d.ip_j} \in E^r_{DG}$
wenn $\exists w, w' \in V_{DG}$ mit $e_{w',w} \in E^r$ und
$\exists w_d \in V_{DG}$ mit $w.name = w_d.name$ und
$\exists j, j' \in \mathbb{N}$, $1 \leq j \leq w_d.out$, $1 \leq l \leq w'.in$,
mit $w'.op_{j'}.ident = w_d.ip_{j'}.loop_ident$

3. Fun^R_{DG} ist eine Erweiterung von Fun_{DG} mit der Funktionalität der Knoten v_u mit $v_u.type =$ RECUP und v_d mit $v_d.type =$ RECDOWN. ◇

Für eine algorithmische Formulierung der obigen formalen Definition sei auf Abschnitt 3.4.4.3 verwiesen.

Satz 3.57

Für einen Strukturgraphen ohne Zyklen wird der Datenflußgraph nach Definition 3.56 identisch zu einem Datenflußgraphen nach Definition 3.52 erzeugt. ◇

Beweis:

- Da es keine rückführenden Kanten in E gibt, d.h. $E^r = \emptyset$, werden keine Knoten v_u mit $v_u.type =$ RECUP und v_d mit $v_d.type =$ RECDOWN erzeugt. Es ist dann $V^R_{DG} = V_{DG}$.
- Die Kanten werden ausschließlich nach Def. 3.56 2. i bis iii und damit analog zu Def. 3.52 2. i bis iii erzeugt. Es ist dann $E^R_{DG} = E_{DG}$. ◇

Als Instanz einer Rekursionsschleife wird im folgenden die Ausführung einer Rekursionsschleife bezüglich einer Iterationstiefe verstanden.

Der zwischen den Knoten v_d und v_u mit $v_u.type =$ RECUP und $v_d.type =$ RECDOWN und $v_u.name = v_d.name$ aufgespannte Teilgraph soll als Rekursions-Teilgraph bezeichnet werden.

Für die Knoten v, v_d und v_u mit $v_u.type$ = RECUP oder $v_d.type$ = RECDOWN und mit $v.name = v_u.name = v_d.name$ (d.h. v_u und v_d sind aus v entstanden) gilt:

i) Der Knoten v_d hat als Parameter die Eingabeparameter von $v \in V$, der Knoten v_u hat als Parameter die Ausgabeparameter von $v \in V$.

ii) Für alle Übergabeparameter und Ergebnisparameter von $v_u, v_d \in V_{DG}$ gelten die üblichen Kantenverbindungen wie in Definition 3.52 angegeben, wobei gilt:

- v_u und v_d werden wie Vorgängerknoten von v behandelt.
- alle Eingaben gehen nicht direkt an die Eingabestelle, sondern über v_d.

  Es gibt nun weder direkte Kanten von einer Ausgabestelle außerhalb des Rekursions-Teilgraphen, noch gibt es direkte Kanten von einer Ausgabestelle in einer Instanz einer Rekursionsschleife zur Eingabestelle einer anderen, tieferen Instanz einer Rekursionsschleife.

  Für Ausgaben an eine höhere Instanz der Rekursionsschleife gilt, daß sie wie bisher über alle zwischenliegenden Knoten gehen und damit auch über v_u.

  Innerhalb einer Instanz einer Rekursionsschleife können jedoch wiederum alle Kanten von der obersten möglichen Ausgabestelle (eines benachbarten, disjunkten Teilgraphens, nach Def. 3.51) direkt an die Eingabestellen der Knoten der gleichen Instanz gehen.
- Die Verbindung eines Knotens w' mit w_d, wenn $E_{w',w} \in E^r$, über den zweiten Identifikator stellt sicher, daß die Kantenverbindung den Parametern des Aufrufs entspricht. Diesen zweiten Identifikator gibt es im Fall eines Rekursionsaufrufs immer für den obersten Knoten der Rekursionsschleife (vergl. Abschnitt 3.3.2 und 3.3.3).

Dieses Schema implementiert eine Rekursion, indem genau die Datenwege der Rekursion durch Kanten realisiert werden:

- Für alle Eingabeparameter des rekursiven Aufrufs gehen Kanten von v_d direkt zu allen Eingabeparametern von Knoten einer tieferen Instanz der Rekursionsschleife, die Eingaben vom Kopf der rekursiven Regel und damit vom rekursiven Aufruf erhalten sollen.
- Für alle Ausgabeparameter des rekursiven Aufrufs gehen Kanten von den Knoten, von denen eine Ausgabe für den Kopf der rekursiven Regel und damit für den rekursiven Aufruf kommt, zu v_u und von dort an die Knoten der Rekursionsschleife, die Ausgaben einer tieferen Instanz der Rekursionsschleife erhalten sollen.

Zusätzlich kann der Knoten v_d die Aufgabe "Sichern des Kontextes" übernehmen, bevor Eingaben an die Rekursionsschleife oder den Terminierungszweig gehen. Analog kann der Knoten v_u die Aufgabe "Restaurieren des Kontextes" in umgekehrter Richtung ausführen.

Satz 3.58

$DG^R_{\mathcal{L}}$ ist ein Datenflußgraph entsprechend Definition 3.49 $\diamond$

Beweis:

Sei $SG^R_{\mathcal{L}} = (V^R, E^R)$ ein Strukturgraph mit Verwaltungsknoten für Rekursionen, $DG^R_{\mathcal{L}} = (V^R_{DG}, E^R_{DG}, Fun^R_{DG})$ ein Datenflußgraph mit Rekursionen.

$DG_{\mathcal{L}}$ ist ein Datenflußgraph (Satz 3.53). Für $DG^R_{\mathcal{L}}$ gilt zusätzlich:

i) zu zeigen: $EK \cup AK = \emptyset$
Die Knoten $v_u, v_d \in V_{DG}$ mit $v_u.type$ = RECUP oder $v_d.type$ = RECDOWN haben sowohl Ein- als auch Ausgabeparameter. Daher gilt: $v_d, v_u \notin EK \cup AK$. Damit wird auch die für $DG^R_{\mathcal{L}}$ geltende Bedingung $EK \cap AK = \emptyset$ nicht verletzt.

ii) zu zeigen: für alle Knoten $v \in V^R_{DG}$ gilt:
- $v.ip_i.in > 0$ für $i = 1, ..., v.in$, $v.in > 0$.
- $v.op_i.out > 0$ für $i = 1, ..., v.out$, $v.out > 0$.

1. Für einen Knoten $w_u \in V^R_{DG}$ mit $w_u.type$ = RECUP gilt:
$\exists w \in V^R_{DG}$ mit $w_u.name = w.name$, $e_{w_u,w} \in E^{vm}$. Außerdem gibt es zu jedem Ausgabeparameter von w sowohl einen korrespondierenden Eingabeparameter, als auch einen korrespondierenden Ausgabeparameter an w_u (Def. 3.56 1). Umgekehrt gibt es zu jedem Eingabeparameter von w_u einen korrespondierenden Ausgabeparameter an w. Damit gilt:

   - von jedem Ausgabeparameter von w gibt es nach Definition 3.56 2.i) eine Kante zum korrespondierenden Eingabeparameter an w_u, bzw. zu jedem Eingabeparameter von w_u eine Kante von einem Ausgabeparameter von w.
   - Wenn für jeden Ausgabeparameter von w_u gilt, daß ein korrespondierender Ausgabeparameter am Knoten w existiert, gibt es an einem Knoten $v \in V^R_{DG}$ mit $e_{v,w} \in E$ nach Definition 3.25 einen korrespondierenden Eingabeparameter. Folglich gibt es auch zu jedem Ausgabeparameter von w_u einen Eingabeparameter an v, so daß nach Definition 3.56 2.i) eine Kante erzeugt werden kann.

2. Für einen Knoten $w_d \in V^R_{DG}$ mit $w_d.type$ = RECDOWN gilt:
$\exists w \in V^R_{DG}$ mit $w_d.name = w.name$, $e_{w_d,w} \in E^{vm}$. Außerdem gibt es zu jedem Eingabeparameter von w sowohl einen korrespondierenden Eingabeparameter, als auch einen korrespondierenden Ausgabeparameter an w_d (Def. 3.56 1). Umgekehrt gibt es zu jedem Ausgabeparameter von w_d einen korrespondierenden Eingabeparameter an w. Damit gilt:

   - von jedem Ausgabeparameter von w_d gibt es nach Definition 3.56 2.iv) eine Kante zum korrespondierenden Eingabeparameter an w, bzw. zu jedem Eingabeparameter von w eine Kante von einem Ausgabeparameter von w_d.
   - Wenn es für jeden Eingabeparameter von w_d einen korrespondierenden Eingabeparameter an w gibt, dann gibt es nach Definition 3.42 zu ihm mindestens einen Ausgabeparameter außerhalb des Teilgraphen mit Wurzel w und es kann nach Definition 3.56 2.ii oder iii eine Kante erzeugt werden.

3. Für einen Knoten $w \in V^R_{DG}$ mit $w.type \neq$ RECDOWN, $w.type \neq$ RECUP gilt:

   a) Zu jedem Ausgabeparameter des Knotens w gibt es einen korrespondierenden Parameter an einem Knoten $v \in V^R_{DG}$ mit $e_{v,w} \in E^{vm}$ im erweiterten Strukturgraphen:

      - existiert kein $w_u \in V^R_{DG}$ mit $w_u.type$ = RECUP, $w_u.name = w.name$ so gilt $v.type \neq$ RECUP. Damit gibt es einen korrespondierenden Parameter nach Satz 3.25.
      - existiert $w_u \in V^R_{DG}$ mit $w_u.type$ = RECUP, $w_u.name = w.name$, so wähle $v = w_u$. Damit gibt es einen korrespondierenden Parameter nach Definition 3.56 1.

   Damit wird nach Definition 3.56 2.i) eine Kante erzeugt.

b) Zu jedem Eingabeparameter des betrachteten Knotens gibt es es laut Konsistenzbedingung für die Vergabe der Parametermodi (Def. 3.42) einen korrespondierenden Parameter, der eine Ausgabe erzeugt.

- existiert $w_d \in V^R_{DG}$ mit $w_d.type$ = RECDOWN, $w_d.name = w.name$, so gibt es für jeden Eingabeparameter an w einen korrespondierenden Ausgabeparameter von w_d und es wird eine Kante nach 3.56 2.iv erzeugt.
- existiert kein $w_d \in V^R_{DG}$ mit $w_d.type$ = RECDOWN, $w_d.name = w.name$, und gibt es zu einem Eingabeparameter an einem Knoten $w \in V^R_{DG}$ einen korrespondierenden Ausgabeparameter an einem Nachfolgeknoten $v \in V^R_{DG}$ mit $e_{w,v} \in E^v \cup E^m$:
  · existiert kein $v_u \in V^R_{DG}$ mit $v_u.type$ = RECUP, $v_u.name = v.name$, dann ist $e_{w,v} \in E^{vm}$.
  · existiert $v_u \in V^R_{DG}$ mit $v_u.type$ = RECUP, so ist $e_{w,v} \in E^{vm}$ $v_u.name = v.name$, so ist $e_{w,v_u} \in E^{vm}$. Für jeden Ausgabeparameter von v gibt es einen korrespondierenden Ausgabeparameter von v_u.

  Damit wird nach 3.56 2.i eine Kante erzeugt.
- existiert kein $w_d \in V^R_{DG}$ mit $w_d.type$ = RECDOWN, $w_d.name = w.name$, und gibt es zu einem Eingabeparameter an einem Knoten einen korrespondierenden Ausgabeparameter an einem Nachfolgeknoten, so muß es nach Definition 3.42 einen Knoten $v \in V^R_{DG}$ geben, der die Wurzel eines benachbarten, disjunkten Teilgraphen ist und einen korrespondierenden Parameter besitzt, der eine Ausgabe erzeugt.
  · existiert kein $v_u \in V^R_{DG}$ mit $v_u.type$ = RECUP, $v_u.name = v.name$, so wird nach 3.56 2.ii oder iii eine Kante erzeugt.
  · existiert $v_u \in V^R_{DG}$ mit $v_u.type$ = RECUP, $v_u.name = v.name$, so gibt es für jeden Ausgabeparameter an v einen korrespondierenden Ausgabeparameter von v_u.

  Damit wird nach 3.56 2.ii oder iii eine Kante erzeugt.

iii) zu zeigen: $e_{v.op_i} = e_{w.ip_j}$ für $i, j \in \mathbb{N} \Rightarrow v \neq w \; \forall v, w \in V$
Es gilt: $w \neq v_d, v_u$, $v \neq v_d, v_u$ mit $w_u.type$ = RECUP, $w_d.type$ = RECDOWN, $w_u.name = w_d.name = w.name$, da die Knoten neu erzeugt werden (Def. 3.56 1).
Für $e_{w,v} \in E^v \cup E^m$ gilt: alle Pfade von $v \in V^R_{DG}$ nach $w \in V^R_{DG}$ und umgekehrt gehen über v_d oder v_u (Def. 3.56 2 i bis iv).
Außerdem gilt für $w' \in V$ mit $e_{w',v} \in E^r$, daß alle Pfade von $w' \in V^R_{DG}$ nach $v \in V^R_{DG}$ ebenfalls über v_d oder v_u gehen (Def. 3.56 2 v und vi). Es gibt außerdem keine Kante von v_d nach v_d und von v_u nach v_u. ◇

Satz 3.59

Betrachtet man für $DG^R_{\mathcal{L}}$ nur die Kanten aus E^{vm}_{DG}, so enthält der Graph keine Zyklen. ◇

Beweis:

Sei $SG^R_{\mathcal{L}} = (V^R, E^R)$ ein Strukturgraph mit Verwaltungsknoten für Rekursionen, $DG^R_{\mathcal{L}} = (V^R_{DG}, E^R_{DG}, Fun^R_{DG})$ ein Datenflußgraph mit Rekursionen. (Die Mengen E^{vm}_{DG} und E^r_{DG} seien wie in Definition 3.56 definiert. E^v, E^m, E^r wie in Definition 3.35, E^{vm} wie in Definition 3.54)

Behauptung: E^{vm} enthält keine Zyklen.

Beweis: E^{vm} ist zunächst über $E^v \cup E^m$ definiert. $E^v \cup E^m$ enthält nach Definition 3.35 und Satz 3.19 keine Zyklen. Aus dieser Menge werden alle Kanten $e_{v,w}$ entfernt, wenn es w_u mit $w_u.type$ = RECUP, $w_u.name = w.name$ gibt und durch Pfade von v nach w über w_u und, falls es w_d mit $w_d.type$ = RECDOWN, $w_d.name = w.name$ gibt, über w_d ersetzt. Diese Pfade erzeugen offensichtlich keine Zyklen. Damit enthält auch E^{vm} keine Zyklen.

Behauptung: Die Kanten aus E^{vm}_{DG} haben genau nicht in die Richtung der Kanten aus E^{vm}, d.h während die Kanten aus E^{vm} nach unten zu Nachfolgeknoten gehen, gehen die Kanten aus E^{vm}_{DG} in genau umgekehrter Richtung nach oben, oder sie sind Orthogonalkanten.

Beweis: Fall i) bis iv) beziehen sich auf Definition 3.56 2.

i) Fall i) definiert eine Kante von $w \in V^R$ nach $v \in V^R$, wenn es eine Kante in umgekehrter Richtung in E^{vm} gibt.

ii) Fall ii) definiert eine Kante von $v' \in V^R$ nach $w \in V^R$, wobei w Nachfolgeknoten von $v \in V^R$ in E^{vm} ist und v' nicht Nachfolgeknoten von v in E^{vm} ist und nach i) bis iv) eine Kante von v' nach v orthogonal oder in umgekehrter Richtung zu den Kanten in E^{vm} erzeugt wurde. Die Fortsetzung der Kante von v' nach v zu einem Nachfolgeknoten von v geht damit ebenfalls entweder orthogonal, oder in umgekehrter Richtung zu den Kanten in E^{vm}.

iii) Fall iii) definiert eine Kante von $w' \in V^R$ nach $w \in V^R$, wenn beide Nachfolgeknoten von $v \in V^R$ im Strukturgraphen sind: $e_{v,w} \in E^R$ und $e_{v,w'} \in E^R$. Diese Kanten sind in E^{vm} offensichtlich nicht enthalten.

    Es gilt aufgrund der Konsistenzbedingungen für Parametermodi (Def. 3.42) und Satz 3.47, sowie Satz 3.48 auch für die Parameter an inneren Knoten: wenn $e_{w',w} \in E^R_{DG}$, dann existiert keine Kante von $e_{w,w'} \in E^R_{DG}$, bzw. kein Pfad $e_{w,w_1}, ..., e_{w_n,w'} \in E^R_{DG}$, wobei $w_1, ..., w_n \in V^R_{DG}$ Nachfolgeknoten von v sind für $n \in \mathbb{N}$.

iv) Fall iv) definiert eine Kante von v_d nach v in umgekehrter Richtung zu einer Kante in E^{vm}.

Schlußfolgerung: Da durch die Orthogonalkanten (Fall iii) kein Zyklus entsteht, kann durch Hinzunahme von Kanten, die ausschließlich nach oben gehen (Fall i, ii und iv), ebenfalls kein Zyklus entstehen. ◇

Bemerkung 3.60

Die Kanten in E^r_{DG} gehen per Definition in umgekehrter Richtung zu den Kanten aus E^r. Da $E^R = E^{vm} \cup E^r$ Zyklen enthält, enthält nun auch $E^R_{DG} = E^{vm}_{DG} \cup E^r_{DG}$ Zyklen. ◇

Bei Anwendung der Magic Sets Methode auf Programme mit Rekursionen entstehen ähnlich Zyklen für Eingaben und Ausgaben. Dabei können jedoch bereits für eine einfache Rekursion mehrere Schleifen jeden Typs (Ein- oder Ausgabe) entstehen. Das hier vorgestellte Verfahren stellt daher eine Verfeinerung der Rekursionsbehandlung unter Magic Sets dar.

3.4.4.3 Algorithmische Formulierung

In diesem Abschnitt soll die Konstruktion eines Datenflußgraphen mit Zyklen aus einem Strukturgraphen in algorithmischer Form geschlossen angegeben werden.

Aufgrund der Graphstruktur arbeitet der Algorithmus rekursiv. Auf eine detailierte Darstellung der notwendigen Unterprozeduren wurde aus Gründen der besseren Lesbarkeit verzichtet. Ihre Funktionalität sollte aus dem Zusammenhang klar ersichtlich sein.

```
bool has_upperlevel (node v ∈ SG, v' ∈ SG)
    /* returns true if v' has a higher level than v, i.e. a cycle was found */

void create_node (node v ∈ SG, node w ∈ DFG)
    /* Creates a node as in 3.52 1., 3.56 1. */

bool has_inputparams (node v ∈ SG)
    /* true if node v has params with mode input */

node create_recdownnode (node v ∈ SG, node v' ∈ SG)
    /* creates a a RECDN node for v' as subnode of v in the structure graph */

node create_recupnode (node v ∈ SG, node v' ∈ SG)
    /* creates a a RECUP node for v' as subnode of v in the structure graph */

parameter param_of (node v ∈ SG)
    /* returns all parameters of v in a sequence */

parameter corresponding_param (w, p)
    /* returns the parameter of w with the same ident as p*/

void create_param_arc (parameter p, parameter q)
    /* creates an arc from parameter p to parameter q */

node other_subnode (node v ∈ SG, node v' ∈ SG)
    /* returns a subnode of v, that is not v' */

bool is_recdown_node (node v ∈ SG)
    /* returns true if v is of type RECDN */

parameter para_arc_other_endnode (parameter p)
    /* returns all parameters connected with an arc to the parameter p in a sequence */

void create_arcs (node v ∈ SG, w ∈ DFG)
begin
    for all v' = arc_endnode (v) do
    begin
        if has_upperlevel (v, v') then
        begin
            if has_inputparams (v') then
            begin
                new node v'' ∈ DFG = create_recdownnode (v, v');
            end;
            new node v'' ∈ DFG = create_recupnode (v, v');
        end;
    end;
    for all v' = arc_endnode (v) do
    begin
```

```
            create_node (v', w');
            for all p' = param_of (w') do
            begin
                if (p'.type = out) and (exists p = corresponding_param (w, p')) then
                    create_param_arc (p', p);
                else
                begin
                    if exists v'' = other_subnode (v, v') and
                         exists p'' = corresponding_param (v'', p) and (p''.type = out) then
                         create_param_arc (p'', p');
                    if exists p = corresponding_param (v, p') then
                         if is_recdown_node (v) then
                             create_param_arc (p, p');
                         else
                             for all p'' = para_arc_other_endnode (p) do
                                 create_param_arc (p'', p');
                end;
            end;
        end;
        return w;
    end;

begin
    v = root ∈ SG;
    new node w ∈ DFG;
    create_node (v, w);
    create_arcs (v, w);
end.
```

3.4.4.4 Komplexe Formen von Rekursionen

1. **Allgemeine, lineare Rekursion**

   Eine allgemeine, lineare Rekursion kann verschiedene Rekursionszweige enthalten. In jedem Rekursionzweig tritt der rekursive Aufruf jedoch nur genau einmal auf ($n, m \in \mathbb{N}$, $n \geq 1$, $m \geq 1$):

   $R_{t_1} \leftarrow T_1.$
   ...
   $R_{t_n} \leftarrow T_n.$
   $R_{r_1} \leftarrow M_1, D_1, R_1, U_1, N_1.$
   ...
   $R_{r_m} \leftarrow M_m, D_m, R_m, U_m, N_m.$

   Die Implementierung der Rekursion muß hier zusätzlich folgendes beinhalten:

   - Für die Verwaltung sind nach wie vor die Knoten v_d und v_u ausreichend. Alle absteigenden Rekursionsschliefen müssen mit v_d beginnen, alle aufsteigenden mit v_u enden.

   - Für die absteigende Rekursionsschleife übernimmt v_d das Einsammeln von und die Vergabe an die verschiedenen absteigenden Rekursionsschleifen, sowie an die

verschiedenen Terminierungszweige. Für die aufsteigende Rekursionsschleife übernimmt der OR-Knoten, der im Strukturgaphen die verschiedenen Schleifen und Terminierungszweige verbindet, das Einsammeln, der Knoten v_u, der direkt auf den OR- Knoten folgt, die Vergabe an die verschiedenen aufsteigenden Rekursionsschleifen und aus der Rekursion heraus.

- In jeder Rekursionstiefe muß dafür gesorgt werden, daß Werte über genau die Ausgabeschleife gehen, die zu derjenigen Eingabeschleife gehört, über die sie, bzw. über die Eingaben aus denen sie erzeugt wurden, gekommen sind. Es ist bei der absteigenden Rekursionsschleife eine Markierung der Tupel nach Iterationstiefe und Nummer der Eingabeschleife nötig, bei der aufsteigenden Rekursionsschleife eine Weitergabe an die nächste Rekursionsschleife anhand dieser Markierung. Diese Aufgabe müssen die Knoten v_d und v_u übernehmen.
- Die Werte gleicher Iterationstiefe, die über jeweils andere Schleifen in die gleiche Rekursionsschleife gekommen sind, dürfen nicht miteinander verarbeitet werden, da sie nicht zusammengehören. Als Instanz einer Rekursionsschleife müssen daher nun auch Knoten (und ihre Kantenverbindungen) gesehen werden, die zur gleichen Iterationstiefe gehören, aber nur über den gleichen Weg über Rekursionsschleifen erreicht wurden.

2. **Kaskadische Rekursion**

   Bei kaskadischen Rekursionen kann im Rekursionszweig der rekursive Aufruf mehr als einmal vorkommen.

   Kaskadische Rekursionen haben daher folgende Form:

   $$R_t \leftarrow T.$$
   $$R_r \leftarrow M_1, D_1, R, U_1, N_1, ..., M_m, D_m, R, U_m, N_m.$$

   Die Ausgabeschleife, die von U_1 startet, erhält in diesem Fall, insbesondere bei $m = 2$ wieder eine eigene Rekursion. Man erhält also in sich geschachtelte Schleifen, die für sich gesehen aber dem Schema einfacher, linearer Rekursionen entsprechen.

   Die Reihenfolge der einzelnen Schleifen ist zwar eindeutig durch die Kanten vorgegeben, die Markierung für die Rekursionstiefe eines Wertes muß hier allerdings die Möglichkeit bieten, gleichzeitig verschiedene Rekursionstiefen zu beinhalten.

3. **Verschachtelte Rekursion**

   Eine verschachtelte Rekursion ist eine Rekursion über mehrere Regeln:

   $$R_t^1 \leftarrow T^1.$$
   $$R_r^1 \leftarrow M^1, D^1, R^k, U^1.$$
   ...
   $$R_t^k \leftarrow T_k.$$
   $$R_r^k \leftarrow M^k, D^k, R^l, U^k.$$
   ...
   $$R_t^l \leftarrow T_l.$$
   $$R_r^l \leftarrow M^k, D^k, R^1, U^k.$$

   mit $k, l \in \mathbb{N}$, $k, l \leq$ Anzahl der Regeln der verschachtelten Rekursion.

   Bei verschachtelten Rekursionen entsteht ein Aufrufzyklus, der über mehrere Regeln geht. Hier gilt als "rekursiver Aufruf" nur der Aufruf einer Regel des Zyklus, statt aller im Zyklus enthaltenen Aufrufstellen.

Die Regel, die als (innere) Aufrufstelle der Rekursion gelten soll, ist jedoch nicht beliebig wählbar. Wenn die Wahl des "rekursiven Aufrufs" beliebig wäre, könnte es bei ineinandergeschachtelten Zyklen, oder Zyklen, die einen Teil gemeinsam haben, dadurch zu Konflikten kommen, daß in einem Zyklus plötzlich zwei "rekursive Aufrufe" existieren.

Der oberste Knoten einer Schleife im Strukturgraphen ist jedoch auch dann eindeutig, wenn die Schleife über mehrere Knoten geht. Zu diesem werden, wie auch bei einfachen, linearen Rekursionen die Verwaltungsknoten erzeugt und die Kanten entsprechend generiert. Es gibt damit auch bei einer verschachtelten Rekursion, die keine Regel in der Rekursion mit mehr als einem Terminierungszweig und mehr als einem Rekursionszweig besitzt, höchstens eine absteigende und genau eine aufsteigende Rekursionsschleife

4. **Kombinationen**

   Kombinationen der verschiedenen Formen von Rekursionen, wie z.B. eine verschachtelte, allgemeine lineare Rekursion lassen sich entsprechend durch eine Kombination der verschiedenen Methoden realisieren.

Bemerkung:

> Der Aufbau des Graphen ist, ohne Berücksichtigung zusätzlicher Knotenfunktionalitäten, bei allen Formen von Rekursionen analog zur einfachen, linearen Rekursion. Es können nun jedoch an RecDown- oder RecUp-Knoten, sowie am obersten Or-Knoten der Rekursion Kanten enden oder beginnen, die zu verschiedenen Zyklen gehören.
>
> Die Funktionalität der Knoten v_d und v_u zum Sichern und Restaurieren des jeweiligen Kontextes muß an die Art der Rekursion und an die Anzahl der vorhandenen Schleifen angepaßt sein. ◇

3.4.4.5 Erhaltung der Stratifizierung bei Rekursionen

Im Unterschied zur Behandlung von Programmen mit der Magic Sets Methode kann es bei dem vorliegenden Modell nicht sein, daß zunächst stratifizierte Programme anschließend nicht mehr stratifiziert sind.

Als Beispiel soll folgende, rein schematisch dargestellte Rekursion dienen:

$F \leftarrow R.$
$R \leftarrow T.$
$R \leftarrow \neg M, D, R, U.$
$T \leftarrow T'.$
$M \leftarrow M'.$
$D \leftarrow D'.$
$U \leftarrow U'.$
$\leftarrow F.$

Sei nun M ein negiertes Literal. Offensichtlich ist das Programm stratifiziert, da M nicht von sich selbst abhängt.

Die gleiche Rekursion hat mit Magic Sets transformiert folgendes Aussehen, wobei vereinfachend von der Annahme ausgegangen wird, daß jedes Literal Eingaben erhält, insbesondere auch F über die Anfrage.

$F \leftarrow mF, R.$
$R \leftarrow mR, T.$
$R \leftarrow mR, \neg M, D, R, U.$
$mR \leftarrow mF.$
$mR \leftarrow mR, \neg M, D.$
$mT \leftarrow mR.$
$mM \leftarrow mR.$
$mD \leftarrow mR, \neg M.$
$mU \leftarrow mR, \neg M, D, R.$
$T \leftarrow mT, T'.$
$M \leftarrow mM, M'.$
$D \leftarrow mD, D'.$
$U \leftarrow mU, U'.$
$mF.$
$\leftarrow F.$

Nun hängt M von sich selbst ab, d.h. M von $\neg M$ und umgekehrt. Das Programm ist also nicht länger stratifizert.

Das Problem nicht (lokal) stratifizierter Programme ist, daß für sie kein eindeutiges minimales semantisches Modell garantiert ist. Auch vom logischen Standpunkt aus ist die Aussage "wenn $\neg f$ gilt, gilt f" offensichtlich ein Widerspruch.

Der Unterschied zwischen den Graphen ist jedoch, daß der Dependency Graph bei der Magic Sets Methode rein bottom-up berechnet wird, d.h. M kann nur berechnet werden, wenn $\neg M$ berechnet wird.

Im vorliegenden Berechnungsmodell erhalten NOT-Knoten dagegen ausschließlich Eingaben. Durch den Algorithmus zur Analyse der Parametermodi wird für alle Parameter des NOT-Knotens der Modus auf Eingabe gesetzt. Die Konsistenzbedingung der bedingten Sicherheit gewährleistet darüberhinaus die Existenz von Eingaben.

Die Aufgabe des Teilgraphen, dessen Wurzel der NOT-Knoten ist, ist damit, zu eingegebenen Werten einen Wahrheitswert zu berechnen. Er erzeugt jedoch keine Ausgaben, die als Werte in der nächsten Iterationsschleife zu Eingaben des NOT-Knotens werden.

Der Teilgraph selbst wird durch das in dieser Arbeit vorgestellte Modell zur Behandlung von Zyklen nicht verändert. Innerhalb des Teilgraphens kann daher die Berechnung problemlos bottom-up stattfinden.

3.4.5 Optimierungen

Nach der Erzeugung des vollständigen Datenflußgraphen bestehen verschiedene Möglichkeiten zur Optimierung:

1. Sind Teilgraphen von ihrer Wurzel bis einschließlich der Blätter identisch, so können sie zusammengefaßt werden, d.h. die Berechnung der Knoten muß nur einmal erfolgen.

   Diese Situation kann auftreten, wenn ein Teilgraph vor der Analyse der Parametermodi aufgrund von Mehrfachreferenzierung vervielfältigt wurde, die Analyse dann aber identische Parametermodi festgestellt hat. Die vom Teilgraphen berechneten Ausgaben müssen dann an verschiedene Knoten geschickt werden.

   Dies reduziert die Anzahl der notwendigen Berechnungen für den Graphen.

2. Folgende Kanten sind nicht notwendig und können daher entfallen:

   - Die Resultatwerte müssen nicht an jeden Knoten weitergegeben werden, sondern nur an diejenigen Knoten, die auch Parameterwerte vom aktuellen Knoten erhalten. Nur wenn der Knoten selbst keine Parameter besitzt, muß der Ergebniswert in jedem Fall zumindest an den AND-Knoten weitergegeben werden, der im Strukturgraph der Vorgängerknoten des betrachteten Knotens ist.
   - Die Kanten von Parametern mit Modus Ausgabe, die nur zu einem lokalen Parameter des Vorgängerknotens führen, können entfallen, da die Werte lokaler Parameter nicht weiterverarbeitet werden. Der an AND-Knoten notwendige Join, der prinzipiell nicht nur für die Werte der normalen Parameter, sondern auch für die Werte lokaler Parameter ausgeführt werden muß, wird an den betreffenden Verwendungsstellen durchgeführt werden, da von allen Ausgabestellen Kanten zu einer Eingabestelle führen (siehe Abschnitt 3.4).
   - Die Kanten die zu Parametern mit Modus Eingabe an inneren Knoten gehen, können entfallen, wenn es sich bei diesen Parametern nicht um die eigentliche Verwendungsstelle handelt. An inneren Knoten kann es genaugenommen nur in zwei Situationen eine Verwendungsstelle geben: bei einer Konstante eines AND-Knotens mit Modus Eingabe wird der eingehende Wert mit dieser Konstante verglichen und bei einem NOT-Knoten wird grundsätzlich die Ausgabe mit der Eingabe verglichen. In allen anderen Fällen werden Werte an Parameter mit Modus Eingabe nicht benötigt.

   Die Reduzierung der Kanten ergibt eine Verringerung des Kommunikationsaufwands, beeinflußt aber nicht die Menge an notwendigen Berechnungen.

Bemerkung: In dem in Kapitel 3.5 folgenden Beispiel werden die oben genannten Optimierungen implizit angenommen.

3.4.6 Zusammenfassung

Der im Rahmen dieser Arbeit erstellte Datenflußgraph unterscheidet sich in wesentlichen Punkten von Datenflußmodellen aus der Literatur, wie sie z.B. in [Arvind 78, Dennis 80, Davis 82] vorgestellt werden.

Ein Vergleich ergibt:

Datenflußgraphen in der Literatur	Datenflußgraph in dieser Arbeit
Ein Knoten hat genau zwei Eingabeparameter.	Ein Knoten kann mehr als zwei Eingabeparameter haben.
Ein Knoten hat genau einen Ausgabeparameter.	Ein Knoten kann mehr als einen Ausgabeparameter haben.
Für jeden Eingabeparameter eines Knotens kommt genau ein Wert an.	Für jeden Eingabeparameter eines Knotens kann mehr als ein Wert ankommen.
Es gibt genau einen Ausgabewert für den Ausgabeparameter eines Knotens.	Es kann mehr als einen Ausgabewert für einen Ausgabeparameter eines Knotens geben.

Datenflußgraphen in der Literatur	Datenflußgraph in dieser Arbeit
Nur ein Copy-Knoten kann einen Wert vervielfältigen und an mehrere Nachfolgeknoten schicken.	Jeder Ausgabewert eines Ausgabeparameters kann an mehrere Nachfolgeknoten gehen.
Es gibt nur einen Ausgabeparameter.	Die Ausgaben verschiedener Ausgabeparameter können an verschiedene Nachfolgeknoten gehen.

Konsequenz:

Zur Ausführung des in dieser Arbeit erstellten Datenflußgraphen muß ein neues Berechnungsmodell erstellt werden.

3.5 Ein Beispiel

Im folgenden soll anhand eines Beispiels die Arbeitsweise des bisher vorgestellten Algorithmus und das Ergebnis, d.h. der aus den Regeln eines $\text{DATALOG}_f^{\neg}$ -Programms erzeugte Datenflußgraph, veranschaulicht werden.

Die folgenden Regeln können in der vorliegenden Form Teil eines Steuerungssystems für autonome Straßenfahrzeuge sein.

Mit ihrer Hilfe soll in diesem Beispiel das Straßenfahrzeug in der Lage sein, aufgrund des Straßentyps und der Verkehrssituation zu entscheiden, ob ein Überholvorgang nötig ist und wenn ja, ob er durchgeführt werden kann, ohne die Sicherheit des Fahrzeugs zu gefährden.

3.5.1 Ein Expertensystem zur Steuerung eines Überholvorgangs

Das Steuerungssystem ist folgendermaßen spezifiziert:

Ein Überholvorgang wird gestartet (start_overtake), wenn er gewünscht wird (overtake_request). Dazu muß zunächst mehr über die Straße bekannt sein (get_road_param), z.B. die Anzahl der Spuren. Es darf kein Überholverbot bestehen (not no_passing) und das Überholmanöver muß prinzipiell vom Fahrer freigegeben sein (passing_ok).

> start_overtake (*RoadType*) ←
> get_road_param (*RoadType*,*Lanes*,*FreeLanes*),
> overtake_request (*Lanes*,*FreeLanes*),
> not (no_passing ()),
> passing_ok (*true*).

Ein Überholvorgang sollte erfolgen (overtake_request), wenn er möglich (change_lane_possible) und wenn er notwendig ist (change_lane_required).

> overtake_request (*Lanes*,*FreeLanes*) ←
> change_lane_possible (*Lanes*,*FreeLanes*,*OwnLane*),
> change_lane_required (*OwnLane*).

Ein Überholvorgang ist in zwei verschiedenen Fällen möglich (change_lane_possible):

- wenn es überhaupt eine zweite Fahrspur gibt ($Lanes \geq 2$) und wenn die Spur neben der eigenen Fahrspur (get_own_lane) frei ist, weil sie in die gleiche Fahrtrichtung geht ($OwnLane < FreeLanes$).
- wenn es überhaupt eine zweite Fahrspur gibt ($Lanes \geq 2$) und wenn die Spur neben der eigenen Fahrspur (get_own_lane) ($OwnLane = FreeLanes$) frei ist, weil kein Gegenverkehr kommt (free_left_lane).

> change_lane_possible (*Lanes*,*FreeLanes*,*OwnLane*) ←
> $Lanes \geq 2$,
> get_own_lane (*OwnLane*),
> $OwnLane < FreeLanes$.

> change_lane_possible (*Lanes*,*FreeLanes*,*OwnLane*) ←
> $Lanes \geq 2$, $Lanes \leq 4$,
> get_own_lane (*OwnLane*),

OwnLane=*FreeLanes*,
free_left_lane (*OwnLane*).

Ein Überholvorgang wird erforderlich (change_lane_required), wenn die mögliche Geschwindigkeit auf der linken Fahrspur (get_velocities) und außerdem die gewünschte eigene Fahrtgeschwindigkeit (get_max_velocity) deutlich schneller ist, als die Geschwindigkeit vorausfahrender Fahrzeuge auf der eigenen Fahrspur (*VelLeft*> *VelFront*+25 und *VelMax*> *VelFront*+25). Gibt es keine Fahrzeuge auf der jeweiligen Fahrspur, so sollten die Werte auf einen Maximalwert gesetzt sein.

change_lane_required (*OwnLane*) ←
get_velocities (*OwnLane*, *VelFront*, *VelLeft*)
get_max_velocity (*VelMax*),
VelLeft> *VelFront*+25,
VelMax> *VelFront*+25.

Die Überprüfung der Fahrspur (free_left_lane) ist erfolgreich wenn die Spur neben der eigenen Fahrspur (*OwnLane*= 1, bzw. *OwnLane*= 2) frei ist (detect_obstacle_lane2 oder detect_obstacle_lane3).

free_left_lane (*OwnLane*) ←
OwnLane= 1,
detect_obstacle_lane2 (*false*).

free_left_lane (*OwnLane*) ←
OwnLane= 2,
detect_obstacle_lane3 (*false*).

Die Geschwindigkeit eines vorausfahrenden Fahrzeugs und eines auf der linken Spur fahrenden Fahrzeugs (get_velocities) ergibt sich aus der Geschwindigkeit der Fahrzeuge auf den einzelnen Spuren (get_lane_velocities) und der Position der eigenen Fahrspur (*OwnLane*= 1, bzw. *OwnLane*= 2).

get_velocities (*OwnLane*, *VelLane1*, *VelLane2*) ←
OwnLane= 1,
get_lane_velocities (*VelLane1*, *VelLane2*, *VelLane3*).

get_velocities (*OwnLane*, *VelLane2*, *VelLane3*) ←
OwnLane= 2,
get_lane_velocities (*VelLane1*, *VelLane2*, *VelLane3*).

Ein Überholverbot (no_passing) besteht, wenn ein Überholverbotsschild aktuell ist (detect_pass_sign).

no_passing () ←
detect_pass_sign (*true*).

Die maximale Geschwindigkeit ergibt sich

- wenn vorhanden (detect_velocity_sign), aus dem zuletzt gelesen Geschwindigkeitsschild (get_allowed_velocity), oder,
- falls keines vorhanden ist (detect_velocity_sign), aus der aktuellen Umgebung wie Straße in der Stadt, Landstraße oder Autobahn (get_environment).

get_max_velocity (*Vel*) ←
detect_velocity_sign (*true*),
get_allowed_velocity (*Vel*).

get_max_velocity (50) ←
detect_velocity_sign (*false*),
get_environment ("*city*").

get_max_velocity (100) ←
detect_velocity_sign (*false*),
get_environment ("*country*"),

get_max_velocity (130) ←
detect_velocity_sign (*false*),
get_environment ("*highway*").

Zusätzlich benötigt man noch Informationen über Art und Parameter der Straße (get_-road_param).

get_road_param ($highway^c$,6^c,3^c)^o.
get_road_param ($road^c$,4^c,2^c)^o.
get_road_param ($street^c$,2^c,1^c)^o.
get_road_param ($lane^c$,1^c,1^c)^o.

Die Schnittstelle zum Datenspeicher wird durch folgende Fakten realisiert:

passing_ok (*Pass_OK*).
get_max_velocity (*VelMax*).
detect_pass_sign (*PassSign*).
detect_velocity_sign (*VelSign*).
get_allowed_velocity (*VelAllowed*).
get_own_lane (*OwnLane*).
get_environment (*Environment*).
detect_obstacle_lane2 (*Obst*).
detect_obstacle_lane3 (*Obst*).
get_lane_velocities (*VelLane1, VelLane2, VelLane3*).

Die wichtigsten verwendeten Parameter haben folgende Bedeutung:

- *RoadType* ist ein Parameter für den Typ der Straße.
- *Lanes* steht für die Anzahl der Spuren einer Straße, *FreeLanes* für die Anzahl der Spuren einer Straße, auf denen der Verkehr in die Fahrtrichtung geht.
- *OwnLane* für die Nummer der eigenen Spur.
- *VelFront* steht für die Geschwindigkeit vorausfahrender Fahrzeuge auf der eigenen Fahrspur, *VelLeft* für die auf der linken Fahrspur.
- *VelMax* ist die eigene maximale Fahrgeschwindigkeit und *VelMax* für die akutell erlaubte Geschwindigkeit.

3.5.2 Transformationsschritte

Bemerkung:

- Der Identifikator, bzw. die Liste von Identifikatoren, ist in eckigen Klammern angegeben.
- In den Regeln bezeichnet "^{i}" daß es sich um einen Parameter vom Typ Eingabe handelt, "^{o}"daß es sich um einen Parameter vom Typ Ausgabe handelt. Ein "*c*"kennzeichnet eine Konstante. Je nach Verwendung im Graph kann daraus ein Eingabe- oder ein Ausgabeparameter werden. Das "^{o}" am Ende jedes Literals steht für den Resultatwert, der immer vom Typ Ausgabe ist.

Die Schritte

- Extraktion von Operationen, Funktionen und Konstanten aus der Parameterliste
- Vergabe der Identifikatoren, sowie
- Analyse der Parametermodi,

ergeben folgende Situation (am Programm dargestellt):

start_overtake $(RoadType[1]^i)^o \leftarrow$
  get_road_param $(RoadType[1]^i, Lanes[2]^o, FreeLanes[3]^o)^o$,
  overtake_request $(Lanes[2]^i, FreeLanes[3]^i)^o$,
  not (no_passing $())^o$,
  $Const[4]^o = true[5]^c$, passing_ok $(Const[4]^o)^o$.

overtake_request $(Lanes[2]^i, FreeLanes[3]^i)^o \leftarrow$
  change_lane_possible $(Lanes[2]^i, FreeLanes[3]^i, OwnLane[6]^o)^o$,
  change_lane_required $(OwnLane[6]^i)^o$.

change_lane_possible $(Lanes[2]^i, FreeLanes[3]^i, OwnLane[6]^o)^o \leftarrow$
  $Lanes[2]^i \geq 2[7]^{c\ o}$,
  get_own_lane $(OwnLane[6]^o)^o$,
  $OwnLane[6]^i < FreeLanes[3]^{i\ o}$.

change_lane_possible $(Lanes[2]^i, FreeLanes[3]^i, OwnLane[6]^o)^o \leftarrow$
  $Lanes[2]^i \geq 2[8]^{c\ o}$, $Lanes[2]^i \leq 4[9]^{c\ o}$,
  get_own_lane $(OwnLane[6]^o)^o$,
  $OwnLane[6]^i = FreeLanes[3]^{i\ o}$,
  free_left_lane $(OwnLane[6]^i)^o$.

change_lane_required $(OwnLane[6]^i)^o \leftarrow$
  get_velocities $(OwnLane[6]^i, VelFront[10]^o, VelLeft[11]^o)^o$
  get_max_velocity $(VelMax[12]^o)^o$,
  $VelLeft[11]^i > VelFront[10]^i + 25[13]^{c\ o}$,
  $VelMax[12]^i > VelFront[10]^i + 25[14]^{c\ o}$.

free_left_lane $(OwnLane[6]^i)^o \leftarrow$
  $OwnLane[6]^i = 1[15]^{c\ o}$.
  $Const[16]^o = false[17]^{c\ o}$, detect_obstacle_lane2 $(Const[16]^o)^o$.

free_left_lane $(OwnLane[6]^i)^o \leftarrow$
  $OwnLane[6]^i = 2[18]^{c\ o}$.
  $Const[19]^o = false[20]^{c\ o}$, detect_obstacle_lane3 $(Const[19]^o)^o$.

get_velocities $(OwnLane[6]^i, VelLane1[10]^o, VelLane2[11]^o) \leftarrow$
  $OwnLane[6]^o = 1[21]^{c\ o}$,
  get_lane_velocities $(VelLane1[10]^o, VelLane2[11]^o, VelLane3[22]^o)$.

get_velocities $(OwnLane, VelLane2[10]^o, VelLane3[11]^o) \leftarrow$
  $OwnLane[6]^o = 2[23]^{c\ o}$,
  get_lane_velocities $(VelLane1[24]^o, VelLane2[10]^o, VelLane3[11]^o)$.

no_passing $()^o \leftarrow$
  $Const[25]^o = true[26]^{c\ o}$, detect_pass_sign $(true[25]^c)^o$.

get_max_velocity $(Vel[12]^o)^o \leftarrow$
  $Const[27]^o = true[28]^{c\ o}$, detect_velocity_sign $(Const[27]^o)^o$,
  get_allowed_velocity $(Vel[12]^o)^o$.

get_max_velocity $(50[12])^o \leftarrow$
$Const[29]^o = false[30]^{c\ o}$, detect_velocity_sign $(Const[29]^o)^o$,
$Env[31]^o = ''city''[32]^{c\ o}$, get_environment $(Env[31]^o)^o$.

get_max_velocity $(100[12])^o \leftarrow$
$Const[33]^o = false[34]^{c\ o}$, detect_velocity_sign $(Const[33]^o)^o$,
$Env[35]^o = ''country''[36]^{c\ o}$, get_environment $(Env[35]^o)^o$.

get_max_velocity $(130[12])\ ^o \leftarrow$
$Const[37]^o = false[38]^{c\ o}$, detect_velocity_sign $(Const[37]^o)^o$,
$Env[39]^o = ''highway''[40]^{c\ o}$, get_environment $(Env[39]^o)^o$.

get_road_param $([1],[2],[3])^o$.
get_road_param $([1],[2],[3])^o$.
get_road_param $([1],[2],[3])^o$.
get_road_param $([1],[2],[3])^o$.

passing_ok $(Pass_OK[4]^o)^o$.
get_max_velocity $(VelMax^o)^o$.
detect_pass_sign $(PassSign[19]^o)^o)$.
detect_velocity_sign $(VelSign[27|29|33|37]^o)^o$.
get_allowed_velocity $(VelAllowed[12]^o)^o$.
get_own_lane $(OwnLane^o[6|6])^o$.
get_environment $(Environment[27|35|39]^o)^o$.
detect_obstacle_lane2 $(Obst[16]^o)^o$.
detect_obstacle_lane3 $(Obst[16]^o)^o$.
get_lane_velocities $(VelLane1[10|24]^o, VelLane2[11|10]^o, VelLane3[22|11]^o)$.

3.5.3 Der Datenflußgraph

Abbildung 36 zeigt den aus dem Beispiel resultierenden Datenflußgraphen.

Bemerkung:

- Für parallele Kanten, d.h. Kanten, die die gleichen Knoten miteinander verbinden, wird nur eine Kante eingezeichnet mit einer Beschriftung für beide Parameter. Auch die Kanten für die Resultatwerte werden nicht zusätzlich eingezeichnet, wenn es bereits eine andere Kante an dieser Stelle gibt.
- Die grau ausgefüllten Knoten stellen Endpunkte, bzw. Blätter des Graphen dar.

Aus dem Charakter des Anwendungsgebietes ergibt sich eine typische Form des folgenden Beispiels:

- Negation ist ein häufig vorkommendes Prinzip, wobei sie sich zumeist auf die Negation von Literalen beschränkt, die keine Parameter erhalten. Damit beschränkt sich die Negation auf die Komplementierung des boolschen Ergebniswertes.

- Rekursionen treten im allgemeinen nicht auf.

  Rekursionen werden prinzipiell für folgende Aufgaben eingesetzt:

  - Konstruktion und Verarbeitung von Listen
    Zusammengesetzte Strukturen wie Listen können zwar auf Datenflußrechnern verarbeitet werden, kommen aber im Rahmen des Anwendungsgebietes nicht vor, da sich die Eingaben von der extensionalen Datenbank auf konkrete Einzelwerte beschränkt.

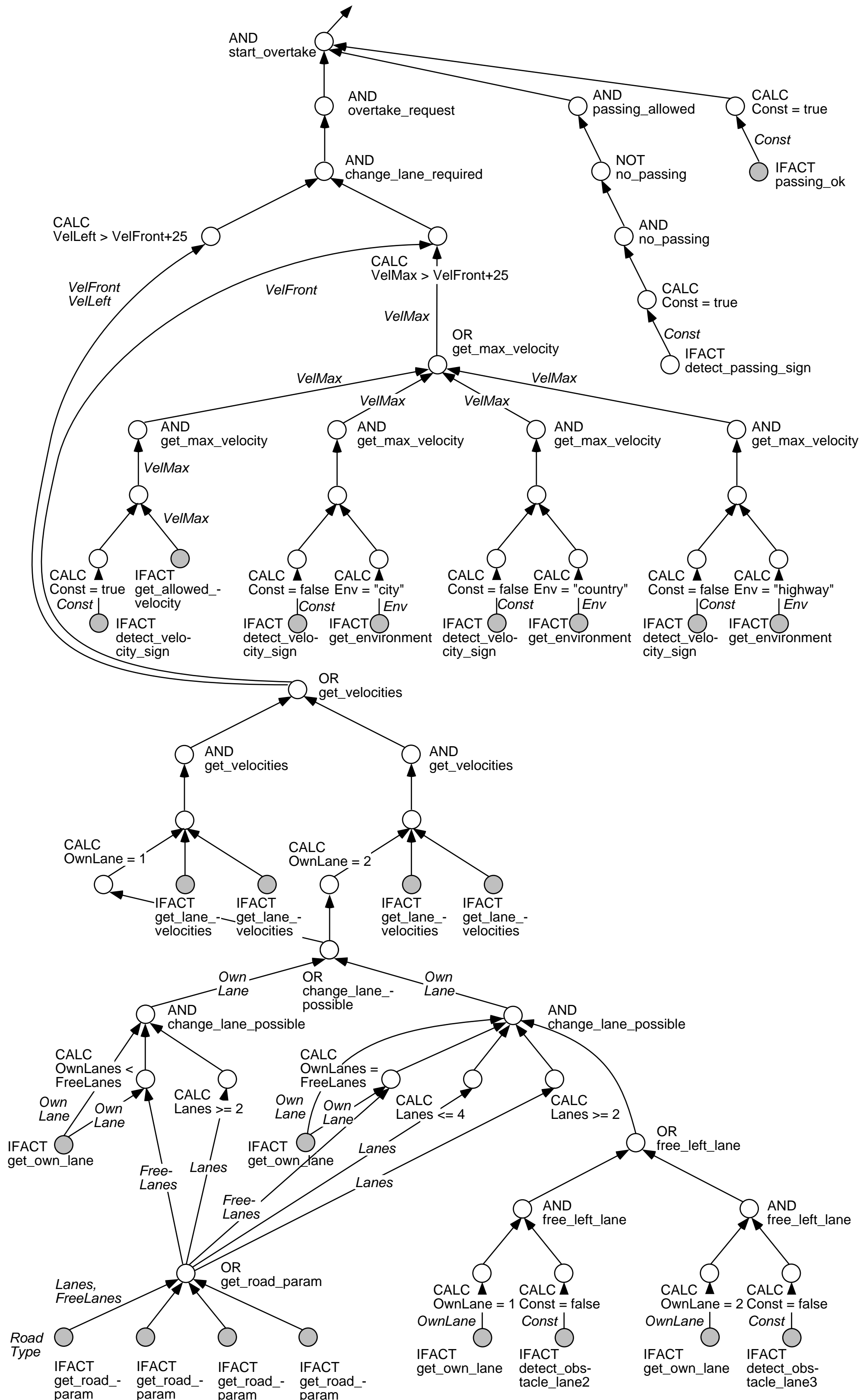


Abbildung 36: Datenflußgraph zum Beispiel

- Berechnung der transitiven Hülle von Relationen
  Im Anwendungsfall geht es speziell um die Berechnung von Ausgabewerten, nicht aber um die Bildung von Relationen über Werten durch Bildung einer transitiven Hülle.

Die Weitergabe der Werte erfolgt im Datenflußgraphen von unten nach oben. Sie werden dabei von IFacts aus der Datenbank ausgegeben, oder durch Facts spezifiziert. Die Ausgabe des Ergebnisses erfolgt über den Query, der in diesem Beispiel durch den Knoten "AND start_overtake" repräsentiert wird.

Die Ausführungssteuerung kann in diesem Graphen rein datengetrieben erfolgen. Eine Eingabe von Daten in umgekehrter Richtung (von oben nach unten) ist ebensowenig notwendig, wie das Verschicken von Anforderungen, um Berechnungen anzustoßen.

3.6 Funktionalität der Knoten

Dieser Abschnitt befaßt sich mit der Spezifikation einer Knotenfunktionalität entsprechend dem Berechnungsmodell von Logikprogrammen.

Zunächst soll die Funktionalität informell beschrieben werden. Als Tupel wird dabei eine Menge von (durch eine Relation) zusammengehörigen Werten verstanden, d.h. Werte, die gemeinsam als Ausgabe erzeugt werden, bzw. als Eingabe verwendet werden.

1. IFACT

   Jedes IFACT addressiert über sein Prädikat einen Bereich des Datenspeichers. In diesem Bereich stehen die Werte, die über das IFACT ausgegeben werden sollen. Jedem Parameter ist dabei genau ein Wert zuzuordnen. Mehrfache Ausgaben, d.h. mehrere Wertetupel, werden an einem IFACT im Rahmen dieses Modells nicht ausgegeben, da es im Anwendungsfall für jeden Parameter zu einem Zeitpunkt grundsätzlich exakt einen Wert gibt.

   Der Wert des Ergebnisparameters ist bei jeder Ausgabe grundsätzlich *success*.

   Die Werte werden zu Beginn jeder Graphauswertung aus dem Datenspeicher eingelesen. Sie können sich während der Graphauswertung unabhängig vom Programmablauf verändern, z.B. durch Ausgaben anderer (parallel ablaufender) Programme, wie z.B. der Sensordatenanalyse. Auch Ergebnisse der Graphauswertung können in den Datenspeicher zurückgeschrieben werden. Änderungen der Werte im Datenspeicher wirken sich jedoch erst auf die nächste Graphauswertung aus.

2. FACT

   Ein FACT kann laut Spezifikation aus Kapitel 3.1.2 in seiner Parameterliste entweder Konstanten enthalten, oder Variablen, die genau zweimal vorkommen.

   - Bei einer Konstante mit Modus in werden alle Eingaben mit dieser Konstanten verglichen. Das Ergebnis des Vergleichs wird über den Ergebnisparameter in Form eines boolschen Wertes ausgegeben.
   - Bei einer Konstante mit Modus out wird der Wert der Konstanten ausgegeben. Der Wert des Ergebnisparameters ist hierbei grundsätzlich *success*.
   - Bei einer Variablen mit Modus in wird der über der zugehörigen Variablen eingegebene Wert ausgegeben. Der Wert des Ergebnisparameters ist hierbei grundsätzlich *success*.

   Eine Kombination obiger Fälle ist ebenfalls möglich. Die einzelnen Ausgaben für den Ergebnisparameter ergeben dann Und-verknüpft den eigentlichen Ergebniswert.

   Gibt es Parameter mit Eingabekanten von verschiedenen Vorgängerknoten, bzw. Parameter, die Eingabekanten von verschiedenen Vorgängerknoten besitzen, muß zunächst ein Join über alle Eingabetupel durchgeführt werden.

3. FUNC

   Der FUNC-Knoten realisiert ein Systemprädikat. Die Ausgabe des FUNC-Knotens kann entweder ein Wert, eine Menge von Werten, oder der boolscher Ergebniswert der Funktionsausführung sein.

   Gibt es verschiedene Vorgängerknoten, so müssen die eingehenden Werte gejoint werden. Entstehen dabei mehrere Eingabetupel, erfolgt der Funktionsaufruf für jedes Eingabetupel.

   Hat der FUNC-Knoten einen boolschen Funktionswert, wird dies über den Ergebniswert des Knotens für jeden Funktionsaufruf weitergegeben (*success* oder *failure*), ansonsten ist der Ergebniswert immer *success*.

4. NOT

   Der NOT-Knoten bildet die Differenzmenge der Eingaben, indem er die Menge der Eingaben von Unterknoten von der Menge der Eingaben, die er von anderen Knoten (nicht Unterknoten) erhält, abzieht. Eine äquivalente Betrachtungsweise ist, daß er den Eingaben, die er von anderen Knoten erhält, Wahrheitswerte zuordnet: zulässig oder nicht zulässig.

   - Ein boolscher Ergebniswert wird negiert weitergegeben. Das Ergebnis ist somit ein Wahrheitswert (*success* oder *failure*).
   - Es wird eine Differenzmenge aus der Menge der Eingabewerte und der Menge Ausgabewerte des zu negierenden Knotens gebildet. Das Ergebnis ist hier somit eine Aussage über die Zulässigkeit der Wertetupel aus der Menge der Eingabewerte.

5. AND

   Der AND-Knoten führt einen Join über die einkommenden Werte aus. Er bildet dafür zunächst ein Kreuzprodukt über alle ankommenden Werte und wählt aus den entstandenen Tupel diejenigen aus, die gleiche Werte an den Stellen von Parametern mit gleichen Namen haben.

6. OR

   Der OR-Knoten gibt alle Wertetupel, die er von Unterknoten erhält, als Menge von Tupeln weiter. Er realisiert damit eine Vereinigung.

   Eingaben von Knoten, die keine Unterknoten sind, werden ignoriert (siehe auch Abschnitt 3.4.5), da sich die eigentliche Verwendungsstelle für diese Eingaben grunsätzlich an den Nachfolgeknoten oder an Knoten in den von den Nachfolgeknoten aufgespannten Teilgraphen befindet. Dies gilt insbesondere auch durch das explizite Setzen des Modus Eingabe an Nachfolgeknoten, wenn ein Parameter am OR-Knoten den Modus Eingabe hat, der korrespondierende Parameter am Nachfolgenoten jedoch zunächst den Modus Ausgabe (siehe Def. 3.39).

7. QUERY

   Der QUERY hat die gleiche Funktionalität wie der AND-Knoten. Er übernimmt zusätzlich lediglich die Ausgabe der Ergebniswerte.

8. RECDOWN

   Der Knoten REC_d kann Werte von verschiedenen Knoten erhalten. Er muß daher einen Join über diese Werte durchführen, um sie gemeinsam weitergeben zu können. Sie erhalten zusätzlich eine Markierung, die angibt, in welcher Rekursionsschleife und Rekursionstiefe sie sich befinden und wie die vorherige Markierung, falls vorhanden, ausgesehen hat.

9. RECUP

   Der Knoten REC_u erhält nach Konstruktion des Datenflußgraphen mit Rekursionen Werte von genau einem OR-Knoten. Daher ist ein Join über die eingehenden Werte nicht notwendig und er kann die Werte, die er erhält, unverändert weitergeben. Er entscheidet jedoch anhand der aktuellen Markierung, an welche Rekursionsschleife oder Ausgabe sie gehen sollen. Zusätzlich restauriert er aus der aktuellen Markierung die vorherige.

Zusätzlich gilt allgemein:

- Alle Knoten haben Ergebnisparameter. Über sie wird das Ergebnis der Knotenausführung im Sinne von "erfolgreich" und "nicht erfolgreich" ausgegeben.

  Die Eingabe-Ergebnisparameter können die Ergebnisse von Unterknoten annehmen. Diese werden wie alle anderen Werte durch die Knotenfunktionalität verarbeitet. Die Ausgabe-Ergebnisparameter geben jeweils den Status der Knotenoperation, erfolgreich (*success*), oder gescheitert (*failure*), an.

- Alle Knoten mit Eingabeparametern von verschiedenen Knoten müssen zunächst einen Join über diesen Eingaben durchführen.

  Untersucht man die Auswertungsmethode eines Top-Down Verfahrens das mit Backtracking arbeitet, stellt man fest, daß bei einem Literal (z.B. einem Fakt), das von verschiedenen anderen Literalen mehr als eine Eingabe für disjunkte Parameter erhält, zumindest ein Kreuzprodukt über die Eingaben gebildet wird, da durch Backtracking (falls es notwendig ist) immer wieder andere Kombinationen von Eingaben gebildet werden. Da der Join, also eine Selektion von Elementen aus dem Kreuzprodukt nach bestimmten Kriterien, für die Ausgaben von verschiedenen Literalen ohnehin an einem die Literale verbindenden Regelkopf durchgeführt werden muß, lassen sich damit überflüssige Eingaben und damit überflüssige Berechnungen sparen.

  Die Korrektheit dieses Verfahrens wird in Abschnitt 3.7 beweisen.

- Da an den Knoten vielfach verschiedene Wertemengen ankommen können, die dann entsprechend miteinander verarbeitet werden müssen, z.B. durch einen Join muß durch das Berechnungsmodell sichergestellt werden, daß alle Wertemengen am Knoten lokal gespeichert werden, bis die Ausführung der Knotenoperation vollständig, d.h. für alle Wertemengen erfolgt ist (siehe Kapitel 4.1.2. Nur so kann gewährleistet werden, daß die Ergebnisse der Knotenoperationen vollständig sind.

Um die Funktionalität der einzelnen Knoten exakt spezifizieren zu können, wird die auch bei Bottom-Up Verfahren übliche Ausdrucksfrom der relationalen Algebra benutzt. So läßt sich die Funktion eines AND-Knotens als *(Equi-)Join* ($\bowtie_=$), die eines OR-Knotens als *Vereinigung* ($\cup$) im Sinne der relationalen Algebra darstellen.

Durch die IFACT-Knoten, die die Schnittstelle zum Datenspeicher darstellen (IFACT), werden Relationen spezifiziert. Die ausgegebenen Werte sind dadurch zu Tupeln zusammengefaßt. Lediglich die Funktionalität von FUNC-Knoten läßt sich nicht mit Methoden der relationalen Algebra ausdrücken. An dieser Stelle soll eine Metadarstellung genügen.

Definition 3.61 *Tupelmengen*

Sei $\mathcal{D}$ eine (beliebige) Domain.

Sei $M_i = \{(m^r_{i,1}, ..., m^r_{i,k_i}) \mid k_i, u_i \in \mathbb{N}, r \leq u_i\}$, für $i \in \mathbb{N}$, eine Menge von Tupeln mit $m^r_{i,1}, ..., m^r_{i,k_i} \in \mathcal{D}$.

Sei $\mathcal{M} = \{M_i \mid i \in \mathbb{N}\}$ die Menge der Tupelmengen.

Für die Tupel einer Menge M_i gilt, daß alle Tupel die gleiche Struktur besitzen:

- Alle Tupel haben die gleiche Anzahl von Elementen.
- Alle Tupel haben die gleichen Attribute.
- Ein Attribut kommt nicht mehr als einmal in einem Tupel vor.

Die Zusammengehörigkeit zwischen einzelnen Werten in Tupeln sei durch eine abstrakte Relation $\approx$ gegeben. ◇

Definition 3.62 *Funktionalität der Knoten*

$DG^R_{\mathcal{L}} = (V^R_{DG}, E^R_{DG}, Fun^R_{DG})$ sei ein Datenflußgraph mit Zyklen zu einem Strukturgraphen $SG_{\mathcal{L}} = (V, E)$. Es gelten die Bezeichnungen aus Definition 3.42. Die Verwendung der Operatoren $\bowtie, \cup$ ist analog zu ihrer Definition in relationaler Algebra.

- $C = \{(c_1, .., c_{v.const})\}$ sei die Relation die nur aus dem Tupel mit den Konstanten des Knotens v besteht.
- M (bzw. $M_1, M_2, ...$) sind Eingabe-Tupelmengen. Die Attribute für die Tupel seien über die Namen der Eingabeparameter am Knoten gegeben. Zwei Elemente haben das gleiche Attribut, wenn sie Eingaben für den gleichen Parameter darstellen.
- f ist ein Funktionssymbol.
- $\bowtie_=$ ist der Equijoin zweier Tupelmengen über gemeinsame Attribute. Er wird auch als Naturaljoin bezeichnet.
- $\cup$ ist die Vereinigung zweier Mengen.
- $\Pi_{v.out}$ ist die Projektion einer Tupelmenge auf die Ausgabeparameter des Knotens.
- $\Pi_{v.in}$ ist die Projektion einer Tupelmenge auf die Eingabeparameter des Knotens.

Das Attribut eines Elementes aus einem Tupel ist die Eingabestelle eines Elements, d.h. der Parameter eines Knotens.

Fun^R_{DG} sei nach Definition 3.56 eine Menge von Knotenfunktionen.

Sei $\varphi : V^R_{DG} \rightarrow Fun^R_{DG}$ eine Funktion die den Knoten des Datenflußgraphen eine Knotenfunktion zuordnet (im Zeichen φ_v). φ sei wie folgt definiert:

1. $v.type$ = IFACT:

   i) $\varphi_v() = (m_1, ..., m_{v.out})$
   $(m_1, ..., m_{v.out-1}$: Ausgabe aus dem Datenspeicher, $m_{v.out} = success$).

2. $v.type$ = FACT:

   i) $\varphi_v(M_1, ..., M_n) = \Pi_{v.out}(C \bowtie_= M_1 \bowtie_= ... \bowtie_= M_n)$, $n \in \mathbb{N}$,
   für $M_i = \{(m^r_{i,1}, ..., m^r_{i,k_i}) \mid u_i \in \mathbb{N},\ r \leq u_i,\ k_i \leq v.in\}$, $i \in \{1, ..., n\}$.

   ii) $\varphi_v() = \Pi_{v.out}(C)$.

   iii) $\varphi_v(..., \emptyset, ...) = \emptyset$.

3. $v.type$ = FUNC:

   i) $\varphi_v(M_1, ..., M_n) = \Pi_{v.out}(f(C \bowtie_= M_1 \bowtie_= ... \bowtie_= M_n))$, $n \in \mathbb{N}$,
   für $M_i = \{(m^r_{i,1}, ..., m^r_{i,k_i}) \mid u_i \in \mathbb{N},\ r \leq u_i,\ k_i \leq v.in\}$, $i \in \{1, ..., n\}$.

   ii) $\varphi_v() = \Pi_{v.out}(f(\Pi_{v.in}(C)))$.

   iii) $\varphi_v(..., \emptyset, ...) = \emptyset$.

