Parallelisierung eines Wissensrepräsentationssystems

5.3.1995

Frank Bergmann

Rogowski Institut für Elektrotechnik, RWTH-Aachen

am Projekt KIT-VM11, FR. 5-12, Technische Universität Berlin,
Franklinstr. 28/29, D-10857 Berlin, Deutschland

Hiermit versichere ich, daß ich die vorliegende Arbeit selbständig verfaßt habe. Es wurden keine anderen Hilfsmittel außer der angegebenen Literatur verwendet.

Diese Arbeit ist nur zum internen Gebrauch bestimmt. Alle Urheberrechte liegen beim Rogowski Institut für Elektrotechnik der RWTH-Aachen und dem Lehrstuhl für Funktionales und Logisches Programmieren der Technische Universität Berlin.
Für den Inhalt wird keine Gewähr übernommen.

Zusammenfassung

Die Fortschritte in der Entwicklung von beschreibungslogischen Systemen (DL) und die zunehmende Verbreitung von massiv parallelen Computer legen den Wunsch der Vereinigung dieser beider Gebiete nahe. In dieser Arbeit analysiere ich die Anwendbarkeit von Parallelismus auf DL-Systeme, und implementiere zwei Versionen des parallelen FLEX-Systems auf Basis der Transputer-Technik und vergleiche die Meßergebnisse.

Das Ergebnis ist, daß eine *distributed objects*-Implementierung sehr gute theoretische und praktische Resultate erzielt, die darauf hoffen lassen, daß auf diese Weise einige praktische Anwendungen der DL in den Bereich der interaktiven Verarbeitung gelangen.

Abstract

Today knowledge representation systems and massive parallel computers have each reached a level of maturity which makes their real world use possible. This paper presents one possible approach to unify both areas to achieve a performance boost for the traditionally resource consuming knowledge representation systems. In this paper I consider computers based on the well established Transputer technology to parallelize the FLEX description logic systems, which belongs to the group of KL-ONE based languages. In contrast to the mainstream of parallel logic research we use coarse-grain data parallelism for our implementation with the perspective to achieve high efficiencies on the given hardware architecture.

Schreibweise

Da sich die Sprache der Computertechnik sehr stark an der angloamerikanischen Begriflichkeit orientiert, würde es zu schwer lesbaren Formulierungen führen, versuchte man alle englischen Begriffe ins Deutsche zu übersetzten. Deshalb schlage ich hier den Weg ein, deutsche Begriffe nur dort zu benutzen, wo sie in den Sprachgebrauch eingegangen sind, und benutze sonst die englischen Formulierungen in Kursiv- und Kleinschreibung. Eine Reihe von oft vorkommenden englischen Begriffen benutze ich in einer eingedeutschten Schreibweise und flektiere diese Wörter auch deutsch, wie z.B. 'Compiler, 'Master' oder 'Worker'.

In den letzten Kapiteln benutze ich für die Beschreibung der Programmstruktur Prolog-Prädikate, bei denen ein '+' vor einem Argument eine Eingabe-Variable kennzeichnet, während ein '-' einen Rückgabewert kennzeichnet.

Inhaltsverzeichnis

1. Einleitung 1

2. Parallele Systeme 4

2.1. Parallele Hardware (SISD/MIMD/SIMD) 4

2.2. Parallele Sprachen 6

2.3. Parallele Ausdrucksmittel 7

2.4. Kommunikationsstrukturen 8

2.5. Gütekriterien für parallele Systeme 9

2.6. Grenzen der Parallelität 10

3. Beschreibungslogik (Description Logic) 12

3.1. Formale Spezifikation 13

3.2. Subsumptionstest 14

3.3. Klassifikation 15

3.4. Propagierung 15

3.5. Situationen 16

4. Paralleles Potential und Datenfluß in DL-Systemen 18

4.1. Datenfluß auf Term-Einführungsebene 19

4.2. Datenfluß auf Operationsebene 22

4.2.1. Subsumption 22

4.2.2. Klassifikation 22

4.2.3. Normalisierung 23

4.2.4. Propagierung 24

4.3. Mikroskopischer Datenfluß 28

4.4. Datenfluß von tableauxbasierten DL-Systemen 28

4.5. DL-Datenfluß Zusammenfassung 29

5. Parallelisierungsstrategien für DL 30

5.1. Farm Parallelismus 31

5.2. Distributed Objects-Parallelismus 33

5.3. Alternativen 33

6. Implementierung 35

6.1. Hardware 35

6.2. Brain Aid Prolog 36

6.3. Implementierung der Farm 39

6.3.1. Verarbeitungsnetz....42
6.3.2. Lastüberwachungsnetz:....42
6.4. Implementierung von Distributed Objects-FLEX....43
6.5. Schlußbemerkungen zur Implementation....46

7. Meßergebnisse und Interpretation....48
7.1. Benchmarks....48
7.2. Die Farm Implementierung....49
7.3. Die Distributed Objects-Implementierung....50
7.3.1. Skalierbarkeit....54
7.4. Vergleich mit anderen forwärtsverkettenden Systemen....58

8. Zusammenfassung und Ausblick....59

9. Bibliographie....60

Anhang....67
Anhang A: Trace einer NLP-Anwendung....67
Anhang B: Der N-Queens Benchmark für BAP....70
Anhang C: Die Implementierung des Farm-Schedulers....72
Anhang D: Die Implementierung des DO-Schedulers....76
Anhang E: Liste der FLEX-Module....82

Abbildungsverzeichnis

Bild 1.1: Aufbau dieser Diplomarbeit. 3
Bild 2.1: Steuerwerk/Verarbeitungswerk Kombinationen. 5
Bild 2.2. Imperative Parallele Sprachen. 7
Bild 2.3. Deklarative Parallele Sprachen. 7
Bild 2.4: Parallele Ausdrucksmittel in einigen Programmiersprachen. 8
Bild 2.5: Amdahls Gesetz, Formeln und Anwendung auf ein 100 Prozessor Netzwerk. 11
Bild 3.1: Ein Situationsbaum. 16
Bild 4.1: DL-Systemzustand und Interaktion. 18
Bild 4.2: DL-System Zustand. 18
Bild 4.3: Ablaufschema einer Objekt-Termeinführung. 19
Bild 4.4: Ablaufschema einer Konzept-Termeinführung. 20
Bild 4.5: Ablaufschema einer Regel-Einführung. 20
Bild 4.6: Ablaufschema einer Batch-Bearbeitung. 21
Bild 4.7: Synchronisation durch Situationen. 21
Bild 4.8: Ablaufschema des normalize compare Algorithmus. 22
Bild 4.9: Ablaufschema einer brute force Klassifikation. 23
Bild 4.10: Ablaufschema der Normalisierung. 23
Bild 4.11: Eine Gruppe von Objekten tauscht Propagierungen aus. 24
Bild 4.12: Ablaufschema der Propagierung. 25
Bild 4.13: Warteschlangenmodell der parallelen Propagierung. 26
Bild 4.14: Exponentieller Anstieg der freien Propagierungen (logarithmischer Maßstab). 28
Bild 4.15: Ablauf der Subsumptionsprüfung in tableauxbasierten DL-Systemen. 29
Bild 5.1: Schema der Farm Kommunikation. 31
Bild 5.2: Kommunikation und Lastverteilung währen der ersten beiden Propagierungsstufen einer FLEX-Applikation. 33
Figur 6.1: Die Hardwarekonfiguration (Matrix) des Multicluster II. 35
Figur 6.2: Die Anzahl der benutzbaren Worker-Prozesse in Abhängigkeit von der Netzwerkgröße. 36
Bild 6.3: Die 'server loop' und das 'client server'-Modell. 37
Bild 6.4: Verteilung der Farmprozesse auf eine 4x4 Matrix. 41
Bild 6.5: Informationsfluß im Verarbeitungsnetz. 42
Bild 6.6: Informationsfluß im Lastüberwachungsnetz. 43
Bild 6.7: Verteilung der Prozesse der DO-Implementierung auf eine 4x4 Matrix. 45
Bild 6.8: Informationsfluß im Verarbeitungsnetz. 46
Bild 6.9: Informationsfluß im Propagierungsüberwachungsnetz. 46
Bild 7.1: Ein object tell und die mit dem betroffenen Objekt verbunden Objekte. 48
Bild 7.2: Ein Benchmark-Beispiel mit 8 Objekten, 3 Konzepten und einer Konnektivität von 3. 49
Bild 7.3: Ausführungszeiten (Sekunden und relativ) der Farm. 49
Bild 7.4: Ausführungszeiten der DO-Implementation. 50
Bild 7.5: Relativer Speedup der DO-Implementation. 51

Bild 7.6: Absolute Task-Laufzeiten (Sekunden) der DO-Implementation. 52
Bild 7.7: Laufzeitverhalten von distributed object-FLEX auf einem 3x3 Netzwerk. 53
Bild 7.8: Shortest Path-Routing-Algorithmus. 54
Bild 7.9: Quadrantenaufteilung der Matrix-Topologie. 55
Bild 7.10: Kommunikation der negativen X-Achse in den 1. und 2. Quadranten. 56
Bild 7.11: Kommunikation vom 3. Quadranten in den 1. Quadranten. 56
Bild 7.12: Eine alternativer shortest path-Routing-Algorithmus mit besserem hot spot-Verhalten. 58

1. Einleitung

Diese Arbeit beschäftigt sich mit der Parallelisierung von beschreibungslogischen Systemen (*description logics* oder auch DL-Systemen). Ziel ist dabei, solche Systeme auf ihr paralleles Potential hin zu untersuchen und geeignete Strategien für eine parallele Implementation zu entwickeln.

Hintergrund dieser Untersuchung bildet die zunehmende Anwendung von DL-Systemen auf praktische Probleme wie Sprachverarbeitung und medizinischen Diagnose. Ermöglicht wurde diese Entwicklung durch Fortschritte in der theoretischen Fundierung und in der Benutzerfreundlichkeit von beschreibungslogischen Systemen. Allerdings führt die praktische Anwendung zu großen Wissensbasen (im Vergleich zu Prototypen), so daß die Bearbeitungsgeschwindigkeit nicht mehr der Forderung nach 'Interaktivität' (Antwortzeiten im Sekunden-Bereich) nachkommt. Die Erörterung diese Probleme hat in der Vergangenheit großen Raum eingenommen und hat zu verschiedenen Lösungsansätzen geführt:

- Algorithmische Optimierungen. Durch Verbesserung der Algorithmen sind wesentliche Geschwindigkeitsgewinne erzielt worden ([Baader et al 92]).

- Die Einschränkung der 'Vollständigkeit'. Vollständigkeit entspricht dem Umfang der logischen Schlußfolgerungen, die aus gegebener Information gezogen werden kann. Durch die Beschränkung der Vollständigkeit sinkt die Anzahl der Schlußfolgerungen, so daß sich die Geschwindigkeit erhöht. Ein Beispiel für Systeme mit eingeschränkter Vollständigkeit ist das System 'Loom' [MacGregor 91].

- Die Einschränkung der 'Expressivität'. Dabei handelt es sich um die Ausdrucksfähigkeit von beschreibungslogischen Sprachen. Schränkt man die Menge der Sprachkonstrukte ein, werden weniger Schlußfolgerungen berechnet, so daß trotz großer Vollständigkeit eine schnelle Verarbeitung erzielt werden kann. Ein Beispiel dafür ist das 'Classic' System [Brachman et al 91].

- Die Verwendung von schnellen Computern oder Programmiersprachen. Insbesondere der Übergang von 'klassischen' KI-Programmiersprachen wie Prolog und Lisp zu 'C' scheint dabei Erfolge zu bringen.

- Die Einschränkung der Größe der Wissensbasis.

In diesem Zusammenhang steht auch das an der TU-Berlin entwickelte Wissensrepräsentationssystem FLEX, das auf den oben genannten Überlegungen basiert. FLEX bringt vor allem zwei neue Aspekte in diese Diskussion:

- Auf der theoretischen Seite die Fundierung der Schlußfolgerungsprozesse durch den Sequenzen-Kalkül, wodurch die Vorzüge einer strikt definierten Semantik mit einem 'vorwärts gerichteten Schlußfolgerungsverfahren' kombiniert werden [Quantz 95].

- Auf der Implementierungsseite ist es durch die selektive Beschränkung der Vollständigkeit einzelner Schlußfolgerungskomponenten durch den Benutzer möglich, in Abhängigkeit von der Anwendung, überflüssige Schlußfolgerungen zu vermeiden [Quantz 95].

FLEX wird derzeit in der semantischen Auswertung im Bereich des VERBMOBIL-Projektes eingesetzt, wo es um die inhaltliche Analyse von natürlichsprachlichen Sätzen geht. Speziell geht es darum, in Abhängigkeit des Kontextes zwischen verschiedenen Bedeutungen eines Wortes zu unterscheiden (z. B. "ausmachen" im Sinn von "Licht ausmachen" oder im Sinn von "Termin ausmachen"). Da die Tendenz zu immer größeren Sprachfragmenten geht, (größerer Anzahl von Wörtern und Satzkonstruktionen) steigen ständig die Anforderungen an die Ausführungsgeschwindigkeiten.

In dieser Arbeit wird ein neuer Weg zur Geschwindigkeitssteigerung untersucht, der durch die Anwendung der Parallelverarbeitung auf Beschreibungslogik gekennzeichnet ist. Diese Technik wird bereits in vielen unterschiedlichen Bereichen von Wissenschaft und Technik angewendet, anfangend bei Simulationen in der Physik über Bildverarbeitung bis zu Echtzeit-Robotersteuerungen. Auch in der KI spielt Parallelität eine zunehmende Rolle, wie beispielsweise an der Entwicklung des japanischen *5th Generation Project* [Moto-Oka 81] zu sehen ist.

Um den Nachweis der Anwendbarkeit solcher Techniken zu erbringen gehe ich im folgenden so vor, daß ich allgemein den Daten- und Kontrollfluß von sogenannten *normalize compare* basierten DL-Systemen analysiere und anschließend die gewonnenen Strukturen in bezug auf die Anwendbarkeit bekannter paralleler Paradigmen prüfe. Auf die Bewertung der verschiedenen Ansätze durch einige formale Kriterien gründet sich dann meine Entscheidung für die Vorgehensweise zur parallelen Implementation des FLEX Systems.

Die im folgenden dargestellten Techniken zur Parallelisierung beschränken sich nicht nur auf das FLEX-System, sondern sind, wie gezeigt werden wird, auch auf die größere Klasse der vorwärtsverkettenden- und agendabasierten Systeme anwendbar.

Im Anschluß an die Implementation des parallelen FLEX interpretiere ich die Messungen des Laufzeitverhaltens der beiden Parallelisierungs-Varianten. Dabei gewinne ich quantitative Aussagen über das Verhalten meines Ansatzes, so daß ich für solche und ähnliche Systeme Vorhersagen über Effizienz und Skalierbarkeit machen kann.

Zusammenfassend läßt sich sagen, daß das Ziel dieser Arbeit die Evaluation des Parallelisierungspotentials von Beschreibungslogik- und ähnlichen Systemen ist. Da es sich bei dieser Aufgabe darum handelt, die Ergebnisse zweier Forschungsgebiete zusammenzuführen (der Parallelverarbeitung und der Beschreibungslogik), gehe ich so vor, daß ich zuerst einen Überblick über die Entwicklungen der jeweiligen Gebiete gebe (Kapitel 2. und 3.). Dann analysiere ich den Datenfluß von *normalize compare* basierten DL-Systemen in bezug auf Parallelisierbarkeit (Kapitel 4.).

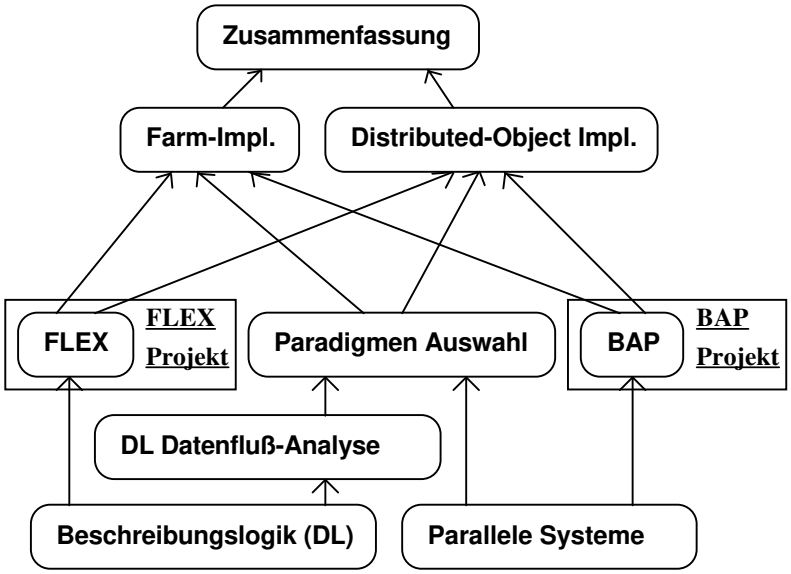


Bild 1.1: Aufbau dieser Diplomarbeit.

Als Ergebnis dieser Diskussion kristallisieren sich bestimmte parallele Paradigmen heraus, von denen ich im weiteren Verlauf der Arbeit zwei Versionen am Beispiel des FLEX-Systems implementiere (Kapitel 6.) und die experimentellen Ergebnisse vergleiche (Kapitel 7.). Die Ergebnisse sollen Aufschluß darüber geben, in welchem Maße Parallelismus auf DL- und ähnliche Systeme anwendbar ist.

2. Parallele Systeme

Motiviert durch den ständig wachsenden Bedarf an Rechenleistung hat die Entwicklung der Computertechnik seit ihrer Entstehung zu einem exponentiellen Anstieg der Rechenleistung von Computern geführt. In den letzten Jahren hat dieses Wachstum zu einer Annäherung an die allgemeinen physikalischen Grenzen geführt, die durch die örtliche Auflösung in den Mikrochips, vor allem aber durch die Begrenzung der Taktfrequenzen durch Transistor-Schaltzeiten bestimmt sind. Da unter diesen Bedingungen Geschwindigkeitszuwächse mit zunehmendem Aufwand verbunden sind, geht der Trend in der Entwicklung neuer Computer stark zur Parallelisierung von Rechnerkomponenten, sowohl innerhalb als auch außerhalb der Zentralen Recheneinheit (*Central Processing Unit*, *CPU*).

Während dabei die Parallelität innerhalb der Mikroprozessor-Technologie durch die Ausnutzung inherenter Parallelität (z.B. *fetch decode execute pipeline*) bzw. durch die Compiler-Schnittstelle für die Benutzer weitgehend verborgen blieb, führte der Übergang zu massiv parallelen Computern in den letzten Jahren zu neuen Anforderungen an die Softwareentwicklung und löste verstärkte Forschungsaktivitäten aus.

Im folgenden stelle ich einige der bekanntesten Schemata vor, die sich im Umfeld dieser neuen Systeme entwickelt haben. Dabei gliedere ich die Darstellung nach den verschiedenen Ebenen, auf denen Parallelität ausgedrückt wird:

- Hardware
- Programmiersprache
- Ausdrucksmittel
- Kommunikationsstrukturen

Diese Unterteilung erscheint mir vorteilhaft, da sich auf dieser Basis Unterschiede in den Programmiersprachen und den Ausdrucksmitteln auf die zugrundeliegenden Hardwareplattformen zurückführen lassen.

2.1. Parallele Hardware (SISD/MIMD/SIMD)

Parallele Verarbeitung von Daten auf Hardwareebene spielt seit den Anfängen der Computertechnik eine Rolle. Anfangs bezog sich dieser Begriff allerdings weniger auf den Bereich der CPU (*Central Processing Unit*) der Computer, sondern beispielsweise auf I/O Subsysteme wie Festplatten, Drucker oder Streamer Bänder, die unabhängig voneinander und gleichzeitig betrieben wurden. Erst die letzten Jahre brachten einen Trend in Richtung paralleler Systeme auf der Mikroprozessorebene, wodurch im Gegensatz zur sequentiellen 'von Neumann' Architektur die enger werdenden physikalischen Grenzen im Bereich von Taktfrequenz und Speicherbandbreite (Transfergeschwindigkeit zwischen Prozessor und Speicher) umgangen werden sollten. Dieser Trend äußert sich dadurch, daß innerhalb eines logischen Computers verschiedene Daten- und/oder Steuerwerke betrieben werden. Die sich daraus

entwickelnde Vielfalt von Hardware-Architekturen führte zu einem wachsenden Bedarf an Kriterien, nach denen die einzelnen Ansätze bewertet und miteinander verglichen werden konnten. [Flynn 72] beispielsweise unterteilt die existierenden Ansätze anhand der Anzahl der unabhängigen Instruktionen bzw. Daten, die zu einem bestimmten Zeitpunkt aktiv sind. [Young 87] und danach [Raina 93] vervollständigten diese Klassifikation durch die Einbeziehung von Speicherorganisation.

Basierend auf Flynn's Unterscheidung zwischen Steuerwerk und Verarbeitungswerk eines Prozessors lassen sich folgende vier Hauptgruppen unterscheiden:

	Ein Steuerwerk (SI)	Mehrere Steuerwerke (MI)
Ein Datenwerk (SD)	SISD: (*single instruction single data*) Die konventionelle sequentielle Architektur. Jedoch läßt sich zwischen dem von Neumann Modell (Programm und Daten in demselben Speicher) und der Harvard Architektur (Programm und Daten in verschiedenen Speichern) unterscheiden.	MISD: (*multiple instruction single data*) Entfällt für massiv parallele Systeme. Allerdings lassen sich u.U. Vorausschauende Ausführung oder '*branch prediction*' innerhalb von Mikroprozessoren hier einordnen.
Mehrere Datenwerke (MD)	SIMD: (*single instruction multiple data*) Ein Steuerwerk bestimmt die Verarbeitung mehrerer Datenströme durch parallele Verarbeitungswerke.	MIMD: (*multiple instruction multiple data*) Unabhängige Steuerwerke bestimmen in zugeordneten Verarbeitungswerken (Hauptgruppe der parallelen Systeme).

Bild 2.1: Steuerwerk/Verarbeitungswerk Kombinationen.

Bei MIMD Systemen, die sich bisher am weitesten verbreiten konnten, und damit die größte Vielfalt an Varianten hervorbrachten, kann zusätzlich zwischen engerer und loserer Kopplung zwischen den verschiedenen Prozessoren unterschieden werden. Enge Kopplung entspricht dabei stärkerer gemeinsamer Nutzung von Ressourcen wie Speicher oder I/O, während losere Kopplung für unabhängigere Systeme steht, die typischerweise über ein Kommunikationsnetzwerk miteinander verbunden sind.

Hinsichtlich der effizienten Anwendung von parallelen Ausdrucksmitteln und Programmierstilen ist diese Unterscheidung von großer Bedeutung, da einige Ausdrucksformen nur bei enger Kopplung von Prozessoren sinnvoll anwendbar sind, (z.B. Monitore[1] und Semaphoren). Andererseits bieten lose gekoppelte Systeme eine höhere Anzahl von nutzbaren Prozessoren, da die bei eng gekoppelten Systemen exklusive Ressource 'Speicher' bei diesen Systemen keinen Engpaß bildet (da jeder Prozessor seinen eigenen Speicher besitzt). Für beide Architekturen lassen sich jeweils Argumente finden, die in unterschiedlichen Anwendungsgebieten mehr oder weniger ausschlaggebend sind.

[1] [Welsh McKeag 81]

In der letzten Zeit scheint sich eine technologische Konvergenz der neu vorgestellten Parallelcomputer anzudeuten, die aus schnellen *off shelf* Mikroprozessoren, einer größeren Menge von Speicher und einem Verbindungsnetzwerk bestehen. Es läßt sich vermuten [Culler et al 93], daß dieser Trend durch wirtschaftliche Gründe verursacht wurde, da Parallelcomputer dieser Art durch die Verwendung von Standartprodukten ein gutes Preis-Leistungs-Verhältnis aufweisen.

Beispiele für enge Kopplung sind *shared memory*-Systeme, bei denen sich mehrere Prozessoren den selben physikalischen Speicher teilen, wie z.B. Sequent Symmetry und viele mehr. Beispiele für lose Kopplung findet man bei *message passing*-Systemen wie Thinking Machines' Connection Machine CM5, Cray T3D, Parsytec Transputer bzw. GC Systeme, Intel Paragon und vielen mehr.

2.2. Parallele Sprachen

Entsprechend der Entwicklung bei der Hardware haben sich in der Software eine große Menge von Programmiersprachen entwickelt, um das vorhandene parallele Potential zu nutzen. Insbesondere der Gegensatz zwischen *shared memory*- und *distributed memory*-Systemen hat zu stark unterschiedlichen Motivationen in der Entwicklung von Programmiersprachen geführt. *Shared memory*-Computer lassen eine wesentlich feinere Aufteilung der parallelen Ausführung zu, während *distributed memory*-Maschinen besser für eine höhere Anzahl von Prozessoren geeignet sind, da hier der Speicher keine globale Ressource bildet und so dieser Engpaß (*bottleneck*) der *shared memory*-Systeme vermieden wird. Diese Unterschiede führen bei den Programmiersprachen zu stark abweichenden Konzeptionen, da i.A. *shared memory*-Kommunikation und -Synchronisation über Prozeß-gemeinsame Variablen abläuft, während *distributed memory*-Computer dazu häufig Nachrichten benutzen.

Auf der Ebene der Sprachdefinition läßt sich weiter eine Einteilung machen zwischen sequentiellen Sprachen, die durch Library-Funktionen erweitert werden, um Parallelität auszudrücken, und um originär parallele Sprachen, bei denen die Parallelität durch spezielle Konstrukte von vornherein in der Konzeption enthalten ist.

Die folgenden Tabellen zählen einige typische parallele Programmiersprachen und -Dialekte für die jeweiligen Architekturen auf. Da es sowohl möglich ist, *shared memory* über Nachrichten zu emulieren als auch Nachrichten über *shared memory*, ist die hier getroffene Einordnung nur von tendenzieller Natur, und es wird immer Argumente für eine andersgeartete Einordnung geben.

	Library-Erweiterung	Syntaxerweiterung
Shared Variable		Parallel Modula Pascal Plus[2]
Message Passing	Logical Systems C[3] PVM[4] Dome[5]	Occam[6] 3L C Charm[7]

Bild 2.2. Imperative Parallele Sprachen.

	Library-Erweiterung	Orginäre Konstrukte
Shared Variable	P-Prolog Port Prolog[8] C-Prolog[9]	And/Or Prolog[10] Parlog[11] Concurrent Prolog[12] Guarded Horn Clauses[13]
Message Passing	Delta Prolog[14] Brain Aid Prolog[15]	Strand 88 Parsytec Prolog[16]

Bild 2.3. Deklarative Parallele Sprachen.

2.3. Parallele Ausdrucksmittel

Es gibt sehr verschiedene Möglichkeiten, Parallelität auszudrücken. Grundlegende Konzepte aller Sprachen sind dabei Kommunikation und Synchronisation zwischen Prozessen [Freeman 75]. Unter einem Prozeß versteht Freeman eine:

> "discrete progression, in time, of discernible changing states"[17]

Außerdem sei ein Prozeß nur beschreibbar durch seine externe Charakteristik, also die Interaktion mit der Umwelt. Dieser *black box*-Ansatz impliziert, daß jeder Prozeß unabhängig von allen anderen Prozessen ist, solange keine Daten-Abhängigkeiten existieren. In anderen Wor-

---

[2] [Welsh, McKeag 81]
[3] [Lsc 88]
[4] PVM (*Parallel Virtual Maschine*) [Geist Et Al 93] ist eine *message passing* Bibliothek, die mit verschiedenen Programmiersprachen benutzt werden kann, wie z.B. Fortran oder 'C'.
[5] [Beguelin et al 94]
[6] [Occam 88]
[7] [Kale 94]
[8] [Lourenço Borges 91]
[9] [Pereira 83]
[10] [Conery et al 83]
[11] [Clark et al 84]
[12] [Shapiro 83]
[13] [Ueda 87]
[14] [Cunha et al 89], [Pereira 86]
[15] [BAP 93]
[16] [Bergmann 89]
[17] In Deutsch etwa: 'ein diskretes Vortschreiten zwischen unterscheidbaren Zuständen mit der Zeit'

ten, solange zwei Prozesse nicht voneinander in bezug auf Eingabedaten abhängen, können sie parallel zueinander laufen. Eine formale Definition von Prozessen mit diesen Eigenschaften kann in [Hoare 80] und [Hoare 85] gefunden werden.

Unter Kommunikation versteht man danach die Übertragung von Information zwischen logisch oder physikalisch getrennten Systemen. Synchronisation wird vom selben Autor als das Zusammenführen der Kontrollflüsse mehrerer Prozesse verstanden. Betrachtet man beispielsweise die Ausführung eines Programms als Zustandswechsel in einem gerichteten Graphen, so läßt sich Synchronisation als das Zusammenführen mehrerer unabhängiger Kontrollflüsse zu einem einzelnen Punkt verstehen.
Kommunikation und Synchronisation sind die grundlegenden Konzepte, die durch eine Vielzahl von Mechanismen ausgedrückt werden können, wie z.B. Locks, Semaphoren, Critical Sections, Streams, Kanäle, Monitore und viele mehr. Für Parallelismus in Logikprogrammiersprachen hat sich besondere Mechanismen entwickelt, die Synchronisation 'datengetrieben' durch den Zustand von Variablen auszudrücken (z.B. [Clark et al 84]).

	Prozesse	Nachrichten	Shared Var's	Semaphoren	Kanäle	Monitore	Flow Anal.
Occam	*	*	*		*		
Logical Systems C	*	*	*	*			
Strand 88	*	*	*		*		
Brain Aid Prolog	*	*					
Parlog							*
Concurrent Prolog							*
And/Or Prolog							*

Bild 2.4: Parallele Ausdrucksmittel in einigen Programmiersprachen.