4. $v.type$ = AND:

   i) $\varphi_v(M_1, ..., M_n) = \Pi_{v.out}(C \bowtie_= M_1 \bowtie_= ... \bowtie_= M_n)$, $n \in \mathbb{N}$,
   für $M_i = \{(m^r_{i,1}, ..., m^r_{i,k_i}) \mid u_i \in \mathbb{N},\ r \leq u_i,\ k_i \leq v.in\}$, $i \in \{1, ..., n\}$.

   ii) $\varphi_v() = C$.

   iii) $\varphi_v(..., \emptyset, ...) = \emptyset$.

5. $v.type$ = QUERY:

i) $\varphi_v(M_1, ..., M_n) = \Pi_{v.out}(C \bowtie_= M_1 \bowtie_= ... \bowtie_= M_n)$, $n \in \mathbb{N}$
(Ausgabe des Programms),
für $M_i = \{(m^r_{i,1}, ..., m^r_{i,k_i}) \mid u_i \in \mathbb{N}, r \leq u_i, k_i \leq v.in\}$, $i \in \{1, ..., n\}$.

ii) $\varphi_v(..., \emptyset, ...) = \emptyset$.

6. *v.type* = OR:

i) $\varphi_v(M_1, ..., M_n) = \Pi_{v.out}(M_1 \cup ... \cup M_n)$, $n \in \mathbb{N}$,
für $M_i = \{(m^r_{i,1}, ..., m^r_{i,v.vorg}) \mid u_i \in \mathbb{N}, r \leq u_i, k_i \leq v.in\}$, $\forall i \in \{1, ..., n\}$.

ii) $\varphi_v(M_1, ..., M_{i-1}, \emptyset, M_{i+1}, ..., M_n) = \varphi(M_1, ..., M_{j-1}, M_{j+1}, ..., M_n)$, $n \in \mathbb{N}$,
für $j \in \{1, ..., n)$.

iii) $\varphi() = \emptyset$.

7. *v.type* = NOT:

i) $\varphi_v(M_1, ..., M_n, M) = (M_1 \bowtie_= ... \bowtie_= M_n) \backslash M$, $n \in \mathbb{N}$,
für $M_i = \{(m^r_{i,1}, ..., m^r_{i,k_i}) \mid u_i \in \mathbb{N}, r \leq u_i, k_i \leq v.in\}$, $i \in \{1, ..., n\}$, $k_i \leq v.in$
und $M = \{(m^r_1, ..., m^r_{v.in}) \mid u \in \mathbb{N}, r \leq u\}$,
wobei für die Elemente von M gelte, daß sie von einem Knoten w mit $w.name = v.name$ erzeugt wurden, für alle anderen, daß sie nicht von w erzeugt wurden.
Bemerkung: Aufgrund der Vorbedinung der bedingten Sicherheit (siehe auch 3.1) gilt, daß alle Attribute von M in $M_1 \bowtie_= ... \bowtie_= M_n$ vorkommen.

ii) $\varphi_v(M_1, ..., M_n, \emptyset) = (M_1 \bowtie_= ... \bowtie_= M_n)$.

iii) $\varphi_v(..., \emptyset, ..., M) = \emptyset$.

8. *v.type* = RECDOWN:

i) $\varphi_v(M_1, ..., M_n) = M_1 \bowtie_= ... \bowtie_= M_n$, $n \in \mathbb{N}$, wobei
$M_i = \{(m^r_{i,1}, ..., m^r_{i,k_i}) \mid u_i \in \mathbb{N}, r \leq u_i, k_i \leq v.in\}, i \in \{1, ..., n\}$.

ii) $\varphi_v(..., \emptyset, ...) = \emptyset$.

9. *v.type* = RECUP:

i) $\varphi_v(M) = M$, wobei
$M = \{(m^r_1, ..., m^r_{v.in}) \mid u_i \in \mathbb{N}, r \leq u_i, k_i \leq v.in\}$.

ii) $\varphi_v(\emptyset) = \emptyset$. ◇

Satz 3.63

Durch die Einbeziehung der Ergebnisparameter in den normalen Berechnungsablauf entsprechend der in Definition 3.49 angegebenen Knotenfunktionalität, wird das von den Knoten zu berechnenden Ergebnis nicht verändert. ◇

Beweis:

- Ein IFACT-Knoten erzeugt die Belegung *success* für den Ergebnisparameter.
- Ein FUNC-Knoten erzeugt eine Belegung für den Ergebnisparameter, die dem Ergebnis der Funktionsausführung entspricht.
- Gibt es zwei Tupel t_1 und t_2 (mit $t_1 \in M_1$ und $t_2 \in M_2$), wobei die Eingabemengen M_1 und M_2 an einem FACT-, AND- oder QUERY-Knoten miteinander gejoint werden, die an allen Stellen mit gleichem Attribut (einschließlich des Ergebniswertes) identische Werte beinhalten, so gilt:

· haben beide Tupel den Ergebniswert *success*, so steht das Ausgabetupel der Knotenoperation mit Ergebniswert *success* für eine bis zu diesem Ausführungszeitpunkt erfolgreiche Berechnung.
· haben beide Tupel den Ergebniswert *failure*, so steht das Ausgabetupel der Knotenoperation mit Ergebniswert *failure* für eine zu diesem Ausführungszeitpunkt gescheiterte Berechnung.

Enthalten sie an allen Stellen mit gleichem Attribut bis auf den Ergebniswert identische Werte, d.h. ein Tupel enthält den Ergebniswert *failure*, so bedeutet dies, daß eine Berechnung mit diesen Werten an einer Stelle (z.B. einem FUNC-Knoten) im Graphen gescheitert ist. Die Nicht-Weitergabe von Werten steht ebenfalls für eine gescheiterte Berechnung.

- Der Ergebniswert des OR-Knotens gibt die Ergebniswerte analog zur Weitergabe der anderen Werten unverändert weiter.
- Für die bei einem NOT-Knoten gebildete Differenzmenge $M_1 \backslash M_2$ und zwei Tupel t_1 und t_2 (mit $t_1 \in M_1$ und $t_2 \in M_2$) gilt:
  · haben t_1 und t_2 den Ergebniswert *success*, so wird das Tupel nicht in die Ausgabemenge $M = M_1 \backslash M_2$ aufgenommen.
  · haben t_1 und t_2 den Ergebniswert *failure*, so wird das Tupel ebenfalls nicht in die Ausgabemenge $M = M_1 \backslash M_2$ aufgenommen.
  · Hat das Tupel t_1 den Ergebniswert *success*, das Tupel t_2 den Ergebniswert *failure*, so bedeutet das, daß die Negation für die Werte des Tupels t_2 gescheitert ist. Es ist daher korrekt, das Tupel t_1 mit Ergebniswert *success* in die Ausgabemenge M aufzunehmen.
  · Hat das Tupel t_1 den Ergebniswert *failure*, das Tupel t_2 den Ergebniswert *success*, so bedeutet das, daß eine Berechnung mit diesen Werten an einer Stelle im Graphen gescheitert ist. Der Erfolg der Negation liefert diesselbe Aussage. Es ist daher korrekt, das Tupel t_1 mit Ergebniswert *failure* in die Ausgabemenge M aufzunehmen, das dann für eine zu diesem Ausführungszeitpunkt gescheiterte Berechnung steht. ◇

Definition 3.64 *Eingabetupel, Ausgabetupel*

Für einen Knoten gelte zu einem diskreten Zeitpunkt: $M = \varphi_v(M_1, ...M_n)$, $n \in \mathbb{N}$, d.h. M ist das Ergebnis der Knotenoperation φ_v auf die Tupelmengen $M_1, ...M_n$.

Ein Tupel $(m^r_{i,1}, ..., m^r_{i,k_i})$, $i, k_i, r \in \mathbb{N}$ werde dann als *Eingabetupel* für einen Knoten v bezeichnet, wenn gilt: $(m^r_{i,1}, ..., m^r_{i,k_i}) \in M_i$ für $i \in \{1, .., n\}$.

Ein Tupel $(m^r_1, ..., m^r_k)$, $k, r \in \mathbb{N}$ werde dann als *Ausgabetupel* für einen Knoten v bezeichnet, wenn gilt: $(m^r_1, ..., m^r_k) \in M$. Es gilt allgemein: $k = v.out$. ◇

3.7 Korrektheit des Verfahrens

Die Vollständigkeit und Korrektheit der Programmauswertung ist für allgemeine Bottom-Up Verfahren (ohne Negation) hinlänglich bewiesen [Ullman 88].

Um die Vollständigkeit und Korrektheit der in dieser Arbeit vorgestellten Auswertungsmethode zu beweisen, ist zu zeigen, daß

1. das Berechnungsmodell des Datenflußgraphen für eine Basisklasse der Programme äquivalent zur Bottom-Up Auswertung dieser Basisklasse ist, d.h. die gleichen Ergebnisse liefert, und
2. die durch Eingaben an Fakten, Systemprädikaten und negierten Literalen entstehenden Änderungen der Graphstruktur Äquivalenzumformungen bezüglich der relationalen Algebra sind.

3.7.1 Semantische Äquivalenz von Basisprogrammen

Definition 3.65 *Basisprogramme*

Basisprogramme sind Programme, für die gilt:
- Fakten sind entweder Schnittstellen zum Datenspeicher (Schnittstellenfakten) oder enthalten nur Konstanten (konstante Programmfakten),
- es gibt keine Systemprädikate,
- es gibt keine Negation. ◇

Bemerkung 3.66

DATALOG-Programme sind Basisprogramme. ◇

Satz 3.67

Die Bottom-Up Evaluation mittels relationaler Ausdrücke von Basisprogrammen ist strukturäquivalent zur Datenflußgraph-basierten Auswertung. ◇

Beweis:

Behauptung: der Strukturgraph repräsentiert eine Bottom-Up Evaluation eines Logikprogramms mittels relationaler Ausdrücke (wenn den Knoten des Strukturgraphen die in Definiton 3.62 definierte Funktionalität zugeweisen wird).

Beweis: der Strukturgraph repräsentiert nach Satz 3.13 einen Operatorbaum: jeder Pfad im Struktrugraphen entspricht einem Berechnungsweg eine Top-Down Auswertung.

Grundlage für eine Bottom-Up Auswertung ist ebenfalls ein Operatorbaum.

Die relationalen Operationen, die in einem Operatorbaum vorkommen können, sind:

- ein *Join* für die konjunktive Verknüpfung konjunktiver Berechnungswege, die durch den Kopf einer Regel stattfindet,
- eine *Vereinigung* für die Vereinigung disjunktiver Berechnungswege, d.h. der von alternativen Regeln ausgegebenen Ergebnisse,

- die explizite *Definition* eines Wertetupel durch Fakten.

Die Knoten eines Strukturgraphen repräsentieren genau diese Elemente:

- jeder Kopf einer Regel wird durch einen AND-Knoten repräsentiert, dessen Funktionalität ein Join ist,
- der OR-Knoten verbindet alternative AND-Knoten miteinander und realisiert eine Vereinigung ihrer Ergebnisse,
- jedes Fakt wird durch einen FACT-Knoten repräsentiert, der ein explizit definiertes Wertetupel ausgibt.

Behauptung: Alle Parameter haben den Modus Ausgabe.

Beweis: Die Analyse der Parametermodi für IFACTS und für FACTS, die nur Konstanten enthalten, ergibt für alle Parameter den Modus Ausgabe.

Wenn alle Parameter an den Blättern des Strukturgraphen den Modus Ausgabe haben, so haben auch alle Parameter an den inneren Knoten des Strukturgraphen den Modus Ausgabe (siehe Def. 3.39).

Behauptung: Gibt es keine Schleifen im Datenflußgraphen, ist der Datenflußgraph bis auf die Richtung der Kanten strukturell identisch zum Strukturgraphen.

Beweis: Haben alle Parameter des Graphen den Modus Ausgabe, werden nur Ausgabekanten nach Definition 3.52 2.i erzeugt.

Da es für jeden Parameter eines Knotens w nach Satz 3.25 einen korrespondierenden Parameter am Knoten v mit $e_{v,w} \in E^v \cup E^m$ gibt, werden für alle Parameter von w Kanten zu den Parametern von v erzeugt. Diese Kanten entsprechen den Kanten des Strukturgraphen. Die Kante ist jedoch entgegengesetzt gerichtet.

Da es keine Parameter mit Modus Eingabe gibt, werden keine Kanten nach Definition 3.52 2.ii bis iv erzeugt, d.h. es werden keine Kanten erzeugt, die zwei Knoten verbinden, die nicht schon im Strukturgraphen mit einer Kante verbunden waren.

Behauptung: Für jede Schleife im Strukturgraphen gibt es genau eine Schleife im Datenflußgraphen.

Beweis: Da es nur Ausgaben gibt, wird für den obersten Schleifenknoten im Strukturgraphen v nur der Knoten v_u mit $v_u.type$ = RECUP, $v_u.name = v.name$ erzeugt.

Für die rückführende Kante mit Endknoten v im Strukturgraphen wird ein entsprechender Pfad über v_u erzeugt. Da durch die rückführende Kante eine Schleife im Strukturgraphen erzeugt wurde, erzeugt dies nun im Datenflußgraphen eine Schleife.

Da v_d nicht existiert, kann keine weitere Schleife erzeugt werden. Damit gibt es genau eine Schleife.

Behauptung: Ein Datenflußgraph mit Zyklen ist (unter Vernachlässigung der Knoten v_u) strukturell identisch zu einem Strukturgraphen mit Zyklen.

Beweis: Zu jeder Schleife im Strukturgraphen gibt es genau eine Schleife im Datenflußgraphen. Darüberhinaus wird sie an entsprechender Stelle erzeugt: für eine rückführende Kante von einem (beliebigen) Knoten w nach v im Strukturgraphen geht die rückführende Kante nun von w nach v_u.

Behauptung: Das Einfügen des Knotens v_u mit $v_u.type$ = RECUP, $v_u.name = v.name$ zu einem obersten Schleifenknoten im Strukturgraphen v ist semantikerhaltend.

Beweis: Im Fall einer Rekursionsschleife wird ein Knoten w_u über dem OR-Knoten für die Wurzel des zyklischen Teilgraphen im Strukturgraphen eingefügt. Es wird nach Definition 3.54 anstelle einer Kante im Strukturgraphen ein Pfad, bestehend aus zwei Kanten über den Knoten v_u, erzeugt. Für die rückführende Kante mit Endknoten v im Strukturgraphen wird ein entsprechender Pfad über v_u erzeugt.

Für einen beliebigen Knoten w von dem aus v erreichbar war, gilt nun, daß w über einen Pfad erreichbar bleibt. Andere Knoten außer v_u können nach wie vor nicht erreicht werden. Da v_u keine eigene Berechnungsfunktionalität hat, bleibt das Ergebnis der Ausführung des Graphen von dieser Veränderung unberührt.

Behauptung: Die Funktionalität aller im Graphen auftretenden Knoten ist identisch zur Funktionalität der Knoten im Operatorbaum, die durch Operationen der relationalen Algebra ausgedrückt wird, oder die Knoten haben keine eigene Funktionalität im Sinne der relationalen Algebra.

Beweis: Die Knoten v_u und v_d mit $v_u.type$ = RECUP, $v_d.type$ = RECDOWN, haben keine Berechnungs-Funktionalität.

FUNC-Knoten kommen für Basisprogramme nicht vor.

FACT-Knoten erhalten keine Eingaben. Daher muß für sie kein zusätzlicher Join über Eingaben durchgeführt werden. Statt dessen spezifizieren sie eine Relation über die in ihnen enthaltenen Konstanten. Ihre Funktionalität ist damit identisch zur Funktionalität in Bottom-Up Verfahren.

Schlussfolgerung: Da für einen Datenflußgraphen für Basisprogramme
- die Struktur äquivalent zur Struktur eines Dependency-Graphen ist,
- die Funktionalität der Knoten äquivalent zur ihrer durch Operationen der relationalen Algebra ausgedrückten Funktionalität in Bottom-Up Verfahren ist und
- bottom-Up ausgewertet wird,

gilt die Äquivalenz der Ergebnisse der Berechnung. ◇

3.7.2 Korrektheit der Transformation

Programme, die nicht den Anforderungen für Basisprogramme genügen, enthalten Literale, deren Parameter den Modus Eingabe besitzen. Solche Literale sind z.B. Systemprädikate oder Programmfakten mit Variablen.

Für diese Programme ist der Datenflußgraph nicht mehr äquivalent zu einem bottom-up ausgewerteten Operatorbaum. Der Unterschied besteht im wesentlich aus den folgenden zwei Punkten:

- Erhält ein Knoten für einen bestimmten Parameter Eingaben, gibt es eine Kante von jeder Wurzel eines disjunkten, benachbarten Teilgraphen, in dem für diesen Parameter Werte erzeugt werden.
- Erhält ein Knoten Eingaben, wird zunächst ein Join über diese Eingaben vor der eigentlichen Knotenoperation durchgeführt.

Satz 3.68

Benötigt ein Knoten v für einen bestimmten Parameter Eingaben, die kein Knoten in dem von v aufgespannten Teilgraphen erzeugt, so wird von jeder Wurzel eines Teilgraphen, der

- ein disjunkter, benachbarter Teilgraph zu einem Teilgraph ist, der den Knoten enthält, und
- in dem für diesen Parameter Werte erzeugt werden,

eine Kante zu dem Knoten erzeugt. ◇

Beweis:

Dies gilt nach Konstruktion des Datenflußgraphen (siehe Def. 3.52) durch die

i) Erzeugung einer "Querkante" von v nach w, wenn ein Nachfolgeknoten w eines AND-Knotens eine Eingabe von einem anderen Nachfolgeknoten v des AND-Knotens erhält. Dies ist die Wurzel eines benachbarten, disjunkten Teilgraphen zu einem Teilgraphen, dessen Wurzel der betrachtete Nachfolgeknoten w darstellt (siehe Def. 3.56 2. ii).

ii) Erzeugung einer "Querkante" von v nach w, wenn ein Nachfolgeknoten w eines Knotens w' eine Eingabe benötigt, die auch der Knoten w' benötigt, und für die w' eine Eingabe von v erhält. Für w' gilt nach i) oder ii), daß w' ein Knoten in einem benachbarten, disjunkten Teilgraphen zu dem Teilgraphen mit Wurzel v ist. Damit gilt auch für w, daß w ein Knoten in einem benachbarten, disjunkten Teilgraphen zu dem Teilgraphen mit Wurzel v ist (siehe Def. 3.56 2. iii).

Der tatsächliche Erzeugungsort für einen als Eingabe verwendeten Wert kann entweder die Wurzel v selbst, oder ein Knoten im von v aufgespannten Teilgraphen sein. Dies ist jedoch unrelevant, da die Ausgabe zunächst immer bis zur Wurzel des Teilgraphen geht (siehe Def. 3.56 2. iv). ◇

Bemerkung 3.69

Erhält ein Knoten v Eingaben, die ein Knoten in dem von v aufgespannten Teilgraphen erzeugt, so gehen diese Eingaben ausschließlich über Kanten, die äquivalent zu Kanten im Strukturgraphen und damit auch zum Operatorbaum für Bottom-Up Verfahren sind (siehe Def. 3.56 2. i, sowie Satz 3.67).

Satz 3.70

Die Operation eines Knotens setzt sich zusammen aus
- einer Operation, die der Operation dieses Knotens in einer Bottom-Up Auswertung eines Operatorbaums entspricht und
- falls der Knoten kein OR-Knoten ist, zusätzlich einem Join. ◇

Beweis:

($\mathcal{D}$ steht für eine nicht näher bestimmte Eingaberelation, die den zulässigen Wertebereich (*Domain*) aller Attribute enthält, E_1 und E_2 stehen für näher bestimmte Eingaberelationen, d.h. konkrete Eingaben.)

1. IFACT, FACT

   Die Grundfunktionalität eines FACTs ist die Definition von Tupeln. Es kann damit als Selektion über $\mathcal{D} \times ... \times \mathcal{D}$ verstanden werden. Konstanten werden im Sinne eines Selektionskriteriums verstanden. Erhält ein FACT im Datenflußmodell Eingaben, so wird zusätzlich zunächst ein Join über diese Eingaben durchgeführt. Konstanten werden dann im Sinne von Eingaben interpretiert.

   Die Funktionalität von IFACTs für die Schnittstelle zu einem Datenspeicher läßt sich nicht mit Ausdrücken der relationalen Algebra darstellen. Abstrakt läßt sich

ein IFACT jedoch ebenfalls als Definition von Tupeln (genauer gesagt einem einzigen Tupel nach Definition) und damit als Selektion über $\mathcal{D} \times \ldots \times \mathcal{D}$ verstehen. Ein IFACT erzeugt grundsätzlich nur Ausgaben.

2. FUNC

   Für die Funktionalität von FUNC gibt es keine direkte Entsprechung in relationaler Algebra. Ein FUNC kann jedoch abstrakt als Selektion aus dem Kreuzprodukt der Eingaben mit den zulässigen und gegebenenfalls unendlichen Wertebereich für die Ausgabeparameter verstanden werden. Konstanten werden im Sinne eines Selektionskriteriums interpretiert.

   Beispiele:

   - *Zuweisungen:* Die Zuweisung X *is* Y kann als Relation $X = Y$ verstanden werden, d.h. durch diese Zuweisung werden alle Tupel spezifiziert, für die diese Relation gilt, gegebenfalls unter zusätzlichen Bedingungen wie der Einschränkung auf bestimmte Belegungen für X und Y durch gegebene Eingaben E_1, E_2. Also läßt sich eine Zuweisung darstellen als: $\sigma_{X=Y}(E_1 \times E_2)$.
   - *Vergleiche:* Ein Vergleich $X < Y$ mit Eingaben E_1, E_2 kann verstanden werden als: $\sigma_{X<Y}(E_1 \times E_2)$.
   - *Unifikationen:* Eine Unifikation ist entweder eine Zuweisung oder ein Vergleich (auf Basis von Termstrukturen).
   - *Funktionen:* Ein Systemprädikat $f(X, Y, Z)$ mit X als Eingabeparameter und Y und Z als Ausgabeparameter, kann mit der Relation f_R und den Eingaben E_1, E_2, E_3 verstanden werden als: $\sigma_{f_R(X,Y,Z)}(E_1 \times E_2 \times E_3)$.
   - *Operationen:* Eine Operation wird wie eine Funktion durch ein Systemprädikat realisiert.

   Erhält der FUNC-Knoten Eingaben, so muß zusätzlich zunächst ein Join über diesen Eingaben durchgeführt werden. Konstanten werden dann im Sinne von Eingaben interpretiert.

3. AND, QUERY

   Die Grundfunktionalität eines AND-Knotens ist die eines Joins. Konstanten werden im Sinne von Eingaben interpretiert. Die Funktionalität des QUERY-Knotens ist analog. Die explizite Ausgabe ist für eine Darstellung der Funktionalität mit Ausdrücken der relationalen Algebra nicht relevant.

4. OR

   Die Grundfunktionalität eines OR-Knoten ist die einer Vereinigung. Die Typen der Nachfolgeknoten beschränken sich, falls der OR-Knoten Eingaben erhält, die nicht von den Nachfolgeknoten stammen, auf AND- und FACT-Knoten. Diese erhalten nach Konstruktion die gleichen Eingaben und führen über die entsprechenden Eingaben einen Join durch. Die von den verschiedenen Nachfolgeknoten stammenden Werte sind außerdem Ergebnisse alternativer Berechnungswege und dürfen daher nicht miteinander gejoint werden. Daher ist ein Join am OR-Knoten überflüssig.

5. NOT

   Die Grundfunktionalität eines NOT-Knoten ist die einer Differenzmengenbildung. Im Datenflußmodell wird zusätzlich zunächst ein Join über eine Teilmenge der Eingaben durchgeführt. Dabei handelt es sich um die Eingaben, die nicht vom Nachfolgeknoten stammen.

6. RecDown
   Der RecDown-Knoten hat keine eigene Grundfunktionaltät im Sinne des Verarbeitens von Tupelmengen. Da er jedoch Eingaben von verschiedenen Knoten erhält, die er gemeinsam weitergibt, führt er einen Join über diese Eingaben durch. Er nimmt damit den Join an nachfolgenden Knoten vorweg. Diese erhalten ihre Eingaben durch Einfügen des RecDown-Knoten nicht mehr von verschiedenen Knoten, sondern von ihm selbst.

7. RecUp
   Der RecUp-Knoten hat keine eigene Grundfunktionalität. Da sein einziger Nachfolgeknoten grundsätzlich ein Or-Knoten ist, gilt für ihn dieselbe Argumentation wie für den Or-Knoten. ⋄

Die schematische Funktionalität der Knoten ist in Abbildung 37 veranschaulicht. Die weißen Felder stehen dabei für die Anteile der Funktionalität, die nicht mit den Methoden der relationalen Algebra ausdrückbar sind und keine eigentliche Auswirkung auf die Funktionalität haben.

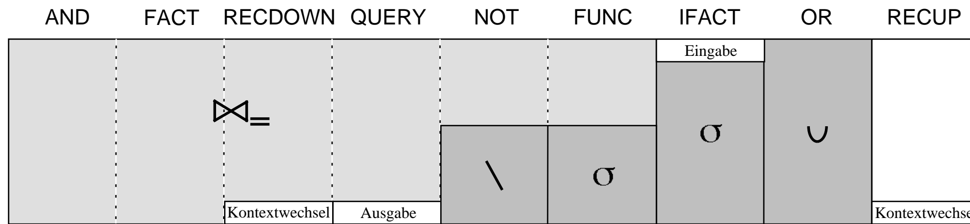


Abbildung 37: Schematische Darstellung der Knotenfunktionalität

Satz 3.71

Die im Fall von Eingaben an Knoten zusätzlich eingeführten Kanten (außer bei Or-Knoten) verändern das Ergebnis der Berechnung nicht. ⋄

Beweis:

Sei $DG^R_{\mathcal{L}} = (V^R_{DG}, E^R_{DG}, Fun^R_{DG})$ ein Datenflußgraph zu einem Strukturgraphen $SG^R_{\mathcal{L}} = (V^R, E^R)$.

Die Begriffe Nachfolge- und Vorgängerknoten bezeichnen auch weiterhin die Ordnung der Knoten, die durch ihre Kantenverbindung im Strukturgraphen gegeben ist.

Für die Beschreibung der Kantenverbindungen im Datenflußgraphen soll die Bezeichnung einer Kante “von Knoten v' nach Knoten w” für die Bezeichnung “von einem Parameter an Knoten v' zu einem Parameter an Knoten w” stehen. Von der Anzahl der Kanten werde außerdem abstrahiert.

Sei $e_{v',w}$ eine nach Satz 3.68 eingeführte Kante zum Knoten w. (Aus der Erzeugung von e folgt, daß w offensichtlich Eingaben benötigt.)

Nach Satz 3.68 gilt darüberhinaus, daß ein Knoten v existiert, mit $v.type =$ And und daß v' ein Nachfolgeknoten von v ist. Außerdem gilt für w, daß w ein Knoten in einem von einem Knoten w' aufgespannten Teilgraphen ist, wobei w' Nachfolgeknoten von v ist. w kann insbesondere auch w' selbst sein.

Allgemein geltende Regeln für Ausdrücke der relationalen Algebra, die in den folgenden Beweisen benötigt werden, sind (siehe auch Anhang A.4):

$$E_1 \bowtie_{cond_3} E_2 = \sigma_{cond_3}(E_1 \times E_2) \quad (1)$$
$$\sigma_{cond_1}(E_1 \times E_2) = \sigma_{cond_1}(E_1) \times E_2 \quad (2)$$
$$\sigma_{cond_1}(\sigma_{cond_2}(E)) = \sigma_{cond_2}(\sigma_{cond_1}(E)) \quad (3)$$
$$(E_1 \bowtie_{cond_3} E_2) \bowtie_{cond_4} E_3 = E_1 \bowtie_{cond_3} (E_2 \bowtie_{cond_4} E_3) \quad (4)$$
$$E_1 \equiv (E_1 \bowtie_{=} E_1) \quad (5)$$
$$E_1 \equiv (E_1 \bowtie_{=} \pi_{cond_1}(E_1)) \text{ (mit Duplikatenelimination)} \quad (6)$$

$cond_1$ und $cond_2$ sind Bedingungen in der nur Attribute von E_1 auftreten, $cond_3$ ist eine Bedingung auf den Attributen von E_1 und E_2, $cond_4$ ist eine Bedingung auf den Attributen von E_2 und E_3

1. $w = w'$.

   Im Operatorbaum gibt es sowohl vom Knoten v', als auch vom Knoten w Kanten zum (AND-)Knoten v. An diesem wird ein Join durchgeführt. Im Datenflußgraph existieren diese Kanten ebenfalls (nach Def. 3.56 2. i).

   Nun ist also zu zeigen, daß durch die zusätzliche Kante von v' nach w und den zusätzlichen Join, der am Knoten w durchgeführt wird, das Ergebnis des AND-Knotens v nicht verändert wird.

   w kann nicht den Typ QUERY, IFACT oder FACT ohne Eingaben haben, da diese Knoten grundsätzlich keine Eingaben erhalten. w kann also nur ein Knoten mit folgendem Typ sein:

   i) OR, RECUP:

   Für diese Knoten gilt, daß sie ausschließlich mit Eingaben von Nachfolgeknoten arbeiten. Eingaben von anderen Knoten werden nach Definition 3.62 nicht berücksichtigt, demnach verändern sie das Ergebnis nicht.

   Abbildung 38 zeigt zur Veranschaulichung auf der linken Seite den Operatorbaum, auf der rechten den Datenflußgraphen für den Fall des OR-Knotens. Der der Abbildung entsprechende Ausdruck in relationaler Algebra lautet für beide Graphen: $E_1 \bowtie_{=} (E_2 \cup E_3)$.

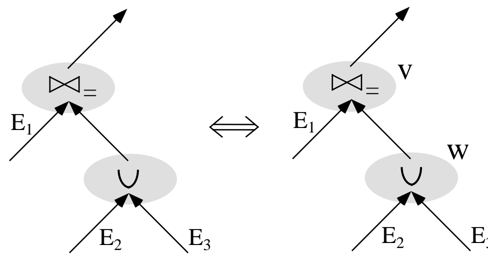


Abbildung 38: Operatorbaum und Datenflußgraph mit Vereinigung

   ii) FUNC, FACT mit Eingaben:

   Diese Knoten können, wie bereits erläutert, als Selektion σ_f verstanden werden. Der Operatorbaum ist daher offensichtlich durch den relationalen Ausdruck $E_1 \bowtie_{=} \sigma_f(E_2)$ darstellbar, der Datenflußgraph durch den relationalen Ausdruck $E_1 \bowtie_{=} \sigma_f(\pi_{E_2}(E_1) \bowtie_{=} E_2)$, wobei $\pi_{E_2}(E_1)$ eine Projektion von E_1 auf Elemente ist, deren Attribute auch in E_2 vorkommen.

   Es ist also zu zeigen: $E_1 \bowtie_{=} \sigma_f(E_2) = E_1 \bowtie_{=} \sigma_f(\pi_{E_2}(E_1) \bowtie_{=} E_2)$.

$$
\begin{aligned}
&E_1 \bowtie_= \sigma_f(E_2) = && (6)\\
&(E_1 \bowtie_= \pi_{E_2}(E_1)) \bowtie_= \sigma_f(E_2) = && (4)\\
&E_1 \bowtie_= (\pi_{E_2}(E_1) \bowtie_= \sigma_f(E_2)) = && (1)\\
&E_1 \bowtie_= \sigma_=(\pi_{E_2}(E_1) \times \sigma_f(E_2))
\end{aligned}
$$

Ist σ_f eine Bedingung auf Attributen von E_2, die nicht in E_1 vorkommen, so gilt:

$$
\begin{aligned}
&\sigma_=(\pi_{E_2}(E_1) \times \sigma_f(E_2)) = && (2)\\
&\sigma_=(\sigma_f(\pi_{E_2}(E_1) \times E_2))
\end{aligned}
$$

Ist σ_f eine Bedingung auf Attributen, die auch in E_1 vorkommen, so gilt für ein Tupel $t \in \sigma_f(\pi_{E_2}(E_1) \times E_2)$, daß es die Selektionsbedingung σ_f erfüllt, aber auch, daß es $\pi_{E_2}(E_1) \times E_2$ ist. Damit ist t auch in $\pi_{E_2}(E_1) \times \sigma_f(E_2)$.

Es gilt also:

$$\sigma_f(\pi_{E_2}(E_1) \times E_2) \subseteq \pi_{E_2}(E_1) \times \sigma_f(E_2)$$

und damit auch:

$$\sigma_=(\sigma_f(\pi_{E_2}(E_1) \times E_2)) \subseteq \sigma_=(\pi_{E_2}(E_1) \times \sigma_f(E_2))$$

Sei $E = (\pi_{E_2}(E_1) \times \sigma_f(E_2)) \backslash (\sigma_f(\pi_{E_2}(E_1) \times E_2))$, dann enthält E nur Tupel $t \in \pi_{E_2}(E_1) \times \sigma_f(E_2)$, die an Stellen mit Attributen aus E_1 auf denen die Selektionsbedingung σ_f gilt, nur Werte haben, die nicht der Selektionsbedingung σ_f entsprechen. Für diese Tupel gilt aber ebenfalls, daß sie an den Stellen mit Attributen aus E_2 auf denen die Selektionsbedingung σ_f gilt, nur Werte haben, die der Selektionsbedingung σ_f entsprechen.

Daher gilt: $(\sigma_=(\pi_{E_2}(E_1) \times \sigma_f(E_2))) \backslash (\sigma_=(\sigma_f(\pi_{E_2}(E_1) \times E_2))) = \emptyset$ und es folgt:

$$\sigma_=(\sigma_f(\pi_{E_2}(E_1) \times E_2)) = \sigma_=(\pi_{E_2}(E_1) \times \sigma_f(E_2))$$

Also folgt:

$$
\begin{aligned}
&E_1 \bowtie_= \sigma_=(\pi_{E_2}(E_1) \times (\sigma_f(E_2)) = && (2)\\
&E_1 \bowtie_= \sigma_=(\sigma_f(\pi_{E_2}(E_1) \times E_2)) = && (3)\\
&E_1 \bowtie_= \sigma_f(\sigma_=(\pi_{E_2}(E_1) \times E_2)) = && (1)\\
&E_1 \bowtie_= \sigma_f(\pi_{E_2}(E_1) \bowtie_= E_2)
\end{aligned}
$$

Abbildung 39 zeigt auf der linken Seite den Operatorbaum, auf der rechten den Datenflußgraphen am Beispiel eines Fakts als betrachteten Nachfolgeknoten.

iii) AND, RECDOWN:

Werden an w die Mengen E_2 und E_3 gejoint, so ist der Operatorbaum offensichtlich durch den relationalen Ausdruck $E_1 \bowtie_= (E_2 \bowtie_= E_3)$ darstellbar, der Datenflußgraph durch den relationalen Ausdruck $E_1 \bowtie_= (\pi_{E_2}(E_1) \bowtie_= (E_2 \bowtie_= E_3))$, wobei $\pi_{E_2}(E_1)$ eine Projektion von E_1 auf Elemente ist, deren Attribute auch in E_2 vorkommen.

Zu zeigen ist: $E_1 \bowtie_= (E_2 \bowtie_= E_3) = E_1 \bowtie_= (\pi_{E_2}(E_1) \bowtie_= (E_2 \bowtie_= E_3)))$.

$$
\begin{aligned}
&E_1 \bowtie_= (E_2 \bowtie_= E_3) = && (6)\\
&(E_1 \bowtie_= \pi_{E_2}(E_1)) \bowtie_= (E_2 \bowtie_= E_3) = && (4)\\
&E_1 \bowtie_= (\pi_{E_2}(E_1) \bowtie_= (E_2 \bowtie_= E_3))
\end{aligned}
$$

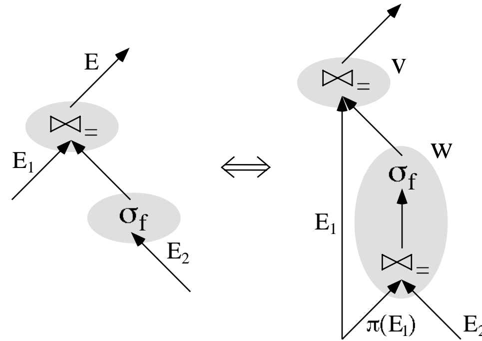


Abbildung 39: Operatorbaum und Datenflußgraph mit Selektion

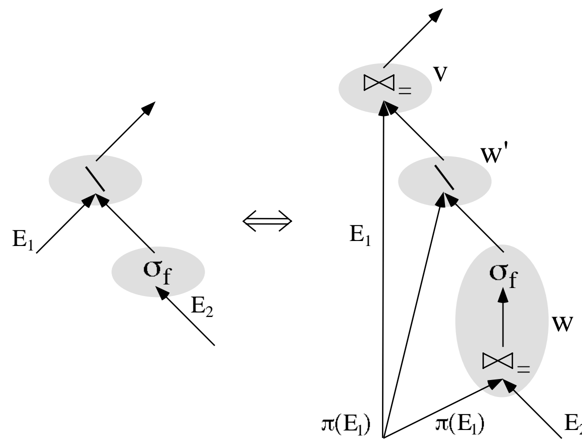


Abbildung 40: Operatorbaum und Datenflußgraph mit Join

Der Fall, daß an w die Mengen $E_2, ..., E_n$ anstelle von E_2 und E_3 gejoint werden, ist analog beweisbar.

Abbildung 40 zeigt auf der linken Seite den Berechnungbaum, auf der rechten den Datenflußgraphen am Beispiel eines Regelkopfs.

iv) NOT:

Die Negation ist eine Differenzmengenbildung, wobei von einer Eingabemenge E_2 eine Menge E_1 abgezogen wird und aufgrund der Vorbedingung der bedingten Sicherheit E_2 die gleichen Attribute enthalten muß, wie E_1. Für den Operatorbaum steht daher der relationale Ausdruck: $E_2 \backslash E_1$, für den Datenflußgraph $E_2 \bowtie_= (E_2 \backslash E_1)$ (siehe Abbildung 41).

Es ist also zu zeigen: $E_2 \backslash E_1 = E_2 \bowtie_= (E_2 \backslash E_1)$.

Sei $E \subset E_2$. Dann haben E und E_2 die gleichen Attribute und es gilt:

$$E \subset E_2 \Rightarrow E \bowtie_= E_2 = E$$

Außerdem gilt nach Definition der Differenzmengenbildung:

$$E_2 \backslash E_1 \subset E_2$$

und somit:

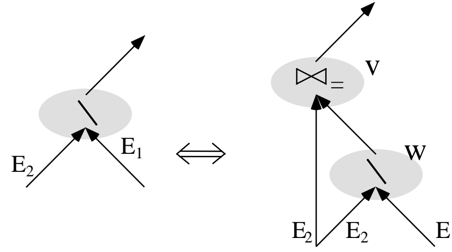


Abbildung 41: Operatorbaum und Datenflußgraph mit Join

$$E_2 \bowtie_= (E_2 \backslash E_1) = E_2 \backslash E_1$$

Ist eine Menge E_2 nicht ausreichend, um als Eingabemenge die Bedingung der bedingten Sicherheit zu erfüllen, werden mehrere Mengen als Eingabe verwendet. Über diese wird zunächst ein Join durchgeführt, bevor die Differenzmenge gebildet wird. Für den Operatorbaum stehe dann der relationale Ausdruck: $(E_2 \bowtie_= ... \bowtie_= E_n) \backslash E_1$, für den Datenflußgraph der relationale Ausdruck $(E_2 \bowtie_= ... \bowtie_= E_n) \bowtie_= (E_2 \bowtie_= ... \bowtie_= E_n) \backslash E_1)$. Dies ist analog beweisbar.

2. $w \neq w'$, d.h. es existiert ein Pfad im Strukturgraphen zwischen w und w'. (Für v' kann immer nur gelten, daß v' direkter Nachfolgeknoten des AND-Knotens v ist.)

   Induktion über die Länge dieses Pfades.

   2a) w ist Nachfolgeknoten von w', d.h. der Pfad zwischen w' und w hat die Länge 1.

   Im Operatorbaum geht dann eine Kante von w zu w' und von w' zum (AND-)Knoten v. Außerdem gibt es nach wie vor eine Kante von v' nach v. Diese Kanten gibt es auch im Datenflußgraph.

   Nun ist zu zeigen, daß durch die zusätzliche Kante von v' nach w und den zusätzlichen Join, der am Knoten w durchgeführt wird, das Ergebnis des AND-Knotens v ebenfalls nicht verändert wird.

   w' und w können nur den gleichen Knotentyp wie in Fall 1 haben (mit gleicher Begründung).

   Habe w' folgenden Typ:

   i) OR, RECUP:

   Zuerst soll folgende allgemeine Aussage gezeigt werden:

$$E_1 \bowtie_= (E_2 \cup E_3) =$$
$$E_1 \bowtie_= (E_2 \cup (\pi_{E_3}(E_1) \bowtie_= E_3))$$

   wobei $\pi_{E_3}(E_1)$ eine Projektion von E_1 auf Elemente ist, deren Attribute auch in E_3 (und damit auch in E_2) vorkommen. (Für eine Vereinigung $E_2 \cup E_3$ gilt, daß beide Mengen die gleichen Attribute besitzen müssen.)

   Ist ein Tupel $t \in \pi_{E_3}(E_1) \bowtie_= E_3$, so gilt sicherlich auch $t \in \pi_{E_3}(E_1) \bowtie_= (E_3 \cup E_2)$, bzw. ist $t \in \pi_{E_3}(E_1) \bowtie_= E_2$, so gilt ebenfalls $t \in \pi_{E_3}(E_1) \bowtie_= (E_3 \cup E_2)$, d.h. es gilt:

$$(\pi_{E_3}(E_1) \bowtie_= E_2) \cup (\pi_{E_3}(E_1) \bowtie_= E_3) \subset \pi_{E_3}(E_1) \bowtie_= (E_2 \cup E_3)$$

Ist umgekehrt ein Tupel $t \in \pi_{E_3}(E_1) \bowtie_= (E_3 \cup E_2)$, so muß t an Stellen gleichen Attributes die gleichen Werte haben wie Tupel in $\pi_{E_3}(E_1)$, aber auch Element von E_3 oder E_2 sein. D.h. t ist entweder in $\pi_{E_3}(E_1) \bowtie_= E_3$, oder in $\pi_{E_3}(E_1) \bowtie_= E_2$, also

$$\pi_{E_3}(E_1) \bowtie_= (E_2 \cup E_3) \subset (\pi_{E_3}(E_1) \bowtie_= E_2) \cup (\pi_{E_3}(E_1) \bowtie_= E_3)$$

Damit gilt Gleichheit:

$$(\pi_{E_3}(E_1) \bowtie_= E_2) \cup (\pi_{E_3}(E_1) \bowtie_= E_3) = \pi_{E_3}(E_1) \bowtie_= (E_2 \cup E_3)$$

Ist ein Tupel $t \in \pi_{E_3}(E_1) \bowtie_= E_3$, so gilt sicherlich auch $t \in E_2 \cup (\pi_{E_3}(E_1) \bowtie_= E_3)$. Da t an Stellen gleichen Attributes die gleichen Werte hat wie Tupel in $\pi_{E_3}(E_1)$, gilt auch: $t \in \pi_{E_3}(E_1) \bowtie_= (E_2 \cup (\pi_{E_3}(E_1) \bowtie_= E_3))$. Ist ein Tupel $t \in \pi_{E_3}(E_1) \bowtie_= E_2$, so gilt auf jeden Fall: $t \in E_2$ und damit auch $t \in E_2 \cup (\pi_{E_3}(E_1) \bowtie_= E_3))$ und, da t an Stellen gleichen Attributes die gleichen Werte hat wie Tupel in $\pi_{E_3}(E_1)$, gilt auch: $t \in \pi_{E_3}(E_1) \bowtie_= (E_2 \cup (\pi_{E_3}(E_1) \bowtie_= E_3))$.

Also:

$$\begin{aligned}&(\pi_{E_3}(E_1) \bowtie_= E_2) \cup (\pi_{E_3}(E_1) \bowtie_= E_3) \subset\\&\pi_{E_3}(E_1) \bowtie_= (E_2 \cup (\pi_{E_3}(E_1) \bowtie_= E_3))\end{aligned}$$

Ist $t \in \pi_{E_3}(E_1) \bowtie_= (E_2 \cup (\pi_{E_3}(E_1) \bowtie_= E_3))$, so gilt $t \in \pi_{E_3}(E_1)$, sowie $t \in \pi_{E_3}(E_1) \bowtie_= E_3)$ oder alternativ $t \in E_2$. Ist $t \in E_2$ gilt damit auch: $t \in \pi_{E_3}(E_1) \bowtie_= E_2)$.

Damit gilt:

$$\begin{aligned}&\pi_{E_3}(E_1) \bowtie_= (E_2 \cup (\pi_{E_3}(E_1) \bowtie_= E_3)) \subset\\&(\pi_{E_3}(E_1) \bowtie_= E_2) \cup (\pi_{E_3}(E_1) \bowtie_= E_3)\end{aligned}$$

und es folgt die Gleichheit:

$$\begin{aligned}&(\pi_{E_3}(E_1) \bowtie_= E_2) \cup (\pi_{E_3}(E_1) \bowtie_= E_3) =\\&\pi_{E_3}(E_1) \bowtie_= (E_2 \cup (\pi_{E_3}(E_1) \bowtie_= E_3))\end{aligned}$$

Zusammengefaßt erhält man:

$$\begin{aligned}&E_1 \bowtie_= (E_2 \cup E_3) =\\&(E_1 \bowtie_= \pi_{E_3}(E_1)) \bowtie_= (E_2 \cup E_3) =\\&E_1 \bowtie_= (\pi_{E_3}(E_1) \bowtie_= (E_2 \cup E_3)) =\\&E_1 \bowtie_= ((\pi_{E_3}(E_1) \bowtie_= E_2) \cup (\pi_{E_3}(E_1) \bowtie_= E_3)) =\\&E_1 \bowtie_= (E_2 \cup (\pi_{E_3}(E_1) \bowtie_= E_3))\end{aligned}$$

w kann als direkter Nachfolgeknoten von w' nach Konstruktion des Strukturgraphen in Definition 3.8 nur folgende Knotentypen haben:

- Für $w.type$ = FACT:

  Operatorbaum: $E_1 \bowtie_= (E_2 \cup \sigma_f(E_3))$.
  Datenflußgraph: $E_1 \bowtie_= (E_2 \cup \sigma_f(\pi_{E_3}(E_1) \bowtie_= E_3))$,
  wobei $\pi_{E_3}(E_1)$ eine Projektion von E_1 auf Elemente ist, deren Attribute auch in E_2, bzw. E_3 vorkommen. (siehe Abbildung 42).

$$\begin{aligned}&E_1 \bowtie_= (E_2 \cup \sigma_f(E_3)) = && (6)\\&(E_1 \bowtie_= \pi_{E_3}(E_1)) \bowtie_= (E_2 \cup \sigma_f(E_3)) = && \text{s.o.}\\&E_1 \bowtie_= (E_2 \cup (\pi_{E_3}(E_1) \bowtie_= \sigma_f(E_3))) = && \text{mit 1. ii)}\\&E_1 \bowtie_= (E_2 \cup \sigma_f(\pi_{E_3}(E_1) \bowtie_= E_3))\end{aligned}$$

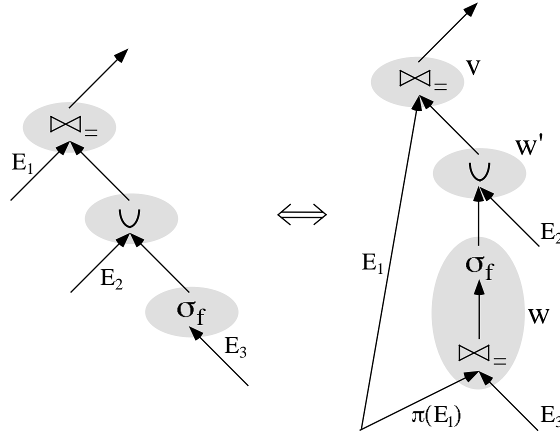


Abbildung 42: Operatorbaum und Datenflußgraph mit Join

- Für $w.type$ = AND:

  Operatorbaum: $E_1 \bowtie_= (E_2 \cup (E_3 \bowtie_= E_4))$.
  Datenflußgraph: $E_1 \bowtie_= (E_2 \cup (\pi_{E_3 \vee E_4}(E_1) \bowtie_= (E_3 \bowtie_= E_4)))$,
  wobei $\pi_{E_3 \vee E_4}(E_1)$ eine Projektion von E_1 auf Elemente ist, deren Attribute auch in E_3 oder E_4 vorkommen. Die Erweiterung von $E_3 \bowtie_= E_4$ auf $E_3 \bowtie_= ... \bowtie_= E_n$ läßt sich analog beweisen.

  $$\begin{array}{ll} E_1 \bowtie_= (E_2 \cup (E_3 \bowtie_= E_4)) = & (6) \\ (E_1 \bowtie_= \pi_{E_3 \vee E_4}(E_1)) \bowtie_= (E_2 \cup (E_3 \bowtie_= E_4)) = & \text{s.o.} \\ E_1 \bowtie_= (E_2 \cup (\pi_{E_3 \vee E_4}(E_1) \bowtie_= (E_3 \bowtie_= E_4))) = & \text{mit 1. iii)} \\ E_1 \bowtie_= (E_2 \cup (\pi_{E_3 \vee E_4}(E_1) \bowtie_= (E_3 \bowtie_= E_4))) & \end{array}$$

ii) FUNC, FACT mit Eingaben:

w kann als direkter Nachfolgeknoten von w' nach Konstruktion des Strukturgraphen in Definition 3.8 nur folgende Knotentypen haben:

- Für $w.type$ = OR gibt es nichts zu zeigen (mit gleicher Begründung wie bei Fall 1).

- Für $w.type$ = FUNC, FACT:

  Operatorbaum: $E_1 \bowtie_= (\sigma_g(\sigma_f(E_2))$.
  Datenflußgraph: $E_1 \bowtie_= (\sigma_g(\sigma_f(E_1 \bowtie_= E_2)))$.

  $$\begin{array}{ll} E_1 \bowtie_= (\sigma_g(\sigma_f(E_2)) = & (6) \\ (E_1 \bowtie_= \pi_{E_2}(E_1)) \bowtie_= (\sigma_g(\sigma_f(E_2)) = & \text{nach 1.ii)} \\ E_1 \bowtie_= (\sigma_g(\pi_{E_2}(E_1) \bowtie_= \sigma_f(E_2))) = & \text{nach 1.ii)} \\ E_1 \bowtie_= (\sigma_g(\sigma_f(\pi_{E_2}(E_1) \bowtie_= E_2))) & \end{array}$$

- Für $w.type$ = AND, RECDOWN:

  (siehe Abbildung 43).

  Operatorbaum: $E_1 \bowtie_= (\sigma_g(E_2 \bowtie_= E_3))$.
  Datenflußgraph: $E_1 \bowtie_= (\sigma_g(\pi_{E_2 \vee E_3}(E_1) \bowtie_= (E_2 \bowtie_= E_3)))$.
  Die Erweiterung auf $E_2 \bowtie_= ... \bowtie_= E_n$ für $E_2 \bowtie_= E_3$ läßt sich analog beweisen.

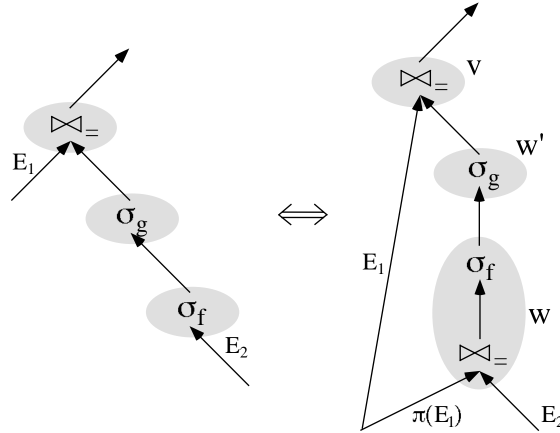


Abbildung 43: Operatorbaum und Datenflußgraph mit Join

$$E_1 \bowtie_= (\sigma_g(E_2 \bowtie_= E_3) = \quad (6)$$
$$(E_1 \bowtie_= \pi_{E_2 \vee E_3}(E_1)) \bowtie_= (\sigma_g(E_2 \bowtie_= E_3) = \quad \text{nach 1.ii)}$$
$$E_1 \bowtie_= (\sigma_g(\pi_{E_2 \vee E_3}(E_1) \bowtie_= (E_2 \bowtie_= E_3)))$$

iii) AND, RECDOWN:

w kann als direkter Nachfolgeknoten von w' nach Konstruktion des Strukturgraphen in Definition 3.8 nur folgende Knotentypen haben:

- Für $w.type$ = OR gibt es nichts zu zeigen (mit gleicher Begründung wie bei Fall 1).
- Für $w.type$ = FUNC, FACT:

  (siehe Abbildung 44).

  Operatorbaum: $E_1 \bowtie_= (E_2 \bowtie_= \sigma_f(E_3))$.
  Datenflußgraph: $E_1 \bowtie_= (E_2 \bowtie_= \sigma_f(\pi_{E_3}(E_1) \bowtie_= E_3))$.

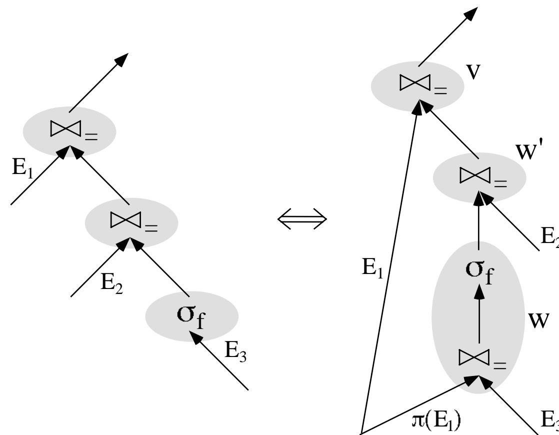


Abbildung 44: Operatorbaum und Datenflußgraph mit Join

$$\begin{array}{ll}
E_1 \bowtie_= (E_2 \bowtie_= \sigma_f(E_3)) = & (6)\\
(E_1 \bowtie_= \pi_{E_3}(E_1)) \bowtie_= (E_2 \bowtie_= \sigma_f(E_3)) = & (4)\\
E_1 \bowtie_= (\pi_{E_3}(E_1) \bowtie_= (E_2 \bowtie_= \sigma_f(E_3))) = & (4)\\
E_1 \bowtie_= ((\pi_{E_3}(E_1) \bowtie_= E_2) \bowtie_= \sigma_f(E_3)) = & \\
E_1 \bowtie_= ((E_2 \bowtie_= \pi_{E_3}(E_1)) \bowtie_= \sigma_f(E_3)) = & (4)\\
E_1 \bowtie_= (E_2 \bowtie_= (\pi_{E_3}(E_1) \bowtie_= \sigma_f(E_3))) = & \text{nach 1.ii)}\\
E_1 \bowtie_= (E_2 \bowtie_= \sigma_f(\pi_{E_3}(E_1) \bowtie_= E_3)) &
\end{array}$$

- Für $w.type$ = AND, RECDOWN:

  Operatorbaum: $E_1 \bowtie_= (E_2 \bowtie_= (E_3 \bowtie_= E_4))$.
  Datenflußgraph: $E_1 \bowtie_= (E_2 \bowtie_= (\pi_{E_3 \vee E_4}(E_1 \bowtie_= (E_3 \bowtie_= E_4)))$.
  Die Erweiterung von $E_3 \bowtie_= E_4$ auf $E_3 \bowtie_= ... \bowtie_= E_n$ läßt sich analog beweisen.

$$\begin{array}{ll}
E_1 \bowtie_= (E_2 \bowtie_= (E_3 \bowtie_= E_4)) = & (6)\\
(E_1 \bowtie_= \pi_{E_3 \vee E_4}(E_1)) \bowtie_= (E_2 \bowtie_= (E_3 \bowtie_= E_4)) = & \text{nach 1.iii)}\\
E_1 \bowtie_= (\pi_{E_3 \vee E_4}(E_1) \bowtie_= (E_2 \bowtie_= (E_3 \bowtie_= E_4))) = & \text{nach 1.iii)}\\
E_1 \bowtie_= (E_2 \bowtie_= (\pi_{E_3 \vee E_4}(E_1) \bowtie_= (E_3 \bowtie_= E_4))) &
\end{array}$$

- Für $w.type$ = NOT:

  Operatorbaum: $E_3 \bowtie_= (E_2 \backslash E_1)$.
  Datenflußgraph: $(E_3 \bowtie_= (\pi_{E_2}(E_3) \bowtie_= E_2) \backslash E_1)$.
  Die Erweiterung von E_3 auf $E_3 \bowtie_= ... \bowtie_= E_n$ läßt sich analog beweisen.

  E_1 und E_2 müssen die gleichen Attribute besitzen.

  Zunächst gilt für ein Tupel $t \in E_2 \backslash E_1$, daß $t \in E_2$, aber $t \notin E_1$. Für $t \in (\pi_{E_2}(E_3) \bowtie_= E_2) \backslash E_1$ gilt $t \in E_2$ und daß t an Stellen mit gleichen Attributen gleiche Werte wie ein Tupel in $\pi_{E_2}(E_3)$ hat. Außerdem gilt $t \notin E_1$. Für $t \in \pi_{E_2}(E_3) \bowtie_= (E_2 \backslash E_1)$ gilt ebenfalls, daß $t \in E_2$ und $t \notin E_1$. Außerdem gilt daß t an Stellen mit gleichen Attributen gleiche Werte wie ein Tupel in $\pi_{E_2}(E_3)$ hat. Es ist also offensichtlich die gleiche Menge.

$$\begin{array}{ll}
E_3 \bowtie_= (E_2 \backslash E_1) = & (6)\\
(E_3 \bowtie_= \pi_{E_2}(E_3)) \bowtie_= (E_2 \backslash E_1) = & \\
E_3 \bowtie_= ((\pi_{E_2}(E_3) \bowtie_= E_2) \backslash E_1) &
\end{array}$$

iv) NOT:

Für die Negation werde zunächst gezeigt:

$$\begin{array}{l}
E_1 \backslash E_2 =\\
E_1 \backslash (E_1 \bowtie_I E_2)
\end{array}$$

wobei E_1 und E_2 die gleichen Attribute besitzen müssen.

Da bei einer Differenzmengenbildung nur Tupel aus E_1 entfernt werden, gilt:

$$\begin{array}{l}
E_1 \backslash E_2 =\\
E_1 \backslash (E_1 \cap E_2)
\end{array}$$

Andererseits gilt:

$$E_1 \bowtie_= E_2 \subset E_1,$$

da E_2 keine Attribute enthält, die nicht in E_1 sind. Analog gilt:

$$E_1 \bowtie_= E_2 \subset E_2$$

und damit gilt:

$$E_1 \bowtie_= E_2 \subset E_1 \cap E_2$$

Da für alle Tupel $t \in E_1\ cupE_2$ gilt, daß $t \times t \in E_1 \times E_2$ ist, gilt offensichtlich auch $t \times t \in \sigma_=(E_1 \times E_2) = E_1 \bowtie_= E_2$, gilt statt der Teilmengenbeziehung sogar Gleichheit:

$$E_1 \bowtie_= E_2 = E_1 \cap E$$

Damit gilt:

$$E_1 \backslash E_2 =$$
$$E_1 \backslash (E_1 \cap E_2) =$$
$$E_1 \backslash (E_1 \bowtie_= E_2)$$

w kann als direkter Nachfolgeknoten von w' nach Konstruktion des Strukturgraphen in Definition 3.8 nur folgende Knotentypen haben:

- Für $w.type$ = Or gibt es nichts zu zeigen (mit gleicher Begründung wie bei Fall 1).

- Für $w.type$ = Func, Fact:

  (siehe Abbildung 45).

  Operatorbaum: $E_1 \backslash \sigma_f(E_2)$.
  Datenflußgraph: $E_1 \bowtie_= (E_1 \backslash (E_1 \bowtie_= E_2))$.

  E_1 und E_2 müssen die gleichen Attribute besitzen.

$$E_1 \backslash \sigma_f(E_2) =$$
$$E_1 \backslash (E_1 \cap \sigma_g(E_2)) =$$
$$E_1 \backslash (E_1 \bowtie_= \sigma_g(E_2)) =$$
$$E_1 \bowtie_= (E_1 \backslash (E_1 \bowtie_= E_2))$$

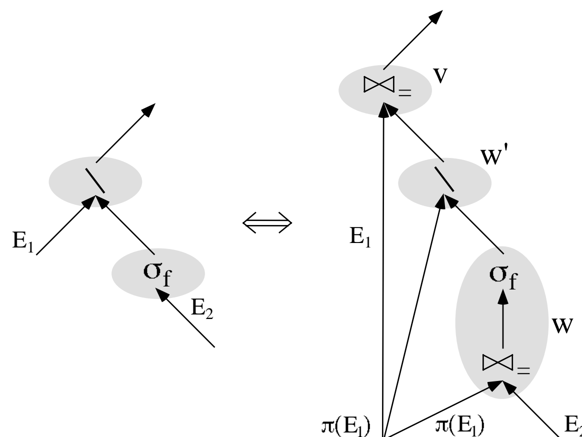


Abbildung 45: Operatorbaum und Datenflußgraph mit Join

- Für $w.type$ = AND, RECDOWN:
  Operatorbaum: $E_1 \backslash (E_2 \bowtie_= E_3)$.
  Datenflußgraph: $E_1 \bowtie_= (E_1 \backslash (E_1 \bowtie_= (E_2 \bowtie_= E_3))$.
  Die Erweiterung von $E_2 \bowtie_= E_3$ auf $E_3 \bowtie_= ... \bowtie_= E_n$ läßt sich analog beweisen.

  E_1 und $(E_2 \bowtie_= E_3)$ müssen die gleichen Attribute besitzen.

$$\begin{aligned}
&E_1 \backslash (E_2 \bowtie_= E_3) = \\
&E_1 \backslash (E_1 \cap (E_2 \bowtie_= E_3)) = \\
&E_1 \backslash (E_1 \bowtie_= (E_2 \bowtie_= E_3)) = \\
&E_1 \bowtie_= (E_1 \backslash (E_1 \bowtie_= (E_2 \bowtie_= E_3))
\end{aligned}$$

2b) w ist nicht direkter Nachfolgeknoten von w', d.h. der Pfad von w' nach w hat eine Länge > 1.

Dies läßt sich sich analog durch Aneinanderreihung der Schritte 1 und 2a) beweisen.

Satz 3.72

Die Bottom-Up Evaluation mittels relationaler Ausdrücke von Logikprogrammen ist äquivalent zur Datenflußgraph-basierten Auswertung. ◇

Beweis:

Die Äquivalenz der Bottom-Up Evaluation mittels relationaler Ausdrücke von Logikprogrammen zur Datenflußgraph-basierten Auswertung für Basisprogramme gilt nach Satz 3.67.

Die Äquivalenz der Ausführung für die Situationen, in denen die strukturelle Äquivalenz aufgrund der vorgenommenen Transformationen des Graphen nicht mehr gilt, gilt nach Satz 3.71.

Die Funktionalität der im Gegensatz zu Basisprogrammen zusätzlichen Knoten ist per Definition korrekt. ◇

Für die Korrektheit des Verfahrens bezüglich Rekursionen sei auf Kapitel 3.4.4.2 verwiesen, da es keine Ausdrucksform für Rekursionen in relationaler Algebra gibt.

Bemerkung:

Es gilt also für die Vergabe der globalen Parametermodi (Kapitel 3.3.4): wenn ein Parameter unbekannten Modus hat, kann, unter der Voraussetzung daß es mindestens ein Vorkommen dieses Parameters mit Modus Ausgabe gibt, der Parameter sowohl den Modus Eingabe als auch den Modus Ausgabe haben, ohne das Berechnungsergebnis des Graphen (d.h. die Ausgabe) zu verändern.

Folgende Folgerungen lassen sich daraus ziehen und wurden in Kapitel 3.3.4 verwendet:

i) Haben alle Verwendungsstellen eines Parameters noch unbekannten Modus, so ist die Wahl einer Ausgabestelle beliebig.

ii) Gibt es bereits eine Stelle mit Modus Ausgabe, können die anderen Vorkommen je nach Bedarf den Modus Eingabe oder Ausgabe erhalten.

iii) Wird durch einen über dem aktuellen Knoten liegenden OR-Knoten eine Änderung des Modus von Ausgabe auf Eingabe erzwungen (siehe Vergabe der Modi bei OR-Knoten), so ist dies zulässig und erzeugt keinen Fehler: bei einer Eingabe unter einem OR-Knoten muß es eine Verwendungsstelle des Parameters mit Modus Ausgabe geben, die mit dieser nicht über einen OR-Knoten verbunden ist; diese ist daher auch eine Ausgabestelle zur neu entstandenen Eingabestelle. ◇

4 Das Berechnungsmodell

Da sich der in dieser Arbeit erzeugte Datenflußgraph in wesentlichen Punkten von den aus der Literatur bekannten Datenflußgraphen unterscheidet (siehe Kapitel 3.4.6), muß ein neues Berechnungsmodell erstellt werden.

Das Berechnungsmodell wird in Abschnitt 4.1 zunächst in seiner grundlegenden Form, der datenflußgetriebenen Ausführung, erläutert. Anschließend wird die optimalere Form der ereignisflußgetriebenen Ausführung in Abschnitt 4.2 vorgestellt. Eine weitere Optimierung, die Mehrschichten-Ausführung, wird in Abschnitt 4.3 diskutiert.

Im Hinblick auf die Steuerung autonomer Systeme ist für das Berechnungsmodell festzulegen, daß die im Datenspeicher befindlichen Eingabedaten, die sich fortwährend durch Berechnungen anderer Module des Steuerungssystem verändern können, nur zu Beginn einer Ausführung eingelesen werden. Die kontinuierlich stattfindenden Änderungen werden daher nur von Ausführung zu Ausführung relevant, sie beeinflussen nicht die Ausführung selbst (siehe Kapitel 1.1.2.1 und 1.1.2.2). Sie müssen daher nicht während einer Ausführung berücksichtigt werden, wie das im Bereich der deduktiven Datenbanken bei Database Updates geschehen muß [Chen 91, Warren 84, Winslett 88].

Für eine Berechnung im Datenflußmodus werden errechnete (Zwischen-)ergebnisse von Ausführung zu Ausführung gelöscht. Damit kann eine Ausführung eine nachfolgende nicht beeinflussen, sie sind völlig unabhängig voneinander. Im Ereignisflußmodus bleiben errechnete Zwischenergebnisse zunächst erhalten. Tritt jedoch an einem Knoten ein Änderung auf, so werden alle Zwischenergebnisse dieses Knotens gelöscht und neu berechnet. In Abschnitt 4.2.4 wird erläutert, warum eine teilweise Löschung und Neuberechnung auf Basis des alten Zwischenergebnisses im Ereignisflußmodus nicht effizient ist und daher nicht durchgeführt wird.

Das Ein-Ausgabe-Verhalten ist in Abbildung 46 veranschaulicht.

Die Anfragen an das System, d.h. die zu berechnenden Ausgabedaten an andere Module des Steuerungssystems, bleiben über die gesamte Lebenszeit und damit von Ausführung zu Ausführung konstant (siehe Kapitel 1.1.2.3).

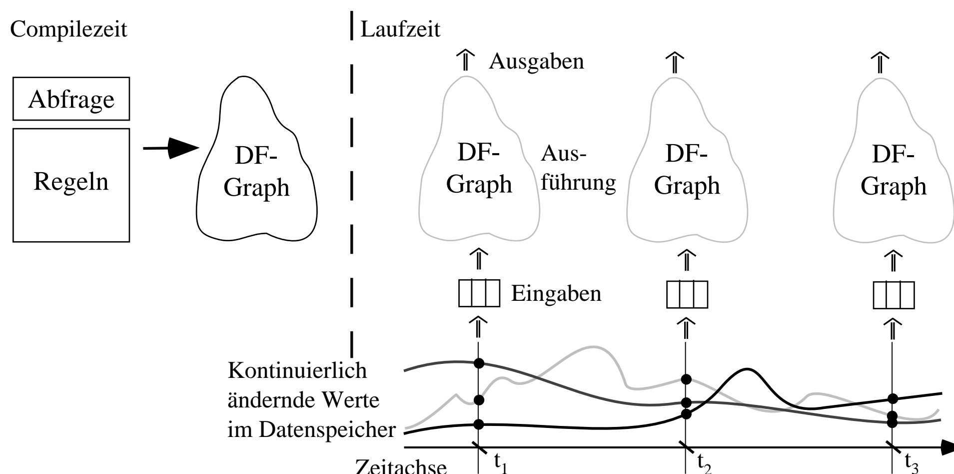


Abbildung 46: Input-Output Verhalten des Berechnungsmodells

4.1 Datengetriebene Ausführung

Um eine genaue Spezifikation der datengetriebenen Ausführung zu ermöglichen, sind
- Struktur und Funktionsweise des Datentransports,
- Struktur und Funktionsweise der Operationsausführung und
- eine geeignete Ausführungsvorschrift
zu beschreiben.

4.1.1 Transport von Daten

Token definieren Behälter für die Datenobjekte, die von Knoten zu Knoten transportiert werden sollen.

In bisherigen Implementierungen von Datenflußgraphen kann für jeden Eingabeparameter ein Eingabedatum durch genau ein Token ankommen. Ist für jeden Eingabeparameter ein Token angekommen, kann der Knoten zur Ausführung gebracht werden.

Diese Methode, Berechnungen anzustoßen, ist in der vorliegenden Implementierung aufgrund der flexiblen Anzahl von Eingabewerten pro Parameter nicht möglich. Da die Daten, die auf der gleichen Eingangskante ankommen, jedoch vom gleichen Vorgängerknoten stammen und aufgrund der Ausführungsvorschrift des Datenflußgraphen gleichzeitig berechnet und weitergegeben werden, können sie in einem *Tokenpaket* zusammengefaßt werden. Ein Tokenpaket besteht dabei aus einer Menge einzelner Token. Bestimmend für die Ausführbarkeit eines Knotens ist nun nicht mehr das Vorhandensein von Token auf jeder Eingangskante, sondern das Vorhandensein von (vollständigen) Tokenpaketen auf je einer der Eingangskanten.

Bei der Ausführung einer Knotenoperation werden alle Tokenpakete von den Eingangskanten entfernt. Die Ergebnisse werden in Form neuer Tokenpakete auf den Ausgangskanten plaziert. Damit stehen sie automatisch als Eingaben für Nachfolgeknoten zur Verfügung.

Ein Token enthält:

i) *eine Zieladresse*
Die Zieladresse ist die Adresse eines Speicherbereichs, der dem Zielknoten für die Aufnahme von Operanden zugewiesen worden ist. Zusätzlich enthält die Zieladresse die Nummer des Eingabeparameters, für den das Datum bestimmt ist.

ii) *Datum*
Das Datum ist der eigentliche Wert des Tokens.

iii) *eine Kontextkennzeichnung*
Ein Kontextkennzeichnung ist eine Zusatzinformation zur Zieladresse. Sie stellt eine Kennzeichnung des Tokens dar, um verschiedene Token als zur gleichen Instanz eines Teilgraphen und zur gleichen Iterationstiefe gehörig zu markieren. Dies ist für die korrekte Realisierung einer Rekursion notwendig.

iv) *einen Identifikator*
Der Identifikator ist eine zusätzliche Markierung, um Werte verschiedener Token als zusammengehörig zu markieren.

Ein Tokenpaket setzt sich zusammen aus einem Rahmen, der Beginn und Ende des Pakets kennzeichnet, und dem Inhalt, bestehend aus einem oder mehreren Token.

Definition 4.1 *Token*

Sei TK die Menge aller Token.

Ein Token $tk \in TK$ sei definiert als $TK = (ZA, \mathcal{X}, \kappa, \beta)$, wobei

i) *ZA= <knr, pnr>* sei eine Zieladresse, wobei *knr* die Nummer des Zielknotens, *pnr* die Nummer des Zielparameters am Zielknoten ist.

ii) $\mathcal{X} \in$ DATA sei ein Element aus einer (unendlichen) Menge von Werten aus gegebenen, zulässigen Wertebereichen (Zeichen, Zeichenketten, ganze Zahlen, Fließkommazahlen, boolsche Werte).

iii) κ sei eine Kontextkennzeichnung.

iv) β sei eine eindeutige Wertekennzeichnung. ◇

Definition 4.2 *Tokenpaket*

Sei $\mathcal{T} = (tk_1, ..., tk_n)$, $n \in \mathbb{N}$ ein Tokenpaket. ◇

Definition 4.3 *Zuordnung von Token zu Kanten*

Sei $DG^R_{\mathcal{L}} = (V^R_{DG}, E^R_{DG}, Fun^R_{DG})$ ein Datenflußgraph mit Zyklen.

$\pi : E_{KP} \rightarrow TK$ sei eine Funktion, die den Kanten $e \in E^R_{DG}$ Token zuordnet:

$$\pi(e) = \begin{cases} \emptyset & \text{wenn auf einer Kante } e \in E_{KP} \text{ kein Token ist.} \\ (\text{tk}_1, ..., \text{tk}_n) & \text{wenn auf der Kante } e \in E_{KP} \\ & \text{das Tokenpaket } \mathcal{T} = (\text{tk}_1, ..., \text{tk}_n) \text{ ist.} \end{cases}$$

Es gelte dann für eine Knoten $v \in V^R_{DG}$:

- $\pi(e_{v.ip_{i,j}})$ bezeichne das Tokenpaket $\mathcal{T}$ auf der Eingangskante j des Parameters $v.ip_i$ (wobei $i \in \{1, .., v.in\}$, $j \in \{1, .., v.ip_i.in\}$).
- $\pi(e_{v.op_{i,j}})$ bezeichne das Tokenpaket $\mathcal{T}$ auf der Ausgangskante j des Parameters $v.ip_i$ (wobei $i \in \{1, .., v.out\}$, $j \in \{1, .., v.op_i.in\}$).
- $\pi_k(e_{v.ip_{i,j}})$ bezeichne das Token tk_k, $k \in \mathbb{N}$, auf der Eingangskante j des Parameters $v.ip_i$ (wobei $i \in \{1, .., v.in\}$, $j \in \{1, .., v.ip_i.in\}$).
- $\pi_k(e_{v.op_{i,j}})$ bezeichne das Token tk_k, $k \in \mathbb{N}$, auf der Ausgangskante j des Parameters $v.ip_i$ (wobei $i \in \{1, .., v.out\}$, $j \in \{1, .., v.op_i.in\}$). ◇

Definition 4.4 *Erweiterung des Begriffs "Wertetupel"*

Ein Wertetupel $(m_1, ..., m_k)$, $k \in \mathbb{N}$ werde im folgenden repräsentiert durch $(tk_1, ..., tk_n)$, wobei $tk_i \in TK$ für $i \in \{1, ..., k\}$ und $tk_i.\mathcal{X} = m_i$. ◇

4.1.1.1 Daten

Für das Datum eines Tokens $\mathcal{X}$ gilt: $\mathcal{X} \in$ DATA.

Die Menge der zulässigen Daten DATA ist weder typ- noch bereichsbeschränkt. Der Typ eines Datums kann ein Zeichen, eine Zeichenkette, eine ganze Zahl, eine Fließkommazahl, oder ein boolscher Wert sein.

Im weiteren sollen jedoch folgende Randbedingungen gelten:

1. Spezifizieren Systemprädikate Operationen über Daten, so sind durch ihre Definition zulässige Typen für die zu verarbeitenden Daten vorgegeben. Entspricht ein Datum nicht einem zulässigen Typ, so scheitert das Systemprädikat per Definition.

2. Kommen an einem beliebigen Knoten für einen Parameter mehrere Werte an, oder müssen ankommende Werte mit einer Konstanten des Knoten gematcht werden, so müssen diese den gleichen Typen haben. Ist dies nicht der Fall, so scheitert die Knotenoperation per Definition.

4.1.1.2 Kontextkennzeichnung

Nach Kapitel 3.4.4 wird unter der Instanz einer allgemeinen linearen Rekursion diejenige Iterationstiefe einer absteigenden und ihrer zugehörigen aufsteigenden Rekursionsschleife gesehen, die über den gleichen Weg erreicht wird.

Für die Realisierung von allgemeinen, linearen Rekursionen (d.h. Rekursionen mit mehreren, alternativen Rekursionsschleifen) müssen die Token eine eindeutige Kennzeichnung erhalten, um sie

- als gemeinsam zur gleichen Instanz einer Rekursionsschleife zugehörig erkennen zu können und
- um sie beim Durchlauf durch die aufsteigenden Rekursionsschleifen derjenigen Rekursionsschleife übergeben zu können, deren Instanz beim Durchlauf über die absteigenden Schleifen in der gleichen Iterationstiefe aktiviert war.

Zur Erzeugung der Kennzeichnung und Rekonstruktion des Wegs eignet sich ein Schema, das im *U-Interpreter* (siehe Kapitel 2.2.2.2) verwendet wird.

Angelehnt an dieses Schema ist eine Kontextkennzeichnung wie folgt definiert:

Definition 4.5 *Kontextkennzeichnung*

$\kappa = < u, b, i >$ sei eine Kontextkennzeichnung,

- u ist das Kontextfeld,
- b ist ein Blockname (z.B. eindeutige Bezeichnung der Rekursionsschleife, bzw. des umgebenden Codes),
- i ist die Iterationsnummer und enthält die Anzahl der Schleifeniterationen. ◇

Gibt es mindestens eine absteigende und eine aufsteigende Rekursionsschleife (im folgenden als vollständige Rekursion bezeichnet), so muß beim Eintritt in eine absteigende Rekursionsschleife eine Operation R durchgeführt werden, die einen Kontextwechsel vornimmt, beim Austritt aus einer ausfsteigenden Rekursionsschleife dagegen eine Operation R^{-1}, die den alten Kontext wiederherstellt.

Definition 4.6 *Sichern und Restaurieren der Kontextkennzeichnung für eine vollständige Rekursion*

Sei $< u, b, i >$ eine Kontextkennzeichnung.

i) Sichern der Kontextkennzeichnung:
$R(<u, b, i>) = <u', b', i+1>$, wobei $u' = <u, b>$, b' sei ein neuer Kontext

ii) Restaurieren der Kontextkennzeichnung:
$R^{-1}(<u', b', i>) = <u, b, i-1>$, wobei $u' = <u, b>$ ◇

Im vorliegenden Modell wird die Operation R durch den Knoten v_d mit $v_d.type =$ RecDown und R^{-1} durch den Knoten v_u mit $v_u.type =$ RecUp realisiert: $R^d := R, R^u := R^{-1}$.

Satz 4.7

Durch die Kontextkennzeichnung ist eindeutig eine Instanz einer Rekursionsschleife bezeichnet. ◇

Beweis:

Seien $tk_r.\kappa = <u, b, i>$, $tk_s.\kappa = <u', b', i'>$

i) Annahme: zwei Token befinden sich in der gleichen Schleife in verschiedenen Iterationstiefen. Dann gilt: $u = u'$, $b = b'$, aber $i \neq i'$ und damit gilt auch: $tk_r.\kappa \neq tk_s.\kappa$.

ii) Annahme: zwei Token befinden sich in der gleichen Iterationstiefe, aber in verschiedenen Schleifen. Dann gilt: $u = u'$, $i = i'$, aber $b \neq b'$ und damit gilt auch: $tk_r.\kappa \neq tk_s.\kappa$.

iii) Annahme: zwei Token befinden sich in der gleichen Iterationstiefe, und in der gleichen Schleife, d.h. es gilt $b = b'$, $i = i'$, haben diese aber über verschiedene Wege erreicht. Dann muß es eine Iterationstiefe $j < i$ gegeben haben, für die galt: $tk_r.\kappa = <u^1, b^1, j>$, $tk_s.\kappa = <u^2, b^2, j>$ mit $b^1 \neq b^2$. Damit gilt für jede nachfolgende Iterationstiefe: $u^1_{neu} := <u^1_{alt}, b^1_{alt}>, u^2_{neu} := <u^2_{alt}, b^2_{alt}>$ und damit gilt: $u^1_{neu} \neq u^2_{neu}$ für alle Iterationstiefen j' mit $j < j' \leq i$. Damit gilt auch $tk_r.\kappa \neq tk_s.\kappa$ für Iterationstiefe i.

Für tk_r und tk_1 mit $tk_1.\kappa = tk_s.\kappa$ muß daher gelten, daß sie sich in der gleichen Schleife mit der gleichen Iterationstiefe befinden und daß sie diese auf dem gleichen Weg erreicht haben. Dies entspricht der Definition der Instanz einer Rekursionsschleife. ◇

Durch die Angabe der Kontextkennzeichnung sind allgemeine lineare Rekursionen realisierbar. Bei diesen muß es nicht nur für jede Rekursionstiefe, sondern auch für dieselbe Rekursionstiefe mehrere Instanzen der Rekursionschleife geben (siehe Kapitel 3.4.4), damit Token der gleichen Iterationstiefe, die über verschiedene Schleifenwege gekommen sind, unterscheidbar sind.

Satz 4.8

In jeder Iterationstiefe ist die Weitergabe von Token an eine aufsteigende Rekursionsschleife eindeutig durch die in der gleichen Iterationstiefe durchlaufene absteigende Rekursionsschleife gegeben. ◇

Beweis:

Beweis durch Induktion über die Iterationstiefe (und damit Anzahl der durchgeführten Operationen R):

1. *Induktionsanfang:* Iterationstiefe $i = 1$

   R und damit auch R^{-1} wird einmal ausgeführt ($\perp$ sei das leere Kontextfeld, b, b' Blockbezeichnungen): $R(<\perp, b, 0>) = <<\perp, b>, b', 1>$
   Für R^{-1} gilt damit: $R^{-1}(<<\perp, b>, b', 1>) = <\perp, b, 0>$
   Die Rückkehr erfolgt daher genau zum Ausgangskontext.

2. *Induktionsschritt:* Iterationstiefe $i > 1$

   (u, u' seine Kontextfelder, b, b' Blockbezeichnungen)
   für die i-te Ausführung von R gilt, wenn die Kontextkennzeichnung in der (i-1)-ten Iterationstiefe $<u, b, i-1>$ war: $R(<u, b, i-1>) = <<u, b>, b', i>$
   Für das entsprechende R^{-1} zur Kontextkennzeichnung in der i-ten Rekursionstiefe $<<u, b>, b', i>$ gilt damit: $R^{-1}(<<u, b>, b', i>) = <u, b, i-1>$,
   d.h. man erhält wieder den Kontext der (i-1)-ten Iterationstiefe.
   Deswegen gilt die Behauptung für beliebige Iterationstiefen. ◇

Durch das iterative Abspeichern von u, b in u bei jedem Eintritt in eine absteigende Rekursionsschleife und durch das Extrahieren von u, b aus u bei jedem Austritt aus einer aufsteigenden Rekursionsschleife, kann die Weitergabe an diejenige aufsteigende Rekursionsschleife gehen, die zur gleichen Instanz gehört, wie die absteigende Rekursionsschleife, die bei gleicher Iterationstiefe durchlaufen wurde.

Bemerkung:

> Durch die Realisierung der Kontextkennzeichnung als dynamische Datenstruktur kann die Kontextkennzeichnung sehr groß werden.
>
> Um zu vermeiden, daß Token mit einer sehr großen Kontextkennzeichnung zu einer hohen Kommunikationslast führen, läßt sich das in der Manchester Dataflow Machine verwendete Verfahren einsetzen. In diesem Verfahren können die Verwaltungsknoten direkt miteinander kommunizieren, so daß es möglich ist, große Teile der Kontextkennzeichnung direkt vom RecDown-Knoten zum RecUp-Knoten zu schicken. In den Token selbst muß sich dann nur jeweils der aktuelle Teil der Kontextkennzeichnung befinden. ◇

Gibt es bei einer Rekursion nur aufsteigende Schleifen, (im folgenden als aufsteigende Rekursion bezeichnet), so ist eine Kontextmarkierung nur notwendig, um Token als gemeinsam zur gleichen Instanz einer Rekursionsschleife zugehörig zu markieren.

Man benötigt daher eine leicht modifizierte Funktion R.

Definition 4.9 *Kontextwechsel für aufsteigende Rekursionen*

$R(<u, b, i>) = <u', b', i'>$, wobei

$u' = u, b$ und

$$i' = \begin{cases} i+1 & \text{für eine Rekursionsfortsetzung} \\ 0 & \text{für eine Rekursionsterminierung} \end{cases}$$ ◇

Die Operation R muß dann v_u mit $v_u.type$ = RecUp zugewiesen werden: $R^u := R$

Bemerkung:

> Die Kontextkennzeichnung kann über die Angabe einer Adresse implementiert werden, um das im Explicite Token Store von Monsoon realisierte Token Matching zu ermöglichen.
>
> Der Blockname und die Iterationstiefe gibt dabei den entsprechenden Speicherrahmen an, in dem der korrekte Platz für die Eintragung der Operanden über Knotenadresse und Operandennummer gefunden werden kann. ◇

4.1.1.3 Identifikatoren

Im Datenflußgraph dieser Arbeit können Werte für verschiedene Ausgabeparameter berechnet werden. Die Werte für einen Ausgabeparameter können, wenn mehrere Ausgabekanten für den Parameter existieren, in Form von Token an unterschiedliche Nachfolgeknoten gehen. Ebenso können die Werte verschiedener Ausgabeparameter an unterschiedliche Nachfolgeknoten gehen. Ein Knoten erzeugt dabei für jede Ausgabekante ein eigenes Token, bzw. im Fall von mehreren Werten, pro Ausgabekante eine Menge von Token. Die Anzahl der Token, die pro Ausgabekante ausgegeben werden, ist dabei für alle Parameter eines Knotens aufgrund der in Kapitel 3.6 angegebenen Funktionalität gleich.

Als *Wertetupel* werden im folgenden angelehnt an Definition 3.64 bezeichnet:

- *Eingabetupel*: alle *Token*, deren Werte gemeinsam als eine Eingabe für einen Knoten verarbeitet werden.
- *Ausgabetupel*: alle *Token*, deren Werte durch die Knotenoperation über einem Eingabetupel erzeugt wurden.

Gibt man Werte, die an einem Knoten gemeinsam als Ausgabetupel durch eine Knotenoperation über einem Eingabetupel erzeugt wurden, in Token getrennt an verschiedene Nachfolgeknoten weiter, so verliert man ohne eine zusätzliche Kennzeichnung die Verbindung zwischen den zusammengehörigen Werten verschiedener Parameter eines Knotens. Diese Zusammengehörigkeit ist jedoch von zentraler Bedeutung, wie folgendes Beispiel verdeutlicht.

Beispiel:

> Geben mehrere IFACT-Knoten, die Unterknoten eines gemeinsamen OR-Knotens sind, jeweils einen Wert für die Parameter X und Y aus, so gibt der OR-Knoten für X und Y jeweils mehrere Werte weiter. Welcher Wert von Y aber einem Wert von X gehört, wird jedoch wichtig, wenn der Join am darüberliegenden AND-Knoten für einen der Werte von X scheitert, wie es im Beispielgraphen für $X = 14$ der Fall ist. Dann muß sowohl der Wert von X, als auch der dazugehörige Wert von Y, 2, aus der Menge der gültigen Werte entfernt werden. Figur 47 veranschaulicht das Beispiel.
>
> Da die Berechnung der zusammengehörigen Werte immer gemeinsam erfolgt, könnte die Verbindung über die Reihenfolge der Werte hergestellt werden: in der Ausgabe des OR-Knotens gehört der erste Wert für X zum ersten Wert für Y, der zweite Wert von X zum zweiten Wert von Y.
>
> Nun kann jedoch auch der Fall auftreten, daß ein Wert für Z, der zu einem Wert für X gehört, als Eingabe für einen weiteren Knoten dient. Dieser kann dann Werte für V und W produzieren, wie in Figur 48 gezeigt: mit $Z = 6$ als Eingabe werden für V und W als Ausgabe von $h(V, W)$ die Tupel (4,8) und (2,4) erzeugt.
>
> Aus jedem Wert für Z können offensichtlich mehrere Werte für V und W entstehen. Diese müssen nun einerseits eine Kennzeichnung besitzen, so daß die Werte untereinander unterscheidbar sind. Gleichzeitig müssen sie aber auch eine Kennzeichnung besitzen, die von Z abstammt. Wird ein Wert von X nachträglich gelöscht, so muß der zugehörige Wert von Z gelöscht werden und damit auch die aus ihm erzeugten Werte von V und W. Umgekehrt müssen, da für $Z = 9$ an $h(V, W)$ kein erfolgreicher Join durchgeführt werden konnte, die zugehörigen Werte $X = 28$ und $Y = 4$ gelöscht werden.

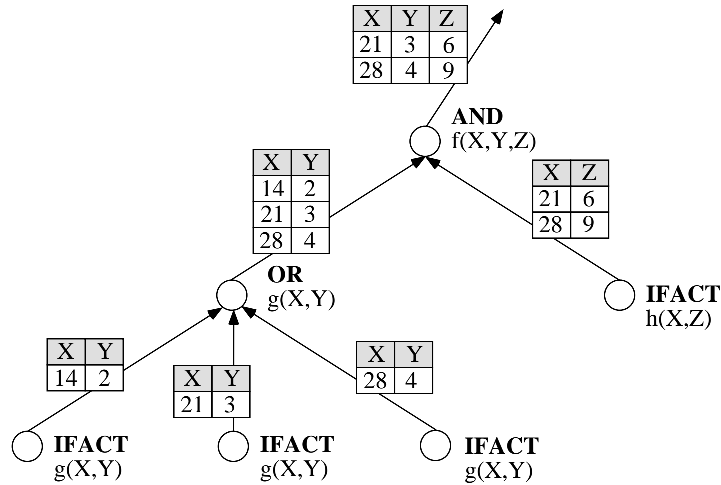


Abbildung 47: Problemstellung: Wertekennzeichnung

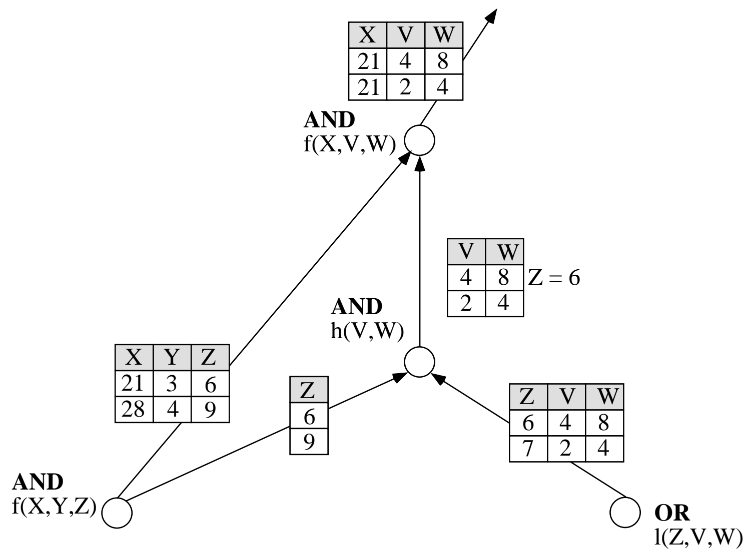


Abbildung 48: Problemstellung: Wertekennzeichnung

> Bei der Konstruktion von Eingabetupeln am obersten AND-Knoten muß der verbleibende Wert für X dann mit beiden Eingabetupeln von V und W zusammengefügt werden. Er muß also entsprechend vervielfältigt werden. ⋄

Die Notwendigkeit einer Wertekennzeichnung wird durch folgende Gründe erzwungen:

- Einerseits muß an einem Knoten für die Durchführung der Knotenoperation bekannt sein, ob zwei Werte aus dem selben Ausgabetupel stammen und daher zusammengehören, oder ob sie unabhängig voneinander sind.
- Andererseits führt die Parallelität des Graphen dazu, daß sich auf Kanten alternative Berechnungsergebnisse befinden können. Alternative Berechnungsergebnisse tragen zu alternativen Lösungen der Lösungsmenge bei, jedoch nicht zur gleichen Lösung (Dies gilt nach Definition der deklarativen Semantik von Logikprogrammen [Sterling 86]). Alternative Werte dürfen also offensichtlich nicht als gemeinsame Eingabe für eine Berechnung verwendet werden.

Abgeleitet daraus läßt sich abstrakt folgende Beziehung zwischen Werten, bzw. zwischen den Token, die die Werte enthalten, definieren.

Definition 4.10 *Relation zwischen Token*

Für zwei Token $tk, tk' \in TK$ gelte:

- $tk \vdash tk'$, wenn tk' aus tk abgeleitet wurde.
  $tk \dashv\vdash tk'$ stehe dann für die Aussage $\exists\ tk''$, so daß $tk'' \vdash tk$, $tk'' \vdash tk'$, d.h. wenn tk und tk' aus dem gleichen Ausgabetupel stammen, bzw. von Token aus dem gleichen Ausgabetupel abgeleitet wurden.
- $tk \perp tk'$, wenn tk und tk' Token mit alternativen Werten darstellen.

Bemerkung: Für zwei Token tk und tk' kann auch gelten: $tk \vdash tk'$ und $tk \dashv\vdash tk'$.

Forderung: Ein Knoten muß gewährleisten,

- daß $tk_{in} \vdash tk_{out}$, wenn der Wert von tk_{in} zur Erzeugung des Wertes von tk_{out} benutzt wird.
- für ein Token tk muß gelten:
  - $tk \dashv\vdash tk_{in} \Rightarrow tk \dashv\vdash tk_{out}$
  - $tk \perp tk_{in} \Rightarrow tk \perp tk_{out}$
  - $\neg\ (tk \dashv\vdash tk_{in}) \wedge \neg\ (tk \perp tk_{in}) \Rightarrow \neg\ (tk \dashv\vdash tk_{out}) \wedge \neg\ (tk \perp tk_{out})$ ◇

Satz 4.11

Befinden sich zwei verschiedene Token $tk, tk' \in TK$ auf einer Kante im Datenflußgraph, so gilt: $tk \perp tk'$. ◇

Beweis:

Ein IFact-Knoten erzeugt per Definition nur ein Wertetupel, d.h. auf jeder seiner Ausgangskanten befindet sich genau ein Token.

Ein Or-Knoten faßt die Ergebnisse verschiedener, alternativer Berechnungswege zusammen. Dadurch können mehrere, alternative Token auf jeder Ausgangskante entstehen.

Ein RecUp-Knoten hat keine eigentliche Funktionalität.

Alle anderen Knoten führen einen Join über den Eingaben durch. Dabei gilt aufgrund der Definition des Kreuzproduktes, daß, wenn auf jeder Kante nur ein Token ankommt, nur ein Ergebnis erzeugt werden kann. Kommen mehrere, alternative Token auf einer Eingangskante an, können daraus mehrere Ergebnisse erzeugt werden. Diese sind laut der Forderung in Definition 4.10 dann ebenfalls alternativ. ◇

Satz 4.12

Die Eingangskanten eines Knotens lassen sich in disjunkte Teilmengen aufteilen, wobei in jeder Teilmenge die Kanten zusammengefaßt sind, über die Token $tk, tk' \in TK$ kommen können, für die gilt: $tk \dashv\vdash tk'$ oder $tk \perp tk'$. Für Token, die über Kanten kommen, die verschiedenen Teilmengen zugeordnet sind, gilt dagegen grundsätzlich weder $tk \dashv\vdash tk'$ noch $tk \perp tk'$. ◇

Beweis:

Diese Aufteilung ist möglich, da die Datenpfade im Graphen eindeutig sind: alle Ausgaben eines Knoten gehen über die gleichen Kanten. Für die Ausgaben eines Knotens gelten die Relationen $\dashv\vdash$ (gleiches Ausgabetupel) oder $\perp$ (alternative Ausgabetupel) nach Satz 4.11. Damit gilt für alle Token *tk, tk'* auf diesen Kanten: $tk \dashv\vdash tk'$ oder $tk \perp tk'$. Handelt es sich dagegen um Kanten, die nicht von einem gemeinsamen Vorgängerknoten erreicht werden können, so kann weder $tk \dashv\vdash tk'$ noch $tk \perp tk'$ gelten. $\diamond$

Mit Hilfe der Relationen $\dashv\vdash$ und $\perp$ lassen sich nun Aussagen über die Konstruktion der Mengen von Eingabetupeln an Knoten treffen.

Definition 4.13 *Eingabetupel für Knoten*

Für die Tupelmengen $M_i \in \mathcal{M}$, $i \in \mathbb{N}$ mit $M_i = \{(tk^r_{i,1}, ..., tk^r_{i,k_i}) \mid k_i, u_i \in \mathbb{N}, r \leq u_i\}$ als Eingabe für einen Knoten gelte (r bezeichnet das r-te Tupel der Eingabemenge M_i):

- $tk^r_{i,l} \dashv\vdash tk^r_{i,l'}$ für $l, l' \leq k_i$.
- $tk^r_{i,l} \dashv\vdash tk^{r'}_{i,l'} \vee tk^r_{i,l} \perp tk^{r'}_{i,l'}$ für $l, l' \leq k_i$, $r, r' \leq u_i$, $r \neq r'$.
- $\nexists i, j \in \mathbb{N}$: $tk^r_{i,l} \perp tk^{r'}_{j,l'}$ für $l \leq k_i$, $l' \leq k_j$, $r \leq u_i$, $r' \leq u_j$. $\diamond$

Folgerung 4.14

Für die Tupelmengen M_i, $i \in \mathbb{N}$ gilt folglich:

$$M_i = \{(tk^r_{i,1}, ..., tk^r_{i,k_i}) \mid k_i, u_i \in \mathbb{N}, r \leq u_i, tk^r_{i,l} \dashv\vdash tk^r_{i,l'} \text{ für } l, l' \leq k_i \text{ und } \nexists l, l' \in \mathbb{N}\ tk^r_{i,l} \perp tk^r_{i,l'} \text{ und } tk^r_{i,l}.\mathcal{X} = tk^r_{i,l'}.\mathcal{X}, \text{ wenn } \exists l, l' \in \mathbb{N} \text{ mit } tk^r_{i,l}.ZA = tk^r_{i,l'}.ZA\}$$

Bemerkung: Die Attribute der Tupel sind über die Zieladressen ZA gegeben. Eine Zieladresse besteht dabei aus der Bezeichnung des Zielknotens und der Bezeichnung des Eingabeparameters. Da für die Tupel gefordert wird, daß sie für den gleichen Parameter den gleichen Wert besitzen müssen, werden doppelte Spalten mit gleichem Attribut zu einer Spalte zusammengefaßt. $\diamond$

Satz 4.15

Die Definition der M_i nach Definition 4.13 erfüllt folgende in Satz 3.61 für Tupelmengen geforderte Eigenschaften:

i) alle Tupel haben die gleiche Anzahl von Elementen.

ii) alle Tupel haben die gleichen Attribute.

iii) ein Attribut kommt nicht mehr als einmal in einem Tupel vor. $\diamond$

Beweis:

i) und ii) gelten nach Satz 4.12: da die Kanten eindeutig in disjunkte Teilmengen aufteilbar sind (wobei für die Token *tk, tk'* auf Kanten einer Menge gilt: $tk \dashv\vdash tk'$ oder $tk \perp tk'$), ist gewährleistet, daß alle Tupel in einer Menge M_i die gleiche Anzahl von Elementen haben. Wählt man als Attribut die Bezeichnung der Eingangskante, bzw. die Zieladresse ZA, haben außerdem alle Tupel einer Menge M_i die gleichen Attribute.

iii) Dies ist durch das Zusammenfassen von Spalten mit gleichem Attribut in Folgerung 4.14 gewährleistet. $\diamond$

Für die Realisierung der Relationen $\dashv\vdash$ und $\perp$ benötigt man eine Wertekennzeichnung, die jedem Token zugeordnet wird.

Dafür eignet sich ein Bitvektor der Länge der Anzahl der benötigten Wertekennzeichnungen. Jede einzelne neu vergebene Wertekennzeichnung ist ein Bitvektor mit einem einzelnen, eindeutig gesetzten Bit. Kombiniert entstehen dann daraus Bitvektoren mit mehreren gesetzten Bits. Aus diesen sind wiederum die Einzelkennzeichnungen trivial ablesbar.

Definition 4.16 *Bitvektoren*

$\mathcal{K} = \wp(\{0,1\}^n)$ sei die Menge eindeutiger Bitvektoren (Wertekennzeichnungen).

$\mathcal{K}_s \subset \mathcal{K}$ sei eine Menge von Startkennzeichnungen:
$\mathcal{K}_s = \{\beta^e \mid \beta^e = (b_1, ..., b_n),\ n \in \mathbb{N}$, wobei $b_i = 1$ für ein $i \in \{1, ..., n\}$ und $b_j = 0$ für alle $j \in \{1, ..., n\}$ mit $j \neq i\} \cup \{\beta^0 \mid \beta^0 = (b_1, ..., b_n),\ n \in \mathbb{N}$, wobei $b_i = 0$ für alle $i \in \{1, ..., n\}\}$.
Der Bitvektoren β^0 repräsentiert dabei die leere Wertekennzeichnung. ◇

Definition 4.17 *Operationen auf Bitvektoren*

Seien $\beta_i, \beta_j \in \mathcal{K}$, $i, j \in \mathbb{N}$, $\beta_i = (b_{i,1}, ..., b_{i,n})$, $\beta_j = (b_{j,1}, ..., b_{j,n})$, $n \in \mathbb{N}$ Bitvektoren:

i) $\subset_b$ bezeichne eine bitweise Teilmengenbeziehung:
$\beta_i \subset_b \beta_j \Leftrightarrow (b_{i,l} = 1, l \in \{1, ..., n\}) \Rightarrow b_{j,l} = 1$

ii) $=_b$ sei über $\subset_b$ definiert:
$\beta_i =_b \beta_j \Leftrightarrow (\beta_i \subset_b \beta_j \ \wedge\ \beta_j \subset_b \beta_i)$

iii) $\cup_b$ bezeichne eine bitweise Vereinigung:
$\beta = \beta_i \cup_b \beta_j$ mit $\beta = (b_1, ..., b_n)$ und $b_l = 1, l \in \{1, ..., n\} \Leftrightarrow (b_{i,l} = 1 \vee b_{j,l} = 1)$

iv) $\cap_b$ bezeichne eine bitweise Konjunktion:
$\beta = \beta_i \cap_b \beta_j$ mit $\beta = (b_1, ..., b_n)$ und $b_l = 1, l \in \{1, ..., n\} \Leftrightarrow (b_{i,l} = 1 \wedge b_{j,l} = 1)$

v) $\backslash_b$ bezeichne eine bitweise Exklusion:
$\beta = \beta_i \backslash_b \beta_j$ mit $\beta = (b_1, ..., b_n)$ und $b_l = 1, l \in \{1, ..., n\} \Leftrightarrow (b_{i,l} = 1 \wedge b_{j,l} = 0)$ ◇

Eine Untersuchung der Funktionsweise der Knotentypen ermöglicht es, Aussagen über die Art der Wertekennzeichnungen und geeignete Methoden der Nutzung zu treffen.

Knotentyp	*Funktionsweise*	*Konsequenz*
IFACT FACT (nur Ausgaben) FUNC (nur Ausgaben)	Ein neues Wertetupel wird erzeugt.	Für das Wertetupel wird eine neue Wertekennzeichnung vergeben.
AND QUERY FACT (mit Eingaben) FUNC (mit Eingaben) RECDOWN	Ein neues Wertetupel entsteht aus einer Kombination (Join) ankommender Wertetupel.	Die Wertekennzeichnung für das neue Wertetupel muß eine Kombination der Wertekennzeichnungen der als Eingabe verwendeten Wertetupel sein.

Knotentyp	*Funktionsweise*	*Konsequenz*
OR RECUP	Die ankommenden Wertetupel werden zusammengefaßt, aber unverändert weitergegeben.	Für jedes Wertetupel kann die Wertekennzeichnung erhalten bleiben.
NOT	Ein neues Wertetupel entsteht aus einer Kombination (Join, Differenzmenge) ankommender Wertetupel.	Die Wertekennzeichnung für das neue Wertetupel muß eine Kombination der Wertekennzeichnungen der als Eingabetupel sein.

Für die Konstruktion der Bitvektoren gilt daher:

- Erzeugt ein Knoten, der keine Eingabetupel erhält, ein Ausgabetupel, so wird für dieses eine eindeutige Startkennzeichnung erzeugt. Alle Werte eines Ausgabetupels erhalten die gleiche Startkennzeichnung. Es gibt keine Knoten, die keine Eingaben und mehr als ein Ausgabetupel erzeugen.
- Gibt ein Knoten Eingabetupel als Ausgabetupel in gleicher Form weiter, so bleiben die zugehörigen Bitvektoren unverändert.
- Erzeugt ein Knoten, der Eingabetupel erhält, aus jeder eindeutigen Kombination im Sinne des Kreuzproduktes, bzw. Joins der Eingabetupel ein Ausgabetupel, so erhalten alle Werte des Ausgabetupels einen Bitvektor der eine Kombination der Bitvektoren der Eingangstupel darstellt. Damit bleibt offensichtlich durch welche Werte sie entstanden sind. Es gibt keine Knoten, die aus einer eindeutigen Kombination mehr als ein Ausgabetupel erzeugen (siehe Funktionalität der Knoten in Kapitel 3.6).

Da verschiedene Ergebnisse an inneren Knoten des Graphen nur durch unterschiedliche Kombinationen von Eingabetupeln erzeugt werden können, müssen Wertekennzeichnungen ausschließlich an den Stellen erzeugt werden, an denen Werte in das Programm (durch Auslesen aus einem Datenspeicher, oder durch Definition von Konstanten) eingegeben werden, statt aus anderen Werten generiert zu werden. Die maximale Anzahl der Wertekennzeichnungen ist folglich die Anzahl der Parameter (von IFACT-Knoten) über die Werte aus dem Datenspeicher ausgegeben werden, sowie die Anzahl von FACT- und FUNC-Knoten, bei denen es nur Ausgaben gibt. Sie kann bei der Erstellung des Graphen bestimmt werden. Eine detailierte Abschätzung erfolgt in Kapitel 5.2.1.

Die folgenden Abbildungen veranschaulichen die Situationen an einzelnen, ausgewählten Knoten: IFACT-Knoten (Abbildung 49), OR-Knoten (Abbildung 50), AND-Knoten (Abbildung 51) und NOT-Knoten (Abbildung 52)

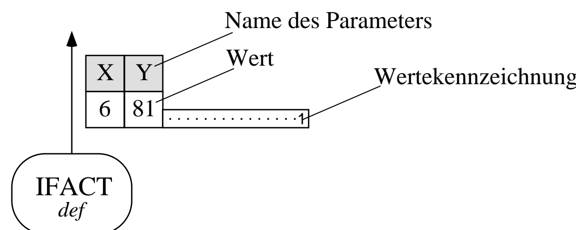


Abbildung 49: Wertekennzeichung an einem IFACT-Knoten

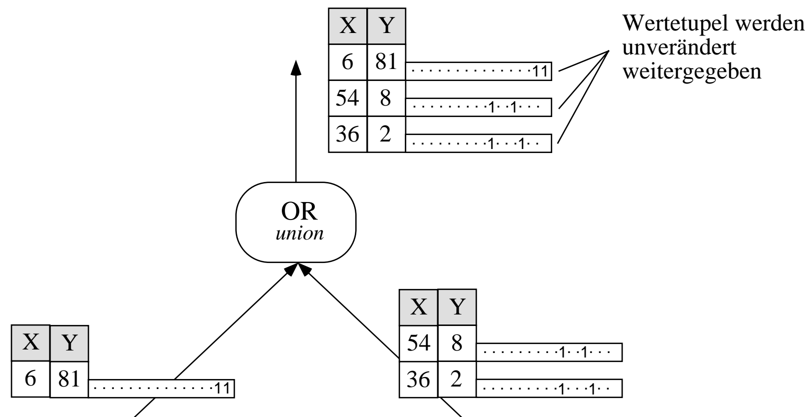


Abbildung 50: Wertekennzeichung an einem OR-Knoten

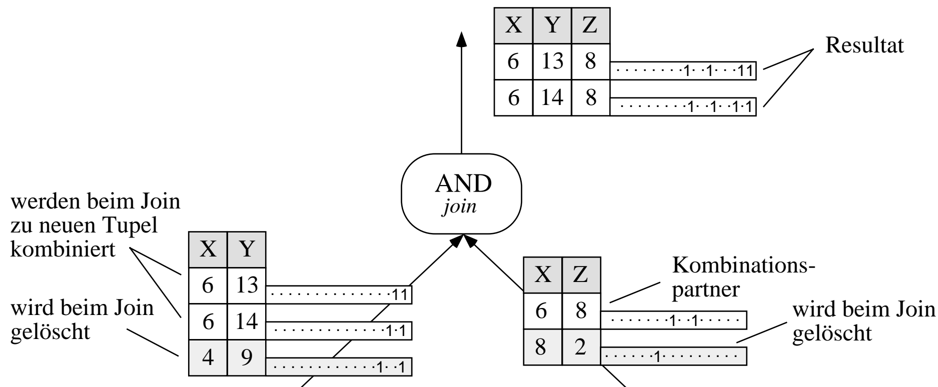


Abbildung 51: Wertekennzeichung an einem AND-Knoten

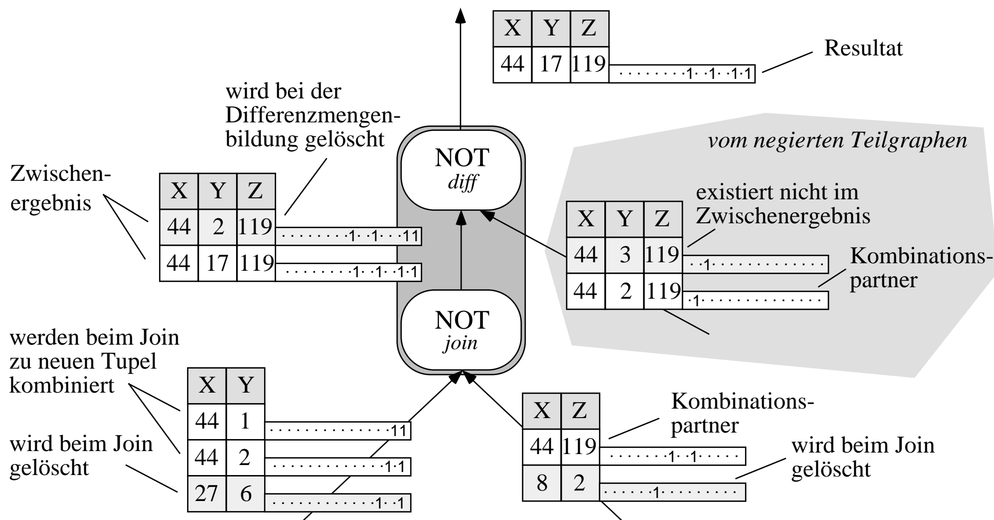


Abbildung 52: Wertekennzeichung an einem NOT-Knoten

Definition 4.18 *Konstruktion von Bitvektoren*

Sei $DG^R_{\mathcal{L}} = (V^R_{DG}, E^R_{DG}, Fun^R_{DG})$ ein Datenflußgraph mit Rekursionen. Sei $v \in V^R_{DG}$.

i) $v.in = 0$:

Sei M die Menge der Ausgabetupel von v. Für M gilt: $|M| = 1$, d.h. $M = (tk_1, .., tk_{v.out})$.

Dann gelte für alle $j \in \{1,...,v.out\}$: $tk_j.\beta = \beta_v$, mit $\beta_v \in \mathcal{K}_s$ und $\beta_v \neq \beta_w$ für $w \in V^R_{DG}$, $v \neq w$.

ii) $v.in > 0$, $v.type$ = AND, $v.type$ = QUERY, $v.type$ = FACT, $v.type$ = FUNC, $v.type$ = RECDOWN, $v.type$ = NOT:

Seien $M_1, ..., M_n$ die Mengen der Eingabetupel, M die Menge der Ausgabetupel von v. Sei nun $M_i = \{(tk^r_{i,1}, ..., tk^r_{i,k_i}) \mid k_i, u_i \in \mathbb{N}, r \leq u_i, k_i \leq v.in\}$ für $i \in \{1, ..., n\}$, sowie $M = \{(tk^r_1, ..., tk^r_k) \mid u \in \mathbb{N}, r \leq u\}$. Sei $t \in M$: $t = \varphi_v(t_1, ..., t_n)$, wenn $t_1 \in M_1, ..., t_n \in M_n$, mit $t = (tk^r_1, ..., tk^r_{v.out})$, $t_i = (tk^{r_i}_{i,1}, ..., tk^{r_i}_{i,k_i})$, $r_i \leq u_i$.

Dann gelte für tk^r_j, $j \in \{1,...,v.out\}$: $tk^r_j.\beta = \bigcup_{i=0}^{n}{}_b \; (tk^{r_i}_{i,1}.\beta \cup_b ... \cup_b tk^{r_i}_{i,k_i}.\beta)$

iii) $v.in > 0$, $v.type$ = OR, $v.type$ = RECUP:

Seien $M_1, ..., M_n$ die Mengen der Eingabetupel, M die Menge der Ausgabetupel von v.

Es gilt aufgrund der Knotenfunktionalität: $(tk_1, ..., tk_{v.out}) \in M_i$ für $i \in \{1, ..., n\}$ $\Leftrightarrow$ $(tk_1, ..., tk_{v.out}) \in M$. Damit bleiben die Wertekennzeichnungen automatisch erhalten. ◇

Definition 4.19 *Alternative Bitvektoren*

Sei $DG^R_{\mathcal{L}} = (V^R_{DG}, E^R_{DG}, Fun^R_{DG})$ ein Datenflußgraph mit Rekursionen.

Die Menge alternativer Bitvektoren $\mathcal{K}_v$ zu einem Knoten $v \in V^R_{DG}$ sei definiert durch alle Bitvektoren, die an diesem Knoten erzeugt bzw. weitergegeben wurden. ◇

Bemerkung 4.20

Eine Menge $\mathcal{K}_v$ enthält nun alle Bitvektoren, die nicht miteinander kombiniert werden dürfen, da es sie aus Tupeln mit alternativen Werte stammen. Alle aus ihnen entstehenden Kombinationen sind ebenfalls alternativ und dürfen nicht miteinander kombiniert werden.

Satz 4.21

Für alle Token tk und tk' mit $tk.\beta \neq tk'.\beta$, $tk.\beta \in \mathcal{K}_v$ und $tk'.\beta \in \mathcal{K}_v$ gilt: $tk \perp tk'$. ◇

Beweis:

Die Aussage gilt, da nach Definition 4.18 alle Token eines Ausgabetupels den gleichen Bitvektor besitzen. Alle Bitvektoren der Ausgabetupel eines Knotens v werden in die Menge $\mathcal{K}_v$ aufgenommen. Gibt es mehrere Ausgabetupel, so sind diese nach Satz 4.11 alternativ, d.h. für die Token gilt die Relation $\perp$.

Es gilt daher: $tk \perp tk' \Leftrightarrow \exists \beta \subset_b tk.\beta, \beta' \subset_b tk'.\beta$, wobei und $\exists v \in V_{DG}^R$ mit $\beta \in \mathcal{K}_v \wedge \beta' \in \mathcal{K}_v$ für einen Knoten v. ◇

In einem Eingabetupel sollen alle Token zusammengefaßt sein, die entweder aus dem gleichen Ausgabetupel stammen, oder aus Token erzeugt wurden, die aus dem gleichen Ausgabetupel stammen. Daher gilt:

- zwei Token gehören zum gleichen Eingabetupel, wenn sie eine identische Wertekennzeichnung besitzen.
- zwei Token gehören zum gleichen Eingabetupel, wenn die Wertekennzeichnung eines der beiden Token in der Wertekennzeichnung des anderen Tokens (im Sinne einer bitweisen Teilmengenbeziehung) enthalten ist.
- zwei Token gehören zum gleichen Eingabetupel, wenn eine Teilmenge ihrer Wertekennzeichnungen identisch ist. Dies gilt jedoch nur unter der Bedingung, daß sie keine alternativen Token darstellen.

Definition 4.22 *Zusammengehörigkeit von Token*

Sei $DG_{\mathcal{L}}^R = (V_{DG}^R, E_{DG}^R, Fun_{DG}^R)$ ein Datenflußgraph mit Rekursionen.

Die Zusammengehörigkeit von Token (i.Z. $\approx_b$) tk, $tk' \in TK$ sei definiert durch:

- $tk.\beta =_b tk'.\beta$,
- $tk.\beta \subset_b tk'.\beta$,
- $tk.\beta \supset_b tk'.\beta$, oder
- $\exists \beta' \subset_b tk.\beta \wedge \beta' \subset_b tk'.\beta$, wobei $\beta_1 = tk.\beta \backslash_b \beta'$, $\beta_2 = tk'.\beta \backslash_b \beta'$ und $\nexists v \in V_{DG}^R$ mit $\beta_1 \in \mathcal{K}_v \wedge \beta_2 \in \mathcal{K}_v$ für $v \in V_{DG}^R$ ◇

Satz 4.23

Gilt für zwei Token tk, $tk' \in TK$ $tk.\beta =_b tk'.\beta$ oder $tk.\beta \subset_b tk'.\beta$, d.h. daß sie identische Wertekennzeichnungen besitzen, oder daß ihre Wertekennzeichnung in einer Teilmengenbeziehung zueinander stehen, dann $\nexists \beta' \subset_b tk.\beta, \beta'' \subset_b tk'.\beta$, so daß $\exists v \in V_{DG}^R$ mit $\beta' \in \mathcal{K}_v \wedge \beta'' \in \mathcal{K}_v$ für $v \in V_{DG}^R$, d.h. sie sind nicht alternativ. ◇

Beweis:

Treten tk und tk' am gleichen Knoten auf und gilt $tk.\beta =_b tk'.\beta$, so sind die Token im gleichen Ausgabetupel und damit nicht alternativ. Gilt $tk.\beta \subset_b tk'.\beta$, so ist tk' aus tk entstanden. Damit können sie ebenfalls nicht alternativ sein.

Treten tk und tk' nicht am gleichen Knoten auf, gilt mit $tk.\beta =_b tk'.\beta$, bzw. $tk.\beta \subset_b tk'.\beta$, daß tk durch mindestens ein Token tk_1 erzeugt wurde, tk' durch ein Token tk_2. Für diese gilt: $tk_1.\beta \subset_b tk$, $tk_2.\beta \subset_b tk'$. Damit gilt auch $tk_1.\beta =_b tk_2.\beta$ oder $tk_1.\beta \subset_b tk_2.\beta$ und die Argumentation kann wiederholt angewendet werden. Sind tk_1 und tk_2 nicht alternativ, gilt: $\nexists \beta' \subset_b tk_1.\beta, \beta'' \subset_b tk_2.\beta$, so daß $\exists v \in V_{DG}^R$ mit $\beta' \in \mathcal{K}_v \wedge \beta'' \in \mathcal{K}_v$ für $v \in V_{DG}^R$. Da tk und tk' aus tk_1 und tk_2 an unterschiedlichen Knoten entstehen, gilt auch für tk und tk': $\nexists \beta' \subset_b tk_1.\beta, \beta'' \subset_b tk_2.\beta$, so daß $\exists v \in V_{DG}^R$ mit $\beta' \in \mathcal{K}_v \wedge \beta'' \in \mathcal{K}_v$ für $v \in V_{DG}^R$. Damit können auch tk und tk' nicht alternativ sein. ◇

Satz 4.24

Für zwei Token tk, $tk' \in TK$ gilt: $((tk \dashv\vdash tk') \wedge \neg(tk \perp tk')) \Leftrightarrow tk \approx_b tk'$. $\diamond$

Beweis:

Beweis durch Induktion über die Anzahl der Kombinationen von Werten:

1. *Induktionsanfang*: Anzahl der Kombinationen ist 0.

   Beide Bitvektoren sind Startkennzeichnungen. $tk.\beta$, $tk'.\beta \in \mathcal{K}_s$. Da die einem Wertetupel zugewiesene Wertekennzeichnung eindeutig ist, gilt: $tk.\beta =_b tk'.\beta \Leftrightarrow ((tk \dashv\vdash tk') \wedge\neg(tk \perp tk'))$ nach Definition 4.10 und Satz 4.23.

2. *Induktionsschritt*:

   O.B.d.A. sei $tk.\beta$ eine kombinierte Wertekennzeichnung. Dann muß der Wert von tk durch eine Kombination aus Werten von Token tk_1, ..., $tk_n \in \mathcal{K}$, $n \in \mathbb{N}$ entstanden sein. Für jedes dieser Token gilt: $tk_i.\beta \subset_b tk.\beta$ für $i \in \{1, ..., n\}$.

   a) Ist $tk'.\beta \in \mathcal{K}_s$, so gilt:

   $\Leftarrow$ Gilt $tk \approx_b tk'$, so existiert $\beta' \subset_b tk.\beta$, wobei $\beta' =_b tk'.\beta$, da $tk'.\beta$ nicht weiter aufteilbar ist. Damit existiert $i \in \{1, ..., n\}$, so daß entweder gilt: $\beta' =_b tk_i.\beta$. oder $\beta' \subset_b tk_i.\beta$. Die Werte von tk_i und tk' haben entweder aufgrund identischer Wertekennzeichnung oder aufgrund der Induktionsvoraussetzung einen Ursprung im selben Wertetupel, d.h. es gilt $tk_i \dashv\vdash tk'$. Mit Satz 4.23 gilt außerdem $\neg(tk_i \perp tk')$. Damit haben auch tk und tk' einen Ursprung im selben Wertetupel und es gilt: $(tk \dashv\vdash tk') \wedge\neg(tk \perp tk')$

   $\Rightarrow$ Gilt $(tk \dashv\vdash tk') \wedge\neg(tk \perp tk')$, d.h. haben tk und tk' einen Ursprung im selben Wertetupel, so muß es, da $tk.\beta$ eine zusammengesetzte Kennzeichnung ist, $i \in \{1, ..., n\}$ geben, so daß tk_i und tk' einen Ursprung im selben Wertetupel haben. Dann gilt nach Induktionsvoraussetzung: $tk_i.\beta \approx_b tk'.\beta$. Damit kann $tk_i.\beta =_b tk'.\beta$ oder $tk_i.\beta \supset_b tk'.\beta$ sein. Somit gilt auch: $tk.\beta \supset_b tk'.\beta$.

   b) Ist $tk'.\beta$ eine kombinierte Wertekennzeichnung, muß der Wert von tk' durch eine Kombination aus Werten von Token tk'_1, ..., $tk'_m \in \mathcal{K}$, $m \in \mathbb{N}$ entstanden sein. Für jedes dieser Token gilt: $tk'_j.\beta \subset_b tk'.\beta$ für $j' \in \{1, ..., m\}$.

   $\Leftarrow$ Voraussetzung: $tk \approx_b tk'$.
   Gilt $tk =_b tk'$, so sind aufgrund der Eindeutigkeit der Wertekennzeichnungen tk und tk' im gleichen Wertetupel und es gilt: $tk \dashv\vdash tk'$ und $\neg(tk \perp tk')$
   Ansonsten existiert $\beta' \subset_b tk.\beta, \beta' \subset_b tk'.\beta$, wobei $\beta_1 = tk.\beta \backslash_b \beta'$, $\beta_2 = tk'.\beta \backslash_b \beta'$ und $\nexists\, v \in V^R_{DG}$ mit $\beta_1 \in \mathcal{K}_v \wedge \beta_2 \in \mathcal{K}_v$. Damit existiert $i \in \{1, ..., n\}, j \in \{1, ..., m\}$: $tk_i.\beta \approx_b tk'_j.\beta$. Nach Induktionsvoraussetzung haben die Werte von tk_i und tk'_j einen Ursprung im selben Wertetupel, d.h. es gilt: $tk_i \dashv\vdash tk'_j$ und $\neg(tk_i \perp tk'_j)$. Damit haben auch tk und tk' einen Ursprung im selben Wertetupel und es gilt: $(tk \dashv\vdash tk') \wedge\neg(tk \perp tk')$.

   $\Rightarrow$ Voraussetzung: $(tk \dashv\vdash tk') \wedge\neg(tk \perp tk')$, d.h. tk und tk' haben einen Ursprung im selben Wertetupel.
   Sind tk und tk' aus demselben Wertetupel, so gilt: $tk.\beta =_b tk'.\beta$ und damit trivialerweise auch $tk \approx_b tk'$.
   Haben tk und tk' einen Ursprung im selben Wertetupel, so muß es $i \in \{1, ..., n\}, j \in \{1, ..., m\}$ geben, so daß tk_i und tk'_j einen Ursprung im selben Wertetupel haben. Dann gilt nach Induktionsvoraussetzung: $tk_i.\beta \approx_b tk'_j.\beta$, Da tk und tk' nach Voraussetzung nicht aus alternativen Wertetupeln entstanden sind, bzw. alternative Wertetupel sind, gilt auch: $\exists \beta' \subset_b tk_i.\beta$, $\beta' \subset_b tk'_j.\beta$, wobei $\beta_1 = tk_i.\beta \backslash_b \beta'$, $\beta_2 = tk'_j.\beta \backslash_b \beta'$, und $\nexists\, v \in V^R_{DG}$ mit $\beta_1 \in \mathcal{K}_v \wedge \beta_2 \in \mathcal{K}_v$. sowie $\exists \beta'' \subset_b tk.\beta, \beta'' \subset_b tk'.\beta$, wobei $\beta' \subset_b \beta''$,

bzw. $\beta' =_b \beta''$ und $\beta_3 = tk.\beta \backslash_b \beta'$, $\beta_4 = tk'.\beta \backslash_b \beta''$. und $\nexists\, v \in V^R_{DG}$ mit $\beta_3 \in \mathcal{K}_v \wedge \beta_4 \in \mathcal{K}_v$. Damit gilt: $tk.\beta \approx_b tk'.\beta$ ◇

Folgerung 4.25

Die Relation $\approx_b$ genügt der Definition der Relation $\approx$ in Definition 3.61. ◇

Beweis:

Sei $M_i = \{(tk^r_{i,1},...,tk^r_{i,k_i}) \mid k_i, r \in \mathbb{N}\}$, für $i \in \mathbb{N}$, eine Menge von Tupeln mit $tk^r_{i,1}$, ..., $tk^r_{i,k_i} \in TK$.

Dies entspricht der erweiterten Definition von Tupelmengen mit Definition 4.4.

Sei $\mathcal{M} = \{M_i \mid i \in \mathbb{N}\}$ die Menge der Tupelmengen.

- Zu zeigen: $\forall i, n_i, k_i \in \mathbb{N}$, $l, l' \leq n_i, l \neq l'$, $r \leq n_i : tk^r_{i,l} \approx_b tk^r_{i,l'}$.

  Nach Definition der Tupelmengen 4.13 gilt für zwei Token tk und tk' in einem Tupel $(tk \dashv\vdash tk') \wedge \neg(tk \perp tk')$. Damit gilt nach Satz 4.24, daß zwei Token tk und tk' genau dann zusammen in einem Wertetupel sind, wenn $tk \approx_b tk'$ gilt.

- Zu zeigen: $\nexists i, j, u_i, n_j, k_i k_j \in \mathbb{N}$, $l \leq k_i$, $l' \leq k_j$, $r \leq u_i$, $r' \leq u_j$ mit $i \neq i' : tk^r_{i,l} \approx tk^{r'}_{i',l'}$, d.h. es gibt kein Element eines Tupels in M_i, das mit einem Element eines Tupels in M_j in Relation $\approx$ steht.

  Dies gilt nach Konstruktion der Tupel in Definition 4.13. ◇

Satz 4.26

Die in Satz 4.10 gestellten Forderungen werden von dem Konstruktionsprinzip der Bitvektoren erfüllt. ◇

Beweis:

i) Zu zeigen: $tk_{out} \dashv\vdash tk_{in}$, wenn der Wert von tk_{in} zur Erzeugung des Wertes von tk_{out} benutzt wird.
   - Für $v.in > 0$, $v.type$ = OR, $v.type$ = RECUP gilt:
     ist $tk_{in} = tk$, so gibt es ein Ausgabetoken $tk_{out} = tk$. Damit gilt für alle tk_{out}: $tk_{out}.\beta =_b tk_{in}$.
   - Für $v.in > 0$, $v.type$ = AND, $v.type$ = QUERY, $v.type$ = FACT, $v.type$ = FUNC, $v.type$ = RECDOWN, $v.type$ = NOT:
     Ist tk_{out} ein Ausgabetoken, das aus tk_{in_1},..., tk_{in_n}, $n \in \mathbb{N}$, Eingabetoken entstanden ist, so gilt: $tk_{out}.\beta = \bigcup_{i=0}^{n}{}_b\, tk_{in_i}.\beta$. Damit gilt $\forall i \in \{1, ..., n\}$: $tk_{out}.\beta \supset_b tk_{in_i}.\beta$ und damit $tk_{out} \dashv tk_{in_i}$.

ii) Zu zeigen: für ein beliebiges Token tk gilt:

   a) $tk \dashv\vdash tk_{in} \Rightarrow tk \dashv\vdash tk_{out}$

      Gilt für tk: $tk \dashv\vdash tk_{in}$ und gilt nach i) $tk_{in} \dashv\vdash tk_{out}$, so gilt aufgrund der Transitivität der Operation $\subset_b$ auch: $tk \dashv\vdash tk_{out}$.

   b) $tk \perp tk_{in} \Rightarrow tk \perp tk_{out}$

      Gilt für tk: $tk \perp tk_{in}$ und gilt nach i) $tk_{in} \dashv\vdash tk_{out}$, so gilt aufgrund der Transitivität der Operation $\subset_b$ auch: $tk \perp tk_{out}$.

c) $\neg\,(tk \dashv\vdash tk_{in}) \wedge \neg\,(tk \perp tk_{in}) \Rightarrow \neg\,(tk \dashv\vdash tk_{out}) \wedge \neg\,(tk \perp tk_{out})$

- Annahme: es gelte $tk \dashv\vdash tk_{out}$. So gilt entweder: tk ist im gleichen Ausgabetupel wie tk_{out}, oder tk stammt aus gleichen Ausgabetupel wie tk_{out}. Ist tk im gleichen Ausgabetupel wie tk_{out}, so wurde es durch Eingabe von tk_{in} erzeugt. Damit gilt: $tk \dashv\vdash tk_{in}$. Stammt tk aus gleichen Ausgabetupel wie tk_{out}, so muß tk auch aus dem gleichen Ausgabetupel wie tk_{in} stammen und es gilt: $tk \dashv\vdash tk_{in}$. Dies ist ein Wiederspruch zur Voraussetzung $\neg\,(tk \dashv\vdash tk_{in})$.
- Annahme: es gelte: $tk \perp tk_{out}$
  - Ist tk in einem alternativen Ausgabetupel wie das Ausgabetupel in dem tk_{out} ist, so wurde es entweder
    · durch Eingabe von tk_{in} erzeugt, dann gilt: $tk \dashv\vdash tk_{in}$, oder
    · durch Eingabe eines zu tk_{in} alternativen Tokens, dann gilt: $tk \perp tk_{in}$.
  - Ist tk nicht in einem alternativen Ausgabetupel, so kann es aufgrund der Eigenschaft $tk_{in} \dashv\vdash tk_{out}$ und $tk \perp tk_{out}$ in einem zu dem Eingabetupel, aus dem tk_{out} entstanden ist, alternativen Eingabetupel gewesen sein. Damit gilt aber: $tk \perp tk_{in}$.
  - Tritt tk weder in einem Eingabetupel, noch in einem Ausgabetupel auf, so gilt, da tk nicht am Knoten entstanden sein kann und aufgrund der Eigenschaft von $\perp$ $(tk_i \perp tk_j \Leftrightarrow \exists\beta \subset_b tk_i.\beta, \beta' \subset_b tk_j.\beta$, wobei $\exists v \in V_{DG}^{R}$ mit $\beta \in \mathcal{K}_v \wedge \beta' \in \mathcal{K}_v$ für einen Knoten $v)$ auch: $tk \perp tk_{in}$.

  Somit gilt: $(tk \perp tk_{out}) \Rightarrow (tk \dashv\vdash tk_{out}) \vee (tk \perp tk_{out})$. ◇

Eine einfache Optimierung der Länge der Bitvektoren besteht darin, Wertekennzeichnungen zu nur dann zu vergeben, wenn an einem inneren Knoten des Graphen (z.B. AND oder OR) mehrere Wertetupel gemeinsam ausgegeben werden.

Gibt es nur ein Wertetupel, ist die Korrespondenz aller Werte allgemeingültig und es wird kein eigener Bitvektor zu Kennzeichnung benötigt. Die Kombinationen von Bitvektoren werden dennoch wie bereits beschrieben erzeugt, wobei der leere Bitvektor das Nichtvorhandensein eines speziellen, eigenen Bitvektors repräsentiert.

Wird z.B. ein Wert ungültig, der den leeren Bitvektor besitzt, so heißt das, daß es keine alternativen Werte für diese Kante gibt, als auch, daß dieser Teilgraph überhaupt kein Ergebnis liefern kann. Folglich können in diesem Teilgraphen alle Werte gelöscht werden, unabhängig davon, ob sie den leeren Bitvektor besitzen oder nicht. Daher muß auch bei einer Kombination mit dem leeren Bitvektor keine zusätzliche Kennzeichnung erzeugt werden.

Definition 4.27 *Graphfragmente*

Als Graphfragmente sollen im folgenden Teile des Datenflußgraphen bezeichnet werden, für die gilt:

- Für die Wurzel v gilt: $v.type$ = AND.
- Ein Knoten w gehört zu einem Graphfragment mit Wurzel v, wenn es einen Pfad von v nach w in E_{DG}^{R} gibt, der keinen Knoten u mit $u.type$ = AND oder $u.type$ = OR enthält. ◇

Eine weitere Optimierung erhält man durch Zerlegung des Graphen in unabhängige Graphfragmente, deren Wurzel jeweils ein AND-Knoten ist (siehe Figur 53).

Eine derartige Aufteilung ist auch bei Rekursionen möglich, der Teilgraph muß dann jedoch die gesamte Rekursion, d.h. beide Rekursionsschleifen umfassen.