Bild 2.4 listet einige Programmiersprachen mit den zur Verfügung stehenden Kommunikations- und Synchronisationsmechanismen auf. Da eine vollständige Darstellung dieses Bereiches den Rahmen dieser Arbeit sprengen würde, verweise ich hier auf die einschlägige Literatur.

2.4. Kommunikationsstrukturen

Unter einer Kommunikationsstruktur verstehe ich die makroskopische Sicht auf die (Nachrichten-) Interaktion von Prozessen in einem System. Solche Strukturen korrespondieren i.A. eng mit dem Datenflußschema der gegebenen Applikation.

Im Verlauf der Entwicklung paralleler Systeme hat sich eine große Zahl von unterschiedlichen Beschreibungsformen und Realisierungen von Datenflußschemata entwickelt. Besonders markante Beispiele für Steuerungsmechanismen sind Datenfluß-Programmiersprachen und Kommunikations-Protokolle wie Occam [Occam 88] oder SDL [Færgemand, Olsen 94], datengetriebene Programmiersprachen wie Concurrent Prolog [Shapiro 83] oder der *communicating sequential processes* Mechanismus von Hoare [Hoare 85]. All diese Systeme erlauben die Konstruktion von sehr komplexen Kommunikations-Schemata, mit deren Hilfe der Applikationsdatenfluß modelliert werden kann.

Das Problem bei diesen komplexen Schemata ist, daß sie ein großes Potential von *deadlocks* besitzen, d.h. daß Situationen auftreten, in denen sich zwei oder mehr Prozesse gegenseitig so blockieren, daß die Systemaktivität zum Stillstand kommt. Verschiedene Techniken wurden entwickelt, um solche Probleme zu behandeln und zu vermeiden (z.B. [Jensen 92]).

In all den möglichen Kommunikationsschemata zeichnen sich vier Schemata durch besonders einfache Strukturen aus, so daß dort die *deadlock*-Behandlung trivial wird.

- **Farm**: Diese Struktur entspricht einer reinen Nebeneinanderbearbeitung von Aufträgen. Dies wird häufig implementiert als eine sternförmige Kommunikationsstruktur, in der ein 'Master' mit mehreren 'Worker'-Prozessen kommuniziert. Alle Nachrichten werden von, bzw. zum Master gesendet. Die einzelnen Worker kommunizieren nicht untereinander. Der Datenfluß in einem Farm-Konstrukt ist durch ein 'Nebeneinander' der Datenverarbeitung gekennzeichnet. Anwendung findet diese Technik beispielsweise in *blackboard*-Systemen [Schoinas 91], im AND/OR-Parallelismus (mit Einschränkungen bei partiellen Abhängigkeiten [Clark et al 84]) und bei der parallelen Auswertung von Spielbäumen.

- **Pipeline**: Diese Struktur entspricht einer reinen 'Nacheinanderbearbeitung' von Aufträgen. Es entsteht eine lineare Kommunikationsstruktur, in der jeder Prozeß mit genau zwei anderen Prozessen verbunden ist, so daß sich eine 'Kette' ergibt. Der Kommunikationsfluß läuft dabei nur in eine Richtung.

- **Farm/Pipeline Kombination**: Aus der Kombination von Farm und Pipeline ergeben sich kombinierte Schemata, deren Kommunikationsfluß aus den primitiven Komponenten zusammengesetzt ist.

- ***Distributed* Objects**: Diese Struktur besteht aus einer Menge von Prozessen, die direkt untereinander kommunizieren. Allerdings ist diese Kommunikation auf asynchrones 'Senden' beschränkt, so daß jeder Prozeß jedem anderen Nachrichten schicken kann, jedoch nicht die Möglichkeit hat, auf Antworten zu warten. Die Deadlockfreiheit dieser Struktur ergibt sich aus der ständigen 'Empfangsbereitschaft' aller beteiligten Prozesse. Prozesse in einer solchen Struktur nenne ich 'Objekte', wenn sie ihren 'inneren Zustand' (*state*) zwischen zwei empfangenen Nachrichten bewahren.

2.5. Gütekriterien für parallele Systeme

In diesem Abschnitt versuche ich einige Kriterien zu explizieren, die dazu dienen sollen, verschiedene Implementationen einer Applikation miteinander vergleichen zu können. Auf diesen Kriterien lassen sich dann Designentscheidungen in bezug auf die Implementierung des FLEX-Systems begründen. Die Definition dieser Begriffe erfolgt in Anlehnung an [Cooley, Turkey 65].

- **Speedup** (absoluter Geschwindigkeitsgewinn): Das Verhältnis einer nicht-parallelisierten, sequentiellen Implementation einer Applikation im Verhältnis zur Ausführungszeit der parallelisierten Applikation ($\frac{T_{seq}}{T_{par}}$).

- **Effizienz**: Die mittlere Auslastung der Prozessoren während der Laufzeit der Applikation, die sich als das Verhältnis $\frac{Speedup}{N}$ ergibt.

- **Skalierbarkeit**: Die maximale Anzahl von Prozessoren auf denen sich eine Applikation mit einer bestimmten Effizienz ausführen läßt. Diese Größe ist stark von Hardware-Charakteristika, wie Kommunikationsbandbreite und Topologie abhängig.

Ziel für die Parallelisierung von DL-Systemen ist möglichst hoher Speedup, hohe Effizienz und hohe Skalierbarkeit über einen möglichst weiten Bereich von Netzwerkgrößen und Topologien. Weiter lassen sich die 'klassischen' Bewertungskriterien aufzählen wie Robustheit und Effizienz (im klassische Sinn).

2.6. Grenzen der Parallelität

Den wichtigsten Faktor für eine effiziente Parallelisierung einer Applikation bildet das parallele Potential der Applikation. Applikationen mit einem streng sequentiellem Datenfluß werden sich nur schlecht parallelisieren lassen, egal welche Methoden dafür verwendet werden.

Ist eine Analyse des Datenflusses und der Datenabhängigkeiten einer Applikation gegeben, so lassen sich maximale Effizienz und maximaler Speedup leicht durch Amdahls Gesetz [Amdahl 1967] berechnen: T_{seq} sei die sequentielle Ausführungszeit einer Applikation. $\alpha = \frac{T_{parallelizable}}{T_{seq}}$ sei der 'parallelisierbare Anteil' der Ausführungszeit, der sich auf N Prozessoren ausführen läßt. Damit ergibt sich:

$$Speedup = \frac{T_{seq}}{T_{parallel}} = \frac{T_{seq}}{T_{seq} * (1-\alpha) + \frac{T_{seq} * \alpha}{N}}$$

$$Speedup = \frac{N}{\alpha + N - N * \alpha}$$

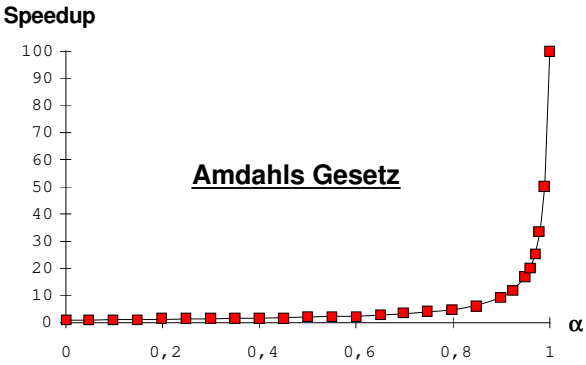


Bild 2.5: Amdahls Gesetz, Formeln und Anwendung auf ein 100 Prozessor Netzwerk.

Die Grafik oben zeigt, daß nur ein $\alpha \approx 1$ zu befriedigenden Ergebnissen führen kann. Ein $\alpha < 1$ bewirkt, daß sich nur eine begrenzte Zahl von Prozessoren sinnvoll einsetzen lassen, was einer Begrenzung der Skalierbarkeit entspricht. Wichtig ist, daß oft auch Kommunikationszeiten

zum sequentiellen Ausführungsteil zu zählen sind (z.B. beim Farm Konstrukt in Kapitel 5.1.), so daß sich α mit wachsender Anzahl der Prozessoren häufig verschlechtert.

3. Beschreibungslogik (Description Logic)

Dieses Kapitel stellt kurz die Grundzüge der Beschreibungslogik (DL) vor, indem die Entwicklung der grundlegenden Ideen nachvollzogen und schließlich eine formale Definition gegeben wird. Diese Darstellung erfolgt in Anlehnung an [Quantz 95].

Beschreibungslogik hat sich gegen Ende der 60er Jahre aus dem Gebiet der Wissensrepräsentation entwickelt. Quillian [Quillian 68] vermutete als erster, daß semantische Strukturen die Grundlage für verschiedene kognitive Leistungen, wie Sprache und Perzeption bilden. Er entwickelte einen Formalismus zur Repräsentation von Wissen auf der Basis von Knoten, die über 'beschriftete Bögen' miteinander verbunden sind. Ein anderes Paradigma wurden 1975 von Minsky entwickelt [Minsky 75]. In seinen *frames* verwendet Minsky 'Objekte', die jeweils eine bestimmte Menge von *slots* aufweisen, über die sie mit anderen Objekten verbunden sein können. *Slots* können zusätzlichen Informationen enthalten, wie z. B. *default*-Werte oder Hinweise auf Prozeduren, die bei Veränderungen des *slot*-Status aktiviert werden (Dämonen). Beide Repräsentationsformalismen weisen gewisse Ähnlichkeiten auf, was sich insbesondere dadurch ausdrückt, daß sich viele Aspekte von semantischen Netzen mit *frames* ausdrücken lassen und umgekehrt. Trotzdem existieren eine Reihe von Unterschieden, vor allem in der Art, wie die Paradigmen die Beschreibung bestimmter Phänomene unterstützen.

Beide, *frames* und semantische Netze, sind Vorläufer der Beschreibungslogik (*description logics*). Obwohl auch hier starke Ähnlichkeiten zu sehen sind, unterscheidet sich die Beschreibungslogik von ihren Vorgängern vor allem durch dem Anspruch auf theoretische Fundierung. Dies entwickelte sich historisch dadurch, daß in der zweiten Hälfte der 70er Jahre die Repräsentationssprachen der Netze, *frames* und Skripte von [Woods 75] und [Hayes 77] heftig in Bezug auf ihre formelle Fundierung angegriffen wurden. Der Hauptangriffspunkt war die Beziehung zwischen Wissensrepräsentation und formaler Logik. Dazu ist zu bemerken, daß Quillian seine semantischen Netze zwischen formaler Logik und natürlicher Sprache angesiedelt hat. Dies impliziert eine expressive Überlegenheit gegenüber der symbolischen Logik. Die entscheidende Frage, die sich auf diesem Hintergrund stellt ist, wie sich feststellen läßt, ob ein semantisches Netz mehr Information enthält als ein anderes. Dies führt zurück zum Fehlen der Beweisbarkeitsrelation in diesen Netzen und zu der Frage, was genau die Bedeutung eines semantischen Netzes ist. Hayes beantwortet diese Frage durch das Zurückführen der *frame* Definitionen auf Formeln der Prädikatenlogik erster Stufe (*first order logic*, FOL) [Hayes 80] und zeigt so, daß *frames* der Prädikatenlogik in bezug auf Expressivität nicht überlegen sind.

Dieser Sicht schloß sich Brachmann in [Brachmann 77], [Brachmann 79] an. Statt aber die semantischen Netze in bezug auf FOL zu formalisieren, untersuchte er im Detail die Bedeutung der einzelnen Konstrukte. Dieser Untersuchung brachte zwei Ergebnisse hervor: zum einen die Unterscheidung zwischen den Ebenen der Implementation, der Logik, der Epistemologie, der Konzepte und der Sprache, zum anderen den 'KL-ONE' Formalismus. Dieses Paradigma bildete in den 80er Jahren die Grundlage für eine ganze Familie von DL-Systemen, die trotz verschiedener Schwerpunkte Ähnlichkeiten in bezug auf die zugrundeliegende Repräsentationsphilosophie aufwiesen. Alle Systeme haben die folgenden Eigenschaften:

- Sie besitzen eine formale Sprache in dem Sinne, daß sich eine formale Spezifikation des Syntax geben läßt.
- Sie besitzen eine formale Semantik, die die Beweisbarkeitsrelation bestimmt
- Sie besitzen einen (effizienten) Algorithmus, der die Beweisbarkeitsrelation berechnet.

3.1. Formale Spezifikation

Formal bestehen diese DL Systeme aus einem Alphabet, termformenden Operatoren und der Beweisbarkeitsrelation:

Das DL-Alphabet besteht aus 'Konzepten' (einstellige Prädikate), 'Rollen' (zweistellige Prädikate) und 'Objekten' (Individuenkonstanten).

Die termformenden Operatoren bilden das Hauptunterscheidungsmerkmal von DL-Logiken und bestehen z.B. aus Konjunktion, Disjunktion, Negation für Konzepte und Rollen, Typ- und Werteinschränkungen für Konzepte und Komposition und Inversion für Rollen.

Drei verschiedene ausgezeichnete Formeln werden in DL-Systemen gebraucht, nämlich 'Termeinführung'[18] ($t_n :< t$ oder $t_n := t$), 'Regeln' ($c_1 \Rightarrow c_2$), und 'Objekt Beschreibungen' ($o :: c$). Mit diesen Primitiven lassen sich Netzwerke aus Begriffen erstellen, die als Grundlage dienen können für zwei grundlegende Anfragetypen:

$$t_1 \; ?< \; t_2$$
$$o \; ?: \; c$$

Die erste Anfrage ist genau dann wahr, wenn der Term t_1 subsummiert wird unter den Term t_2 (also. t_2 genereller ist als t_1), die zweite genau dann, wenn das Objekt o eine Instanz des Konzeptes c ist..

In der Implementierung von DL-Systemen werden hauptsächlich zwei verschiedene Schlußfolgerungs-Paradigmen verwendet. Ursprünglich wurden Schlußfolgerungen durch einen *normalize compare* Algorithmus realisiert, bei dem zuerst Konzepte und Rollen in eine einheitliche 'Normalform' transformiert und dann strukturell verglichen werden. Dabei ist interessant, daß Objekte genauso behandelt werden können wie Konzepte, was zur Folge hat, daß dieselben Algorithmen zur Normalisierung und zum Vergleich benutzt werden können.

Ende der 80er Jahre wurden dann *Tableaux Methoden* entwickelt, die den Resolutionsverfahren der FOL entsprachen [Donini et al 91a, Donini et al 91b]. Der darauf aufsetzende Subsumptionsalgorithmus hat dabei den Vorteil, daß er eine exzellente Basis für theoretische Untersuchungen bietet. Es war nicht nur möglich, die Korrektheit und Vollständigkeit dieser Algorithmen zu beweisen, sie erlaubten auch, die Berechenbarkeit und Handhabbarkeit (*tractability*) verschiedener DL-Dialekte zu untersuchen.

---

[18] Wir benutzen den Begriff 'Term' hier sowohl für Konzepte als auch für Rollen.

Der Hauptnachteil tableauxbasierter Subsumptionsalgorithmen besteht allerdings in der Tatsache, daß sie nicht konstruktiv sind, sondern Ablehnungs-Techniken (*refutation techniques*) verwenden. Um z.B. die Subsumption $c_1 \subseteq c_2$ zu beweisen, wird gezeigt, daß der Term $c_1 \wedge \neg c_2$ inkonsistent wird, also daß $\Gamma \cup \{o :: c_1 \wedge \neg c_2\}$ nicht erfüllbar ist. Obwohl dies ein eleganter Ansatz für die Subsumptionsprüfung ist, führt er zu starken Effizienzproblemen im Bereich von Abfragen. Um z.B. die Instanzen eines Konzeptes 'c' zu finden, müßte man sämtliche Objekte der Modellierung auf die Erfüllbarkeit $\Gamma \cup \{o :: \neg c\}$ testen.[19]

In den meisten existierenden Systemen werden Inferenzregeln mehr als Produktionsregeln verstanden, die dazu benutzt werden, bestimmte Folgerungen aus den initialen Informationen im voraus zu berechnen. Dies korrespondiert mehr mit dem Kalkül des natürlichen Schließens (*natural deduction*) [Prawitz 65] oder dem Sequenzen Kalkül (*sequent calculi*) [Gentzen 65], zwei Deduktionssystemen aus dem Kontext der Prädikatenlogik (*first order logic*). In [Royer, Quantz 92] wurde daher vorgeschlagen, die Vorteile der normalize-compare Ansätze mit denen der tableauxbasierten Systemen zu vereinigen, indem der Sequenzen Kalkül statt tableauxbasierter Methoden zur Charakterisierung der Deduktionsregeln verwendet wird. Wie die Tableaux-Methoden bietet der Sequenzen Kalkül einen logisch konsistenten Rahmen, erlaubt aber im Gegensatz zu Tableaux Methoden einen konstruktiven Theorembeweis.

Basierend auf diesen in [Royer, Quantz 92] und [Royer, Quantz 94] beschriebenen Ideen wurde das DL-System 'FLEX' an der Technische Universität Berlin entwickelt. Dieses System stellt die Grundlage für die Parallelisierung dar, die im folgenden entwickelt wird. Jedoch werden dabei keine speziellen Eigenschaften von FLEX benutzt, so daß die Ergebnisse auch auf viele andere DL- und vorwärtsverkettende Systeme anwendbar sind.

3.2. Subsumptionstest

Um Subsumption zwischen zwei Termen zu testen, werden beide zuerst 'normalisiert' und dann strukturell verglichen. Das Format der Normalisierungsregeln hängt dabei stark von der Mächtigkeit des verwendeten DL-Dialekts ab. Für die Ausführungen in dieser Arbeit ist es hinreichend, Normalformen als Menge von Atomen zu verstehen, so daß sich alle Normalisierungsregeln in der folgenden Form schreiben lassen:

$$\alpha_1,\ldots,\alpha_n \longrightarrow \beta$$

d.h. wenn die Atome $\alpha_1,\ldots,\alpha_n$ in der Normalform enthalten sind, so wird β hinzugefügt. Während des folgenden Vergleichs wird dann getestet, ob für jedes Atom in der Bezugs-Normalform ein spezielleres Atom in der zu subsummierenden Normalform gefunden wird.

[19] Siehe [Schaerf 94] für tableaux-basierte Algorithmen für Schlußfolgerungen auf Objekt-Ebene und [Kindermann 95] für eine Diskussion von Effizienzproblemen

3.3. Klassifikation

Während der Einführung neuer Terme in eine Modellierung wird jeder Term klassifiziert, d.h. er wird mit allen vorher eingeführten Termen verglichen. Als Ergebnis erhält man eine *Subumptionshierarchie* für Konzepte und Rollen, die jeweils einen gerichteten antizyklischen Graphen darstellen. Klassifikation ist implementiert als ein Suchvorgang zwischen Superkonzepten und Subkonzepten, d.h. durch eine Subsumptionsvergleiche zwischen dem neuen Term und allen vorher eingeführten.

3.4. Propagierung

Die beiden bisher beschriebenen Schlußfolgerungskomponenten werden im allgemeinen *terminological reasoning* genannt. Im Gegensatz dazu wenden wir unsere Aufmerksamkeit nun auf das *assertional reasoning*, d.h. auf das Schlußfolgern auf Objektebene.
Der Hauptunterschied zwischen terminologischem und assertorischem Schlußfolgern ist, daß das erste inherent lokal ist, währen das zweite inherent global ist. Schlußfolgern auf Objektebene läßt sich im Allgemeinen in einen lokalen und einen globalen Teil aufspalten.
Im lokalen Teil bestimmen wir für jedes Object das *most specific concept*, das es instanziert. Dieses wird mit Hilfe des standard '*normalize compare*' Algorithmus in Verbindung mit der Bestimmung der direkten Superkonzepte zur Klassifikation gemacht. Wir normalisieren also die Beschreibung eines Objektes und erhalten dabei eine Normalform, die wir mit allen anderen Objekten in der Konzepthierarchie vergleichen. Zusätzlich zur Normalisierung haben wir dabei alle DL-Regeln anzuwenden. Dies wird erreicht durch die Anwendung aller Regeln, deren linke Seiten unter die Objekt Normalform subsummiert werden können. Nach diesem Verarbeitungsschritt wird die neue Normalform wieder normalisiert und klassifiziert und der Vorgang des Anwendens und Normalisierens solange angewendet, bis sich keine Veränderungen mehr ergeben [Owsnicki-Klewe 88].
In der dann folgenden nicht-lokalen Phase werden bestimmte Informationen der Normalform zu anderen Objekten 'propagiert'. Als Beispiel können die folgenden Propagierungen dienen:

$$
\begin{aligned}
&o_1 :: \text{all}(r\,,\,c\,)\ \&\ r : o_2 && \longrightarrow o_2 :: c\\
&o_1 :: r\ \&\ \text{atmost}(1,\,r\,),\ o_2 :: c && \longrightarrow o_1 :: \text{all}(r\,,\,c\,)\\
&o_1 :: r_1 : o_2,\ o_2 :: r_2 : o_3 && \longrightarrow o_1 :: r_1\ \text{comp}\ r_2 : o_3\\
&o_1 :: r : o_2 && \longrightarrow o_2 :: \text{inv}(r\,) : o_1
\end{aligned}
$$

Wir nennen alle diese Regeln nicht-lokal, weil Informationen eines Objektes o_1 eine Auswirkung auf ein anderes Objekt o_2 haben können. Abhängig von der 'Konnektivität' der Objekte können dann propagierte Änderungen ihrerseits wieder Konsequenzen für andere Objekte haben, und so weiter. Im Extremfall kann eine Änderung an einem einzelnen Objekt so einen Zustandswechsel des gesamten Systems zur Folge haben.

3.5. Situationen

Situationen sind eine Erweiterung der Beschreibungslogik, die erlauben, den Zustand der assertionale Schlußfolgerungskomponente zu parametrisieren. Basis bildet die Situationstheorie von Barewise und Perry [Barewise, Perry, 83], in der eine von Frege abweichende Semantik definiert wird.

Den Hintergrund für Situationen bildet die Idee, daß in beschreibungslogischen Kontexten Mehrdeutigkeiten auftreten können. Als Beispiel kann man sich verschiedene Lesarten eines Wortes einer natürlichen Sprache vorstellen wie Schloß (Gebäude) - Schloß (Teil einer Tür). Da diese beiden Bedeutungen unvereinbar miteinander sind, können sich nicht gleichzeitig in derselben Wissensbasis enthalten sein. Nun ist die Idee, die Wissensbasis in zwei Varianten zu spalten, die den Bedeutungen entsprechen. Treten weitere Mehrdeutigkeiten auf, so ergibt sich ein ganzer 'Situationsbaum'.

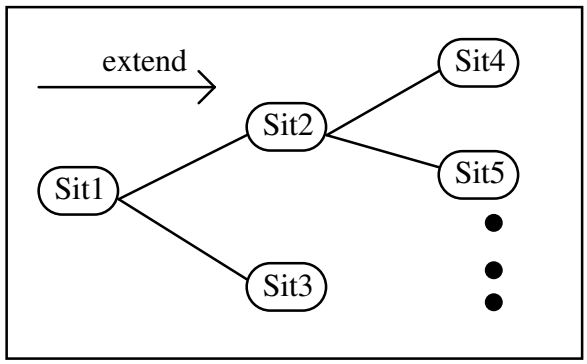


Bild 3.1: Ein Situationsbaum.

Unter der Annahmen, daß 'sit1' den Zustand vor der Mehrdeutigkeit bezeichnet, so läßt sich das Beispiel oben in der Sprache der Beschreibungslogik so formulieren:

ich :: besitze : schloß in sit1.
sit2 ≤ sit1.
sit3 ≤ sit1.
schloß :: gebäude in sit2.
schloß :: ist_teil_von : tür in sit3.

(Ich besitze ein Schloß (sit1). Dieses Schloß ist entweder ein Gebäude (sit2) oder ist Teil einer Tür (sit2)).

Auf diese Konstellation können jetzt z.B. andere Regeln angewendet werden, wenn beispielsweise bekannt ist daß 'das Schloß klemmt'. In diesem Fall wird die 'Gebäude' Lesart (sit2) verworfen und nur die 'Teil einer Tür' Lesart (sit3) bleibt erhalten.

Der '≤' Operator steht für den Begriff der 'Erweiterung', der zwischen zwei Situationen gelten kann. Erweitert eine Situation S2 eine Situation S1 (S1 ≤ S2), so bedeutet das, daß alles, was

in S1 gilt auch in S2 gilt. Darüber hinaus kann weiteres Wissen in S2 vorhanden sein, was aber S1 nicht beeinflußt.

4. Paralleles Potential und Datenfluß in DL-Systemen

In Zusammenfassung des letzten Kapitels läßt sich sagen, daß die grundlegende Architektur von DL-Systeme aus zwei Datenbank-Komponenten besteht, deren Zustand durch Termeinführungen geändert und durch Abfragen inspiziert werden kann. Der bei diesen Aktionen auftretende Datenfluß wird im folgenden analysiert und in Bezug auf das Potential für Parallelisierung untersucht.

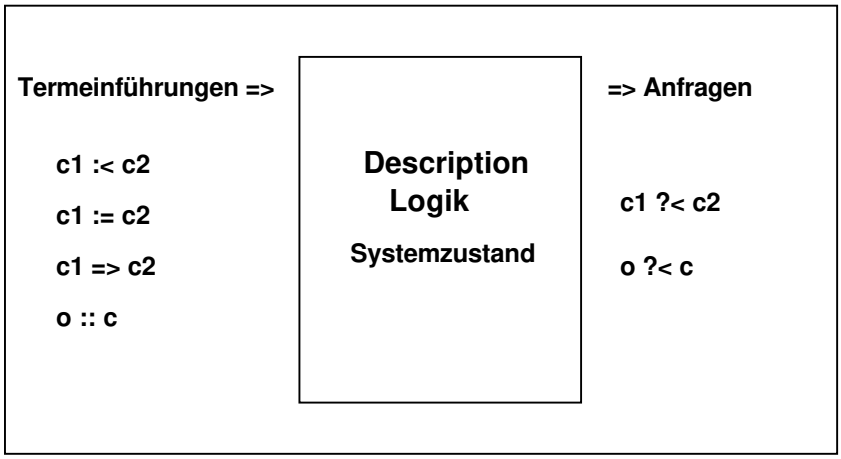


Bild 4.1: DL-Systemzustand und Interaktion.

Bild 4.1 zeigt ein DL-System als *black box* mit den zur Verfügung stehenden Interaktionen mit der Außenwelt. Der Systemzustand läßt sich dabei vereinfachend als Konzepthierarchie darstellen, in der die im System existierenden Objekte 'eingeordnet' sind.

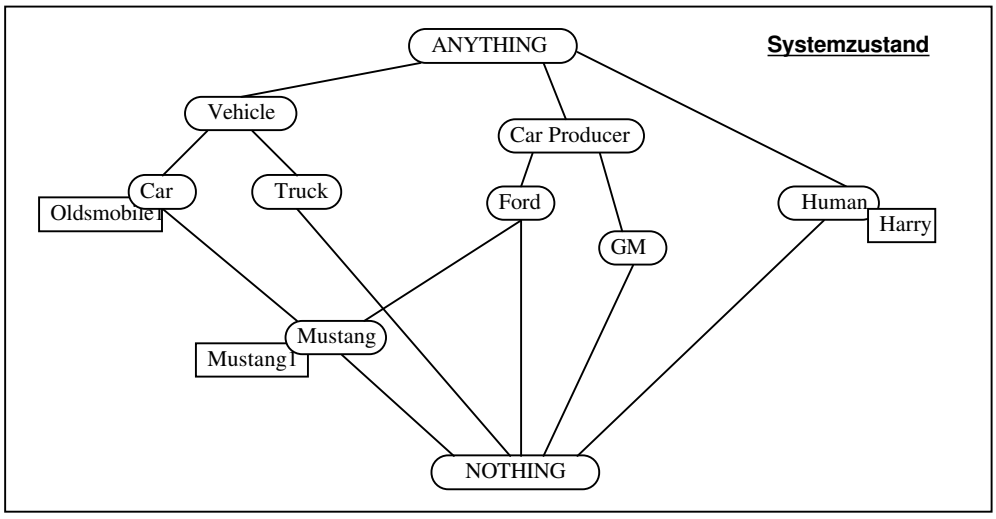


Bild 4.2: DL-System Zustand.

Es läßt sich dabei eine Unterteilung in einen Konzept- und einen Objektraum vornehmen, was der Unterscheidung zwischen der terminologischen und assertorischen Schlußfolgerungskomponente entspricht. Formal wird dabei der terminologische Teil als ein gerichteter, antizyklischer Graph (DAG) dargestellt, der assertorische Teil als Menge von Individuen mit Bezügen zueinander und zur terminologischen Komponente.

4.1. Datenfluß auf Term-Einführungsebene

Durch die konstruktive (vorwärts verkettende) Anwendung der Deduktionsregeln in FLEX werden wesentliche Teile[20] der Konzepthierarchie und der Objektbeschreibungen sofort nach einer Termeinführungen berechnet. Dieses Vorgehen leitet sich aus der Forderung ab, die Ausführungszeit für Anfragen kurz zu halten [Kindermann 95].

In der folgenden Auflistung werden die Verarbeitungsschritte für die einzelnen Transaktionen erläutert. Dabei werden Normalisierung, Klassifikation und Propagierung als grundlegende Operationen angenommen:

- $o::c$: Die Beschreibung eines Objektes wird erweitert oder ein neues Objekt wird eingeführt. Als Folge wird die Objektbeschreibung (Normalform) normalisiert und klassifiziert, um die Position des Objektes in der Terminologie zu ermitteln.