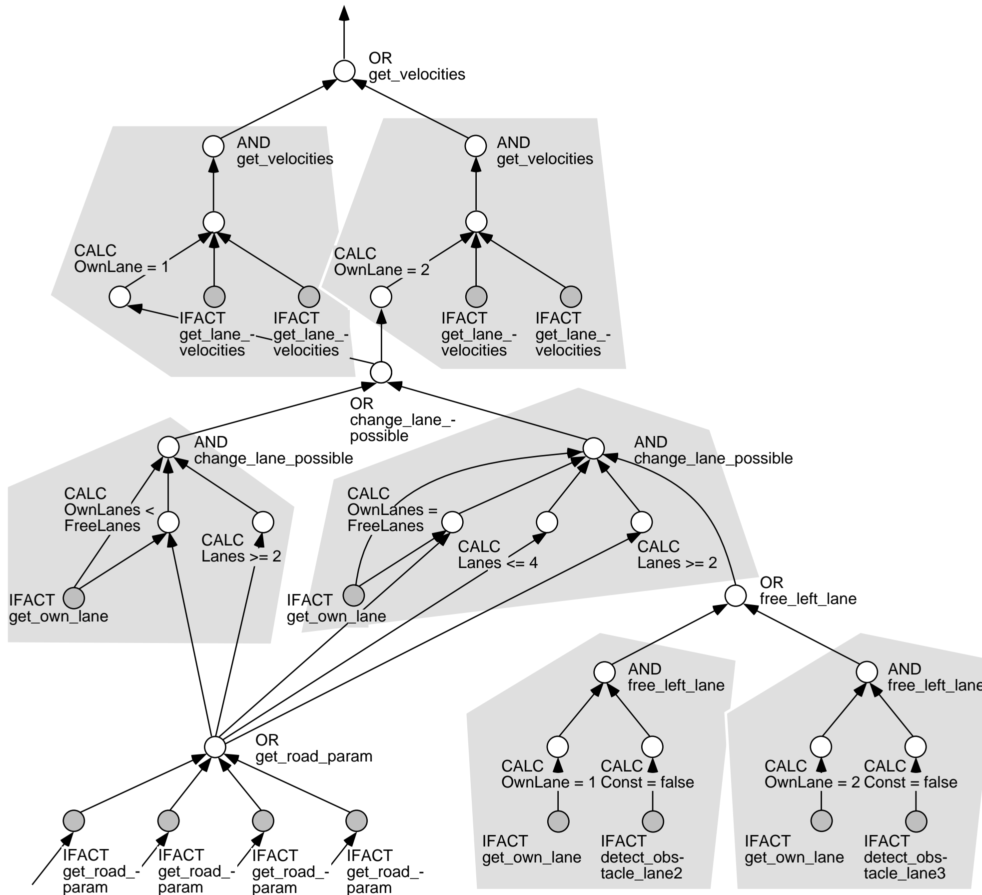


Abbildung 53: Aufteilung des Graphen in Graphfragmente

Eingaben an Graphfragmente kommen immer von der Wurzel des vorhergehenden Graphfragments, nicht jedoch von einem beliebigen anderen Knoten aus dem Graphfragment. (Dies gilt nach Satz 3.68). Das bedeutet, daß die Werte, die über die Wurzel des Graphfragments ausgegeben werden, keine Wertekennzeichnung besitzen müssen, die noch Abhängigkeiten mit anderen, im gleichen Teilgraphen erzeugten Werten festhält. Folglich können alle in einem Graphfragment vergebenen Wertekennzeichnungen nach Abarbeitung der Wurzel des Graphfragments gelöscht werden. An die Ausgabetupel können dann neue Wertekennzeichnungen vergeben werden. Erhalten bleiben müssen dabei lediglich die Wertekennzeichnungen der Eingabewerte an den Graphfragmenten, da diese Zusammengehörigkeiten zu Werten repräsentieren können, die in einem weiter oben liegenden Teil des Graphen noch relevant werden.

Durch Freigabe und Neuvergabe von eindeutigen Wertekennzeichnungen werden insgesamt weniger verschiedene eindeutige Wertekennzeichnungen benötigt. Damit bleibt auch die notwendige maximale Länge des Bitvektors klein.

Bemerkung 4.28 *Abschätzung der maximalen Länge der Bitvektoren*

Sei $SG^{E}_{\mathcal{L}} = (V_{SG}, E_{SG})$ ein erweiterter Strukturgraph. Die Analyse wird anhand des Strukturgraphen vorgenommen.

Sei $\mathcal{K} = \wp(\{0,1\}^n)$ die Menge der eindeutigen Wertekennzeichnungen.

Dann gilt für $n \in \mathbb{N}$:

1. Unoptimierter Fall:

   $n = |\{v \mid v.in = 0\}|$.

2. Einfach optimierter Fall (*len, len'* seien für die Definition benötigte Funktionen):

   a) für $v \in V$ mit $v.in = 0$, $v.type$ = FUNC, $v.type$ = FACT, oder $v.type$ = FUNC: $len(v) = 0$

   b) für $v \in V$ mit $v.type$ = OR: $len(v) = len'(w_1) + ... + len'(w_n)$,
   wenn gilt: $\exists w_1, ..., w_n \in V$ für $n \in \mathbb{N}$ mit $e_{v,w_1}, ..., e_{v,w_n} \in E$
   wobei für len' gilt:
   $$len'(w_i) = \begin{cases} len(w_i) & \text{wenn } len(w_i) \geq 1 \\ 1 & \text{wenn } len(w_i) = 0 \end{cases}$$

   c) für $v \in V$ mit $v.type$ = NOT, $v.type$ = RECDOWN, $v.type$ = RECUP: $len(v) = len(w)$
   wenn $w \in V$ mit $e_{v,w} \in E$

   d) für $v \in V$ mit $v.in \neq 0$, $v.type$ = FACT, $v.type$ = FUNC, $v.type$ = AND, $v.type$ = QUERY: $len(v) = len(w_1) + ... + len(w_n)$,
   wenn gilt: $\exists w_1, ..., w_n \in V$ für $n \in \mathbb{N}$ mit $e_{v,w_1}, ..., e_{v,w_n} \in E$

   Für n gilt dann: $n = len(v)$ mit $v.type$ = QUERY.

   Durch die Addition an einem OR-Knoten werden alle alternativen Zweige bis zu den Blättern (rekursiv) gezählt. Da nur dann Bitvektoren vergeben werden sollen, wenn die Token tatsächlich über einen OR-Knoten gehen, muß die Initialisierung mit 0 erfolgen. Um sie jedoch am OR-Knoten richtig zählen zu können, bedarf es dort einer Umdefinierung auf 1 über die Hilfsfunktion len'. Für die anderen Knoten gilt, daß, wenn ein leerer Bitvektor ankommt, dieser nicht durch einen eindeutigen an dieser Stelle ersetzt werden muß. Daher benötigt man hier die Funktion len' nicht.

   Beim NOT-Knoten genügt die Übernahme der Anzahl vom Nachfolgeknoten, da alle anderen Eingaben z.B. mindestens am darüberliegenden AND-Knoten verknüpft und dort gezählt werden.

   Die Knoten RECDOWN und RECUP geben die Werte nur weiter, hier ändert sich an der Anzahl der benötigten Kennzeichnungen nichts.

3. Optimierter Fall mit Rückgabe der Vektoren:

   Für $v \in V_{DG}^R$ mit $v.in \neq 0$, $v.type$ = AND, $v.type$ = QUERY: $len_r(v) \ll len(v)$.

   Damit gilt auch für $n_r = len_r(v)$ mit $v.type$ = QUERY: $n_r \ll n$.

   In diesem Fall läßt sich nur eine Abschätzung vornehmen, da die Mächtigkeit der Ergebnismenge eines Joins von den Werten der Eingabemengen abhängig ist. Die maximale Mächtigkeit ist die eines Kreuzproduktes, wie es in 1. und 2. eingeht. Im allgemeinen sind jedoch deutlich weniger Tupel das Ergebnis eines Joins als eines Kreuzproduktes. Eine genauere Abschätzung wird in Kapitel 5.2.1 beschrieben, die hier analog Anwendung finden kann. ◇

Bemerkung 4.29 *Abschätzung des maximal benötigten Speicherplatzes für alternative Bitvektoren*

Es ist ausreichend, die Menge der alternativen Bitvektoren $\mathcal{K}_v$ für diejenigen Knoten $v \in V^R_{DG}$ zu bestimmen, für die gilt: *v.type* = Or, da nur dort Bitvektoren alternativ werden können.

Die Mengen $\mathcal{K}_v$ für Knoten $v \in V^R_{DG}$ mit *v.type* $\neq$ Or können daher keine neuen Abhängigkeiten enthalten.

Der benötigte Speicherplatz ergibt sich dann aus der Anzahl der Or-Knoten multipliziert mit der maximalen Anzahl benötigter Bitvektoren. ◇

Da der für das Abspeichern alternativer Bitvektoren benötigte Speicherplatz offensichtlich durch die Länge der Bitvektoren und die Anzahl der in einem Programm enthaltenen Or-Knoten beschränkt ist, läßt sich ein Assoziativspeicher verwenden. Damit ist die Überprüfung zweier Bitvektoren auf Alternativität in linearer Zeit möglich.

4.1.2 Knoten- und Kantenrepräsentation

Knoten und abgehende Kanten definieren in Datenflußgraphen Operationen, Speicherplatz für die Operanden und Adressen von Nachfolgeknoten, die Ergebnisse erhalten. Der Speicherplatz für Operanden wird durch Eingangskanten symbolisiert, die Adressen der Nachfolgeknoten durch Ausgangskanten.

In bisher bekannten Datenflußgraphen kommt an jedem Knoten für die dort auszuführende Operation für jeden Eingabeparameter pro Iterationstiefe genau ein Wert an. Dies ermöglicht das Abspeichern des Wertes im Knoten selbst, wenn für jede Iterationstiefe eine Kopie des Knotens existiert. Alternativ können Werte in einem speziell für den Knoten für jede Rekursionstiefe reservierten Speicherplatz (z.B. einem Activity Frame, wie er im hybriden Datenflußrechner Monsoon implementiert ist) eingetragen werden. Die am Knoten zu verarbeitenden Werte werden damit lokal gespeichert.

Im vorliegenden Datenflußmodell kann es jedoch vorkommen, daß für die gleiche Iterationstiefe mehrere Werte für einen Parameter ankommen. Eine korrekte Verarbeitung der Werte ist jedoch erst dann möglich, wenn alle miteinander zu verarbeitenden Werte am Knoten angekommen sind. Ein Knoten muß also eine flexible Größe besitzen, da die Anzahl der Werte bei jeder Ausführung unterschiedlich sein kann und zum Zeitpunkt der Erzeugung des Knotens nicht bekannt ist.

Die Darstellung eines Knotens in Graphform ist in Figur 54 zu sehen. Es ergibt sich der in Figur 55 beispielhaft gezeigte Knotenaufbau für die Implementierung auf einem Datenflußrechner.

Durch die Angabe der Operation im Knoten kann die Verarbeitungseinheit die im Knoten mitgelieferten Werte der Funktionalität des Knotens entsprechend korrekt verarbeiten.

Da Ausgabewerte direkt an Nachfolgeknoten versendet werden, ohne selbst im Knoten abgespeichert zu werden, genügt eine Angabe, an welche Knoten die Ergebniswerte geschickt werden müssen. Für jede Ausgabekante eines Parameters im Graphen muß daher in der Knotenstruktur die Zieladresse des Wertes abgespeichert werden.

Eine solche Adresse besteht dabei aus
- der Adresse des Knotens, zu dem der Wert geschickt werden soll und

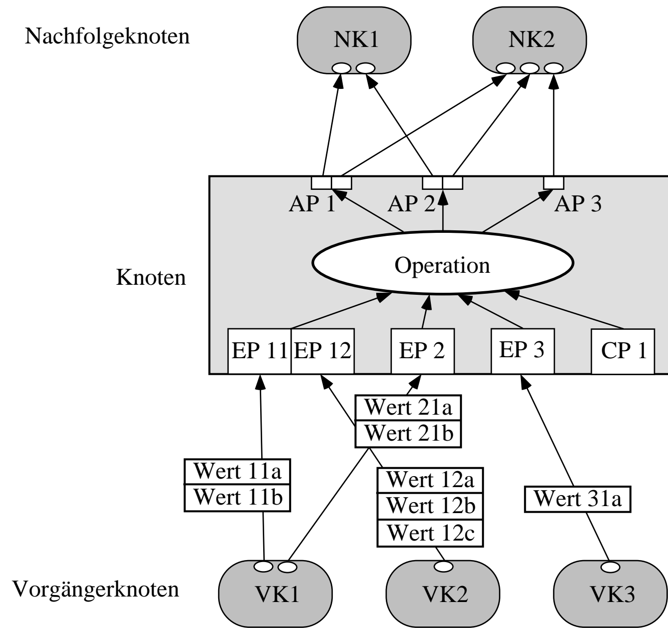


Abbildung 54: Knoten im Graphen

Knotenoperation	Operation			
Ausgabeparameter	AP 1	1	Adresse Nachfolgeknoten 1	
	AP 1	2	Adresse Nachfolgeknoten 2	
	AP 2	1	Adresse Nachfolgeknoten 1	
	AP 2	2	Adresse Nachfolgeknoten 2	
	AP 3	1	Adresse Nachfolgeknoten 2	
Konstanten	CP 1	1	Konstante 1	
Eingabeparameter	EP 1	1	[1] Wert 1 1 a	5
	EP 1	2	[2] Wert 1 2 a	7
	EP 2	1	[3] Wert 2 1 a	6
	EP 3	1	[4] Wert 3 1 a	0
			[5] Wert 1 1 b	0
			[6] Wert 2 1 b	0
			[7] Wert 1 2 b	8
			[8] Wert 1 2 c	0

feste Länge (AP 1 … EP 3)
flexible Länge (Wert 1 1 b … Wert 1 2 c)
Parameter
Nummer der "Kante"
Wert
Verweis auf nächsten Wert

Abbildung 55: Beispielhafte Struktur eines Knotens

- einer Angabe des Parameters, der den Wert erhalten soll.

Die Anzahl der Kanten pro Ausgabeparameter ist bei der Erstellung des Graphen bereits bekannt, daher kann dies im Knoten entsprechend fest kodiert sein.

Konstanten werden im Knoten dauerhaft in dafür reservierte Speicherplätze eingetragen.

Der Knoten muß außerdem Platz für alle ankommenden Eingabewerte haben. Dabei ist zunächst für jede eingehende Kante ein eigener Speicherplatz reserviert. Kommt an einer Kante mehr als ein Wert an, wird dieser flexibel an das Ende des Knotens angehängt. Über einen Verweis, der in diesem Fall aus der Nummer des Speicherplatzes bestehen kann, kann der jeweils nächste Wert vom vorhergehenden Wert aus gefunden werden.

Die Obergrenze für die Anzahl der Werte kann bei der Erzeugung des Graphen abgeschätzt werden. Sie bestimmt sich in grober Annäherung aus der Anzahl der durch IFACT-Knoten eingegebenen Werte über Parameter. In Kaptitel 5.2.1 findet man dazu genauere Betrachtungen.

Bei der Erzeugung eines Tokens für die Ausgangskante eines Ausgabeparameters wird die Zieladresse von der Definition der Zieladresse im Knoten übernommen.

4.1.3 Terminierung der Rekursion

Eine wesentliche Eigenschaft von Rekursionen, wie sie das in dieser Arbeit vorgestellte Modell vorgibt, ist, daß Rekursionen nicht dann terminieren dürfen, wenn der Terminierungszweig zum erstenmal erfolgreich ausgeführt wurde. Das Ergebnis des Terminierungszweigs könnte im späteren Verlauf ungültig werden. Eine Form von Backtracking, um dann ein weiteres Rekursionsergebnis zu berechnen gibt es jedoch nicht. Daher muß eine Rekursion alle möglichen Ergebnisse berechnen.

Eine Rekursion in $\text{DATALOG}_f^{\neg}$ muß jedoch nicht endlich sein. Sie kann dennoch alle korrekten Ergebnisse liefern [Sterling 86]. Oder umgekehrt: selbst wenn alle korrekten Ergebnisse berechnet wurden, muß eine Rekursion nicht zwangsläufig terminieren.

Ein einfaches Abbruchskriterium, das auch in Bottom-Up Verfahren angewendet wird, liefert die Aussage, daß eine Rekursionsschleife nur dann neue Ergebnisse liefern kann, wenn sich die aktuelle Menge von Werten von der des letzten Durchlaufs unterscheidet, d.h. neue oder andere Werte hinzugekommen sind. Dies entspricht dem Erreichen eines Fixpunktes, d.h. der Inhalt der Wertemenge bleibt dann konstant.

Wenn es nur eine Ausgabeschleife gibt, läßt sich dieses Abbruchskriterium auf diese anwenden. Wenn es eine Eingabeschleife gibt, muß dieses Abbruchskriterium für diese gelten. Für jede erfolgreiche Terminierung der Eingabeschleife muß dann die Ausgabeschleife genauso oft durchlaufen werden.

Mit dieser Methode der Überprüfung konstant bleibender Wertemengen läßt sich auch eine Rekursion beenden, die nicht terminiert, weil sich die an die Rekursionsschleife eingegebene Wertemenge nie verändert und gleichzeitig der Terminierungsfall nicht erfolgreich ist. Das Ergebnis ist hier das Scheitern der gesamten Rekursion.

Die Abbruchsbedingung über einen Fixpunkt ist im allgemeinen jedoch nur dann möglich, wenn in der oder den Rekursionsregeln, sowie in den von den Rekursionsregeln aufgerufenen Regeln nur Fakten (FACTs und IFACTs) vorkommen.

Treten Systemprädikate auf, so gibt es mehrere Möglichkeiten:

- Es existiert ebenfalls ein Fixpunkt der Schleife, d.h. die in der Schleife erzeugten Wertemenge wird nach endlich vielen Durchläufen stationär.
- Ein Systemprädikat vor dem Aufruf der Rekursion in der Rekursionsregel stellt selbst eine Abbruchbedingung dar, indem es nach endlich vielen Durchläufen scheitert, d.h. der Ergebniswert ist *failure* und die Schleife wird nicht fortgesetzt.

- Ein Systemprädikat scheitert nicht nach endlich vielen Durchläufen und es verändert in jedem Schleifendurchlauf die Werte, die an die nächste Rekursionstiefe weitergegeben werden derart, daß kein Fixpunkt existiert.

Der letzte Fall tritt jedoch nicht auf, wenn man nur konsistente Programme als Ausgangsprogramm zuläßt, da das Problem nicht durch die Konstruktion des Graphen entsteht, sondern im ursprünglichen $\text{DATALOG}_f^{\neg}$-Programm liegt: tritt dort ein Systemprädikat auf, das nicht nach endlich vielen Durchläufen scheitert, sondern für jede Rekursionstiefe neue Werte liefert, kann auch das $\text{DATALOG}_f^{\neg}$ -Programm nicht terminieren (vorausgesetzt alle Lösungen sollen gefunden werden). Ein solches $\text{DATALOG}_f^{\neg}$ -Programm genügt allerdings nicht mehr den Konsistenzbedinungen (Def. 3.1).

Ein Möglichkeit, eine solche Situation dennoch abzufangen, ist die Einführung einer maximal zulässigen Anzahl von Schleifendurchläufen. Dies könnte durch das Einfügen eines Zählers in den Knoten REC_d geschehen. Wird die maximal zugelassene Rekursionstiefe überschritten, scheitert der Rekursionszweig als Ganzes. Im Hinblick auf das Anwendungsgebiet sollte dies jedoch nur unter zusätzlicher Ausgabe von Fehlermeldungen geschehen.

Enthält eine rekursiv aufgerufene Regel ein negiertes Literal, so wird für dieses Literal zunächst die vollständige Ausgabe berechnet, da für eine Negation gilt, daß alle Parameter den Modus Ausgabe besitzen. Die eigentliche Negation erfolgt durch Bildung einer Differenzmenge, wobei die Menge für den ersten Operanden der Differenzbildung vollständig durch andere Literale berechnet werden kann (siehe Definition 3.1). Da für die Differenzmengenbildung gilt, daß sie monoton in ihrem ersten Operanden ist, bedeutet dies für die Schleife, daß sie ebenfalls durch Erreichen eines Fixpunktes der in der Schleife erzeugten Wertemenge beendet werden kann.

Die Ausführung einer Rekursion endet folglich dann,
1. wenn die Ausführung eines Knotens im Zyklus scheitert,
2. wenn eine maximal zulässige Anzahl an Zyklen erreicht wurde, oder
3. wenn sich die Menge der im Zyklus transportierten Werte nicht weiter verändert (Erreichen eines Fixpunktes).

Ist der Terminierungszweig für Werte in unterschiedlichen Rekursionstiefen erfolgreich, werden die Ergebnisse am obersten Knoten der Rekursion gesammelt und gemeinsam weitergegeben.

Für die erste Abbruchbedingung gilt:

- wenn die Ausführung eines Knotens im Zyklus scheitert, ohne eine Ausgabe zu produzieren, dann kann der Zyklus automatisch nicht fortgesetzt werden, da für nachfolgende Knoten entsprechende Eingaben fehlen.
- wenn die Ausführung eines Knotens im Zyklus durch die Erzeugung eines Tokens für den Ergebnisparameter mit dem Wert *failure* als gescheitert markiert wird und dieses bis zum Knoten RECDOWN gelangt, wird das Token nur in den Terminierungszweig, aber nicht in die nächste absteigende Rekursionsschleife geschickt. Der Knoten RECUP behandelt *failure*-Token wie andere Token auch und sendet sie an die entsprechende aufsteigende Schleife, oder aus der Rekursion heraus.

Für die zweite Abbruchbedingung gilt:

- wenn es sowohl eine aufsteigende, als auch eine absteigende Rekursionsschleife gibt, dann kontrolliert der Knoten RECDOWN die Anzahl. Wurde die maximale Anzahl erreicht, so wird das Token nur an den Terminierungszweig weitergegeben, aber nicht an die nächste Schleife.

- wenn es nur eine aufsteigende Rekursionsschleife gibt, dann kontrolliert der Knoten RecUp die Anzahl. Wurde die maximale Anzahl erreicht, so wird das Token nur aus der Rekursion ausgegeben.

Für die dritte Abbruchsbedingung benötigt man jedoch zusätzlich ein speicherndes Verhalten der Knoten, d.h. der Werte an den Eingangskanten der Knoten, da sonst die Werte einer Rekursionstiefe nicht mit den Werten der letzten Rekursionstiefe verglichen werden können. Werte müssen nunmehr nicht mehr nur für die Dauer einer vollständigen Knotenausführung lokal gespeichert werden, sondern bis zur nächsten Ausführung. Da in der nächsten Ausführung neu ankommende Werte die Werte der vorhergehenden Ausführung nicht einfach überschreiben dürfen, muß der Knoten eine zusätzliche zur Verwaltung alter und neuer Werte erhalten, die aber das Berechnungsmodell selbst nicht beeinflußt.

Für die Realisierung von Rekursionen kann daher die Eigenschaft des Datenflußmechanismus, daß ein Knoten ausgeführt werden kann, wenn auf jeder Eingangskante ein Datum anliegt, nicht aufrecht erhalten werden. Statt dessen muß für das Datenflußmodell dieser Arbeit gelten, daß ein Knoten dann ausgeführt werden kann, wenn auf jeder Kante mindestens ein Datum anliegt und kein weiteres mehr eintreffen kann. Die Entscheidung über die Ausführbarkeit wird durch ein Schrittsteuerungsverfahren möglich, das bereits im Rahmen des Ereignisflußmodells [Hahn 85, Hahn 92] und in der Arbeit von [Müller 96] eingesetzt wurde. Im Bereich der deduktiven Datenbanken gibt es Analogien zum Verfahren der differentiellen Iteraltion [Balbin 87, Güntzer 87].

Es müssen daher sowohl der Kontrollmechanismus, als auch der Datenmechanismus abgeändert werden:

- Für den Kontrollmechanismus benötigt man eine zum Datenflußgraphen externe Entscheidungsinstanz, welche Knoten wann ausführbar sind.
- Für den Datenmechanismus muß es möglich sein, ankommende Token mit schon vorhandenen zu vergleichen und, falls der Vergleich Unterschiede ergibt, diese zu ersetzen.

Es gibt eine Reihe von Verfahren für die Erkennung von (unendlichen) Rekursionen, wie z.B. [Ramakrishnan 95].

Ein Verfahren, das sehr effizient das Erreichen eines Fixpunktes erkennt, wurde von [Gelder 87] vorgestellt. Es basiert auf einer Tabellierung der in der Schleife erreichten Werte und die Analyse der Tabelleneinträge durch zwei Zeiger, die abarbeitungsbedingt unterschiedlich schnell weitergeschaltet werden.

4.1.4 Ausführungsvorschrift

Das ranginduzierte Datenfluß-Berechnungsmodell *RDG* basiert auf

i) eine Berechnungsvorschrift, bestehend aus
   - einer Darstellung der Übergangsfunktionsberechnung unter Berücksichtigung der Strukturierung mittels Elementarfunktionen,
   - einer Angabe der Art der Übermittlung von Daten zwischen Elementarfunktionen, die in einer Ergebnisproduzent - Argumentverbraucher Beziehung stehen, und

ii) einem Mechanismus zur Auswahl der Elementarfunktionen, die aktiviert, d.h. für eine Auswertung festgelegt werden.

Der Datenflußgraph enthält einen Knoten in eindeutiger Weise für jede zur Berechnung der Übergangsfunktion benötigten Anwendung einer Elementarfunktion. Die Kanten zwischen den Knoten spiegeln die Ergebnisproduzent - Argumentverbraucher Beziehung wieder.

Der Mechanismus zur Auswahl von zu aktivierenden Elementarfunktionen basiert auf einer Rangordnung des Datenflußgraphen und einem darauf basierenden Selektionsverfahren.

Eine Rangordnung ist eine Zerlegung der Menge der Knoten des Datenflußgraphen in disjunkte Mengen (Ränge), die jeweils nur voneinander datenunabhängige Befehle enthalten. Durch ein Selektionsverfahren wird schrittweise der jeweils nächste auszuführende Rang bestimmt. Diese Auswahl hängt im allgemeinen vom zuvor selektierten Rang ab. Gibt es keine rückwärtsgerichteten Kanten von Knoten des aktuellen Ranges aus, so ist der zu selektierende Rang der dem aktuellen Rang nachfolgende Rang. Gibt es dagegen rückwärtsgerichtete Kanten, so ist der zu selektierende Rang der niedrigste Rang in dem sich Knoten befinden, an denen diese Kanten enden.

Aufgrund ihrer Datenunabhängigkeit können Knoten eines Ranges parallel (oder in beliebiger Reihenfolge) ausgeführt werden.

Bemerkung:

$e_{v,w} \in E^R_{DG}$ stehe im folgenden als Abkürzung für
$\exists i \in \{1,...,v.out\}$, $\exists j \in \{1,...,w.in\}$, $e_{v.op_i,w.ip_j} \in E^R_{DG}$ ◇

Definition 4.30 *Partielle Ordnung auf Knoten*

Sei $DG^R_{\mathcal{L}} = (V^R_{DG}, E^R_{DG}, Fun^R_{DG})$ ein Datenflußgraph mit Zyklen.

i) $<_v$ sei definiert durch: $v <_v w$ für $v, w \in V^R_{DG}$,
wenn
- $\exists e_{v,w} \in E^{vm}_{DG}$,
  wobei $E^{vm}_{DG} \subset E^R_{DG}$ definiert seien wie in Definition 3.47.
- wenn $v <_v u$ und für $u <_v w$ $v, u, w \in V^R_{DG}$ dann gilt auch $v <_v w$

ii) $=_v$ sei definiert durch: $v =_v w$, wenn gilt $v = w$ für $v, w \in V^R_{DG}$

$v \leq_v w$ stehe für $v <_v w \vee v =_v w$.

Satz 4.31

$\leq_v$ ist eine partielle Ordnung auf der Menge der Knoten V^R_{DG}.

Beweis:

Eine Ordnung ist definiert als eine Relation, die reflexiv, transitiv und antisymmetrisch ist:

- Für jeden Knoten $v \in V^R_{DG}$ gilt: $v =_v v$ und demzufolge auch $v \leq_v v$.
- $\leq_v$ ist per Definition transitiv.
- Seien $v, w \in V^R_{DG}$ mit $v \leq_v w$ und $w \leq_v v$. Angenommen es gilt nicht $v =_v w$, dann existieren nach Definition Kanten von v nach w und Kanten von w nach v. Da die Menge E^{vm}_{DG} nach Satz 3.49 keine Zyklen beinhalten, führt dies zu einem Widerspruch. Daher muß gelten: $v \leq_v w \wedge w \leq_v v \Rightarrow v =_v w$. ◇

Bemerkung 4.32

Die Ordnung ist partiell, da nicht alle Knoten $v \in V_{DG}^R$ miteinander vergleichbar sind. ◇

Definition 4.33 *Zulässige Rangordnung*

$RANK$: $V_{DG}^R \rightarrow \mathbb{N}$ sei die Menge von Rangordnungsfunktionen.

Eine zulässige Rangordnungsfunktion $rank \in RANK$ ist monoton, d.h. für $v, w \in V_{DG}^R$:
$v <_v w \Rightarrow rank(v) < rank(w)$ ◇

Definition 4.34 *Rangordnung des Datenflußgraphen*

Sei $DG_{\mathcal{L}}^R = (V_{DG}^R, E_{DG}^R, Fun_{DG}^R)$ ein Datenflußgraph mit Zyklen. $E_{DG}^{vm} \subset E_{DG}^R$ sei definiert wie in Definition 3.56.

$rank_{DG} \in RANK$ sei definiert als

a) für alle Knoten $v \in V_{DG}^R$ mit $v.in = 0$ gilt: $rank_{DG}(v) = 0$.

b) für alle Knoten $v \in V_{DG}^R$ mit $v.in \neq 0$ gilt: $rank_{DG}(v) = n$, $n \in \mathbb{N}$, wenn $\exists w \in V_{DG}^R$ mit $rank_{DG}(w) = n - 1$ mit $e_{w,v} \in E_{DG}^{vm}$ und $\nexists w' \in V_{DG}^R$ mit $rank_{DG}(w') = n$ mit $e_{w',v} \in E_{DG}^{vm}$ ◇

Satz 4.35

$rank_{DG} \in RANK$ ist eine zulässige Rangordnung. ◇

Beweis:

Zu zeigen ist: $v, w \in V_{DG}^R$: $v <_v w \Rightarrow rank_{DG}(v) < rank_{DG}(w)$.

Induktionsanfang $n = 0$:

Für $v \in V_{DG}^R$ mit $rank_{DG}(v) = 0$ gilt: wenn $v.in = 0$ existiert kein $w \in V_{DG}^R$ mit $w <_v v$. Damit gilt $rank_{DG}(v) = 0$.

Induktionsschritt $n \rightarrow n + 1$:

Für $v \in V_{DG}^R$ mit $rank_{DG}(v) > 0$ gilt: da $v.in \neq 0$ existiert $w \in V_{DG}^R$ mit $e_{w,v} \in E_{DG}^{vm}$. Sei $rank_{DG}(w) = $ n. Damit gilt: $w <_v v$, sowie: $rank_{DG}(v) = rank_{DG}(w) + 1 = n + 1$ und damit: $rank_{DG}(w) < rank_{DG}(v)$. ◇

Definition 4.36 *Datenunabhängigkeit*

Zwei Knoten $v, w \in V_{DG}^R$ für die weder $v <_v w$ noch $w <_v v$ gilt, heißen datenunabhängig. ◇

Satz 4.37

Alle Knoten eines Ranges sind datenunabhängig. ◇

Beweis:

Für alle Knoten $v, w \in V_{DG}^R$ gelte $rank_{DG}(v) = rank_{DG}(w)$.

Damit gilt nicht: $rank_{DG}(v) < rank_{DG}(w)$, bzw. $rank_{DG}(w) < rank_{DG}(v)$. Folglich gilt nicht $v <_v w$, bzw. $w <_v v$. ◇

Definition 4.38 *Ranginduziertes Datenfluß-Berechnungsmodell*

Ein ranginduziertes Datenfluß-Berechnungsmodell
$RDG = (V^R_{DG}, E^R_{DG}, Fun^R_{DG}, TK, rank_{DG}, active, sel)$
zu einem Datenflußgraphen mit Zyklen $DG^R_{\mathcal{L}} = (V^R_{DG}, E^R_{DG}, Fun^R_{DG})$ sei definiert als

i) einer endlichen, nicht-leeren Menge von Maschinenfunktionen Fun^R_{DG}.

   $\phi_M : V^R_{DG} \rightarrow Fun^R_{DG}$ sei eine Funktion, die den Knoten des Datenflußgraphen eine Maschinenfunktion zuordnet.

ii) einer Menge von Token TK,

iii) einer zulässige Rangordnungsfunktion $rank_{DG} \in RANK$,

iv) einer Funktion $active$: $V \rightarrow \mathbb{B}$.

v) einer Selektionsfunktion sel: $\mathbb{N} \rightarrow \mathbb{N}$:

   sei $t \in \mathbb{N}$ (ein diskreter Zeitpunkt)

   $sel(0) = 0$

$$sel(\text{t+1}) = \begin{cases} \text{rank}_{DG}(v) + 1 & \text{für } sel(t) = rank_{DG}(v) \text{ wenn } \nexists w \in V^R_{DG}, \\ & \text{mit } e_{v,w} \in E^r_{DG} \\ \text{rank}_{DG}(w) & \text{für } sel(t) = rank_{DG}(v) \text{ und } \exists w \in V^R_{DG}, \\ & \text{mit } e_{v,w} \in E^r_{DG} \text{und } active(w) \\ & \text{und } \nexists w' \in V^R_{DG} \text{ mit } e_{v,w'} \in E^r_{DG} \\ & \text{und } \text{rank}_{DG}(w') < \text{rank}_{DG}(w) \\ 0 & \text{für } sel(t) = rank_{DG}(v) \text{ wenn } v.type = \text{QUERY} \end{cases}$$ ◇

In der realen Implementierung benötigen alle Knoten offensichtlich zwei zusätzliche Attribute: ein “Rang”-Attribut und ein “Aktiviert”-Attribut. Die Auswahl der aktuell zu berechnenden Knoten erfolgt dann anhand eines Vergleichs ihres Ranges mit dem aktuell ausgewählten Rang durch Überprüfung des “Rang”-Attributes und, im Fall von Knoten, an denen rückwärtsgerichtete Kanten enden, zusätzlich durch eine Überprüfung das “Aktiviert”-Attributes.

Bemerkung 4.39

1. Es gibt keine rückführenden Kanten, die vom QUERY-Knoten ausgehen, da der QUERY-Knoten den leeren Kopf der Abfrage repräsentiert und daher nicht von einem Literal in einer Regel referenziert werden kann.

2. Es gibt keine rückführenden Kanten, die zu Knoten mit Rang 0 gehen, da eine rückführende Kante immer von dem Knoten ausgeht, der den Kopf der Regel mit dem rekursiven Aufruf repräsentiert. In dieser Regel muß es jedoch mindestens ein Literal vor dem rekursiven Aufruf geben, da der Aufruf nach Definition 3.4 selbst keine Konstanten enthalten kann, der rekursive Aufruf mit Variablen einer Regel, die nur den rekursiven Aufruf enthält, aufgrund der Identität jedes erneuten Aufrufs, nicht terminieren kann. Dies ist jedoch nach Definition 3.1 nicht zulässig.

◇

Durch die Einteilung des Graphen in Ränge ist nicht nur festgelegt, in welcher Reihenfolge die Knoten ausgeführt werden können, sondern auch, welche Knoten parallel oder in beliebiger Reihenfolge ausgeführt werden können:

- sind alle Knoten eines Rangs n ausgeführt worden, und es gibt von keinem Knoten mit Rang n eine rückwärtsgerichtete Kante zu einem Knoten mit Rang m, $m \geq n$, so können nun die Knoten vom Rang $n+1$ ausgeführt werden.
- existiert von einem Knoten mit Rang n eine rückwärtsgerichtete Kante zu einem Knoten mit Rang m, $m \geq n$, aber keine zu einem Knoten mit Rang m', $m' < m$, so ist der nächste Rang mit auszuführenden Knoten der Rang m.

Definition 4.40 *Zustand eines Knotens*

- Der aktuelle Zustand eines Knotens v zu einem diskreten Zeitpunkt $t \in \mathbb{N}$ sei definiert über die Belegung seiner Eingangskanten mit Token:

  $\Omega(v,t) = (tk_1^r, ..., tk_k^r)$, wenn $(tk_1^r, ..., tk_k^r)$, $k, r \in \mathbb{N}$, Eingabetupel des Knotens v zum Zeitpunkt t sind.

- Der Ausführungszustand eines Knotens v zu einem diskreten Zeitpunkt $t \in \mathbb{N}$ sei definiert als:

$$\Omega_\varphi(v,t) = \begin{cases} \emptyset & \text{wenn } \nexists t' \leq t \text{ mit } sel(t) = rank(t) \\ (tk_1, ..., tk_k)^l & \text{wenn } \exists t' \leq t \text{ mit } sel(t) = rank(t) \\ & \text{und } \Omega(v,t') = (tk_1, ..., tk_k)^l \end{cases}$$

active sei dann definiert als:

- $active(v) = true$, wenn $\Omega(v,t) \neq \Omega_\varphi(v,t)$
- $active(v) = false$ sonst. ◇

Definition 4.41 *Aktivierung und Ausführung eines Knotens*

Sei $RDG = (V_{DG}^R, E_{DG}^R, Fun_{DG}^R, TK, rank_{DG}, sel)$ ein ranginduziertes Datenfluß-Berechnungsmodell.

i) *Aktivierung*

   Ein Knoten $v \in V_{DG}^R$ ist aktiviert, wenn gilt: $rank_{DG}(v) = sel(t)$ für $t \in \mathbb{N}$

ii) *Werteberechnung*

   Die Werteberechnung erfolgt unter Anwendung der in Kapitel 3.6 definierten Funktion $\varphi_v \in Fun_{DG}^R$: $M = \varphi_v(M_1, ..., M_n)$, $n \in \mathbb{N}$, wobei $M_1, ..., M_n$ Mengen von Eingabetupel, M eine Menge von Ausgabetupeln ist.

iii) *Ergebnisverteilung*

   Alle Ausgabekanten eines Parameters erhalten dieselben Token: ist $(tk_1^r, ..., tk_{v.out}^r) \in M$, $r \in \mathbb{N}$, so gilt $\pi_r(e_{v.op_{i,j}}) = tk_i^r$ für $i \in \{1, ..., v.out\}$. ◇

Bemerkung 4.42

Definiert man den Zustand des Graphen durch die den Kanten des Graphen zugeordneten Token (siehe Def. 4.40) ist durch Definition 4.41 ein Zustandsübergang definiert. ◇

Folgerung 4.43

Der Folgezustand, der sich nach Ausführung aller Knoten eines Ranges einstellt, ist unabhängig von der Reihenfolge der Ausführung der Knotenoperationen. ◇

Beweis:

Ein Zustandsübergang bewirkt nur an Ein- und Ausgabekanten eines Knotens Änderungen. Da alle Knoten eines Ranges weder gemeinsame Eingangs- noch Ausgangskanten besitzen, ist das Ergebnis des Zustandsübergangs unabhängig von der Reihenfolge der Ausführung der Knoten. ◇

Definition 4.44 *Berechnung*

Eine Berechnung sei definiert als zusammenhängende Teilfolge $t_0, .., t_n \subset \mathbb{N}$ für die gilt:

- $sel(t_0) = 0$

- $sel(t_n) = rank(v)$ mit $v.type =$ QUERY ◇

Bemerkung 4.45

Durch $t_0, t_1, ...$ sei eine Folge diskreter Zeitpunkte definiert.

Die Initialisierung der Berechnung erfolgt mit $t = t_0$, wobei $sel(t_0) = 0$ gelte: alle Knoten v mit $rank(v) = sel(0)$ sind aktiviert und werden ausgeführt.

Für $t \geq 0$ sind zum Zeitpunkt $t + 1$ alle Knoten mit $rank(v) = sel(t + 1)$ aktviert und werden ausgeführt.

Die Berechnung terminiert zum Zeitpunkt $t + 1 = t_n$, wenn für v mit $rank(v) = sel(t_n)$ gilt: $v.type =$ QUERY $sel(t + 1)$. ◇

Bemerkung 4.46

Auf die Einteilung der Knoten in Ränge können Scheduling-Algorithmen angewendet werden, um eine optimalere, d.h. gleichmäßigere Aufteilung der Knoten auf Ränge zu erhalten.

Die Rangeinteilung, wie sie bisher stattfindet, entspricht einem ASAP (*as soon as possible*) -Scheduling. Man erreicht bereits eine deutliche Verbesserung, indem man ALAP (*as late as possible*) -Scheduling anwendet, da erfahrungsgemäß aufgrund der Baumstruktur die Anzahl der Knoten mit höherem Rang abnimmt. ◇

4.2 Ereignisflußgetriebene Ausführung

Das Ereignisflußmodell leitet sich vom Datenflußmodell ab. Der Datenmechanismus übermittelt das Ergebnis einer Befehlsausführung jedoch nur dann den nachfolgenden Befehlen, wenn sich dieses vom zuletzt übermittelten Datum unterscheidet. Übertragen werden daher nur Datumsänderungen. Im Unterschied zum Datenflußmodell müssen die Knoten, bzw. die Eingangskanten an den Knoten daher speicherndes Verhalten aufweisen. Damit dürfen auch Daten nach der Operationsausführung nicht von den Eingangskanten entfernt werden. Der Kontrollmechanismus des Ereignisflußmodells definiert einen Befehl genau dann als ausführbar, wenn auf mindestenes einer Eingangskante ein Ereignis eingetroffen ist und keine weiteren Ereignisse mehr eintreffen können [Müller 96].

Im vorliegenden Berechnungsmodell gibt es zwei Möglichkeiten Datenfluß durch Ereignisfluß zu ersetzen:

- Bei in Schleifen ablaufenden Berechnungen.

- Von Gesamtausführung zu Gesamtausführung.

Experimentelle Untersuchungen haben gezeigt, daß die Anwendung des Ereignisflußprinzips bei der Berechnung in Schleifen nicht den dafür notwendigen Mehraufwand an Verwaltungstätigkeiten aufwiegt. Zudem kann eine ereignisflußgetriebene Berechnung ähnlich wie eine inkrementelle Berechnung in Schleifen mit Negation durch die Nichtmonotonie der Negation zu Problemen und inkorrekten Ergebnissen führen.

Dagegen ist ein ereignisflußgetriebenes Berechnungsmodell sinnvoll, das Änderungen von Gesamtausführung zu Gesamtausführung berücksichtigt, da der Datenflußgraph in sich stetig wiederholender Folge abgearbeitet wird und sich sowohl die Eingabedaten, als auch die an den Knoten erzeugten Daten nur zu einem geringen Anteil ändern.

Auch hier wird jedoch von einer inkrementellen Form der Berechnung abgesehen. Knoten, an denen Änderungen aufgetreten sind, berechnen ihre Ausgaben aus allen vorliegenden Eingabedaten neu. Würden nur diejenigen Ausgaben neu berechnet werden, die sich aus veränderten oder neu hinzugekommenen Daten ergeben, so müssten außerdem Daten gelöschht werden, die durch fehlende Eingaben nicht erneut berechnet werden können. Die Konsequenz ist, daß man zusätzlich zu den normalen, bereits eingeführten Token verschiedene weitere Formen von Token benötigt: Update-Token und Delete-Token. Da der Join, eine der zeitintensivsten Operationen, an den meisten Knoten ausgeführt werden muß und für die korrekte und vollständige Berechnung eines Joins bei nur einer Änderung an einem Eingabewert im allgemeinen ohnehin fast alle anderen Eingabewerte herangezogen werden, ist der Verwaltungsaufwand für die zusätzlichen Tokentypen zu hoch im Vergleich zum Gewinn durch die Ersparnis einiger weniger Operationen.

Der eigentliche Vorteil des Ereignisflußprinzips kommt dann zum Tragen, wenn es ganze Teilgraphen gibt, deren Eingabedaten über eine ganze Reihe von Auswertungen konstant bleiben.

Die im Ereignisflußmodus stattfindende Ausführung macht dabei nicht die jeweils vorhergehende Ausführung und ihr Ergebnis ungültig. Sie berechnet vielmehr ein Ergebnis für das nachfolgende Zeitfenster unter Benutzung bereits erzielter Teilergebnisse, wenn sich deren Eingabedaten seit der letzten Ausführung nicht verändert haben.

4.2.1 Allgemeine Verwendung des Ereignisflußprinzips

Das Ereignisflußprinzip wurde bislang hauptsächlich im Bereich der Fehlersimulation digitaler Schaltkreise von [Hahn 85], [Hahn 92] verwendet. Bei der Fehlersimulation muß eine digitale Schaltung immer wieder durchgerechnet werden, wobei pro Rechengang ein anderes Testmuster anliegt, das sich möglicherweise von Durchlauf zu Durchlauf nicht sehr verändert.

Dies ermöglicht eine nur teilweise Neuberechnung der Schaltung, wenn in den Teilen, in denen sich die Zwischenergebnisse nicht ändern, die Ergebnisse der letzten Rechnung gespeichert werden.

4.2.2 Anwendbarkeit des Ereignisflußprinzips auf das Modell

Im vorliegenden Modell muß der Graph kontinuierlich neu ausgewertet werden, während Änderungen der Daten im Graph nur zum Teil auftreten: die Anzahl der Änderungen im Graph ist direkt abhängig von der Art der Daten in der Datenbank, d.h. von der Häufigkeit mit der sich diese Daten ändern.

Die Daten, auf die ein Steuerungssystem zugreift, um Steuerungsausgaben für das autonome System zu berechnen, lassen sich grob klassifizieren:

- Allgemeine Systemdaten
  (z.B. Maximale Geschwindigkeit, Maximale Steigungsfähigkeit, Maximale Zeit um einen Nothalt durchzuführen)
- Allgemeine Daten über die Mission
  (z.B. Aufgabe, veranschlagte Zeitdauer)
- Allgemeine Daten über die Umgebung
  (z.B. Landkarte, Straßentyp, Geländetyp, Landschaftstyp)
- Daten über den Missionsfortschritt
  (z.B. Anfahrt beendet, Ziel noch zu entdecken)
- Daten über den Zustand des autonomen Systems
  (z.B. bewegt sich, dreht sich, bremst ab)
- Daten über die aktuelle Umgebung
  (z.B. Hindernis in kurzer Entfernung, Straße macht eine Kurve)

In dieser Klassifikation sind die Daten, die statisch sind, zuerst aufgeführt und die Daten, die sehr dynamisch sind, zuletzt. Dazwischen gibt es alle möglichen Abstufungen.

Als Änderungshäufigkeit ist beispielsweise zu erwarten:

- Allgemeine Systemdaten: gar nicht
- Allgemeine Daten über die Mission: von Mission zu Mission
- Allgemeine Daten über die Umgebung: im 10 Minutenbereich
- Daten über den Missionsfortschritt: im Minutenbereich
- Daten über den Zustand des autonomen Systems: im Sekundenbereich
- Daten über die aktuelle Umgebung: im Hundertstelsekundenbereich

Nur die Daten, die sich im Hundertstelsekundenbereich ändern, bewirken bei einem Echtzeitsystem von Graphauswertung zu Graphauswertung eine Neuberechnung von

Teilen des Graphen, da die Auswertung des Graphen im Bereich von Hundertstelsekunden wiederholt werden muß. Alle anderen Daten ändern sich sehr viel seltener und bewirken nur jede 100. oder sogar 1000. Graphauswertung eine Neuberechnung.

An der Klassifizierung der Daten ist außerdem zu erkennen, daß die Daten sich entsprechend ihrer Klassifikation jeweils unterschiedliche Aufgabenbereichen zuordnen lassen. Damit werden die Daten von ganz unterschiedlichen Regeln, bzw. sogar Programmteilen benötigt. Man erhält die in Figur 56 gezeigte Zuordnung von Datentypen zu ganzen Modulen des Steuerungssystems.

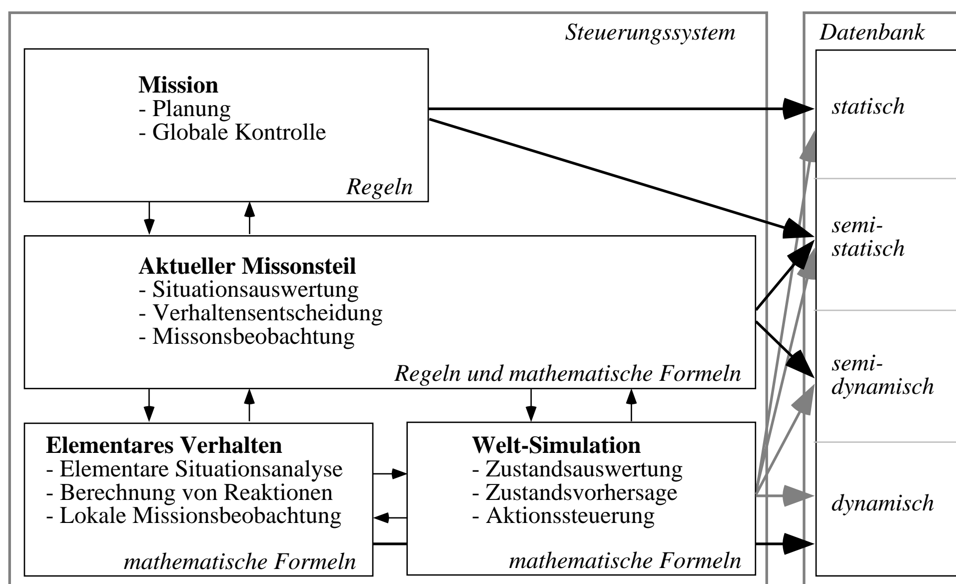


Abbildung 56: Korrespondierende Schichten des Steuerungssystems und Datentypen

Betrachtet man einzelne Module genauer, so läßt sich unter der Voraussetzung, daß die Module des Steuerungssystems modular aufgebaut sind, feststellen, daß im allgemeinen ganze Teile des Moduls nur wenige Klasse von Daten als Eingabe benötigen.

Fazit: Für ein modular aufgebautes Steuerungssystem gilt, daß alle Zweige des Graphen, die nicht die unmittelbare Situationswahrnehmung behandeln, nicht von Auswertung zu Auswertung neu berechnet werden müssen.

4.2.3 Ereignisflußfunktionalität des Datenflußgraphen

Der Kontrollmechanismus des Datenflußprinzips muß zur Realisierung des Ereignisflußprinzips verändert werden:

- *Selektionsregel*
  Da Werte auf den Eingabekanten eines Knotens nach seiner Ausführung nicht gelöscht werden, kann das Vorhandensein von Operanden nicht als Kriterium für das Anstoßen einer Berechnung gewertet werden. Darüberhinaus soll gewährleistet sein, daß sie nur dann stattfindet, wenn am Knoten keine Änderung der Eingaben mehr eintreffen kann. Auch dies ist nur über eine externe Selektionsinstanz möglich.

- *Feuerungsregel*
  Die Auswertung eines Knotens soll nur dann stattfinden, wenn sich mindestens einer der Eingabewerte verändert hat.
- *Verteilungsregel*
  Es sollen nur Ergebnisse verschickt werden, die in diesem Berechnungsschritt tatsächlich neu erzeugt wurden.

Da zur Realisierung von Rekursionen Kanten mit speicherndem Verhalten notwendig sind, die nur dann eine neue Eingabe annehmen, wenn sich diese von der letzten unterscheidet, muß der Datenmechanismus im Vergleich zum ranggesteuerten Datenflußmodell nicht modifiziert werden.

Definition 4.47 *Ereignisgetriebene Aktivierung und Ausführung eines Knoten*

Sei $RDG = (V^R_{DG}, E^R_{DG}, Fun^R_{DG}, TK, rank_{DG}, sel)$ ein ranginduziertes Datenfluß-Berechnungsmodell.

i) *Aktivierung*

Ein Knoten $v \in V^R_{DG}$ ist aktiviert, wenn gilt: $rank_{DG}(v) = sel(t)$ für $t \in \mathbb{N}$ und $active(v) = true$.

ii) *Werteberechnung*

Die Werteberechnung erfolgt unter Anwendung der in Kapitel 3.6 definierten Funktion $\varphi_v \in Fun^R_{DG}$:
$M = \varphi_v(M_1, ..., M_n)$, $n \in \mathbb{N}$, wobei $M_1, ..., M_n$ Mengen von Eingabetupel, M eine Menge von Ausgabetupeln ist.

iii) *Ergebnisverteilung*

Alle Ausgabekanten eines Parameters erhalten dieselben Token, wenn sich das Ergebnis vom Ergebnis der letzten Ausführung unterscheidet:
ist $(tk^r_1, ..., tk^r_{v.out}) \in M$, $r \in \mathbb{N}$, so gilt $\pi_r(e_{v.op_{i,j}}) = tk^r_i$ für $i \in \{1, ..., v.out\}$. $\diamond$

In der realen Implementierung erfolgt die Auswahl der zu aktuell zu berechnenden Knoten bei allen Knoten anhand eines Vergleichs ihres Ranges mit dem aktuell ausgewählten Rang durch Überprüfung des "Rang"-Attributes und durch eine Überprüfung das "Aktiviert"-Attributes.

Ein Knoten berechnet die Ergebnisse für alle Eingabeparameter, unabhängig davon, welche davon sich verändert haben. Damit ist das Ergebnis, das er erzeugt, vollständig. Dieses wird nun an alle Nachfolgeknoten gesendet und ersetzt dort die auf der entsprechenden Eingabekante liegenden Ergebnis der letzten Ausführung, falls es sich von diesem unterscheidet.

Durch das Ersetzen des alten Ergebnis werden automatisch alle ungültig gewordene Werte gelöscht. Es können außerdem neue Werte hinzukommen.

4.2.4 Vergleich mit inkrementeller Bottom-Up Ausführung

Eine optimierende Form der ereignisgetriebenen Ausführung ist eine inkrementelle Berechnung.

Bei der inkrementellen Berechnung wird eine Knotenoperation nur über diejenigen Eingaben ausgeführt, die sich seit der letzten Ausführung verändert haben. Dies verringert

auf der einen Seite die Menge der notwendigen Berechnungen, erhöht andererseits jedoch den notwendigen Verwaltungsaufwand: es muß nun nicht nur möglich sein, neue Ergebnisse zu verschicken, sondern auch Ergebnisse, die in dieser Ausführung nicht mehr berechnet wurden, zu löschen.

Dafür gilt:

- Aktuelle Eingabewerte werden wie bisher mit den Eingabewerten für die vorherige Ausführung verglichen.
- Aktuelle Ausgabewerte müssen nun zusätzlich mit den Ausgabewerten der vorherigen Ausführung verglichen werden.

Man benötigt Token mit einem zusätzlichen Gültigkeitsflag:

- Ein *delete*-Token signalisiert einem Nachfolgeknoten, daß ein zuvor versendeter Ausgabewert bei dieser Knotenausführung nicht mehr berechnet wurde.
- ein *new*-Token signalisert wie bisher die Weitergabe eines (neu berechneten) Wertes.

Die einmalige Ausführung des Modells im Ereignisflußmodus entspricht einer einfachen, semi-naiven Ausführung, da immer nur tatsächliche Veränderungen in den Tupelmengen für die weitere Berechnung berücksichtigt werden:

- es wird der gesamte Graph bottom-up ausgeführt.
- beim Auftreten von Schleifen wird nur dort eine Schleife erneut ausgeführt, wenn sich eine Veränderung in der Menge Eingabewerte ergeben hat. Dabei wird die Schleife nur über für die neu hinzugekommen und die gelöschten Werte ausgeführt.

Die wiederholte Ausführung des Modells im Ereignisflußmodus bewirkt, daß nur die Teile des Graphen neu ausgeführt werden müssen, an denen sich Änderungen an den Blätter, d.h. an den IFACT-Knoten durch Veränderung der Daten in der extensionalen Datenbank ergeben haben.

Dies entspricht Update-Database Modellen, wie sie z.B. aus [Spruit, 95, Montesi 97] bekannt sind.

Bemerkung: In der Realität hat sich gezeigt, daß der Verwaltungsaufwand für verschiedene Typen von Token hoch ist. Zusätzlich führt das Versenden von *delete*-Token z.B. bei Knoten mit Join-Funktionalität zu einer vollständigen Neuberechnung, da das zu löschende Tupel im allgemeinen durch das im Join enthaltene Kreuzprodukt in nahezu alle Ergebnistupel miteingeht.

Daher erweist es sich als kostengünstiger, grundsätzlich an allen Knoten, an denen eine Änderung der Eingabetupel stattfindet, eine vollständige Berechnung über alle Eingabewerte durchzuführen und die Ergebnisse vollständig an Nachfolgeknoten weiterzugeben.

4.3 Mehrschichten-Ausführung

Unter Mehrschichten-Ausführung versteht man die gleichzeitige Ausführung mehrerer Berechnungsfronten im Graphen. Dies kann durch eine Zusammenfassung von Rängen des Graphen zu Schichten derart geschehen, daß Berechnungen, die in verschiedenen Schichten ausgeführt werden, voneinander unabhängig bleiben. Dann kann eine neue Berechnung gestartet werden, bevor die vorherige beendet wurde.

Wird das Berechnungsmodell auf einem Mehrprozessorsystem mit einer hohen Anzahl von Prozessoren ausgeführt, läßt sich mit Hilfe des Mehrschichten-Verfahrens eine bessere Auslastung der Prozessoren und damit vor allem ein höherer Durchsatz erreichen.

Das Mehrschichten-Verfahren ermöglicht außerdem eine, über die in der Ausgangsspezifikation hinausgehende, erweiterte Alarmbehandlung. Die frühe Erkennung einer Alarmsituation kann durch Integration entsprechneder Funktionsknoten (FUNC-Knoten) den Stop fortgeschrittener Berechnungsfronten und die bevorzugte Behandlung der Alarmsituation ermöglichen.

4.3.1 Multiple Berechnungsfronten

Die Rangordnung des Graphen eröffnet bei Verfügbarkeit mehrerer Prozessoren eine zusätzliche Möglichkeit zur Optimierung der Auswertung. Da die Auswertung im vorliegenden Berechnungsmodell strikt in eine Richtung fortschreitet, kann eine neue Graphauswertung gestartet werden, bevor die letzte beendet wurde.

Um zu verhindern, daß es dabei zu einer Verletzung der Unabhängigkeit der Berechnungen kommen kann, ist der Graph in Schichten einzuteilen. Dabei liegt den Schichten die Aufteilung des Graphen in Ränge zugrunde. Eine Schicht umfaßt im allgemeinen mehrere Ränge. Die Aufteilung muß so gewählt sein, daß es nur Kanten gibt, die von Knoten einer Schicht ausgehend, entweder an Knoten der gleichen Schicht enden, oder aber an Knoten einer direkt darüberliegenden Schicht. Es darf also keine Kanten geben, die eine ganze Schicht "überspringen". Zusätzlich sollte eine Rekursionsschleife vollständig in einer Schicht liegen, um ein echtes "Vorwärtsschreiten" der Auswertungsfront zu gewährleisten.

In jeder zweiten Schicht kann dann eine unabhängige Auswertungsfront arbeiten. Zwischen zwei Berechnungsfronten, d.h. zwischen zwei aktiven Schichten, muß grundsätzlich eine Schicht liegen, in der aktuell keine Berechnungen stattfinden. Sie wird daher Pufferschicht genannt. Kanten aus einer unteren, aktiven Schicht, die diese Schicht verlassen, können aufgrund der oben genannten Bedingungen für die Aufteilung des Graphen in Schichten nur in der Pufferschicht enden. Damit können Ausgaben einer unteren Berechnungsfront nicht in die übernächste oder eine noch weiter oberhalb liegende, ebenfalls aktive Schicht gelangen und so die Unabhängigkeit der Berechnungen gefährden.

Die folgende Definition beschreibt formal die Einteilung der Knoten des Graphen in Schichten basierend auf der Einteilung der Knoten in Ränge.

Definition 4.48 *Schicht*

Sei $DG_{\mathcal{L}}^{R} = (V_{DG}^{R}, E_{DG}^{R}, Fun_{DG}^{R})$ ein Datenflußgraph mit Zyklen. $E_{DG}^{vm}, E_{DG}^{r} \subset E_{DG}^{R}$ sei definiert wie in Definition 3.56.

Die Funktion *layer* sei definiert als:

- $layer\ (v) = 0$ wenn $rank(v) = 0$.
- $layer\ (w) < layer\ (w') \Leftrightarrow (\nexists v \in V^{R}_{DG}$ mit $e_{v,w}, e_{v,w'} \in E^{vm}_{DG}$ und $rank(v) \leq rank(w)$ und $\nexists e_{w',w} \in E^{r}_{DG})$ ◇

Figur 57 zeigt allgemein die Rangordnung eines Graphen. IFACT-Knoten haben grundsätzlich den Rang 0, da sie keine Eingangskanten besitzen.

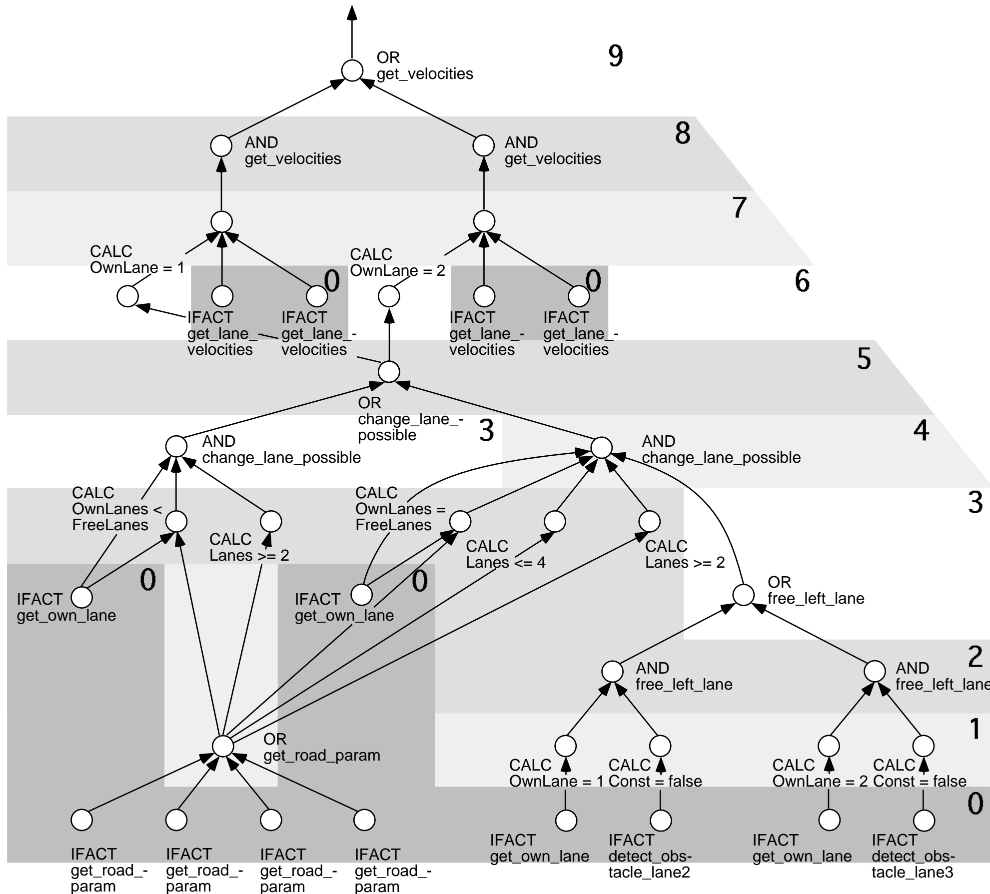


Abbildung 57: Rangordnung eines Graphen

Die in einer Auswertungsfront aktiven Knoten aus einer Schicht schicken ihre Ergebnisse zu Knoten der gleichen oder der direkt nachfolgenden Schicht. Keine nachfolgende Schicht wird dabei übersprungen.

Abbildung 58 zeigt eine mögliche Schichteneinteilung für den Graphen aus Abbildung 57, wobei die gewählte Mindestanzahl von Rängen pro Schicht zwei ist.

Der Wechsel verschiedener, parallel ausgeführter Berechnungsfronten muß möglichst gleichzeitig in die jeweils nächsthöhere Schicht stattfinden. Daher darf der Wechsel einer Pufferschicht, an deren Knoten Ergebnisse aus der darunterliegenden, aktiven Schicht angekommen sind, zu einer aktiven Schicht erst dann erfolgen, wenn
- alle Knoten der darunterliegenden, aktiven Schicht ausgeführt wurden,
- keine Knoten anderer aktiven Schicht mehr ausgeführt werden müssen.

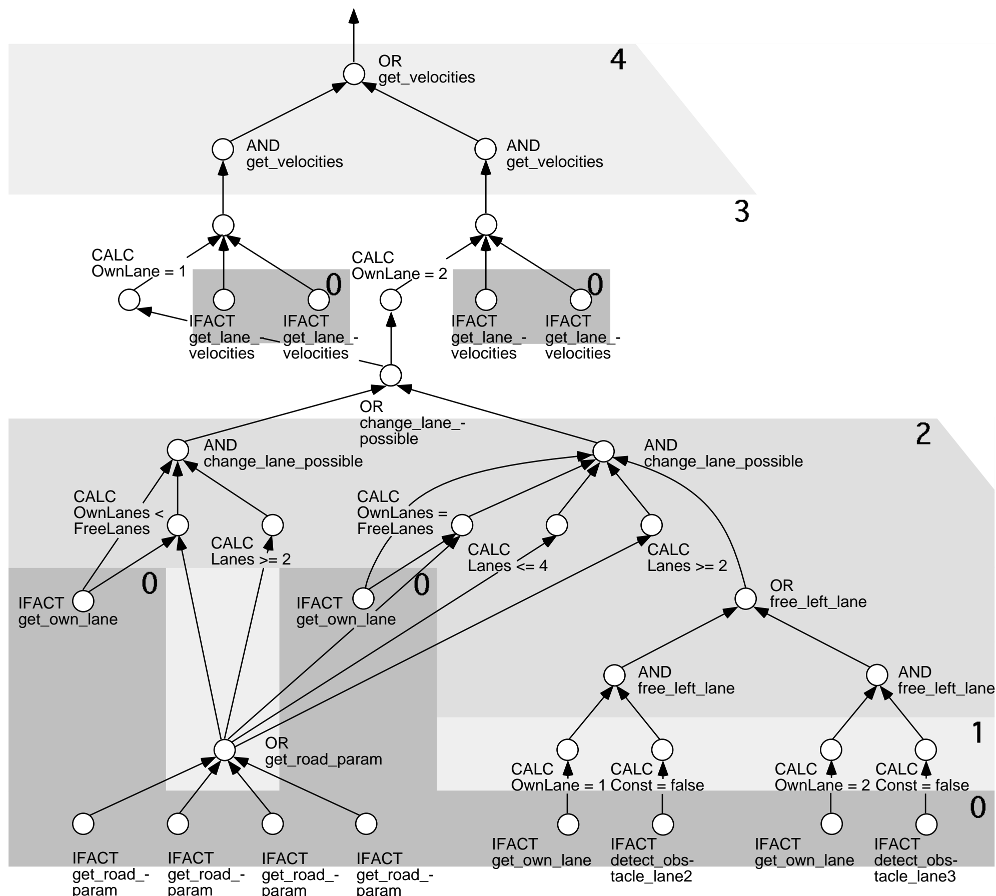


Abbildung 58: Layering eines Graphen

Alle bislang aktiven Schichten werden dann gleichzeitig zu Pufferschichten und umgekehrt.

Enthalten die Schichten unterschiedlich viele Ränge, so müssen Berechnungsfronten, die Schichten mit wenigen Rängen verlassen wollen, lange auf Berechnungsfronten in Schichten mit vielen Rängen warten. Ist die Anzahl der Ränge pro Schicht stark unterschiedlich, z.B. gibt es Schichten mit der doppelten Anzahl von Rängen wie andere Schichten, empfiehlt es sich daher, Schichten mit wenig Rängen, die direkt übereinander liegen, zu einer einzigen Schicht zusammenzufassen. Optimal ist es, wenn alle Berechnungsfronten etwa gleich lange zum Durchlaufen ihrer jeweilig aktiven Schicht benötigen. Eine ideale Aufteilung stellt daher eine möglichst gleichmäßige Aufteilung der Ränge in Schichten dar.

Für eine geeignete Anzahl von Berechnungsfronten im Graphen sind nun zwei Kriterien bestimmend:

1. Die maximale Anzahl der Schichten s.
   Es können dann maximal $\frac{s}{2}$ Berechnungsfronten gleichzeitig ausgeführt werden.

2. Die maximale Anzahl der zur Verfügung stehenden Prozessoren.
   Ist k die maximale Anzahl der Knoten pro Rang, so sind maximal k Prozessoren ausgelastet. Stehen $m * k$ Prozessoren zur Verfügungen, so könnten m Berechnungsfronten gleichzeitig ausgeführt werden.

Die Anzahl der Berechnungsfronten n bestimmt sich somit zu: $n = \mathsf{min}\ (\frac{s}{2}, m)$.

4.3.2 Priorisierung von Alarmbehandlungen

Treten Alarmsituationen auf, so sollten diese so früh wie möglich erkannt und behandelt werden können. Schreiten mehrere Berechnungsfronten gleichzeitig im Graphen fort, so ist eine zusätzliche Beschleunigung der Reaktionszeit dadurch möglich, daß bei Auftreten einer Alarmsituation alle anderen, weiter fortgeschrittenen Berechnungsfronten gestoppt werden können und anschließend eine entsprechende Reaktion bevorzugt durchgeführt werden kann.

Voraussetzung dafür ist, daß der Graph eine, zu einem späten Ausführungszeitpunkt stattfindende, entsprechende Alarmbehandlung enthält. Das gewünschte Verhalten erreicht man dann durch Einfügen eines Graphfragments mit "Verwaltungsknoten" in den Schichten, an denen die Reaktion, bzw. Alarmbehandlung stattfindet und entsprechender Signalkanten zu ihnen von den Stellen an denen eine Alarmsituation festgestellt wird.

Die Einteilung des Graphen in Schichten darf durch die Signalkanten nicht berührt werden. Vielmehr sind Signalkanten von "normalen" Kanten zu unterscheiden und damit die einzigen Kanten, die Schichten "überspringen" dürfen.

Abbildung 59 zeigt das Graphfragment.

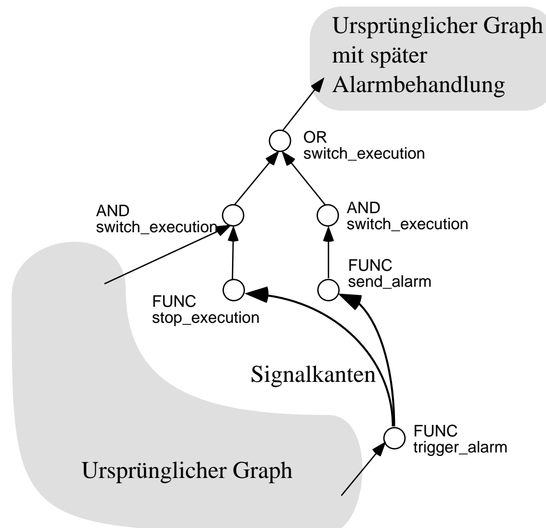


Abbildung 59: Graphfragment mit Verwaltungsknoten

Man benötigt in der Ausgangsspezifikation ein spezielles Systemprädikat das die bevorzugte Alarmbehandlung anstößt. Dieses wird durch einen entsprechenden FUNC-Knoten realisiert: FUNC trigger_alarm(...).

Das Graphfragment für den Empfang des Signals muß dann aus zwei weiteren FUNC-Knoten, stop_execution(...) und send_alarm(...), entsprechenden AND-Knoten und einem OR-Knoten bestehen, um die Verbindung mit dem ursprünglichen Graphen und die

Verwaltung der entsprechenden Funktionalität, den Stop anderer Berechnungsfronten und das Anstoßen der Alarmbehandlung, zu leisten.

Der FUNC-Knoten stop_execution ist dabei zuständig dafür, daß alle anderen Berechnungsfronten im ursprünglichen Graphen gestoppt werden. Dies kann dadurch erreicht werden, daß er durch jede ankommende Berechnungsfront mit einer beliebig zu wählenden Eingabe getriggert wird und als Ergebniswert *failure* ausgibt, wenn auf der Signalkante ein entsprechendes Signal ankommt. Zusätzlich wird ein AND-Knoten benötigt, der den FUNC-Knoten mit dem ursprünglichen Teil des Graphen verbindet und die korrekte Reaktion auf das *failure*, nämlich die Nicht-Weitergabe aller von Unterknoten ankommenden Werte, übernimmt.

Der zweite FUNC-Knoten send_alarm übernimmt das Anstoßen der Alarmbehandlung, wenn er über die Signalkante entsprechende Werte erhält. Er muß über einen zusätzlichen AND-Knoten und dann über einen OR-Knoten mit dem Rest des Graphen werden, damit die von ihm gelieferte Informationen unabhängig vom Rest des Graphen weitergeleitet werden können.

Abbildung 60 zeigt das Beispiel aus Abschnitt 4.3.1, wobei Schicht 5 die neu hinzugefügten Knoten enthält und zwischen Schicht 2 und 5 die Signalkanten eingefügt wurden.

Im Fall einer Alarmsituation werden alle bislang angestarteten Berechnungsfronten, die noch nicht zu einem Ausgabeergebnis geführt haben, irrelevant. Sie können daher nicht nur problemlos gestoppt, sondern auch ganz verworfen werden.

Die Ausführung muß jedoch nach Behandlung der Alarmsituation in den normalen Modus zurückkehren. Da während der Behandlung der Alarmsituation neue Berechnungsfronten starten können (dies wird durch die Signalkanten nicht verhindert), können diese über den FUNC-Knoten trigger_alarm das Ende einer Alarmsituation melden. Die Signale auf den Signalkanten werden zeitverzögert zurückgesetzt, sobald alle dazwischen liegenden, restlichen Berechnungsfronten gesoppt sind. Die Ausführung des Graphen kann dann wie bisher stattfinden.

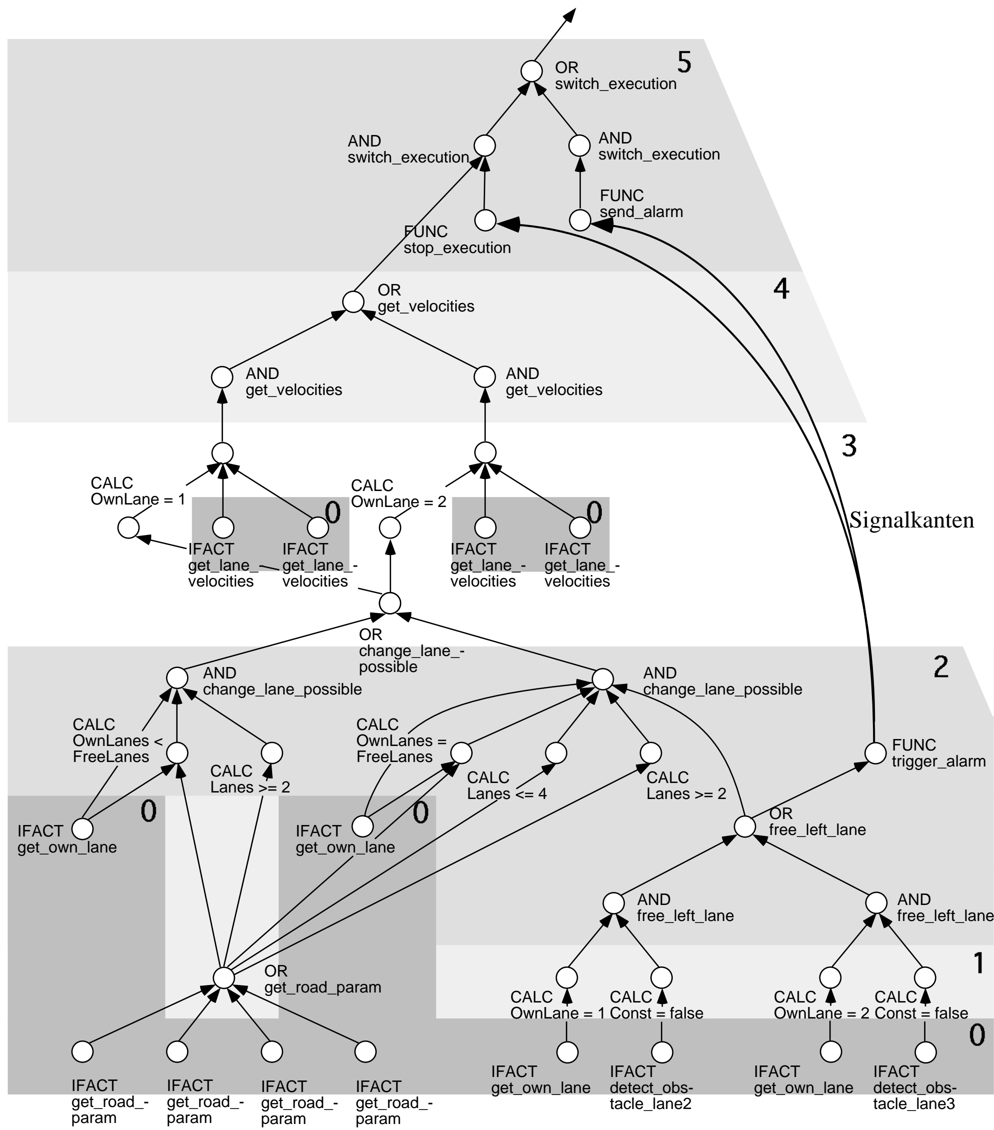


Abbildung 60: Beispielgraph mit Signalkanten und Verwaltungsknoten

4.4 Integration weiterer Komponenten

Die wissensbasierte Komponente eines Steuerungssystem ist nur ein Teil des gesamten Steuerungssystems. Ein vollständiges Steuerungssystem enthält im allgemeinen eine auf mathematischen Algorithmen basierenden Komponente zur schnellen und exakten Berechnung von Grundfunktionen.

Zusätzlich sind jedoch oftmals weitere Komponenten zu finden, wie z.B. ein Modul bestehend aus neuronalen Netzen zum eigenständigen Lernen der Umgebungsbedingungen [Thorpe 91a].

Im speziellen Anwendungsfall von [Dickmanns 95] enthält das Steuerungssystem zusätzlich ein Simulationsmodul, das wesentliche Aufgaben zur leichteren Entdeckung und Erkennung von Objekten in der Umgebung sowie zur Verhaltensvorhersage erfüllt.

4.4.1 Mathematische Formeln

Die Grundfunktionen eines autonom-mobilen Systems, wie zum Beispiel Spurhalten, Bremsen oder Beschleunigen unter verschiedenen Randbedingungen, werden im allgemeinen durch Formeln dargestellt. Es ist wichtig daß diese schnell berechnet werden können und exakte, in Zahlenwerten ausdrückbare Ergebnisse liefern, um die technischen Steuerungselemente des mobilen Systems ansprechen zu können.

Ist das Steuerungssystem in verschiedene Module aufgeteilt, die über die Schnittstelle eines dynamischen Datenspeichers miteinander kommunizieren, wie im Anwendungsbeispiel [Dickmanns 91] gegeben, können die verschiedenen Module unterschiedlich implementiert sein, d.h. je nach Bedarf in einer prozeduralen Sprache für einen von Neumann-Prozessor, oder als Datenflußgraph. Eine wesentlich effizientere Berechnung ist jedoch gegeben, wenn die Schnittstelle und damit die Ausführung eines Kommunikationsprotokolls entfallen kann und die mathematische Berechung von Grundfunktionalitäten zu einem integralen Bestandteil der Datenflußgraph-basierten Auswertung wird.

Für diese Integration gilt nun folgendes:

- Einfache mathematische Ausdrücke lassen sich trivial in Graphen umwandeln, z.B. siehe [Davis 82].
- Um iterative Berechnungen zu realisieren, benötigt man Schleifen. Schleifen sind jedoch elementare Bestandteile des allgemeinen Datenflußmodells und stellen daher kein Problem dar.
- Um komplexe mathematische Funktionen zu berechnen benötigt man Funktions-, beziehungsweise Prozeduraufrufe.
  Dies ist im vorliegenden Modell ein bereits realisiertes Konzept.
- Bei Berechnungen reichen häufig einfache Datentypen nicht mehr aus. Statt dessen werden komplexe Datenstrukturen wie zum Beispiel Bäume benötigt.
  Es gibt jedoch Methoden um komplexe Datenstrukturen wie Arrays oder Bäume in Datenflußmodellen zu verwalten, z.B. die von [Arvind 80, Arvind 89] vorgestellten I-Structures, oder das von [Gaudiot 86] verwendete Array Handling.

Die Berechnung komplexer mathematischer Funktionen mittels Datenflußgraphen wurde bereits von [Müller 96] realisiert. In dieser Arbeit geht es um die Simulation digitaler Schaltkreise auf Circuit-Level. Dabei werden im wesentlichen die für die Simulation auf dieser Ebene notwendigen Differentialgleichungssysteme durch ein Relaxationsverfahren mittels nichtlineare Gleichungssysteme ausgewertet.

4.4.2 Neuronale Netze

Eigenständiges Lernen ist eine wichtige Voraussetzung, um Forderungen nach Flexibilität und Erweiterbarkeit der Einsatzmöglichkeiten eines Steuerungssystems erfüllen zu können.

So kann ein autonom-mobiles System durch Erlernen der verschiedenen Straßenrandkriterien sowohl auf einem Sandweg fahren, als auch auf einer Autobahn.

4.4.2.1 Grundlagen

Neuronale Netze sind Graphen, deren Knoten Neuronen genannt werden. Sie lassen sich in Schichten einteilen, wobei es mindestens zwei Schichten geben muß: Eingabeneuronen und Ausgabeneuronen. Dazwischenliegende Schichten enthalten sogenannte "versteckte" Neuronen, da weder auf deren Eingabe noch deren Ausgabe direkt zugegriffen werden kann. Die Kantenverbindung ist im allgemeinen gerichtet und azyklisch und stell eine n:m Verbindung von Knoten einer Schicht zu den Knoten der nächsten Schicht dar.

Ein neuronales Netz kennt zwei verschiedene Modi: Die Lernphase und die Auswertungsphase.

1. Bei der Auswertungsphase ist jedem Neuron i eine Berechnungsfunktion zugeordnet:

   $$s_i = f_i \sum(w_{ij} * s_j - \Phi_i)$$

   wobei Φ_i einen Schwellwert für das Neuron i angibt,
   w_{ij} die Gewichtung der Kante von Neuron j ist und
   s_j den von Neuron j übergebenen Wert darstellt.

   f_i ist eine Ausgabefunktion, die je nach Art des Netzes unterschiedlich sein kann. Beispiele für f_i sind:

   - eine Schwellwertfunktion:

   $$f_i(X) = \begin{cases} 1 & \text{für } x \geq 0 \\ 0 & \text{sonst} \end{cases}$$

   - eine Sigmoidfunktion:

   $$f_i(X) = \frac{1}{1+e^{-x}}$$

2. Die eigentliche Entscheidung über die Ausgabe wird durch die Gewichtung der Kanten getroffen. Um diese richtig zu setzen, benötigt ein neuronales Netz eine Lernphase.

   In dieser wird das aktuell berechnete Ergebnis mit dem gewünschten Ergebnis verglichen, der Unterschied festgstellt, und anhand des Fehlerwertes eine Korrektur der Gewichtung vorgenommen. Dies ist ein iterativ zu wiederholender Prozeß, bis eine Menge von tatsächlichen Ausgaben mit einer Menge von gewünschten Ausgaben übereinstimmt.

   Es gibt verschiedene Formeln, mit denen dieses Lernen realisiert werden kann. Die bekannteste Lernfunktion ist sicherlich *Backpropagation*. Hierbei bestimmt sich das

Gewicht im nächsten Zeitpunkt aus dem aktuellen Gewicht, das um einen Faktor verändert wird, in den der Anteil des aktuellen Gewichts am Gesamtfehler eingeht.

$$w_{ij}(t+1) := w_{ij}(t) + (-\Delta(t))\frac{\partial E(t)}{\partial w_{ij}}$$

Hierbei gibt Δw_{ij} die Änderung für das Gewicht w_{ij} an. $E(t)$ ist der Fehler der Ausgabe zum Zeitpunkt t im Vergleich zur gewünschten Ausgabe. Die Formel $\Delta(t)\frac{\partial E(t)}{\partial w_{ij}}$ beschreibt eine Gradientenfunktion, $\Delta(t)$ ist dabei die Schrittweite der Gradientenfunktion, oder anders ausgedrückt die Lernrate.

Der Fehler berechnet sich als Quadrat des euklidschen Abstands zwischen dem tatsächlichen Ergebnis s_k jeder Ausgabekante k und dem gewünschten Ergebnis t_k.

$$E := \sum_{k=1}^{p}(t_k - s_k)$$

4.4.2.2 Integration in das Modell

Aufgrund der Graphform neuronaler Netze (Graph mit gerichteten Kanten) lassen sich diese trivial auf Datenflußgraphen übertragen. Die Funktionalität der Neuronen kann dabei dem Datenflußgraphen als Knotenfunktionalität zugewiesen werden. Die Knoten des Datenflußgraphen müssen dabei die Möglichkeit besitzen, an den Eingangskanten zusätzlich zu den eingehenden Werten das Gewicht der Kante abzuspeichern.

Der Datenflußgraph muß darüberhinaus ebenso wie das neuronale Netz zwei verschiedene Ausführungsmodi kennen: Lernen und Berechnen. Das Umschalten muß dabei nicht durch eine Knotenmarkierung geschehen, sondern kann durch die Hardware geschehen: je nach gesetztem Modus wird die eine oder die andere Funktionalität des Knotens ausgeführt.

Entsprechend diesen beiden Modi transportieren die Token im Fall Lernen den Fehlerwert, mit dessen Hilfe das Gewicht einer Eingangskante zu korrigieren ist, zu jedem Knoten. Im Fall Berechnen transportieren sie dagegen tatsächliche Werte.

4.4.3 Fuzzy Logic

Im Bereich der Steuerung autonom-mobiler Systeme gibt es häufig Situationen, in denen

- sich weniger konkreten Aussagen treffen lassen, als vielmehr Abschätzungen bestimmter Situationen,
- die eingegebenen Daten Ungenauigkeiten und Unsicherheiten unterliegen,
- sowie Aussagen nur mit einer gewissen Wahrscheinlichkeit zutreffen.

Um mit solchen Abschätzungen umgehen und sinnvolle Schlußfolgerungen aus diesen ziehen zu können, bietet sich das Konzept der Fuzzy Logic an.

4.4.3.1 Grundlagen

Das Konzept der *Fuzzy Sets* befaßt sich mit der Darstellung von Klassen, deren Zuordnung, z.B. zu einer Menge von Wahrheitswerten $[0, 1]$ nicht exakt bestimmbar ist. Vielmehr geht es um Aussagen wie “fast sicher”, “akzeptabel”, “weniger akzeptabel”. Die eigentliche Aussage entsteht dabei durch sinnvolle Kombinationen verschiedener

Mengen [Dubois 93]. Dabei gibt es gerade im logischen Bereich eine Reihe von Operatoren wie Konjunktion, Disjunktion, oder auch Schnittmengenbildung, die sich auf Fuzzy Sets anwenden lassen.

Die Funktionalität der verschiedenen logischen Operatoren ist dabei [Lee 72]:

- $T(A) = \mathsf{min}(T(B_1), ..., T(B_n)$, wenn $A = B1 \wedge ... \wedge B_n$ gilt.
- $T(A) = \mathsf{max}(T(B_1), ..., T(B_n)$, wenn $A = B1 \vee ... \vee B_n$ gilt.
- $T(A) = 1 - T(B)$, wenn $A = \neg B$ gilt.

Die zweiwertige Logik wird damit zu einem Spezialfall von Fuzzy Logic.

Beispiel:

Eine Regel in Fuzzy Prolog hat z.B. folgende Form:

$\mathsf{youthful}(X) \leftarrow \mathsf{age}\ (X, Y),\ \mathsf{young}(X).$

und die Fakten (Ausschnitt aus der Menge der tatsächlichen Fakten) mit ihren Wahrheitswerten dazu sind:

$\mathsf{age}(A, 29), T(\mathsf{age}(A, 29)) = 0.8.$
$\mathsf{young}(29), T(\mathsf{young}(29)) = 0.75.$

Der Wahrheitswert der Regel errechnet sich dann als:

$T(\mathsf{youthful}(X)) = \mathsf{min}(\mathsf{age}(A, 29), \mathsf{young}(29)) = \mathsf{min}\ (0.8, 0.75) = 0.75.$

◇

Eine weit verbreite Anwendung findet sich im Bereich der Steuerung und Messung [Krantz 71], sowie im Bereich der Robotik [Hiro 93], [Yasu 93]. Hier wird z.B. eine Menge von Situationen durch Fuzzy Sets und die entsprechenden Reaktionen darauf durch Regeln beschrieben. In einer aktuellen Situation wird dann anhand des Grads der Zurordenbarkeit zu definierten Situationen die konkrete Reaktion berechnet.

Das Prinzip von Fuzzy Sets läßt sich auch auf logische Programmiersprachen wie z.B. Prolog übertragen. [Mukaidono 93] hat hierfür einen Protoyp entwickelt, der auf einer erweiterten SLD-Resolution beruht.

4.4.3.2 Integration in das Modell

Die Erweiterung von Logik auf Fuzzy Logic bedeutet im wesentlichen eine Änderung der Knotenfunktionalität. Statt z.B. einem Join muß nun zusätzlich eine Minimumsfunktion für einen And-Knoten implementiert werden. Analog muß für den Or-Knoten zusätzlich eine Maximumsfunktion implementiert werden.

Die Token, die die eigentlichen Werte transportieren, können unverändert bleiben. Dagegen muß das Token, das vom Ergebnisparameter ausgegeben wird, anstelle eines Wahrheitswertes *failure* oder *success*, eine Wahrscheinlichkeit enthalten. Ergebnistoken müssen daher grundsätzlich weitergegeben werden, sie können nicht, wie in Abschnitt 3.4.5 beschrieben, teilweise eingespart werden.

Auch aufwendigere Formen von Fuzzy Logic, wie das in [Mukaidono 93] vorgestellte *Fuzzy Prolog* lassen sich auf Datenflußgraphen übertragen. In diesem Modell kommt eine Gewichtung der Schlußfolgerung, sowie Aussagen über die Sicherheit von Regeln hinzu. Beide Aussagen lassen sich jedoch über eine Ausdehnung des Funktionsumfanges von Knoten realisieren.

4.4.4 Simulationssystem

Die räumliche und zeitliche Simulation von Objekten der realen Welt ermöglicht Voraussagen über das Verhalten der Objekte in naher Zukunft.

Dies erleichtert einerseits die Erkennung und Beobachtung von Objekten, wenn bekannt ist, in welchem Bereich des aufgenommenen Bildes sich ein Objekt befinden kann. Andererseits läßt sich mit einer gewissen Wahrscheinlichkeit sein Bewegungsverhalten voraussagen und durch Angabe von Formeln darstellen.

Simulationssysteme sind im allgemeinen komplex und mit unterschiedlichsten Techniken aufgebaut. Im Zusammenhang mit dieser Arbeit, kann daher nur die Anbinung an die datenflußbasierte Auswertung von Interesse sein.

Die einfache Methode ein Simulationsmodul zu integrieren, ist der Austausch der Informationen über die gemeinsame Schnittstelle, den Datenspeicher. Ein solcher Austausch ist jedoch im allgemeinen langsam und unflexibel. Es ist außerdem nur dann möglich, wenn Simulationsergebnisse für das wissensbasierte Steuerungsmodul ausschließlich jeweils zu Beginn einer neuen Auswertung von Interesse sind.

Wesentlich wichtiger ist jedoch, mit Teilberechnungen des wissensbasierten Steuerungsmoduls Simulationen anzustoßen und mit Hilfe der erzeugten Ergebnisse weitere Entscheidungen fällen zu können. Auf diese Art lasssen sich verschiedenste Szenarien modellieren. Die Einschränkung auf ein relevantes Szenarium erfolgt dann durch Plausiblisierung mittels des Vergleichs mit der weiteren Sensordaten-Analyse in den nächsten Zeitschritten.

Der Datenflußgraph benötigt also die Funktionalität einer Schnittstelle zu einem Simulationssystem, die während einer Ausführungsphase angesprochen werden kann, sowohl für das Anstoßen einer Simulation, als auch die Aufnahme von Simulationsergebnissen.

Wesentliche Eigenschaften, die die Schnittstelle erfüllen muß, sind:

- Die Schnittstelle muß sich auf zwei verschiedene Knoten aufteilen:

  die Simulation kann in einem Mehrprozessor-Rechensystem parallel zur Ausführung, bzw. zu einem Teil der Ausführung des Datenflußgraphen berechnet werden. Sie sollte daher über einen Knoten angestartet werden, während die Ergebnisse ein anderer, im Graphen weiter oben liegender Knoten, aufnimmt.

  Würde der gleiche Knoten, der die Simulation anstartet, auf die Ergebnisse warten, könnte die Ausführung des Graphen (mit dem nächsten Rang) dagegen erst fortfahren, wenn die Simulation beendet ist.

- Der Rang des Knotens, der die Simulation anstartet, muß so klein wie möglich gewählt sein, damit die Simulation so früh wie möglich angestartet werden kann. Dies ist automatisch gegeben, wenn die Zuordnung der Ränge nach der bisherigen Methode geschieht.

  Der Rang des Knotens, der auf das Simulationsergebnis wartet, muß so groß wie möglich gewählt sein, damit so viele Teile des Datenflußgraphen wie möglich parallel zur Simulation ausgeführt werden können. Dieser Knoten sollte daher von der bisherigen Zuordnung der Ränge ausgeschlossen werden. Statt dessen gilt für ihn: der Knoten erhält den Rang n, wenn der Knoten mit dem kleinsten Rang (ohne Berücksichtigung rückführender Kanten) zu dem eine Kante hinführt, den Rang $n + 1$ hat.

Möglich eine Erweiterung der Ausgangsspezifikation durch spezielle Systemprädikte, die im Graphen durch FUNC-Knoten realisiert werden:

1. trigger_simulation(...)
   mit Übergabe geeigneter Parameterwerte für das Anstoßen einer Simulation und
2. get_simulation_results(...)
   mit der Übergabe der entsprechenden Simulationsergebnisse.

Zwischen diesen Knoten besteht nur durch die Anbindung des Simulationssystems eine Verbindung. Im Rahmen des Datenflußgraphen sind sie voneinander unabhängig und daher auch nicht durch eine Kante miteinander verbunden.

Die oben beschriebene Rangeinteilung gilt jedoch nur für die einfache Ausführung. Für eine Mehrschichten-Ausführung müssen beide Knoten FUNC trigger_simulation und FUNC get_simulation_results in einer einzigen Schicht, oder in direkt aufeinanderfolgenden Schichten liegen, das das Berechnungsmodell bislang keine Identifikation verschiedener Berechnungsfronten vorsieht. Würden die beiden Knoten nicht in einer Schicht liegen, ginge daher die Zuordnung der Simulationsergebnisse zur korrekten Berechnungsfront, d.h. die Berechnungsfront, die die Simulation angestartet hat, verloren.

Um zwei Knoten einer Schicht zuzuordnen, bzw. zwei Schichten, die direkt aufeinander folgen, müssen sie durch eine zusätzlich einzufügende Kante verbunden sein, beispielsweise eine "Simulationskante". Darüberhinaus sollte der Unterschied zwischen den Rängen nur gering sein. Es empfiehlt sich also im Fall einer Mehrschichten-Ausführung, das bisher vorgestellte Modell der Rangeinteilung weiterzuverwenden.

5 Implementierung

Die in dieser Arbeit aufgeführten Algorithmen wurden mit dem Programmsystems **EECAS** (**E**ngineering **E**nvironment for **C**ontrol of **A**utonomous **S**ystems) implementiert und getestet.

EECAS besteht aus zwei Komponenten:

- einem Compiler, der die Ausgangsspezifikation in einen Datenflußgraphen übersetzt und
- einem Simulationssystem, mit dem die Ausführung des Datenflußgraphen auf einem von Neumann Rechner nachgebildet wird.

  Die Implementierung des vorgestellten Datenflußmodells im Rahmen eines produktiven Systems kann auf verschiedene Arten erfolgen. Eine interpretative Auswertung auf einer von Neumann-Rechnerarchitektur wird in Abschnitt 5.2 vorgestellt. In Abschnitt 5.3 wird eine geeignete Datenflußrechnerarchitektur für die direkte Ausführung diskutiert. Abschnitt 5.4 beschreibt den Einsatz auf weiteren Zielplattformen.

EECAS wurde in C++ unter Unix erstellt. Für die Oberfläche wurde Motif verwendet.

5.1 Der Compiler

Der Compiler in **EECAS** übersetzt DATALOG$_f^{\neg}$-Programme in Datenflußgraphen.

Abbildung 61 zeigt das Funktionsprinzip des Compilers.

Der Programmablauf umfaßt folgende Schritte, für die u.a. ein detailiertes (Fehler-) Protokoll erstellt werden kann:

- Einlesen der Elemente des DATALOG$_f^{\neg}$-Programms, Normalisierung des Programms (siehe Kapitel 3.1.3) und Erzeugung aller Knoten mit entsprechenden Parametern.

- Aufbau des Strukturgraphen anhand der vom Programm vorgegebenen Struktur (Kapitel 3.2.1). Dabei erfolgt gegebenenfalls eine Duplizierung von mehrfachreferenzierten Teilgraphen (Kapitel 3.3.3).

- Vergabe eindeutiger Parameternamen und Analyse der Parametermodi anhand der Struktur des Graphen und der Verwendungsstellen der Parameter (Kapitel 3.3.4).

- Generierung des Datenflußgraphen unter Duplizierung der Knoten und Erstellung der Kanten anhand der Modusinformation der Parameter (Kapitel 3.4.2 und 3.4.4).

Der Compiler enthält zusätzlich einen Editor zum Erstellen oder Verändern der Ausgangsspezifikation in DATALOG$_f^{\neg}$.

Zusätzlich sind Funktionen wie das Laden und Speichern des resultierenden Datenflußgraphen möglich.

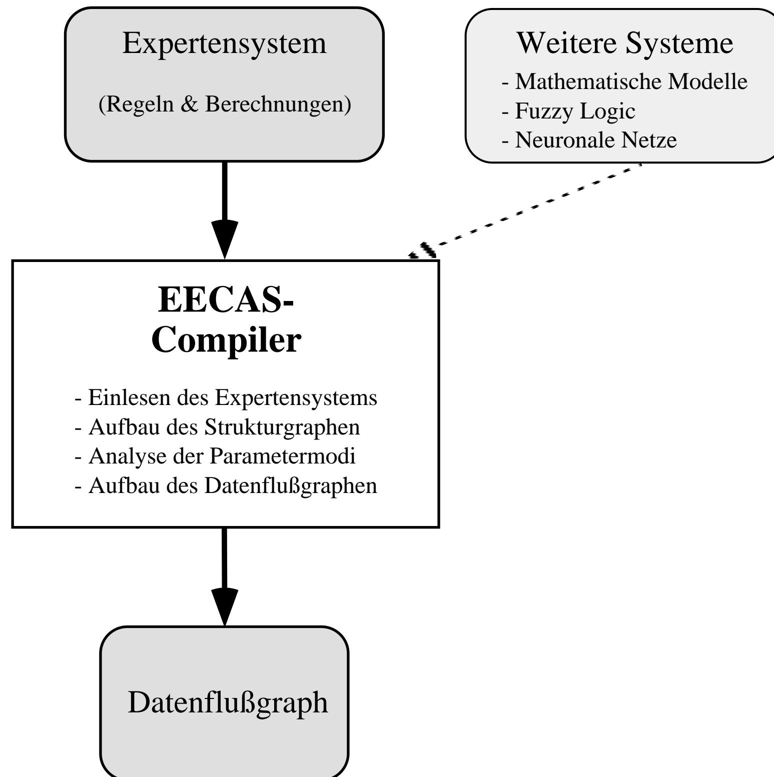


Abbildung 61: Funktionsprinzip des Compilers

5.2 Der Simulator

Der Simulator in EECAS bietet folgende Auswertungsvarianten:

- Auswertung des Datenflußgraphen im ranggesteuerten Datenflußmodus

  In diesem Modus wird bei jeder Ausführung der Graph als Ganzes berechnet. Die Rangsteuerung ist dabei Voraussetzung für die korrekte Realisierung von Rekursionen (Kapitel 4.1.4).

- Auswertung des Datenflußgraphen im ranggesteuerten Ereignisflußmodus.

  Bei diesem Modus werden nur die Teile des Graphen gerechnet, in denen sich Werte geändert haben. Die Rangsteuerung ist grundsätzliche Voraussetzung für die korrekte Behandlung des Ereignisflusses über die korrekte Realisierung von Rekursionen hinaus (Kapitel 4.2.3).

- Auswertung des Graphen mit dem Mehrschichten-Verfahren.

  Zusätzlich zur Auswertung im Datenfluß- oder Ereignisflußmodus wird die jeweils nächste Auswertungsphase sobald wie möglich angestartet (Kapitel 4.3.1).

Die Ausgabe des Simulators besteht aus den “Query-Daten”, d.h. den in der Spezifikation geforderten Ausgaben und aus Bewertungsdaten, mit deren Hilfe die Leistung des Simulators gemessen werden können.

Abbildung 62 zeigt das Funktionsprinzip des Simulators.

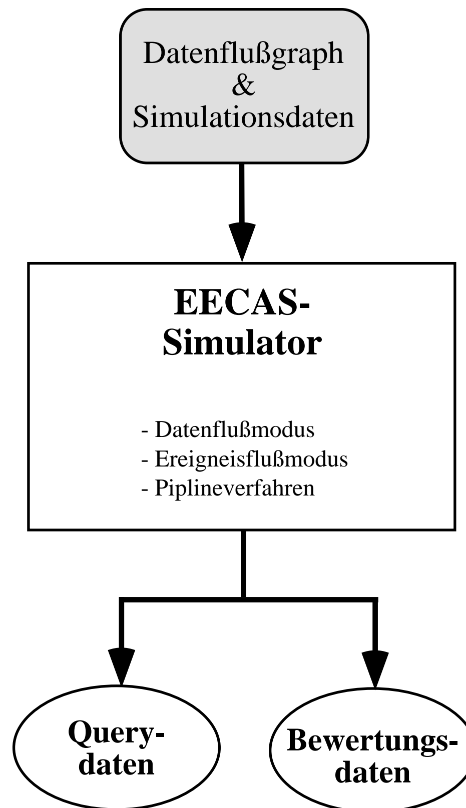


Abbildung 62: Funktionsprinzip des Simulators

5.2.1 Abschätzung der Leistung

Für Systeme die unter Echtzeitbedingungen arbeiten sollen, benötigt man eine zuverlässige Abschätzung der maximalen Ausführungszeit.

Im Datenflußgraphen kann dies durch die Ermittlung des längsten Pfades geschehen. Der längste Pfad sei hierbei jedoch nicht durch die Anzahl der Kanten definiert, sondern durch ein maximales Gewicht, das durch die Anzahl der Knoten und einer Gewichtung nach ihrer maximalen Ausführungszeit gegeben ist.

Eine Gewichtung der Knoten kann wiederum durch Analyse der maximal zu verarbeitenden Eingaben erhalten werden (analog zur Analyse der maximal benötigten Anzahl von Kennzeichnungen, siehe Kapitel 4.1.1.3).

Die Ergebnisse dieser Analyse sind statisch. Die Analyse kann daher zum Übersetzungszeitpunkt stattfinden.

Definition 5.1 *Gewichtung von Knoten*

Sei $DG^R_{\mathcal{L}} = (V^R_{DG}, E^R_{DG}, Fun^R_{DG})$ ein Datenflußgraph mit Zyklen zu einem Strukturgraphen $SG_{\mathcal{L}} = (V, E)$.

Das Gewicht *gew* eines Knotens $v \in V^R_{DG}$ sei berechnet sich durch:

- $ops(v) * \xi_z$ ist die Ausführungszeit einer Knotenoperation für einen (Equi-)Join, wenn dieser durchgeführt wird.

- $ops(v)$ * φ, wenn der Knoten eine Operation ausführt mit Zeitdauer φ.

Dabei ist $ops(v)$ die Anzahl der Tupel, die als Eingabe zu verarbeiten sind, und es gilt:

1. Schnittstellen-Fakten und Programmfakten, die nur Konstanten enthalten, führen pro Aufruf eine Operation, d.h. das Einlesen der Werte, durch.

   Für $v \in V_{DG}^R$ mit $v.in = 0$:
   $ops(v) = 1$

2. Die maximale Anzahl der an einem OR-Knoten ankommenden Tupel läßt sich dabei durch (rekursive) Betrachtung der Anzahl der an den Unterknoten verarbeiteten Tupel berechnen. Da der OR-Knoten die Tupel nur weitergibt, errechnet sich die Anzahl der von ihm zu verarbeitenden Tupel durch Addition über die Anzahl der ankommenden Tupel.

   Für $v \in V_{DG}^R$ mit $v.in \neq 0$, $v.type =$ OR:
   $ops(v) = ops(w_1) + ... + ops(w_n)$,
   wenn gilt: $w_1, ..., w_n \in V_{DG}^R$ und $\exists i, j, j' \in \mathbb{N}$, $i \in \{1, ..., n\}$ mit $e_{w_i.op_j, v.ip_{j'}} \in E_{DG}^R$

3. Programm-Fakten, die Eingaben vom Programm erhalten, Funktionsknoten, AND- und QUERY-Knoten führen einen Join über die ankommenden Tupel durch. Ein Join ist eine Selektion über einem Kreuzprodukt. Das Selektionskriterium (es müssen gleiche Werte in Tupeln an Stellen mit gleichem Attribut sein), läßt sich jedoch nicht statisch bestimmen. Die Anzahl der maximal zu verarbeitenden Tupel errechnet sich daher aus dem Produkt der von verschiedenen Knoten ankommenden Anzahl von Tupeln.

   Für $v \in V_{DG}^R$ mit $v.in \neq 0$, $v.type =$ FACT, $v.type =$ FUNC:
   $v.type =$ AND, $v.type =$ QUERY:
   $ops(v) = ops(w_1) \otimes ... \otimes ops(w_n)$,
   wenn gilt: $w_1, ..., w_n \in V_{DG}^R$ und $\exists i, j, j' \in \mathbb{N}$, $i \in \{1, ..., n\}$ mit $e_{w_i.op_j, v.ip_{j'}} \in E_{DG}^R$

   Dabei gilt für "$\otimes$" aufgrund der Eigenschaften des Joins:

   - $ops(w_i) \otimes ops(w_j) = ops(w_i)$ für $i = j$.

   - $ops(w_i) \otimes ops(w_j) \leq ops(w_i) * ops(w_j)$ für $i \neq j$.

   Eine genauere Abschätzung der Größe des Join-Resultats läßt sich beispielsweise mit Hilfe der Join-Trefferrate [Silberschatz 86] angeben. Da jedoch im vorliegenden Anwendungsfall die Werte einem unendlichen Wertebereich (Domain) entstammen, können standardisierte Modelle nicht angewendet werden. Die Abschätzung ist daher nur empirisch ermittelbar.

4. Für Not-Knoten gilt ebenfalls, daß sie über alle Eingaben bis auf diejenige, die vom direkten Unterknoten kommt, einen Join durchführen.

   Für $v \in V_{DG}^R$ mit $v.in \neq 0$, $v.type =$ NOT:
   $ops(v) = ops(w_1) + ... + ops(w_n)$,
   wenn gilt: $w_1, ..., w_n \in V_{DG}^R$ und $\exists i, j, j' \in \mathbb{N}$, $i \in \{1, ..., n\}$ mit $e_{w_i.op_j, v.ip_{j'}} \in E_{DG}^R$,
   aber nicht $e_{v,W} \in E$

Die Knotenausführungszeit errechnet sich dann aus der Anzahl der im Knoten zu verarbeitenden Tupel mal der entsprechenden Knotenausführungszeit (φ oder ξ). In einzelnen Fällen, wie z.B. beim FUNC-Knoten kann sich diese auch aus φ und ξ zusammensetzen.

- $v.type$ = IFACT: $gew(v) = \varphi_{z,IFACT}$ ($ops = 1$)
- $v.type$ = FACT: $gew(v) = ops(v) * \xi_z$
- $v.type$ = FUNC: $gew(v) = ops(v) * \xi_z + ops(v) * \varphi_{z,FUNC}$
- $v.type$ = AND: $gew(v) = ops(v) * \xi_z$
- $v.type$ = QUERY: $gew(v) = ops(v) * \xi_z$
- $v.type$ = OR: $gew(v) = ops(v) * \varphi_{z,OR}$
- $v.type$ = NOT: $gew(v) = ops(v) * \xi_z + ops(v) * \varphi_{z,Not}$
- $v.type$ = RECDOWN: $gew(v) = ops(v) * \varphi_{z,RECDOWN}$ ◇
- $v.type$ = RECUP: $gew(v) = ops(v) * \varphi_{z,RECUP}$

Bemerkung 5.2

Die Abschätzung des Joins läßt sich bezüglich der Graphstruktur weiter optimieren.

Im schlechtesten Fall kann ein Kreuzprodukt von n Tupel von w_i mit m Tupeln w_j für $i \neq j$ als Ergebnis $n * m$ Tupel ergeben.

Für n Tupel von w_i mit n Tupel von w_i ergibt der (Equi-)Join jedoch immer n Tupel als Ergebnis.

Da gilt: $w_1 \bowtie_= w_1 \bowtie_= w_2 = w_1 \bowtie_= w_2$

läßt sich obige Abschätzung optimieren:

für v mit v führt einen Join durch, gilt:
$ops(v) = ops(w_1) \times .... \times ops(w_{k_1-1}) \times ops(w_{k_m+1}) \times .... \times ops(w_l) \times .... \times ops(w_n)$, wenn gilt: $w_1, ..., w_n \in V^R_{DG}$, $k_1, ..., k_m, l \in \{1, ..., n\}$, $l \neq k_1, ..., l \neq k_m$ und $\exists i, i', j, j' \in \mathbb{N}$, $i \in \{1, ..., n\}$ mit $e_{w_i,v} \in \mathbb{N}$ und $e_{w_{k_1},w_l}, ..., e_{w_{k_m},w_l} \in E^R_{DG}$. ◇

Bemerkung 5.3

Im günstigsten Fall gibt es keine alternativen Ergebnisse und damit OR-Knoten. Damit gilt für jeden Knoten v: $ops(v) = 1$. ◇

Definition 5.4 *Gesamtausführungszeit*

Ist k_r die Anzahl der Knoten des Ranges r, p die Anzahl der zur Verfügung stehenden Prozessoren und gilt $p \geq k$, so berechnet sich die maximale Gesamtausführungszeit *Maxop* als Summe über das größte Gewicht $maxop_r$ jedes Ranges r aus einer Sequenz von n Rängen.

Sei $maxop_r$ das maximale Gewicht des Ranges r, n die Anzahl der Ränge. Dann gilt für *Maxop*:

$$Maxop = \sum_{r=1}^{n} maxop_r$$

Können aufgrund einer beschränkten Anzahl von Prozessoren nicht alle Knoten eines Ranges parallel ausgeführt werden, so errechnet sich $maxop_r$ aus der Summe der l größten Gewichte $maxop_{r,i}$, $i \in \{1, ..., l\}$ eines Ranges. l berechnet sich dabei aus $\mathsf{round}(k_r/p)$, wobei round eine Funktion ist, die auf den jeweils nächstgrößeren ganzzahligen Wert aufrundet.

Seien $maxop_{r,i}$ für $i = 1, ..., l$ die l größten Gewichte des Ranges r, wenn k_r die Anzahl der Knoten im Rang r, p die Anzahl der Prozessoren und $l = \mathsf{round}(k_r/p)$ ist. n sei die Anzahl der Ränge. Dann gilt für *Maxop*:

$$Maxop = \sum_{r=1}^{n} \sum_{i=1}^{l} maxop_{r,i}$$ ◇

5.2.2 Testergebnisse

Die Echtzeitfähigkeit der wissensbasierten Steuerung eines autonom-mobilen Fahrzeugs soll abschließend anhand eines konkreten Beispiels gezeigt werden. Zu diesem Zweck wurde die Autobahnstrecke Passau-Nord bis Aicha mit einem PKW mit einer Geschwindigkeit bis zu 130 km/h befahren und auftretende Verkehrssituationen wie z.B. Kurven, Steigungen und Gefälle, Geschwindigkeitsbegrenzung an einer Baustelle, andere Fahrzeuge, sowie Überholmanöver protokolliert. Nach diesem Protokoll wurden dann Eingabedaten im Sinne des autonom-mobilen Fahrzeugs VaMoRs (siehe Anhang B) generiert. Die protokollierte Fahrt dauerte 10 Minuten. Gemäß einer angenommenen Taktzeit der Sensordaten von 300 ms wurde das Protokoll in 2000 Datensätze umgesetzt.

Als Steuerungsspezifikation diente das in PROLOG (nach den Prinzipien von DATALOG$_f^{\neg}$) modellierte Modul "Behaviour Decision" von VaMoRs. Dieses wurde zusätzlich um weitere Regeln zur Geschwindigkeits- und Lenkungskontrolle erweitert, die im Originalsystem von anderen Modulen, wie z.B. Vehicle Control, übernommen werden. Die Charakteristik der Spezifikation ist in Abbildung 63 wiedergegeben.

Anzahl der Eingabefakten (u.U. mit mehreren Datenwerten)	31
Anzahl der von den Eingabefakten eingelesenen Datenwerte	42
Anzahl der Abfragen	1
Anzahl der Regeln	110
Anzahl der Knoten des vom System erzeugten Graphen	849
Anzahl der Ränge	23

Abbildung 63: Kennwerte der Steuerspezifikation

Für die periodische Auswertung des Steuerungsgraphen wurde das System mit dem GCC- 2.95.1-Compiler bei höchster Stufe 5 der Code-Optimierung übersetzt. Als Rechnerplattform wurde ein Athlon-PC mit einer Taktfrequenz von 600 MHz und einem Speicherausbau auf 256 MB verwendet.

5.2.3 Rangordnung

Für die Verteilung der 849 Knoten des Steuergraphen wurden drei Scheduling Verfahren geprüft:

1. ASAP (*as soon as possible*)
   ASAP ist ein heuristisches Scheduling, das darauf basiert, daß jeder Knoten so früh wie möglich ausgeführt werden soll, d.h. er muß in den niedrigsten möglichen Rang (Rang mit der kleinsten Kennziffer) einsortiert werden.
2. ALAP (*as late as possible*)
   ALAP ist das Gegenstück zu ASAP. Bei diesem heuristischen Scheduling soll jeder Knoten so spät wie möglich ausgeführt werden, d.h. er muß in den höchsten möglichen Rang (Rang mit der höchsten Kennziffer) einsortiert werden.

3. ALAP^M (*modified ALAP*)
   ALAP^M ist ein modifiziertes ALAP Scheduling, bei dem die Knoteneinteilung zwischen ASAP und ALAP per Zufall stattfindet, wobei die Wahrscheinlichkeit einer Zurordnung zu einem Rang mit der Höhe des Rangs wächst. Dadurch kann erreicht werden, daß auch für Knoten in (nach ASAP) niedrigeren Rängen, die im Verlauf des Verfahrens zeitlich später eingeordnet werden, noch Spielraum bei der Neuverteilung auf Ränge bleibt.

Während einer Auswertung des Graphen können sich die Eingabedaten bereits ändern, was jedoch keinen Einfluß auf die aktuelle Auswertung des Graphen haben darf. Deshalb müssen die Knoten der Eingabefakten vom Scheduling ausgenommen werden und alle dem Rang 0 zugeordnet werden.

Hintergrund für die Verwendung dreier Verfahren war, daß ASAP wie Abbildung 64 zeigt, zu einer überproportionalen Belegung der unteren Ränge (d.h. der Ränge mit niedrigen Kennziffern) führt. Solange als Plattform ein Monoprozessor-System dient, ist dies unerheblich. Da das Anwendungsbeispiel aber auch die Möglichkeiten des Einsatzes eines Mehrprozessor-Systems zeigen soll, wurde auch das alternativ geprüfte ALAP-Scheduling, das zwar das überproportionale Belegen der unteren Ränge vermeidet, dafür aber gemäß Abbildung 64 höhere Ränge überproportional belegt, verworfen und schließlich das Verfahren ALAP^M mit gleichmäßigerem Verteilungsergebnis verwendet.

Für den Zweck dieses Abschnittes kann die Qualität der Knotenverteilung als hinreichend angesehen werden. Für ein Produktionssystem hingegen sei auf bessere Verfahren, wie z.B. das Force Directed-Verfahren [Paulin 89] hingewiesen.

5.2.4 Ergebnisse

Als Ergebnis des 2000-maligen Auswertens des Steuergraphen zeigt Abbildung 65 im Datenfluß- (DF) wie auch im Ereignisflußmodus (EF). Je Rang wird die mittlere Anzahl Knotenausführungen, der elementaren Knotenoperationen und der im Mittel auftretenden Kommunikationsbedarf, gemessen in der Anzahl versendeter Token, angegeben.

Ein Token hat je nach Datenwert eine Größe von 11-19 Byte. Dies setzt sich zusammen aus der Zielknotenadresse (4 Byte Adresse, 1 Byte Parameternummer), dem eigentlichen Wert (4 Byte bei int, 8 Byte bei float, für string werden im Anwendungsbeispiel maximal 12 Byte benötigt) und dem Bitvektor (im Anwendungsfall 2 Byte).

Abbildung 66 zeigt den gemessenen, durchschnittlichen Rechenzeitbedarf im Datenfluß- bzw. im Ereignisflußmodus (in der Abbildung mit DF bzw. EF gekennzeichnet). Der zum Vergleich aufgeführte Grenzwert von 300 ms entspricht dem in VaMoRs realisierten Zeitrahmen der Sensordatenanalyse, mit der bereits 1988 ein (damaliger) Geschwindigkeitsrekord von 90 km/h auf Autobahnen erzielt werden konnte.

Aus den Werten der Abbildung 66 läßt sich ableiten, daß mit der heute verfügbaren PC-Rechenleistung die Steuerungsaufgabe sowohl im Datenfluß- wie auch im Ereignisflußmodus in Echtzeit gelöst werden kann. Die Zeitreserven zum Grenzwert können genutzt werden, die Steuerungsaufgabe durch zusätzliche Regeln weiter zu präzisieren, wobei der Eignisflußmodus die wesentlich weitergehende Option darstellt.

Betrachtet man aber die Steuerung eines Fahrzeugs anstatt in einer Autobahnumgebung in einer Stadtumgebung, so ist leicht vorstellbar, daß dann darüber hinausgehend eine wesentlich umfangreichere Regelmenge notwendig wird. Zusätzlich ist zu vermuten, daß der Analysezeitraum für die Sensordaten deutlich geringer als 300 ms werden muß, um

Rang	Anzahl der Knoten je Rang			Bemerkung
	ASAP	ALAP	ALAP^M	
0	252	252	252	Interface-Fakten
1	**271**	6	**94**	ASAP, ALAP^M: maximale Anzahl von Knoten
2	135	4	89	
3	50	2	68	
4	35	4	50	
5	12	2	41	
6	9	1	29	
7	5	4	18	
8	4	18	11	
9	2	17	35	
10	2	13	28	
11	17	34	31	
12	15	**158**	18	ALAP: maximale Anzahl von Knoten
13	4	97	11	
14	1	83	2	
15	15	64	13	
16	8	39	12	
17	7	11	30	
18	1	22	12	
19	1	12	2	
20	1	4	1	
21	1	1	1	Kopf der Regel zur Abfrage
22	1	1	1	Abfrage
Knoten insgesamt	849	849	849	

Abbildung 64: Verteilung der Knoten auf Ränge bei verschiedenen Scheduling Verfahren

die Reaktionszeit zu verbessern. Deshalb soll, wie bereits angedeutet, auch der Einsatz eines Mehrprozessor-Systems, marktgängig sind z.B. SMP-Systeme mit vier bzw. acht Prozessoren und einem Shared Memory von einem Gigabyte je Prozessor, betrachtet werden. Der Zeitbedarf für die Kommunikation bleibt bei Verwendung von Shared Memory-Kommunikation vernachlässigbar klein im Vergleich zur Knotenausführung.

Die sich ergebende Lastverteilung bei Einsatz eines Mehrprozessor-Systems bei einer Verteilung der Knoten des Graphens mittels des ALAP^M-Verfahrens und die jeweils notwendige, mittlere Rechenzeit je Graphenauswertung zeigt Abbildung 67.

Für diese Abschätzung wurde die Ausführungszeit jeder Knotenauswertung etwas vereinfachend gleich einer Zeiteinheit ($\approx$ 0.2391 ms) als Durchschnittswert gesetzt. Analysiert man den Code des ablaufenden Simulationssystems, so zeigt die Analyse zwar, das der Code für das Ausführen eines AND-Knotens, der eine Join-Operation in Kombination mit der Bildung eines Kreuzproduktes und einer Selektion erfordert, umfangreicher ist, als der Codeumfang für die Ausführung eines OR-Knotens, dafür aber der Code anderer Knoten, z.B. von Faktenknoten, wesentlich geringer ist. Es kann deshalb in guter Näherung mit einem Durchschnittswert agiert werden.

Eine Umrechnung dieser Zahlen in Werte für die erzielbare Steigerung der Auswerteleistung als Funktion der Prozessorzahl zeigt Abbildung 68. In Abbildung 69 ist schließlich die relative Steigerung der Auswerteleistung graphisch im Vergleich zur Gerade für eine lineare Steigerung dargestellt. Diese Abbildung 69 zeigt im Prinzip das für das Verteilen

Rang	mittlere Anzahl der Knoten-ausführungen		mittlere Anzahl der elementaren Kno-tenoperationen		mittlere Kommuni-kationslast (Tokenanzahl)	
	DF	EF	DF	EF	DF	EF
0	252.0	252.0	678.0	121.9	836.0	203.3
1	94.0	11.1	282.0	33.2	156.0	22.0
2	89.0	9.7	505.1	30.9	121.0	13.6
3	68.0	11.1	295.3	49.3	147.4	9.5
4	50.0	11.8	265.2	88.6	81.2	22.2
5	41.0	11.5	166.2	56.8	49.0	12.8
6	29.0	11.9	317.6	38.3	129.0	15.0
7	18.0	8.5	108.6	78.6	33.4	16.0
8	11.0	8.1	113.0	97.2	11.3	7.4
9	35.0	3.4	214.0	8.8	163.0	59.1
10	28.0	3.0	128.0	6.4	15.0	4.6
11	31.0	5.5	397.3	271.9	53.6	10.2
12	18.0	5.4	95.6	39.8	15.7	2.1
13	11.0	3.1	80.9	22.9	13.7	8.2
14	2.0	0.3	9.0	1.4	45.0	9.9
15	13.0	2.7	39.0	1.1	24.0	0.7
16	12.0	2.3	36.0	0.9	21.0	0.6
17	30.0	3.1	156.2	8.5	55.0	2.3
18	12.0	1.3	178.9	4.2	22.0	2.2
19	2.0	0.9	44.0	38.0	13.0	16.0
20	1.0	0.9	14.0	13.0	42.0	38.9
21	1.0	1.0	68.0	68.0	15.0	15.0
22	1.0	1.0	2.0	2.0	0.0	0.0

Abbildung 65: Durchschnittliche Menge von Operationen und Kommunikationen im Datenflußmodus pro Iteration

Grenzwert	Istwert	
	Datenflußmodus	Ereignisflußmodus
300 ms	203 ms	106 ms

Abbildung 66: Vergleich der erzielten Istwerte mit dem Grenzwert

eines Datenfluß-/Ereignisfluß-Graphen auf ein Mehrprozessor-System typische, lineare Ansteigen der Auswerteleistung im Bereich hinreichend großer Last je Prozessor - im vorliegenden Beispiel somit etwa bis zu einer Anzahl von acht Prozessoren.

Für eine Schichteneinteilung des Graphen erweist sich ein reines ALAP-Scheduling am optimalsten, da durch die “as late as possible”-Verteilung der Knoten die Anzahl der Kanten, die mehrere Ränge überspringen müssen, gering ist. Je flexibler die Knoten dagegen auf Ränge aufgeteilt werden, desto geringer wird die Anzahl der Schichten ausfallen. Hier könnte gegebenenfalls eine Scheduling-Technik Abhilfe schaffen, die die gewünschte Größe der Schichten berücksichtigt (z.B. 1 Schicht soll ca. 4 Ränge umfassen) und innerhalb dieser Schicht die Knoten nach Möglichkeit gleichmäßig auf die Ränge verteilt.

Es hat sich jedoch gezeigt, daß sich, trotz der deutlich besseren Gleichverteilung von Knoten auf Ränge bei ALAP^M, bereits bei einem ALAP-Scheduling bei einer Ausführung auf einem Mehrprozessorsystem bereits ein ähnlicher Geschwindigkeitsgewinn erzielen läßt.

	Anzahl der Prozessoren							
Rang	1	2	4	8	16	32	64	128
0	60.25	30.13	15.06	7.65	3.83	1.91	0.96	0.48
1	22.48	11.24	5.74	2.87	1.43	0.72	0.48	0.24
2	21.28	10.76	5.50	2.87	1.43	0.72	0.48	0.24
3	16.26	8.13	4.06	2.15	1.20	0.72	0.48	0.24
4	11.96	5.98	3.11	1.67	0.96	0.48	0.24	0.24
5	9.80	5.01	2.63	1.43	0.72	0.48	0.24	0.24
6	6.93	3.60	1.91	0.96	0.48	0.24	0.24	0.24
7	4.30	2.15	1.20	0.72	0.48	0.24	0.24	0.24
8	2.63	1.43	0.72	0.48	0.24	0.24	0.24	0.24
9	8.37	4.30	2.15	1.20	0.72	0.48	0.24	0.24
10	6.69	3.35	1.67	0.96	0.48	0.24	0.24	0.24
11	7.41	3.83	1.91	0.96	0.48	0.24	0.24	0.24
12	4.30	2.15	1.20	0.72	0.48	0.24	0.24	0.24
13	2.63	1.43	0.72	0.48	0.24	0.24	0.24	0.24
14	0.48	0.24	0.24	0.24	0.24	0.24	0.24	0.24
15	3.11	1.67	0.96	0.48	0.24	0.24	0.24	0.24
16	2.87	1.43	0.72	0.48	0.24	0.24	0.24	0.24
17	7.17	3.59	1.91	0.96	0.48	0.24	0.24	0.24
18	2.87	1.43	0.72	0.48	0.24	0.24	0.24	0.24
19	0.48	0.24	0.24	0.24	0.24	0.24	0.24	0.24
20	0.24	0.24	0.24	0.24	0.24	0.24	0.24	0.24
21	0.24	0.24	0.24	0.24	0.24	0.24	0.24	0.24
22	0.24	0.24	0.24	0.24	0.24	0.24	0.24	0.24
Gesamt-zeit in ms	202.99	102.81	53.09	28.72	15.57	9.35	6.96	5.74

Abbildung 67: Benötigte Zeiteinheiten bei ALAP^M-Scheduling

Anzahl der Prozessoren	Benötigte Zeit in ms		Geschwindigkeits-steigerung	Relative Geschwindigkeits-steigerung
	DF	EF		
1	203.0	106.0	1.0	1.0
2	103.0	53.8	1.97	1.97
4	53.1	27.8	3.82	1.94
8	28.7	15.0	7.08	1.85
16	15.5	8.1	13.06	1.85
32	9.3	4.9	21.77	1.67
64	6.9	3.6	29.28	1.34
128	5.74	3.0	35.38	1.21

Abbildung 68: Leistungssteigerung bei Mehrprozessorsystemen

Bemerkung: Weitere im Test erzielte Daten sind:

1. Anzahl der Schichten bei einer Vorgabe von ca. 4 Rängen pro Schicht:
   - bei ALAP: 8
     (durch die Einteilung der Knoten in möglichst hohe Ränge , d.h. Ränge mit hohen Kennziffern, gibt es wenig Kanten, die einen Rang überspringen),
   - bei ALAP^M: 6

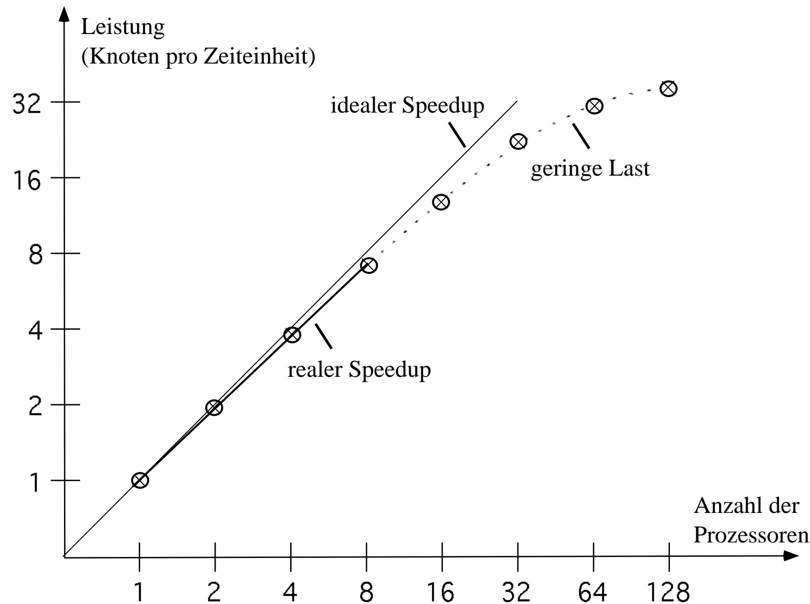


Abbildung 69: Graphische Darstellung der Leistungssteigerung

(durch die stärkere Gleichverteilung der Knoten auf Ränge entstehen Kanten, die Ränge überspringen. Dadurch lassen sich insgesamt geringfügig weniger Schichten bilden).

2. Anzahl der benötigten Bitvektoren: 29

5.3 Ein Datenfluß-/Ereignisfluß-Maschinenmodell

Die optimale Zielplattform für Datenflußgraphen ist eine Datenflußrechnerarchitektur. Diese wird hier, aufgrund der für das Ereignisflußmodell notwendigen Verwendung von Ereignisfluß-Prozessoren anstelle von Datenfluß-Prozessoren *Eventflow Special Purpose Computer* genannt.

In Kapitel 2.2 wurden bereits die grundlegenden Prinzipien der Organisation von statischen und dynamischen Datenflußrechnerarchitekturen vorgestellt.

Aufgrund der durch Rekursionen häufig vorkommenden Schleifen ist die Realisierung des Berechnungsmodells auf einem dynamischen Datenflußrechner der Implementierung auf einem statischen eindeutig vorzuziehen. Einer der bekanntesten und am weitesten entwickelten Datenflußrechner ist *Monsoon*. Er wurde am Massachusetts Institute of Technology (MIT) als Nachfolger der *MIT Tagged-Token Dataflow Architecture* entwickelt [Papadopoulos 90].

Die Realisierung des in dieser Arbeit entwickelten Datenflußmodells birgt jedoch folgende weitere Aspekte für eine geeignete Rechnerorganisation:

- Operationen sind nicht dyadisch
- es gibt mehr als zwei Zielverweise.
- Werte müssen für zukünftige Operationen gespeichert bleiben.

- Zusätzlich zu den Tags ist eine weitere Kennzeichnung der Daten mit Bitvektoren in Token notwendig.

Folgende Einheiten müssen daher in ihrer Funktionalität erweitert werden:

- Der Aktivierungsrahmen muß Platz für mehr als zwei Parameter zur Verfügung stellen. Pro Parameter muß mehr als ein Wert gespeichert werden können.
- Zu jedem Aktivierungsrahmen muß es einen zweiten assozierten geben: in einem Aktivierungsrahmen müssen die Eingabetoken auch nach Aktivierung der Knotenoperation gespeichert bleiben, in einem zweiten müssen Ergebnisse eintreffen können, ohne die alten Token zu überschreiben. Erst nachdem vollständig alle Ergebnisse eingetroffen sind, können die Inhalte beider Aktivierungsrahmen verglichen werden. Nach der Aktivierung eines Knotens kann der Inhalt des Aktivierungsrahmen mit den veralteten Token gelöscht werden, um Platz für neue Token zu schaffen. Durch dieses (Wechselpuffer-) Prinzip ist immer nur einer der bei den Aktivierungsrahmen aktuell für die Knotenoperation im Gebrauch.
- Der Einheit Operand Matching muß zusätzlich eine weitere Einheit zur Seite gestellt werden, die einen Knoten hinsichtlich seiner Aktivierung überprüft, wenn der Vergleich der Inhalte beider zu einem Knoten gehörenden Aktivierungsrahmen einen Unterschied ergibt.
- Zusätzlich zu den Tags ist eine weitere Kennzeichnung der Daten in Token notwendig. Die Form Tag Einheit muß die Generierung von Wertekennzeichnungen (Neugenerierung oder Kombination aus bestehenden Wertekennzeichnung) übernehmen. Außerdem ist eine Erweiterung der Vergleichseinheit (Matching Unit) notwendig, um Token mit Beitvektoren kennzeichnen zu können.

5.4 Weitere Implementierungsplattformen

Die Entwicklung in der Geschwindigkeit von General Purpose Prozessoren schreitet so rasch voran, daß der Geschwindigkeitsvorteil, den Spezialprozessoren zu einem bestimmten Zeitpunkt bieten, bereits in kurzer Zeit durch den stetigen Leistungsgewinn von General Purpose Prozessoren eingeholt wird. Daraus folgt, daß eine interpretative Ausführung eines Datenfluß-, bzw. Ereignisflußgraphen auf einem General Purpose Prozessor eine günstigere Alternative sein kann.

Vorstellbar ist daher die Ausführung des Datenflußgraphen (im Datenfluß- bzw. Ereignisflußmodus) durch folgende Methoden (siehe auch Abbildung 70):

1. *Von Neumann Interpreter*

   Hierbei handelt es sich um die interpretative Ausführung des Datenflußgraphen ähnlich zum EECAS-Simulator auf einer Einprozessor-von Neumann Rechnerarchitektur. Diese Variante weist die jeweils längste Rechenzeit auf.

2. *Hardware Accelerated Interpreter*

   Eine weitere Variante, um bestehende Algorithmen zu beschleunigen, besteht in der Nutzung von beschleunigender Spezialhardware in Form eines Koprozessors. Aktuelle Ansätze aus dem Gebiet des Hardware/Software Codesign zeigen, daß die automatische Generierung von solchen Hardwarebausteinen aus z.B. softwaresprachenbasierten Spezifikationen durchaus bereits im Rahmen des Realisierbaren liegen.