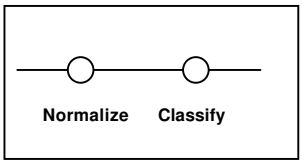


Bild 4.3: Ablaufschema einer Objekt-Termeinführung.

- $t_n :< t$ oder $t_n := t$: Ein Term wird in die Terminologie eingeführt oder erweitert. Dabei wird die linke Seite des Terms normalisiert, klassifiziert, und in der Konzepthierarchie ihrer Position entsprechend eingebunden. Anschließend müssen alle diejenigen Objekte 'nachklassifiziert' werden, die in direkter Beziehung zu den betroffenen Konzepten stehen. Aus der Nachklassifikation können sich weitere Propagierungen ergeben.

[20]Hier findet ein Kompromiß zwischen dem Berechnen von Schlußfolgerungen zur Eingabezeit bzw. der Abfragezeit statt. Näheres dazu bei [Kindermann 95].

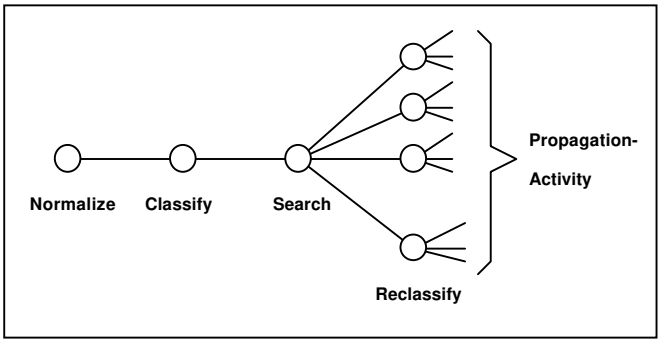


Bild 4.4: Ablaufschema einer Konzept-Termeinführung.

- $c_1 \Rightarrow c_2$: Eine neue DL-Regel wird in die Terminologie eingeführt. Für alle Objekte im System muß getestet werden, ob die linke Seite der neuen Regel auf sie anwendbar ist. Tritt dieser Fall bei einem Objekt auf, so folgt die oben beschriebene Propagierungsaktivität. Alle Operationen auf Konzepten und Objekten können dank der Konfluenzeigenschaft (Kapitel 4.2.4.) parallel ausgeführt werden.

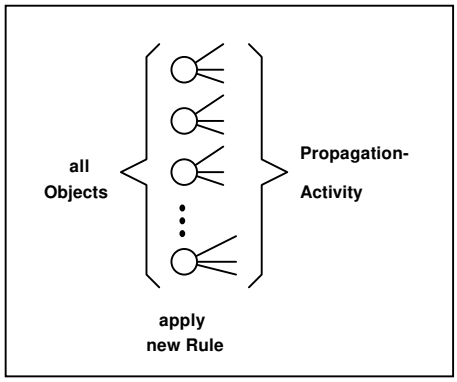


Bild 4.5: Ablaufschema einer Regel-Einführung.

- Häufig werden DL-Systeme so betrieben, daß viele Einführungen gemeinsam gemacht werden ('Batch-Betrieb'), um eine Datenbasis aufzubauen, die dann durch einzelne Kommandos abgefragt wird [Kindermann 95]. Aus der gemeinsamen Einführung von Termen und der Konfluenzeigenschaft (Kapitel 4.2.4.) folgt, daß auch diese Operationen parallel ausgeführt werden können. Deshalb können im Batch-Modus mehrere Term-Einführungen parallel bearbeitet werden.

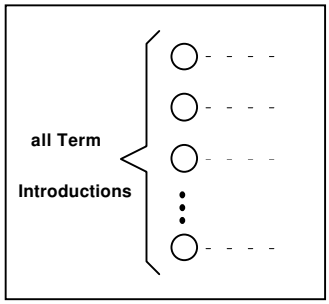


Bild 4.6: Ablaufschema einer Batch-Bearbeitung.

Die Parallelität auf Term-Einführungsebene erleidet eine Einschränkung durch das Situationskonzept. Dies liegt darin begründet, daß das Ziehen von Schlußfolgerungen in einer Situation i.A. von dem Zustand vorheriger Situationen abhängig ist und somit nicht parallel ausgeführt werden kann. Aus diesem Grund stellt jede Situation einen 'Synchronisationspunkt' dar. Soll die nächste Aktion gestartet werden, so müssen vorher alle laufenden Aktionen der vorherigen Situation beendet sein.

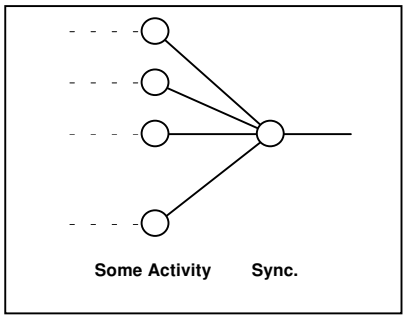


Bild 4.7: Synchronisation durch Situationen.

Allerdings gewinnt dieses Problem in den untersuchten Anwendungen (Kapitel 7.1.) nur geringes Gewicht. Dies erklärt sich aus den folgenden Gründen:

- Viele Anwendungen benutzen keine Situationen.
- Bei der Anwendungen von 'Defaults' [Quantz 95] ist die Berechnungskomplexität der einzelnen Situationen (*default spaces*) sehr groß.
- Beispiele wie in Anhang A zeigen, daß durch einen einzelnen Tell eine große Menge von Propagierungen innerhalb einer Situation ausgelöst werden.

Ein Problem in einem anderen Bereich entsteht durch die gleichzeitige Ausführung von Tells, die die Neueinführung von Objekten bewirken. Werden diese Tells gleichzeitig ausgeführt (was durch die parallele Ausführung auftreten kann), so würde ein Objekt zweimal eingeführt werden, was zu einem inkonsistenten Systemzustand führen würde. Aus diesem Grund ist eine Koordinierung von Objekteinführungen notwendig, die in Kapitel 6.5. näher erläutert wird.

4.2. Datenfluß auf Operationsebene

Die drei im vorigen Kapitel als atomar angenommen Operationen Normalisierung, Klassifikation und Propagierung können ihrerseits auf ihren Datenfluß hin untersucht werden und zeigen ein hohes Potential an Nebenläufigkeit.

4.2.1. Subsumption

Wie in dem Kapitel über Beschreibungslogik erwähnt, wird Subsumption in den hier betrachteten DL-Systemen durch den *normalize compare* Algorithmus implementiert. Definitionsgemäß besteht dieser Algorithmus aus zwei Teilen, nämlich aus einer Normalisierung und einem folgenden Vergleich. Dabei existiert eine Abhängigkeit zwischen diesen beiden Komponenten derart, daß der Vergleich erst nach abgeschlossener Normalisierung ausgeführt werden kann.

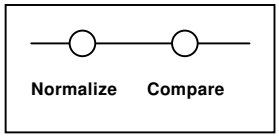


Bild 4.8: Ablaufschema des *normalize compare* Algorithmus.

4.2.2. Klassifikation

Klassifikationsalgorithmen lassen sich in '*brute force*'- und höher entwickelte Verfahren einteilen [Baader et al 92]. Die grundlegende Idee ist dabei, eine Normalform mittels Subsumptionstests in eine gegebene Hierarchie einzuordnen. Im *brute force* Fall wird dabei die gegebene Normalform mit den Normalformen aller bereits existierender Konzepte mit Hilfe der Subsumptionsrelation verglichen, was mit hoher Effizienz parallelisierbar ist (da die einzelnen Subsumptionstests unabhängig von einander sind).

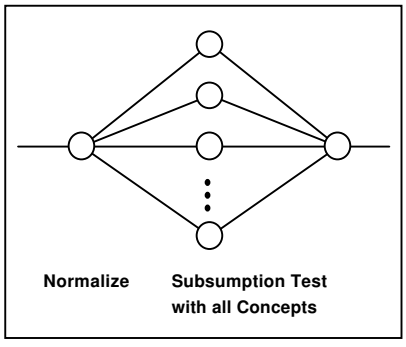


Bild 4.9: Ablaufschema einer *brute force* Klassifikation.

Allerdings sind wesentlich 'intelligentere' Verfahren bekannt [Baader et al 92], die zu komplizierteren Abhängigkeiten zwischen den einzelnen Subsumptionstests führen und deren Datenfluß damit nicht in einer einfachen Abbildung darstellbar ist. Zu beachten ist, daß eine Parallelisierung des *brute force* Verfahrens sich immer mit der optimierten sequentiellen Version vergleichen muß, so daß nur geringe Effizienz ($\frac{t_{seq}}{t_{par} * N}$) zu erwarten sind.

4.2.3. Normalisierung

Normalisierung beruht auf der sukzessiven Anwendung verschiedener Normalisierungsregeln, die Schlußfolgerungsregeln [Royer, Quantz 93] repräsentieren. Dieser Vorgang läßt sich für jede Regel in zwei Teile unterteilen, nämlich die Prüfung der Anwendbarkeit und der anschließenden Bearbeitung. Zu jeder Regel gehören einige Schlüsselatome in der Normalform, die für die Anwendung der Regel zwingend notwendig sind. Jeder einzelne Anwendbarkeitstest unabhängig von allen anderen, so daß sich für die Normalisierung das folgende Schema ergibt:

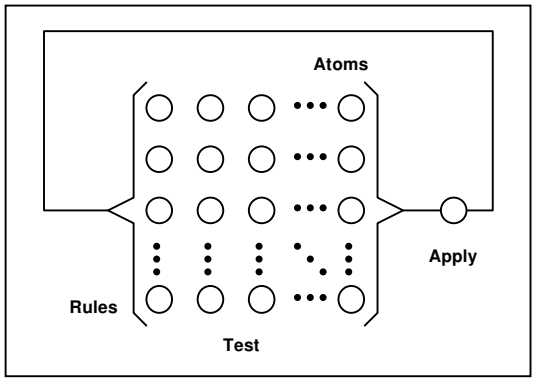


Bild 4.10: Ablaufschema der Normalisierung.

Da es sich bei der Normalisierung um einen Algorithmus ähnlich dem der 'Termersetzungs-Systeme' mit Konfluenzeigenschaft (Unabhängigkeit des Ergebnisses von der Ausführungsreihenfolge, Kapitel 4.2.4.) handelt, ist die Korrektheit des Ergebnisses in jedem Fall gesichert.

Ein denkbarer Ansatz für die Parallelisierung der Normalisierung ist die parallele Anwendung aller Normalisierungsregeln [Royer, Quantz 93] oder die gleichzeitige Normalisierung verschiedener Teile einer Objekt- oder Konzeptbeschreibung. Bei dem anschließenden Vergleich kann z.B. die Suche nach passenden generelleren Atomen verteilt werden.
Allerdings sind sowohl die Normalisierungs- als auch Vergleichsparallelisierung a priori ungünstig, da die erste im hohen Maße iterativ ist (nach der Anwendung einer Normalisierungsregel müssen i.A. wieder alle anderen Regeln getestet werden bis sich nichts mehr ändert) und letztere eine a priori ungleichmäßige Verteilung der Last (nämlich in Abhängigkeit von der Anzahl der vorhanden Atome) besitzt. Weiter wirkt sich ein zusätzlicher Synchronisationspunkt zwischen den beiden Vorgängen negativ aus, da hierbei die Ausführung notwendigerweise sequentialisiert wird, was vor allem in größeren Systemen zu hohem *overhead* führt. Weiter liegt die 'Körnigkeit' all dieser Operationen in der Größenordnung von 10 - 1000 Unifi-

kationen, so daß eine Parallelisierung nur in dafür ausgelegten feinkörnigen Systemen[21] sinnvoll ist. Dann ist beispielsweise bei der Anwendung von Normalisierungsregeln eine Trennung von Anwendungstest und -Ausführung denkbar, was insbesondere in Systemen mit feinkörnigem Parallelismus zu einer Steigerung der Effizienz führen könnte. Weiter würde von einer Parallelisierung der Subsumption sowohl die terminologische als auch die assertorische Schlußfolgerungskomponente profitieren.

4.2.4. Propagierung

Propagierung läßt sich beschreiben als ein vorwärtsverkettender Schußfolgerungsvorgang, bei dem die einzelnen Inferenzen an verschiedenen Objekten auftreten.

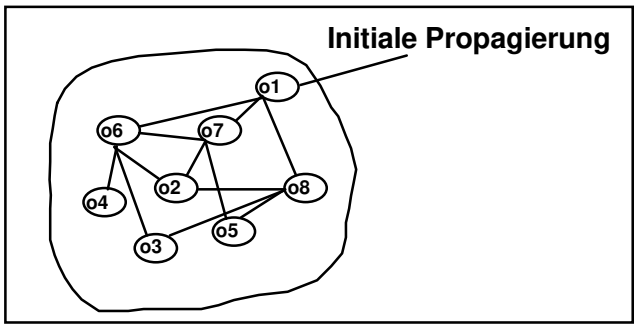


Bild 4.11: Eine Gruppe von Objekten tauscht Propagierungen aus.

Wie bereits oben beschrieben, handelt es sich dabei um einen nicht-lokalen Vorgang, da Inferenzen bei einem Objekt im Extremfall Auswirkungen auf alle anderen Objekte im System haben können. Dieser Vorgang läßt sich quantitativ durch die 'Konnektivität' des Systems beschreiben, indem festgestellt wird, wieviel Objekte im Durchschnitt von den Änderungen eines Objekts betroffen sind. Dieser 'Fan Out' spielt insofern eine große Rolle, als bei einer Konnektivität >1 und festen Propagierungsschritten eine exponentielle Zunahme der im System befindlichen Propagierungen festzustellen ist.

[21]Für eine Gegenüberstellung von feinkörigen und grobkörnigen Systemen siehe [Young 87]

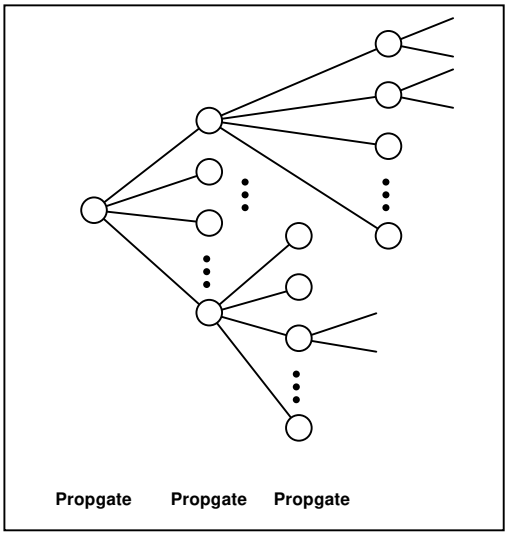


Bild 4.12: Ablaufschema der Propagierung.

Dieses Wachstum endet erst, wenn die neue Information im zunehmenden Maße in das System integriert ist, so daß die Konnektivität sinkt und das System einen 'Fixpunkt' erreicht. Da Propagierungen in beschreibungslogischen Systemen monoton sind und in beliebiger Reihenfolge ausgeführt werden können, wird derselbe Fixpunkt immer erreicht. Wir nennen diese wichtige Eigenschaft 'Konfluenz'.

Bild 4.12 zeigt die ersten Stufen nach dem Start des Propagierungsprozesses. In diesem Beispiel erzeugt jede Propagierung genau drei weitere Propagierungen. Dies erzeugt eine 'Kettenreaktion', so daß sich die Anzahl der 'freien Propagierungen' exponentiell vervielfacht. Dieser Vorgang ebbt ab, wenn die neue Information in das System integriert wird und die Konnektivität abnimmt.

Propagierungen an verschiedenen Objekten können parallel abgearbeitet werden, da sie sich gegenseitig nicht beeinflussen. Wichtig ist dabei, daß während des gesamten Ablaufs der Propagierung die terminologische Wissenskomponente konstant bleibt. Dies ist bei den betrachteten NLP Anwendungen ([Quantz, Schmitz 94], [Kindermann 95]) der Fall.

Auf Basis der hier getroffenen Annahmen läßt sich das Propagierungsverhalten durch Aufträge ('Tasks') beschreiben, deren Laufzeit einer statistischen Wahrscheinlichkeitsverteilung gehorcht. Im Anschluß an die Bearbeitung eines Tasks werden Nachfolge-Tasks erzeugt (in Bild 4.13 symbolisiert durch den '*multiplier*'), deren Anzahl von der Interkonnektivität der modellierten Wissens-Domäne und ihrer 'laufenden Nummer' abhängt. Diese laufende Nummer ergibt sich als '0' für von außen kommenden (initiale) Tasks und als $k+1$ wenn der Vorgänger-Task die laufende Nummer k hatte. Die Eigenschaft der Konfluenz (Kapitel 4.2.4.) garantiert eine Termination des Vorgangs. Daraus ergibt sich die folgende Beschreibung[22]:

[22][Gordon, Newell 67]

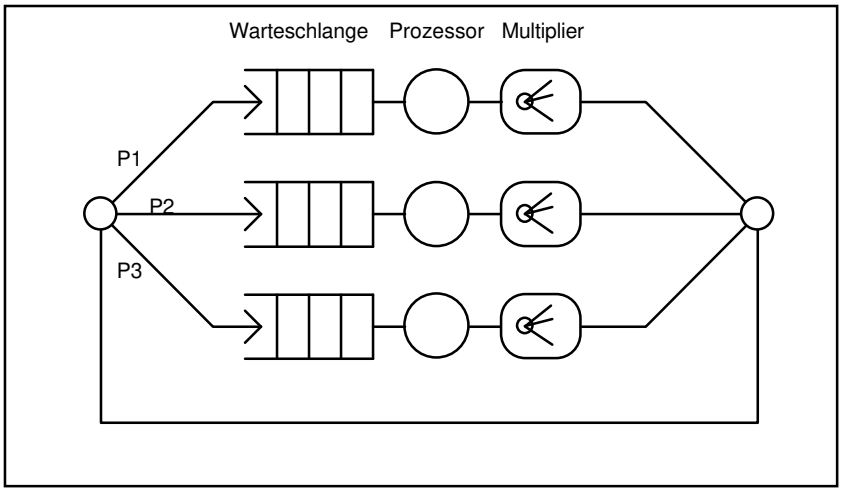


Bild 4.13: Warteschlangenmodell der parallelen Propagierung.

Jeder Task hat eine Laufzeit T_{Task} mit $P(T_{Task} = t) = \lambda(t)$ und erzeugt N Nachfolger mit der Funktion $P(Succ_{Task} = n) = \mu(k,n)$, die die laufenden Nummer der Tasks berücksichtigt. Die Nachfolger werden wieder in das System zurückgekoppelt, nachdem ihre laufende Nummer k um 1 erhöht wurde. Beide Funktionen λ und μ sind abhängig von der betrachteten Applikation.

Folgende Betrachtung führt zu der Verteilung μ der Zufallsvariablen N: Jedes Objekt steht mit einer Anzahl von anderen Objekten in Verbindung (über Rollen). Wird 'neue' Information zu einem Objekt hinzugefügt, so wirkt sich das jeweils mit der Wahrscheinlichkeit P_0 auf die verbundenen Objekte aus (z.B. über Rollenfüllerbeschränkungen) und es werden Propagierungen an diese Objekte verschickt. Die verbundenen Objekte ihrerseits schicken inverse Rollenfüllerinformation (die Information, daß das zweite Objekt und das erste verbunden sind) und eventuell neu gewonnene Informationen an die mit ihnen verbundenen Objekte. Informationen, die dabei an das ursprünglich Objekt zurückgeschickt werden, führen dabei i.A. nicht zu weiteren Propagierungen (weil es sich dabei um die inverse Rollenfüllerinformation handelt), während neu gewonnene Information zu einer Fortführung des Vorgangs führt. Als Parameter für die Anzahl der neu ausgelösten Propagierungen μ ergeben sich damit die Interkonnektivität j der Objekte, der Anteil P_{neu} der 'neuen' Information jeder Propagierung für neue Informationen, und die Propagierungswahrscheinlichkeit P_{prop} für neue Information.

Bei der von außen ausgelösten ersten Propagierung ist alle Information 'neu', so daß die Anzahl der ausgelösten Propagierungen sich bei gleichverteilter Wahrscheinlichkeit als binominalverteilt ergibt:

$$\mu(0,k) = \binom{N_O}{k} * p^k * q^{N_O - k},\ k \le N_O$$

mit p als P_{prop}, q als $1-p$ und N_O als Anzahl der mit dem Objekt verbunden Nachbarobjekte (in Worten: die Wahrscheinlichkeit dafür, genau k Nachfolgepropagierungen zu erzeugen ist binominal verteilt zwischen 0 und der Anzahl der verbundenen Objekte).

Aus dieser Formel läßt sich bereits eine wichtige Größe berechnen, nämlich die Auslastungsverteilung der N_{Proz} Prozessoren im Warteschlangenmodell: Nimmt man für jedes Objekte einen Prozessor an, so ist die Anzahl der beschäftigten Prozessoren gleich der Anzahl der Propagierungen. Sind weniger Prozessoren vorhanden (mehrere Objekte werden von einem Prozessor bearbeitet), so besteht eine höhere Belegungswahrscheinlichkeit für die Prozessoren.

Bei der zweiten Propagierung gilt dasselbe wie bei der ersten Propagierung, da noch keine propagierte Information zum Auslöser zurückgekehrt ist und somit $P_{neu} = 1$ ist. Für alle weiteren Propagierungen nimmt durch die Konfluenz-Eigenschaft des Netzes der Anteil der 'neuen' Information ab, so daß die Anzahl der im Netz befindlichen Propagierungen abnimmt.

Leider ist dieser Verlauf sehr schwer zu berechnen, da es sich um einen 'gedächtnisbehafteten' Vorgang handelt. Für die zweite Stufe ergeben sich zwar noch die folgenden Formeln:

für z=0:

$$P(z=0) = \sum_{i=0}^{\infty} P(X=i) * P(Y=0)^i$$

bzw. für k>0:

$$P(z=s) = \sum_{i=1}^{\infty} P(X=i) * \sum_{\underline{k} \in IK} \prod_{j=1}^{n} P(Y=k_j)$$

wobei P(X=i) die Wahrscheinlichkeit dafür ist, daß in einem Propagierungsschritt genau i Folgepropagierungen erzeugt werden.

Diese Formeln lassen zwar gewisse Aussagen über die Dauer und Auslastung der Startup-Phase des Netzwerks zu, leider kommt in der Formel eine nicht geschlossene Summation vor, so daß sich diese Berechnung nicht analytisch vornehmen läßt. Eine Simulation des Systemverhaltens wäre eine interessante Aufgabe, die mir leider in der zur Verfügung stehenden Zeit nicht möglich ist. Deshalb beschränke ich mich im folgenden lediglich auf qualitative Betrachtungen.

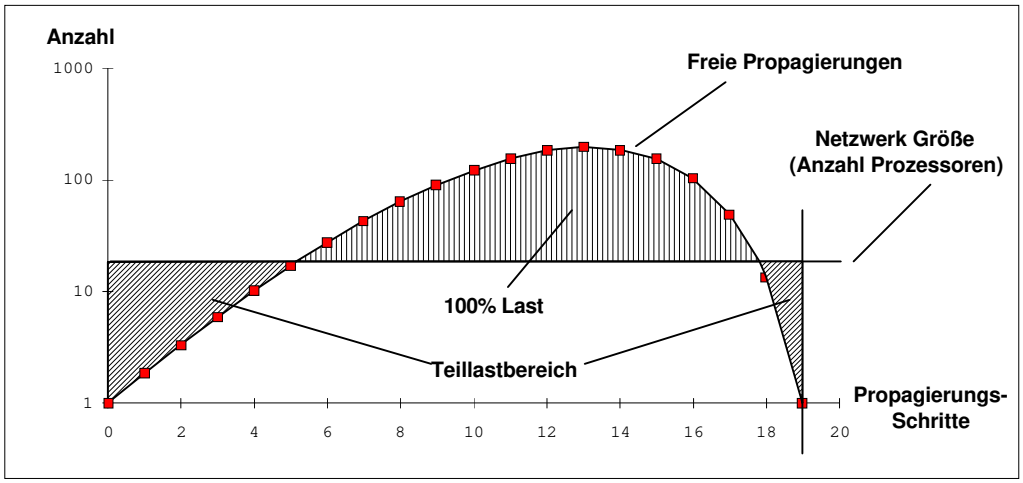


Bild 4.14: Exponentieller Anstieg der freien Propagierungen (logarithmischer Maßstab).

Bild 4.14 zeigt qualitativ den Anstieg und Fall der freien Propagierungen im System unter dem Aspekt einer festen Anzahl von verfügbaren Prozessoren. Dabei ist zu beachten, daß (bei begrenzter Anzahl der Prozessoren) die Ausführung des mittleren Teils deutlich länger dauert als die Ausführung der Propagierungen am Anfang und Ende, da in der 'Mitte' jeder Prozessor pro Schritt mehrere Propagierungen auszuführen hat. Das heißt, da die Fläche unter der Kurve gleichbleiben muß, daß der Mittlere Teil stark 'in die Länge gestreckt' würde (logarithmischer Maßstab). Die Effizienz der Parallelisierung ist dabei offensichtlich abhängig von dem Anteil des Teillastbereiches an der gesamten Bearbeitungsdauer. Die Granularität der Parallelisierung ist im Vergleich zu den vorher behandelten Schlußfolgerungsverfahren hoch, da jede Propagierung i.A. mehrere Subsumptionen voraussetzt.

4.3. Mikroskopischer Datenfluß

Dem in den vorangegangenen Abschnitten untersuchten Datenfluß auf Applikationsebene läßt sich der große Bereich von Ablaufschemata der zugrundeliegenden Implementierung gegenüberstellen. Dieser Bereich umfaßt alle Operationen, die auf der Ebene der zugrundeliegenden Datenstrukturen wie Listen, Bäume und Verbände angesiedelt sind. Sehr viel Arbeit ist in der Vergangenheit dabei im Bereich der Sprache Prolog auf die parallele Implementation von Listenbearbeitung verwendet worden [Bevemyr 93]. Hier werden datenflußgesteuerte Konzepte genutzt [Wise 82], deren zugrundeliegenden Operationen bis auf die Ebene von Assembler-Anweisungen verteilt werden und entsprechend fein parallelisiert werden können.

4.4. Datenfluß von tableauxbasierten DL-Systemen

Wie bereits in Kapitel 3.2. beschrieben, ist der Schlußfolgerungsprozeß in tableauxbasierten Systemen im Gegensatz zu den *normalize compare*-Systemen rückwärts gerichtet (*backward propagation*) und 'nicht-konstruktiv'. Das bedeutet, daß beispielsweise für den Beweis der Subsumption $c_1 \subseteq c_2$ gezeigt wird, daß der Term $c_1 \wedge \neg c_2$ inkonsistent wird, also daß $\Gamma \cup \{o :: c_1 \wedge \neg c_2\}$ nicht erfüllbar ist.

Dieser Nachweis wird erbracht, indem der Term $c_1 \wedge \neg c_2$ der bereits existierenden Modellierung hinzugefügt und dann durch die Anwendung von Schlußfolgerungsregeln in das System 'integriert' wird.

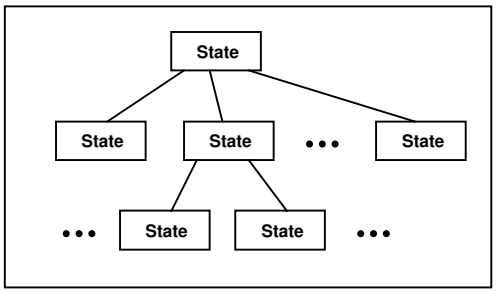


Bild 4.15: Ablauf der Subsumptionsprüfung in tableauxbasierten DL-Systemen.

Dieser Integrations-Vorgang ist dadurch charakterisiert, daß durch die Anwendung verschiedener Schlußfolgerungsregeln aus einem Zustand mehrere Fogezustände entstehen, die ihrerseits wieder Folgezustände haben können. Es entsteht ein Baum, dessen Bearbeitung sich gut parallelisieren läßt, da die Teilaufgaben große Unabhängigkeit aufweisen. Es ist beispielsweise möglich, die Subtasks der ersten Ebene mit Hilfe des `farm()` Prädikates aus Kapitel 6.2. zu verteilen, was zu einer sehr einfachen Lösung führt. Es ist jedoch zu bedenken, daß die Anzahl der zu ziehenden Schlußfolgerungen im Vergleich mit konstruktiven Verfahren erheblich größer ist, da viele Inferenzen mehrfach gezogen werden müssen.

4.5. DL-Datenfluß Zusammenfassung

Dieses Kapitel zeigt, daß es eine große Vielfalt von Möglichkeiten gibt, DL-Systeme zu parallelisieren. Die Hauptunterschiede der Varianten liegen dabei in der Granularität der zu verteilenden Aufgaben. Auf Basis dieser Untersuchung sind sehr viele unterschiedliche Strategien denkbar, DL-Systeme zu parallelisieren. Die Kriterien, die dafür ausschlaggebend sind, sind bereits in Kapitel 2.5. benannt worden.

5. Parallelisierungsstrategien für DL

Im vorigen Kapitel ist gezeigt worden, daß der Datenfluß von DL-Systemen eine große Mengen von parallelen Abläufen enthält, die Potential für Parallelisierungen bieten. In diesem Kapitel wird in groben Umrissen gezeigt, auf welche Art und Weise solche parallelisierbaren Abläufe tatsächlich für parallele Verarbeitung genutzt werden können.

Die Ergebnisse des vorigen Kapitels zusammen mit der zunehmenden Entwicklung von massiv parallelen Computern auf Basis der *loosely coupled*-MIMD-Technologie weisen dabei auf eine vorteilhafte Implementierung des assertorischen Inferenz-Moduls auf der Ebene der Propagierungen hin. Der wesentliche Vorteil eines solchen Ansatzes vor allen anderen hier dargestellten Alternativen gründet sich auf die folgenden Punkte:

- große Granularität der Teilaufgaben
- große Anzahl Teilaufgaben bei großen Problemen
- geringes Kommunikationszeit/Berechnungszeit Verhältnis
- gute Verfügbarkeit massiv paralleler Computer auf Basis der MIMD Technik

In diesem Fall wird diese Auswahl von geeigneten Parallelisierungsparadigmen vor allem durch die folgenden Punkte bestimmt:

- die Länge der einzelnen Aufgaben ist nicht *a priori* bekannt
- die Anzahl der Teilaufgaben ist nicht bekannt
- es ist nicht bekannt, in welche Richtung sich der 'Propagierungsfluß' bewegen wird

Diese Kriterien schließen viele Verfahren aus, die einen im voraus bekannten Datenfluß voraussetzen, wie z.B. Datenfluß-Parallelität. Zusammen mit der groben Granularität der Teilaufgaben und der guten Verfügbarkeit von MIMD Computern spricht deshalb viel für grobkörnige Paradigmen wie CSP [Hoare 85], die insbesondere auf massiv parallelen MIMD Computern mit Erfolg in verschiedenen Bereichen angewendet werden wie z. B. in der Bildverarbeitung [Burkhard et al 94] oder der Strömungssimulation [Strietzel 94].
Im Rest dieses Kapitels werden zwei Paradigmen näher analysiert, die der grobkörnigen Struktur von Parallelität auf Propagierungsebene entsprechen.

5.1. Farm Parallelismus

Die 'Farm'-Kommunikations-Struktur in Bild 5.1 ist ein oft genutztes Schema im Bereich industrieller Anwendungen wie Bildverarbeitung [Burkhard et al 94], finite Elemente [Diekmann et al 94] und vielen mehr. Es ist quantitativ leicht zu behandeln (siehe unten) und erlaubt eine große Anzahl von Optimierungen, wie z.B. Lastausgleich und hierarchische Organisation.

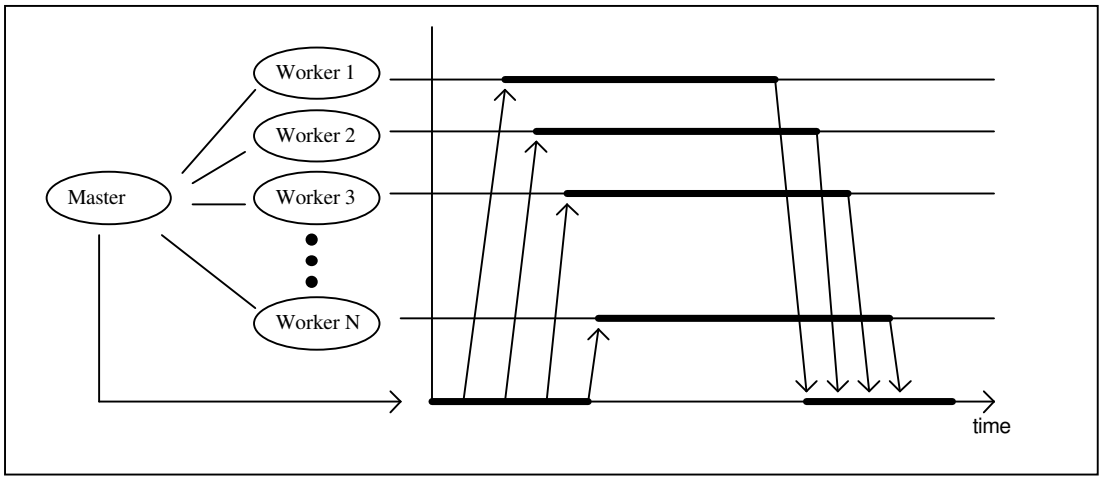


Bild 5.1: Schema der Farm Kommunikation.

Das Konstrukt besteht aus mehreren parallelen Prozessen mit einer vorgegebenen Struktur: Ein Prozeß (Master) ist verantwortlich für die Auf- und Verteilung der Anwendung auf eine Reihe von Worker-Prozessen. Diese bearbeiten ihre Teilaufgaben parallel und liefern dann die Ergebnisse zurück zum Master. Reine Farm-Strukturen werden oft bei Anwendungen eingesetzt, deren Teilaufgaben eine *a priori* bekannte Laufzeit haben, wie z.B. Bildverarbeitung.

Probleme treten allerdings bei unterschiedliche Laufzeiten der Teilaufgaben auf, bei denen sich eine schlechtere Auslastung der Worker ergibt, da zum Schluß nur noch Teile des Netzes ausgelastet sind. Weiter stellt die Kommunikationsverbindung zum Master einen Engpaß dar, der mit steigender Anzahl der Worker immer stärker an Bedeutung gewinnt und schließlich das Laufzeitverhalten dominiert.

Quantitative Aussagen über die Skalierbarkeit der Farm lassen sich verhältnismäßig einfach treffen. Zur Modellierung der Farm ignoriert man alle Einflüsse bis auf die Netzwerkbandbreite, mit der der Master-Prozeß mit den Worker-Prozessen kommunizieren kann. Dadurch erhält man eine Abschätzung der maximalen Anzahl der Worker-Prozesse in Abhängigkeit von der Netzauslastung (Effizienz).

Zur Berechnung der Laufzeit folgt man in Bild 5.1 einem Pfad, der vom Start der Applikation bis zu deren Ende läuft. Man erhält für die Gesamtlaufzeit der Applikation auf N Prozessoren mit deterministischen Subtask-Laufzeiten:;

$$T_{gesamt} = (\text{N-1})*(T_{SetupTask}+T_{SendMessage})+T_{MessageDelay}+\frac{T_{seq}}{N}+T_{MessageDelay}+T_{CollectTask}$$

$$Speedup = \frac{T_{seq}}{T_{gesamt}}$$

$$Effizienz = \frac{Speedup}{N}$$

Um beispielsweise eine Effizienz von 50% zu erreichen berechnet sich die maximale Anzahl der Prozessoren mit dem Ansatz:

$$50\% \quad = \frac{T_{seq}}{N * T_{gesamt}}$$

mit der Annahme (o.B.d.A), daß $T_{SetupTask} > T_{CollectTask}$:

$$50\% \quad < \frac{T_{seq}}{T_{seq} + N^2 * \left(T_{SetupTask} + T_{SendMessage}\right)}$$

so daß sich für die maximale Anzahl der Prozessoren ergibt:

$$N \quad < \sqrt{\frac{T_{seq}}{T_{SetupTask} + T_{SendMessage}}}$$

Setzt man in diese Formel Werte aus dem 'N-Queens'-Benchmark von Brain Aid Prolog Systems (siehe Anhang B) mit 6 'Damen' ein, so ergibt sich:

T_{seq} = 220ms

$T_{SetupTask}$ = 0.5ms

$T_{SendMessage}$ = 1.5ms

=> N < 10.5 (Anzahl der Prozessoren)

(Messungen mit BAP V1.1, T800 17 MHz)

Dies bedeutet, daß selbst dieser Benchmark, der als sehr 'parallelisierungsfreundlich' gilt, nur eine sehr eingeschränkte Skalierbarkeit aufweist[23].

[23]Zu bemerken ist, daß der N-Queens Benchmark die angenehme Eigenschaft hat, daß die Rechenzeit T_{seq} mit der Fakultät des Parameters N wächst, während $T_{SetupTask}$ und $T_{SendMessage}$ im wesentlichen gleich bleiben. Mit einer wachsenden Problemgöße (Anzahl der Damen) verbessert sich damit die Skalierbarkeit, so daß es mit diesem Benchmark immer möglich ist, eine hohe Effizienz zu erzielen.

5.2. *Distributed Objects*-Parallelismus

Im *distributed objects*-Paradigma (DO-Paradigma) wird die zentrale Management-Institution (Master) durch lokales Wissen der Prozesse (interpretiert als Objekte) untereinander ersetzt. Weiter muß die Kontrollstrategie verändert werden, so daß sich aus dem Mikroverhalten der einzelnen Objekte das gewünschte Makroverhalten des Systems herausbildet.
Derartige System werden u.a. von der 'Verteilten Künstlichen Intelligenz' (VKI) untersucht und entwickelt [Dossier 91]. Das hier vorgestellte Verfahren unterscheidet sich jedoch in der Zielsetzung stark von bekannten VKI-Systemen, da im Gegensatz zu 'autonom verhandelnden Agenten' ein ganz bestimmtes Datenfluß-Schema als Anforderung vorgegeben ist. Weiter dominiert bei dieser Anwendung der Effizienz-Aspekt der Parallelisierung die Implementierung. Deshalb kann man unter Berufung auf [Bond, Gasser 88] diesen Ansatz in das Gebiet des 'Verteilten-Problem-Lösens' einordnen.

Der grundlegende Unterschied zwischen dem Farm und DO Paradigma besteht in der Perspektive der Parallelisierung: in der Farm werden Aufgaben verteilt, im DO-Paradigma Objekte.

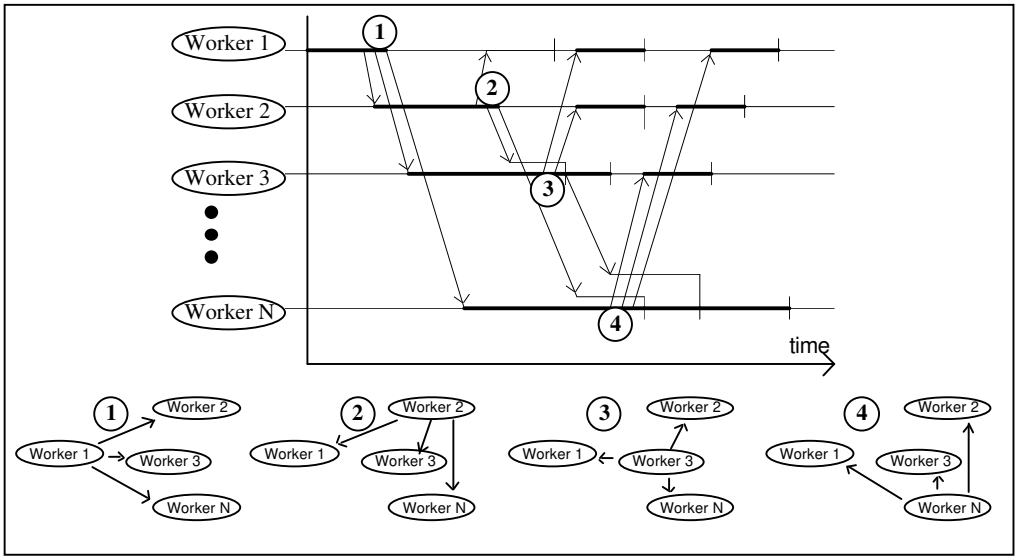


Bild 5.2: Kommunikation und Lastverteilung währen der ersten beiden Propagierungsstufen einer FLEX-Applikation.

Die effiziente Lastverteilung im Netz stellt dabei ein Problem dar, wenn gegen Ende der Verarbeitung einer Aufgabe nur noch wenige Objekte Propagierungen vorliegen haben. Spezielle Lastverteilungsverfahren ('Objekt Verlagerung') können hier jedoch Abhilfe schaffen (Kapitel 7.3.).

5.3. Alternativen

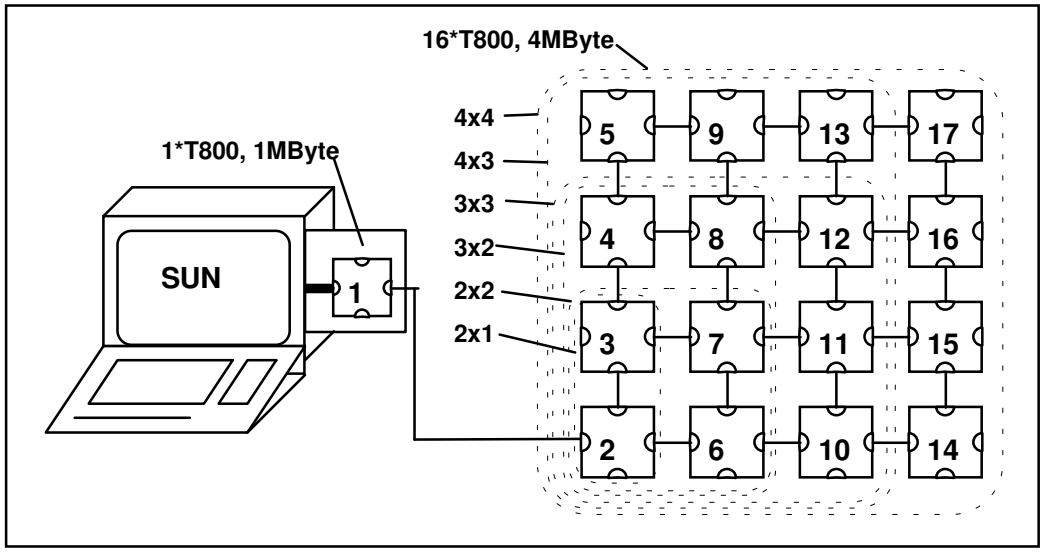
Trifft man eine Design-Entscheidung für die Implementierung auf eng gekoppelten *shared memory*-Multiprozessor-Computern, so spricht im Gegensatz zu dem oben vorgestellten Ansatz viel für eine Parallelisierung der Subsumption, da diese Operation die Grundlage so-

wohl für terminologische als auch für die assertorischen Schlußfolgerungskomponente bildet. Rechner dieses Typs bestehen gewöhnlich aus wenigen Prozessoren (Größenordnung 2-32 Knoten, im Gegensatz zu MIMD-Computern die mehrere tausend Knoten enthalten können), die über einen gemeinsamen Speicher und somit über eine sehr hohe Kommunikationsbandbreite verfügen. In diesem Kontext verliert das Kriterium der Skalierbarkeit an Gewicht, und eine feinere Verteilung gewinnt durch hohe Kommunikationsbandbreiten an Effizienz.

6. Implementierung

6.1. Hardware

Die Wahl für die Implementierung von Parallel-FLEX wurde für einen Parsytec Multicluster II getroffen. Dieses massiv paralleles System besteht aus 16 INMOS T800 Transputern, mit je 4 MByte RAM. Diese T800 Mikroprozessoren gehören der Klasse der RISC-Prozessoren an und wurden vom Design her speziell auf eine möglichst einfache Vernetzbarkeit ausgelegt, so daß der Aufbau von beliebig großen und beliebig strukturierten Parallelcomputersystemen möglich wird, indem man Transputer über sogenannte *Links* zusammenschaltet. Ein *Link* ist eine serielle Hochgeschwindigkeits-Schnittstelle, die zwei Transputer mit 20 MBit/s untereinander verbindet (Punkt zu Punkt Kommunikation). Ein Kommunikationsnetzwerk kommt dadurch zustande, daß jeder Transputer Nachrichten, die nicht für ihn bestimmt sind, an einen geeigneten Nachbarknoten weiterleitet, der näher an dem entsprechenden Ziel liegt. Die auf diese Weise das Netzwerk querenden Nachrichten verursachen nur einen sehr geringen Overhead, da jeder *Link* vor einem eigenständig arbeitenden DMA (*direct memory access*) Interface getrieben wird so daß die Prozessorleistung kaum beeinflußt wird.



Figur 6.1: Die Hardwarekonfiguration (Matrix) des Multicluster II.

Das Bild oben zeigt die physikalische Topologie des verwendeten Parallelrechners (4x4 Matrix). Die gestrichelten Rechtecke deuten dabei 'Subnetze' an, die für Vergleichsmessungen verwendet werden.

Die physikalische Topologie spielt allerdings in der Anwendung unter dem verwendeten Betriebssystem (siehe nächstes Kapitel) keine Rolle, da alle im Netzwerk existierenden Prozesse durch ein transparentes *message passing system* verbunden sind, und da bei der gegebenen Netzwerkgröße die Kommunikationszeiten noch keine Rolle spielen (Kapitel 7.3.1.).

Topologie	2x1	2p1	2x2	3x2	3x3	4x3	4x4
Prozessoren	3	4	5	7	10	13	17
worker	1	2	3	5	8	11	15

Figur 6.2: Die Anzahl der benutzbaren Worker-Prozesse in Abhängigkeit von der Netzwerkgröße.

Bild 6.2 stellt tabellarisch die Netzwerk-Topologie gegen die Anzahl der verwendbaren Knoten (die Namen der Topologie in der ersten Zeile entspricht den Namen der Topologie im vorigen Bild). Die Differenz von 2 nicht nutzbaren Prozessoren ergibt sich aus dem geringen Speicher der einzelnen Knoten: Prozessor 1 (mit 1 MByte RAM) kann nicht einmal den gesamten FLEX-Code laden, während auf Prozessor 2 die 'Shell' lokalisiert ist, die die Generation von Objekten synchronisiert. Im Normalfall wird dieser Prozeß mit einem Worker zusammen auf einen beliebigen Prozessor plaziert, da er an sich keine Rechenzeit beansprucht. Jedoch reicht bei der gegebenen Konfiguration der Speicher nicht mehr für den Worker-Prozeß, so daß die Shell auf einen eigenen Prozessor ausgelagert wurde.

6.2. Brain Aid Prolog

Für die Implementation von FLEX haben wir die Programmiersprache 'Brain Aid Prolog' gewählt. Dabei handelt es sich um ein 'standard' Clocksin & Mellish Prolog [Clocksin Mellish 84] mit parallelen Library-Erweiterungen, so daß sich ein CSP-ähnliches (*communicating sequential processes*) [Hoare 85] Paradigma ergibt. Dabei wird Parallelität explizit ausgedrückt mit den Begriffen 'Prozeß' und 'Message'. Ein 'Prozeß' in BAP entspricht einem gewöhnlichen Prolog Prozeß in einem Multitasking-Betriebssystem. Eine 'Message' ist ein beliebiger Prolog-Term, der zwischen zwei Prozessen ausgetauscht wird. Messages werden gesendet und empfangen mit den send_msg(Dest, Msg) und rec_msg(Sender, Msg) Library-Prädikaten. Message Sender- und Empfänger werden durch ihre Prozeß Id (PID) identifiziert. Ein Transport-Subsystem übernimmt transparent den Transport von Messages, wobei die Reihenfolge erhalten bleibt, wenn mehrere Messages von einem Sender zu demselben Empfänger geschickt werden. Wenn eine Message an ihrem Ziel ankommt, wird sie hinten in der dem Prozeß zugeordneten 'Mail Box' abgelegt.

Aufbauend auf dieser *message passing*-Ebene setzt die *client server*-Ebene auf, die dadurch charakterisiert ist, daß jeder Prozeß eine '*server loop*' ausführt:

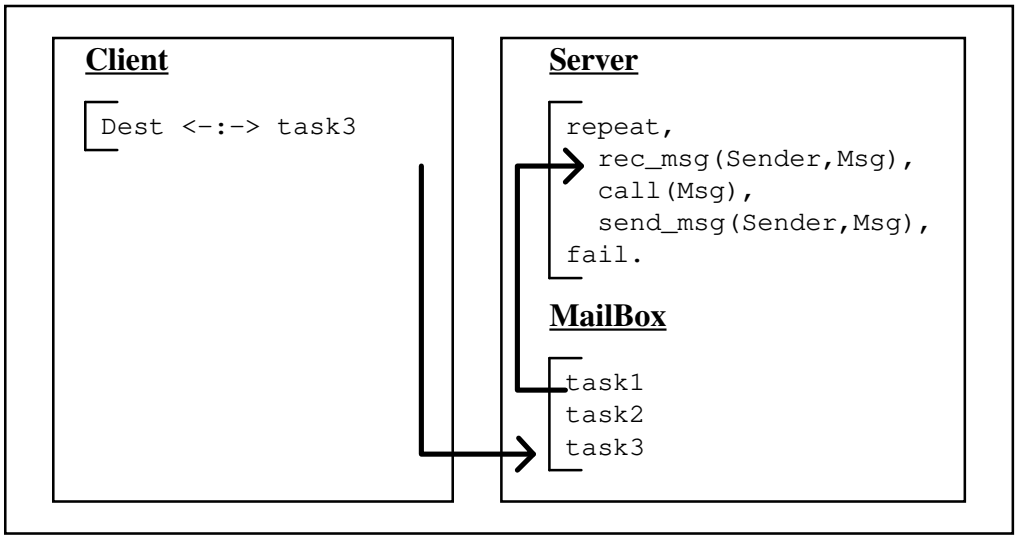


Bild 6.3: Die '*server loop*' und das '*client server*'-Modell.

Dadurch, daß jeder Prozeß (in der *repeat - fail* Schleife) immer empfangsbereit ist[24], ergibt sich ein konsistentes Schema, in dem jeder Prozeß mit jedem anderen kommunizieren kann. Dazu werden die folgenden Primitive definiert, die diese Funktionalität verbergen:

DestProcess :-> Message

Über den unidirektionalen Message Operator ':->' wird eine Message zu einem Prozeß gesendet. Beim empfangenden Prozeß wird die Message in der 'Mail Box' asynchron zwischengespeichert, bis der Prozeß mit der Bearbeitung aller vorigen Messages fertig ist. Der sendende Prozeß braucht nicht auf die Abarbeitung der Message zu warten sondern macht sofort mit dem nachfolgenden Befehl weiter. Deshalb können auch keine Ergebnisse zurückgeliefert werden.

DestProcess <-:-> Message

Über den bidirektionale Message Operator '<-:->' wird eine Message gesendet wie über den ':->' Operator. Jedoch wartet in diesem Fall der Sender solange, bis der Empfänger mit der Bearbeitung der Message fertig geworden ist (und eine Antwort-Message geschickt hat). Enthielt die erste Message freie Variablen, die bei der Bearbeitung der Message gebunden wurden (und deren Wert durch die Antwort zurückgegeben wurde), so werden auf der Senderseite die freien Variablen an die Ergebnisse gebunden und damit diese Werte zurückgegeben. Dieses Schema ähnelt dem RPC (*remote procedure call*) [Nelson 81] Mechanismus aus anderen verteilten Betriebsystemen.

create_process(ProcessorId, ProcessId)

[24]solange nicht während der Ausführung einer Nachricht suspendierende Kommunikation auftritt

create_process(ProcessId)

Erzeugt einen neuen Prozeß (auf dem angegebenen Prozessor) und liefert dessen Nummer (PID) zurück. Wenn kein Prozessor explizit spezifiziert ist, so übernimmt die Standardstrategie zur Verteilung der Netzwerkauslastung die Auswahl.

kill_process(ProcessId)

Löscht einen Prozeß.

Das folgende Beispiel zeigt, wie sich auf diesem Schema noch weitere Konstrukte auf höheren Ebenen aufsetzten lassen. In diesem Programm wird ein `farm(TaskList)` Prädikat definiert, das eine Liste von Termen entgegennimmt und diese Terme als 'Aufträge' parallel ausführt.

```
% ----------------------------------------------------------
% farm(TaskList)(i): distributes the tasks in TaskList
% across the network and returns the resutlts.
% ----------------------------------------------------------
farm(TaskList):-                    % send out tasks
   member(Task,TaskList),           % FOR all tasks..
     create_process(Proc),          % create a new process
     assert(pending(Proc)),         % remember pending task
     Proc :-> (                     % unidirectional message,..
       call(Task),                  % REMOTE: bind variables and..
       assert(result(Task))         % ..store result temporarily
     ),
   fail.
farm(TaskList):-                    % send out tasks
   findall(Proc,retract(pending(Proc)),ProcList),
   farm_collect(ProcList,TaskList)  % collect results and unify..
                                    % ..with input list
farm_collect([],[]):-!.
farm_collect([Proc|ProcList],[Task|TaskList]):-
   Proc <-:-> retract(result(Task)),% get result (bidirect. message)
   kill_process(Proc),              % clean up after execution
   farm_collect(ProcList,TaskList).
```

Ein Möglicher Aufruf für diese Prädikat könnte z.B. so aussehen (wir nehmen an, daß `factorial(N, Fac)` die Variable `Fac` an die Fakultät von `N` bindet:

```
?- farm([factorial(5,A),factorial(6,B),factorial(7,C)]).

      A = 120
      B = 720
      C = 5040
OK
```

Obwohl die Implementation von Parallel-FLEX durch die Art und Weise, wie BAP Parallelität ausdrückt, stark vereinfacht wurde, ist es einfach möglich, dieselben parallelen Techniken auch in anderen Programmiersprachen auszudrücken, wie z.B. LISP oder C. Dies ist insbesondere deshalb ein Vorteil, da das parallele Paradigma auch nach einer Migration, beispielsweise von Prolog nach C, weiterbenutzt werden kann.

6.3. Implementierung der Farm

Dieses Kapitel beschreibt die erste von zwei parallelen Implementationen des FLEX-Systems. Grundgedanke dieser 'Farm' ist die Verteilung von unabhängigen Aufgaben von einem 'Master' zu mehreren 'Worker' Prozessen. Dabei übernimmt der Master sowohl das Management als auch die Synchronisation der Ergebnisse. Die Teilaufgaben entsprechen in diesem Ansatz den einzelnen Propagierungen bzw. Propagierungs-Berechnungen. Ein Listing findet sich in Anhang C.

Voraussetzung für die Bearbeitung der einzelnen Propagierungs-Aufgaben auf den Worker-Prozessen ist das Vorhandensein aller Daten der terminologischen Schlußfolgerungskomponente und einiger Informationen über den Zustand der assertorischen Komponente. Diese Daten werden zusammen mit dem Versenden der Propagierungen zu den Worker gesendet, so daß jeder Worker stets mit der aktuellen 'Terminologie' arbeitet.

Die Erzeugung von neuen Objekten stellt insofern ein Problem dar, als daß der Fall auftreten kann, daß, ausgelöst durch zwei verschiedene 'Tells', zwei Worker unabhängig voneinander versuchen, ein bestimmtes neues Objekt zu erzeugen (siehe Batch-Parallelität in Kapitel 4.1.). Es ist wichtig dies zu verhindern, da sonst eine Inkonsistenz in der assertorischen Schlußfolgerungskomponente auftreten würde, der im weiteren zu fehlerhaftem Systemverhalten führt. Zur Lösung dieses Problems werden die von Workern erzeugte Propagierungen zwischengespeichert, bis die Ergebnisse der noch laufenden Propagierung in den Systemzustand integriert worden sind.

Die folgende Figur zeigt das Ablaufschema des Farm 'Schedulers' (des Verteilerprozesses) unter Berücksichtigung der oben genannten Probleme. Dieser Algorithmus basiert auf den Datenstrukturen zur Implementierung einer Liste von Propagierungen und einer Liste von Workern. Die Propagierungsliste ist mit der initialen Propagierung vorbelegt, die 'Serverliste' enthält die dem Scheduler zugänglichen Worker-Prozesse. Die Begriffe Server und Worker werden im folgenden synonym verwendet, wobei 'Server' der *client server*-Ebene von BAP entstammt und 'Worker' dem Farm-Paradigma.

0. **Start**

1. Sammele alle Propagierungen für ein bestimmtes Objekt aus der Propagierungsliste

2. **Wenn** Propagierungen in der Liste gefunden wurden, sammele alle relevanten Bestandteile des Systemzustands zusammen und schicke den Systemzustand, die Propagierungen und die Beschreibung des betreffenden Objekts zu einem Worker. Der Server wird aus der Serverliste als der zuletzt benutzte ausgewählt (LRU). Wenn kein Server frei ist, warte bis ein beliebiger Server frei geworden ist, schicke diesem Server den neuen Auftrag und integriere dann die soeben empfangen Ergebnisse in den Systemzustand. Ersetze dabei die alte Objektbeschreibung durch die neue und füge aufgetretene Propagierungen an das Ende der Propagierungsliste an.

4. **Sonst** integriere ankommende Ergebnisse in den Systemzustand

5. **Wiederhole** den Vorgang bei 1) solange, bis keine Aufträge mehr in Bearbeitet sind und sich keine Propagierungen mehr in der Propagierungsliste befinden.

6. **Ende**

Die Worker-Prozesse führen den folgenden Algorithmus aus:

0. **Start**

1. **Warte** auf das Eintreffen eines Auftrags

2. Aktualisiere den Zustand der terminologischen Schlußfolgerungskomponente und führe die Propagierungen aus.

3. Sammele die Ergebnisse der Verarbeitung ein und schicke sie zurück zum Sender des Auftrags.

4. **Wiederhole** den Vorgang bei 1).