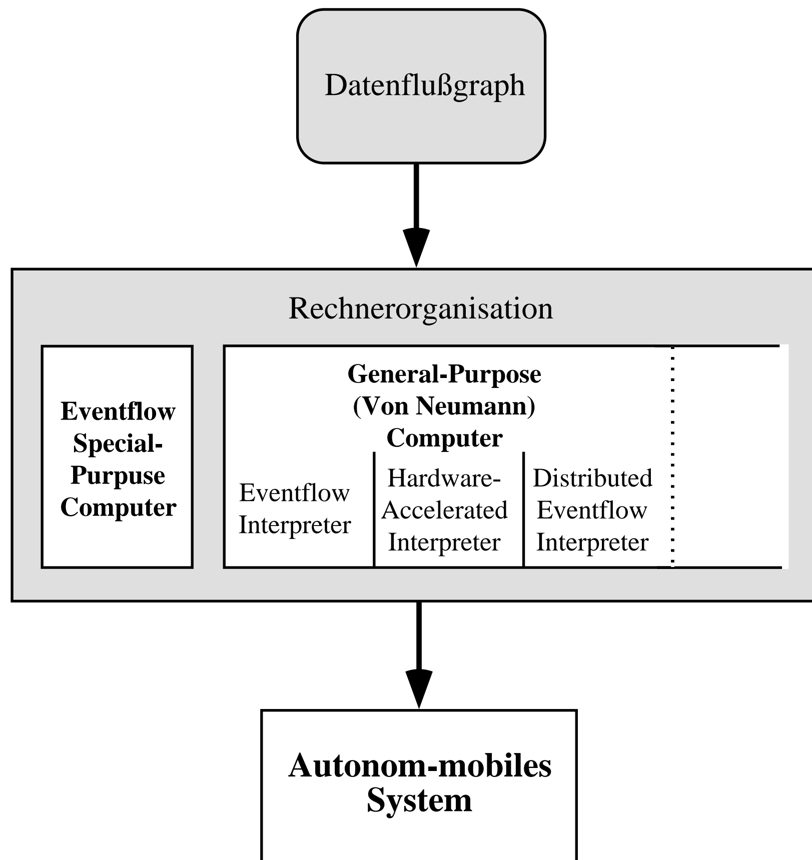


Abbildung 70: Ablauf des Programms

Ein Überblick über die im Rahmen dieses Forschungsgebietes verwendeten Methoden wird in [Micheli 97] gegeben. Beispielhaft soll hier z.B. ein Ansatz erwähnt werden, der eine Programmbeschleunigung verwendet, die von einer Laufzeitanalyse des Programms ausgehend zeitintensive Programmteile in Hardware realisiert ohne weitere Optimalitätsbetrachtungen miteinzubeziehen [Parameswaran 93]. Ein alternativer Ansatz ist z.B. Prozeßgraphen, die im wesentlich Aufrufabhängigkeiten repräsentieren, anhand gegebener Kostenfunktionen heuristisch (mit Hilfe von Greedy-Methoden) auf Hard- und Software aufzuteilen [Gupta 95].

Auch die zunehmende Verbreitung von frei konfigurierbarer Hardware (FPGA) erschließt neue Möglichkeiten zur Programmbeschleunigung für bestimmte Anwendungsgebiete oder als Ersatz für den Einsatz von Spezialrechnern.

Denkbar wäre hier z.B., zum Übersetzungszeitpunkt des Steuerungssystems eine Spezifikation für die Konfigurierung von FPGAs mit Kommunikationsinformation erzeugen zu lassen, die geschwindigkeitskritische Abschnitte durch Hardwareunterstützung beschleunigt.

3. *Distributed Eventflow Interpreter*

Eine Möglichkeit zur Verringerung der Rechenzeit besteht häufig in der Verwendung von mehreren Prozessoren, die entweder auf einen gemeinsamen Speicher zugreifen (*tightly coupled system*) oder jeweils einen eigenen Speicher besitzen (*loosely coupled system*). Verwendet man ein tightly-coupled System mit mehreren identischen,

gleichberechtigten Prozessoren, so bezeichnet man dies als *symmetric multiprocessing.*

Obwohl verteiltes Rechnen auf mehreren Prozessoren heute bereits in vielen Gebieten zum Einsatz kommt, hat es doch einige Nachteile. Die in [Papadopoulos 91] aufgeführten Problematiken wurden bereits in Kapitel 4.1 beschrieben.

Ein prinzipielles Problem beim Einsatz von mehreren Prozessoren besteht jedoch auch in der Notwendigkeit, eine bezüglich Kriterien wie Effizienz oder Kosten optimale oder zumindest "gute" Verteilung der Rechenaufgaben (hier: der Knotenauswertungen) auf die Prozessoren zu finden.

Im Fall der ranggeordneten Datenflußgraphen bestehen die Abhängigkeiten zwischen einzelnen Knotenberechnungen nur zwischen Knoten in verschiedenen Rängen. Demnach können die einzelnen Knoten eines Ranges ohne Nebenbedingungen auf die zur Verfügung stehenden Prozessoren verteilt werden, was einfache Heuristiken wie das "Auffüllen der Prozessoren mit nach Rechenzeitbedarf sortierten Knotenauswertungen" zum Erreichen hinreichend genauer Lösungen für das eigentlich NP-vollständige "Rucksackproblem" [Garey 79] ermöglicht. Bei diesem Problem geht es um die Verteilung verschieden großer Behälter in gleich große Container, so daß alle Container möglichst gleich beladen sind.

Bei einer Ausführung des Datenflußgraphen im datenflußgetriebenen Modus hat man eine statische Verteilung der Rechenlast (*statische Last*). Diese ist bereits bei der Erstellung des Graphen bestimmbar und pro Knoten ausschließlich von der Anzahl seiner Eingangakanten und der Anzahl der über sie ankommenden Token abhängig. Abschnitt 5.2.1 behandelt diese Abschätzungen, die als Gewicht des Knotens $\widehat{\mathrm{gew}}(v)$ bezeichnet werden.

Die statische Last L_s eines Systems zur Berechnung der Knoten in Rang i läßt sich dann ausdrücken durch:

$$L_s = \sum_{v \in V \mid \mathrm{rank}(v) = i} \mathrm{gew}(v),$$

wobei $\mathrm{gew}(v)$ das Gewicht eines Knotens wie in ... definiert bezeichnen soll. Die dynamische Last L_d ergibt sich dann aus

$$L_d = \sum_{\substack{v \in V \mid \mathrm{rank}(v) = i \\ v \text{ hat geänderte Eingabewerte}}} gew(v).$$

Problematischer wird eine solche Aufteilung aber im Fall von Ereignisfluß-Modellen. Während die statische Last des Systems, die durch die Berechnung aller Knotenauswertungen eines Ranges entsteht, im Rahmen der obigen Überlegungen handhabbar ist, erschwert sich das Problem im Falle *dynamischer Last*, d.h. wenn eine ereignisflußgetriebene Auswertungsstrategie verwendet wird. Dies bezeichnet die Methode, daß nur diejenigen Knotenauswertungen berechnet werden müssen, bei denen sich gegenüber der vorhergehenden Auswertung eine Eingangsgröße geändert hat (siehe Kapitel 4.2).

Durch diese Optimierung kann eine deutliche Einsparung an Rechenaufwand erfolgen, der ansonsten in die Berechnung unveränderter Werte investiert werden würde. Allerdings benötigt man für eine effiziente Ausführung eine Einteilung der Knoten in Ränge, so daß alle Knoten eines Ranges nur auf Ergebnisse von Knoten in darunterliegenden Rängen zugreifen. Bei Berechnung in rangmonotoner Reihenfolge kann gewährleistet werden, daß Knoten in niedrigeren Rängen bereits bearbeitet wurden und daß das Fehlen eines Ereignisses bedeutet, daß sich bei den entsprechenden Werten keine Veränderung ergeben hat.

Dazu muß allerdings eine Synchronisation der Berechnung nach der Bearbeitung jedes einzelnen Ranges stattfinden, damit global sichergestellt werden kann, daß alle Eingabewerte für den nächsten Rang vorliegen. Die Kosten für diese Synchronisation belaufen sich prinzipiell auf einen einmaligen Abgleich mit einer zentralen Einheit, die den aktuell zu berechnenden Rang vorgibt. Allerdings wird der Zeitpunkt dieses Abgleichs von der Rechenzeit des bei der Verarbeitung eines Ranges am längsten rechnenden Prozessors bestimmt. Liegt eine ungleichmäßige Verteilung der Rechenlast auf die Prozessoren vor, so kann es passieren, daß an dieser Stelle Wartezeiten für alle anderen Prozessoren in Kauf genommen werden müssen.

Als schwierig stellt sich das Problem des schwer abschätzbaren durch Ereignisfluß induzierten Rechenaufwandes heraus.

Selbst bei einer gleichmäßigen Verteilung der statischen Last (d.h. der Last, die bei Auswertung sämtlicher Knoten entstehen würde) kann durch verschiedene Grade der Änderungshäufigkeiten von Werten eine dynamische Last zwischen 0% und 100% der statischen Last je Prozessor auftreten. Durch die möglicherweise ungleiche Prozessorenauslastung vergrößert sich die maximale Laufzeit der Berechnungen aller Prozessoren gegenüber einer optimalen Aufgabenverteilung.

Während die vorgestellten Abschätzungen für die statische Last, die zur Erzeugungszeit des Graphen ausgewertet werden können, eine hinreichende Genauigkeit für die Verteilung der statischen Last auf mehrere Prozessoren bieten, ist die dynamische Last erst zur Laufzeit bestimmbar und daher schwer abschätzbar.

Eine statistische Analyse des Problems (allerdings bedingt durch die Problemstellung mit identischen Knotenberechnungszeiten) präsentieren Fischer und Kämpke in [Fischer 86]:

Gegeben sei ein beliebiger, aber fest gewählter Rang i und eine Partitionierung der n_i Knoten dieses Ranges auf m Prozessoren, so daß die statische Last L_s gleichmäßig verteilt ist (d.h. jeder Prozessor erhält entweder $\lceil \frac{L_s}{m} \rceil$ oder $\lfloor \frac{L_s}{m} \rfloor$ Last zugeteilt). Für die Knoten dieses Ranges gelte eine (unabhängige) Auswertungswahrscheinlichkeit p_i. Bezeichnet X eine Zufallsvariable, die die Anzahl der insgesamt benötigten verteilten Knotenberechnungen auf Rang i angibt, so ist X binomialverteilt mit Parametern p_i und $\frac{n_i}{m}$. Der Erwartungswert $E[X] = p_i \cdot \frac{n_i}{m}$ gibt die zu erwartende Gesamtlast für die verteilte Berechnung des Ranges an, wenn keine Synchronisationskosten entstehen.

Betrachtet man die Situation, in der die Knoten nach obiger Methode an die m Prozessoren verteilt werden (wobei hier die statische Last als Grundlage der Verteilung dient), so erhält man, wenn man die Rechenlast der einzelnen Prozessoren als binomialverteilte Zufallsvariablen X_j mit $j \in \{1, \ldots, m\}$ mit Parametern p_i und $\frac{n_i}{m}$ modelliert, eine Abschätzung für den Synchronisationsmehraufwand: für die Berechnung der längsten Rechenzeit eines Prozessors wird der Erwartungswert $E[\max\{X_1, \ldots, X_m\}$ verwendet. Der Synchronisationsmehraufwand beträgt demnach

$$E[\max\{X_1, \ldots, X_m\}] - E[X],$$

woraus sich der relative Synchronisationsmehraufwand als

$$\frac{E[\max\{X_1, \ldots, X_m\}] - E[X]}{E[X]}$$

ergibt.

Für diese Abschätzungen werden numerisch stabile Berechnungsverfahren angegeben und Ergebnisse über den durch Synchronisation entstehenden Mehraufwand,

abhängig von der Auswertungswahrscheinlichkeit eines Knotens und der Anzahl der Prozessoren, präsentiert. Dabei zeigt sich, daß z.B. im Fall von 256 Prozessoren und 10000 Knoten für eine Änderungswahrscheinlichkeit im Bereich von 0,01 bis 0,4 der relative Mehrwaufwand im Bereich von 1,3 bis 1,04 liegt, was zeigt, daß das Verfahren den durch Änderungsfluß verringerten Berechnungsaufwand nicht durch Synchronisationsmehraufwand wieder einbüßt.

Weitere Ergebnisse dieser Arbeit zeigen die gute, beinahe lineare Skalierbarkeit des Verfahrens in der Anzahl der Prozessoren.

Hagerer und Lang beschreiben in [Hagerer 91] eine statistische Abschätzung, in der für das daten- und ereignisflußbasierte Berechnungsmodell des MuSiC-Simulationsrechners [Hahn 85, Hahn 85] die Auswirkungen von ereignisgetriebener Aktivierung von Knoten auf die entstehende Rechenlast modelliert wird. Dabei werden verschiedene Möglichkeiten vorgestellt, die Wahrscheinlichkeit für die Auswertung eines Knotens zu beschreiben und, darauf basierend, Methoden zur Partitionierung von Knoten angegeben, die die mit ihrer Ausführungswahrscheinlichkeit gewichteten Knoten gleichmäßig auf die Prozessoren verteilen.

Des weiteren werde die Auswirkungen von Kommunikationskosten untersucht. Dies resultiert im Vorschlag einer Partitionierung der Knoten im Sinne von bevorzugten Knoten "prefered nodes" [Bhuyan 85] vor.

6 Ergebnis der Arbeit

Es wurde ein Modell erstellt, mit dem sich ein in DATALOG$_f^{\neg}$ spezifiziertes wissensbasiertes System zur Steuerung autonom-mobiler Systeme datenflußbasiert effizient parallel ausführen läßt. Dabei ist das spezielle Merkmal dieser Systeme, daß sie

- wenige, unveränderbare Anfragen
- in steter Wiederholung
- auf sich kontinuierlich verändernden Eingabedaten

zu beantworten haben.

Die im Datenspeicher befindlichen Eingabedaten werden jeweils zu Beginn jeder Ausführung eingelesen, so daß sich die kontinuierlich stattfindenden Änderungen nur von Ausführung zu Ausführung bemerkbar machen, im übrigen aber nicht die Ausführung selbst beeinflussen. Errechnete (Zwischen-)ergebnisse werden bei jeder neuen Ausführung je nach Berechnungsmodell entweder generell von Ausführung zu Ausführung gelöscht (Datenflußmodus), oder nur wenn eine Änderung der Eingabedaten erfolgt (Ereignisflußmodus).

Die neben der Parallelisierbarkeit für eine Ausführung in Echtzeit unbedingt notwendige Reduktion redundanter Berechnungen während einer Ausführung, die durch alternative Pfade im Graphen entstehen, geschieht dabei in ähnlicher Form wie bei dem aus dem Bereich der deduktiven Datenbanken bekannten Selection Pushing. Der Unterschied hierzu ist jedoch, daß eine Einschränkung auf relevante Daten nicht durch eine einschränkende Abfrage, d.h. eine Abfrage mit vorgegebenen konstanten Werten, möglich wird, sondern durch direkte und sofortige Verwendung von in anderen, konjunktiv verknüpften Pfaden, berechneten Zwischenergebnissen, ohe dabei jedoch die rein datengetriebene Abarbeitungsrichtung aufgeben zu müssen.

Im folgenden sollen zunächst noch einmal alle bekannten Methoden aufgeführt werden, die im Rahmen der Arbeit verwendet oder modifiziert wurden. Außerdem sollen alle Methoden aufgeführt werden, die neu entwickelt wurden. Der Übersichtlichkeit halber geschieht dies nach der, der Arbeit entsprechenden, Zweiteilung in

- Erstellung des Datenflußgraphen und
- Entwicklung eines Berechnungsmodells für den Datenflußgraphen.

1. **Erstellung des Datenflußgraphen**

   Verwendete bekannte Methoden:

   - Darstellung eines Logikprogramms als Operatorbaum.

   Modifizierte und erweiterte Methoden:

   - Extraktion von Funktion und Listen aus der Parameterliste.
   - Rektifizierung von Regeln.
   - Codeduplizierung zur Erzeugung eindeutiger Bindungsmuster.

   Neu entwickelte Methoden:

   - Die Bestimmung der Parametermodi im globalen Zusammenhang ermöglicht die Unabhängigkeit von der Reihenfolge von Regeln und Literalen im Rumpf von Regel.

- Die Struktur des Datenflußgraphen löst das Problems sehr großer, bzw. unendlicher Wertemengen durch Kanten für direkte Eingaben.
- Die Struktur der Rekursion mit zwei Rekursionsschleifen realisiert eine unidirektionale Ausführung der Rekursion unter Beibehaltung der Stratifizierung des Programms.

2. **Daten- und ereignisflußgetriebenes Ausführungsmodell**

   Verwendete bekannte Methoden:

   - Rangordnung des Graphen für eine modifizierte Auswahl- und Selektionsfunktion.
   - Speicherndes Verhalten der Kanten zur Ermöglichung einer ereignisaktivierten Ausführung.
   - Ereignisflußgetriebene Ausführung für eine deutlich höhere Ausführungseffizienz.

   Modifizierte und erweiterte Methoden:

   - Kontextkennzeichnung zur Verwaltung von Rekursionen.
   - Flexible Datenstruktur der Knoten für eine nicht vorausbestimmbare Anzahl von Eingabetoken.
   - Schichteneinteilung der Knoten des Graphen für die Kooperation von Berechnungsfronten in Alarmsiutationen und eine größere Skalierbarkeit der Rechenleistung.

   Neu entwickelte Methoden:

   - Einführung von Wertekennzeichnungen für die Verwaltung der direkten Eingaben.

Es wurde ein formaler Beweis geführt, daß die Transformationen, die zum Datenflußgraphen führen, gegenüber einem entsprechenden Dependency-Graphen semantikerhaltend sind.

Zusätzlich beinhaltet jedes Kapitel vollständige Beweise, die zusätzlich die Korrektheit der einzelnen verwendeten Algorithmen belegen.

Zusammenfassend läßt sich daher festhalten, daß in dieser Arbeit im Rahmen der Zielsetzung "wissensbasierte Steuerung autonom-mobiler Systeme" ein neues datenflußgetriebenes Berechnungsmodell für Logikprogramme (DATALOG$_f^{\neg}$-Programme) entwickelt wurde, wobei folgende Probleme gelöst wurden:

- Bestimmung der Datenabhängigkeiten bei vollständiger Unabhängigkeit von der Reihenfolge von Regeln sowie von Literalen im Rumpf von Regeln.
- Bottom-Up Verfahren mit garantierter Vermeidung sehr großer bzw. unendlicher Wertemengen trotz Verwendung von z.B. Operationen, Funktionen und Systemprädikaten; dabei notwendiges Finden der Tupelzugehörigkeit von Token in linearer Zeit.
- Unidirektionale Auswertung von Rekursionen unter garantierter Beibehaltung der Stratifizierung.
- Garantierte Auswertbarkeit des Graphen durch eine reale oder emulierte Datenflußrechnerarchitektur bzw. -organisation.

- Nutzung des Ereignisflußprinzips: Rechenleistung wird nur dort investiert, wo sie etwas zum Ergebnis beitragen kann.
- Das Schichtenmodell ermöglicht die Kooperation von zu unterschiedlichen Zeitpunkten gestarteten Berechnungsfronten in Alarmsituationen.

In das Modell können außerdem u.a. folgende Elemente integriert werden:

- Lineare Gleichungs- und Differentialgleichungssysteme für die Berechnung von Basisverhalten,
- Neuronale Netze für selbständiges Lernen,
- Fuzzy Logic für unscharfes Entscheiden,
- Simulationssysteme für die Modellierung.

Alle in der Arbeit vorgestellten Algorithmen wurden im Programmsystem EECAS realisiert. Dieses Programmsystem besteht aus einem Compiler, der eine gegebene Logik-Spezifikation für den wissensbasierten Teil eines Steuerungssystems in einen Datenflußgraphen gemäß dem in dieser Arbeit vorgestellten Berechnungsmodell transformiert. Die zweite Komponente von EECAS ist ein Simulator, der die Ausführung des Berechnungsmodells übernimmt. Damit wurde eine problemorientierte Entwicklungsumgebung geschaffen, mit der Spezifikationen für die wissensbasierte Steuerung autonom-mobiler Systeme erstellt und getestet werden können.

Das autonome Fahrzeug VaMoRs dient der Arbeit als konkretes Anwendungsbeispiel, mit dessen Hilfe Testdaten gewonnen werden konnten, die die vorgenommenen Abschätzungen hinsichtlich Geschwindigkeit und Komplexität bestätigen.

Zusammenfassend läßt sich sagen, daß das Simulationssystem von EECAS bereits auf einem Monoprozessorsystem die Ausführung einer gegebenen, realen Spezifikation unter Echtzeitbedingungen ermöglicht und damit in dieser Form bereits in einem autonom-mobilen Systemen zum Einsatz kommen könnte.

Bei Verwendung entsprechender Mehrprozessorsysteme steht darüberhinaus zusätzlich ausreichend Spielraum für den Einsatz weitaus umfangreicherer, komplexerer Spezifikationen zur Verfügung. Somit ließen sich beispielsweise Stadtfahrten, die wesentlich mehr Regeln benötigen als Autobahnfahrten, realisieren.

A Formale Grundlagen

A.1 Prädikatenlogik 1. Stufe

In diesem Kapitel finden sich ergänzende Definitionen zur Prädikatenlogik, die nicht direkt in der Arbeit benötigt werden.

Definition A.1 *Formeln*

Formeln sind induktiv wie folgt definiert:

i) Jede atomare Formel ist eine Formel.

ii) Sind F und G Formeln, so sind auch $(\neg F), (F \wedge G), (F \vee G), (F \rightarrow G)$ und $(F \leftrightarrow G)$.

iii) Ist F eine Formel und x eine Variable, so sind $(\forall x F)$ und $(\exists x F)$ Formeln.

$F \leftarrow G$ kann auch als $G \rightarrow F$ geschrieben werden. $F \vee G$ steht für $\neg F \rightarrow G$, $F \wedge G$ steht für $\neg(F \rightarrow \neg G)$ $\exists x F$ steht für $\neg \forall x \neg F$.

Eine ausführlichere Darstellung der Formalismen ist in [Lloyd 87] zu finden. ◇

Definition A.2 *Klauseln*

Eine *Klausel* ist eine Formel der Form

$$\forall(L_1 \vee ... \vee L_m)$$

mit Literalen $L_1, ..., L_m$ für $m \in \mathbb{N}$. ◇

Bemerkung A.3

Die Äquivalenz der Klausel

$$\forall(\neg A_1 \vee ... \vee \neg A_m \vee B_1 \vee ... \vee B_n \vee C_1 \vee ... \vee C_k)$$

mit der Formel

$$\forall(A_1 \wedge ... \wedge A_m \wedge \neg B_1 \wedge ... \wedge \neg B_n \rightarrow C_1 \vee ... \vee C_k)$$

wird z.B. in [Lloyd 87] gezeigt.

Die deduktive Regel

$$C_1, ...C_k \leftarrow A_1, ..., A_m, not B_1, ..., B_n$$

ist dabei lediglich eine andere Schreibweise für diese Formel. ◇

A.2 Eigenschaften von Logikprogrammen

Die folgenden Definitionen sind in ausführlicherer Form in [Cremers 94] zu finden.

Definition A.4 *Bezeichnung von Regeln*

Eine deduktive Regel $C_1, ...C_k \leftarrow A_1, ..., A_m, notB_1, ..., B_n$ heißt

disjunktiv	falls $k > 1$ und m und n beliebig.
positiv disjunktiv	falls $k > 1$ und $n = 0$.
disjunktiver Fakt	falls $k > 1$ und $m = n = 0$.
normal	falls $k = 1$ und m und n beliebig.
definit oder positiv	falls $k = 1$ und $n = 0$.
Fakt	falls $k = 1$ und $m = n = 0$.
normale Query	falls $k = 0$ und m und n beliebig.
definite Query	falls $k = 0$ und $n = 0$.
leer	falls $k = 0$ und $m = n = 0$. ◇

Definition A.5 *Normale und definite Programme*

1. i) Eine normale Programmklausel ist eine Klausel.
   ii) Ein normales Programm ist eine endliche Menge von normalen Programmklauseln.
   iii) Eine normale Abfrage ist eine Abfrage.
2. i) Eine definite Programmklausel ist eine Klausel, deren Literale ausschließlich positive Atome sind.
   ii) Ein definites Programm ist eine endliche Menge von definiten Programmklauseln.
   iii) Eine definite Abfrage ist eine Abfrage, deren Literale ausschließlich positive Atome sind. ◇

Definition A.6 *Datalog- und Datalog$^\neg$-Programme*

i) Ein Datalog-Programm ist ein definites Programm, wobei als es in der Sprache außer Konstanten keine weiteren Funktionssymbole gibt.

ii) Ein Datalog$^\neg$-Programm ist ein normales Programm, wobei als es in der Sprache außer Konstanten keine weiteren Funktionssymbole gibt.

Die Begriffe Klausel und Abfrage sind analog definiert. ◇

Definition A.7 *Rektifizierte Literale*

Ein rektifiziertes Literal ist ein Literal, in dessen Parameterliste keine Konstanten auftreten und kein Parametername mehr als einmal vorkommt[Ullman 88]. ◇

Definition A.8 *Sichere und bedingt sichere Regeln*

- Eine Regel ist *sicher*, wenn jede im Rumpf auftretende Variable mindestens in einem positiven Literal im Rumpf vorkommt.
- Eine Regel ist *bedingt sicher*, wenn jede im Rumpf auftretende Variable entweder im Kopf oder in einem positiven Literal im Rumpf vorkommt. ◇

Definition A.9 *Stratifizierte Programme*

- Ein Programm ist genau dann *stratifiziert*, wenn den Literalen eine Ordnungshierarchie derart zugewiesen werden kann, so daß ein negiertes Atom nur in einer Regel auftritt, deren Kopf eine Ordnung strikt größer der Ordnung des Atoms hat, und daß positive Atome nur in Regeln auftreten, deren Kopf die gleiche oder eine größere Ordnung hat.
- Ein Programm ist genau dann *lokal stratifiziert*, wenn den Literalen eine Ordnungshierarchie derart zugewiesen werden kann, so daß ein negiertes Grundtom nur in einer instanziierten Regel auftritt, deren Kopf eine Ordnung strikt größer der Ordnung des Atoms hat, und daß positive Grundatome nur in instanziierten Regeln auftreten, deren Kopf die gleiche oder eine größere Ordnung hat.
- Ein Programm ist genau dann *schwach stratifiziert*, wenn kein Grundatom in dem Moment negativ von sich selbst abhängt, wenn alle instanziierten Regeln mit Literalen im Rumpf, die bereits als falsch bekannt sind, oder mit Köpfen, die bereits als wahr bekannt sind, entfernt werden [Ross 94].
- Ein Programm ist genau dann *modular stratifiziert*, wenn alle rekursiven Komponenten lokal stratifiziert sind, sobald alle instanziierten Regeln mit einem negierten Literal im Rumpf entfernt werden [Ross 94]. ◇

A.3 Substitution und Unifikation

Definition A.10 *Substitution, Unifikation*

i) Eine Substitution ist eine Abbildung von einer endlichen Menge von Variablen in eine endliche Menge von Termen: $\Phi = \{X_1/t_1, ..., X_n/t_n\}$ heißt Substitution, falls $X_1, ..., X_n$ mit $n \in \mathbb{N}$ paarweise verschiedene Variablen sind, d.h. $X_i \neq X_j \ \forall i \neq j \in \{1, ..., n\}$ und $t_1, ..., t_n$ Terme mit $X_i \neq t_i \ \forall i \in \{1, ..., n\}$ sind.

ii) Φ heißt strikt, falls $X_i \notin t_i \ \forall i \in \{1, ..., n\}$ mit $n \in \mathbb{N}$.

iii) Φ heißt Umbennung, wenn $t_1, ..., t_n$, $n \in \mathbb{N}$ Variablen sind.

Eine strikte Substitution bedeutet also die Ersetzung von Variablen in einem Ausdruck durch andere Variablen, oder durch Terme, wobei neu eingeführte Variablen bisher noch nicht in dem Ausdruck enthalten sein dürfen. Dies kann man notfalls durch Umbenennung der neuen Variablen erreichen.

iv) Zwei Terme heißen dann *unifizierbar*, wenn eine Substitution für die Variablen in beiden Termen exisitiert, so daß sie syntaktisch identisch werden. Diese Substitution nennt man auch *Unifikation*. Ein *Unifikator* bestimmt eine allgemeine Instanz und umgekehrt.

v) Eine Substitution Φ heißt *allgemeinster Unifikator* (`mgu`, *most general unifier*) von zwei Termen t_1 und t_2, falls gilt:

- Φ ist Unifikator von t_1 und t_2 und
- für jeden Unifkator Λ von t_1 und t_2 existiert eine Substitution Θ mit $\Lambda = \Theta \circ \Phi$.

◇

Das Kernstück des Berechnungsmodells von PROLOG-Programmen ist der Unifikationsalgorithmus. Unifikation ist die Basis für fast alle Arten automatischer Ableitungen und logischer Schlußfolgerungen aus einem gegebenen Regelsystem.

Definition A.11 *Der Unifikationsalgorithmus*

Eingabe: t_1, t_2.

Ausgabe: $\Phi = mgu\,(t_1, t_2)$ oder $failure$.

Algorithmus:

```
Φ = {};
Stack := {t1 = t2};
failure := FALSE;
while Stack ≠ {} and not failure do
   for {X = Y} ∈ Stack do
      Stack := Stack\{X = Y};
   if var(X) and var(Y) then;
   else if var(X) and (X ∉ Y) then
      forall X ∈ Φ and forall X ∈ Stack do [Y/X];
      Φ := Φ ∪ {X = Y};
   else if var(Y) and (Y ∉ X) then
      forall Y ∈ Φ and forall Y ∈ Stack do [X/Y];
      Φ := Φ ∪ {Y = X};
   else if X = f(X1, ..., Xn) and Y = f(Y1, ..., Yn),
            func(f) and n > 0 then
      Φ := Φ ∪ {Xi = Yi} ∀i = 1, ..., n;
   else
      failure = TRUE;
```

$var(X)$ ist dann erfolgreich, wenn X eine Variable ist, $func(f)$, wenn f ein Funktionssymbol ist. ◇

Mit diesem Unifikations-Algorithmus kann für zwei Terme immer der allgemeinste Unifikator gefunden werden falls dieser existiert [Sterling 86].

Ein wesentliches Element ist dabei der oben nicht aufgeführte *Occur Check*, mit dessen Hilfe erkannt werden kann, ob eine zu ersetzende Variable in dem ersetzenden Term auftritt. Dieser ist wichtig um zyklische und damit unendliche Ersetzungen zu vermeiden.

Auf dem Konzept der Unifikation baut nun der abstrakte Interpreter für ein PROLOG-Program $\mathcal{P}$ auf. $\mathcal{G}$ ist dabei eine Abfrage an das PROLOG-Programm.

Das Prinzip des Unifikationsalgorithmuses ist, daß die Abfrage, die an das Programm gestellt wird, mit dem Kopf einer Regel oder mit einem Fakt des Programms unifiziert werden muß. Gelingt die Unifikation mit dem Kopf einer Regel, so wird die Regel angewendet. Anwendung bedeutet dabei, daß die Literale im Rumpf der Regel zu weiteren Abfragen werden. Die Menge der Abfragen bezeichnet man als *Resolvente*.

Der Algorithmus endet, wenn keine Abfrage mehr unifiziert werden kann. In diesem Fall ist die Abfrage gescheitert und der Algorithums liefert als Antwort *failure* zurück. Oder er endet, wenn die Resolvente leer ist, d.h. alle Abfragen erfolgreich unifiziert werden konnten. Dies ist gleichbedeutend damit, daß für jede freie Variable der ursprünglichen Abfrage eine gültige Belegung gefunden wurde und der Algorithmus *success* zurückgibt [Sterling 86]. Es ist auch möglich, daß im *success*-Fall für eine oder mehrere Variablen mehr als eine gültige Belegungen zurückgegeben wird. Dies ist gleichbedeutend mit einer Lösungsmenge, die mehr als eine erfolgreiche Lösung enthält. Der Algorithums arbeitet solange, bis er alle möglichen Lösungen gefunden hat.

Definition A.12 *Ein abstrakter Interpreter für Logikprogramme*

Eingabe: $\mathcal{G} \in Goals$,
$\mathcal{P} \in Programs$.

Ausgabe: Eine Instanz von $\mathcal{G}$, die eine logische Schlußfolgerung aus $\mathcal{P}$ ist, oder *failure*.

Algorithmus:

```
R := G.
while R ≠ {} do
   for p_0 ∈ R do
      if p'_0 ← p_1, ..., p_n ∈ P, p_0 = Φ(p'_0) then
         R = R\p_0;
         R = R ∪ p_1, ..., p_n;
         R = Φ(R);
         G = Φ(G);
      else
         failure = TRUE;
if R = {} then
   return G;
else
   return failure; ◇
```

Definition A.13 *Berechnung, Berechnungsabfolge*

Die *Berechnung* einer Menge von Abfragen $\mathcal{Q} = \mathcal{Q}_0$ durch ein Programm $\mathcal{P}$ ist eine (möglicherweise unendliche) Folge von Tripeln $(\mathcal{Q}_i, \mathcal{G}_i, \mathcal{C}_i)$, $i \in \mathbb{N}$. $\mathcal{Q}_i$ ist eine Menge von Abfragen, $\mathcal{G}_i$ ist eine Abfrage aus $\mathcal{Q}_i$ und $\mathcal{C}_i$ ist eine Klausel $p_0 \leftarrow p_1, ..., p_n$ in $\mathcal{P}$, die derart umbenannt wurde, daß sie nur Variablennamen enthält, die nicht in $\mathcal{Q}_j$, $0 \leq j \leq i$ vorkommen.

Für alle $i > 0$ ist $\mathcal{Q}_{i+1}$ das Ergebnis der Ersetzung von $\mathcal{G}_i$ durch den Rumpf von $\mathcal{C}_i$ in $\mathcal{Q}_i$ und der Anwendung der Substitution Φ_i, dem allgemeinsten Unifikator von $\mathcal{G}_i$ und dem Kopf von $\mathcal{C}_i$. Oder das Ergebnis ist die Konstante *success*, wenn $\mathcal{G}_i$ die einzige Abfrage in $\mathcal{Q}_i$ ist und der Rumpf von $\mathcal{C}_i$ leer ist, bzw. die Konstante *fail*, wenn $\mathcal{G}_i$ und der Kopf von $\mathcal{C}_i$ nicht unifizierbar sind.

Eine *Berechnungsabfolge* $(\mathcal{Q}_i, \mathcal{G}_i, \mathcal{C}_i)$, $i \in \mathbb{N}$ eines Logikprogramms ist eine Sequenz von Paaren $(\mathcal{G}_i, \Phi'_i)$, wobei Φ'_i eine Teilmenge des allgemeinsten Unifikators Φ_i ist, die im i-ten Schritt für die Variablen aus $\mathcal{G}_i$ berechnet wird. ◇

Bemerkung A.14

Da Variablennamen in einer Klausel immer lokal sind, können verschiedene Klauseln gleich benannte Variablen enthalten, die aber völlig unabhängig voneinander sind. Um daraus entstehende Namenskonflikte zu vermeiden, müssen bei der Abarbeitung Variablen einer Klausel umbenannt werden, falls bereits Variablen mit identischen Namen in anderen Klausel vorkommen. Ein neuer Name darf ebenfalls nicht schon vergeben sein. ◇

A.4 Begriffe der Relationale Algebra

Klauseln deduktiver Datenbanken können in Operationen der relationalen Algebra übersetzt werden [Cremers 94].

Im relationalen Datenmodell werden Daten durch *Relationen* organisiert. Eine Relation ist eine endliche Teilmenge eines kartesischen Produkts. Die Elemente einer Relation heißen *Tupel*. Diese Tupel bestehen aus Attributen.

In der relationalen Algebra werden verschiedene Operationen definiert, mit deren Hilfe aus Relationen neue Relationen berechnet werden können.

Zu den primitiven Operationen der relationalen Algebra gehören, wobei P und Q Relationen sind:

- $P \cup Q$ ist eine Vereinigung,
- $P \times Q$ ist ein kartesisches Produkt,
- $\sigma_{i=a}(P)$ bezeichnet die Menge aller Tupel aus P, die in der i-ten Komponente den Wert a enthalten, $\sigma_{i=j}(P)$ die Menge aller Tupel aus P, die in der i-ten Komponente den gleichen Wert wie in die j-ten Komponente enthalten.
- $\pi_{i_1,\ldots i_k}(P)$ ist die Projektion der Tupel aus P auf neue Tupel, in denen nur die Komponenten enthalten sind, die in den ursprünglichen Tupeln in den Spalten $i_1, \ldots i_k$ enthalten sind,
- $P - Q$ entspricht der mengentheoretischen Differenz.

Eine aus den primitiven Operationen abgeleitete, wichtige Operation ist der Join:

$P \bowtie_{i=j} Q := \sigma_{i=r+j}(P \times Q)$,

wobei i die i-te Spalte von P bezeichnet, j die j-te Spalte von Q und r die Anzahl der Spalten von P ist.

Der Equijoin ist darüber hinaus definiert als:

$P \bowtie_{=} Q := \pi_{\{i_1,\ldots,i_r,j_1,\ldots,j_s\} \setminus \{j \in \{j_1,..,j_s\} \mid \exists k \in \{1,..,r\}: j=i_k\}}(\sigma_{i_{k_1} \hat{=} r+j_{l_1},\ldots,i_{k_n} \hat{=} r+j_{l_n}}(P \times Q))$,

wenn r die Anzahl der Spalten von P ist, s die Anzahl der Spalten von Q, $n \in \mathbb{N}$, $k_1, \ldots, k_n \in \{1, \ldots, r\}$, $l_1, \ldots, l_n \in \{1, \ldots, s\}$. $j \hat{=} i$ bedeute dabei, daß die Spalte i von P das gleiche Attribut besitzt wie die Spalte j von Q.

Eine Auswahl algebraische Gesetze [Vossen 91]:

1. Kommutativgesetz für Joins und Produkte
   $E_1 \bowtie_{cond} E_2 \equiv E_2 \bowtie_{cond} E_1$
   $E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$
   $E_1 \times E_2 \equiv E_2 \times E_1$
   (*cond* ist eine Bedingung auf den Attributen von E_1 und E_2)

2. Assoziativgesetz für Joins und Produkte
   $(E_1 \bowtie_{cond_1} E_2) \bowtie_{cond_2} E_3 \equiv E_1 \bowtie_{cond_1} (E_2 \bowtie_{cond_2} E_3)$
   $(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$
   $(E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3)$
   ($cond_1$ ist eine Bedingung auf den Attributen von E_1 und E_2, $cond_2$ ist eine Bedingung auf den Attributen von E_2 und E_3)

3. Kaskade von Selektionen
$\sigma_{cond_1}(\sigma_{cond_2}(E)) \equiv \sigma_{cond_1 \wedge cond_2}(E)$
$\sigma_{cond_1}(\sigma_{cond_2}(E)) \equiv \sigma_{cond_2}(\sigma_{cond_1}(E))$
($cond_1$ und $cond_2$ sind Bedingungen auf den Attributen von E)

4. Vertauschen von Selektionen und Produkt/Join
$\sigma_{cond_1}(E_1 \bowtie_{cond_2} E_2) \equiv \sigma_{cond_1}(E_1) \bowtie \sigma_{cond_2}(E_2)$
$\sigma_{cond_1}(E_1 \bowtie E_2) \equiv \sigma_{cond_1}(E_1) \bowtie (E_2)$
$\sigma_{cond_1}(E_1 \times E_2) \equiv \sigma_{cond_1}(E_1) \times (E_2)$
($cond_1$ ist eine Bedingung in der nur Attribute von E_1 auftreten, $cond_2$ ist eine Bedingung auf den Attributen von E_1 und E_2)

5. Vertauschung von Selektion und Vereinigung
$\sigma_{cond}(E_1 \cup E_2) \equiv \sigma_{cond}(E_1) \cup \sigma_{cond}(E_2)$
(E_1 und E_2 müssen dieselben Attribute besitzen, *cond* ist eine Bedingung auf den Attributen von E_1 und E_2)

6. Vertauschung von Selektion und Differenz
$\sigma_{cond}(E_1 \backslash E_2) \equiv \sigma_{cond}(E_1) \backslash \sigma_{cond}(E_2)$
(E_1 und E_2 müssen dieselben Attribute besitzen, *cond* ist eine Bedingung auf den Attributen von E_1 und E_2)

7. Vertauschung von Selektion und Join
$\sigma_{cond}(E_1 \bowtie E_2) \equiv \sigma_{cond}(E_1) \bowtie \sigma_{cond}(E_2)$
(*cond* ist eine Bedingung auf den gemeinsamen Attributen von E_1 und E_2)

A.5 Graphen

Es folgen einige Begriffsdefinitionen für Graphen und Bäume.

Definition A.15 *Graph*

Ein Graph $\mathcal{G}$ ist ein Tupel (V, E), wobei

i) V eine Menge von Knoten ist und

ii) $E \subset V \times V$ eine Menge von Kanten zwischen den Knoten. ◇

Definition A.16 *Gerichteter Graph*

Ein gerichteter Graph $\mathcal{G}_d$ ist ein Tupel (V, E_d), mit:

i) V ist eine Menge von Knoten.

ii) $E_d \subset V \times V$ ist eine Menge gerichteter Kanten, wobei $e_{u,v}$ für $e \in E_d$ steht, wobei e die Knoten u und $v \in V$ in der Richtung von u nach v verbindet.

iii) Ein Pfad im Graph ist ein Tupel $(e^1, ..., e^n)$, $n \in \mathbb{N}$ mit $e^1, ..., e^n \in E_d$ und $\exists v_1, ..., v_n, v_{n+1} \in V : \ e^i = e^i_{v_i, v_{i+1}} \ \forall i \in \{1, ..., n\}$.

iv) Ein Pfad ohne Zyklen ist ein Tupel $(e^1, ..., e^n), n \in \mathbb{N}$, wobei
$e^i = e^i_{v_i, v_{i+1}} \in E_d \ \forall i \in \{1, ..., n\} \ \wedge \ v_i \neq v_j \ \forall i \neq j \in \{1, ..., n+1\}$. ◇

Definition A.17 *Baum*

Ein Baum $\mathcal{T}$ ist ein gerichteter Graph (V, E_d).

i) Die Wurzel w des Baumes ist ein Knoten $w \in V$, für den gilt:
$\exists v \in V\ e_{w,v} \in E_d\ \wedge\ \forall v' \in V\ \ \nexists e_{v',w} \in E_d$.

Außerdem gilt: $\forall w' \in V : (\exists v \in V\, e_{w',v} \in E_d \wedge \forall v' \in V\ \ \nexists e_{v',w'} \in E_d \Rightarrow w = w')$.

Die Wurzel ist also ein Knoten, vom dem Kanten ausschließlich wegführen. Definitionsgemäß hat ein Baum genau eine Wurzel.

ii) Ein Zweig eines Baumes ist ein Pfad im Baum, der keine Zyklen enthält.

iii) Ein Blatt v eines Baumes ist ein Knoten $v \in V$, für den gilt:
$\exists u \in V\ e_{u,v} \in E_d\ \wedge\ \forall u' \in V\ \ \nexists e_{v,u'} \in E_d$.

Ein Blatt ist also ein Knoten des Baumes von dem keine Kante mehr ausgeht. ◇

Die folgenden Begriffe werden definiert, um umgangssprachlich bestimmte Sachverhalte eindeutig bezeichnen zu und damit Definitionen und Sätze einfacher formulieren zu können.

Definition A.18 *Unterknoten, Teilgraph, Graphfragment*

Sei $\mathcal{G}_d = (V, E_d)$ ein gerichteter Graph.

- Ein Knoten $w \in V$ ist ein *Nachfolgeknoten* eines Knotens $v \in V$, wenn $e_{v,w} \in E_d$. v ist dann der *Vorgängerknoten* von w.
- Ein *Teilgraph* $\mathcal{G}(v)_d = (V^v, E^v_d)$ mit Wurzel $v \in V$ ist ein gerichteter Graph mit:
  $v \in V^v$ und
  $\forall w \in V$ für die ein Pfad $(v, ..., w) \in \mathcal{G}_d$ existiert, ist $w \in V^v$.
  $\forall w, w' \in V^v$ mit $e_{w,w'} \in E_d$ ist $e_{w,w'} \in E^v_d$.
- Ein *Graphfragment* $\mathcal{G}_f(v)_d = (V^v_f, E^v_{fd})$ mit Wurzel $v \in V$ ist ein gerichteter Graph mit:
  $v \in V^v_f$, $w \in V_f$ wenn ein Pfad $(v, ..., w) \in \mathcal{G}_d$ exisitert und
  ist $w \in V_f$ für den ein Pfad $(v, ..., w', ..., w) \in \mathcal{G}_d$ existiert, so ist $w' \in V^v_f$.
  $\forall w, w' \in V^v_f$ mit $e_{w,w'} \in E_d$ ist $e_{w,w'} \in E^v_{fd}$.
- Ein Knoten $w \in V$ ist *unterhalb* eines Knotens $v \in V$, wenn $w \in \mathcal{G}(v)_d$, d.h. wenn w in dem Teilgraphen enthalten ist, den v aufspannt. Entsprechend ist v *oberhalb* von w. v ist dann ein *Vorgängerknoten* (Vaterknoten) von w, w ein *Nachfolgeknoten* (Unterknoten) von v.
- Ein Vorgängerknoten $w \in V$ eines Teilgraphen $\mathcal{G}(v)_d = (V^v, E^v_d)$ ist ein Knoten $e \notin \mathcal{G}(v)_d$ mit $\exists v' \in \mathcal{G}(v)_d : \exists e_{w,v'} \in E_d$. ◇

Bemerkung: Der Unterschied zwischen einem Graphfragment und einem Teilgraphen besteht folglich darin, daß ein Teilgraph, der von einem Knoten v aufgespannt wird, alle Knoten und Kanten enthält, die auf beliebigen Pfaden, die von v ausgehen, bis zu den Blättern existieren, während für ein Graphfragment lediglich gelten muß, daß es, wenn es außer der Wurzel einen weiteren Knoten enthält, auch alle Knoten und Kanten enthalten muß, die auf einem Pfad zwischen der Wurzel und diesem Knoten liegen.

B Das Steuerungssystem VaMoRs

Das Fahrzeugführungssystem VaMoRs soll im folgenden ausführlicher vorgestellt werden. Das Modul *Behaviour Decision* ist für die Komponente des Fahrverhaltens des Fahrzeuges zusändig, die künstliche Intelligenz benötigt. Dieses wird daher detailliert vorgestellt.

Abbildung 71: Das autonome Straßenfahrzeug VaMoRs

In den Jahren 1988 - 1992 hielt das Fahrzeug den Geschwindigkeitsrekord bei autonom-mobilen Systemen mit 90km/h auf Autobahnen.

B.1 Funktionsweise von VaMoRs

Die grundlegenden Aufgaben einer autonomen Fahrzeugsteuerung auf befestigten Straßen sind [Dickmanns 91]:

- Bestimmung des Straßenverlaufs.
- Bestimmung der Position des eigenen Fahrzeugs relativ zur Straße.
- Auffinden aller möglichen Hindernisse in Bewegungsrichtung.
- Interpretation der Straßenmarkierungen.
- Vorausschauende Bestimmung des Verhaltens anderer Fahrzeuge.

Notwendige Funktionen des Fahrzeugs sind dabei das Fahren innerhalb der Fahrspur, die Möglichkeit zum Wechsel der Fahrspur und das richtige Reagieren auf andere Verkehrsteilnehmer in Form von Kolonnenfahren und Überholen [Brüdigam 93].

Das erste Testfahrzeug mußte deutlich größer sein, als es für die Unterbringung der eigentlichen Zielfunktion nötig wäre, um bei Testfahrten eine umfangreiche Kontrolle,

sowie Programmweiterentwicklung und Fehlersuche zu ermöglichen. Es benötigt daher Raum für einen Arbeitsplatz mit Monitoren, Entwicklungsrechnern und Interfaces. Aus diesem Grund wurde für den ersten Prototypen ein 5-Tonnen Kastenwagen vom Typ Daimler-Benz L 508 D verwendet. Für die Entwicklung des autonomen Fahrens wurde das mit Automatikgetriebe ausgestattete Fahrzeug entsprechend umgerüstet. Zusätzlich zum Kamerasystem enthält es verschiedene im Fahrzeug angebrachte Sensoren, die notwendige Informationen wie Fahrgeschwindigkeit und Lenkwinkel liefern. Gas und Lenkung werden über Schrittmotoren betätigt, die Bremse über ein Hydraulikaggregat [Brüdigam 94].

Das System wird durch Drücken eines Startknopfs durch den Sicherheitsfahrer, der sich auf dem Fahrersitz befindet, gestartet. Durch Betätigen eines Notausschalters oder der Fußbremse kann der Fahrer jederzeit den Rechner abschalten und die Steuerung wieder selbst übernehmen.

Das bifokale Kamerasystem zur Wahrnehmung besteht aus zwei miteinander gekoppelten Kameras mit unterschiedlicher Brennweite. Die Weitwinkelkamera gestattet einerseits einen großen Blickwinkel zur Erfassung eines umfangreichen Teils des Nahbereichs, andererseits erlaubt die Telekamera eine gute Ortsauflösung weiter weg, um Hindernisse frühzeitig erkennen zu können. Der Einsatz einer stabilisierenden Plattform erlaubt schnelle Blickrichtungsänderungen und das Fixieren einzelner Objekte. Die Bilddaten werden mit Hilfe eines Kalmann-Filters zur Objekterkennung im Bild vorverarbeitet und anhand räumlicher und zeitlicher Modellvorstellungen über Aussehen und Verlauf der Straße, sowie über Aussehen und Bewegung möglicher Hindernisse analysiert.

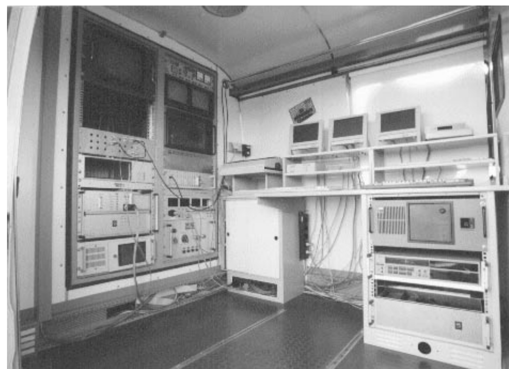


Abbildung 72: Innenansicht von VaMoRs

Mittlerweile ist im Versuchsfahrzeug nicht nur ein Kamerasystem für die Vorrausschau installiert, sondern auch ein Kamerasystem für den Blick nach hinten, um beispielsweise auf einer Autobahn erkennen zu können, ob der rückwärtige Teil der Überholspur frei ist, oder ob dort ein schnelleres Fahrzeug aufholt. Eine zusätzliche Erweiterung wäre außerdem noch eine seitliche Kamera rechts, um ein sicheres Einscheren nach einem Überholvorgang zu gewährleisten [Dickmanns 95].

Der erwartungsbasierte 4D-Ansatz zum mitlaufenden Verstehen dynamischer Prozesse hat sich als besonders effizient und leistungsfähig herausgestellt: Parallel zur realen Welt wird im Rechner eine Vorstellungswelt aufgebaut, die wie die reale mit drei Raum- und einer Zeitkoordinate repräsentiert wird. Für jedes Objekt, das im Aufgabenzusammenhang wesentlich ist, wird hier dessen räumliche Form und dessen räumliche Lage mit

den Geschwindigkeitskomponenten sowie das Wissen über die Gesetzmäßigkeiten seines allgemeinen Bewegungsverhaltens dargestellt. Mit entsprechenden Annahmen über Störgrößen und Steuerungseingaben kann das Bewegungsverhalten für den nächsten Meßzeitpunkt vorhergesagt werden. Durch Vergleichen der Meßvorhersagen mit dem Ergebnis der nächsten Meßung kann ein gewisser Lerneffekt erzielt werden. Dies erleichtert die Erkennung und Verfolgung von Objekten wie Straße und andere Verkehrsteilnehmer mit der Kamera um ein Vielfaches [Dickmanns 97].

Die Straße, auf der gefahren wird, ist das wichtigste Objekt, das erkannt werden muß. Mittlerweile können hügelige Straßen mit der Zahl ihrer Spuren und den Krümmungsparametern erkannt werden. Andere Fahrzeuge sind als Mitbenutzer der gleichen Verkehrsfläche, ebenfalls wichtige Objekte. Ihre Erfassung geschieht in mehreren Phasen. Zur Entdeckung wird im Bereich der Fahrbahn in der eigenen Spur in weiter Vorausschau und in der Nebenspur im Nahbereich gesucht.

Abbildung 73: Blick aus dem Fenster

In nicht allzuferner Zukunft muß außerdem die Erkennung von Verkehrszeichen ermöglicht werden, um beispielsweise auf eine Geschwindigkeitsbeschränkung oder ein Überholverbot korrekt reagieren zu können.

Die Fähigkeiten zum Spurfahren auf Straßen oder zum Konvoi-Fahren erfordern keine Elemente künstlicher Intelligenz oder aufwendige Planungsverfahren, sondern können durch Regelalgorithmen, im Sinne von Regeltechnik, realisiert werden, die auf Abweichungen von Sollwerten mit den richtigen Steuerungsausgaben reagieren. Anders sieht es dagegen bei erweiterten Funktionen, wie beispielsweise einem Überholmanöver, aus, denen ein umfangreiches Regelsystem zur Entscheidungsfindung zugrunde liegt [Dickmanns 92].

B.2 Systemstruktur von VaMoRs

Das verwendete Transputersystem erlaubt einen modularen Aufbau der Systemarchitektur [Mündemann 93]:

`SENS`: Sensoren für den Fahrzeugzustand
`FZG`: Fahrzeugsteuerung (Gas, Kupplung, Bremse)
`BVV`: Bildvorverarbeitung

`PL`: Steuerung für die Kameraplattform
`Tele`: Digitalisierung der Bilddaten der Telekamera
`WW`: Digitalisierung der Bilddaten der Weitwinkelkamera
`KTele`: Telekamera
`KWW`: Weitwinkelkamera

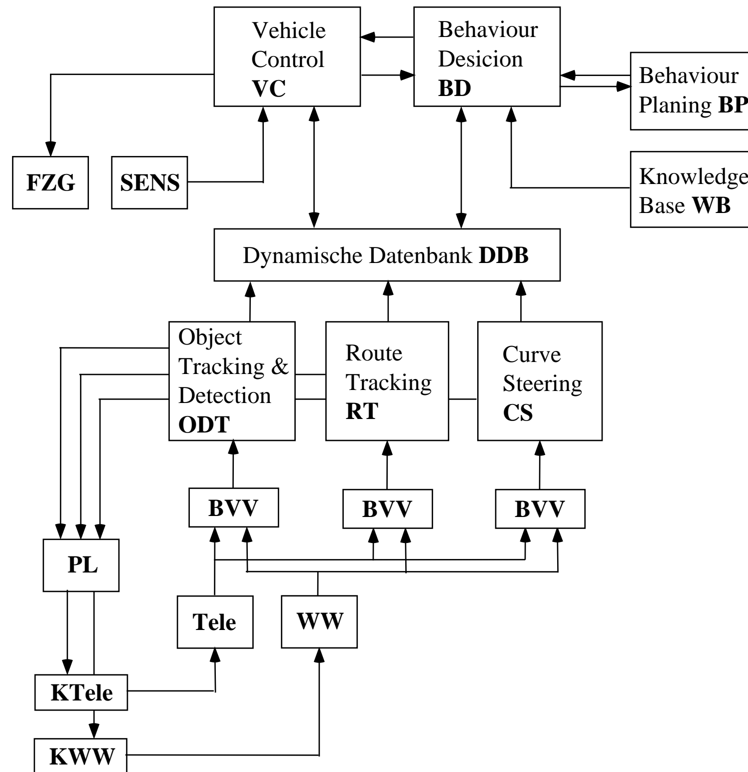


Abbildung 74: Struktur des Steuerungssystems

Die Bildvorverarbeitung übernimmt das Modul Kronos [Dickmanns 93b]. Für die Objekterkennung sind die Module *Route Tracking*, Rt, [Behringer 93], *Object Detection and Tracking*, Odt, [Thomanek 93] und *Curve Steering*, Ms, [Müller 95] zuständig, die sich direkt an die Bildvorverarbeitung angliedern. Ihre Ergebnisse gehen an eine globale, dynamische Datenbank, die *Dynamic Data Base*, Ddb, [Dickmanns 93a] auf die alle Module Zugriff haben, die für eine Verhaltensplanung zuständig sind.

Im Detail befaßt sich *Route Tracking* mit der Erkennung und Verfolgung der Straße, einschließlich all ihrer Parameter, wie Breite, Anzahl der Spuren und horizontale wie vertikale Krümmung. *Object Detection and Tracking* entdeckt andere Verkehrsteilnehmer und ermittelt ihre Position, sowie Bewegungsrichtung und Geschwindigkeit. *Curve Steering* ist noch im Aufbau und soll das Abbiegen an Kreuzungen realisieren, bei dem eine einmündende Straße korrekt erkannt werden und im Verlauf die gesamte Erkennung und Analyse auf diese Straße umgestellt werden muß.

Dem Modul *Vehicle Control*, Vc, [Brüdigam 93] obliegt die Längs- und Querführung des Fahrzeugs und damit die Berechnung der einzelnen Steuergrößen. Verhaltensentscheidungen werden nur für grundlegende Fahrmanöver wie Spurfahren, Konvoi-Fahren,

Spurwechsel und Gefahrenstop getroffen. Sie lassen sich aufgrund bekannter Umgebungsparameter berechnen und erfordern kein Regelsystem.

Das Modul *Behaviour Decision*, BD, [Kujawski 93] ist für die wissensbasierten und damit "intelligenzgesteuerten" Reaktionen des Fahrzeugs zuständig, im Gegensatz zu den reflexartigen Reaktionen, die das Modul *Vehicle Control* übernimmt. *Behaviour Decision* obliegt die Entscheidung über das längerfristige Verhalten, insbesondere darüber, ob ein Spurwechsel durchgeführt werden soll oder nicht. Auch hier können grundlegende Verhaltensmodi berechnet werden, weitergehende Entscheidungen, besonders in Alternativfällen, werden aber durch ein Regelsystem getroffen.

B.3 Das Modul Behaviour Decision

Das Modul Behaviour Desicion ist für ein "intelligentes" Verhalten des Fahrzeugs verantwortlich. Ihm obliegt die Entscheidung über die Aktionen des Fahrzeugs in Beziehung zu anderen Verkehrsteilnehmern, also insbesondere darüber, ob ein Spurwechsel zum Zwecke eines Überholmanövers durchgeführt werden soll oder nicht [Kujawski 93]. Die Verhaltensentscheidung erfolgt aufgrund der vom Benutzer vorgegebenen Planungsgeschwindigkeit, sowie der aktuellen Verkehrssituation in der Umgebung des Fahrzeugs.

Die Kommunikation mit den Modulen ODT und RT erfolgt über die *Dynamic Data Base* (DDB). Hier werden Daten über die Straße und über andere Verkehrsteilnehmer ausgetauscht. Der Datenaustausch zum VC findet auf direktem Weg mit dem *Message Passing System* (MPS) [Holt 93] statt, um Verzögerungen bei der Datenübertragung zu vermeiden. Von VC erhält BD Daten über den Zustand des Fahrzeugs wie Geschwindigkeit und Lage in der Fahrbahnspur. BD sendet an VC Kommandos zurück, die das weitere Verhalten des Fahrzeugs bestimmen und von VC realisiert werden wie: "Führe einen Spurwechsel durch". Zusätzlich stellt ein Dialog-Manager, *Dialogue Manager* (DM), [Mündemann 92] die Schnittstelle zwischen dem Sicherheitsfahrer und BD her, da ein Überholmanöver derzeit aus Sicherheitsgründen noch explizit freigegeben werden muß.

In der derzeitigen Realisierung sind folgende Fahrverhalten möglich:

- Fahren in der Fahrspur,
- Spurwechsel auf die linke/rechte Fahrspur,
- Abbruch des Spurwechsels auf die linke/rechte Fahrspur,
- Pendeln in der Fahrspur, um dem nachfolgenden Verkehr den Wunsch, die Fahrspur zu wechseln, zu signalisieren
- Abbremsen in der Fahrspur auf eine gegebene Geschwindigkeit, um ein Einscheren in die rechte Fahrspur zu ermöglichen.

Bisher wird nur ein Spurwechsel über maximal eine Fahrspur betrachtet.

B.3.1 Schnittstellen zu den anderen Modulen

In den folgenden Tabellen sind diejenigen Variablen aufgelistet, die von anderen Modulen mit Werten belegt werden und von BD verwendet werden, oder die dazu dienen von BD Daten an andere Module zu schicken.

Die Daten, die vom Regelsystem direkt verwendet werden, sind fett geschrieben. Die anderen Daten werden zur Berechnung der Trajektorien und der Geschwindigkeiten der anderen Verkehrsteilnehmer benötigt und tauchen im Regelsystem nicht auf.

Von der DDB erhält BD folgende Eingangsdaten über das eigene Fahrzeug:

AmvLength	Länge des Fahrzeugs *in m*
AmvLength	Breite des Fahrzeugs *in m*
AmvHeigth	Höhe des Fahrzeugs *in m*
AmvMass	Masse des Fahrzeugs *in kg*
AmvAmax	Maximale Beschleunigung des Fahrzeugs *in* m/s^2

Von VC erhält BD folgende Eingangsdaten über das eigene Fahrzeug:

VCState	Zustand des Moduls VC
vms	Betrag der Geschwindigkeit des Fahrzeugs *in* m/s
AmvVX	Geschwindigkeit entlang der Fahrbahn *in* m/s
AmvVY	Geschwindigkeit quer zur Fahrbahn *in* m/s
AmvScalarAcc	Betrag der Beschleunigung des Fahrzeugs *in* m/s^2
AmvAccX	Beschleunigung entlang der Fahrbahn *in* m/s^2
AmvAccY	Beschleunigung quer zur Fahrbahn *in* m/s^2
AmvAccZ	Beschleunigung senkrecht auf der Fahrbahn *in in* m/s^2
emergency_stop	Flag, daß VC gerade einen Nothalt durchführt
convoy	Flag, daß VC dem vorausfahrenden Fahrzeug nachfährt
trafficator_ack	Flag, das den Status der Blinker anzeigt
chtll_ack	Flag, daß VC eine Aufforderung zum Spurwechsel auf die linke Fahrspur erhalten hat
chtrl_ack	Flag, daß VC eine Aufforderung zum Spurwechsel auf die rechte Fahrspur erhalten hat
lch_active	Flag, daß VC den Spurwechsel durchführt

Von RT erhält BD folgende Eingangsdaten über die Straße:

RTState	Zustand des Moduls RT
AmvOffset	Ablage des Fahrzeugs von der Fahrspurmitte der Fahrspur, auf der sich das Fahrzeug befindet *in m*
HorC0	horizontale Krümmung der Fahrbahn *in* m^{-1}
HorC1	horizontale Krümmungsänderung der Fahrbahn *in* m^{-2}
VertC0	vertikale Krümmung der Fahrbahn *in* m^{-1}
VertC1	vertikale Krümmungsänderung der Fahrbahn *in* m^{-2}
LeftLaneExists	Flag, das die Existenz einer linken Fahrspur anzeigt
LeftLaneWidth	Breite der linken Fahrspur *in m*
RightLaneExists	Flag, das die Existenz einer rechten Fahrspur anzeigt

RightLaneWidth	Breite der rechten Fahrspur *in m*
CwReadyLeft	Flag, das während eines Spurwechsels auf die linke Fahrspur gesetzt wird
CwReadyRight	Flag, das während eines Spurwechsels auf die rechte Fahrspur gesetzt wird

Von ODT erhält BD für jedes endeckte Fahrzeug folgende Eingangsdaten:

ODTState	Zustand des Moduls ODT
TPId	Identifikationsnummer
TpOffset	Ablage des entdeckten Fahrzeugs von der Mitte seiner Fahrspur *in m*
TpDistance	Entfernung zum entdeckten Fahrzeug *in m*
TpLength	Länge des entdeckten Fahrzeugs *in m*
TpWidth	Breite des entdeckten Fahrzeugs *in m*
TpHeight	Höhe des entdeckten Fahrzeugs *in m*
TpScalarSpeed	Betrag der Geschwindigkeit des entdeckten Fahrzeugs *in m/s*
TpVx	Geschwindigkeit entlang der Fahrspur *in m/s*
TpVy	Geschwindigkeit quer zur Fahrspur *in m/s*
TpScalarAcc	Betrag der Beschleunigung des entdeckten Fahrzeugs *in* m/s^2
TpScalarAccX	Beschleunigung entlang der Fahrspur *in* m/s^2
TpScalarAccY	Beschleunigung quer zur Fahrspur *in* m/s^2

Vom Benutzer über DM eingegeben erhält BD folgende Eingangsdaten:

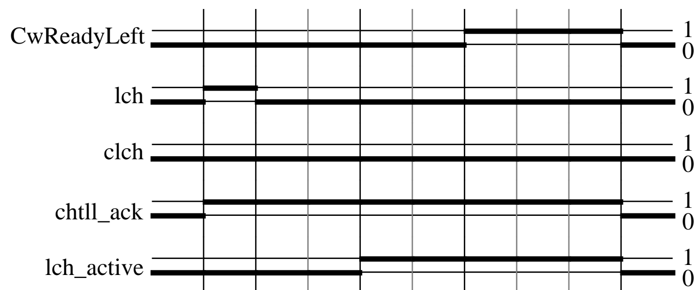
DMState	Zustand des Moduls DM
Vplanned	Planungsgeschwindigkeit *in m/s*
PassingAllowed	Flag für die Erlaubnis zum Überholen
dmccll	Größe, über die der Benutzer einen autonomen Spurwechsel auf die linke Fahrspur abbricht
dmccrl	Größe, über die der Benutzer einen autonomen Spurwechsel auf die rechte Fahrspur abbricht
dmcllp	Größe, über die der Benutzer einen autonomen Spurwechsel auf die linke Fahrspur freigibt
dmccllp	Größe, über die der Benutzer einen autonomen Abbruch des Spurwechsel auf die linke Fahrspur freigibt
dmcrlp	Größe, über die der Benutzer einen autonomen Spurwechsel auf die rechte Fahrspur freigibt
dmccrlp	Größe, über die der Benutzer einen autonomen Abbruch des Spurwechsel auf die rechte Fahrspur freigibt

BD sendet an VC folgender Eingangsdaten:

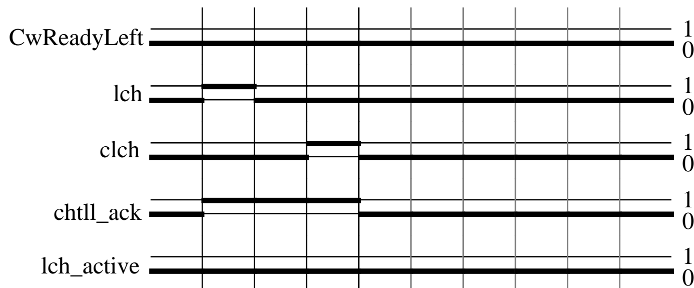
BDStateFlag	das anzeigt, daß BD aktiv ist
emergency_stop	Flag über das BD einen Nothalt kommandieren kann
trafficators	Größe über die die Fahrtrichtungsanzeiger angesteuert werden
v	Geschwindigkeit, die VC einhalten soll *in m/s*
lch	Größe, über die BD einen Spurwechsel auf die linke/rechte Fahrspur kommandiert (links: -1, rechts: 1, sonst: 0)
clch	Flag, das einen Abbruch des Spurwechsels kommandiert
oscillate	über dieses Flag kommandiert BD ein Pendeln in der Spur

Für die Durchführung eines Spurwechsels und eines Spurwechselabbruchs ist zwischen BD und VC ein detailliertes Protokoll vereinbart.