5. **Ende**

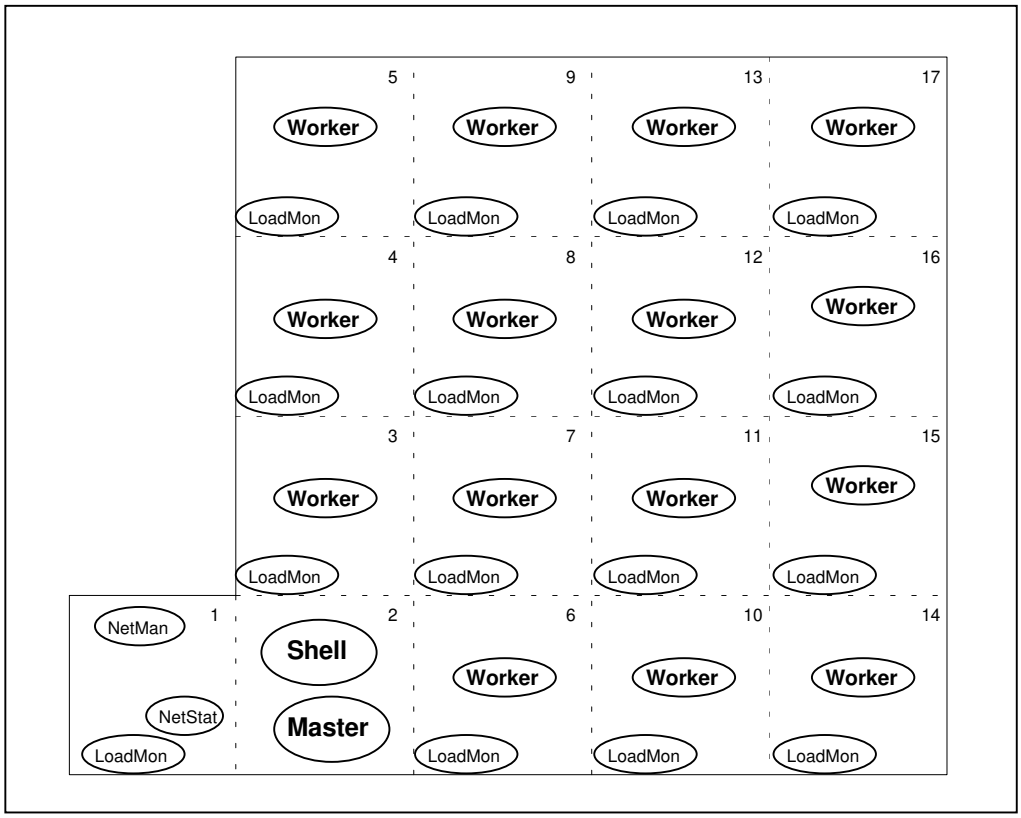


Bild 6.4: Verteilung der Farmprozesse auf eine 4x4 Matrix.

Bild 6.4 zeigt die Verteilung der Prozesse auf die 4x4 Matrix, wobei die 'Shell' und der 'Master' Prozeß auf Prozessor 2 plaziert sind und jeder andere Prozessor innerhalb des Multiclusters je einen Worker-Prozeß beherbergt. Prozessor 1 (auf dem SUN *interface board*) ist nicht geeignet für die Ausführung von FLEX, da er nicht genug Speicher besitzt (1 MByte im Gegensatz zu 4 MByte auf allen anderen Knoten). Statt dessen führt dieser Prozessor Verwaltungsfunktionen aus, wie Lastmanagement und das Führer der globalen Prozeßliste.

Die Kommunikations-Struktur des Gesamtsystems besteht aus zwei voneinander weitgehend unabhängigen Systemen:

6.3.1. Verarbeitungsnetz

Die 'Shell', der 'Master' und die Workerprozesse bilden zusammen das Verarbeitungsnetz, daß die Bearbeitung der FLEX-Aufträge übernimmt. Initiiert werden die Aktionen des Masters durch Benutzer-Eingaben aus dem 'Shell' Prozeß. Diese Shell übernimmt außerdem die Behandlung von Eingaben aus 'Batch-Files', konvertiert alle Eingaben in ein FLEX-internes Format und schickt die sie an den Master. Der Master beherbergt zentral den FLEX-System-Zustand und führt terminologische Schlußfolgerungen selber aus. Informationen die den assertorischen Teil von FLEX betreffen werden als Propagierungen gemäß dem Farm-Algorithmus an die Worker-Prozesses weitergeleitet.
Die wichtigsten Datenstrukturen (außerhalb des eigentlichen FLEX-Systems) sind die Liste aller Worker-Prozesse und die Liste der zu bearbeitenden Propagierungen. Beide Listen befinden sich auf dem Master Prozeß, der damit den Ablauf des assertorischen Schließens steuert.

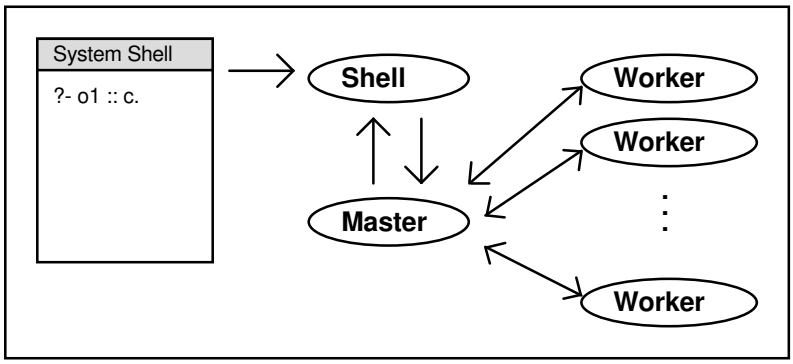


Bild 6.5: Informationsfluß im Verarbeitungsnetz.

6.3.2. Lastüberwachungsnetz:

Die Lastmonitore ('LoadMon') bilden zusammen mit dem Network-Manager ('NetMan') und dem Network-Status-Manager ('NetStat') das Lastüberwachungsnetz. Dieses Netz protokolliert die Aktivität des Netzes und zeigt sie zusammen mit den freien Prozessor-Resourcen in einem speziellen Fenster auf dem Host-Server[25] an. Dazu messen die Lastmonitore in regelmäßigen Abständen die Aktivität auf ihrem Prozessor. Diese Daten senden sie zum Network-

[25]Der 'Host-Server' ist bei Brain Aid Prolog derjenige Computer, der die Schnittstelle zwischen dem Parallelcomputer und dem Benutzer darstellt und die Verwaltung der Ein/Ausgabe übernimmt.

Manager, der eine ausgezeichnete Position im Netz hat und von allen Prozessen zugänglich ist. Dort werden die Lastinformationen gespeichert und vom Network-Status-Manager als Grundlage für die regelmäßigen Statusreports benutzt. Diese Reports dienen zur Suche von Fehlern im FLEX-System, die sich auf die Allokation von Ressourcen oder durch ungleichmäßige Belastung des Netzwerks auswirken. Diese Funktion entspricht dem '*Profiler*' in sequentiellen Systemen.
Die wichtigste Datenstruktur innerhalb des Lastüberwachungsnetzes ist die Liste der Prozessor-Resourcen auf dem Network Manager Prozeß. Diese Struktur enthält für jeden Prozessor die Liste aller Prozesse, den Auslastungsgrad und den freien Speicher.

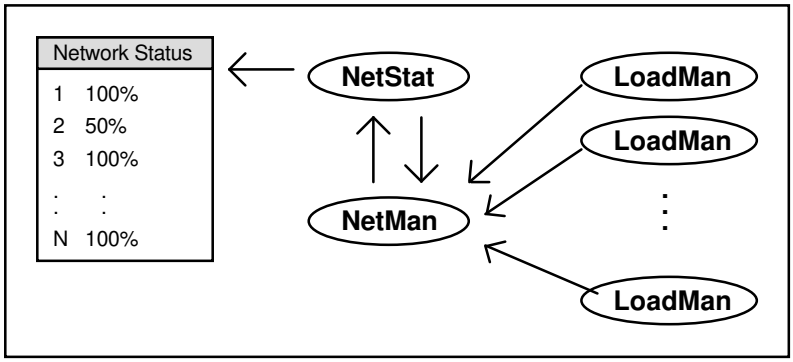


Bild 6.6: Informationsfluß im Lastüberwachungsnetz.

6.4. Implementierung von *Distributed Objects*-FLEX

Im Gegensatz zu der Farm Implementierung besitzt die DO-Implementierung keinen 'zentralen' Zustand der assertorischen Schlußfolgerungskomponente. Während der Zustand der terminologischen Komponente auf einem zentralen 'FLEX-Server' untergebracht ist (und weiter sequentiell ausgeführt wird), ist der assertorische Zustand über alle Prozessoren im Netz verteilt. Diese Situation führt zu einigen unerwünschten Effekten, so z.B. zu der Tatsache, daß es keinen einheitlich 'beobachtbaren' [Bender 82] System-Zustand gibt. Dies hat vor allem Auswirkungen auf die Generierung von neuen Objekten (die nicht doppelt erzeugt werden dürfen) und auf das Terminieren des Algorithmus. Ohne weiter Hilfsmittel ist es nämlich gar nicht möglich festzustellen, ob der Propagierungsvorgang terminiert hat, da zwar einzelne Prozessoren keine Aufträge mehr in Bearbeitung haben können, jedoch während der Zeit, die Nachrichten brauchen, um eine zentrale Instanz (z.B. die Benutzer-Shell) darüber zu benachrichtigen, können von anderen Prozessoren weitere Propagierungen ausgelöst werden.
Um dieses Problem zu lösen wurde die Generierung von neuen Objekten auf den 'FLEX-Server' Prozeß verlagert, der die alleinige Kompetenz zur Erzeugung und Plazierung von Objekten hat. Dieser Prozeß spielt eine besondere Rolle im *distributed objects*-FLEX, da dort auch die terminologische Schlußfolgerungen gezogen werden. Die auftretenden Änderungen werden dann an alle Objekte weitergeleitet, die dann in Bezug auf die Änderung entsprechend 'nachklassifiziert' werden. Dieser Prozeß aber stellt keinen *bottleneck* für die Systemleistung dar, weil Objekterzeugung sehr selten vorkommt und keine Rechenleistung für Schlußfolgerungsberechnungen benötigt. Weiter ist das Zusammenlegen der Generierungsaktivitäten mit der Terminologischen Komponente problemlos, da neu generierte Objekte in jedem Fall klassifiziert werden müssen.

Als weitere Komponente gibt es für Performance-Messungen einen '*load manager*'-Prozeß, der von allen Objektservern die Information über die Auslastung (= Anzahl der wartenden Propagierungen) erhält. Dieser Prozeß ist vor allem für die Feststellung des Terminierens des Propagierungsmechanismus zuständig, indem periodisch getestet wird, ob noch Propagierungen vorliegen. Finden zwischen zwei aufeinanderfolgenden Perioden keine Propagierungen statt, so hat das System einen 'Fixpunkt' (Kapitel 4.2.4.) erreicht und der Algorithmus hat terminiert.[26] Weiter ist dieser Prozeß für die Erkennung ungleichmäßiger Lastzustände zuständig und für die Relokation von Objekten zur besseren Lastverteilung (siehe Kapitel 7.3.).
Das Ziehen der assertorischen Schlußfolgerungen wird von Objekt Servern ausgeführt, die jeweils nur für eine bestimmte Anzahl von Objekten verantwortlich sind und den folgenden Algorithmus ausführen:

```
0.  Start

1.  Warte auf das Eintreffen einer Propagierung.

2.  Führe die Propagierung aus.

3.  Sende entstandene Folgepropagierungen an diejenigen Server, die
    die entsprechenden Objekte verwalten.

4.  Wiederhole den Vorgang bei 1).

5.  Ende
```

Um unter Punkt 3) Propagierungen verschicken zu können, besitzt jeder einzelne Objekt-Server die Liste aller im System befindlichen Objekte mit den Adressen der 'zuständigen' Objekt-Server. Diese 'Objektliste' wird zentral von der Shell verwaltet, wobei Änderungen (z.B. durch die Einführung von neuen Objekten) an alle im System vorhandenen Objektserver weitergegeben werden.

[26]Dabei ist wichtig, daß die Abfrageperioden länger sind als die maximale Nachrichtenlaufzeit innerhalb des Systems.

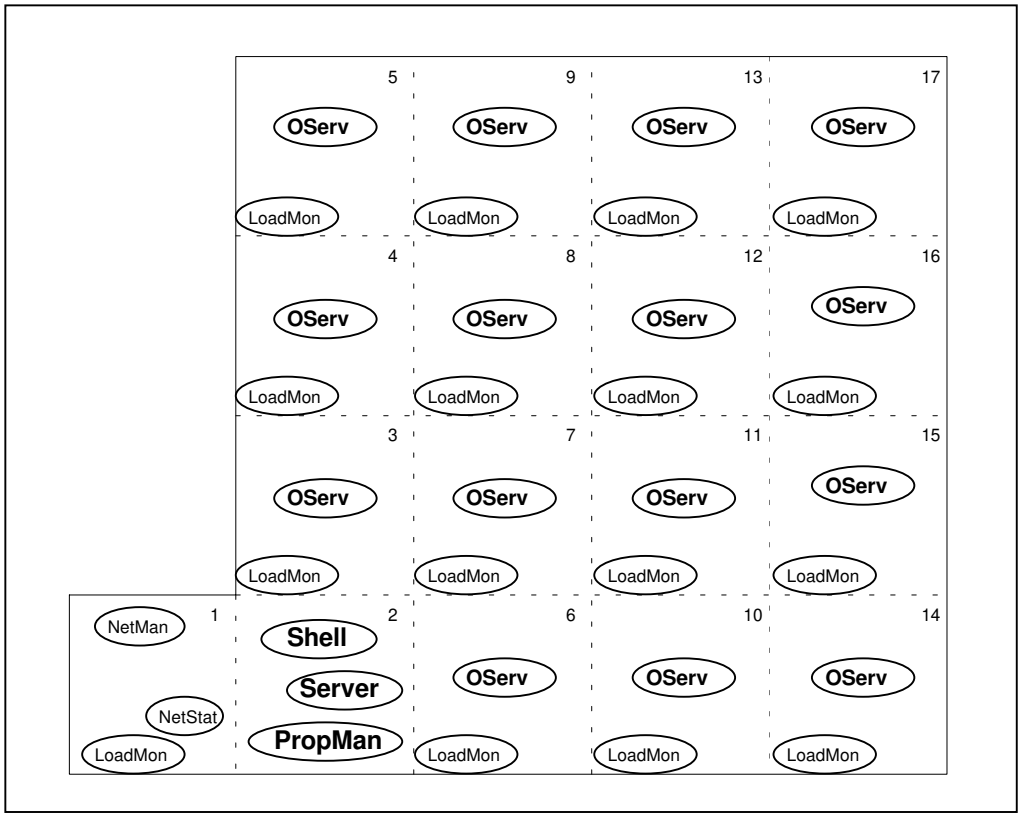


Bild 6.7: Verteilung der Prozesse der DO-Implementierung auf eine 4x4 Matrix.

Die Plazierung der Prozesse ist ähnlich wie bei der Farm, wobei die Shell und der Server auf Prozessor 2 plaziert sind und alle anderen Prozessoren innerhalb des Multiclusters je einen Objekt-Server-Prozeß besitzt. Prozessor 1 (auf dem SUN-Interface-Board) ist nicht geeignet für die Ausführung von FLEX, da er nicht genug Speicher besitzt. Die Kommunikations-Struktur besteht zusätzlich zu dem 'Verarbeitungsnetz' und dem 'Lastüberwachungsnetz' aus dem Propagierungsüberwachungsnetz, das die Anzahl der noch zu bearbeitenden Aufträge der einzelnen Objekt Server enthält.

Verarbeitungsnetz: Die Shell, der FLEX-Server und die Objekt-Server bilden zusammen das Verarbeitungsnetz. Dieses Netz ähnelt dem der Farm-Implementation, jedoch zeigt das folgende Bild den Unterschied, der vor allem in der Kommunikation der Objekt-Server untereinander besteht (im Vergleich: in der Farm kommunizierende die Worker nur mit dem Master). Der Master beherbergt wie bei der Farm den Zustand der terminologischen Schlußfolgerungskomponente, jedoch existieren Kopien dieses Zustandes auf jedem Objekt-Server. Änderungen in der terminologischen Komponente werden (sequentiell) vom Master bearbeitet und dann an alle Objekt-Server geschickt, worauf dann alle im System existierenden Objekte 'nachklassifiziert' werden (in Bezug auf die veränderte terminologische Situation neu klassifiziert werden), woraus wieder Propagierungen folgen können.

Die wichtigste Datenstruktur im Verarbeitungsnetz ist die 'Objektliste', die jedem Objekt die Adresse des zugehörigen Objektservers zuordnet. Diese Liste befindet sich auf allen beteiligten Prozessen und wird vom FLEX-Server (der auch für die Einführung von neuen Objekten zuständig ist) zentral verwaltet.

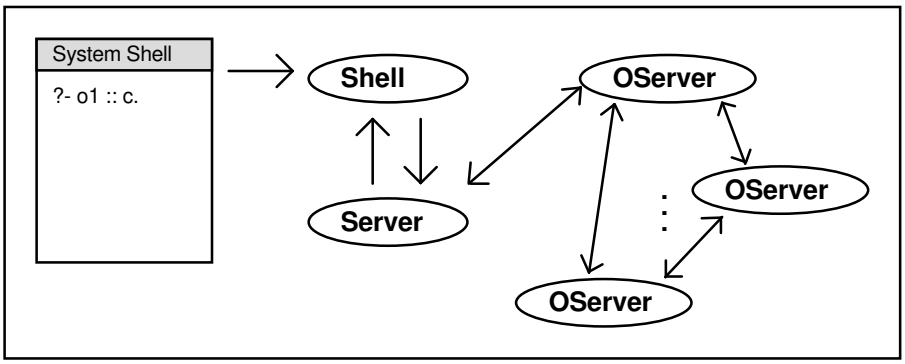


Bild 6.8: Informationsfluß im Verarbeitungsnetz.

Lastüberwachungsnetz: Diese Komponente ist identisch mit dem Lastüberwachungsnetz der Farm Implementierung.

Propagierungsüberwachungsnetz: Dieses Subsystem überwacht die gleichmäßige Auslastung der Objekt Server. Im Gegensatz zur Lastüberwachung beziehen sich diese Lastdaten nicht auf den Zustand des Prozessors sondern auf die Vorhersage der Zeit zur Abarbeitung der aktuell anstehenden Propagierungen. Dies spielt eine besondere Rolle bei der Vermeidung sogenannter 'Nachläufer', d.h. Objekt Server, die noch eine größere Anzahl von Aufträgen zu bearbeiten haben, während der Rest des Netzes bereits fertig ist. Weiter wird durch die Propagierungsüberwachung die Terminierung des Algorithmus festgestellt.

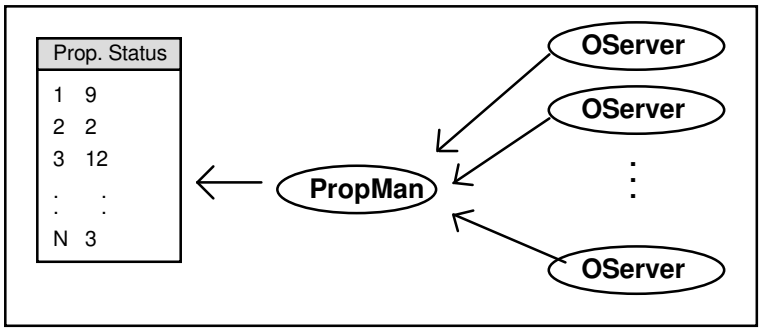


Bild 6.9: Informationsfluß im Propagierungsüberwachungsnetz.

6.5. Schlußbemerkungen zur Implementation

Sowohl bei der Farm- als auch bei der *distributed objects*-Implementierung entstehen komplexe Kommunikations-Strukturen, die verhältnismäßig schwer handhabbar sind. Insbesondere Fehler, die nicht die Funktion sondern nur das Zeitverhalten beeinflussen, sind recht schwer zu finden. Deshalb kommen Entwicklungshilfen wie dem parallelen BAP-'*source level'*-Debugger oder auch dem Lastüberwachungssystem eine noch größere Rolle zu, als dies in sequentiellen Architekturen der Fall ist.

7. Meßergebnisse und Interpretation

7.1. Benchmarks

Das Hauptanwendungsgebiet für das FLEX-System in der KIT-Forschungsgruppe besteht in der Verarbeitung natürlicher Sprachen (NLP - *natural language processing*) [Quantz 93],[Quantz, Schmitz 94]. Ziel bei den Leistungsmessungen an parallelen Implementationen des FLEX-Systems sind deshalb Aussagen zur Parallelisierbarkeit von NLP-Anwendungen.

Leider ergaben sich bereits bei kleineren NLP-Testbeispielen Speicherprobleme bei den auf 4 MByte RAM begrenzten Parallelrechner-Knoten, insbesondere da diese Prozessoren kein virtuelles Speicher-Verwaltungs-System (VMM) besitzen, mit der bei gängigen Workstation-Computern ein größerer (virtueller) Speicher simuliert wird.

Um trotzdem aussagekräftige Messungen machen zu können habe ich den Ablauf verschiedener NLP Anwendungen protokolliert (Anhang A zeigt ein solches Beispiel). Es stellte sich heraus, daß der Ablauf vielfach so verläuft, daß durch einem *object tell* eine Anzahl von Propagierungen erzeugt und dann abgearbeitet werden. Diese Feststellung hat mich zu der Vermutung geführt, daß die Anzahl der Propagierungen vor allem von der Anzahl der Objekte abhängt, die mit dem vom Tell betroffenen Objekt über Rollen verbunden sind, die in Bezug zu der im Tell enthaltenen Information stehen (Konnektivität). Dieses Modell entspricht auch gut dem in Kapitel 3.4. und Kapitel 4.2.4. beschriebenen Ablauf von Propagierung.

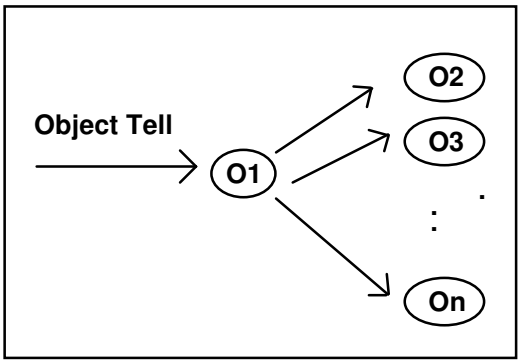


Bild 7.1: Ein '*object tell*' und die mit dem betroffenen Objekt verbunden Objekte.

Dieses Ergebnis habe ich dazu benutzt, einen 'Benchmark-Generator' (Programm zur Erzeugung von 'Geschwindigkeits-Meß-Beispielen') zu erstellen, dessen 'Output' den NLP-Anwendungen in bezug auf die Anzahl der vorkommenden Konzepte, Objekte und der Konnektivität weitgehend entspricht.

```
r :< rtop.          |  o1 :: r:o3 and r:o2 and r:o8.
c1 :< ctop.         |  o2 :: r:o4 and r:o7 and r:o2.
c2 :< ctop.         |  o3 :: r:o7 and r:o2 and r:o1.
c3 :< ctop.         |  o4 :: r:o1 and r:o8 and r:o6.
                    |  o5 :: r:o2 and r:o7 and r:o8.
c1 => all(r,c2).    |  o6 :: r:o1 and r:o7 and r:o5.
c2 => all(r,c3).    |  o7 :: r:o3 and r:o8 and r:o4.
c3 => all(r,c1).    |  o8 :: r:o7 and r:o4 and r:o6.
```

Bild 7.2: Ein Benchmark-Beispiel mit 8 Objekten, 3 Konzepten und einer Konnektivität von 3.

Bild 7.2 zeigt einen solchen Benchmark. In der ersten Zeile (r:< $rtop$) wird eine Rolle r definiert (eine zweistellige Relation, die z.B. eine Beziehung "ist_kind_von" oder "steht_über" entspricht). Die nächsten drei Zeilen (c_X:< $ctop$) definierten drei Konzepte (einstellige Relationen, die z.B. Eigenschaften ausdrücken könne). Mit diesen Konzepten werden in den nächsten Zeilen ($c_X \Rightarrow all(r,c_Y)$) Regeln formuliert, die Sachverhalte ausdrücken können, wie: 'wenn X ein Mensch ist, dann ist dasjenige Objekt, das mit X in einer Beziehung "ist_kind_von" steht, ein Elternteil'.

Im Rest des Beispiels werden dann einzelne Objekte eingeführt, die mit mehreren anderen Objekten in irgendeiner Beziehung stehen. Diese Beziehungen können zyklisch sein, was in dem Bild der 'Elternschaft' keinen Sinn macht, jedoch eine große Anzahl von Propagierungen auslöst (bei geringem Speicherbedarf), was für die Verwendung als Benchmark sehr wichtig ist.

7.2. Die Farm Implementierung

	(seq)	1	2	3	5	8
c10_3_2	(58) (1.0)	78 (0.74)	63 (0.92)	55 (1.05)	56 (1.03)	58 (1.0)
c20_3_2	(177) (1.0)	253 (0.7)	185 (0.95)	159 (1.11)	160 (1.1)	162 (1.09)

Bild 7.3: Ausführungszeiten (Sekunden und relativ) der Farm.

Bild 7.3 zeigt die Ausführungszeiten für die Farm Implementierung. Die erste Zeile enthält die Benchmark-Namen, die aus drei Zahlen bestehen, die die Anzahl der Objekte, Konzepte bzw. die Konnektivität angeben (beispielsweise bedeutet '20_5_3', daß der Benchmark aus 20 Objekten und 5 Konzepten besteht, und daß jedes Objekt mit je drei anderen Objekten verbunden ist). Die folgenden Kolumnen geben die Ausführungszeiten in Abhängigkeit von der Anzahl der Prozessoren wieder, wobei die 'seq' Kolumne die Ausführungszeiten für die nichtparallelisierte Version wiedergibt.

Das ungenügende Ergebnis dieser Implementation läßt sich auf Basis der verhältnismäßig hohen Kosten für die Verteilung des Systemzustands interpretieren. Vor der eigentlichen Bearbeitung einer Propagierung ist es notwendig, große Teile der terminologischen und assertorischen Schlußfolgerungskomponente auf die Worker-Prozesse zu übertragen. Diese Über-

tragung erklärt den Overhead (von ca. 25%) der 1-Prozessor Version gegenüber der sequentiellen Ausführung.
Dagegen ist die geringe Steigerung der Geschwindigkeit bei der Ausführung auf mehreren Prozessoren auf den Arbeitsaufwand zurückzuführen, der bei der Integration der Propagierungsergebnisse in den Systemzustand auf dem Master anfällt. Dieser Arbeitsanteil läuft sequentiell auf dem Master-Prozeß ab, so daß 'Amdahls Gesetz' (Kapitel 2.6.) zum tragen kommt. Aus dem maximalen Speedup von ≈ 1.5 (zwischen der 1-Prozessor-Version und der 3-Prozessor-Version) läßt sich auf ein Verhältnis von $\alpha \approx 0.5$ schließen.

Obwohl es mit Sicherheit einiges Potential für Optimierungen der Farm-Implementation gibt, wurde diese Strategie an dieser Stelle nicht weiterverfolgt und statt dessen mehr Aufmerksamkeit auf die vielversprechendere DO-Implementierung gelegt.

7.3. Die *Distributed Objects*-Implementierung

Bild 7.4 zeigt die Laufzeiten der Benchmark-Suite für der DO-Implementierung. Die Tabelle ist genauso aufgebaut wie Bild 7.3, die Meßgenauigkeit ist aber mit ±2 Sekunden deutlich ungenauer, da das Terminieren des Algorithmus wegen der Probleme der Beobachtbarkeit (siehe Kapitel 6.4.) nur schlecht zu bestimmen ist.

	1	2	3	5	8	11	15
c10_3_2	**59 (1.0)**	**30 (1.9)**	32 (1.8)	18 (3.3)	18	21	18
c10_3_3	**43 (1.0)**	**28 (1.5)**	28 (1.5)	18 (2.4)	16	17	15
c10_3_4	**327 (1.0)**	**159 (2.0)**	141 (2.3)	105 (3.1)	87	74	73
c20_3_2	**179 (1.0)**	**97 (1.8)**	**59 (3.0)**	58 (3.1)	45	37	40
c20_3_3	?	**145**	**129**	58	56	66	51
c20_3_4	?	**240**	**173**	**164**	**70**	74	68
c20_5_3	?	**527**	**355**	**173**	**155**	141	160
c20_5_4	?	**344**	**453**	**411**	**185**	126	189
c40_3_2	?	**314**	**176**	**137**	**105**	111	72
c40_3_3	?		**258**	**231**	**141**	111	87
c40_3_4	?		**569**	**467**	**264**	319	230
c80_3_2	?		**1032**	**665**	**225**	**200**	**181**
c80_3_3	?				**443**	**336**	**266**
c80_3_4	?				**947**	**662**	**583**

Bild 7.4: Ausführungszeiten der DO-Implementation.

Zu bemerken ist, daß das Fehlen der Werte in der linken unteren Hälfte auf Speichermangel zurückzuführen ist. Die Benchmarks sind grob nach Komplexität angeordnet, so daß dem unteren, linken Dreieck große (speicherbedürftige) Benchmarks auf kleinen Parallelrechner-Netzen entsprechen. Diese fehlenden Werte bilden insofern ein besonderes Problem, als daß die sequentiellen Ausführungszeiten die Referenz für die Berechnung des Speedup darstellen, so daß sich dieser Wert nicht berechnen läßt. Zwar lassen sich relative Geschwindigkeitsgewinne auch in der unteren Hälfte berechnen, doch haben diese Zahlen einen relativen Charakter, da

die Bezugsgröße nicht exakt bekannt ist. Das folgende Bild zeigt die so errechneten Speedup-Werte.

	1	2	3	5	8	11	15
c10_3_2	**100**	**193**	181	333	315	272	315
c10_3_3	**100**	**153**	153	238	358	238	268
c10_3_4	**100**	**200**	226	307	366	425	436
c20_3_2	**100**	**180**	**296**	306	388	472	426
c20_3_3		**100**	**112**	250	258	220	284
c20_3_4		**200**	**290**	**300**	**676**	640	705
c20_5_3		**200**	**339**	**697**	**778**	855	753
c20_5_4		**200**	**153**	**169**	**370**	539	365
c40_3_2		**200**	**352**	**452**	**584**	558	849
c40_3_3			**300**	**333**	**545**	691	879
c40_3_4			**300**	**366**	**645**	534	740
c80_3_2			[27]**300**	**464**	**1369**	**1540**	**1710**
c80_3_3					**800**	**1054**	**1332**
c80_3_4					**800**	**1145**	**1300**

Bild 7.5: Relativer Speedup der DO-Implementation.

Einen Ausweg aus dieser Situation läßt sich durch die Einführung der Größe 'Absolute Task-Laufzeit' finden, die unabhängig von der Prozessoranzahl die aktiven Zeiten aller Prozessoren aufsummiert und dadurch 'Nachläufer' aus der Laufzeit herausrechnet. Diese Größe gibt Hinweise auf die Effizienz (bzw. den Overhead) der eigentlichen Parallelisierung, unter Vermeidung der (zufälligen) Verteilung der Aufgaben.

[27]In diesem Fall scheint die Ausführung auf 15 Prozessoren effizienter zu sein als auf 3 Prozessoren. Dies ist auch ohne weiteres möglich, da die Verteilung der Teilaufgaben auf die Prozessoren zufällig erfolgt, so daß auf 3 Prozessoren längere 'Nachläufer' auftreten können, als auf 15 Prozsessoren.

	1	2	3	5	8	11	15
c10_3_2	**59**	**56**	74	61	68	82	68
c10_3_3	**43**	**45**	43	52	65	74	42
c10_3_4	**327**	**310**	302	343	313	312	334
c20_3_2	**179**	**187**	171	235	296	275	342
c20_3_3		**283**	**330**	254	311	421	348
c20_3_4		**452**	**440**	**453**	**448**	453	390
c20_5_3		**1013**	**803**	**708**	**791**	926	931
c20_5_4		**572**	**1110**	**1263**	**1277**	1033	1150
c40_3_2		**608**	**498**	**582**	**551**	727	819
c40_3_3			**670**	**665**	**951**	753	746
c40_3_4			**1631**	**1647**	**1792**	1448	1428
c80_3_2			**3034**	**3277**	**1628**	**1813**	**2205**
c80_3_3					**3327**	**2878**	**3224**
c80_3_4					**5277**	**5171**	**5538**

Bild 7.6: Absolute Task-Laufzeiten (Sekunden) der DO-Implementation.

Allerdings hat diese Größe Probleme bezüglich der Genauigkeit, da sie sich aus den Aktivitätsmessungen des Netzwerks ergibt. Dabei können sich Ungenauigkeiten aufsummieren, so daß beispielsweise in der Zeile 'c10_3_3' die Ausführung auf 15 Prozessoren effizienter zu sein scheint, als auf einem Prozessor, was nicht möglich ist. Trotzdem erklärt diese Tabelle z.B. die Werte in Bild 7.5, Zeile 'c80_3_3', wo auf 11 Prozessoren ein Speedup von 15 gemessen wird, indem starke Ineffizienzen für den Bezugswert auf 3 Prozessoren gemessen werden.

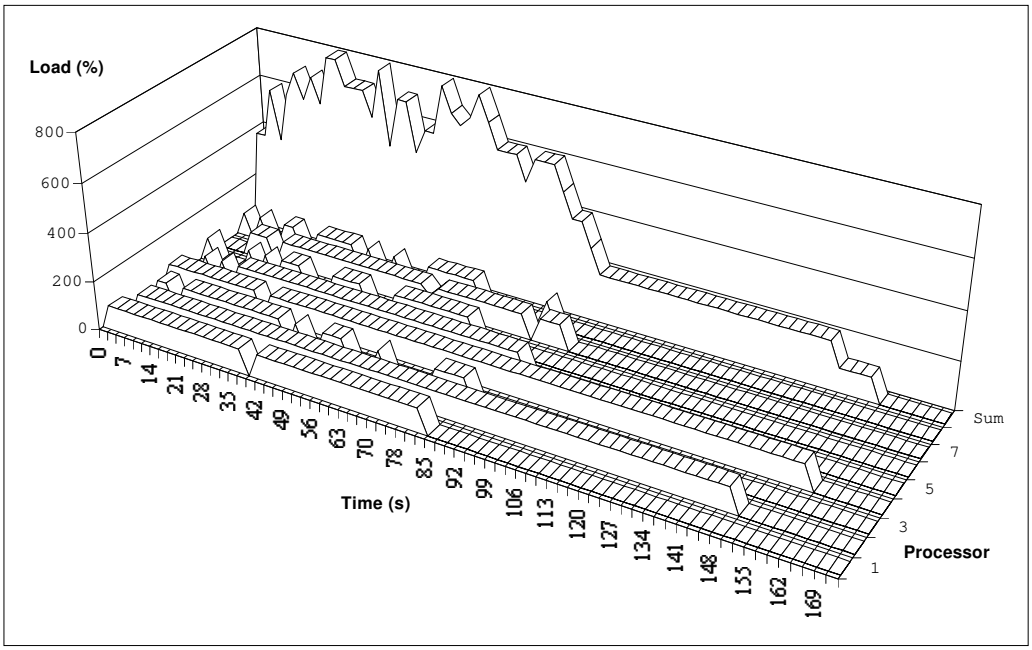


Bild 7.7: Laufzeitverhalten von *distributed object*-FLEX auf einem 3x3 Netzwerk.

Das Absinken der Leistung im oberen, rechten Dreieck (im nicht fettgedruckten Bereich) ist zu einem großen Teil auf 'Nachläufer' zurückzuführen. Das folgende Bild zeigt den Verlauf der Netzwerk-Auslastung während einer Benchmark-Ausführung. Die ersten 8 Reihen zeigen die Arbeitsphasen der einzelnen Prozessoren, während die letzte Reihe die Gesamt-Netzwerkauslastung darstellt. Unter 'Nachläufer' verstehe ich denjenigen oder diejenigen Prozessoren, die als einzige noch längere Zeit laufen, nachdem der Rest des Netzes mit der Bearbeitung des Hauptteils der Aufgaben fertig geworden ist. Bei den Testläufen im oberen, rechten Dreieck bestimmen Nachläufer stark die Gesamtlaufzeit der Benchmarks dadurch, daß verhältnismäßig wenige Aufträge (Propagierungen) auf eine wachsende Zahl von Prozessoren verteilt werden. Deshalb können nur diejenigen Kombinationen von Benchmark und Netzwerk als relevant gelten, bei denen überhaupt genügend Aufträge für das Netzwerk erzeugt werden.

Unter diesen Beschränkungen stimmt das experimentelle Bild der Laufzeiten gut mit dem Warteschlangenmodell in Kapitel 4.2.4. überein. Es ergibt sich für Amdahls Gesetz ein α nahe 1, das nach den theoretischen Überlegungen einer großen Skalierbarkeit des Ansatzes entspricht. Bedingung dafür sind jedoch der Netzgröße entsprechende Anwendungen, die eine hohe Anzahl von Propagierungen erzeugen und damit in der Lage sind, entsprechend viele Prozessoren auszulasten.

Zusammenfassend läßt sich sagen, daß im fettgedruckten Bereich ein sehr befriedigender Speedup feststellen ist. Im Rahmen der gegebenen Meßgenauigkeiten lassen sich Effizienzen von 80% - 99% vermuten. Genauere Aussagen lassen sich wohl nur mit Parallelrechnerknoten mit größerem Hauptspeicher machen. Damit würden sich die Benchmarks in der unteren Hälfte auf 1-Prozessor-Netzen ausführen lassen, so daß die Bezugswerte für die Speedup-Berechnung zugänglich werden.

7.3.1. Skalierbarkeit

Zu einer quantitativen Aussage zur Skalierbarkeit des DO-Ansatzes kommt man, wenn man die endliche Übertragungsbandbreite des Kommunikations-Netzwerkes berücksichtigt. Geht man von einer einfachen Netzwerk-Topologie aus, und nimmt man an, daß Kommunikationsvorgänge statistisch verteilt sind (was durch die zufällige Verteilung von Objekten auf das Netz in erster Näherung erfüllt ist) und gleichverteilt miteinander kommunizieren, so läßt sich folgendermaßen argumentieren:

Zusammenfassung der Annahmen: Jeder Prozessor eines matrixförmigen Netzwerkes (X_0, Y_0) erzeugt mit der Senderate λ Nachrichten einer konstanten Länge L an andere Knoten (X_i, Y_i) mit gleicher Wahrscheinlichkeit.

Behauptung: Die maximale Belastung eines Links im Netzwerk ist:

$$\text{Last} = \lambda * L * \left\langle \begin{matrix} \textit{Anzahl der Wege, die} \\ \textit{durch den Link führen} \end{matrix} \right\rangle$$

Annahme eines Routingalgorithmus:

Das Routing durch das Netzwerks erfolgt durch den Algorithmus des kürzesten Weges: Ist der X-Abstand größer oder gleich dem Y-Abstand so wird ein Schritt in X-Richtung gemacht, sonst in Y-Richtung.

0. **Start**

1. Beginne auf dem sendenden Prozessor.
2. **Wenn** der X-Abstand zum Empfänger größer oder gleich dem Y-Abstand ist, so gehe einen Schritt in X-Richtung, **sonst** in Y-Richtung.
3. **Wiederhole** bei 2) solange, bis der Zielprozessor erreicht ist.

4. **Ende**

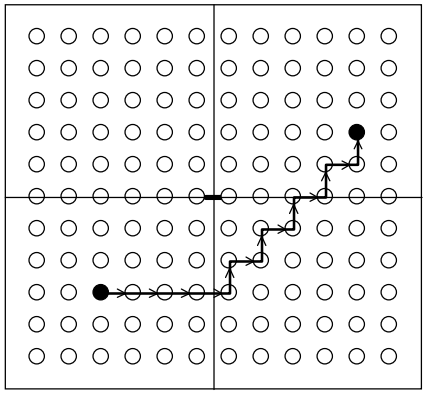


Bild 7.8: *Shortest Path*-Routing-Algorithmus.

Ansatz: Um die Belastung der einzelnen Links im Netz zu berechnen, können wir uns darauf beschränken, die Anzahl der Nachrichten zu zählen, die den gegebenen Link queren, wenn jeder Prozessor einmal eine Nachricht zu jedem anderen sendet (da statistische Gleichverteilung angenommen wurde).

Wir nehmen eine 'Matrix'-Topologie an mit der Breite 2n und der Höhe 2n-1. Diese Topologie eignet sich hinreichend gut als Approximation für eine quadratische Matrix (ein häufiger Fall in der Praxis), läßt sich aber leichter berechnen.

Um die Betrachtungen an dieser Stelle zu vereinfachen, beschränken wir uns auf die Betrachtung des 'maximal belasteten Links' im Netz (*hot spot*). An dieser Stelle fängt die Kommunikation zuerst an, die 'Laufzeit zu dominieren' (der entscheidende Faktor für das Verhalten der Laufzeit zu werden).

Wie leicht zu zeigen ist, befindet sich dieser 'Heiße Punkt' genau in der Mitte des Netzes. Zu diesem Ergebnis kommt man durch eine Abschätzung der Anzahl der Verbindungen, die durch einen Link führen. Betrachtet man Links weiter am Rand, so reduziert sich die Anzahl der einander gegenüberliegenden Prozessoren, deren Kommunikationspfad durch diesen Link kommunizieren kann. Die Form des betrachteten Netzes ([2n] * [2n-1]) hat den Vorteil, daß genau ein zentraler Link existiert.

Weiter nutzen wir die Symmetrien des Netzes aus, indem wir uns auf den Fall von Nachrichten aus dem 3. und 4. Quadranten in den 1. und 2. Quadranten beschränken. Nachrichten innerhalb der linken oder rechten Hälfte brauchen nicht betrachtet zu werden, da diese nicht über den zentralen Link laufen. Die Gesamtbelastung des zentralen Links ergibt dann als 'zweiseitige' Kommunikation des nun zu behandelnden Falles.

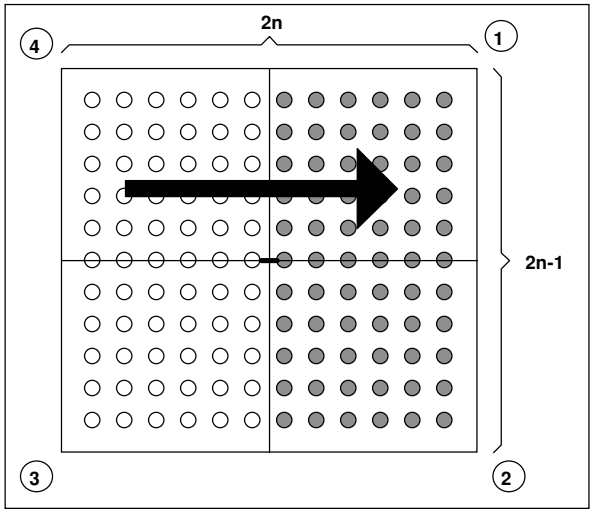


Bild 7.9: Quadrantenaufteilung der Matrix-Topologie.

Um alle Nachrichten zu zählen, die durch den zentralen Link gehen, wenn jeder Prozessor des 3. und 4. Quadranten eine Nachricht an jeden Prozessor des 1. und 2. Quadranten schickt, machen wir eine Fallunterscheidung, in der wir alle Quellen und Ziele unterteilen:

Fall 1: Quelle sind die Prozessoren in der mittleren Halbgerade in der linke Hälfte (entsprechend der negativen X-Achse). Alle Nachrichten, die an Prozessoren innerhalb der rechten Diagonalen liegen (siehe Bild), laufen über den zentralen Link. Es ergeben sich [n] * [n^2] Nachrichten.

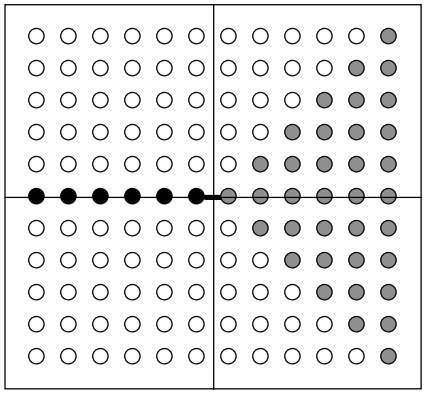


Bild 7.10: Kommunikation der negativen X-Achse in den 1. und 2. Quadranten.

Fall 2: Betrachtet werden alle anderen Prozessoren im 3. Quadranten. Nachrichten von diesen Prozessoren gehen nur dann durch den zentralen Link, wenn sie an Prozessoren der nach oben versetzten Diagonalen im entgegengesetzten Quadranten gerichtet sind. Es ergeben sich (doppelt wegen der vertikalen Symmetrie) 2*[n*(n-1)]*[n-1] Nachrichten.

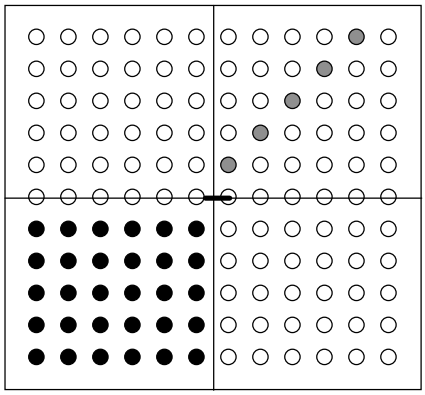


Bild 7.11: Kommunikation vom 3. Quadranten in den 1. Quadranten.

Ergebnis: Als Summe ergeben sich

$$\left\langle \begin{matrix} \textit{Anzahl der Wege, die durch} \\ \textit{den zentralen Link führen} \end{matrix} \right\rangle = [n] * [n^2] + 2*[n*(n-1)]*[n-1] = 3n^3 - 4n^2 + 2n$$

Nachrichten in jede der beiden Richtungen (Symmetrie).

Interpretation: Bei den angenommenen Randbedingungen wie Topologie, Routing Algorithmus und zufälliger Kommunikation ergibt sich eine maximale Kommunikationsbelastung im *hot spot* von $O_{Routing}\left(n^3\right)$ mit der Seitenlänge der Matrix bzw. $O_{Routing}\left(n^{1,5}\right)$ mit der Anzahl der Prozessoren.

Wendet man dieses Ergebnis auf eine Applikation an, die dieselbe - oder eine höhere Ordnung in der Rechen- wie in der Kommunikationskomplexität hat, so ergibt sich ein höherer Exponent in der Gesamt-Kommunikationskomplexität, so daß stets eine obere Schranke für die maximal erreichbare Leistung im Netz besteht. Im Fall der *distributed objects*-Implementation von FLEX wächst der Kommunikationsaufwand konstant mit der Problemgröße (Anzahl der Objekte), da der Datenumfang der einzelnen Propagierungen im wesentlichen gleichbleibt[28], wogegen das Ziehen von Schlußfolgerungen komplexer wird[29]. Diese Überlegungen rechtfertigen die Vermutung, daß die Kommunikationskomplexität langsamer wächst als die Rechenkomplexität.

Nimmt man trotzdem als pessimistische Abschätzung gleiche Komplexitäten an, und setzt man exemplarische Werte für Nachrichtenfrequenz λ, Länge der Nachrichten L und der Kommunikationsbandbreite μ ein:

λ=0.5/s
L=1024 Bytes
μ=1.0MByte/s

so erhalten wir:

$\mu \geq (3n^3-4n^2+4n-2) * L * \lambda$
$\mu/(L*\lambda) = 2000 \geq (3n^3 - 4n^2 + 2n)$
$\Rightarrow 9 \geq n$

$\Rightarrow 18*17 =$ **306 ≥ "Anzahl der Prozessoren in der Matrix"**

Dieser Wert zeigt größenordnungsmäßig die Skalierbarkeitsgrenze der DO-Implementation von FLEX. Jedoch ist diese Grenze recht konservativ abgeschätzt. Weiter ist es möglich, diese Grenze nach oben zu treiben, beispielsweise durch einen Routing Algorithmus nach Bild 7.12, der eine Häufung der Nachrichten im 'Zentrum' des Netzes vermeidet und die Belastung besser verteilt. Genauere Untersuchen in dieser Richtung würden jedoch weit über den Rahmen dieser Arbeit hinausgehen, so daß ich hier darauf verzichte.

[28]Propagierungen übertragen Differenzen von Informationen.
[29]Die Differenz-Information der Proagierungen werden während der Normalisierung mit allen bisher gefundenen Informationen verglichen. Anwendungsabhängig ergibt sich eine Komplexität von n bis n^3.

```
0.  Start
1.  Wenn der sendende Prozeß näher
    am X- als am Y-Rand der Matrix
    liegt dann laufe in Y-Richtung
    solange bis die X-Koordinate des
    Empfängers erreicht ist. Sonst
    laufe in X-Richtung solange bis
    die Y-Koordinate des Empfängers
    erreicht ist.
2.  Laufe entlang der verbleibenden
    Achse zum Empfänger.
4.  Ende
```

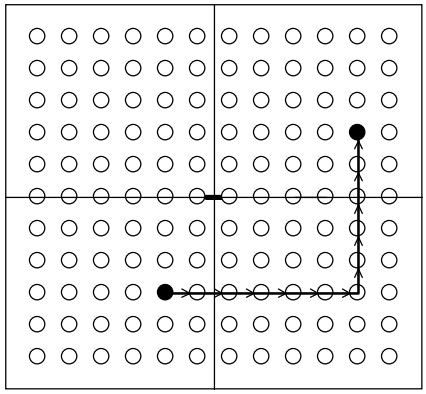


Bild 7.12: Eine alternativer *shortest path*-Routing-Algorithmus mit besserem *hot spot*-Verhalten.

Wichtigstes Ergebnis dieser Abschätzung ist die Abhängigkeit der Skalierbarkeit von den Parametern μ, L und λ unter den angenommenen Bedingungen:

$$\frac{\mu}{L*\lambda} = (3n^3 - 4n^2 + 2n)$$

$$\frac{\mu}{L*\lambda} \approx 3n^3 \qquad \text{(unter Vernachlässigung der } n^2\text{- und n-Glieder)}$$

$$\text{Skalierbarkeit} \sim 4n^2 \qquad \text{(Matrix-Form der Topologie)}$$

$$\text{Skalierbarkeit} \approx 4*\sqrt[\frac{3}{2}]{\frac{\mu}{3L*\lambda}}$$

Damit hängt die Skalierbarkeit mit dem Exponenten 1.5 von dem Verhältnis Rechenzeit/Kommunikation ab.

7.4. Vergleich mit anderen forwärtsverkettenden Systemen

Bringt man die Struktur des FLEX-Systems auf eine kurze Basis, so ergibt sich ein 'über Objekte' vorwärtsverkettendes System. Schlußfolgerungen werden 'an' Objekten gezogen und an andere Objekte weitergeleitet. Ähnliche Strukturen gibt es bei kommerziellen Expertensystemen, wie z.B. Nexpert Object oder G2. Beide Systeme arbeiten regelbasiert auf der Basis von Objekten. Diese Strukturen bieten prinzipiell dasselbe Potential für Parallelisierungen wie FLEX, jedoch mit der Einschränkung, daß dort die Reihenfolge der Ausführung eine Auswirkung auf das Ergebnis haben kann. Weiter werden Regeln in beiden Systemen auch rückwärts angewendet, was nicht von dem hier vorgestellten Schema abgedeckt wird. Es wäre eine interessante Aufgabe, die Anwendbarkeit der hier vorgestellten Techniken auf solche Systeme weiter zu untersuchen.

8. Zusammenfassung und Ausblick

Die Ergebnisse der parallelen Implementation von FLEX sind im Wesentlichen sehr befriedigend. Zwar zeigt sich in der Farm-Implementation, daß dieses Konstrukt nur begrenzt dazu geeignet ist DL-System zu parallelisieren, die DO-Implementierung dagegen zeigt eine große Effizienz und eine große Skalierbarkeit, so daß vermutet werden kann, daß eine Parallelisierung von DL-Systemen mit diesem Paradigma sogar kommerziell interessant ist.

Aufgrund mangelnder Hardware-Ausstattung ist es nicht möglich, exakte Aussagen über das Verhalten von Netzen mit mehr als 8 Prozessoren zu treffen, jedoch lassen die theoretischen Überlegungen auf eine Skalierbarkeit von mehreren Hundert Prozessoren schließen, wobei diese Größe hauptsächlich von dem Kommunikation/Rechenzeit Verhältnis abhängig ist, wie in Kapitel 7.3.1. beschrieben. Dieses Verhältnis scheint für große Applikationen besser zu werden.

Weiter kann ich hier die Vermutung äußern, daß das in dieser Arbeit angesetzte Verfahren der *distributed objects* zur Parallelisierung einer großen Klasse von vorwärtsverkettenden Schlußfolgerungsverfahren geeignet ist. Diese Aussage bleibt noch genauer zu untersuchen, was in einer folgenden Arbeit zu leisten ist.

9. Bibliographie

[Agha 86] G. Agha, 'Actors: A model of concurrent computation in distributed systems', MIT Press, 1986

[Andrews, Schneider 83] G. R. Andrews, F. B. Schneider, 'Concepts and notations for concurrent programming', ACM Computer Surveys, 15(1), 1983

[Amdahl 1967] G. M. Amdahl, 'Validity of Single-Processor Approach to Achieving Large-Scale Computing Capability', Proc. AFIPS Conf., pp. 483-485, Reston, VA., 1967

[Baader et al 92] F. Baader, B. Hollunder, B. Nebel, H. -J. Profitlich, 'An Empirical Analysis of Optimization Techniques for Terminological Representation Systems', KR-92, 270-281

[Bal 90] H. Bal, 'Programming Distributed Systems', Prentice-Hall, 1990

[BAP 93] Brain Aid Systems GbR: F. W. Bergmann, M. Ostermann, G. von Walter, 'Brain Aid Prolog Language Reference', Brain Aid Systems GbR, Aachen, 1993

[Barewise, Perry, 83] J. Barewise, J. Perry, 'Situations and Attitudes', MIT Press, Cambridge, MS, 1983

[Beguelin et al 94] A. Beguelin, E. Seligman, M. Starkey, 'Dome: Distributed object migration environment', Technical Report CMU-CS-94-153, Carnegie Mellon University, Pittsburgh, PA, 1994

[Bender 82] H. Bender, 'Korrekte Zugriffe zu verteilten Daten', (Hrsg.: W. Brauer), Springer Verlag, 1983

[Bevemyr 93] J. Bevemyr, 'A Recursion Parallel Prolog Engine', UPMAIL Technical Report No. 72, Computer Science Department, Uppsala University, Sweden, 1993

[Bergmann 89] F. W. Bergmann, 'Parsytec Prolog 2.0 Reference Manual', Parsytec GmbH, Aachen, 1989

[Bergmann 94] F. W. Bergmann, 'Data Parallelism in Description Logics', TAT' 94, 12-14

[Bond, Gasser 88] A. Bond, L. Gasser, 'Readings in Distributed Artificial Intelligence', Morgan Kaufmann, Los Angeles, CA, 1988

[Brachman et al. 91] R. Brachman, D. L. McGuiness, P. F. Patel-Schneider, L. Alperin Resnick, A. Borgida, 'Living with CLASSIC: When and How to Use a KL-ONE -like Language', in J. Sowa (Ed.), 'Principles of Semantic Networks: Explorations in the Representation of Knowledge', San Mateo: Morgan Kaufmann, 1991, 401-456

[Burge 75] W. H. Burge, 'Stream Processing Functions', IBM Journal of Research and Developement 19(1), pp. 12-25 (Januar 1975)

[Burkhard et al 94] H. Burkhard, A. Bienick, R. Klaus, M. Nölle, G. Schreiber, H. Schulz-Mirbach 'The Parallel Image Processing System PIPS' in R. Flieger, R. Grebe (eds.), Parallelrechner Grundlagen und Anwendung, IOS Press, Amsterdam, Netherlands, 1994, 288-293

[Clark et al 84] K. Clark, Keith, S. Gregory, 'PARLOG: Parallel Programming in Logic' DOC Research Report 84/4, Departement of Computing, Imperial College of Science and Technology, 1984

[Clocksin, Mellish 84] W. F. Clocksin, C. S. Mellish, 'Programming in Prolog', Springer Verlag, 1984

[Conery et al 83] J. S. Conery, 'The AND/OR Process Model for Parallel Interpretation of Logic Programs', Information and Computer science Technical Report 204, University of California, Irvine, 1983

[Culler et al 93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken, 'LogP: Towards a Realistic Model of Parallel Computation', in Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, 1993

[Cunha et al 89] J. A. Cunha, M. C. Ferreira, L. M. Perreira, 'Programming in Delta Prolog', in Proceedings of the International Conference on Logic Programming, 1989

[Diekmann et al 94] R. Diekmann, D. Meyer, B. Monien, 'Parallele Partitionierung unstrukturierter Finite Elemente Netze auf Transputernetzwerken' in R. Flieger, R. Grebe (eds.), Parallelrechner Grundlagen und Anwendung, IOS Press, Amsterdam, Netherlands, 1994, 317-326

[Dijkstra 68] E. W. Dijkstra, "Cooperating sequential processes", in Programming Languages, F. Genuys (ed.), Academic Press, New York, NY, 1968

[Donini et al. 91a] F. M. Donini, M. Lenzerini, D. Nardi, W. Nutt, 'The Complexity of Concept Languages', KR'91, 151-162

[Donini et al. 91b] F. M. Donini, M. Lenzerini, D. Nardi, W. Nutt, 'Tractable Concept Languages', IJCAI-91, 458-463

[Dossier 91] A. C. Dossier, 'Intelligence Artificielle Distribuee', Bulletin de l'AFIA 6, 1991

[Dworatschek 73] S. Dworatschek, 'Einführung in die Datenverarbeitung', de Gruyter, Berlin 1973

[Enslow 74] P. H. Enslow, Jr., (ed.) 'Multiprocessors and Parallel Processing', John Wiley & Sons, New York, 1974

[Eswaran et al 76] K. Eswaran, et al, 'The notions of consistency and predicate locks in a database system'. CACM, 19, 11, Nov. 1976, 624-633

[Færgemand, Olsen 94] O. Færgemand, A. Olsen, 'Introduction to SDL-92', Computer Networks and ISDN Systems 26 1994, p. 1143-1167.

[Flynn 72] M. J. Flynn, 'Some computer organizations and their effectiveness', IEEE Transactions on Computers, C-21(9), p 948-960, 1972

[Freeman 75] P. Freeman, 'Software Systems Principles: A Survey', Science Research Associates Inc., 1975

[Geist et al 94] A. Geist et al, 'PVM 3 User's Guide and Reference Manual', Engineering Physics and Mathematics Division, Oak Ridge National Laboratory, USA, 1994

[Gentzen 34] G. Gentzen, 'Untersuchungen über das logische Schließen', Math. Zeitschrift 39, 176-210, 405-431, 1965

[De Giacomo, Lenzerini 94] G. De Giacomo, M. Lenzerini, 'On the correspondence between description logics and logics of programs' in International Workshop on Description Logics, DFKI Report D-94-10, 1994

[Gurd et al 85] J.R. Gurd, C.C. Kirkham, I. Watson, 'The Manchester prototype dataflow computer', Comm. of the ACM, 28 (1), New York, Seite 24 ff

[Hansen 77] P. B. Hansen, 'The Architecture of Concurrent Programs', Prentice Hall, Inc., Englewood Cliffs, NJ, 1977

[Hayes 77] P. H. Hayes, 'In Defense of Logic', IJCAI-77, 559-565

[Hayes 80] P. H. Hayes, 'The Logic of Frames' in D. Metzing (ed.), 'Frame Conceptions and Text Understanding', de Gruyter, Berlin, 1980, S. 46-61

[Hoare 80] C. A. R. Hoare, 'A Model for Communicating Sequential Processes', Prepring 80-1, Department of Computer Science, University of Wollongong, 1980

[Hoare 85] C. A. R. Hoare, 'Communicating Sequential Processes', Prentice Hall, Englewood Cliffs, N.J., USA, 1985

[Hoppe et al. 93] T. Hoppe, C. Kindermann, J. J. Quantz, A. Schmiedel, M. Fischer, 'Back V5 Tutorial & Manual', KIT Report 100, Technische Universität Berlin, 1993

[Jensen 92] K. Jensen, 'Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use', Volume 1, Basic Concepts, EATCS Monographs on Theoretical Computer Science Springer-Verlag, Berlin, 1992.