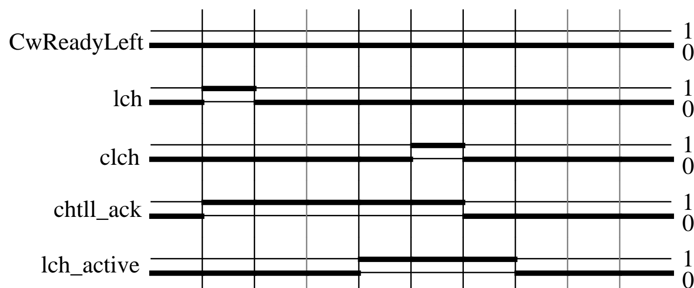
1. Spurwechsel nach links ohne Abbruch:



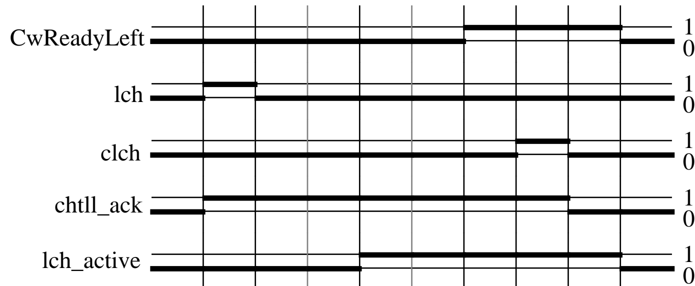
2.a Spurwechsel nach links mit Abbruch eines noch nicht ausgeführten Spurwechsels:



2.b Spurwechsel nach links mit Abbruch eines begonnenen Spurwechsels vor Erreichen der neuen Spur:



2.c Spurwechsel nach links mit Abbruch eines begonnenen Spurwechsels nach Erreichen der neuen Spur:



Die Angaben des Protokolls sind:

CwReadyLeft:	Flag, das während eines Spurwechsels auf die linke Fahrspur gesetzt wird
ch:	Größe, über die BD einen Spurwechsel auf die linke/rechte Fahrspur kommandiert
clch:	Flag, das einen Abbruch des Spurwechsels kommandiert
chtll_ack:	Flag, das anzeigt, daß VC eine Aufforderung zum Spurwechsel auf die linke Fahrspur erhalten hat
lch_active:	Flag das anzeigt, daß VC den Spurwechsel durchführt

B.3.2 Ablauf des Programms

Da das autonome Fahrzeug in seiner Fahrsicherheit mindestens mit den Leistungen eines menschlichen Fahrers vergleichbar sein soll, darf die Reaktion auf äußere Ereignisse nicht schlechter als die eines manuell gesteuerten Fahrzeugs sein. Das heißt z.B., daß ein eingeleiteter Spurwechsel bei einer veränderten Verkehrsituation jederzeit sofort abgebrochen können werden muß.

Im allgemeinen wird eine Reaktionszeit für einen menschlichen Fahrer von 0.6 s ein Ereignis, das er im Blickfeld hat, veranschlagt, 0.8 s für ein Ereignis, dem er erst seinen Blick zuwenden muß. Somit werden hier für weitere Überlegungen 0.6 s als Vergleich in Betracht gezogen [Kujawski 93].

Rechnet man eine mittlere, realistische Verzögerungszeit von 300 ms für das Einlesen des Videobildes, das Auswerten durch die Bildverarbeitung KRONOS, das Erkennen von Straße und anderen Verkehrsteilnehmern durch RT und ODT, die Reaktion des Moduls VC auf Kommandos von BD und für die Übermittlung der Daten über die DDB und dem MPS, zum VC und zurück, so verbleiben für das Modul BD maximal 300 ms Reaktionszeit.

Das Modul gliedert sich in zwei wesentliche Teile: Der erste Teil behandelt den Spurwechsel und sendet, entsprechend dem aktuellen Zustand des Fahrzeugs und dem gewünschten Folgezustand, die Kommandos an VC. Die Aufgabe des zweiten Teils ist es, zu berechnen, ob ein Spurwechsel gefahrlos möglich ist, oder ob er vielleicht abgebrochen werden muß. Dafür stehen wahlweise zwei Funktionen zur Verfügung, eine zur genaueren, aber langsamen Berechnung mit Hilfe eines numerischen Extrapolationsverfahrens, die andere zur weniger genauen, aber schnellen Berechnung.

Der Ablauf des Programms BD ist im folgenden stark vereinfacht skizziert:

```
do
   initialisiere Variablen
   warte auf die Statusmeldungen der DDB und der Module VC und DM
   if Statusmeldungen sind ok then
      repeat
         repeat
         until eine Meldung von DDB empfangen wurde and
               eine Meldung von VC empfangen and
               eine Meldung von DM empfangen wurde
         berechne die Geschwindigkeiten der anderen Verkehrsteilnehmer
         berechne die die Größen zur Bewertung der Trajektorien
         setze alle Steuerflags und -größen für VC
         berechne den Nachfolgezustand
         sende eine Meldung an VC
      until Abbruchbedingung von DM empfangen
   end
```

Ein Modell der Fahrbahn, das sich aus der eigenen Fahrspur und der Fahrspuren, die rechts und links der eigenen liegen, falls sie vorhanden sind, zusammensetzt, wird aus den Daten errechnet, die BD vom Modul RT erhält. Mit der vom Modul ODT gemeldeten vektoriellen Geschwindigkeiten anderer Verkehrsteilnehmer, in Bezug auf ein xyz-Koordinatensystem, werden die Trajektorien, d.h. die Fahrtverläufe des eigenen Fahrzeugs und der anderen Verkehrsteilnehmer extrapoliert. Dabei werden auch Krümmungsänderungen der Spur in die Berechnung miteinbezogen, die durch oder bei einen Spurwechsel entstehen.

Ein Spurwechsel, den ein anderer Verkehrsteilnehmer durchführen will, wird dadurch erkannt, daß seine Geschwindigkeit senkrecht zu seiner Fahrspur einen bestimmten Schwellwert überschreitet. Weiterhin wird davon ausgegangen, daß der Verkehrsteilnehmer seinen Spurwechsel dann beendet, wenn er die Mitte der neuen Fahrbahn erreicht hat.

Die numerische Extrapolation, die drei Trajektorien für das eigene Fahrzeug errechnet, eine für das Beibehalten der Spur und jeweils eine für einen Spurwechsel nach links oder nach rechts, und jeweils eine Trajektorie für jeden anderen Verkehrsteilnehmer bestimmt, arbeitet mit einem Zeitraster von 50 ms und einem Extrapolationshorizont von 7.5 s.

Um anzugeben, welche der drei möglichen Trajektorien des Fahrzeugs für einen bestimmten Zeitpunkt sicher ist, wird der seitliche Abstand des Fahrzeugs zu jedem anderen Verkehrsteilnehmer zu diesem Zeitpunkt entsprechend der betrachteten Trajektorie berechnet. Ist dieser zu klein, wird zusätzlich der tatsächliche Abstand entlang der Fahrbahn berechnet, sowie ein erforderlicher Mindestabstand. Diese Werten für jeden Verkehrsteilnehmer und für eine Reihe von Zeitpunkten zusammengenommen bilden ein Maß für die Sicherheit der Trajektorie.

Das einfachere Verfahren berücksichtigt dabei nicht die Krümmung der Fahrbahn. Außerdem wird der Verlauf eines Spurwechsels nicht differenziert verfolgt, sondern es wird vereinfacht davon ausgegangen, daß die neue Spur zusätzlich zur alten sofort auf ihrer ganzen Breite belegt ist. Das Verfahren liefert damit nicht die gewünschte Genauigkeit, liegt aber in der Berechnungszeit zwei Größenordnungen unter dem numerischen Extrapolationsverfahren und wird derzeit verwendet um unter der für das Modul bestimmten Berechnungszeit von 300 ms zu bleiben.

Der Hauptteil des Modul Bd enthält die Regeln, die das Verhalten des Fahrzeugs, abhängig von der im Extrapolationsteil berechneten Verkehrssituation, bestimmen. Sie werden im folgenden entsprechend ihrer Schreibweise in der Programmiersprache C angegeben.

Die Regel von 1 bis 7 berechnen interne Zustandsvariablen des Moduls Bd:

Die Bewertung der Sicherheit der drei Trajektorien für das Fahrzeug sind in den Variablen *bl* (linke Trajektorie), *br (rechte)* und *bo* (gleiche Spur) abgelegt. Die Werte *vov*, d.h. die Geschwindigkeit des vorausfahrenden Fahrzeugs, und *linksv*, die Geschwindigkeit des Fahrzeugs auf der linken Spur werden ebenfalls von Bd berechnet. Existiert kein links fahrendes Fahrzeug, ist *linksv* unendlich.

1. `cim = (vms >= VMIN)`
   car is moving: Das Fahrzeug ist in Bewegung (`cim`), wenn die aktuelle Geschwindigkeit (`vms`) größer als ein Schwellwert von 1.0 m/s ist, der sich bereits in der Praxis bewährt hat.

2. `llcr = (Vplanned > vms + VEPSILON) &&`
   `convoy && (linksv > vov + VEPSILON)`
   left lane change request: Ein Spurwechsel auf die linke Fahrbahn soll eingeleitet werden, wenn die Planungsgeschwindigkeit (`Vplanned`) einen bestimmten Betrag (`vepsilon`) größer als die aktuelle Geschwindigkeit des Fahrzeugs (`vms`) ist, das Fahrzeug einem anderen nachfährt (`convoy`), und die Geschwindigkeit eines auf der linken Fahrspur fahrenden Fahrzeugs (`linksv`) ebenfalls um den Betrag (`vepsilon`) größer als die Geschwindigkeit des vorausfahrenden Fahrzeuges (`vov`) ist.

3. `rlcr = RightLaneExists`
   right lane change request: Ein Spurwechsel auf die rechte Fahrbahn ist immer wünschenswert, wenn diese existiert.

4. `cllp = cim && PassingAllowed && LeftLaneExists && (bl > 1.0) &&`
   `(dmcllp == FREIGABE)`
   change left lane is possible: Ein Spurwechsel auf die linke Fahrspur ist möglich, wenn das Fahrzeug fährt (`cim`), kein Überholverbot besteht (`PassingAllowed`), die linke Fahrbahn existiert (`LeftLaneExists`) und als sicher gilt (`bl`) und der Benutzer das Überholen freigibt (`dmcllp`).

5. `ccll = chtll_ack &&`
   `(((!CwReadyLeft && (bl < ABBRUCH_WERT) && (bo >= 1.0)) //`
   `((CwReadyLeft && (bo < ABBRUCH_WERT) && (br >= 1.0))) &&`
   `(dmccllp == FREIGABE)) // (dmccll == FREIGABE))`
   cancel change left lane: Ein Spurwechsel auf die linke Fahrspur soll abgebrochen werden, wenn er vom Modul Vc bestätigt ist (`chtll_ack`), die Trajektorie für ein weiteres Druchführen des Spurwechsels (`bl/bo`) unsicher ist und die Trajektorie für den Spurabbruch (`bo/br`) sicher ist. Hierbei muß unterschieden werden, ob der Spurwechsel schon soweit durchgeführt ist, daß Rt seine Suchfenster auf die linke Fahrspur umgesetzt (`CwReadyLeft`) hat. Ein Abbruch des Spurwechsels muß über DM freigegeben werden (`dmccllp`), kann aber auch von ihm ausgelöst werden (`dmccll`).

6. `crlp = cim && RightLaneExists && (br >= 1.0) &&`
   `(dmcrlp == FREIGABE)`
   change right lane is possible: Wie bei Regel 4, nur für rechts statt links.

7. `ccrl = chtrl_ack &&`
   `(((!CwReadyRight && (br < ABBRUCH_WERT) && (bo >= 1.0)) //`

`((CwReadyRight && (bo < ABBRUCH_WERT) && (bl >= 1.0))) &&`
`(dmccrlp == FREIGABE)) // (dmccrl == FREIGABE))`
cancel change right lane: Wie bei Regel 5, nur für rechts statt links.

Alle weiteren Regeln geben die Übergangsbedingungen in einem Zustandsübergangsdiagramm an. Figur 75 zeigt das Zustandsübergangsdiagramm, daß durch die Regeln 8 bis 34 beschrieben wird. Da sich der Wechsel auf die linke und der Wechsel auf die rechte Spur sehr ähnlich sind, sind der linke und der rechte Ast im Zustandsübergangsdiagramm nahezu identisch.

Die folgenden Regeln 8 und 8.6 beschreiben das Fahren in der Spur einschließlich dem Verfolgen eines vorausfahrenden Fahrzeugs in derselben Spur und behandeln einen möglichen Überholvorgang.

8. `llcr && LeftLaneExists && PassingAllowed`
   zeigt dem Fahrzeug grundsätzlich an, daß ein Überholen möglich ist: `S1` $\longrightarrow$ `S2`.

9. `llcr && LeftLaneExists && PassingAllowed && (time > T1)`
   nach einer Mindestwartezeit (`T1`) von 5.0s kann der Überholvorgang beginnen: `S2` $\longrightarrow$ `S3`.

Die Regeln 10 bis 20 beschreiben den Ablauf eines Überholvorgangs mit möglichem Abbruch.

10. `!llcr // !LeftLaneExists // !PassingAllowed`
    hebt die Überholmöglichkeit von Regel 8 wieder auf und bricht damit jeden Überholvorgang ab: `S2,S3` $\longrightarrow$ `S1`.

11. `cllp && llcr && (time <= T2)`
    setzt den Überholvorgang fort: `S3,S7` $\longrightarrow$ `S4`; `S8` $\longrightarrow$ `S5`. `S4` und `S5` geben an das Modul Vc die Anweisung den Blinker zu setzen, `S5` setzt zusätzlich die Größen für das Pendeln zurück.

12. `cllp && llcr && (time > T2)`
    die Zeit für den Blinkvorgang (`T2`) ist verstrichen: `S4` $\longrightarrow$ `S5`.

13. `!cllp // !llcr`
    bricht den Überholvorgang während des Blinkens ab: `S4` $\longrightarrow$ `S6`. `S6` setzt den Blinker zurück.

14. `chtll_ack`
    ist Bestätigung von Vc zum Ausführen eines Spurwechsels auf die linke Fahrspur: `S5` $\longrightarrow$ `S7`. `S7` führt den Spurwechsel durch.

15. `!lch_active && !chtll_ack`
    wartet auf die Beendigung des Spurwechsels von Vc: `S7` $\longrightarrow$ `S6`.

16. `llcr && (time > T2)`
    überschreitet die Wartezeit für einen Spurwechsel eine gewisse Höchstzeit, reduziert `S8` den Wert für den Sicherheitsabstand und ein erneuter Überholvorgang wird versucht: `S3` $\longrightarrow$ `S8`. Wird die Wartezeit ein zweites Mal überschritten, veranlaßt `S8.6` ein Pendeln des Fahrzeugs in der Spur mit gesetztem Blinker: `S8` $\longrightarrow$ `S8.6`.

17. `ccll`
    bricht einen Spurwechsel während seiner Ausführung ab, falls es erforderlich ist: `S7` $\longrightarrow$ `S10`. In `S10` wird der rechte statt dem linken Blinker gesetzt.

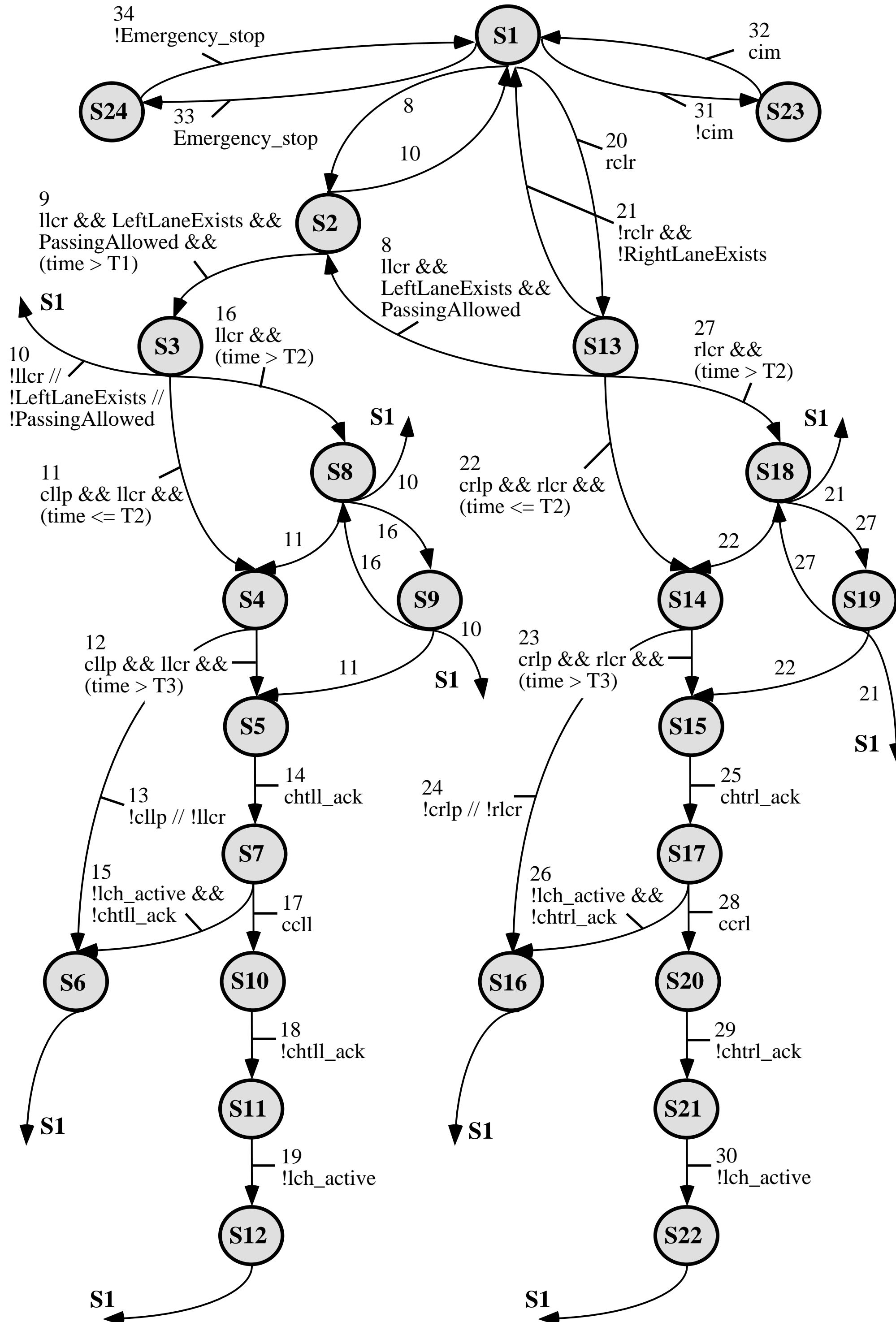


Abbildung 75: Das Zustandsübergangsdiagramm

18. `!chtll_ack`
    bestätigt das Ende des Abbruchs des Spurwechsels durch Vc: `S10` ⟶ `S11`. `S11` setzt die Abbruchforderung zurück.

19. `!lch_active`
    beendet den Abbruch mit Zurücksetzen des rechten Blinkers: `S11` ⟶ `S12`.

Die Regeln 20 bis 30 beschreiben den Ablauf eines Überholvorgangs mit möglichem Abbruch.

20. `rclr`
    beschreibt den Wunsch eines Spurwechsels in die rechts liegende Fahrbahn, falls eine solche existiert: `S1` ⟶ `S13`.

21. `!rclr`
    hebt den Wunsch eines Spurwechsels nach rechts wieder auf: `S13` ⟶ `S1`.

22. `crlp && rlcr && (time <= T2)`
    beschreibt den Fall, daß keine linke Spur existiert, oder der Wunsch nach links zu wechseln ist nicht vorhanden, aber es existiert eine rechte Fahrspur. Das bedeutet, daß ein Fahrspurwechsel nach rechts unbedingt eingeleitet werden muß: `S3,S7` ⟶ `S4`; `S8` ⟶ `S5`. (Siehe Regel 11).

23. `crlp && rlcr && (time > T3)`
    `S4` ⟶ `S5` (Siehe Regel 12)

24. `!crlp // !rlcr`
    `S4` ⟶ `S6` (Siehe Regel 13)

25. `chtrl_ack`
    `S5` ⟶ `S7` (Siehe Regel 14)

26. `!lch_active && !chtrl_ack`
    `S3` ⟶ `S8`; `S8` ⟶ `S8.6` (Siehe Regel 15)

27. `rlcr && (time > T2)`
    `S7` ⟶ `S10` (Siehe Regel 16)

28. `ccrl`
    `S10` ⟶ `S11` (Siehe Regel 18)

29. `!chtrl_ack`
    `S10` ⟶ `S11` (Siehe Regel 9)

30. `!lch_active`
    `S11` ⟶ `S12` (Siehe Regel 20)

Die Regeln 31 bis 34 beschreiben Halt- und Nothalt-Situationen.

31. `!cim`
    hält das Fahrzeug, weil der Verkehr stockt, führt von jedem der Zustände zusätzlich ein Pfeil zu `S23`, außer vom Zustand `S24`. Alle Einstellungen werden zurückgesetzt. `S1` ⟶ `S23`

32. `cim`
    überführt das Fahrzeug in den Zustand `S1`: `S23` ⟶ `S1`

33. `Emergency_stop`
    von Vc löst einen Nothalt aus: `S1` ⟶ `S24`. In `S24` wird die Warnblinkanlage eingeschaltet. Von jedem der Zustände führt zusätzlich ein Pfeil zu `S24`, außer vom Zustand `S23`. Alle Einstellungen werden zurückgesetzt.

34. `!Emergency_stop`
    überführt das Fahrzeug in den Zustand `S1`: `S24` ⟶ `S1`. Die Warnblinkanlage wird ausgeschaltet.

B.4 C-Sourcecode des Moduls Behaviour Decision

Es folgt das originale C-Programm des Moduls Behaviour Decision, der das Regelsystem für die Zustandsübergänge des Automaten enthält.

```
const ACTIVE = 1         PASSIVE = 0        STOP = -1
      LEFT = -1          FRONT = 0          RIGHT = 1
      BOTH = 2           NONE  = 0
      FREIGABE = 1       GESPERRT = 0
      VMIN = 1.0         VEPSILON   = 10.0
      TR_SICHER = 1.0    TR_ABBRUCH = 0.75
      T1 = 2.0           T2 = 10.0          T3 = 5.0

typedef char bool

        struct {
            /* Daten aus der DDB von RT und ODT */
        } ddbpakettype

        struct {
            /* Daten von VC */
        } vcpakettype

        struct {
            /* Daten von DM */
        } dmpakettype

        struct {
            /* interne Daten von DB */
        } dbinternpakettype

        struct {
            /* Daten von DB an VC */
        } dbpakettype


bool DDB_State (int *State)
{
    /* Status von der DDB anfordern */
}

bool VC_State (int *State)
{
    /* Status von VC anfordern */
}

bool DM_State (int *State)
{
    /* Status vom DM anfordern */
}

bool DDB_Message (ddbpakettype *DDBPaket)
{
    /* Daten von der DDB anfordern */
}

bool VC_Message (vcpakettype *VCPaket)
{
    /* Daten von VC anfordern */
}

bool DM_Message (dmpakettype *DMPaket)
{
    /* Daten vom DM anfordern */
}
```

```
void Send_Message_VC (&BDPaket)
{
    /* Daten an VC schicken */
}

% --------------------- Initialising program ------------------------------

void Init_Variables
  (bd_pakettype *BDPaket, ddb_pakettype *DDBPaket,
  vc_pakettype *VCPaket, dm_pakettype *DMPaket)
{
  BDPaket->BDState        = PASSIVE,
  BDPaket->lch            = OWN,
  BDPaket->clch           = FALSE,
  BDPaket->trafficators   = NONE,
  BDPaket->oscillate      = FALSE,
  BDPaket->emergency_stop = FALSE,
  DDBPaket->RTState       = PASSIVE,
  DDBPaket->ODTState      = PASSIVE,
  VCPaket->VCState        = PASSIVE,
  DMPaket->DMState        = PASSIVE;
}

% Waiting for activation of module behaviour decision
bool Wait_for_Activate
  (ddb_pakettype *DDBPaket, vc_pakettype *VCPaket, dm_pakettype *DMPaket)
{
  repeat
  until ((DDBRT_State (&DDBPaket->RTState) && (DDBPaket->RTState = ACTIVE) &&
          DDBODT_State (&DDBPaket->ODTState) && (DDBPaket->ODTState = ACTIVE)  &&
          VC_State (&VCPaket->VCState) && (VCPaket->VCState = ACTIVE)  &&
          DM_State (&DMPaket->DMState) && (DMPaket->DMState = ACTIVE)) ||
         DMPaket->DMState = STOP);
  return (DMPaket->DMState != STOP);
}

% --------------------- Non expert part ------------------------------------

void Calculate_Trajectory
  (ddb_pakettype *DDBPaket, vc_pakettype *VCPaket, int where, float *b)
{
  /* Formel zur Berechnung einer Trajektorie */
}

void Calculate_Trajectories
  (ddb_pakettype *DDBPaket, vc_pakettype *VCPaket,
   float *bown, float *bleft, float *bright)
{
  Calculate_Trajectory (DDBPaket, VCPaket, FRONT, bown);
  Calculate_Trajectory (DDBPaket, VCPaket, LEFT,  bleft);
  Calculate_Trajectory (DDBPaket, VCPaket, RIGHT, bright);
}

void Calculate_Velocity
  (ddb_pakettype *DDBPaket, vc_pakettype *VCPaket, int where, float *v)
{
  /* Formel zur Berechnung der Geschwindigkeiten eines anderen Fahrzeugs */
}

void Calculate_Velocities
  (ddb_pakettype *DDBPaket, vc_pakettype *VCPaket,
   float *leftv, float *frontv)
{
  Calculate_Velocity (DDBPaket, VCPaket, FRONT, frontv);
  Calculate_Velocity (DDBPaket, VCPaket, LEFT,  leftv);
}
```

```
% --------------------- Expert part ----------------------------------------

% Aquiring basic variables
void Calculate_BD_Variables
  (bdintern_pakettype *BDIntern, ddb_pakettype *DDBPaket,
   vc_pakettype *VCPaket, dm_pakettype *DMPaket, float bown,
   float bleft, float bright, float leftv, float frontv)
{
  % Car is moving
  BDIntern->cim =
      VCPaket->vms >= VMIN;

  % Left lane change request
  BDIntern->llcr =
      DDBPaket->LeftLaneExists && VCPaket->convoy &&
      (DMPaket->Vplanned > VCPaket->vms + VEPSILON) &&
      (leftv > frontv + VEPSILON);

  % Right lane change request
  BDIntern->rlcr =
      DDBPaket->RightLaneExists;

  % Change to left lane is possible
  BDIntern->cllp =
      BDIntern->cim && DMPaket->PassingAllowed && DDBPaket->LeftLaneExists &&
      (bleft > TR_SICHER) && (DMPaket->dmcllp == FREIGABE);

  % Cancel change to left lane
  BDIntern->ccll =
      VCPaket->chtll_ack &&
      ((!DDBPaket->CwReadyLeft && (bleft < TR_ABBRUCH) && (bown >= TR_SICHER)) //
       (DDBPaket->CwReadyLeft && (bown < TR_ABBRUCH) && (bright >= TR_SICHER) &&
        (DMPaket->dmccllp == FREIGABE)) //
       ((DMPaket->dmccll == FREIGABE) && (DMPaket->dmccllp == FREIGABE)));

  %  Change to right lane is possible
  BDIntern->crlp =
      BDIntern->cim &&
      DDBPaket->RightLaneExists &&
      (bright > TR_SICHER) && (DMPaket->dmcrlp == FREIGABE);

  % Cancel change to right lane
  BDIntern->ccrl  =
      VCPaket->chtrl_ack &&
      ((!DDBPaket->CwReadyRight && (bright < TR_ABBRUCH) && (bown >= TR_SICHER)) //
       (DDBPaket->CwReadyRight && (bown < TR_ABBRUCH) && (bleft >= TR_SICHER) &&
        (DMPaket->dmccrlp == FREIGABE)) //
       ((DMPaket->dmccrl == FREIGABE) && (DMPaket->dmccrlp == FREIGABE)));
}

% Calculating new state
Decide_Next_State (int *state,
   bdintern_pakettype *BDIntern, ddb_pakettype *DDBPaket,
   vc_pakettype *VCPaket, dm_pakettype *DMPaket)
{
  if (VCPaket->emergency_stop && (state != 23))
    state = 24;
  else if (!BDIntern->cim && !VCPaket->emergency_stop && (state != 24))
    state = 23;
  else if (state == 1)
  {
    if (BDIntern->llcr && DDBPaket->LeftLaneExists && DMPaket->PassingAllowed)
      state = 2;
    else if (BDIntern->rclr)
      state = 13;
  }
  else if (state == 2)
```

```
{
  if (!BDIntern->llcr || !DDBPaket->LeftLaneExists || !DMPaket->PassingAllowed)
    state = 1;
  else if (BDIntern->llcr && DDBPaket->LeftLaneExists && DMPaket->PassingAllowed &&
           (time > T1))
    state = 3;
}
else if (state == 3)
{
  if (!BDIntern->llcr || !DDBPaket->LeftLaneExists || !DMPaket->PassingAllowed)
    state = 1;
  else if (BDIntern->cllp && BDIntern->llcr && (time <= T2))
    state = 4;
  else if (BDIntern->llcr && (time > T2))
    state = 8;
}
else if (state == 4)
{
  if (BDIntern->cllp && BDIntern->llcr && (time > T3))
    state = 5;
  else if (!BDIntern->cllp || !BDIntern->llcr)
    state = 6;
}
else if (state == 5)
{
  if (VCPaket->chtll_ack)
    state = 7;
}
else if (state == 6) state = 1;
else if (state == 7)
{
  if (!VCPaket->lch_active && !VCPaket->chtll_ack)
    state = 6;
  else if (DMPaket->ccll)
    state = 10;
}
else if (state == 8)
{
  if (!BDIntern->llcr || !DDBPaket->LeftLaneExists || !DMPaket->PassingAllowed)
    state = 1;
  else if (BDIntern->cllp && BDIntern->llcr && (time <= T2))
    state = 4;
  else if (BDIntern->llcr && (time > T2))
    state = 9;
}
else if (state == 9)
{
  if (!BDIntern->llcr || !DDBPaket->LeftLaneExists || !DMPaket->PassingAllowed)
    state = 1;
  else if (BDIntern->cllp && BDIntern->llcr && (time <= T2))
   state = 5;
  else if (BDIntern->llcr && (time > T2))
   state = 8;
}
else if (state == 10)
{
  if (!VCPaket->chtll_ack)
   state = 11;
}
else if (state == 11)
{
  if (!VCPaket->lch_active)
    state = 12;
}
else if (state == 12) state = 1;
else if (state == 13)
{
```

```
    if (!BDIntern->rclr || !DDBPaket->RightLaneExists)
      state = 1;
    else if (BDIntern->llcr && DDBPaket->LeftLaneExists && DMPaket->PassingAllowed)
      state = 2;
    else if (BDIntern->crlp && BDIntern->rlcr && (time <= T2))
      state = 14;
    else if (BDIntern->rlcr && (time > T2))
      state = 18;
  }
  else if (state == 14)
  {
    if (BDIntern->crlp && BDIntern->rlcr && (time > T3))
      state = 15;
    else if (!BDIntern->crlp || !BDIntern->rlcr)
      state = 16;
  }
  else if (state == 15)
  {
    if (VCPaket->chtrl_ack)
      state = 17;
  }
  else if (state == 16) state = 1;
  else if (state == 17)
  {
    if (!VCPaket->lch_active && !VCPaket->chtrl_ack)
      state = 16;
    else if (DMPaket->ccrl)
      state = 20;
  }
  else if (state == 18)
  {
    if (!BDIntern->rlcr || !DDBPaket->RightLaneExists)
      state = 1;
    else if (BDIntern->crlp && BDIntern->rlcr && (time <= T2))
      state = 14;
    else if (BDIntern->rlcr && (time > T2))
      state = 19;
  }
  else if (state == 19)
  {
    if (!BDIntern->rlcr || !DDBPaket->RightLaneExists)
      state = 1;
    else if (BDIntern->crlp && BDIntern->rlcr && (time <= T2))
      state = 15;
    else if (BDIntern->rlcr && (time > T2))
      state = 18;
  }
  else if (state == 20)
  {
    if (!VCPaket->chtrl_ack)
      state = 21;
  }
  else if (state == 21)
  {
    if (!VCPaket->lch_active)
      state = 22;
  }
  else if (state == 22) state = 1;
  else if (state == 23)
  {
    if (cim) state = 1;
  }
  else if (state == 24)
  {
    if (!VCPaket->emergency_stop)
      state = 1;
} }
```

```
Act_Next_State (int *state)
{
  if (state == 1)
  {
    BDPaket->trafficators = NONE,
    BDPaket->emergency_stop = FALSE,
  }
  else if (state == 4)
    BDPaket->trafficators = LEFT;
  else if (state == 5)
  {
    BDPaket->trafficators = LEFT;
    BDPaket->oscillate = FALSE;
    BDPaket->lch = LEFT;
  }
  else if (state == 6)
  {
    BDPaket->trafficators = NONE;
    BDPaket->lch = OWN;
  }
  else if (state == 7)
    BDPaket->lch = OWN;
  else if (state == 8)
  {
    BDPaket->trafficators = LEFT;
    BDPaket->oscillate = FALSE;
  }
  else if (state == 9)
  {
    BDPaket->trafficators = LEFT;
    BDPaket->oscillate = TRUE;
  }
  else if (state == 10)
  {
    BDPaket->trafficators = RIGHT;
    BDPaket->clch = TRUE;
  }
  else if (state == 11)
    BDPaket->clch = FALSE;
  else if (state == 12)
    BDPaket->trafficators = NONE;
  else if (state == 14)
    BDPaket->trafficators = RIGHT;
  else if (state == 15)
  {
    BDPaket->trafficators = RIGHT;
    BDPaket->oscillate = FALSE;
    BDPaket->lch = LEFT;
  }
  else if (state == 16)
  {
    BDPaket->trafficators = NONE;
    BDPaket->lch = OWN;
  }
  else if (state == 17)
    BDPaket->lch = OWN;
  else if (state == 18)
  {
    BDPaket->trafficators = NONE;
    BDPaket->oscillate = FALSE;
  }
  else if (state == 19)
  {
    BDPaket->trafficators = RIGHT;
    BDPaket->oscillate = TRUE;
  }
```

```
  else if (state == 20)
  {
    BDPaket->trafficators = LEFT;
    BDPaket->clch = TRUE;
  }
  else if (state == 21)
    BDPaket->clch = FALSE;
  else if (state == 22)
  {
    BDPaket->trafficators = NONE;
    BDPaket->lch = OWN;
  }
  else if (state == 23)
  {
    BDPaket->trafficators = NONE;
    BDPaket->oscillate = FALSE;
    BDPaket->lch = OWN;
    BDPaket->clch = FALSE;
  }
  else if (state == 24)
  {
    BDPaket->emergency_stop = TRUE;
    BDPaket->trafficators = BOTH;
    BDPaket->oscillate = FALSE;
    BDPaket->lch = OWN;
    BDPaket->clch = FALSE;
  }
}

% Main routine
Decide_Behaviour (int *state)
{
  Decide_Next_State (state);
  Act_Next_State (state);
}

% --------------------- Main part -----------------------------------------

main()
{
  float bown, bleft, bright, leftv, frontv;
  int state;
  ddb_pakettype        DDBPaket;
  vc_pakettype         VCPaket;
  dm_pakettype         DMPaket;
  bd_pakettype         BDPaket;
  bdintern_pakettype   BDIntern;
  Init_Variables (&BDPaket, &DDBPaket, &VCPaket, &DMPaket);
  if Wait_for_Activate (&DDBPaket, &VCPaket, &DMPaket)
  {
    BD_Paket.BDState = ACTIVE;
    do
    {
      repeat
      until (DDB_Message (&DDBPaket) && VC_Message (&VCPaket) &&
             DM_Message (&DMPaket))
      Calculate_Trajectories (DDBPaket, &VCPaket, &bown, &bleft, &bright);
      Calculate_Velocities   (&DDBPaket, &VCPaket, &leftv, &frontv);
      Calculate_BD_Variables (&BDIntern, &DDBPaket, &VCPaket, &DMPaket,
                              bown, bleft, bright, leftv, frontv);
      Decide_Behaviour (&state, &BDIntern, &DDBPaket, &VCPaket, &DMPaket);
      Send_Message_VC (&BDPaket);
    }
    while (DM_Paket.DMState != STOP);
  }
  BD_Paket.BDState = STOP;
}
```

B.5 Spezifikation des Regelteils in PROLOG

Dieser Teil des Anhangs enthält den Programmcode des Regelteils des Moduls Behaviour Desicion in $\text{DATALOG}_f^{\neg}$.

Bemerkungen:

- Vom Original-C-Programm wurde nur der Teil der sich mit der Entscheidungsfindung befaßt nach $\text{DATALOG}_f^{\neg}$transferiert.
- Fakten, die eine Schnittstelle zur Datenbank darstellen und im Prinzip lediglich einen boolschen Wert liefern müssen, ("gültig" oder "nicht gültig") benötigen dafür einen Parametert, da nur so ein Zugriff auf den Inhalt der Datenbank möglich ist. Der Ergebniswert des Fakts genügt dafür nicht.
- Da sich mit $\text{DATALOG}_f^{\neg}$nur Regeln der Form "if ... then ..." ausdrücken lassen mußte bei der Umsetzung der "if ... then ... else ..." Regeln auf eine Vervollständigung der Abfrage geachtet werden.
- Die Zahlen der Regeln "rule8" bis "rule34" entsprechen den Ziffern im Zustandsübergangsdiagramm.
- Die häufig vorkommenden Negationen sind unproblematisch, da ausschließlich Literale negiert werden, die keine Parameter besitzen, d.h. es wird nur der boolsche Wert des Ergebnisparameters negiert.

```
% --------------------- Basic Calculations ---------------------------------

%  time steps
time_less (Time) :-
    clock (CL), time (TL), CL < TL.

%  car is moving
car_moves () :-
    vc_vms (VMS), vmin (VMIN), VMS >= VMIN.

%  lelt lane change request
left_changereq () :-
    ddb_LeftLaneExists (true), vc_convoy (true),
    dm_Vplanned (VP), vc_vms (VMS),
    vel_difference (VEPSILON),
    get_leftv (LEFTV), get_frontv (FRONTV),
    VP > VMS + VEPSILON, LEFTV > FRONTV + VEPSILON.

%  right lane change request
right_changereq () :-
    ddb_RightLaneExists (true).

%  change to left lane is possible
change_left_possible () :-
    car_moves (),
    dm_PassingAllowed (true), ddb_LeftLaneExists (true),
    get_bleft (BLEFT), get_secure_val (TR_SICHER),
    BLEFT > TR_SICHER,
    dm_dmcllp ("freigabe").

%  cancel change to left lane
cancel_change_left () :-
    vc_chtll_ack (true), ddb_CwReadyLeft (true),
    get_bleft (BLEFT), get_bown (BOWN),
    get_cancel_val (TR_ABBRUCH), get_secure_val (TR_SICHER),
    BLEFT < TR_ABBRUCH, BOWN >= TR_SICHER.
```

```
cancel_change_left () :-
    vc_chtll_ack (true), ddb_CwReadyLeft (true),
    get_bown (BOWN), get_bright (BRIGHT),
    get_cancel_val (TR_ABBRUCH), get_secure_val (TR_SICHER),
    BOWN < TR_ABBRUCH, BRIGHT >= TR_SICHER,
    dm_dmccllp ("freigabe").
cancel_change_left () :-
    vc_chtll_ack (true),
    dm_dmccll ("freigabe"), dm_dmccllp ("freigabe").

%  change to right lane is possible
change_right_possible () :-
    car_moves (),
    ddb_RightLaneExists (true),
    get_bright (BRIGHT), get_secure_val (TR_SICHER),
    BRIGHT > TR_SICHER,
    dm_crlp ("freigabe").

%  cancel change to right lane
cancel_change_right () :-
    vc_chtrl_ack (true),
    ddb_CwReadyRight (true),
    get_bright (BRIGHT), get_bown (BOWN),
    get_cancel_val (TR_ABBRUCH), get_secure_val (TR_SICHER),
    BRIGHT < TR_ABBRUCH, BOWN >= TR_SICHER.
cancel_change_right () :-
    vc_chtrl_ack (true),
    ddb_CwReadyRight (true),
    get_bown (BOWN), get_bleft (BLEFT),
    get_cancel_val (TR_ABBRUCH), get_secure_val (TR_SICHER),
    BOWN < TR_ABBRUCH, BLEFT >= TR_SICHER,
    dm_dmccrlp ("freigabe").
cancel_change_right () :-
    vc_chtrl_ack (true),
    dm_dmccrl ("freigabe"),
    dm_dmccrlp ("freigabe").


% --------------------- Rules for State Transistions -----------------------

rule8() :-
    left_changereq (),
    ddb_LeftLaneExists (true), dm_PassingAllowed (true).

rule9() :-
    get_time (T1), not (time_less (T1)).

rule10() :-
    not (left_changereq ()).
rule10() :-
    ddb_LeftLaneExists (false).
rule10() :-
    dm_PassingAllowed (false).

rule11() :-
    ddb_LeftLaneExists (true), dm_PassingAllowed (true),
    change_left_possible (), left_changereq (),
    get_time (T2), time_less (T2).

rule12() :-
    change_left_possible (), left_changereq (),
    get_time (T3), not (time_less (T3)).

rule13() :-
    not (change_left_possible ()).
rule13() :-
    not (left_changereq ()).
```

```
rule14() :-
    vc_chtll_ack (true).

rule15() :-
    vc_lch_active (false), vc_chtll_ack (false),
    not (cancel_change_left ()).

rule16() :-
    ddb_LeftLaneExists (true), dm_PassingAllowed (true),
    left_changereq (),
    get_time (T2), not (time_less (T2)).

rule17() :-
    cancel_change_left ().

rule18() :-
    vc_chtll_ack (false).

rule19() :-
    vc_lch_active (false).

rule20() :-
    right_changereq ().

rule21() :-
    not (right_changereq ()).
rule21() :-
    ddb_RightLaneExists (false).

rule22() :-
     ddb_RightLaneExists (true),
     change_right_possible (), right_changereq (),
     get_time (T2), time_less (T2).

rule23() :-
    change_right_possible (), right_changereq (),
    get_time (T3), not (time_less (T3)).

rule24() :-
    not (change_right_possible ()).
rule24() :-
    not (right_changereq ()).

rule25() :-
    vc_chtrl_ack (true).

rule26() :-
    vc_lch_active (false), vc_chtrl_ack (false),
    not (cancel_change_right ()).

rule27() :-
    ddb_RightLaneExists (true),
    right_changereq (),
    get_time (T2), not (time_less (T2)).

rule28() :-
    cancel_change_right ().

rule29() :-
    vc_chtrl_ack (false).

rule30() :-
    vc_lch_active (false).

rule31() :-
    State =\= 23,
    vc_emergency_stop (true).
```

```
rule32() :-
    car_moves ().

rule33() :-
    State =\= 24,
    vc_emergency_stop (true),
    not (car_moves ()).

rule34() :-
    vc_emergency_stop (false).

% --------------------- State Transistions ---------------------------------

calc_statetransition (State, 24) :- rule31 ().

calc_statetransition (State, 23) :- rule33 ().

calc_statetransition (1, 2)      :- rule8 ().
calc_statetransition (1, 13)     :- rule20 ().

calc_statetransition (2, 1)      :- rule10 ().
calc_statetransition (2, 3)      :- not (rule8 ()), rule9 ().

calc_statetransition (3, 1)      :- rule10 ().
calc_statetransition (3, 4)      :- rule11 ().
calc_statetransition (3, 8)      :- rule16 ().

calc_statetransition (4, 5)      :- rule12 ().
calc_statetransition (4, 6)      :- rule13 ().

calc_statetransition (5, 7)      :- rule14 ().

calc_statetransition (6, 1).

calc_statetransition (7, 6)      :- rule15 ().
calc_statetransition (7, 10)     :- rule17 ().

calc_statetransition (8, 1)      :- rule10 ().
calc_statetransition (8, 4)      :- rule11 ().
calc_statetransition (8, 9)      :- rule16 ().

calc_statetransition (9, 1)      :- rule10 ().
calc_statetransition (9, 5)      :- rule11 ().
calc_statetransition (9, 8)      :- rule16 ().

calc_statetransition (10, 11)    :- rule18 ().

calc_statetransition (11, 12)    :- rule19 ().

calc_statetransition (12, 1).

calc_statetransition (13, 1)     :- not (rule8 ()), rule20 ().
calc_statetransition (13, 2)     :- rule8 ().
calc_statetransition (13, 14)    :- not (rule8 ()), rule22 ().
calc_statetransition (13, 18)    :- not (rule8 ()), rule27 ().

calc_statetransition (14, 15)    :- rule23 ().
calc_statetransition (14, 16)    :- rule24 ().

calc_statetransition (15, 17)    :- rule25 ().

calc_statetransition (16, 1).

calc_statetransition (17, 16)    :- rule26 ().
calc_statetransition (17, 20)    :- rule28 ().
```

```
calc_statetransition (18, 1)     :- rule21 ().
calc_statetransition (18, 14)    :- rule22 ().
calc_statetransition (18, 19)    :- rule27 ().

calc_statetransition (19, 1)     :- rule21 ().
calc_statetransition (19, 15)    :- rule22 ().
calc_statetransition (19, 18)    :- rule27 ().

calc_statetransition (20, 21)    :- rule29 ().

calc_statetransition (21, 22)    :- rule30 ().

calc_statetransition (22, 1).

calc_statetransition (23, 1)     :- rule32 ().

calc_statetransition (24, 1)     :- rule34 ().

calc_statetransition (State, State).


% ---------------------- Outputs of actual state ----------------------------

execute_stateactions (1, 0, 0, 0, 0, 0).
execute_stateactions (2, 0, 0, 0, 0, 0).
execute_stateactions (3, 0, 0, 0, 0, 0).
execute_stateactions (4, 0, 0, 0, 2, 0).
execute_stateactions (5, 2, 0, 0, 2, 0).
execute_stateactions (6, 0, 0, 0, 0, 0).
execute_stateactions (7, 0, 0, 0, 2, 0).
execute_stateactions (8, 0, 0, 0, 2, 0).
execute_stateactions (9, 0, 0, 1, 2, 0).
execute_stateactions (10, 0, 1, 0, 1, 0).
execute_stateactions (11, 0, 0, 0, 1, 0).
execute_stateactions (12, 0, 0, 0, 0, 0).
execute_stateactions (13, 0, 0, 0, 0, 0).
execute_stateactions (14, 0, 0, 0, 1, 0).
execute_stateactions (15, 1, 0, 0, 1, 0).
execute_stateactions (16, 0, 0, 0, 0, 0).
execute_stateactions (17, 0, 0, 0, 1, 0).
execute_stateactions (18, 0, 0, 0, 1, 0).
execute_stateactions (19, 0, 0, 1, 1, 0).
execute_stateactions (20, 0, 1, 0, 2, 0).
execute_stateactions (21, 0, 0, 0, 2, 0).
execute_stateactions (22, 0, 0, 0, 0, 0).
execute_stateactions (23, 0, 0, 0, 0, 0).
execute_stateactions (24, 0, 0, 0, 3, 1).


% ---------------------- Main Routine ----------------------------------------

% Control of Module Behaviour Decision
check_activation (wait, wait)     :- dm_state (DMS), DMS =\= 0, DMS =\= 2.
check_activation (wait, wait)     :- ddb_rt_state (RTS), RTS =\= 2.
check_activation (wait, wait)     :- ddb_odt_state (ODTS), ODTS =\= 2.
check_activation (wait, wait)     :- vc_state (VCS), VCS =\= 2.
check_activation (wait, off)      :- dm_state (0).
check_activation (wait, active)   :- dm_state (2),
                                     ddb_rt_state (2), ddb_odt_state (2),
                                     vc_state (2).
check_activation (active, active) :- dm_state (DMS), DMS =\= 0.
check_activation (active, off)    :- dm_state (0).
```

```
% Main routine
behaviour_decision (1, 0, 0, 0, 0, 0) :-
    get_vehicle_activation (Active),
    check_activation (Active, wait)  .

behaviour_decision (0, 0, 0, 0, 0, 0) :-
    get_vehicle_activation (Active),
    check_activation (Active, off).

behaviour_decision (2, LCH, CLCH, OSCI, TRAFF, EMST) :-
    get_vehicle_activation (Active),
    check_activation (Active, active),
    get_vehicle_state (OldState),
    calc_statetransition (OldState, State),
    execute_stateactions (State, LCH, CLCH, OSCI, TRAFF, EMST).

:- behaviour_decision (BDStateFlag, LCH, CLCH, OSCILLATE, TRAFFICATORS, EMERGENCY_STOP).


% --------------------- Interface ------------------------------------------

% Own
clock (CL).
time (TL).
get_time (TIME).
get_vehicle_state (State).
get_vehicle_activation (Active).
vmin (VMIN).
vel_difference (VEPSILON).
get_leftv (LEFTV).
get_frontv (FRONTV).
get_bleft (BLEFT).
get_bright (BRIGHT).
get_bown (BOWN).
get_secure_val (TR_SICHER).
get_cancel_val (TR_ABBRUCH).

% VC
vc_state (VCS).
vc_vms (VMS).
vc_emergency_stop (B).
vc_convoy (B).
vc_chtll_ack (B).
vc_chtrl_ack (B).
vc_lch_active (B).

% DDB (RT und ODT)
ddb_rt_state (RTS).
ddb_odt_state (ODTS).
ddb_RightLaneExists (B).
ddb_LeftLaneExists (B).
ddb_CwReadyLeft (B).
ddb_CwReadyRight (B).

% DM
dm_state (DMS).
dm_Vplanned (VP).
dm_PassingAllowed (B).
dm_dmccll (ACTION).
dm_dmccrl (ACTION).
dm_dmcllp (ACTION).
dm_dmccllp (ACTION).
dm_dmcrlp (ACTION).
dm_dmccrlp (ACTION).
```

B.6 Datenflußgraph-Darstellung des Regelteils

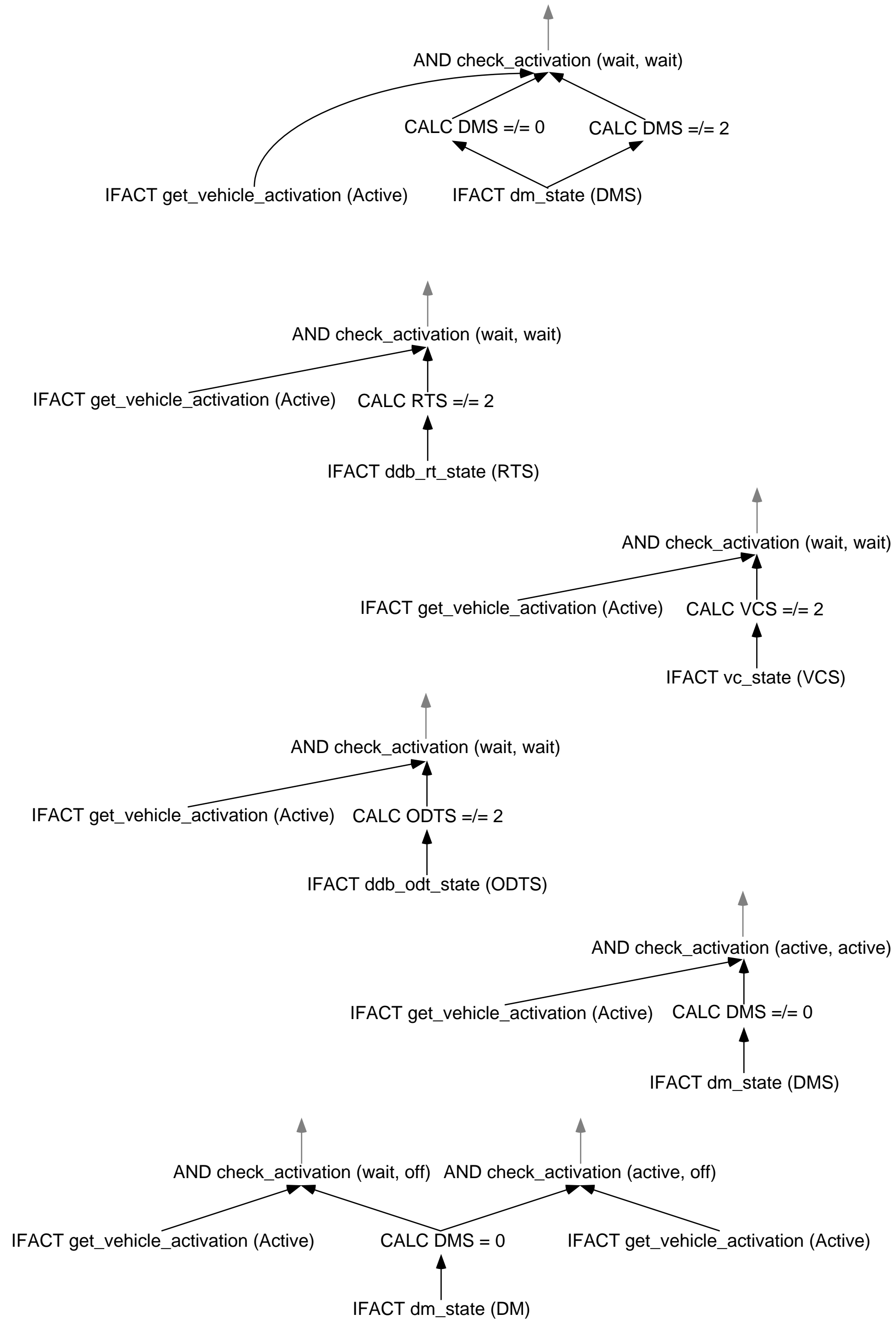
Die folgenden Abbildungen zeigen die verschiedenen Teile des Datenflußgraphen zur $\text{DATALOG}_f^{\neg}$-Version des Moduls Behaviour Decision von VaMoRs.

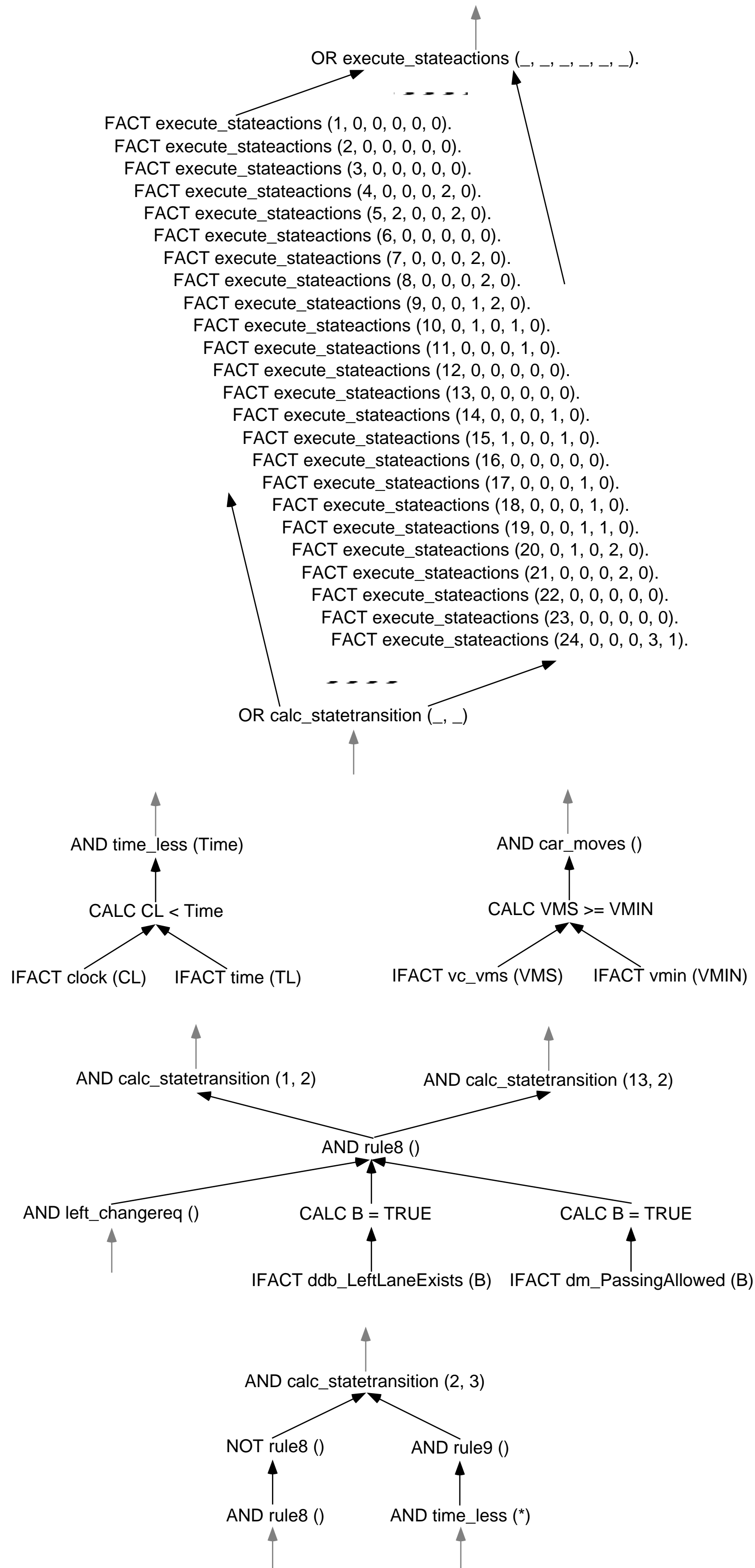
Bemerkung:

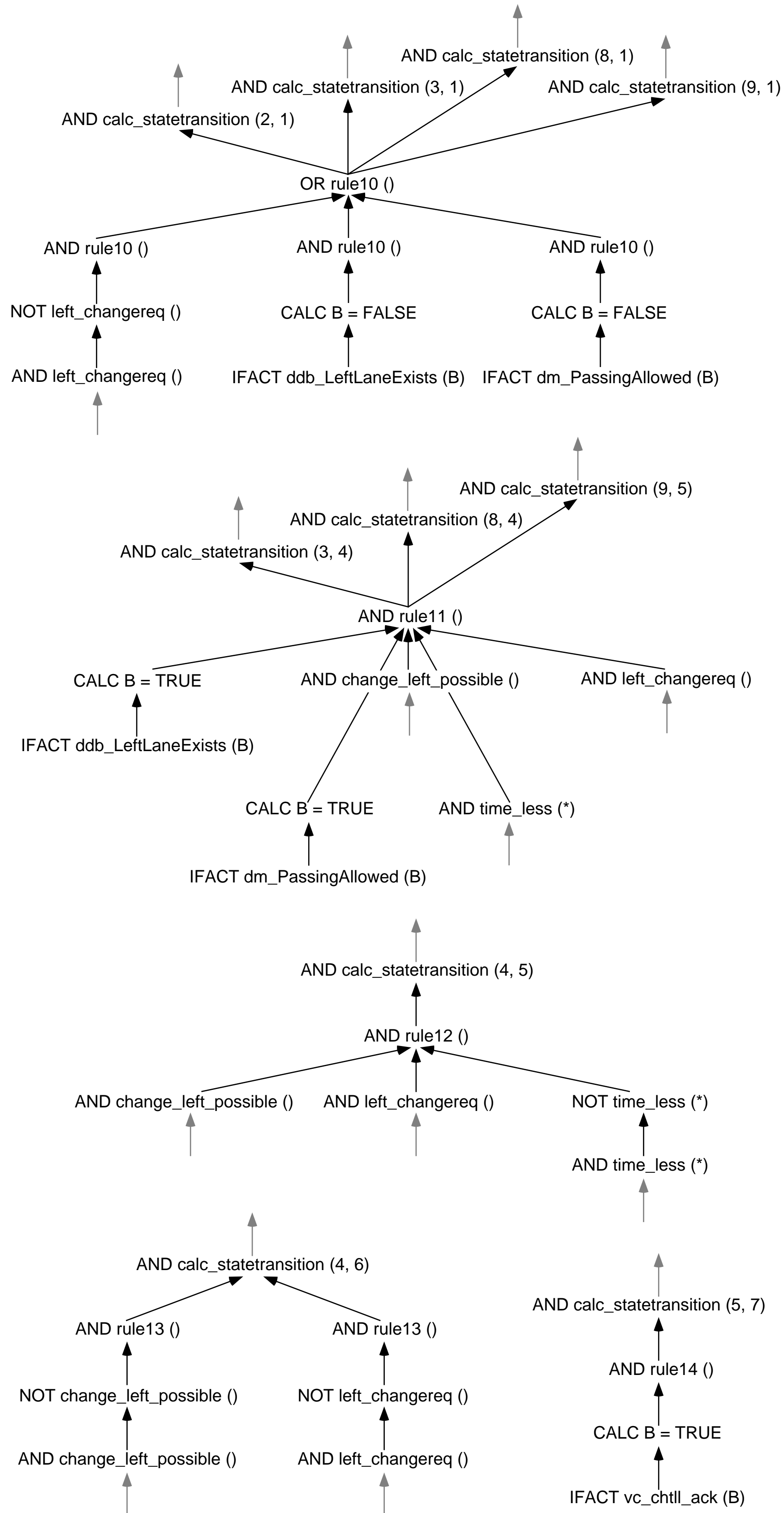
- Die Graphen mußten aus Platzgründen in Teilgraphen aufgeteilt werden. Sie sind jedoch vollständig aufgeführt. Graue Pfeile bedeuten, daß entweder von anderen Teilgraphen Eingaben kommen, oder an andere Teilgraphen Ausgaben weitergehen.
- In OR-Knoten sind die Parameternamen grundsätzlich "_". Diese Nameswahl beruht darauf, daß es aufgrund der verschiedenen Parameternamen der nachfolgenden AND-Knoten nicht möglich ist, konkrete Parameternamen zu vergeben.
- Der "*" in der Parameterliste einiger Knoten besagt, daß es sich bei diesem Parameter um einen Parameter mit Eingabemodus handelt. Dieser wird nicht über den Knoten eingegeben, sondern von unten direkt an die entsprechenden Verwendungsstellen gegeben.