[Kale 94] L. V. Kale, 'A Tutorial Introduction to Charm', Department of Computer Science, University of Illinois at Urbana Champaign

[Kindermann 95] C. Kindermann, 'Verwaltung assertorischer Inferenzen in terminologischen Wissensbanksystemen', PhD Thesis (submitted), TU Berlin, 1995

[Lamport 76] L. Lamport, 'Time, clocks and the ordering of events in a distributed system', Mass. Comp. Ass., CA-7601-2911, March 1976, CACM, 21, 7, July 1978, 558-565

[Landin 65] P. J.. Landin, 'The Correspondence Between ALGOL 60 and Church's Lambda Notation: Part 1' in Communication sof the ACM 8(2), pp. 89-101 (Februar 1965).

[Lann 77] G. Le Lann, 'Distributed Systems - towards a formal approach', Proc. IFIP 77, Norht Holland, 1977, 155-160

[Lourenço, Borges 91] J. M. Lourenço, J. L. Borges, 'Ambiente de execuçao de processos Prolog para arquitecturas baseadas em Transputers', Relatório Técnico, RT-63/91, Departamento de Informática, Universidade Nove de Lisboa, Portugal, 1991

[LSC 88] Logical Systems, 'C Reference Manual', Logical Systems, 1988

[MacGregor 91] R. MacGregor, 'Using a Description Classifier to Enhance Deductive Inference' in 'Proceedings of the Seventh IEEE Conference on AI Applications', 141-147, Miami, FL, 1991

[Mattos 85] P. Mattos, 'Program design for concurrent systems', INMOS technical notes 23, INMOS Limited

[May, Shepherd 85] D. May, R. Shepherd, 'Communicating Processes Computers', INMOS technical notes 5, INMOS Limited

[Minsky 75] M. Minsky, 'A Framework for Representing Knowledge' in P. H. Winston (ed.), 'The Psychology of Computer Vision', McGraw-Hill, NY, 1975, S. 211-277

[Moto-Oka 81] T. Moto-Oka (ed.), 'Fifth Generation Computer Systems', Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, Japan, Oktober 19-22, 1981

[MPI 94] Message Passing Interface Forum, 'MPI: A message-passing interface standard', Technical Report CS-94-230, University of Tennessee, Knoxville, TN, 1994

[Nelson 81] B. Nelson, 'Remote Procedure Calls', Technical Report CSL-81-9, Xerox Parc, Palo Alto, 1981

[Occam 88] INMOS Limited, 'Occam 2 Reference Manual', 1988

[Owsnicki-Klewe 88] B. Owsnicki-Klewe, 'Configuration as a Consistency Maintenance Task', in W. Hoeppner (Ed.), Proceedings of GWAI'88, Berlin: Springer, 1988, 77-87

[Packer 85] J. Packer, 'Exploiting Concurrency; A ray tracing example', INMOS technical notes 7, INMOS Limited

[Pereira 83] F. Pereira, ed. 'C-Prolog User's Manual', Departement of Artificial Intelligence, Edinburgh, 1983

[Pereira 86] L. M. Pereira, L. Monteiro, J. Cunha, J. N. Aparicio, 'Delta Prolog: A distributed backtracking extension with events', in Third international Conference on Logic Programming, p. 69-83, 1986

[Pontelli, Gupta 94] E. Pontelli, G. Gupta, 'Design and Implementation of Parallel Logic Programming Systems', Proceedings of ILPS'94 Post Converence Workshop, 1994

[Prawitz 65] D. Prawitz, 'Natural Deduction', Alquist & Wiksell, Uppsala

[Quantz 93] J.J. Quantz, 'Interpretation as Exception Minimization', IJCAI-93, 1310-1315

[Quantz 95] J. J. Quantz, 'Preferential Disambiguation in Natural Language Processing', PhD Thesis (submitted), Fachbereich 13 - Informatik, Technische Universität Berlin, 1995

[Quantz, Schmitz 94] J.J. Quantz, B. Schmitz, 'Knowledge-Based Disambiguation for Machine Translation', Minds and Machines 4, 39-57, 1994

[Raina 93] Raina, 'Emulation of a Virtual Shared Memory Architecture', PhD Thesis, Faculty of Engineering, University of Bristol, 1993

[Quillian 68] M. R. Quillian, 'Semantic Memory' in M. Minsky (ed.), 'Semantic Information Processing', MIT Press, Cambridge, MS, 1968, S. 216-270

[Royer, Quantz 92] V. Royer, J.J. Quantz, 'Deriving Inference Rules for Terminological Logics', in D. Pearce, G. Wagner (eds), Logics in AI, Proceedings of JELIA'92, Berlin: Springer, 1992, 84-105

[Royer, Quantz 93] V. Royer, J. J. Quantz, 'Deriving Inference Rules for Description Logics: a Rewriting Approach into Sequent Calculi', Kit Report 111, Technische Universität Berlin, 1993

[Royer, Quantz 94] V. Royer, J.J. Quantz, 'On Intuitionistic Query Answering in Description Bases', in A. Bundy (Ed.), CADE-94, Berlin: Springer, 1994, 326-340

[Rozier, Martins 87] M. Rozier, J. L. Martins, 'Distributed Operating Systems: Theory and Practice', Springer Verlag, Berlin, 1987

[Schoinas 91] G. Schoinas, 'POSYBL: Implementing the Blackboard Model in a Distributed Memory Environment Using Linda' in G. Wilson (ed.), 'Linda-Like Systems and their Implementation', Technical Report 91-13, S.105-116, Edinburgh Parallel Computing Centre, 1991

[Shapiro 83] E. Y. Shapiro, 'A subset of Concurrent Prolog and its implementation', Technical Manual TR-003, Institute for New Generation Computer Technology, 1983

[Schaerf 94] A. Schaerf, 'Query Answering in Concept-Based Knowledge Representation Systems: Algorithms, Complexity, and Semantic Issues', Dissertation Thesis, Dipartimento di Informatica e Sistemistica, Università di Roma 'La Sapienza', 1994

[Strietzel 94] Strietzel, 'Large Eddy Simulation turbulenter Strömungen auf MIMD-Systemen', in R. Flieger, R. Grebe (eds), Parallelrechner Grundlagen und Anwendung, IOS Press, Amsterdam, Netherlands, 1994, 357 - 366

[Ueda 87] K. Ueda, 'Guarded Horn Clauses' in Ehud Hapiro (ed.) 'Concurrent Prolog', Volume 1 p. 140-156, MIT Press, 1987

[Warren 88] D. H. Warren, 'Implementations of Prolog', in International Conference and Symposium on Logic Programming, Tutorial No. 3, Seattle, WA, 88

[Welsh McKeag 81] J. Welsh, M. McKeag, 'Strukturierte Systemprogrammierung mit Pascal Plus', Carl Hanser Verlag, 1981

[Wise 82] M. J. Wise, 'A Parallel Prolog: The Construction of a Data Driven Model', ACM Symposium on LISP and Functional Programming, Pittsburg, PA, 1982, S.56-66

[Woods 75] W. A. Woods, 'What's in A Link: Foundations for Sematnic Networks' in D. G. Bobrow, A. M. Collins (eds.), 'Elements of Discourse Understanding', Cambridge University Press, 1981, S.300-334

[Young 87] M. Young et al, 'The duality of memory and communication in the implementation of a multiprocessor operating system', ACM, 1987

Anhang

Anhang A: Trace einer NLP-Anwendung

```
| ?- test(ref1, _).

"oh ja nach meinem terminkalender wie waere es im oktober"

OBJECT TRACE time(17, 39, 58) : obj_propagate: 0 Entries in [add([], nf(t, c, [
OBJECT TRACE time(17, 39, 58) : obj_propagate: 1 Entries in [add(obj_1, nf(t, c
OBJECT TRACE time(17, 39, 59) : obj_propagate: 0 Entries in [add(obj_1, cname(s
OBJECT TRACE time(17, 40, 00) : obj_propagate: 0 Entries in []
tell succeded : obj_1::(anything and anything and anything)and fillers(cond, []

OBJECT TRACE time(17, 40, 2) : obj_propagate: 1 Entries in [add(ex_ja, nf(t, c,
OBJECT TRACE time(17, 40, 2) : obj_propagate: 2 Entries in [add(ex_ja, cname(ke
OBJECT TRACE time(17, 40, 2) : obj_propagate: 1 Entries in [add(obj_1, nf(t, c,
OBJECT TRACE time(17, 40, 3) : obj_propagate: 0 Entries in [add(obj_1, cname(si
OBJECT TRACE time(17, 40, 5) : obj_propagate: 0 Entries in []
tell succeded : obj_1::(anything and anything and keyword:ex_ja)and fillers(con

OBJECT TRACE time(17, 40, 6) : obj_propagate: 0 Entries in []
tell succeded : obj_1::(anything and anything and anything)and fillers(cond, []

OBJECT TRACE time(17, 40, 6) : obj_propagate: 0 Entries in []
tell succeded : obj_2::monate_c in ext_sit0

OBJECT TRACE time(17, 40, 09) : obj_propagate: 4 Entries in [add(mofy, nf(t, c,
OBJECT TRACE time(17, 40, 10) : obj_propagate: 5 Entries in [add(obj_2, nf(t, c
OBJECT TRACE time(17, 40, 10) : obj_propagate: 6 Entries in [add(obj_2, cname(o
OBJECT TRACE time(17, 40, 10) : obj_propagate: 5 Entries in [add(obj_2, cname(o
OBJECT TRACE time(17, 40, 10) : obj_propagate: 4 Entries in [add(obj_2, cname(o
OBJECT TRACE time(17, 40, 10) : obj_propagate: 3 Entries in [add(obj_3, nf(t, c
OBJECT TRACE time(17, 40, 12) : obj_propagate: 2 Entries in [add(obj_3, cname(s
OBJECT TRACE time(17, 40, 13) : obj_propagate: 1 Entries in [add(obj_3, nf(t, c
OBJECT TRACE time(17, 40, 14) : obj_propagate: 0 Entries in [add(obj_3, cname(s
OBJECT TRACE time(17, 40, 15) : obj_propagate: 0 Entries in []
tell succeded : obj_3::designator:10 and pred:mofy and ref:obj_2 in ext_sit0

OBJECT TRACE time(17, 40, 19) : obj_propagate: 10 Entries in [add(aux_sein, cna
OBJECT TRACE time(17, 40, 19) : obj_propagate: 9 Entries in [add(vorf_wie, cnam
OBJECT TRACE time(17, 40, 19) : obj_propagate: 8 Entries in [add(wie_waers, cna
OBJECT TRACE time(17, 40, 19) : obj_propagate: 7 Entries in [add(obj_3, nf(t, c
OBJECT TRACE time(17, 40, 20) : obj_propagate: 8 Entries in [add(aux_sein, nf(t
OBJECT TRACE time(17, 40, 20) : obj_propagate: 9 Entries in [add(vorf_wie, nf(t
OBJECT TRACE time(17, 40, 21) : obj_propagate: 10 Entries in [add(wie_waers, nf
OBJECT TRACE time(17, 40, 21) : obj_propagate: 11 Entries in [add(int, nf(t, c,
OBJECT TRACE time(17, 40, 21) : obj_propagate: 12 Entries in [add(conj, nf(t, c
OBJECT TRACE time(17, 40, 21) : obj_propagate: 13 Entries in [add(int, cname(co
OBJECT TRACE time(17, 40, 21) : obj_propagate: 12 Entries in [add(conj, cname(c
OBJECT TRACE time(17, 40, 21) : obj_propagate: 11 Entries in [add(obj_1, nf(t,
OBJECT TRACE time(17, 40, 23) : obj_propagate: 17 Entries in [add(obj_1, cname(
OBJECT TRACE time(17, 40, 24) : obj_propagate: 23 Entries in [add(obj_1, nf(t,
OBJECT TRACE time(17, 40, 26) : obj_propagate: 29 Entries in [add(obj_1, cname(
OBJECT TRACE time(17, 40, 28) : obj_propagate: 35 Entries in [add(obj_1, nf(t,
OBJECT TRACE time(17, 40, 29) : obj_propagate: 41 Entries in [add(obj_1, cname(
OBJECT TRACE time(17, 40, 31) : obj_propagate: 47 Entries in [add(obj_1, nf(t,
OBJECT TRACE time(17, 40, 33) : obj_propagate: 53 Entries in [add(obj_1, cname(
OBJECT TRACE time(17, 40, 34) : obj_propagate: 59 Entries in [add(obj_1, nf(t,
OBJECT TRACE time(17, 40, 36) : obj_propagate: 65 Entries in [add(obj_1, cname(
OBJECT TRACE time(17, 40, 38) : obj_propagate: 71 Entries in [add(obj_1, nf(t,
OBJECT TRACE time(17, 40, 39) : obj_propagate: 77 Entries in [add(obj_1, cname(
OBJECT TRACE time(17, 40, 41) : obj_propagate: 83 Entries in [add(ex_ja, cname(
OBJECT TRACE time(17, 40, 41) : obj_propagate: 82 Entries in [add(vorf_wie, cna
OBJECT TRACE time(17, 40, 41) : obj_propagate: 81 Entries in [add(wie_waers, cn
```

```
OBJECT TRACE time(17, 40, 41) : obj_propagate: 80 Entries in [add(obj_3, nf(t,
OBJECT TRACE time(17, 40, 43) : obj_propagate: 79 Entries in [add(ex_ja, nf(t,
OBJECT TRACE time(17, 40, 43) : obj_propagate: 78 Entries in [add(vorf_wie, nf(
OBJECT TRACE time(17, 40, 43) : obj_propagate: 77 Entries in [add(wie_waers, nf
OBJECT TRACE time(17, 40, 43) : obj_propagate: 76 Entries in [add(ex_ja, cname(
OBJECT TRACE time(17, 40, 43) : obj_propagate: 75 Entries in [add(vorf_wie, cna
OBJECT TRACE time(17, 40, 43) : obj_propagate: 74 Entries in [add(wie_waers, cn
OBJECT TRACE time(17, 40, 43) : obj_propagate: 73 Entries in [add(obj_3, nf(t,
OBJECT TRACE time(17, 40, 45) : obj_propagate: 72 Entries in [add(ex_ja, nf(t,
OBJECT TRACE time(17, 40, 45) : obj_propagate: 71 Entries in [add(vorf_wie, nf(
OBJECT TRACE time(17, 40, 45) : obj_propagate: 70 Entries in [add(wie_waers, nf
OBJECT TRACE time(17, 40, 45) : obj_propagate: 69 Entries in [add(ex_ja, cname(
OBJECT TRACE time(17, 40, 45) : obj_propagate: 68 Entries in [add(vorf_wie, cna
OBJECT TRACE time(17, 40, 45) : obj_propagate: 67 Entries in [add(wie_waers, cn
OBJECT TRACE time(17, 40, 45) : obj_propagate: 66 Entries in [add(obj_3, nf(t,
OBJECT TRACE time(17, 40, 47) : obj_propagate: 65 Entries in [add(ex_ja, nf(t,
OBJECT TRACE time(17, 40, 47) : obj_propagate: 64 Entries in [add(vorf_wie, nf(
OBJECT TRACE time(17, 40, 47) : obj_propagate: 63 Entries in [add(wie_waers, nf
OBJECT TRACE time(17, 40, 47) : obj_propagate: 62 Entries in [add(ex_ja, cname(
OBJECT TRACE time(17, 40, 47) : obj_propagate: 61 Entries in [add(vorf_wie, cna
OBJECT TRACE time(17, 40, 47) : obj_propagate: 60 Entries in [add(wie_waers, cn
OBJECT TRACE time(17, 40, 47) : obj_propagate: 59 Entries in [add(obj_3, nf(t,
OBJECT TRACE time(17, 40, 49) : obj_propagate: 58 Entries in [add(ex_ja, nf(t,
OBJECT TRACE time(17, 40, 49) : obj_propagate: 57 Entries in [add(vorf_wie, nf(
OBJECT TRACE time(17, 40, 49) : obj_propagate: 56 Entries in [add(wie_waers, nf
OBJECT TRACE time(17, 40, 49) : obj_propagate: 55 Entries in [add(ex_ja, cname(
OBJECT TRACE time(17, 40, 49) : obj_propagate: 54 Entries in [add(vorf_wie, cna
OBJECT TRACE time(17, 40, 49) : obj_propagate: 53 Entries in [add(wie_waers, cn
OBJECT TRACE time(17, 40, 49) : obj_propagate: 52 Entries in [add(obj_3, nf(t,
OBJECT TRACE time(17, 40, 51) : obj_propagate: 51 Entries in [add(ex_ja, nf(t,
OBJECT TRACE time(17, 40, 51) : obj_propagate: 50 Entries in [add(vorf_wie, nf(
OBJECT TRACE time(17, 40, 51) : obj_propagate: 49 Entries in [add(wie_waers, nf
OBJECT TRACE time(17, 40, 51) : obj_propagate: 48 Entries in [add(ex_ja, cname(
OBJECT TRACE time(17, 40, 51) : obj_propagate: 47 Entries in [add(vorf_wie, cna
OBJECT TRACE time(17, 40, 51) : obj_propagate: 46 Entries in [add(wie_waers, cn
OBJECT TRACE time(17, 40, 51) : obj_propagate: 45 Entries in [add(obj_3, nf(t,
OBJECT TRACE time(17, 40, 53) : obj_propagate: 44 Entries in [add(ex_ja, nf(t,
OBJECT TRACE time(17, 40, 53) : obj_propagate: 43 Entries in [add(vorf_wie, nf(
OBJECT TRACE time(17, 40, 53) : obj_propagate: 42 Entries in [add(wie_waers, nf
OBJECT TRACE time(17, 40, 53) : obj_propagate: 41 Entries in [add(ex_ja, cname(
OBJECT TRACE time(17, 40, 53) : obj_propagate: 40 Entries in [add(vorf_wie, cna
OBJECT TRACE time(17, 40, 53) : obj_propagate: 39 Entries in [add(wie_waers, cn
OBJECT TRACE time(17, 40, 53) : obj_propagate: 38 Entries in [add(obj_3, nf(t,
OBJECT TRACE time(17, 40, 55) : obj_propagate: 36 Entries in [add(vorf_wie, nf(
OBJECT TRACE time(17, 40, 55) : obj_propagate: 35 Entries in [add(wie_waers, nf
OBJECT TRACE time(17, 40, 55) : obj_propagate: 34 Entries in [add(ex_ja, cname(
OBJECT TRACE time(17, 40, 55) : obj_propagate: 33 Entries in [add(vorf_wie, cna
OBJECT TRACE time(17, 40, 55) : obj_propagate: 32 Entries in [add(wie_waers, cn
OBJECT TRACE time(17, 40, 55) : obj_propagate: 31 Entries in [add(obj_3, nf(t,
OBJECT TRACE time(17, 40, 57) : obj_propagate: 30 Entries in [add(ex_ja, nf(t,
OBJECT TRACE time(17, 40, 57) : obj_propagate: 29 Entries in [add(vorf_wie, nf(
OBJECT TRACE time(17, 40, 57) : obj_propagate: 28 Entries in [add(wie_waers, nf
OBJECT TRACE time(17, 40, 57) : obj_propagate: 27 Entries in [add(ex_ja, cname(
OBJECT TRACE time(17, 40, 57) : obj_propagate: 26 Entries in [add(vorf_wie, cna
OBJECT TRACE time(17, 40, 57) : obj_propagate: 25 Entries in [add(wie_waers, cn
OBJECT TRACE time(17, 40, 58) : obj_propagate: 24 Entries in [add(obj_3, nf(t,
OBJECT TRACE time(17, 40, 59) : obj_propagate: 23 Entries in [add(ex_ja, nf(t,
OBJECT TRACE time(17, 40, 59) : obj_propagate: 22 Entries in [add(vorf_wie, nf(
OBJECT TRACE time(17, 40, 59) : obj_propagate: 21 Entries in [add(wie_waers, nf
OBJECT TRACE time(17, 40, 59) : obj_propagate: 20 Entries in [add(ex_ja, cname(
OBJECT TRACE time(17, 40, 59) : obj_propagate: 19 Entries in [add(vorf_wie, cna
OBJECT TRACE time(17, 40, 59) : obj_propagate: 18 Entries in [add(wie_waers, cn
OBJECT TRACE time(17, 41, 00) : obj_propagate: 17 Entries in [add(obj_3, nf(t, c
OBJECT TRACE time(17, 41, 01) : obj_propagate: 16 Entries in [add(ex_ja, nf(t, c
OBJECT TRACE time(17, 41, 01) : obj_propagate: 15 Entries in [add(vorf_wie, nf(t
OBJECT TRACE time(17, 41, 01) : obj_propagate: 14 Entries in [add(wie_waers, nf(
OBJECT TRACE time(17, 41, 01) : obj_propagate: 13 Entries in [add(ex_ja, cname(k
OBJECT TRACE time(17, 41, 01) : obj_propagate: 12 Entries in [add(vorf_wie, cnam
OBJECT TRACE time(17, 41, 01) : obj_propagate: 11 Entries in [add(wie_waers, cna
OBJECT TRACE time(17, 41, 02) : obj_propagate: 10 Entries in [add(obj_3, nf(t, c
OBJECT TRACE time(17, 41, 03) : obj_propagate: 9 Entries in [add(ex_ja, nf(t, c,
```

```
OBJECT TRACE time(17, 41, 03) : obj_propagate: 8 Entries in [add(vorf_wie, nf(t,
OBJECT TRACE time(17, 41, 03) : obj_propagate: 7 Entries in [add(wie_waers, nf(t
OBJECT TRACE time(17, 41, 03) : obj_propagate: 6 Entries in [add(ex_ja, cname(ke
OBJECT TRACE time(17, 41, 03) : obj_propagate: 5 Entries in [add(vorf_wie, cname
OBJECT TRACE time(17, 41, 03) : obj_propagate: 4 Entries in [add(wie_waers, cnam
OBJECT TRACE time(17, 41, 04) : obj_propagate: 3 Entries in [add(obj_3, nf(t, c,
OBJECT TRACE time(17, 41, 05) : obj_propagate: 2 Entries in [add(ex_ja, nf(t, c,
OBJECT TRACE time(17, 41, 05) : obj_propagate: 1 Entries in [add(vorf_wie, nf(t,
OBJECT TRACE time(17, 41, 05) : obj_propagate: 0 Entries in [add(wie_waers, nf(t
OBJECT TRACE time(17, 41, 05) : obj_propagate: 0 Entries in []
tell succeded : obj_1::((anything and verb_modus:conj)and satz_typ:int and keyw

OBJECT TRACE time(17, 41, 07) : obj_propagate: 6 Entries in [add(ex_ja, cname(ke
OBJECT TRACE time(17, 41, 08) : obj_propagate: 5 Entries in [add(vorf_wie, cname
OBJECT TRACE time(17, 41, 08) : obj_propagate: 4 Entries in [add(wie_waers, cnam
OBJECT TRACE time(17, 41, 08) : obj_propagate: 3 Entries in [add(obj_3, nf(t, c,
OBJECT TRACE time(17, 41, 09) : obj_propagate: 2 Entries in [add(ex_ja, nf(t, c,
OBJECT TRACE time(17, 41, 09) : obj_propagate: 1 Entries in [add(vorf_wie, nf(t,
OBJECT TRACE time(17, 41, 09) : obj_propagate: 0 Entries in [add(wie_waers, nf(t
OBJECT TRACE time(17, 41, 09) : obj_propagate: 0 Entries in []

positiver Vorschlag
```

17.39.58 - 17.41.09 = 71 Sekunden Gesamtzeit

Anhang B: Der N-Queens Benchmark für BAP[30]

```
PREDICATE pqueens(integer).
pqueens(N) :-
      get_my_id(MyID),                % process id for messages
      integers(1,N,Ns),
      select(First,Ns,RestNs),
        exec( (                       % emit task
           permutation(RestNs,PermutatedRest),
             safe([First|PermutatedRest]),
             write([First|PermutatedRest]),nl,
           fail
           ;
           send_msg(MyID,done(First))
           )),
        asserta( pq_reply(First)),% keep Reply Port in mind!
      fail.
pqueens(_):-
      retract(pq_reply(First)),  % collect acknowledgments
        rec_msg(_,done(First)),  % wait for the reply
      fail.
pqueens(_).


PREDICATE integers(integer,integer,int_list).
integers(Low,High,[Low|Rest]) :-
      Low =< High,!,
      M is Low + 1,
      integers(M,High,Rest).
integers(_,_,[]).

PREDICATE safe(int_list).
safe([]).
safe([Q|Qs]) :-
      notattack(Q,1,Qs),
      safe(Qs).

PREDICATE notattack(integer,integer,int_list).
notattack(_,_,[]).
notattack(X,N,[Y|Ys]) :-
      A is Y - X,
      N \= A,
      B is X - Y,
      N \= B,
      C is N + 1,
      notattack(X,C,Ys).

PREDICATE permutation(int_list,int_list).
permutation(Xs,[Z|Zs]) :-
      select(Z,Xs,Ys),
      permutation(Ys,Zs).
permutation([],[]).

PREDICATE select(integer,int_list,int_list).
```

[30]Entnommen aus dem Brain Aid System Tutorial [BAP 93]

```
select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Zs]) :-
      select(X,Ys,Zs).
```

Anhang C: Die Implementierung des Farm-Schedulers[31]

```
%-------------------------------------------------------------
% obj_store(+ObjectName, +NF, +ExtSit, +SuperList)
%
% Add the object to the internal world-state and
% complete the references to the superconcept cache.
%-------------------------------------------------------------
PREDICATE obj_store(void,void,void,void).
obj_store(Object, NF, ExtSit, SuperList):-
       (
         \+ name_get(_,Object),                          % if name doesn't exist
         obj_store_new(Object, NF, IntSit, SuperList)    % store new object
       ;
         obj_store_local(Object, NF, IntSit, SuperList)  % store already existing object
       ),!.

%-------------------------------------------------------------
% obj_store_new(+ObjectName, +NF, +ExtSit, +SuperList)
%
% Create a new entry for Object.
%-------------------------------------------------------------
PREDICATE obj_store_new(void,void,void,void).
obj_store_new(Object, NF, IntSit, SuperList):-
       name_gen_id(Object,_),                          % generate new Id
       obj_store_local(Object, NF, IntSit, SuperList),!. % store on local DB

%-------------------------------------------------------------
% obj_store_local(+ObjectName, +NF, +ExtSit, +SuperList)
%-------------------------------------------------------------
PREDICATE obj_store_local(void,void,void,void).
obj_store_local(Object, NF, Sit, SuperList):-
       convert_sit_to_intsit(Sit,IntSit),              % ensure consistent NF position
       convert_obj_to_objkey(Object,Key),              % make Obj out of Key
       make_concept_instances(Key, IntSit, SuperList), % integrate NF into concept cache
       retractall(onf(Key,_,IntSit,_)),
       assertz(onf(Key,NF,IntSit,SuperList)),!.        % add new NF to local db

%-------------------------------------------------------------
% propagate_props(PropListIn,PropListOut,Status)
%
% Extract all "add" arguments to Obj from the PropListIn
% Status is 'ok', if call succeeded, 'error' otherwise.
%-------------------------------------------------------------
PREDICATE propagate_props(void,void,void).
propagate_props([],[],ok):-!.
propagate_props(PropListIn,PropListOut,ok):-
       nonvar(PropListIn),
       var(PropListOut),
       filter_concepts(PropListIn, Concs),
       conc_unify_list(Concs, PropSum),
       PropListIn = [add(Obj,_,CurSit)|_],             % find out dest object
       assert_object(Obj,PropSum,CurSit,PropListOut),!.
propagate_props(_,[],error).                           % ran into an error

%-------------------------------------------------------------
% obj_start_propagation(+PropagationList)
%
% Successively executes propagations and sorts new
% pending props. PropList = list_of(add(Obj,C,Sit))
%-------------------------------------------------------------
PREDICATE obj_start_propagation(void).
obj_start_propagation(PropList):-              % this predicate is executed by the SHELL
       obj_prop_assert(PropList),              % initially append propagations to agenda
       fail.
obj_start_propagation(_):-                     % successively sort propagations
       repeat,                                 % WHILE propagations or pending servers
         obj_prop_loop_collect(Obj,PropList),  % find all propagations for one Obj
         flex_debug(note,[propagate(Obj,PropList),": Propagating:\n"]),
         ( PropList = [] ->                    % IF no more propagations currently
           farm_synchronize_one(NewPropList,Status), % wait for result from a pending Server
```

[31] gekürzt

```
          obj_prop_assert(NewPropList)          % put new props int DB
        ;
          Status=ok,
          farm_distribute(Obj,PropList)                % distribute propagations to servers
        ),
        ( Status=error,                                % IF "clash"
          farm_synchronize_all(_,_),                   % THEN wait for all pending replies
          !,fail                                       % AND fail
        ; true
        ),
      PropList = [],                                   % UNTIL no more propagations
      findall(S,database(pend_ser(S)),PendingSList),   % get the list of pending servers
      PendingSList = [],                               % AND no more servers active
      findall(P,database(prop(P)),PendingPropList),    % get the list of pending servers
      PendingPropList = [],                            % AND no more new propagations
      !.

%-------------------------------------------------------------
%       obj_prop_loop_collect(Object,PropagationList)
%
% Collect all propagations for the object, for which the
% first propagation (on the agenda) is meant
%-------------------------------------------------------------
PREDICATE obj_prop_loop_collect(void,void).
obj_prop_loop_collect(Obj,PropList):-
       prop(add(Obj,_,Sit)),                           % get the first object in the list
       % now extract all messages for obj
       Prop = add(Obj,_,Sit),
       findall(Prop, retract(prop(Prop)), PropList),!.
obj_prop_loop_collect(_,[]).

%-------------------------------------------------------------
% obj_normalform(+Object,+ExtSit,-NormalForm)
% obj_normalform(+Object,+ExtSit,-NormalForm,-MSC)
%
% Returns the NormalForm of an Object in a specific
% situation. If a NormalForm doesn't exist in Sit try all
% previous ones.
%-------------------------------------------------------------
PREDICATE obj_normalform(void,void,void).
obj_normalform(ObjIn,Sit,NF):-
       obj_normalform(ObjIn,Sit,NF,_).

% To be executed by any process
PREDICATE obj_normalform(void,void,void,void).
obj_normalform(ObjIn,Sit,NF,MSC):-
       nonvar(ObjIn),
       nonvar(Sit),
       convert_obj_to_objkey(ObjIn,ObjKey),
       ( IntSit = Sit ; sit_extern(Sit,_,IntSit)),       % convert Ext to IntSit
       (PrevSit = IntSit; sit_previous(IntSit,PrevSit)), % select all previous sits
       database(onf(ObjKey,NF,PrevSit,MSC)),!.

PREDICATE obj_alloc_server(void).
obj_alloc_server(Server):-                       % if one server is available, return it:
       retract(flex_server(Server)),!.           % just return the topmost server
obj_alloc_server(Server):-                       % no server currently available
       :<- flex_add_server(Server),!.            % wait until free message arrives

PREDICATE obj_free_server(void).
obj_free_server(Id):-
       get_my_id(MyId),
       ( database(flex_shell(Master)) ->
         Master :-> flex_add_server(MyId)
       ; error([obj_free_server(Id)," on Process ",MyId,": entry does not exist:\n"])
       ),!.

%-------------------------------------------------------------
% farm_distribute(_,PList)
%
% distribute the PropList to a free server
%-------------------------------------------------------------
PREDICATE farm_distribute(void,voidlist).
farm_distribute(_,PList):- write(farm_distribute(PList),"\n"),fail.
farm_distribute(_,[]):-!.
farm_distribute(_,PList):-
       obj_alloc_server(Server),                 % get a server or wait until free
       farm_pack_db(DB),                         % collect entire DB into one variable
       get_my_id(MyId),
```

```
        Server :-> (                                % tell server to WORK
        obj_unpack_db(DB),                          % restore identical (!) Db
              propagate_props(PList,NewPList,Status),      % do the WORK ..
              obj_pack_db([                         % pack relevant database entries
                     obj_name(_,_),
                     obj_id(_),
                     onf(_,_,_,_),
                     ext_sit(_,_,_,_),
                     inst(_,_),
                     int_sit(_,_)
              ],ServerDb),
                obj_free_server(Server),            % add server to list of free FLEX servers
              MyId :-> propagate_result(NewPList,Status,ServerDb)
        ),
        assertz(pend_ser(Server)),!.                % add entry to pending server list

%-------------------------------------------------------------
% farm_synchronize_one(NewPList,Status)
%
% grap any pending server and wait until terminates WORK
%-------------------------------------------------------------
PREDICATE farm_synchronize_one(voidlist,void).
farm_synchronize_one(NewPList,Status):- write(farm_synchronize_one(NewPList,Status),"\n"),fail.
farm_synchronize_one(NewPList,Status):-
        database(pend_ser(_)),                      % there exists at least one pending server
        Server <-: propagate_result(NewPList,Status,ServerDb), % wait for reply from server
        retract(pend_ser(Server)),                  % remove server from list
        farm_merge_db(ServerDb),!.
farm_synchronize_one([],ok):-
        \+ database(pend_ser(_)),                   % there was not pending server
        !.
farm_synchronize_one(NewPList,Status):-
        flex_debug(fatal,[farm_synchronize_one(NewPList,Status),': FAILED unexpectedly.']), fail.

%-------------------------------------------------------------
% farm_synchronize_all(NewPList,Status)
%
% wait until all servers have terminated
%-------------------------------------------------------------
PREDICATE farm_synchronize_all(voidlist,void).
farm_synchronize_all(NewPList,Status):- write(farm_synchronize_all(NewPList,Status),"\n"),fail.
farm_synchronize_all(NewPList,Status):-
        retract(pend_ser(Server)),                  % FOR all pending servers
          Server <-: propagate_result(NewPList,Status,ServerDb),  % wait for reply
          assertz(new_prop_list(NewPList)),         % temporarily store new Prop list
          assertz(new_status(Status)),              % temporarily store status
          farm_merge_db(ServerDb),                  % integrate new DB into master DB
        fail.
farm_synchronize_all(NewPList,Status):-
        findall(X, (retract(new_prop_list(PList)), member(X,PList)), NewPList),
        findall(Y, retract(new_status(Y)), StatusList),
        farm_eval_status_list(StatusList,Status),!.

%-------------------------------------------------------------
% check if "error" occures in StatusList.
%-------------------------------------------------------------
PREDICATE farm_eval_status_list(voidlist,void).
farm_eval_status_list([],ok):-!.
farm_eval_status_list([error|_],error):-!.
farm_eval_status_list([_|Rest],Status):-
        farm_eval_status_list(Rest,Status).

%-------------------------------------------------------------
% farm_merge_db(DbList)
%
% insert the NEW information of a server DB into the master DB
%-------------------------------------------------------------
PREDICATE farm_merge_db(voidlistlist).
farm_merge_db(DbList):-
        member(Db,DbList),
          member(DbItem,Db),
            farm_merge_db_add_item(DbItem),
        fail.
farm_merge_db(_).

% add a single item into the master DB
PREDICATE farm_merge_db_add_item(void).
farm_merge_db_add_item(Item):-
        database(Item),!.                           % Item alread exists => skip
```

```
farm_merge_db_add_item(onf(Id,NF,Sit,MSC)):-              % Item does not exist
        (
          database(obj_name(Id,ObjName))
        ;
          flex_debug(error,[farm_merge_db(Id),': FAILED unexpectedly.']),fail
        ),!,
        obj_store(ObjName,NF,Sit,MSC),!.                  % integrate into sys. status
farm_merge_db_add_item(obj_name(Id,Name)):-               % Item does not exist
        assertz(obj_name(Id,Name)),!.
farm_merge_db_add_item(obj_id(Id)):-                      % Item does not exist
        database(obj_id(OldId)),
        max(Id,OldId,MaxId),                              % one worker process per transputer
        retractall(obj_id(_)),                            % kill ALL entries
        assertz(obj_id(MaxId)),!.
farm_merge_db_add_item(Item):-
        assertz(Item),!,
        flex_debug(warning,[farm_merge_db(Item),': fund unexpected item.']),fail.
```

Anhang D: Die Implementierung des DO-Schedulers[32]

```
%-------------------------------------------------------------
% obj_store(+ObjectName, +NF, +ExtSit, +SuperList)
%
% Add Object to the internal world-state and complete
% references to Supers
% Returns TRUE on success, creates an error with bad arguments
%-------------------------------------------------------------
PREDICATE obj_store(void,void,void,void).
obj_store(Object, NF, ExtSit, SuperList):-
        (
          \+ name_get(_,Object),                          % if name doesn't exist ..
          obj_store_new(Object, NF, IntSit, SuperList)    % store new object
        ;
          obj_store_old(Object, NF, IntSit, SuperList)    % store already existing object
        ),!.

%-------------------------------------------------------------
% obj_store_new(+ObjectName, +NF, +ExtSit, +SuperList)
%
% DistributedObject treat the generation of a new objects using several different
% schemes. The scheme used here right now is to use a random generator
%-------------------------------------------------------------
PREDICATE obj_store_new(Object, NF, IntSit, SuperList):-
obj_store_new(Object, NF, IntSit, SuperList):-
        % we are on FlexShell: create new object
        obj_dist_create_obj(Object, NF, IntSit, SuperList),!.     % create object and tell all
servers about the object

%-------------------------------------------------------------
% obj_store_old(+ObjectName, +NF, +ExtSit, +SuperList)
%
% Identical for all modes of FLEX. Farm and Sequential act on local db anyway,
% DistObj treats only local objects, so there shouldn't be a problem.
%-------------------------------------------------------------
PREDICATE obj_store_old(void,void,void,void).
obj_store_old(Object, NF, IntSit, SuperList):-
        database(obj(Object,_,Server)),                   % find out serving process
        (
          get_my_id(Server),                              % on local server
          obj_store_local(Object, NF, IntSit, SuperList)
        ;
          Server :-> obj_store_local(Object, NF, IntSit, SuperList) % on remote server
        ),!.

%-------------------------------------------------------------
% obj_store_local(+ObjectName, +NF, +ExtSit, +SuperList)
%-------------------------------------------------------------
PREDICATE obj_store_local(void,void,void,void).
obj_store_local(Object, NF, Sit, SuperList):-
        convert_sit_to_intsit(Sit,IntSit),
        convert_obj_to_objkey(Object,Key),
        make_concept_instances(Key, IntSit, SuperList),
        retractall(onf(Key,_,IntSit,_)),
        assertz(onf(Key,NF,IntSit,SuperList)),!.

%-------------------------------------------------------------
% obj_dist_create_obj(ObjName, NF, IntSit, SuperList)
%
% create an object from a 'propagation prototype'.
%-------------------------------------------------------------
PREDICATE obj_dist_create_obj(void).
obj_dist_create_obj(add(ObjName,_,Sit)) :-
        obj_dist_create_obj(ObjName,nf_ctop,Sit,[]),!.

% this routine is to be executed on the local server.
PREDICATE obj_dist_create_obj(void,void,void,void).
obj_dist_create_obj(ObjName, NF, IntSit, SuperList):-
        database(flex_shell(Shell)),                      % get Id of FlexShell
        \+ get_my_id(Shell),!,                            % check if <SELF> FlexShell
```

[32] gekürzt

```
        % we know that this is NOT the FlexShell: To maintain a
        % central management of object creation send this message
        % to the FlexShell:
        Shell :-> obj_store(ObjName, NF, IntSit, SuperList),!.    % send task to Shell.
obj_dist_create_obj(ObjName, NF, IntSit, SuperList):-
        database(obj(ObjName,_,_)),!,                       % someone else generated object
        error(["Double defined object ",ObjName,"\n"]),    % Severe: message
        flex_debug(fatal,[obj_dist_create_obj(ObjName, NF, IntSit, SuperList),
        ': object double defined.']), fail.

obj_dist_create_obj(ObjName, NF, IntSit, SuperList):-
        % only executes on FlexShell
        name_gen_id(ObjName,ObjKey),                        % generate new Id

% ********** Strategy 1: get the list of all server an select one randomly ********
%       findall(FS,database(flex_server(FS)),ServerList),  % get FLEX-Server list
%       random_member(Server,ServerList),                   % select a random FLEX server

% ********** Strategy 2: (centralized create): take next server and put at the end ******
        retract(flex_server(Server)),                       % catch topmost server
        assertz(flex_server(Server)),                       % put at bottom
        assertz(obj(ObjName,object,Server)),                % assert a temp object ID
        assertz(obj_name(ObjKey,ObjName)),                  % .. and its name

%       database(flex_shell(Shell)),
        Cast = (                                            % tell all servers about existence
              retractall(obj(ObjName,_,_)),                 % avoid double entries
              assertz(obj(ObjName,object,Server)),          % inform all servers about the new object
              retractall(obj_name(ObjKey,_)),               % avoid double entries
                assertz(obj_name(ObjKey,ObjName))
               ),                                           % .. and its name

        obj_dist_broadcast(Cast),                           % assert object to all servers

        obj_pack_db([obj_id(_), int_sit(_,_), ext_sit(_,_,_,_)],PackedDb),
        Server :-> (
              obj_unpack_db(PackedDb),
              obj_store_local(ObjName, NF, IntSit, SuperList)     % tell server to store information
        ),!.

%-------------------------------------------------------------
% obj_start_propagation(+PropagationList)
%
% Successively executes propagations and sorts new
% pending props. PropList = list_of(add(Obj,C,Sit))
%-------------------------------------------------------------
PREDICATE obj_start_propagation(void).
obj_start_propagation(PropList):-                           % successively sort propagations
        member(Prop,PropList),                              % extract single propagations
          Prop = add(Obj,_,_),                              % extract destination
          (
            \+ database(obj(Obj,_,ServerId)),               % entry doesn't exist
            flex_debug(note,[propagate(Obj,Prop),': did not find Object entry - creating']),
            obj_dist_create_obj(Prop),                      % create a suitable object
            fail
          ; true
          ),
          database(obj(Obj,_,ServerId)),                    % get related sever id
          flex_debug(note,[propagate(Obj,Prop),": Propagating:\n"]),
          ServerId :-> obj_dist_propagate(Prop),            % distribute propagation
        fail.
obj_start_propagation(_):-!.


PREDICATE obj_dist_propagate(void).
obj_dist_propagate(Prop):-
        PropListIn = [Prop],
        filter_concepts(PropListIn, Concs),
        conc_unify_list(Concs, PropSum),
        PropListIn = [add(Obj,_,CurSit)|_],                 % find out dest object
        assert_object(Obj,PropSum,CurSit,PropListOut),!,% do the WORK
        obj_start_propagation(PropListOut),!.               % distribute propagations to other objects

%-------------------------------------------------------------
%       Propagation Auxillaries
%-------------------------------------------------------------

% assert all Props from PropList to agenda
PREDICATE obj_prop_assert(void).
```

```
obj_prop_assert(PropList):-
       member(NewProp,PropList),
         assertz(prop(NewProp)),
       fail.
obj_prop_assert(_).

% collect all propagations for the object, for which the first propagation (on the agenda) is meant
PREDICATE obj_prop_loop_collect(void,void).
obj_prop_loop_collect(Obj,PropList):-
       prop(add(Obj,_,Sit)),                          % get the first object in the list
       % now extract all messages for obj
       Prop = add(Obj,_,Sit),
       findall(Prop, retract(prop(Prop)), PropList),!.
obj_prop_loop_collect(_,[]).

% extract all "add" arguments to Obj from the PropListIn
PREDICATE propagate_props(void,void,void).
propagate_props([],[],ok):-!.
propagate_props(PropListIn,PropListOut,ok):-
       nonvar(PropListIn),
       var(PropListOut),
       filter_concepts(PropListIn, Concs),
       conc_unify_list(Concs, PropSum),
       PropListIn = [add(Obj,_,CurSit)|_], % find out dest object
       assert_object(Obj,PropSum,CurSit,PropListOut),!.
propagate_props(_,[],error).

%-------------------------------------------------------------
% obj_instanciates(+Object, +Concept, +Sit)
%
% Check if Object is an instance of Concept in Sit
% Returns TRUE on success, FAILS otherwise
% NO runtime errors
% NOTE: This predicate is optimized to speed, if the
% object is already known.
%-------------------------------------------------------------
PREDICATE obj_instanciates(void,void,void).
obj_instanciates(ObjIn, ConceptIn, Sit):-
       is_concept(ConceptIn),                         % already concept name or key
       convert_obj_to_objkey(ObjIn,Object),           % must be a valid object
       convert_sit_to_intsit(Sit,IntSit),
       obj_normalform(Object,IntSit,_,DirectSupers),  % get list of direct supers
       member(DirectSuper,DirectSupers),              % for all supers:
       get_super_concepts(DirectSuper,AllSupers),     % get transitive closure of supers
       member(TransSuper,AllSupers),                  % get one trans super for one of the
direct supers
       get_instance(TransSuper,IntSit,Object),!.      % check object instance of trans super

obj_instanciates(ObjIn, ConceptIn, Sit):-             % ConceptIn is a NormalForm!
       convert_obj_to_objkey(ObjIn,Object),
       obj_instance(ConceptIn,Sit,Object),!.

%-------------------------------------------------------------
% obj_instances(+Concept, +Sit, -ObjectList)
%
% Looks for all objects that are member of Concept
% or the supers of Concept in Sit or situations
% before Sit.
% Returns ObjectList: the list of found objects.
% Runtime Error on unknow Concept or bad arguments
%-------------------------------------------------------------
PREDICATE obj_instances(void,void,void).
obj_instances(Concept,Sit,NameList):-
       nonvar(Sit),
       is_sit(Sit),
       setof(Object,obj_instance(Concept,Sit,Object),ObjectList),
       convert_objkeylist_to_objnamelist(ObjectList,NameList),!.

%-------------------------------------------------------------
% NONDETERM obj_instance(+Concept, +Sit, -Object)
% NONDETERM obj_instance(+ConceptList, +Sit, -Object)
%
% Looks for all objects that are member of Concept
% or the direct supers of Concept in Sit or situations
% before Sit.
%-------------------------------------------------------------
PREDICATE obj_instance(void,void,void).
obj_instance(Concept, Sit, Inst):- % get instances of direct supers
       conc_direct_supers_equal(Concept,SuperList,new),
```

```
        % (new=new key, SuperList = list of direct supers, old=existing key)
        % get the list of direct supers (because objects may be put under
        % the supers in a case where there is no better concept)
        convert_sit_to_intsit(Sit,IntSit),
        member(SuperConc,SuperList),
        get_instance(SuperConc,IntSit,Inst),              % NONDETERM: get all instances
        conc_subsumes(Inst,Concept).                      % check that really instance

obj_instance(Concept, Sit, Object):- % get instances of direct supers
        conc_direct_subs_equal(Concept,SubList,_),        % get the list of direct subs
        convert_sit_to_intsit(Sit,CheckSit),
        member(SubConc,SubList),
        get_sub_concepts(SubConc,ConceptList),            % get transitive closure towards nothing
        member(CheckConcept,ConceptList),
        get_instance(CheckConcept,CheckSit,Object).

% NONDETERM: get all instances for a specific concept in a given internal situation of earlier
PREDICATE get_instance(void,void,void).
get_instance(Concept,CheckSit,Object):-
        database(inst(Concept,InstList)),
        member(inst(Object,SitList),InstList),
        member(IntSit,SitList),
        ( IntSit = CheckSit ; sit_previous(CheckSit,IntSit)).

%-------------------------------------------------------------
% obj_msg(+Object,+ExtSit,-MSC)
% obj_msc(+Object,+ExtSit,+Filter,-MSC)
%-------------------------------------------------------------
PREDICATE obj_msc(void,void,void).
obj_msc(Obj,ExtSit,MSCNames):-
        obj_normalform(Obj,ExtSit,_,MSC),
        convert_conckeylist_to_concnamelist(MSC,MSCNames),!.

PREDICATE obj_msc(void,void,void,void).
obj_msc(Obj,ExtSit,Filter,FilterMSCNames):-
        obj_normalform(Obj,ExtSit,_,MSC),
        hc_direct_supers_list(c,MSC,Filter,FilterMSC),
        convert_conckeylist_to_concnamelist(FilterMSC,FilterMSCNames),!.

%-------------------------------------------------------------
% obj_normalform(+Object,+ExtSit,-NormalForm)
% obj_normalform(+Object,+ExtSit,-NormalForm,-MSC)
%
% Returns the NormalForm of an Object in a specific
% situation. If a NormalForm doesn't exist in Sit try all
% previous ones.
%-------------------------------------------------------------
PREDICATE obj_normalform(void,void,void).
obj_normalform(ObjIn,Sit,NF):-
        obj_normalform(ObjIn,Sit,NF,_).

% To be executed by any process
PREDICATE obj_normalform(void,void,void,void).
obj_normalform(ObjIn,Sit,NF,MSC):-
        nonvar(ObjIn),
        nonvar(Sit),
        convert_obj_to_objname(ObjIn,ObjName),
        database(obj(ObjName,_,ObjServer)),                       % find out serving process
        name_get(ObjKey,ObjName),
        obj_dist_get_onf(ObjServer,ObjKey,NF,Sit,MSC),
        nonvar(NF),!.                                             % ObjServer DID return value
obj_normalform(Obj,Sit,NF,MSC):-

PREDICATE obj_dist_get_onf(void,void,void,void,void).
obj_dist_get_onf(ObjServer,ObjKey,NF,Sit,MSC) :-
        get_my_id(ObjServer),!,                                   % message to <self> => 'call' is ok
        ( IntSit = Sit ; sit_extern(Sit,_,IntSit)),               % convert Ext to IntSit
        (PrevSit = IntSit; sit_previous(IntSit,PrevSit)),         % select all previous sits
        database(onf(ObjKey,NF,PrevSit,MSC)),!.                  % returns only ONE solution
obj_dist_get_onf(ObjServer,ObjKey,NF,Sit,MSC) :-
        obj_pack_db([int_sit(_,_),ext_sit(_,_,_,_)],PackedDb),    % pack situations
        ObjServer <-:-> (
              obj_unpack_db(PackedDb),                            % make situations are up to date
              ( IntSit = Sit ; sit_extern(Sit,_,IntSit)),         % convert Ext to IntSit
              (PrevSit = IntSit; sit_previous(IntSit,PrevSit)),   % select all previous sits
              database(onf(ObjKey,NF,PrevSit,MSC))                % returns only ONE solution
        ),!.
```

```
PREDICATE obj_alloc_server(void).
obj_alloc_server(Server):-                              % if available, return it:
       retract(flex_server(Server)),!.                  % just return the topmost server
obj_alloc_server(Server):-                              % no server currently available
       :<- flex_add_server(Server),!.                   % wait until free message arrives

PREDICATE obj_free_server(void).
obj_free_server(Id):-
       get_my_id(MyId),
       ( database(flex_shell(Master)) ->
         Master :-> flex_add_server(MyId)
       ; error([obj_free_server(Id)," on Process ",MyId,": flex_shell(Tid,Pid) entry does not
exist:\n"])
       ),!.


%-------------------------------------------------------------
%       UTILITIES
%-------------------------------------------------------------

% restore database
PREDICATE obj_unpack_db(voidlistlist).
obj_unpack_db(DbList):-
       member(Db,DbList),                               % get list of clauses all for one db-predicate
         Db = [First|_],                                % extract first (any) entry of the list
         First =.. [FirstFunc|FirstArgList],            % decompose for functor and arguments
         build_empty_arg_list(FirstArgList,EmptyFirstArgList),
         FirstProto =.. [FirstFunc|EmptyFirstArgList],  % build a prototype for this db predicate
         retractall(FirstProto),                        % remove all entries already there
         member(DbItem,Db),                             % insert new entries instead
           assertz(DbItem),
       fail.
obj_unpack_db(_).

% replace a given list by a list of same size but with unbound arguments
PREDICATE build_empty_arg_list(voidlist,voidlist).
build_empty_arg_list([],[]):-!.
build_empty_arg_list([_|T],[_|Rest]):-
       build_empty_arg_list(T,Rest).


% Distributed objects only require the TBOX to be distributed this way!
PREDICATE dist_pack_db(voidlistlist).
dist_pack_db(_):-
       member(DbList,
            [                               % Tbox  Abox    Ibox    Ebox    Other
            par_enable(_),                  %                               Execution Mode
            compute_pred(_,_,_,_),          % T
            bottom(_,_),                    % T
            conc_count(_),                  % T
            current_int_sit(_),             %       A
            dNo(_),                         %               I
            definition(_,_,_),              % T
            err_count(_),                   %                       E
            err_disable(_,_),               %                       E
            err_flag(_),                    %                       E
            err_level(_,_),                 %                               E
            err_disable(_,_),               %                       E
            ext_sit(_,_,_,_),               %       A
            flex_server(_),                 %                                       ServerMan
            flex_shell(_),                  %                               ServerMan
            hc_HC(_),                       % T
            hierarchy(_,_,_,_,_,_,_),       % T
            iconc(_,_,_,_),                 %               I
% NO!       inst(_,_),                      %       A
            int_sit(_,_),                   %       A
            key(_),                         % T
            key_count(_),                   % T
            key_filter(_,_),                % T
            key_nfo_nft(_,_,_),             % T
            name_key(_,_),                  % T
            obj_id(_),                      %       A
            op(_,_,_),                      %                               Parser
            sit_count(_),                   %       A
            top(_,_),                       % T
            transact(_,_,_),                %       A
            uc_count(_)                     % T
            ]),
         findall(DbList,database(DbList),DbListList),
```

```
          assertz(db_list(DbListList)),
        fail.
dist_pack_db(DbList):-
        findall(X,retract(db_list(X)),DbList),!.


% get selected databases for server. This way we can selectively choose elements
PREDICATE obj_pack_db(voidlist,voidlistlist).
obj_pack_db(PredList,_):-
        member(DbList,PredList),                    % select Pred from PredList
          findall(DbList,database(DbList),DbListList),
          assertz(db_list(DbListList)),
        fail.
obj_pack_db(_,DbList):-
        findall(X,retract(db_list(X)),DbList),!.

%-------------------------------------------------------------
%       obj_dist_broadcast(Goal)
%
%       executes Goal on all FLEX SERVERS in the network, but
%       NOT on the FLEX SHELL.
%-------------------------------------------------------------
PREDICATE obj_dist_broadcast(void).
obj_dist_broadcast(Goal):-                          % this clause must be before the <self> clause
below!
        get_my_id(MyId),
        database(flex_server(Server)),
          MyId \= Server,                           % treate especially
          Server :-> Goal,                          % non-suspending message passing
        fail.
obj_dist_broadcast(Goal):-                          % put this clause as second behind the ":->" one,
to allow for parallel execution in case that all processes have to do the same task
        get_my_id(MyId),
        database(flex_server(MyId)),                % if Process itself is a flex_server ..
        call(Goal),                                 % execute GOAL on own process
        fail.
obj_dist_broadcast(_).


PREDICATE obj_dist_update_tbox.
obj_dist_update_tbox:-
        dist_pack_db(PackedDb),
        obj_dist_broadcast(
              obj_unpack_db(PackedDb)
        ),!.

PREDICATE obj_dist_update_abox.
obj_dist_update_abox:-
        obj_pack_db([obj_id(_), int_sit(_,_), ext_sit(_,_,_,_)],PackedDb),
        obj_dist_broadcast(
              obj_unpack_db(PackedDb)
        ),!.


PREDICATE get_obj_list(void).
get_obj_list(ObjList):-
        % supposes to be executed on a process with a
        % obj(_,_,_) database
        findall(Obj,(Obj = obj(_,_,_),database(Obj)),ObjList),!.

PREDICATE broadcast(void).
broadcast(Goal):-
        get_obj_list(ObjList),
          member(obj(_,_,OID),ObjList),
          OID :-> Goal,
        fail.
broadcast(_).
```

Anhang E: Liste der FLEX-Module

`active.bap`	3305	Schalter für Normalisierungsregeln
`auxpred.bap`	1431	Hilfsprädikate
`basics.bap`	1333	Hilfsprädikate
`cache.bap`	4389	Hierarchy Cache für Konzepte
`classify.bap`	13416	Der Klassifier
`compute.bap`	7332	Test/Computer Erweiterung für FLEX
`conc.bap`	5122	Konzept Unifikation und Subsumption
`convert.bap`	9730	Konverter Routinen
`data.bap`	6320	Speicher für Definitionen und Namen
`dbox.bap`	16004	Default-Modul
`debug.bap`	13772	Fehler-Modul
`definiti.bap`	8197	Term Einführungen
`defspace.bap`	2447	Defaultspaces-Modul
`entity.bap`	2918	uniforme Behandlung von Entitäten
`error.bap`	7955	Error-Handler
`feature.bap`	1105	Behandlung von Features
`filter.bap`	1257	Filter-Modul
`flags.bap`	718	verschiedene Flags
`flex.bap`	9119	FLEX-Hauptmodul
`flexcom.bap`	12751	FLEX-Commands
`grammar.bap`	728	Parser-Hilfsmodul
`hc.bap`	13979	Hierarchy-Cache
`hcdata.bap`	2323	Hierarchy-Cache Hilfsmodul
`ibox.bap`	2872	I-Link Modul
`k_op.bap`	1367	K-Operator Modul
`loadman.bap`	5327	Load-Management für Parallel FLEX
`macro.bap`	3765	Behandlung von Makro-Regeln
`name.bap`	4184	Namens-Module
`nameserv.bap`	1186	Hilfsmodul für Namen
`nf.bap`	15358	Behandlung von Normalformen
`nf1.bap`	3258	Hilfsmodule für die Behandlung von Normalformen
`nrules.bap`	7702	Normalisierungsregeln
`nrules1.bap`	6885	Normalisierungsregeln
`objaux.bap`	5588	Hilfsmodul für Objektmanagement
`object.bap`	29729	Objektmanagement
`objinst.bap`	2967	Hilfsmodul für Objektmanagement
`objlist.bap`	3910	Hilfsmodul für Objektmanagement
`objutil.bap`	8421	Hilfsmodul für Objektmanagement
`obj_ass.bap`	7211	Behandlung von Objekt-Tells
`obj_aux.bap`	825	Hilfsmodul für Objektmanagement
`oman.bap`	16456	Management von parallelen Objekten
`omsg.bap`	3222	Management von parallelen Objekten
`op.bap`	259	Operatoren
`parser.bap`	6728	FLEX Parser

`parser1.bap`	2410	Hilfmodul für FLEX Parser
`quintus.bap`	4954	Kompatibilitätsmodul Quintus Prolog -> BAP
`reftest.bap`	2859	Referenz-Tests für FLEX
`role.bap`	5890	Behandlung von Rollen
`rvm_coll.bap`	627	Role-Value-Map Behandlung
`sit.bap`	18403	Situationsmanagement
`subsumpt.bap`	12407	Subsumption
`transact.bap`	9228	Transaktionsmanagement für die ABox
`trans_r.bap`	5469	Parser Hilfsmodul
`types.bap`	9759	Parser Typen