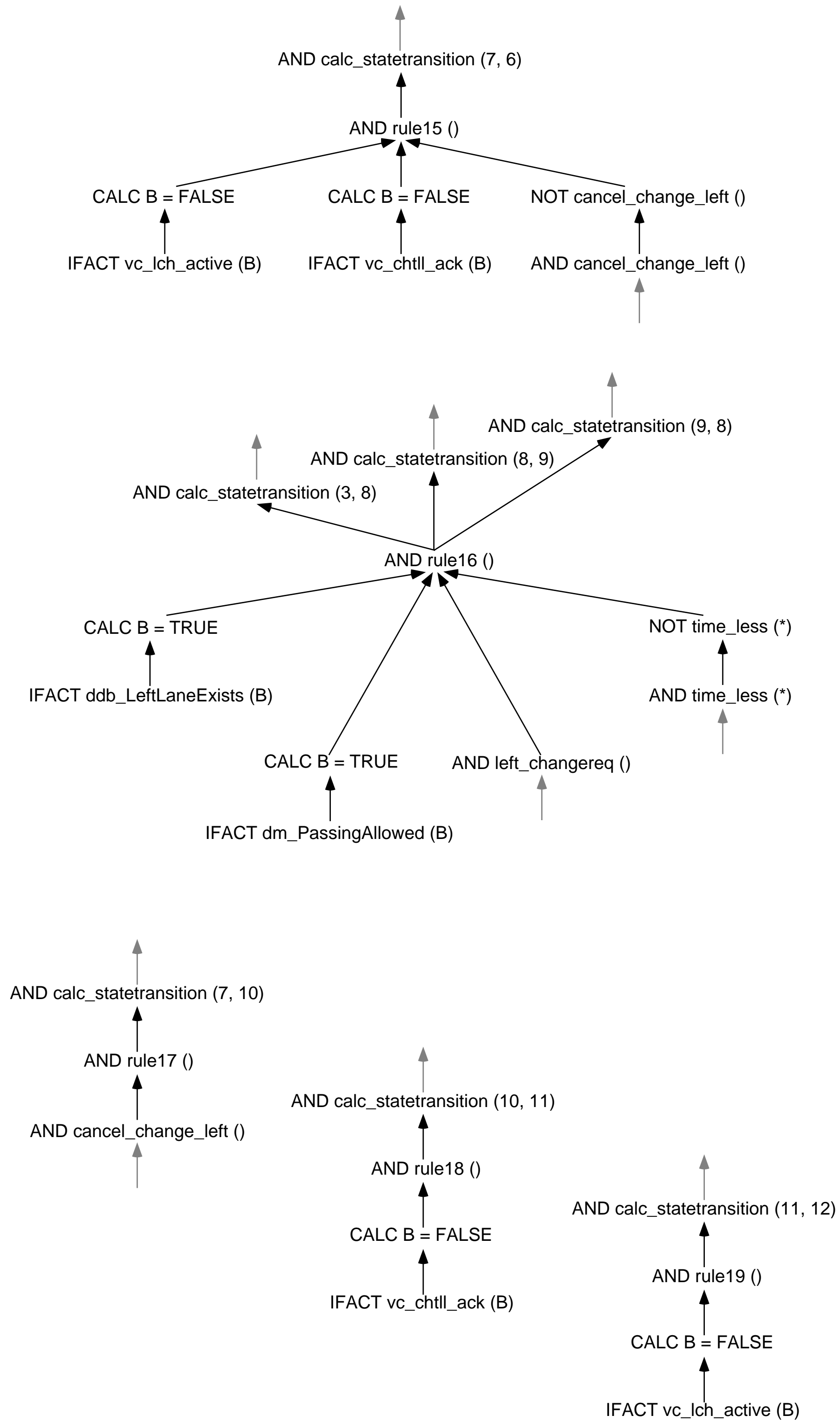
QUERY
OR behaviour_decision (_, _, _, _, _, _)
AND behaviour_decision (0, 0, 0, 0, 0, 0)
AND behaviour_decision (1, 0, 0, 0, 0, 0)
CALC NewState = off
CALC NewState = wait
AND behaviour_decision (2, LCH, CLCH, OSCI, TRAFF, EMST)
CALC NewState = active
OR check_activation (Active, *)
OR execute_stateactions (*, LCH, CLCH, OSC, TRAFF, EMST)

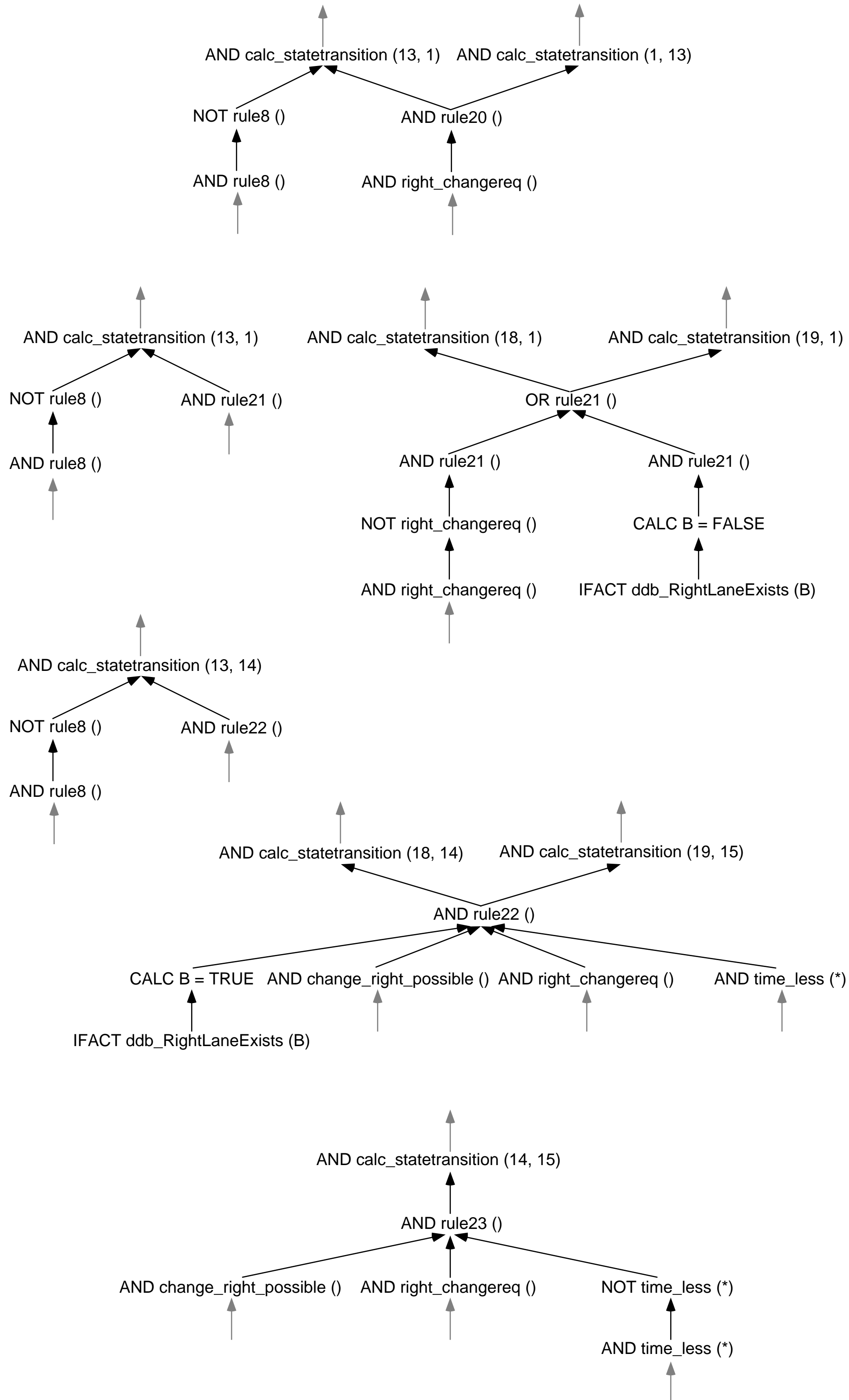
AND check_activation (active, active)
CALC DMS = 2
CALC VCS = 2
IFACT dm_state (DMS)
IFACT vc_state (VCS)
IFACT get_vehicle_activation (Active)
CALC RTS = 2
CALC ODTS = 2
IFACT ddb_rt_state (RTS)
IFACT ddb_odt_state (ODTS)

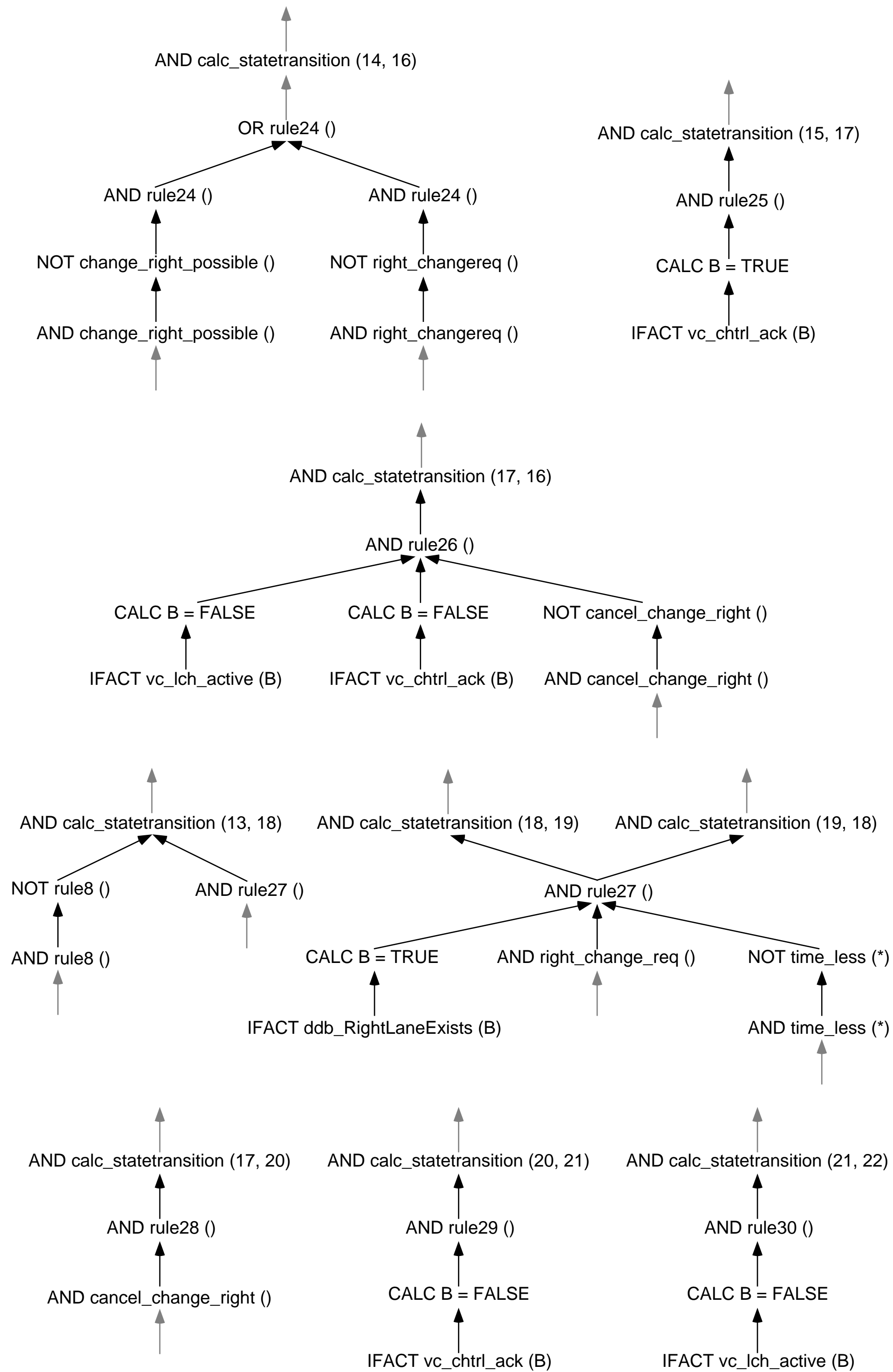

AND check_activation (wait, wait)
CALC DMS =/= 0
CALC DMS =/= 2
IFACT get_vehicle_activation (Active)
IFACT dm_state (DMS)
AND check_activation (wait, wait)
IFACT get_vehicle_activation (Active)
CALC RTS =/= 2
IFACT ddb_rt_state (RTS)
AND check_activation (wait, wait)
IFACT get_vehicle_activation (Active)
CALC VCS =/= 2
IFACT vc_state (VCS)
AND check_activation (wait, wait)
IFACT get_vehicle_activation (Active)
CALC ODTS =/= 2
IFACT ddb_odt_state (ODTS)
AND check_activation (active, active)
IFACT get_vehicle_activation (Active)
CALC DMS =/= 0
IFACT dm_state (DMS)
AND check_activation (wait, off)
AND check_activation (active, off)
IFACT get_vehicle_activation (Active)
CALC DMS = 0
IFACT get_vehicle_activation (Active)
IFACT dm_state (DM)

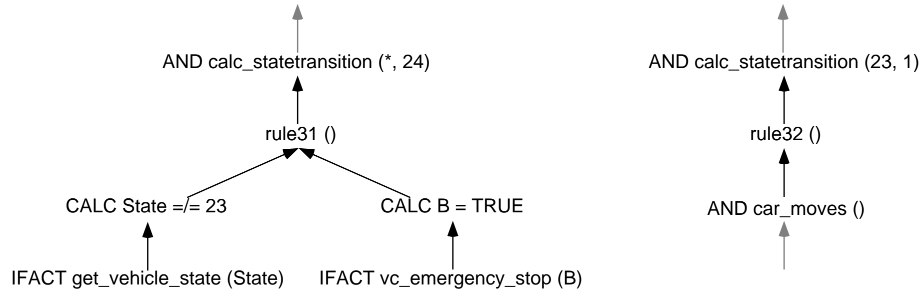

OR execute_stateactions (_, _, _, _, _, _).
FACT execute_stateactions (1, 0, 0, 0, 0, 0).
FACT execute_stateactions (2, 0, 0, 0, 0, 0).
FACT execute_stateactions (3, 0, 0, 0, 0, 0).
FACT execute_stateactions (4, 0, 0, 0, 2, 0).
FACT execute_stateactions (5, 2, 0, 0, 2, 0).
FACT execute_stateactions (6, 0, 0, 0, 0, 0).
FACT execute_stateactions (7, 0, 0, 0, 2, 0).
FACT execute_stateactions (8, 0, 0, 0, 2, 0).
FACT execute_stateactions (9, 0, 0, 1, 2, 0).
FACT execute_stateactions (10, 0, 1, 0, 1, 0).
FACT execute_stateactions (11, 0, 0, 0, 1, 0).
FACT execute_stateactions (12, 0, 0, 0, 0, 0).
FACT execute_stateactions (13, 0, 0, 0, 0, 0).
FACT execute_stateactions (14, 0, 0, 0, 1, 0).
FACT execute_stateactions (15, 1, 0, 0, 1, 0).
FACT execute_stateactions (16, 0, 0, 0, 0, 0).
FACT execute_stateactions (17, 0, 0, 0, 1, 0).
FACT execute_stateactions (18, 0, 0, 0, 1, 0).
FACT execute_stateactions (19, 0, 0, 1, 1, 0).
FACT execute_stateactions (20, 0, 1, 0, 2, 0).
FACT execute_stateactions (21, 0, 0, 0, 2, 0).
FACT execute_stateactions (22, 0, 0, 0, 0, 0).
FACT execute_stateactions (23, 0, 0, 0, 0, 0).
FACT execute_stateactions (24, 0, 0, 0, 3, 1).
OR calc_statetransition (_, _)
AND time_less (Time)
CALC CL < Time
IFACT clock (CL)
IFACT time (TL)
AND car_moves ()
CALC VMS >= VMIN
IFACT vc_vms (VMS)
IFACT vmin (VMIN)
AND calc_statetransition (1, 2)
AND calc_statetransition (13, 2)
AND rule8 ()
AND left_changereq ()
CALC B = TRUE
CALC B = TRUE
IFACT ddb_LeftLaneExists (B)
IFACT dm_PassingAllowed (B)
AND calc_statetransition (2, 3)
NOT rule8 ()
AND rule9 ()
AND rule8 ()
AND time_less (*)

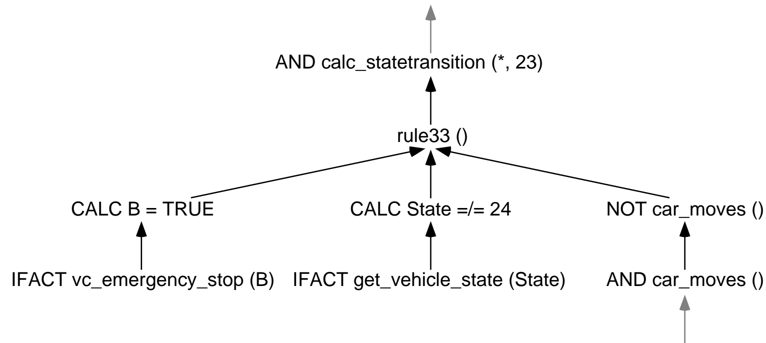

AND calc_statetransition (8, 1)
AND calc_statetransition (3, 1)
AND calc_statetransition (9, 1)
AND calc_statetransition (2, 1)
OR rule10 ()
AND rule10 ()
AND rule10 ()
AND rule10 ()
NOT left_changereq ()
CALC B = FALSE
CALC B = FALSE
AND left_changereq ()
IFACT ddb_LeftLaneExists (B)
IFACT dm_PassingAllowed (B)
AND calc_statetransition (9, 5)
AND calc_statetransition (8, 4)
AND calc_statetransition (3, 4)
AND rule11 ()
CALC B = TRUE
AND change_left_possible ()
AND left_changereq ()
IFACT ddb_LeftLaneExists (B)
CALC B = TRUE
AND time_less (*)
IFACT dm_PassingAllowed (B)
AND calc_statetransition (4, 5)
AND rule12 ()
AND change_left_possible ()
AND left_changereq ()
NOT time_less (*)
AND time_less (*)
AND calc_statetransition (4, 6)
AND calc_statetransition (5, 7)
AND rule13 ()
AND rule13 ()
AND rule14 ()
NOT change_left_possible ()
NOT left_changereq ()
CALC B = TRUE
AND change_left_possible ()
AND left_changereq ()
IFACT vc_chtll_ack (B)

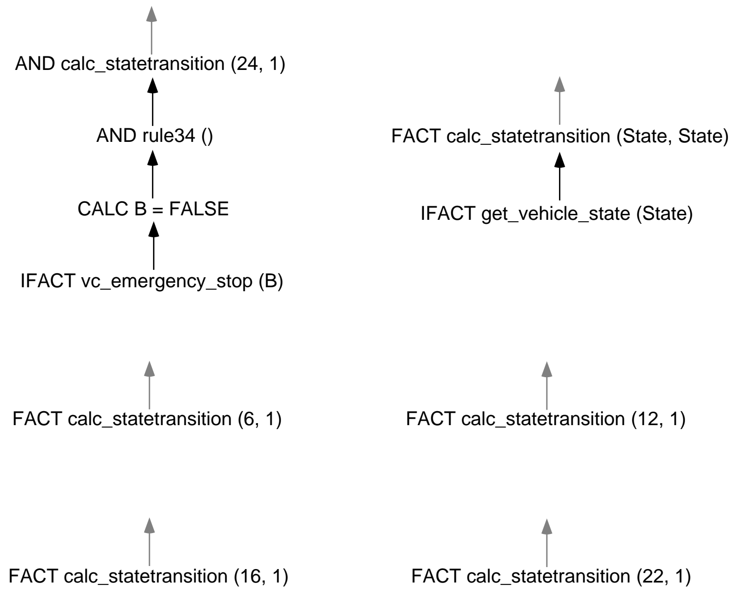

AND calc_statetransition (7, 6)
AND rule15 ()
CALC B = FALSE
CALC B = FALSE
NOT cancel_change_left ()
IFACT vc_lch_active (B)
IFACT vc_chtll_ack (B)
AND cancel_change_left ()
AND calc_statetransition (9, 8)
AND calc_statetransition (8, 9)
AND calc_statetransition (3, 8)
AND rule16 ()
CALC B = TRUE
NOT time_less (*)
IFACT ddb_LeftLaneExists (B)
AND time_less (*)
CALC B = TRUE
AND left_changereq ()
IFACT dm_PassingAllowed (B)
AND calc_statetransition (7, 10)
AND rule17 ()
AND cancel_change_left ()
AND calc_statetransition (10, 11)
AND rule18 ()
CALC B = FALSE
IFACT vc_chtll_ack (B)
AND calc_statetransition (11, 12)
AND rule19 ()
CALC B = FALSE
IFACT vc_lch_active (B)

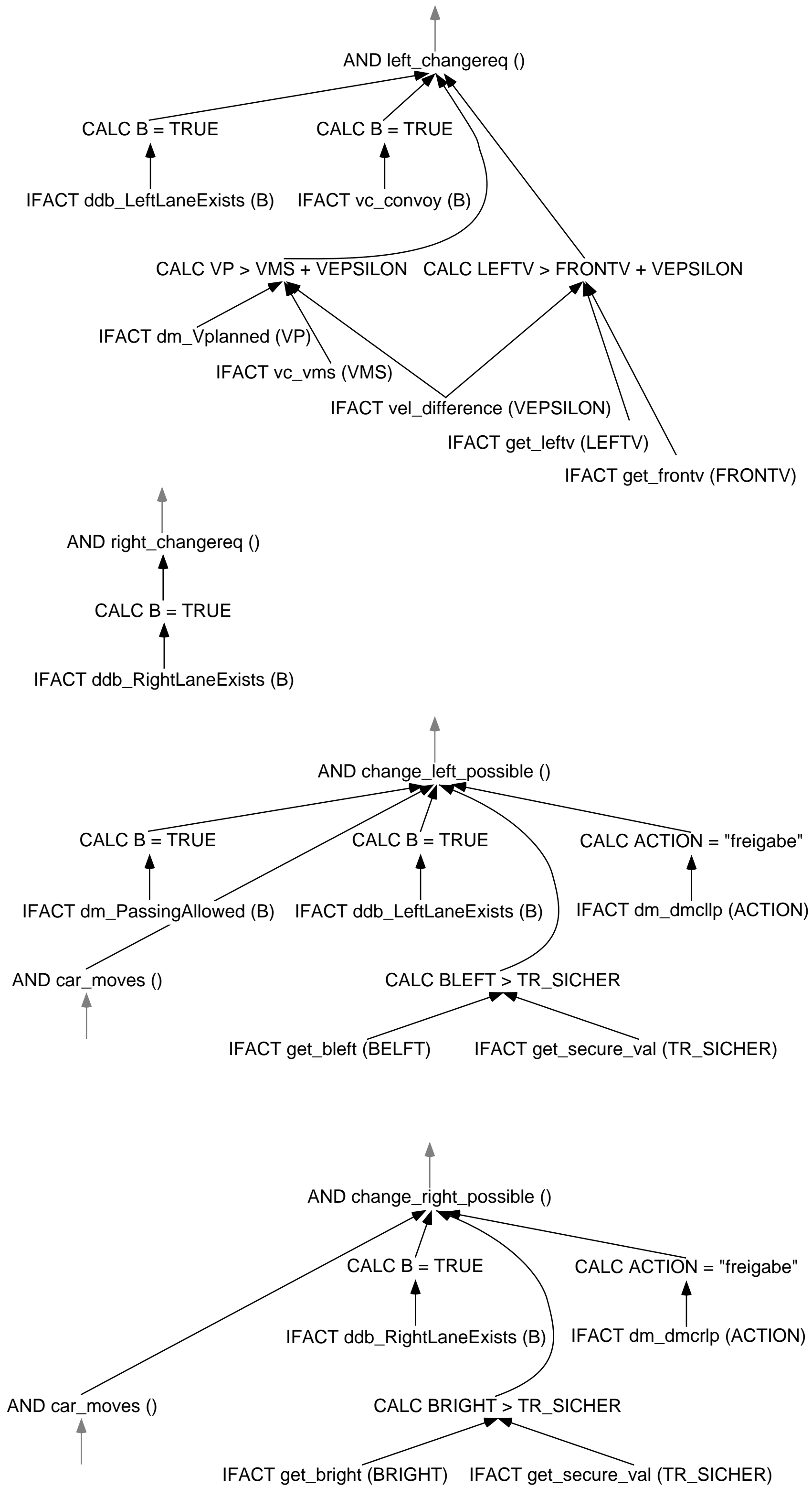

AND calc_statetransition (13, 1)
AND calc_statetransition (1, 13)
NOT rule8 ()
AND rule20 ()
AND rule8 ()
AND right_changereq ()
AND calc_statetransition (13, 1)
AND calc_statetransition (18, 1)
AND calc_statetransition (19, 1)
NOT rule8 ()
AND rule21 ()
OR rule21 ()
AND rule8 ()
AND rule21 ()
AND rule21 ()
NOT right_changereq ()
CALC B = FALSE
AND right_changereq ()
IFACT ddb_RightLaneExists (B)
AND calc_statetransition (13, 14)
NOT rule8 ()
AND rule22 ()
AND rule8 ()
AND calc_statetransition (18, 14)
AND calc_statetransition (19, 15)
AND rule22 ()
CALC B = TRUE
AND change_right_possible ()
AND right_changereq ()
AND time_less (*)
IFACT ddb_RightLaneExists (B)
AND calc_statetransition (14, 15)
AND rule23 ()
AND change_right_possible ()
AND right_changereq ()
NOT time_less (*)
AND time_less (*)

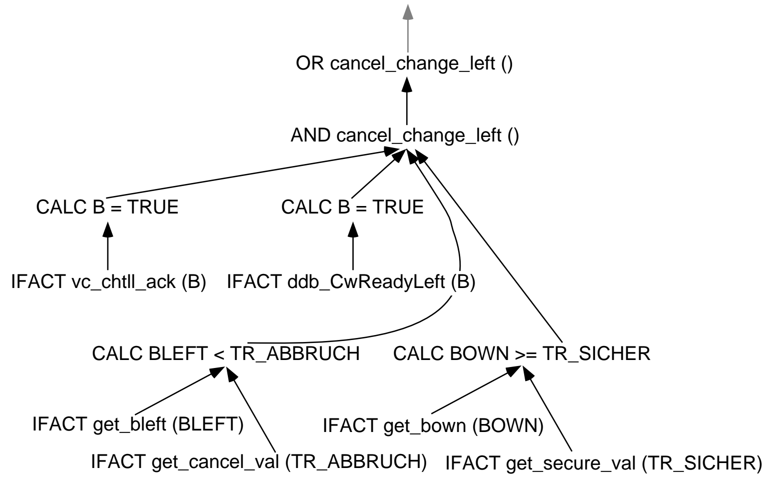

AND calc_statetransition (14, 16)
OR rule24 ()
AND rule24 ()
AND rule24 ()
NOT change_right_possible ()
NOT right_changereq ()
AND change_right_possible ()
AND right_changereq ()
AND calc_statetransition (15, 17)
AND rule25 ()
CALC B = TRUE
IFACT vc_chtrl_ack (B)
AND calc_statetransition (17, 16)
AND rule26 ()
CALC B = FALSE
CALC B = FALSE
NOT cancel_change_right ()
IFACT vc_lch_active (B)
IFACT vc_chtrl_ack (B)
AND cancel_change_right ()
AND calc_statetransition (13, 18)
NOT rule8 ()
AND rule27 ()
AND rule8 ()
AND calc_statetransition (18, 19)
AND calc_statetransition (19, 18)
AND rule27 ()
CALC B = TRUE
AND right_change_req ()
NOT time_less (*)
IFACT ddb_RightLaneExists (B)
AND time_less (*)
AND calc_statetransition (17, 20)
AND rule28 ()
AND cancel_change_right ()
AND calc_statetransition (20, 21)
AND rule29 ()
CALC B = FALSE
IFACT vc_chtrl_ack (B)
AND calc_statetransition (21, 22)
AND rule30 ()
CALC B = FALSE
IFACT vc_lch_active (B)

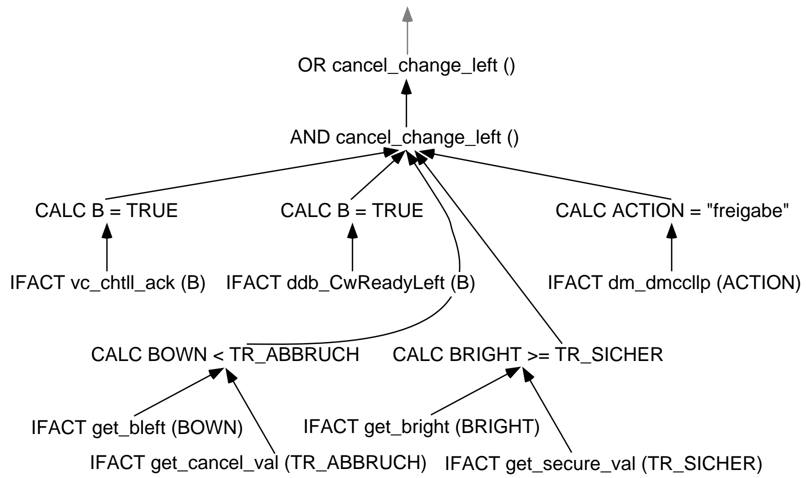

AND calc_statetransition (*, 24)
rule31 ()
CALC State =/= 23
CALC B = TRUE
IFACT get_vehicle_state (State)
IFACT vc_emergency_stop (B)
AND calc_statetransition (23, 1)
rule32 ()
AND car_moves ()

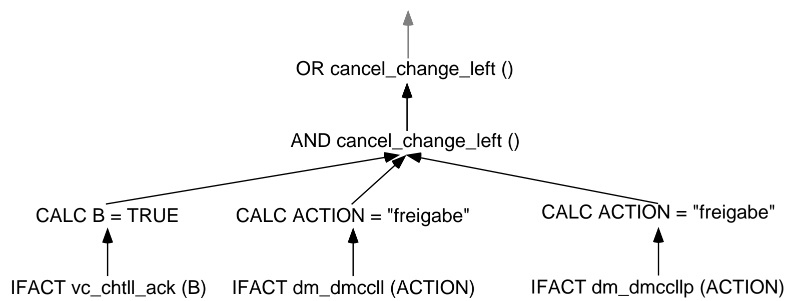

AND calc_statetransition (*, 23)
rule33 ()
CALC B = TRUE
CALC State =/= 24
NOT car_moves ()
IFACT vc_emergency_stop (B)
IFACT get_vehicle_state (State)
AND car_moves ()

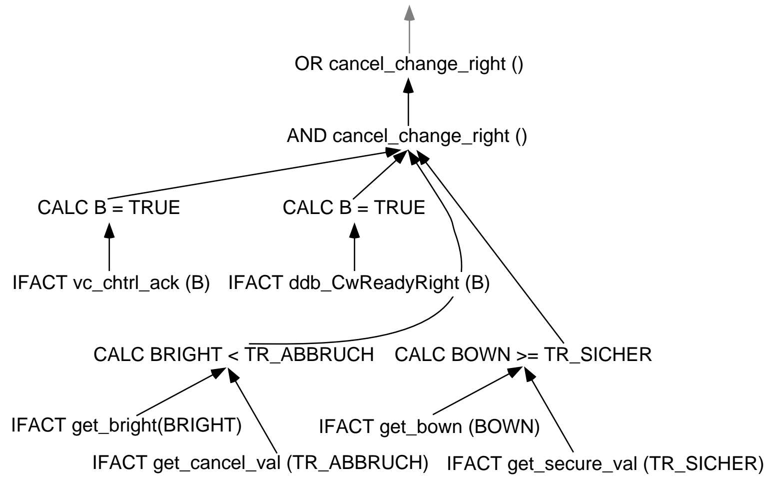

AND calc_statetransition (24, 1)
AND rule34 ()
CALC B = FALSE
IFACT vc_emergency_stop (B)
FACT calc_statetransition (State, State)
IFACT get_vehicle_state (State)
FACT calc_statetransition (6, 1)
FACT calc_statetransition (12, 1)
FACT calc_statetransition (16, 1)
FACT calc_statetransition (22, 1)

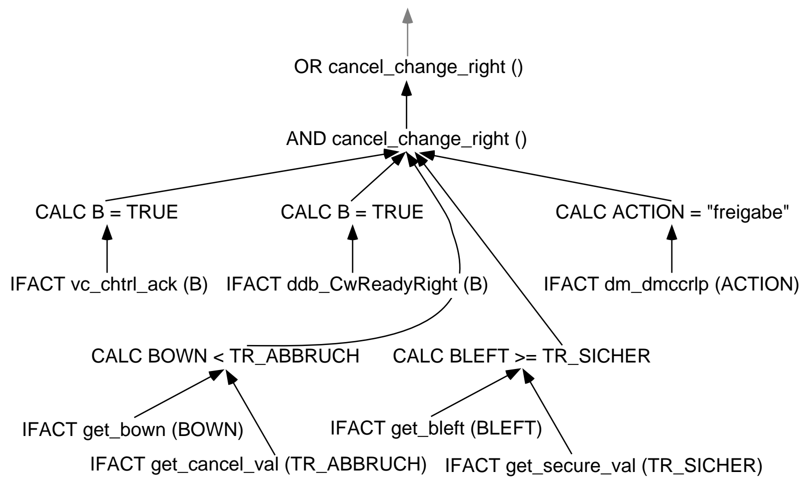

AND left_changereq ()
CALC B = TRUE
CALC B = TRUE
IFACT ddb_LeftLaneExists (B)
IFACT vc_convoy (B)
CALC VP > VMS + VEPSILON
CALC LEFTV > FRONTV + VEPSILON
IFACT dm_Vplanned (VP)
IFACT vc_vms (VMS)
IFACT vel_difference (VEPSILON)
IFACT get_leftv (LEFTV)
IFACT get_frontv (FRONTV)
AND right_changereq ()
CALC B = TRUE
IFACT ddb_RightLaneExists (B)
AND change_left_possible ()
CALC B = TRUE
CALC B = TRUE
CALC ACTION = "freigabe"
IFACT dm_PassingAllowed (B)
IFACT ddb_LeftLaneExists (B)
IFACT dm_dmcllp (ACTION)
AND car_moves ()
CALC BLEFT > TR_SICHER
IFACT get_bleft (BELFT)
IFACT get_secure_val (TR_SICHER)
AND change_right_possible ()
CALC B = TRUE
CALC ACTION = "freigabe"
IFACT ddb_RightLaneExists (B)
IFACT dm_dmcrlp (ACTION)
AND car_moves ()
CALC BRIGHT > TR_SICHER
IFACT get_bright (BRIGHT)
IFACT get_secure_val (TR_SICHER)

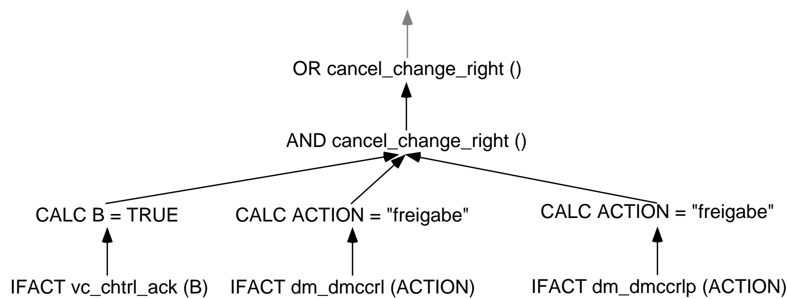

OR cancel_change_left ()
AND cancel_change_left ()
CALC B = TRUE
CALC B = TRUE
IFACT vc_chtll_ack (B)
IFACT ddb_CwReadyLeft (B)
CALC BLEFT < TR_ABBRUCH
CALC BOWN >= TR_SICHER
IFACT get_bleft (BLEFT)
IFACT get_bown (BOWN)
IFACT get_cancel_val (TR_ABBRUCH)
IFACT get_secure_val (TR_SICHER)


OR cancel_change_left ()
AND cancel_change_left ()
CALC B = TRUE
CALC B = TRUE
CALC ACTION = "freigabe"
IFACT vc_chtll_ack (B)
IFACT ddb_CwReadyLeft (B)
IFACT dm_dmccllp (ACTION)
CALC BOWN < TR_ABBRUCH
CALC BRIGHT >= TR_SICHER
IFACT get_bleft (BOWN)
IFACT get_bright (BRIGHT)
IFACT get_cancel_val (TR_ABBRUCH)
IFACT get_secure_val (TR_SICHER)


OR cancel_change_left ()
AND cancel_change_left ()
CALC B = TRUE
CALC ACTION = "freigabe"
CALC ACTION = "freigabe"
IFACT vc_chtll_ack (B)
IFACT dm_dmccll (ACTION)
IFACT dm_dmccllp (ACTION)


OR cancel_change_right ()
AND cancel_change_right ()
CALC B = TRUE
CALC B = TRUE
IFACT vc_chtrl_ack (B)
IFACT ddb_CwReadyRight (B)
CALC BRIGHT < TR_ABBRUCH
CALC BOWN >= TR_SICHER
IFACT get_bright(BRIGHT)
IFACT get_bown (BOWN)
IFACT get_cancel_val (TR_ABBRUCH)
IFACT get_secure_val (TR_SICHER)


OR cancel_change_right ()
AND cancel_change_right ()
CALC B = TRUE
CALC B = TRUE
CALC ACTION = "freigabe"
IFACT vc_chtrl_ack (B)
IFACT ddb_CwReadyRight (B)
IFACT dm_dmccrlp (ACTION)
CALC BOWN < TR_ABBRUCH
CALC BLEFT >= TR_SICHER
IFACT get_bown (BOWN)
IFACT get_bleft (BLEFT)
IFACT get_cancel_val (TR_ABBRUCH)
IFACT get_secure_val (TR_SICHER)


OR cancel_change_right ()
AND cancel_change_right ()
CALC B = TRUE
CALC ACTION = "freigabe"
CALC ACTION = "freigabe"
IFACT vc_chtrl_ack (B)
IFACT dm_dmccrl (ACTION)
IFACT dm_dmccrlp (ACTION)

Literatur

[Albus 91] J. S. Albus, “Outline for a Theory of Intelligence”, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol 21, 1991, pp. 473-509

[Altenkrüger 87] D. E. Altenkrüger, “Wissensdarstellung für Expertensysteme”, B.I. Wissenschaftsverlag, 1987

[Amarel 91] S. Amarel, “AI Research in DARPAs Strategic Computing Initiative”, *IE-EE expert*, No. 3, 1991, pp. 7-11

[Antsalkis 91] P. J. Antsaklis, K. M. Passino, S. J. Wang, “An Introduction to Autonomous Control Systems”, *IEEE Control Systems*, No. 4, 1991, pp. 5-13

[Apt 89] Krzysztof R. Apt, Roland N. Bol, Jan Willem Klop, “On the Safe Termination of PROLOG Programs”, Giorgio Levi, Maurizio Martelli (Eds.): *Logic Programming*, Proceedings of the Sixth International Conference, Lisbon, Portugal, June 19-23, 1989. MIT Press 1989,pp. 353-368

[Apt 91] Krzysztof R. Apt, Dino Pedreschi, “Proving Termination of General Prolog Programs”, Takayasu Ito, Albert R. Meyer (Eds.): *Theoretical Aspects of Computer Software*, International Conference TACS '91, Sendai, Japan, September 24-27, 1991, Proceedings. Lecture Notes in Computer Science, Vol. 526, Springer, 1991, pp. 265-289

[Aravindan 95] C. Aravindan, P. M. Dung, “On the Correctness of Unfold/Fold Transformation of Normal and Extended Logic Programs”, *Journal of Logic Programming*, Vol. 24, No. 3, 1995, pp. 201-217

[Arvind 78] Arvind, K. P. Gostelow, “Dataflow Computer Architecture: Research and Goals”, *UCI-ICS Report 113*, University of California, Irvine, ICS, 1978

[Arvind 80] Arvind, R. E. Thomas, “I-Structures: An efficient Data Type for Functional Languages”, *TM-178*, Laboratory for Computer Science, MIT, Cambridge, 1980

[Arvind 81] Arvind, V. Kathail, “A Multiple Processor Data Flow Machine that Supports Generalized Procedures”, *Proc. of the 18th International Symposium on Computer Architecture*, 1981, pp. 291-302

[Arvind 82] Arvind, K. P. Gostelow, “The U-Interpreter”, *IEEE Computer*, Vol. 15, 1982, pp. 42-49

[Arvind 83] Arvind, D. E. Culler, “The Tagged Token Dataflow Architecture”, *Internal Report*, Massachusetts Institute of Technology, 1983

[Arvind 89] Arvind, R. S. Nikhil, “I-Structures: Data Structures for Parallel Computing”, *ACM Transactions on Programming Languages and Systens*, Vol. 11, Nr. 4, 1989, pp. 598-632

[Arvind 91] Arvind, L. Bic, T. Ungerer, “Evolution of Data-Flow Computers” *Advanced Topics in Data-Flow Computing*, L. Bic, J. Gaudiot, Prentice Hall Verlag, 1991, pp. 3-33

[Aström 85] K. J. Aström, “Process Control - Past, Present and Future”, *IEEE Control Systems*, No. 3, 1985, pp. 3-10

[Bader 90] Peter Bader, “Auswertung von Logik-Programmen mit einem Message-Passing-Modell”, *Dissertation*, 1990

[Balbin 86] I. Balbin, K. Ramamohanarao, “A Differential Approach to Query Optimisation in Recursive Databases” Technical Report 86/7, University of Melbourne, Dep. of Computer Science, 1986

[Balbin 87] I. Balbin, K. Ramamohanarao, "A Generalization of the Differential Approach to Recursive Query Evaluation", *Journal of Logic Programming*, Vol. 4, Nr. 3, pp. 259-262, 1987

[Banks 91] S. B. Banks, C. S. Lizza, "Pilot's Associate - A Cooperative, Knowledge-Based System Application", *IEEE Expert*, No. 3, 1991, pp. 18-29

[Behringer 93] R. Behringer, "Straßenerkennung aus Bildfolgenanalyse mittels rekursiver Schätzfunktionen zur autonomen Führung eines Straßenfahrzeugs", *Dissertation*, Universität der Bundeswehr, München, Fakultät für Luft- und Raumfahrttechnik, 1993

[Bic, 84a] L. Bic, "Execution of Logic Programs on a Dataflow Architecture", *IEEE SIGARCH Newsletter*, Vol. 12, No. 3, 1984, pp. 290-296

[Bic 84b] L. Bic, "A Data-Driven Model for Parallel Interpretation of Logic Programs", *Proc. of the Int. Conference on fifth generation Computer Systems*, 1984, pp. 517-523

[Bic 87] L. Bic, C. Lee, "A Data-Driven Model for a Subset of Logic Programming", *ACM Transactions on Programming Languages and Systems*, Vol. 9, 1987, pp. 618-645

[Bic 89] L. Bic, R.L. Hartmann, "AGM: a dataflow database machine", *ACM Transactions on Database Systems*, Vol. 14, No.1, 1989, pp. 114-146

[Brass 95a] S. Brass, J. Dix, "Disjunctive semantics based opon partial and bottom-up evaluation", *logic Programming, Proc. of the twelfth International Conference on Logic Programming*, L. Sterling, MIT Press, 1995, pp. 199-213

[Brass 95b] S. Brass, J. Dix, "A general approach to bottom-up computation of disjunctive semantics", *Nonmonotonic Extensions of Logic Programming*, J. Dix, L.M. Pereira, T.C. Przymusinski, Springer Verlag, Vol. 927, 1995, pp. 127-155

[Brass 97a] S. Brass, B. Freitag, U. Zukowski, "Transformation-Based Bottom-Up Computation of the Well-Founded Model", *Lecture Notes in Artificial Intelligence: Non-Monotonic Extensions of Logic Programming* J. Dix, L. Pereira, T. Przymusinski, Springer-Verlag, Berlin, 1997, pp. 171-201

[Brass 97b] S. Brass, B. Freitag, U. Zukowski, "Improving the Alternating Fixpoint: The Transformation Approach" *Lecture Notes in Artificial Intelligence: Logic Programming and Nonmonotonic Reasoning*, J. Dix, U. Furbach, A. Nerode, Springer-Verlag, Berlin, 1997, pp. 40-59

[Bratko 87] I. Bratko, *Prolog, Programming for Artificial Intelligence*, Addison-Wesley Verlag, 1987

[Bhuyan 85] L.M. Bhuyan, "An Analysis of Processor-Memory Interconnection Networks", *IEEE Trans. on Comp.* Vol. 34, No. 3, pp.279-283, 1985.

[Brüdigam 93] C. Brüdigam, "Das Fahrzeugführungsmodul VC (Vehicle Control)", *Report*, Universität der Bundeswehr, München, Fakultät für Luft- und Raumfahrttechnik, 1993

[Brüdigam 94] C. Brüdigam, "Intelligente Fahrmanöver sehender autonomer Fahrzeuge in autobahnähnlicher Umgebung", *Dissertation*, Universität der Bundeswehr, München, Fakultät für Luft- und Raumfahrttechnik, 1994

[Bruynooghe 88] M. Bruynooghe, P. Weemeeuw, M. De Hondt, "On implementing logic programming languages on a dataflow architecture", *2nd European Symposium on Programming*, Lecture Notes on Computer Science, Springer Verlag, 1988, pp. 359-372

[Bültzingsloewen 88] G. Bltzingsloewen, K.R. Dittrich, C. Iochpe, R.-P. Liedke, P.C. Lockemann, M. Schryro, "KARDAMOM - A Dataflow Database Machine For Real-Time Applications", *SIGMOD Record* Vol. 17, No.1, 1988, pp.44-50

[Bültzingsloewen 89] G. Bltzingsloewen, K.R. Dittrich, C. Iochpe, R. Kramer R.-P. Liedke, P.C. Lockemann, M. Schryro, "Design and Implementation of KARDAMOM - A Set-oriented Data Flow Database Machine", *Sixth International Workshop on Database Machines*, Lecture Notes in Computer Science, Vol. 368, Springer Verlag, 1989, pp.18-33

[Buck 99] K. Buck, "Seriöser Roboterick", *Computer Zeitung*, Nr. 23, 1999

[Chen 91] W. Chen, "Declarative Specification and Evaluation of Database Updates" *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases* Lecture Notes in Computer Science, Vol. 566, Springer Verlag, 1991, pp. 147-166

[Chen 93] W. Chen, D.S. Warren, "Query-evaluation under the well founded semantics", *Proc. of the twelfth ACM AIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1993, pp. 168-179

[Chen 95] W. Chen, T. Swift, D.S. Warren, "Efficient top-down computation of queries under the well founded semantics", *Journal of Logic Programming*, Vo.24, Nr. 2, 1995, pp. 161-199

[Chen 96] W. Chen, D.S. Warren, "Tabled evaluation with delaying for general logic programs", *Journal of the ACM*, Vol. 43, Nr. 1, 1996, pp. 20-74

[Clark 84] K. L. Clark, S. Gregory, "PARLOG: Parallel Programming in Logic", *Research Report 84/4*, Departement of Computing, Imperial College of Science and Technology, England, 1984

[Cremers 94] A. B. Cremers, U. Griefhahn, R. Hinze, *Deduktive Datenbanken - Eine Einführung aus der Sicht der logischen Programmierung*, Vieweg-Verlag, 1994

[Culler 90] D. E. Culler, "The exlicit Token Store", *Journal of Parallel and Distributed Computing*, Vol. 10, 1990, pp. 289-307

[Davis 82] A. E. Davis, R. M. Keller, "Data Flow Program Graphs", *IEEE Computer*, Vol. 15, No. 2, 1982, pp. 26-41

[Debray 88] S. K. Debray, D. S. Warren, "Automatic Mode Inference for Logic Programs", *Journal of Logic Programming*, Vol. 5, No. 3, 1988, pp. 207-229

[Debray 89] S. K. Debray, "Static Inference of Modes and Data Dependencies in Logic Programs". *ACM Transactions on Programming Languages and Systems*, Vol. 11, No.3, 1989, pp. 418-450

[Delgado 92a] S. A. Delgado-Rannauro, "OR-Parallel Logic Computational Models", *Implementations of Distributed Prolog*, P. Kascuk, M. J. Wise, Wiley Verlag, 1992, pp. 5-26

[Delgado 92b] S. A. Delgado-Rannauro, "Restricted AND- and AND/OR-Parallel Logic Computational Models", *Implementations of Distributed Prolog*, P. Kascuk, M. J. Wise, Wiley Verlag, 1992, pp. 121-141

[Delgado 92c] S. A. Delgado-Rannauro, "Stream AND-Parallel Logic Computational Models", *Implementations of Distributed Prolog*, P. Kascuk, M. J. Wise, Wiley Verlag, 1992, pp. 239-257

[Dennis 75] J. B. Dennis, D. P. Misunas, "A Preliminary Architecture for a Basic Data-Flow Processor", *2nd Annual Symposium on Computer Architecture*, 1975, pp. 126-132

[Dennis 80] J. B. Dennis, "Data Flow Supercomputers", *IEEE Computer*, Vol. 13, No. 2, 1980, pp. 48-56

[DeWitt 82] D.J.DeWitt, H. Boral, "Applying Dataflow Techniques to Database Machines", *IEEE Computer*, Vol. 15, 1982, pp.57 -63

[DeWitt 82] D.J.DeWitt, R.H. Gerber, G. Graefe, M.L. Heytens, K.B. Kumar, M. Muralikrishna, "GAMMA - A High Performance Dataflow Database", *Twelfth International Conference on Very Large Data Bases*, 1986, pp.228-237

[Dickmanns 91] E. D. Dickmanns, "System Architecture for Road Vehicles Capable of Vision", *Report*, Universität der Bundeswehr, München, Fakultät für Luft- und Raumfahrttechnik, 1991

[Dickmanns 92] E. D. Dickmanns, "Automobile lernen sehen", *Report*, Universität der Bundeswehr, München, Fakultät für Luft- und Raumfahrttechnik, 1992

[Dickmanns 93a] E. D. Dickmanns, "The Dynamic Data Base User's Guide", *Interner Bericht 93/03*, PROMETHEUS PRO-ART, Informatik-Forschungsgruppe, Universität der Bundeswehr, München, 1993

[Dickmanns 93b] E. D. Dickmanns, "KRONOS-Benutzerhandbuch", *Interner Bericht 93/01*, PROMETHEUS PRO-ART, Informatik-Forschungsgruppe, Universität der Bundeswehr, München, 1993

[Dickmanns 95] E. D. Dickmanns, "Performance Improvements for Autonomous Road Vehicles", *IAS4, Conference Proceedings*, Karlsruhe 1995, pp.2-14

[Dickmanns 97] E. D. Dickmanns, "Vehicles capable of dynamic vision", *Procedings of the 15th International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997

[Dodd 90] T. Dodd, *Prolog: A logical approach*, Oxford Science Publications, 1990

[Dubois 93] D. Dubois, H. Prade, R.R., Yager, "Introduction to Fuzzy Sets for Intelligent Systems", *Fuzzy Sets for Intelligent Systems*, D. Dubois, H. Prade, R.R, Yager, Morgan Kaufmann Publisher, Inc., 1993

[Emden 76] M. H. van Emden, R. Kowalski, "The semantics of predicate logic as a programming language", *Journal of the ACM*, Vol. 23, No. 4, 1976

[Fagg 93] A.H. Fagg, M.A. Lewis, J.F. Montgomery, G.A. Bekey, "The USC Autonomous Flying Vehicle: an experiment in real-time behaviour-based Control", *Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1993, pp.1173-1180

[Fischer 86] K. Fischer, T. Kämpke, "A Contribution to the Performance Evaluation of the Music Eventflow Computer Architecture", MIP–Bericht Universität Passau, MIP-8615, 1986.

[Gardner 91] P. A. Gardner, J. C. Sheperdson, "Unfold/Fold transformations of logic programs", *Computational Logic*, J.-L. Lassez, G. Plotkin, MIT Press, 1991, pp. 565-583

[Garey 79] M. R. Garey, *Computers and intractability: a guide to the theory of NP-completeness*, New York, 1979

[Gaudiot 86] J.-L. Gaudiot, "Structure Handling in Data-Flow Systems", *IEEE Transactions on Computers*, Vol. 35, No. 6, 19986, pp. 849-502

[Gelder 87] A. Van Gelder, "Efficient Loop Detection in Prolog using the Tortoise-and-Hare Technique" *Journal of Logic Programming*, Vol. 4, 1987, pp. 23-31

[Gelder 88] A. Van Gelder, K. Ross, J.S. Schlipf, "Unfounded sets and well-founded semantics for general logic programs", *Proc. of the seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1988, pp. 221-230

[Gelder 89] A. Van Gelder, "The alternating fixpoint of logic programs with negation", *Journal of Computer and System Sciences*, Vol. 47, 1993, pp. 185-221,

[Gelder 91] A. Van Gelder, K. Ross, J.S. Schlipf, "The well-founded semantics for general logic programs", *Journal of the ACM*, Vol. 38, 1991 pp. 620-650

[Gelder 93] A. Van Gelder, "The alternating fixpoint of logic programs with negation", *Journal of Computer and System Sciences*, Vol 47, No. 1, 1993, pp. 185-221

[Gelfond 88] M. Gelfond, V. Lifschitz, "The stable model semantics for logic programming", *Logic programming, Proc. of the fifth International Conference and Symposium*, R.A. Kowalski, K.A. Bowen, MIT Press, 1988, pp. 1070-1080

[Gregory 87] S. Gregory, *Parallel Logic Programming in PARLOG*, Addison-Wesley Verlag, 1987

[Güntzer 87] U. Güntzer, W. Kießling, R. Bayer, "On the Evaluation of Recursion in (Deductive) Database Systems by Efficient Differential Fixpoint Iteration" *Proceedings of the Third International Conference on Data Engineering* pp. 120-129, 1987

[Gupta 95] R.K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, Norwell, Massachusetts, 1995.

[Gurd 80] J. R. Gurd, C. C. Kirkham, I. Watson, "The Manchester Prototype Dataflow Computer", *Communications of the ACM*, Vol. 28, No. 1, 1980

[Haas 99] J. Haas, "Echtzeit-Korrespondenzprobleme in Bildsequenzen und Subpixelgenauigkeit" - noch zu veröffentlichende Dissertation -, Uni Passau, ?

[Hagerer 91] A. Hagerer, A.S. Lang, "An Approach for Balancing the Workload of the Munich Simulation Computer", *Advances in Parallel and Distributed Simulation – Proceedings of the SCS Multiconference in Parallel and Distributed Simulation (Anaheim, California)*, 1991.

[Hahn 85] W. Hahn, K. Fischer, "Music - an Event-Flow Computer for fast Digital Design Simulation", *22nd IEEE/ACM Design Automation Conference*, 1985, pp. 338-344

[Hahn 85] W. Hahn, "MuSiC: A Simulation Engine based on Event–Flow", *European Simulation Multiconference*, 1987, pp 346-354.

[Hahn 92] W. Hahn, A. Hagerer, H. Anger, "Update-Dataflow Computing: A Way to Supercomputing in Discrete Simulation", *Volume I of Software Architecture*, Elsevier Science Publishers B. V. (North-Holland), 1992, pp. 509-517

[Halim, 84] Z. Halim, I. Watson, "An OR-Parallel Data-driven Model for Logic Programs", *Proc. of International Workshop on High-Level Computer Architecture*, 1984, pp. 26-36

[Halim 86] Z. Halim, "A Data Driven Machine for OR-Parallel Evaluation of Logic Programs", *New Generation Computing*, 1986, pp. 5-33

[Hasegawa 84] R. Hasegawa, M. Amamiya, "Parallel Execution of Logic Programs based on Dataflow Concept", *International Conference on Fifth Generation Computer Systems*, 1984

[Hasegawa 85] R. Hasegawa, M. Amamiya, "An Architecture for List-Processing Oriented Dataflow Machine", *REVIEW of the Electrical Communication Laboratories*, Vol. 32, No. 5, 1985

[Hiraki 91] K. Hiraki, S. Skiguchi, T. Shimada, "Status Report of SIGMA-1 Dataflow Supercomputer", *Advanced Topics in Dataflow Computing*, J.-L. Gaudiot, L. Bic, Prentice-Hall, 1991

[Hiro 93] K. Hirota, Y. Arai, S. Hachisu, "Fuzzy Controlled Robot Arm playing tow-dimensional Ping-Pong Game", *Fuzzy Sets for Intelligent Systems*, D. Dubois, H. Prade, R.R, Yager, Morgan Kaufmann Publisher, Inc., 1993

[Holt 93] V. von Holt, "System- und Kommunikationsstruktur im VITA-Transputersystem des ISF/UniBwM", *Report* Universität der Bundeswehr, München, Fakultät für Informatik, 1993.

[Ito 85] N. Ito, "Data-flow Based Execution Mechanism of Parallel and Concurrent Prolog", *New Generation Computing*, No. 3, 1985, pp. 15-41

[Ito 86] N. Ito, "The Architecture and Preliminary Evaluation Results of the Experimental Parallel Inference Machine PIM-D", *13th Annual Symposium on Computer Architecture*, 1986

[Kacsuk 90] P. Kascuk, *Execution Models of Prolog for Parallel Computers*, MIT Press, Cambridge, Massachusetts, 1990

[Kacsuk 91] P. Kascuk, "A Parallel Prolog Abstract Machine and its Multi-Transputer Implementation", *The Computer Journal*, Vol. 34, No. 1, 1991, pp 52-63

[Kacsuk 92] P. Kascuk, "Distributed Data Driven Prolog Abstract Machine", *Implementations of Distributed Prolog*, P. Kascuk, M. J. Wise, Wiley Verlag, 1992, pp. 89-118

[Kanamori 87] T. Kanamori, K. Horiuchi, "Construction of Logic Programs on Generalized Unfold/Fold Rules", *4th International Conference on Logic Programming*, pp. 744-768,1987

[Kemp 89] D.B. Kemp, K. Ramamohanarao, I. Balbin, K. Meenashki, "Propagating Constraints in recursive deductive databases", *Proceedings of the North American Conference on Logic Programming*, 1989, pp. 16-20

[Kemp 95] D.B. Kemp, D. Srivastava, P.J. Stuckey, "Bottom-up evaluation and query optimization of well-founded models", *Theoretical Computer Science*, Vol 146, 1995, pp. 145-184

[Kifer 86] M. Kifer, E. L. Lozinski: "Filtering Data Flow in Deductive Databases" *International Conference on Database Theory*, Lecture Notes in Computer Science, Vol. 243, 1986, pp. 186-202

[Knauss 87] W. Knaus, *Turbo-Prolog: Grundlagen, Programmiertechniken, Anwendungen*, Hanser-Verlag, 1987

[Krantz 71] D.H. Krantz, D.D. Luce, P. Suppes, A. Tversky, *Foundations of Measurement - Vol. I: Additive and Polynominal Representations*, Academiuc Press, New York, 1971

[Kreutzer 91] W. Kreutzer, B. McKenzie, *Programming for Artificial Intelligence, Methods, Tools and Applications*, Adison-Wesley Verlag, 1991

[Kujawski 93] C. Kujawski, “Der Modul Behaviour Decision”, *Report* Universität der Bundeswehr, München, Fakultät für Informatik, 1993

[Lee 72] R.C.T. Lee, “Fuzzy Logic and the Resolution Principle”, *Journal of the ACM*, Vol. 19, No. 1, 1972, pp. 109-119

[Lloyd 87] J. W. Lloyd, *Foundations of Logic Programming*, Second, Extended Edition, Springer-Verlag, 1987

[Mann 96] H. Mann, H. Schiffelgen, R. Froriep, *Einführung in die Regelungstechnik*, 7. Auflage, Hanser-Lehrbuch, 1996

[Micheli 97] G. de Micheli, R.K. Gupta, “Hardware/Software Co–Design”, *Proc. IEEE, Vol. 85, No. 3, March 1997*

[Montesi 97] D. Montesi, E. Bertino, M. Martelli, “Transactions and Updates in Deductive Databases” *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 5, pp. 784-797 1997

[Müller 96] M. Mller, “Schnelle Circuit-Level-Simulation auf einer Ereignisfluarchitektur”, *Disseration*, Universitt Passau, 1996

[Müller 95] N. Müller, S. Baten, “Image Processing Based Navigation with an Autonomous Car”, *IAS4, Conference Proceedings*, 1995, pp. 591-598

[Mündemann 92] K. Mündemann, “Programm-Modul DIALOGUE_MANAGER, Preliminary Version (Direct Communication)”, *Interner Bericht 9271*, PROMETHEUS PRO-ART, Informatik-Forschungsgruppe, Universität der Bundeswehr, München, 1992

[Mündemann 93] K. Mündemann, “Module zu Missionsplanung und Situationsdarstellung”, *Interner Bericht 9271*, PROMETHEUS PRO-ART, Informatik-Forschungsgruppe, Universität der Bundeswehr, München, 1993

[Mukaidono 93] M. Mukaidono, Z. Shen, L. Ding, “Fundamentals of Fuzzy Prolog”, *Fuzzy Sets for Intelligent Systems*, D. Dubois, H. Prade, R.R, Yager, Morgan Kaufmann Publisher, Inc., 1993

[Olin 91] K. E. Olin, D. Y. Tseng, “Autonomous Cross Country Navigation (An Integrated Perception and Planning System)”, *IEEE expert*, No. 4, 1991, pp. 16-30

[Papadopoulos 90] G. M. Papdopoulos, D. E. Culller, “Monsoon: an Explicit Token Store Dataflow Architecture”, *Proceedings of the 17th Int. Symp. on Computer Architecture*, Seattle, 1990

[Papadopoulos 91] G. M. Papadopoulos, K. R. Traub, “Multithreading: A Revisionist View of Dataflow Architectures”, *Proceedings of the 18th Int. Symp. on Computer Architecture*, Toronto, 1991

[Parameswaran 93] M.F. Parkinson, P.M. Taylor, S. Parameswaran, “An Automated Hardware/Software Codesign (HSC) using VHDL”, *Procceedings First Asia Pacific Conference on Hardware Description Languages and their Applications (APCHDLSA '93)*, December 1993.

[Paulin 89] P.G. Paulin, J.P. Knight, “Force Directed Scheduling for the behavioral synthesis of ASICs,” *IEEE Transactions on Computer Aided Design*, Vol.8, pp. 661-679, 1989.

[Pelov 99] N. Pelov, M. Bruynooghe, “Proving Failure of Queries for Definite Logic Programs using XSB-Prolog” *Lecture Notes in Artificial Intelligens*, H. Ganzinger, D. McAllester, A. Voronkov No. 1705, 1999, pp.358-375

[Ramakrishnan 88] R. Ramakrishnan, “Magic Templates: A Spellbinding Approach to Logic Programs” *Proceedings of the International Conference on Logic Programming*, Seattle, USA, 1988, pp. 140-159

[Ramakrishnan 90] R. Ramakrishnan, I. S. Mumik, S. J. Finkelstein, H. Pirahesh, “Magic Conditions”, *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, Nashville, USA, 1990, pp.314-330

[Ramakrishnan 95] R. Ramakrishnan, P. Rao, K. F. Sagonas, T. Swift, D. S. Warren, “Efficient Tabling Mechanisms for Logic Programs”, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming*, Leon Sterling, Tokyo,Japan, MIT Press, 1995, pp. 697-711

[Ross 94] K.A. Ross, “Modular Stratification and Magic Sets for Datalog Programs with Negation”, *Journal of the ACM*, Vol. 41, No. 6, 1994, pp. 1216-1266

[Sagonas 94] K. Sagonas, T. Swift, D.S. Warren, “XSB as an efficient deductive database engine” *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data*, R.T. Snodgrass, M. Winslett, 1994, pp. 442-453

[Sastry 88] A.V.S. Sastry, L.M. Patnaik, “A Dataflow Architecture for Parallel Execution of Logic Programs”, *International Conference on Parallel Processing*, 1988

[Sastry 91] A.V.S. Sastry, L.M. Patnaik, “OR-Parallel Evaluation of Logic Programs on a Multi-Ring Dataflow Machine”, *New Generation Computing*, Vol. 10, No. 1, pp.23-54, 1991

[Schreiner 95] F. Schreiner, R. Onken, “Fahrerunterstutzungssystem DAISY/LKW”, *PROMETHEUS PRO-DRIVER, Abschludokumentation*, Mnchen, 1995

[Schütz 93] H. Schütz, “Tupelweise Bottom-up-Auswertung von Logikprogrammen”, *Dissertation*, TU-München, 1993

[Seki 93] H. Seki, “Unfold/Fold Transformation of General Logic Programs for the Well-Founded Semantics”, *Journal of Logic Programming*, Vol. 16, No. 1, 1993, pp. 5-23

[Shapiro 83] E. Shapiro, “A Subset of Concurrent Prolog and its Interpreter”, *Tech. Report TR-003*, IOCT-Institute for New Generation, Computer Technology, Tokyo, Japan, 1983.

[Shapiro 86] E. Shapiro, “Concurrent Prolog: A progress report”, *Fundamentals of Artificial Intelligence*, W. Bibel and Ph. Jorrand, Springer Verlag, 1986, pp. 277-313

[Shimada 86] T. Shimada, K. Hiraki, S. Sekiguchi, K. Nishada, “Evaluation of a Single Processor of a Prototype Data Flow Computer SIGMA-1 for Sientific Computations”, *Proc. of the 13th Int. Symp. on Computer Architecture*, 1986, pp. 226-234

[Silberschatz 86] A. Silberschatz, H. Korth, *Database System Concepts*, McGraw-Hill, 1986

[Somogyim 96] Z. Somogyim F. Henderson, T. Conway, “The Execution Algorithm of Mercury, an efficient purely declarative Logic Programming Language”, *Journal of Logic Programming*, Vol. 29, No. 1, 1996 pp. 17 - 64

[Spruit, 95] P. Spruit, R. Wieringa, J.-J. Meyer, “Aziomatization, Declarative Semantics and Operational Semantics of Passive and Active Updates in Logic Databases” Journal of Logic and Computation Vol. 5 No. 1, pp. 27-70, 1995

[Srivastava 93] D. Srivastava, R. Ramakrishnan, “Pushing constraint selections”, *Journal of Logic Programming*, Vol. 16, No.3-4, 1993, pp.361-414

[Sterling 86] L. Sterling, E. Shapiro, *The Art of Prolog, Advanced Programming Techniques*, Massachusetts Institute of Technology, MIT Press, 1986

[Tamaki 84] H. Tamaki, T. Sato, "Unfold/Fold Transformations for Logic Programs", *2nd International Conference of Logic Programming*, 1984, pp. 127-138

[Thomanek 93] F. Thomanek, "Hinderniserkennung und Verfolgung", *Report* Universität der Bundeswehr, München, Fakultät für Informatik, 1993

[Thorpe 91a] C. Thorpe, M. Herbert, T. Kanade, et. al., "Toward Autonomous Driving: The CMU Navlab (Part I - Perception)", *IEEE expert*, No. 4, 1991, pp. 31-42

[Thorpe 91b] C. Thorpe, M. Herbert, T. Kanade, et. al., "Toward Autonomous Driving: The CMU Navlab (Part I - Perception)", *IEEE expert*, No. 4, 1991, pp. 44-52

[Thulasirman 92] K. Thulasirman, *Graphs: Theory and Algorithms*, Wiley-Intersicence Publication, 1992

[Treleaven 82a] P.C. Treleaven, D. R. Brownbridge, R. P. Hopkins, "Data Driven and Demand Driven Computer Architecture", *Computing Surveys*, 1982, pp. 95-143

[Treleaven 82b] P. C. Treleaven, R. P. Hopkins, P. W. Rautenbach, "Compining Data Flow and Control Flow Computing", *Computer Journal*, Vol. 25, No. 2, 1982, pp. 207-217

[Tseng 88a] C.-C. Tseng, P. Biswas, "A Data-Driven Parallel Execution Model for Logic Programs." *Logic Programming, Proceedings of the Fifth International Conference and Symposium*, R.A. Kowalski, K.A. Bowen, 1988, pp. 1204-1222

[Tseng 88b] C.-C. Tseng, P. Biswas, "A Data-Driven Abstract Machine Model for Parallel Execution of Logic Programs", *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1988, pp. 1059-1070

[Ueda 86] K. Ueda, "Guarded Horn Clauses", *Ph.D. Thesis*, University of Tokyo, 1986

[Umeyama 83] S. Umeyama, K. Tamura, "A Parallel Execution Model of Logic Programs", *Proc. of the 10th Int. Symp. on Computer Architecture*, 1983, pp. 349-355

[Ullman 85] J. D. Ullman, "Implementation of logical query languages for databases" *ACM Transactions on Database Systems*, Vol. 10, No. 3, 1985, pp. 289-321

[Ullman 88] J. D. Ullman, *Principles of Database and Knowledge-Base Systems*, Volume I and II, Computer Science Press, 1988

[Ungerer 93] T. Ungerer, *Datenflußrechner*, B.G. Teubner Stuttgart, 1993

[Veen 86] A. H. Veen, *Dataflow Machine Architecture*, *ACM Computing Surveys*, Vol. 18, No. 4, 1986, pp. 365-396

[Vossen 91] G. Vossen, *Datamodels, Database Languages and Database Management Systems*, Adisson-Wesley Verlag, 1991

[Warren 84] D. S. Warren, "Database Updates in Pure Prolog", *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, 1984, pp. 244-253

[Werner 94] S. Werner, S. Kraus "Landeanflug mit maschinellem Sehen", *Forschungsbericht FB 94-1*, Institut für Systemdynamik und Flugmechanik, Fakultät für Luft- und Raumfahrttechnik, Universität der Bundeswehr Muenchen, 1994.

[Werner 97] S. Werner, "Maschinelle Wahrnehmung für den bordautonomen automatischen Hubschauberflug", *Ph.D. Thesis*, Institut für Systemdynamik und Flugmechanik, Universität der Bundeswehr München, Fakultät für Luft- und Raumfahrttechnik, 1997.

[Winslett 88] M. Winslett, "A Model-Based Approach to Updating Databases with Incomplete Information" *ACM Transactions on Database Systems*, Vol. 13, 1988, pp. 167-196

[Wise 82] M. J. Wise, "A Parallel Prolog: The Construction of a Data Driven Model", *ACM Conference on LISP and Functional Programming*, 1982, pp. 56-67

[Wise 86] M. J. Wise, *Prolog Multiprocessors*, Prentice-Hall, 1986

[Yang 87] R. Yang, *P-Prolog: A parallel programming languge*, World Scientific, 1987

[Yasu 93] S. Yasunobu, S. Miamoto, H. Ihara, "Fuzzy Control for Automatic Train Operation Systems", *Fuzzy Sets for Intelligent Systems*, D. Dubois, H. Prade, R.R, Yager, Morgan Kaufmann Publisher, Inc., 1993

[Zapp 88] A. Zapp, "Automatische Straßenfahrzeugführung durch Rechnersehen" *Report*, Universität der Bundeswehr, München, Fakultät für Luft- und Raumfahrttechnik, 1988

[Zukowski 96] U. Zukowski, B. Freitag, "Adding Flexibility to Query Evaluation for Modularly Stratified Databases", *Proceedings of the International Conference and Symposium on Logic Programming*, 1996, pp.304-419

[Zukowski 97] U. Zukowski, B. Freitag, "The deductive database system LOLA", *Lecture Notes in Artificial Intelligence: Logic Programming and Nonmonotonic Reasoning*, J. Dix, U. Furbach, A. Nerode, Springer Verlag, Berlin, 1997, pp. 375-386